



METATRUST

Security Assessment for **DLC-link-solidity**

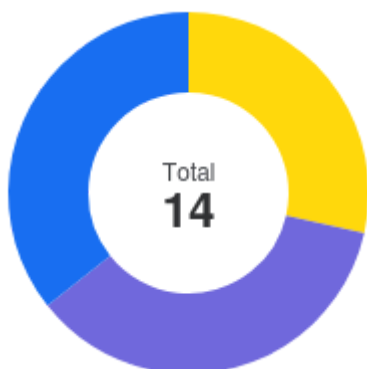
May 07, 2024






Executive Summary

Overview			
Project Name	DLC-link-solidity		
Codebase URL	https://github.com/DLC-link/dlc-solidity		
Scan Engine	Security Analyzer		
Scan Time	2024/05/07 08:00:00		
Commit Id	b042c767bbfa3de024222ed762a37cdbfa2ecd4992420f4b42554fce506179087b91fdf72ae44ccb		

Total			
Critical Issues	0		
High risk Issues	0		
Medium risk Issues	4		
Low risk Issues	5		
Informational Issues	5		

Critical Issues		The issue can cause large economic losses, large-scale data disorder, loss of control of authority management, failure of key functions, or indirectly affect the correct operation of other smart contracts interacting with it.
High Risk Issues		The issue puts a large number of users' sensitive information at risk or is reasonably likely to lead to catastrophic impacts on clients' reputations or serious financial implications for clients and users.
Medium Risk Issues		The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to moderate financial impact.
Low Risk Issues		The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low-impact in view of the client's business circumstances.
Informational Issue		The issue does not pose an immediate risk but is relevant to security best practices or Defence in Depth.



	Critical Issues	0%	0
	High risk Issues	0%	0
	Medium risk Issues	29%	4
	Low risk Issues	36%	5
	Informational Issues	36%	5

Summary of Findings



MetaScan security assessment was performed on **May 07, 2024 08:00:00** on project **DLC-link-solidity** with the repository on branch **default branch**. The assessment was carried out by scanning the project's codebase using the scan engine **Security Analyzer**. There are in total **14** vulnerabilities / security risks discovered during the scanning session, among which **4** medium risk vulnerabilities, **5** low risk vulnerabilities, **5** informational issues.

ID	Description	Severity	Alleviation
MSA-001	When the whitelisted contracts are the approved signers	Medium risk	Fixed
MSA-002	Probably mis-calculate the <code>_signerCount</code>	Medium risk	Fixed
MSA-003	The mint fee does not be sent to the fee recipient	Medium risk	Fixed
MSA-004	Centralized Roles	Medium risk	Acknowledged
MSA-005	Potential being out of gas as the increasement of <code>dlcs</code>	Low risk	Fixed
MSA-006	Upgradeable contracts missing a storage gap variable <code>__gap[50]</code>	Low risk	Fixed
MSA-007	The init threshold value from design document mismatches its implementation	Low risk	Acknowledged
MSA-008	The improper values of the <code>_threshold</code> and the <code>_minimumThreshold</code> may block the <code>_attestorMultisigIsValid</code> function	Low risk	Fixed
MSA-009	Upgradeable contracts missing a storage gap variable <code>__gap[50]</code>	Low risk	Fixed
MSA-010	Lack of validating the <code>_threshold</code>	Informational	Fixed
MSA-011	Gas optimazation	Informational	Fixed
MSA-012	Combining the chain id and contract address when calculating the UUID	Informational	Acknowledged
MSA-013	Lack of the fee rate boundary	Informational	Fixed
MSA-014	Missing emit event	Informational	Fixed

Findings

Medium risk (4)

1. When the whitelisted contracts are the approved signers

 Medium risk Security Analyzer

The whitelisted contracts are able to create a DLC with the `createDLC` function, and mark a DLC's status as `READY`, with the `createDLC` function.

The approved signers are able to update a DLC's status from `READY` to `FUNDED`, with the `setStatusFunded` function.

The DLC's creator, i.e. the whitelisted contract, is able to update a DLC's status from `FUNDED` to `CLOSING`, with the `closeDLC` function.

The approved signers are able to update a DLC's status from `CLOSING` to `CLOSED`, with the `postCloseDLC` function.

In a nutshell, the whitelisted contracts are able to update a DLC's status from none, and `FUNDED` to `READY`, and `CLOSING`. The approved signers are able to update a DLC's status from `READY`, and `CLOSING` to `FUNDED`, and `CLOSED`

If the whitelisted contracts are the approved signers, a DLC can go through states `READY`, `FUNDED`, `CLOSING`, and `CLOSED` without any restriction, which is a centralization risk.

File(s) Affected

contracts/DLCManager.sol #198-208

```
198     function createDLC(  
199         uint256 valueLocked,  
200         string calldata btcFeeRecipient,  
201         uint256 btcMintFeeBasisPoints,  
202         uint256 btcRedeemFeeBasisPoints  
203     )  
204     external  
205     override  
206     onlyWhiteListedContracts  
207     whenNotPaused  
208     returns (bytes32)
```

contracts/DLCManager.sol #247-264

```
247     function setStatusFunded(
248         bytes32 uuid,
249         string calldata btcTxId,
250         bytes[] calldata signatures
251     ) external whenNotPaused onlyApprovedSigners {
252         _attestorMultisigIsValid(abi.encode(uuid, btcTxId), signatures);
253         DLCLink.DLC storage dlc = dlcs[dlcIDsByUUID[uuid]];
254
255         if (dlc.uuid == bytes32(0)) revert DLCNotFound();
256         if (dlc.status != DLCLink.DLCStatus.READY) revert DLCNotReady();
257
258         dlc.fundingTxId = btcTxId;
259         dlc.status = DLCLink.DLCStatus.FUNDED;
260
261         DLCLinkCompatible(dlc.protocolContract).setStatusFunded(uuid, btcTxId);
262
263         emit SetStatusFunded(uuid, btcTxId, msg.sender);
264     }
```

contracts/DLCManager.sol #247-251

```
247     function setStatusFunded(
248         bytes32 uuid,
249         string calldata btcTxId,
250         bytes[] calldata signatures
251     ) external whenNotPaused onlyApprovedSigners {
```

contracts/DLCManager.sol #270-272

```
270     function closeDLC(
271         bytes32 uuid
272     ) external onlyCreatorContract(uuid) whenNotPaused {
```

contracts/DLCManager.sol #290-294

```
290     function postCloseDLC(
291         bytes32 uuid,
292         string calldata btcTxId,
293         bytes[] calldata signatures
294     ) external whenNotPaused onlyApprovedSigners {
```



Recommendation

Consider adding checks to ensure that approved signers and whitelisted contracts are not at the same address.

Alleviation Fixed

The team fixed this issue by adding an extra role based check in commit 08ad24555bea64d5a678f1a68c88165af4922217

2. Probably mis-calculate the `_signerCount`

 Medium risk Security Analyzer

The `addApprovedSigner` function adds an account as a signer and increases the counter `_signerCount`. Meanwhile, the `removeApprovedSigner` validates the `_signerCount` and ensures it not to be less than the `_minimumThreshold`.

Taking this scenario into account,

- The admin adds Alice as a signer, and assumes the `_minimumThreshold` is 2;

- The admin repeatedly adds Bob with the `addApprovedSigner` function 5 times, due to the function lack of checking if the `signer` is approved or not, which results in the `_signerCount` to be 6;
- The admin removes Alice and Bob from the approved signers, which results in the number of valid signers being 0. Because the `_signerCount` is 4, which is greater than the `_minimumThreshold` that is 2.

As a result, the validation of the `removeApprovedSigner` fails to keep a minimum threshold of signers.

File(s) Affected

contracts/DLCManager.sol #366-377

```
366     function addApprovedSigner(address signer) external onlyAdmin {
367         _approvedSigners[signer] = true;
368         _signerCount++;
369     }
370
371     function removeApprovedSigner(address signer) external onlyAdmin {
372         if (_signerCount == _minimumThreshold)
373             revert ThresholdMinimumReached(_minimumThreshold);
374         if (!_approvedSigners[signer]) revert SignerNotApproved(signer);
375         _approvedSigners[signer] = false;
376         _signerCount--;
377     }
```

Recommendation


Consider checking if a signer is approved before from the `addApprovedSigner` function.

Alleviation Fixed

The team resolved this issue by replacing the `addApprovedSigner` function with the function `grantRole` that check the role existence, in commits 4e6e428e714ab4d44d29d1c32d6eb9446352b1a9 and 08ad24555bea64d5a678f1a68c88165af4922217

3. The mint fee does not be sent to the fee recipient

 Medium risk

 Security Analyzer

The `mintFeeRate` is initialized as 0, thus, there is no mint fee so far.

The admin can update the `mintFeeRate` and `_btcFeeRecipient` with the `setMintFeeRate` function and the `setBtcFeeRecipient` function, if the `mintFeeRate` is greater than 0, the protocol will charge a mint fee, however, the fee does not be sent to the fee recipient.

File(s) Affected

contracts/TokenManager.sol #225-233

```
225     function setStatusFunded(
226         bytes32 uuid,
227         string calldata btcTxId
228     ) external override whenNotPaused onlyDLCManagerContract {
229         DLCLink.DLC memory dlc = dlcManager.getDLC(uuid);
230
231         _mintTokens(dlc.creator, _getFeeAdjustedAmount(dlc.valueLocked));
232         emit SetStatusFunded(uuid, btcTxId, dlc.creator);
233     }
```

contracts/TokenManager.sol #121-121

```
121         mintFeeRate = 0; // 0% dlcBTC fee for now
```

contracts/TokenManager.sol #327-330

```
327     function setMintFeeRate(uint256 newMintFeeRate) external onlyDLCAdmin {
328         mintFeeRate = newMintFeeRate;
329         emit SetMintFeeRate(newMintFeeRate);
330     }
```

contracts/TokenManager.sol #346-351

```
346     function setBtcFeeRecipient(
347         string calldata btcFeeRecipient
348     ) external onlyDLCAdmin {
349         _btcFeeRecipient = btcFeeRecipient;
350         emit SetBtcFeeRecipient(btcFeeRecipient);
351     }
```


Recommendation


Recommend sending the mint fee to the fee recipient.

Alleviation Fixed

The team fixed this issue by sending fees to the fee recipient in the commit 45ed634b84e537b6201d27af721df22ca0d71c77.

4. Centralized Roles

 Medium risk

 Security Analyzer

In the **DLCBTC** contract, the owner has the privilege of the following functions:

- **mint**: This function allows the owner to mint new DLCBTC tokens and assign them to a specified address.
- **burn**: This function allows the owner to burn existing DLCBTC tokens from a specified address.

In the **DLCManager** contract, the admin has the privilege of the following functions:

- **pauseContract**: Pauses the contract, preventing further execution of certain functions;
- **unpauseContract**: Unpauses the contract, allowing execution of previously paused functions;
- **getThreshold**: Retrieves the current threshold value for signature validation;
- **setThreshold**: Sets a new threshold value for signature validation;
- **addApprovedSigner**: Adds an approved signer for attestation;
- **removeApprovedSigner**: Removes an approved signer for attestation.

In the **DLCManager** contract, the approved signers has the privilege of the following functions:

- **setStatusFunded**: Confirms that a DLC was 'funded' on the Bitcoin blockchain.
- **postCloseDLC**: Triggered after a closing Tx has been confirmed Bitcoin.

In the **DLCManager** contract, the whitelisted contracts have the privilege of the following functions:

- **createDLC**: Triggers the creation of an Announcement in the Attestor Layer.

In the **TokenManager** contract, the DLC admin has the privilege of the following functions:

- **whitelistAddress**: Whitelist an address for creating new vaults;
- **unwhitelistAddress**: Unwhitelist an address from creating new vaults;
- **setMinimumDeposit**: Set the minimum deposit amount for creating vaults;
- **setMaximumDeposit**: Set the maximum deposit amount for creating vaults;
- **setMintFeeRate**: Set the mint fee rate for minting dlcBTC tokens;
- **setBtcMintFeeRate**: Set the BTC mint fee rate for creating vaults;
- **setBtcRedeemFeeRate**: Set the BTC redeem fee rate for closing vaults;
- **setBtcFeeRecipient**: Set the BTC fee recipient address for fee collection;
- **setWhitelistingEnabled**: Enable or disable whitelisting of addresses for vault creation;
- **updateDLCManagerContract**: Update the DLCManager contract address;
- **transferTokenContractOwnership**: Transfer ownership of the token contract;

- **pauseContract**: Pause the contract to prevent further actions;
- **unpauseContract**: Unpause the contract to allow actions to resume.

File(s) Affected

contracts/DLCBTC.sol #29-35

```
29     function mint(address to, uint256 amount) external onlyOwner {
30         _mint(to, amount);
31     }
32
33     function burn(address from, uint256 amount) external onlyOwner {
34         _burn(from, amount);
35     }
```

contracts/DLCManager.sol #344-374

```
344     //                                ADMIN FUNCTIONS                                //
345     //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
346
347     function pauseContract() external onlyAdmin {
348         _pause();
349     }
350
351     function unpauseContract() external onlyAdmin {
352         _unpause();
353     }
354
355     function getThreshold() external view onlyAdmin returns (uint16) {
356         return _threshold;
357     }
358
359     function setThreshold(uint16 newThreshold) external onlyAdmin {
360         if (newThreshold < _minimumThreshold)
361             revert ThresholdTooLow(_minimumThreshold);
362         _threshold = newThreshold;
363         emit SetThreshold(newThreshold);
364     }
365
366     function addApprovedSigner(address signer) external onlyAdmin {
367         _approvedSigners[signer] = true;
368         _signerCount++;
369     }
370
371     function removeApprovedSigner(address signer) external onlyAdmin {
372         if (_signerCount == _minimumThreshold)
373             revert ThresholdMinimumReached(_minimumThreshold);
374         if (!_approvedSigners[signer]) revert SignerNotApproved(signer);
```


contracts/DLCManager.sol #244-261

```
244     * @param   btcTxId   DLC Funding Transaction ID on the Bitcoin blockchain.
245     * @param   signatures Signatures of the Attestors.
246     */
247     function setStatusFunded(
248         bytes32 uuid,
249         string calldata btcTxId,
250         bytes[] calldata signatures
251     ) external whenNotPaused onlyApprovedSigners {
252         _attestorMultisigIsValid(abi.encode(uuid, btcTxId), signatures);
253         DLCLink.DLC storage dlc = dlcs[dlcIDsByUUID[uuid]];
254
255         if (dlc.uuid == bytes32(0)) revert DLCNotFound();
256         if (dlc.status != DLCLink.DLCStatus.READY) revert DLCNotReady();
257
258         dlc.fundingTxId = btcTxId;
259         dlc.status = DLCLink.DLCStatus.FUNDED;
260
261         DLCLinkCompatible(dlc.protocolContract).setStatusFunded(uuid, btcTxId);
```

contracts/DLCManager.sol #195-235

```
195     * @param   btcRedeemFeeBasisPoints   Basis points of the redeeming fee.
196     * @return   bytes32   A generated UUID.
197     */
198     function createDLC(
199         uint256 valueLocked,
200         string calldata btcFeeRecipient,
201         uint256 btcMintFeeBasisPoints,
202         uint256 btcRedeemFeeBasisPoints
203     )
204         external
205         override
206         onlyWhiteListedContracts
207         whenNotPaused
208         returns (bytes32)
209     {
210         bytes32 _uuid = _generateUUID(tx.origin, _index);
211
212         dlcs[_index] = DLCLink.DLC({
213             uuid: _uuid,
214             protocolContract: msg.sender,
215             valueLocked: valueLocked,
216             timestamp: block.timestamp,
217             creator: tx.origin,
218             status: DLCLink.DLCStatus.READY,
219             fundingTxId: "",
220             closingTxId: "",
221             btcFeeRecipient: btcFeeRecipient,
222             btcMintFeeBasisPoints: btcMintFeeBasisPoints,
223             btcRedeemFeeBasisPoints: btcRedeemFeeBasisPoints
224         });
225
226         emit CreatedDLC(
227             _uuid,
228             valueLocked,
229             msg.sender,
230             tx.origin,
231             block.timestamp
232         );
233
234         dlcIDsByUUID[_uuid] = _index;
235         _index++;
```

contracts/DLCManager.sol #287-307

```
287      * @param   btcTxId   Closing Bitcoin Tx id.
288      * @param   signatures Signatures of the Attestors.
289      */
290      function postCloseDLC(
291          bytes32 uuid,
292          string calldata btcTxId,
293          bytes[] calldata signatures
294      ) external whenNotPaused onlyApprovedSigners {
295          _attestorMultisigIsValid(abi.encode(uuid, btcTxId), signatures);
296          DLCLink.DLC storage dlc = dlcs[dlcIDsByUUID[uuid]];
297
298          if (dlc.uuid == bytes32(0)) revert DLCNotFound();
299          if (dlc.status != DLCLink.DLCStatus.CLOSING) revert DLCNotClosing();
300
301          dlc.closingTxId = btcTxId;
302          dlc.status = DLCLink.DLCStatus.CLOSED;
303
304          DLCLinkCompatible(dlc.protocolContract).postCloseDLCHandler(
305              uuid,
306              btcTxId
307          );
308      }
```

contracts/TokenManager.sol #299-381

```
299     function whitelistAddress(  
300         address addressToWhitelist  
301     ) external onlyDLCAdmin {  
302         _whitelistedAddresses[addressToWhitelist] = true;  
303         emit WhitelistAddress(addressToWhitelist);  
304     }  
305  
306     function unwhitelistAddress(  
307         address addressToUnWhitelist  
308     ) external onlyDLCAdmin {  
309         _whitelistedAddresses[addressToUnWhitelist] = false;  
310         emit UnwhitelistAddress(addressToUnWhitelist);  
311     }  
312  
313     function setMinimumDeposit(  
314         uint256 newMinimumDeposit  
315     ) external onlyDLCAdmin {  
316         minimumDeposit = newMinimumDeposit;  
317         emit SetMinimumDeposit(newMinimumDeposit);  
318     }  
319  
320     function setMaximumDeposit(  
321         uint256 newMaximumDeposit  
322     ) external onlyDLCAdmin {  
323         maximumDeposit = newMaximumDeposit;  
324         emit SetMaximumDeposit(newMaximumDeposit);  
325     }  
326  
327     function setMintFeeRate(uint256 newMintFeeRate) external onlyDLCAdmin {  
328         mintFeeRate = newMintFeeRate;  
329         emit SetMintFeeRate(newMintFeeRate);  
330     }  
331  
332     function setBtcMintFeeRate(  
333         uint256 newBtcMintFeeRate  
334     ) external onlyDLCAdmin {  
335         btcMintFeeRate = newBtcMintFeeRate;  
336         emit SetBtcMintFeeRate(newBtcMintFeeRate);  
337     }  
338  
339     function setBtcRedeemFeeRate(  
340         uint256 newBtcRedeemFeeRate  
341     ) external onlyDLCAdmin {  
342         btcRedeemFeeRate = newBtcRedeemFeeRate;  
343         emit SetBtcRedeemFeeRate(newBtcRedeemFeeRate);  
344     }  
345  
346     function setBtcFeeRecipient(  
347         string calldata btcFeeRecipient  
348     ) external onlyDLCAdmin {  
349         _btcFeeRecipient = btcFeeRecipient;  
350         emit SetBtcFeeRecipient(btcFeeRecipient);  
351     }  
352  
353     function setWhitelistingEnabled(  
354         bool isWhitelistingEnabled  
355     ) external onlyDLCAdmin {
```

```

356         whitelistingEnabled = isWhitelistingEnabled;
357         emit SetWhitelistingEnabled(isWhitelistingEnabled);
358     }
359
360     function updateDLCManagerContract (
361         address newDLCManagerAddress
362     ) external onlyDLCAdmin {
363         dlcManager = IDLCManager(newDLCManagerAddress);
364         _grantRole(DLC_MANAGER_ROLE, newDLCManagerAddress);
365         emit NewDLCManagerContract (newDLCManagerAddress);
366     }
367
368     function transferTokenContractOwnership (
369         address newOwner
370     ) external onlyDLCAdmin {
371         dlcBTC.transferOwnership(newOwner);
372         emit TransferTokenContractOwnership (newOwner);
373     }
374
375     function pauseContract () external onlyPauser {
376         _pause();
377     }
378
379     function unpauseContract () external onlyPauser {
380         _unpause();
381     }

```

Recommendation


Consider implementing a decentralized governance mechanism or a multi-signature scheme that requires consensus among multiple parties before pausing or unpausing the contract. This can help mitigate the centralization risk associated with a single owner controlling critical contract functions. Alternatively, you can provide a clear justification for the centralization aspect and ensure that users are aware of the potential risks associated with a single point of control.


Alleviation Acknowledged

The team responded that the DLCAdmin role will be a Gnosis SAFE Multisig account. The team would post the members/threshold of this multisig on their public channels for providing trust. Eventually, it could be delegated to a DAO or a multisig contract.

Low risk (5)

1. Potential being out of gas as the increasement of **d1cs**

 Low risk

 Security Analyzer

The function **getFundedTxIds** gets DLCs whose status is **FUNDED** by iterating the whole **d1cs** mapping.

The number of funded DLC would not be a big number, however, the **d1cs** mapping increases day by day, and will be a big size one. As a result, the iterating of the whole **d1cs** will cost much gas in the future, moreover, probably result in a out of gas error.

File(s) Affected

contracts/DLCManager.sol #331-341

```

331     function getFundedTxIds() public view returns (string[] memory) {
332         string[] memory _fundedTxIds = new string[](_index);
333         uint256 _fundedTxIdsCount = 0;
334         for (uint256 i = 0; i < _index; i++) {
335             if (dlcs[i].status == DLCLink.DLCStatus.FUNDED) {
336                 _fundedTxIds[_fundedTxIdsCount] = dlcs[i].fundingTxId;
337                 _fundedTxIdsCount++;
338             }
339         }
340         return _fundedTxIds;
341     }

```

Recommendation

Consider getting transactions from the **dlcs** by a specified range, like **startIndex**, and **endIndex**.

Alleviation Fixed

The team fixed this issue by limiting the range for read, in the commit a5ee45262973ac0d80e0894c59787a8d076b0ad5.

2. Upgradeable contracts missing a storage gap variable __gap[50]



Low risk



Security Analyzer

Contracts **TokenManager**, and **DLCManager** are upgradeable contracts, which missing the corresponding storage gap variable **__gap[50]** for the future upgradeable.

Storage Gaps | Openzeppelin: *Storage gaps are empty reserved space in storage that is put in place in Upgradeable contracts. It allows us to freely add new state variables in the future without compromising the storage compatibility with existing deployments.*

File(s) Affected

contracts/DLCManager.sol #28-32

```

28 contract DLCManager is
29     Initializable,
30     AccessControlDefaultAdminRulesUpgradeable,
31     PausableUpgradeable,
32     IDLCManager

```

contracts/TokenManager.sol #34-38

```

34 contract TokenManager is
35     Initializable,
36     AccessControlDefaultAdminRulesUpgradeable,
37     PausableUpgradeable,
38     DLCLinkCompatible

```

Recommendation

Consider adding storage gaps for upgradeable contracts.

Alleviation Fixed

The team fixed this issue by adding a gap variable, in the commit 7a42606b39c0f25639f241bb7b87f2b961690947.

3. The init threshold value from design document mismatches its implementation



Low risk



Security Analyzer

The document [DLC.Link Technical Architecture v1.2](#) mentioned that to mitigate this risk, we are designing for a minimum of 7 total node operators, and a minimum threshold of 4.

But, in the `initialize` function, the `_minimumThreshold` is initialized as 2.

File(s) Affected

contracts/DLCManager.sol #110-110

```
110         _minimumThreshold = 2;
```

Recommendation


Consider checking the design and the implementation, and align them.


Alleviation Acknowledged

The team acknowledged this finding.

The improper values of the `_threshold` and the

4. `_minimumThreshold` may block the `_attestorMultisigIsValid` function

 Low risk

 Security Analyzer

The improper values of the `_threshold` and the `_minimumThreshold` may block the `_attestorMultisigIsValid` function

Taking the below scenario into account:

- The `_minimumThreshold` is initialized as 2;
- The admin sets the `_threshold` as 4 with the `setThreshold` function, the function executes successfully because the condition `4 < 2` is false;
- The admin adds three approved signers, Alice, Bob, and Carol, and removes Carol, which still matches the requirement of the check on the `_minimumThreshold` from the `removeApprovedSigner`;
- Alice or Bob set the DLC as funded with their 2 signatures, which meet the requirement of the minimum threshold that is 2, then, the check on the condition `signatures.length < _threshold` returns true, which results in a revert.

In a nutshell, in the above scenario, the number of approved signatures matches the requirement of the minimum threshold `_minimumThreshold`, but does not match the threshold `_threshold`.

File(s) Affected

contracts/DLCManager.sol #359-377

```

359     function setThreshold(uint16 newThreshold) external onlyAdmin {
360         if (newThreshold < _minimumThreshold)
361             revert ThresholdTooLow(_minimumThreshold);
362         _threshold = newThreshold;
363         emit SetThreshold(newThreshold);
364     }
365
366     function addApprovedSigner(address signer) external onlyAdmin {
367         _approvedSigners[signer] = true;
368         _signerCount++;
369     }
370
371     function removeApprovedSigner(address signer) external onlyAdmin {
372         if (_signerCount == _minimumThreshold)
373             revert ThresholdMinimumReached(_minimumThreshold);
374         if (!_approvedSigners[signer]) revert SignerNotApproved(signer);
375         _approvedSigners[signer] = false;
376         _signerCount--;
377     }

```

contracts/DLCManager.sol #163-163

```

163         if (signatures.length < _threshold) revert NotEnoughSignatures();

```


Recommendation


Consider checking the thresholds design, and ensuring the minimum threshold meets the requirement of the `_attestorMultisigIsValid` function.

Alleviation Fixed

The team solved this issue by adding a check between the threshold and the `_minimumThreshold` in the commit `08ad24555bea64d5a678f1a68c88165af4922217`.

5. Upgradeable contracts missing a storage gap variable `__gap[50]`

 Low risk

 Security Analyzer

For the PR(<https://github.com/DLC-link/dlc-solidity/pull/47/files>), the contract `DLCBTC`(<https://github.com/DLC-link/dlc-solidity/blob/4a778ca4015be74775e6f8afab94d5261ffa37c6/contracts/DLCBTC.sol#L23>) misses the corresponding storage gap variable `__gap[50]` for the future upgradeable.

Storage Gaps | Openzeppelin: Storage gaps are empty reserved space in storage that is put in place in Upgradeable contracts. It allows us to freely add new state variables in the future without compromising the storage compatibility with existing deployments.

File(s) Affected

Recommendation


Consider adding storage gaps for upgradeable contracts.


Alleviation Fixed

The finding is addressed in the commit `9eb8411ede3befe856f34468bf8f84153a3ccc38`

Informational (5)

1. Lack of validating the `_threshold`

 Informational

 Security Analyzer

The `setThreshold` function checks that the `newThreshold` aka the `_threshold` should equal to or be greater than the `_minimumThreshold`.

However, the `initialize` function lacks checking if the `_threshold` is less than the `_minimumThreshold`, which may receive invalid values for `_threshold` and `_minimumThreshold`.

File(s) Affected

contracts/DLCManager.sol #102-111

```
102     function initialize(  
103         address adminAddress,  
104         uint16 threshold  
105     ) public initializer {  
106         __AccessControlDefaultAdminRules_init(2 days, adminAddress);  
107         _grantRole(DLC_ADMIN_ROLE, adminAddress);  
108         _threshold = threshold;  
109         _index = 0;  
110         _minimumThreshold = 2;  
111     }
```

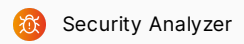
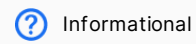
Recommendation

Recommend validating the `_threshold` from the `initialize` function so as to be consistent with the logic of the `setThreshold` function.

Alleviation Fixed

The team fixed this issue by adding check on the `threshold`, in the commit 08ad24555bea64d5a678f1a68c88165af4922217.

2. Gas optimazation



The `getFundedTxIds` function repeatedly read the storage variable `_index`, which cost many gas, due to the read of storage variable is more expensive than that of read of the memory variable, epecially the `for` loop reads the `_index` mutple times.

File(s) Affected

contracts/DLCManager.sol #332-334

```
332     string[] memory _fundedTxIds = new string[](_index);  
333     uint256 _fundedTxIdsCount = 0;  
334     for (uint256 i = 0; i < _index; i++) {
```

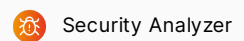
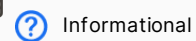
Recommendation

Consider caching the `_index` with a memory variable to save gas.

Alleviation Fixed

The team fixed this issue by refactoring codes, in the commit a5ee45262973ac0d80e0894c59787a8d076b0ad5

3. Combining the chain id and contract address when calculating the UUID



In order to keep the UUID unique across multiple chains and multiple contract, it is recommended taking the chain id and current contract address into account when calculating the UUID, as DLC.link will support multiple EVM compatible chains.

File(s) Affected

contracts/DLCManager.sol #142-150

```
142     function _generateUUID(
143         address sender,
144         uint256 nonce
145     ) private view returns (bytes32) {
146         return
147             keccak256(
148                 abi.encodePacked(sender, nonce, blockhash(block.number - 1))
149             );
150     }
```



Recommendation

Recommend combining the chain id and current contract address when calculating the UUID.

Alleviation Acknowledged

The team mitigated this finding by involving the chain id, in the calculation of UUID, in the commit 1e7b04de83dadba8aa4bd46b264dbc8729c2e163.

4. Lack of the fee rate boundary

 Informational Security Analyzer

The functions `setMintFeeRate`, `setBtcMintFeeRate`, and `setBtcRedeemFeeRate`, update the fee rates `mintFeeRate`, `btcMintFeeRate`, and `btcRedeemFeeRate`. However, there is no boundary for their values, which may result in unexpected results. E.g. if the `mintFeeRate` is 10000, the recipient will not gain any token.

File(s) Affected

contracts/TokenManager.sol #327-344

```
327     function setMintFeeRate(uint256 newMintFeeRate) external onlyDLCAdmin {
328         mintFeeRate = newMintFeeRate;
329         emit SetMintFeeRate(newMintFeeRate);
330     }
331
332     function setBtcMintFeeRate(
333         uint256 newBtcMintFeeRate
334     ) external onlyDLCAdmin {
335         btcMintFeeRate = newBtcMintFeeRate;
336         emit SetBtcMintFeeRate(newBtcMintFeeRate);
337     }
338
339     function setBtcRedeemFeeRate(
340         uint256 newBtcRedeemFeeRate
341     ) external onlyDLCAdmin {
342         btcRedeemFeeRate = newBtcRedeemFeeRate;
343         emit SetBtcRedeemFeeRate(newBtcRedeemFeeRate);
344     }
```


Recommendation

Recommend adding fee rate boundary.

Alleviation Fixed

The team fixed this finding by adding the boundary check on the fees, in the commit c00c66da585a4facad80041af8e966a081c62fad.

5. Missing emit event

 Informational Security Analyzer

For the PR(<https://github.com/DLC-link/dlc-solidity/pull/47/files>), functions, blacklist, and unblacklist(<https://github.com/DLC-link/dlc-solidity/blob/4a778ca4015be74775e6f8afab94d5261ffa37c6/contracts/DLCBTC.sol#L58-L62>) update state variables are

recommended to emit event, which is good for tracking states' update.

File(s) Affected**Recommendation**

Emit event for the above key functions.

Alleviation Fixed

The finding is addressed in the commit [9eb8411ede3befe856f34468bf8f84153a3ccc38](#)

Disclaimer

This report is governed by the stipulations (including but not limited to service descriptions, confidentiality, disclaimers, and liability limitations) outlined in the Services Agreement, or as detailed in the scope of services and terms provided to you, the Customer or Company, within the context of the Agreement. The Company is permitted to use this report only as allowed under the terms of the Agreement. Without explicit written permission from MetaTrust, this report must not be shared, disclosed, referenced, or depended upon by any third parties, nor should copies be distributed to anyone other than the Company.

It is important to clarify that this report neither endorses nor disapproves any specific project or team. It should not be viewed as a reflection of the economic value or potential of any product or asset developed by teams or projects engaging MetaTrust for security evaluations. This report does not guarantee that the technology assessed is completely free of bugs, nor does it comment on the business practices, models, or legal compliance of the technology's creators.

This report is not intended to serve as investment advice or a tool for investment decisions related to any project. It represents a thorough assessment process aimed at enhancing code quality and mitigating risks inherent in cryptographic tokens and blockchain technology. Blockchain and cryptographic assets inherently carry ongoing risks. MetaTrust's role is to support companies and individuals in their security diligence and to reduce risks associated with the use of emerging and evolving technologies. However, MetaTrust does not guarantee the security or functionality of the technologies it evaluates.

MetaTrust's assessment services are contingent on various dependencies and are continuously evolving. Accessing or using these services, including reports and materials, is at your own risk, on an as-is and as-available basis. Cryptographic tokens are novel technologies with inherent technical risks and uncertainties. The assessment reports may contain inaccuracies, such as false positives or negatives, and unpredictable outcomes. The services may rely on multiple third-party layers.

All services, labels, assessment reports, work products, and other materials, or any results from their use, are provided "as is" and "as available," with all faults and defects, without any warranty. MetaTrust expressly disclaims all warranties, whether express, implied, statutory, or otherwise, including but not limited to warranties of merchantability, fitness for a particular purpose, title, non-infringement, and any warranties arising from course of dealing, usage, or trade practice. MetaTrust does not guarantee that the services, reports, or materials will meet specific requirements, be error-free, or be compatible with other software, systems, or services.

Neither MetaTrust nor its agents make any representations or warranties regarding the accuracy, reliability, or currency of any content provided through the services. MetaTrust is not liable for any content inaccuracies, personal injuries, property damages, or any loss resulting from the use of the services, reports, or materials.

Third-party materials are provided "as is," and any warranty concerning them is strictly between the Customer and the third-party owner or distributor. The services, reports, and materials are intended solely for the Customer and should not be relied upon by others or shared without MetaTrust's consent. No third party or representative thereof shall have any rights or claims against MetaTrust regarding these services, reports, or materials.

The provisions and warranties of MetaTrust in this agreement are exclusively for the Customer's benefit. No third party has any rights or claims against MetaTrust regarding these provisions or warranties. For clarity, the services, including any assessment reports or materials, should not be used as financial, tax, legal, regulatory, or other forms of advice.