Security Audit Report for Delorean Self Insured Vaults

Date: June 2, 2023

Version: 1.0

Contact: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by BlockSec Audit Team
Date: 2023.06.02 08:32:08 +02'00'

Contents

Report Manifest

Item	Description
Client	Delorean Exchange
Target	Delorean Self Insured Vaults

Version History

Version	Date	Description
1.0	June 2, 2023	First Release

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 5 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the smart contracts [1] of the Delorean Exchange, i.e., self insured vaults use yield from the insured token to pay for their own insurance. Note that, the scope of this audit does **NOT** cover all the modules in the repository. Only the following three smart contracts are audited:

- src/providers/Y2KEarthquakeV1InsuranceProvider.sol
- src/vaults/SelfInsuredVault.sol
- src/libraries/Math.sol

Hence in this audit, we assume that all the other code and the corresponding dependency (e.g., Y2K [2]) are reliable.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Delorean	`Version 1`	`75ec1b8342bee44add65e143c1382c4ab7739b55`
	`Version 2`	`3a74abdcd4819b666d9e2b9bee4425d35e1e19ee`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

[1] https://github.com/delorean-exchange/siv-contracts/tree/initial

[2] https://github.com/Y2K-Finance/Earthquake

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Auditing Approaches

This section provides an overview of the auditing approaches we adopted and the corresponding checkpoints we focused on. Specifically, we conduct the audit by engaging the following approaches:

- **Static analysis**. We scan smart contracts with both open-source and in-house tools, and then manually verify (reject or confirm) the issues reported by them.
- **Fuzzing testing**. We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an in-house fuzzing tool (developed and customized by our security research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Manual code review**. We manually review the design and the corresponding implementation of the whole project in a comprehensive manner, and check the known attack surface accordingly.

Generally, the checkpoints fall into four categories, i.e., **software security**, **DeFi security**, **NFT security** and **additional recommendation**. The main concrete checkpoints are summarized in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The above checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [3] and Common Weakness Enumeration [4]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[3] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[4] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find **four** potential issues. Besides, we also have **five** recommendations.

- High Risk: 2
- Medium Risk: 2
- Recommendation: 5

ID	Severity	Description	Category	Status
1	Medium	Lack of validation for duplicated reward tokens	DeFi Security	Fixed
2	Medium	Lack of validation for duplicated providers	DeFi Security	Fixed
3	High	Incorrect payout assignment	DeFi Security	Fixed
4	High	Improper update of 'totalShares'	DeFi Security	Fixed
5	-	Remove unused code from the function '_purchaseForNextEpoch'	Recommendation	Fixed
6	-	Remove redundant check for the function 'claimRewards'	Recommendation	Fixed
7	-	Remove unused code regarding the function '_projectEpochYield'	Recommendation	Fixed
8	-	Remove unused variable 'nextEpochId' from the function '_computeAccumulatePayouts'	Recommendation	Fixed
9	-	Improper update for 'claimedEpochIndex'	Recommendation	Fixed

The details are provided in the following sections.

2.1 Software Security

2.2 DeFi Security

2.2.1 Lack of validation for duplicated reward tokens

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description

As shown in listing 2.1, the function `addRewardToken` does not verify the existence of the input `rewardToken`. Therefore, the list `rewardTokens` might have duplicated reward tokens if the admin mistakenly adds the existing token.

```
526    function addRewardToken(address rewardToken) external onlyAdmin {
527        require(rewardToken != address(0), "SIV: zero reward token");
528        rewardTokens.push(rewardToken);
529    }
```

Listing 2.1: SelfInsuredVault.sol

As a result, the function `claimRewards` would always fail due to the duplicated reward tokens. For example, there is a duplicated reward token `T` existing in the list `rewardTokens` with index `i` and `j`, and

the user has `x` amount of token `T` accumulated in the vault. Once the user invokes `claimRewards()`, `owed` will obtain the accumulated reward tokens for the user. Then, the following `for` loop will go through `owed` and transfer the accumulated reward tokens to the user. Since there is a duplicated token `T`, `owed[i] == owed[j] == x`. However, the i-th iteration will update `userYieldInfos[msg.sender][T].accumulatedYield` to zero. Therefore, in the j-th iteration, the check (at line 462) can never be passed.

```
451    function claimRewards() external returns (uint256[] memory) {
452        _harvest();
453
454        uint256[] memory owed = _previewClaimRewards(msg.sender);
455
456        require(owed.length == rewardTokens.length, "SIV: claim length");
457
458        for (uint8 i = 0; i < uint8(owed.length); i++) {
459            address t = address(rewardTokens[i]);
460
461            _updateYield(msg.sender, t);
462            require(owed[i] == userYieldInfos[msg.sender][t].accumulatedYield, "SIV: claim acc");
463
464            userYieldInfos[msg.sender][t].accumulatedYield = 0;
465
466            IERC20(rewardTokens[i]).safeTransfer(msg.sender, owed[i]);
467            globalYieldInfos[t].claimedYield += owed[i];
468        }
469
470        return owed;
471    }
```

Listing 2.2: SelfInsuredVault.sol

Impact Users can never claim their rewards due to the duplicated reward token.

Suggestion Add a validation check in the function `addRewardTokens` to avoid duplicated reward tokens.

2.2.2 Lack of validation for duplicated insurance providers

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description

As shown in listing 2.3, the function `addInsuranceProvicer` does not verify the existence of the input provider in the list `providers`. Therefore, the list `providers` might have duplicated providers if the admin mistakenly adds the existing provider.

```
531    function addInsuranceProvider(IInsuranceProvider provider_, uint256 weight_) external
           onlyAdmin {
532        require(providers.length == 0 ||
533                provider_.epochDuration() == providers[0].epochDuration(), "SIV: same duration");
534        require(provider_.paymentToken() == paymentToken, "SIV: payment token");
535        require(providers.length < MAX_PROVIDERS, "SIV: max providers");
536
```

```
537         uint256 sum = weight_;
538         for (uint256 i = 0; i < weights.length; i++) {
539             sum += weights[i];
540         }
541
542         require(sum < MAX_COMBINED_WEIGHT, "SIV: max weight");
543
544         providers.push(provider_);
545         weights.push(weight_);
546     }
```

Listing 2.3: SelfInsuredVault.sol::addInsuranceProvider

As shown in listing 2.4, since there is a check for the duplication of providers (lines 481-498), the user may claim extra payouts.

```
474     function _pendingPayouts(address who) internal view returns (uint256) {
475         uint256 deltaAccumulatdPayouts = _computeAccumulatePayouts(who);
476
477         // `deltaAccumulatdPayouts` includes [startEpochId, nextEpochId), but we
478         // also want [nextEpochId, currentEpochId].
479         uint256 accumulatedPayouts;
480         UserEpochTracker storage tracker = userEpochTrackers[who];
481         for (uint256 i = 0; i < providers.length; i++) {
482             IInsuranceProvider provider = providers[i];
483             uint256 currentEpochId = provider.currentEpoch();
484
485             EpochInfo[] storage infos = providerEpochs[address(provider)];
486
487             // TODO: GAS: Use nextId field + mapping instead of list
488             for (uint256 j = 0; j < infos.length; j++) {
489                 EpochInfo storage info = infos[j];
490                 if (info.epochId < tracker.nextEpochId) continue;
491
492                 accumulatedPayouts += (tracker.nextShares * info.payout) / info.totalShares;
493
494                 // Check below expected to be redundant, since the last element
495                 // should be for the current epoch
496                 if (info.epochId == currentEpochId) break;
497             }
498         }
499
500         return (userEpochTrackers[who].accumulatedPayouts +
501                 accumulatedPayouts +
502                 deltaAccumulatdPayouts -
503                 userEpochTrackers[who].claimedPayouts);
504     }
```

Listing 2.4: SelfInsuredVault.sol::_pendingPayouts

Impact Users can claim extra payouts due to duplicated insurance providers.

Suggestion Add a validation check to avoid duplicated reward tokens in the function `addInsuranceProvider`.

2.2.3 Incorrect payout assignment

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description

As shown in listing 2.5, the function `claimVaultPayouts` collects all payouts from unclaimed epochs for different insurance providers and assigns all claimed payouts in the last epoch of `providerEpoch`. This is problematic and might result in incorrect payout assignments for users.

```
576    function claimVaultPayouts() external {
577        for (uint256 i = 0; i < providers.length; i++) {
578            IERC20 pt = providers[i].paymentToken();
579            uint256 before = pt.balanceOf(address(this));
580            uint256 amount = providers[i].claimPayouts();
581            assert(amount == pt.balanceOf(address(this)) - before);
582            if (amount > 0) {
583                EpochInfo[] storage infos = providerEpochs[address(providers[i])];
584                infos[infos.length - 1].payout += amount;
585            }
586        }
587    }
```

Listing 2.5: SelfInsuredVault.sol::claimVaultPayouts

Impact The problematic collection of unclaimed payouts will result in incorrect payout assignments for users.

Suggestion Use a variable such as 'epochId' or 'epochIndex' as input for the function `claimPayouts` to claim the payout accordingly.

2.2.4 Improper update of 'totalShares'

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description

Based on the code (listing 2.6), the function `_updateProviderEpochs` updates the `providerEpochs` if it is unsynchronized with `vault.epochs()`. However, this update could be problematic if there are no pending epochs. Specifically, when there is a deposit or withdrawal (which triggers the function `_updateProviderEpochs`) and there is no pending epoch, the `deltaShare` will be updated to the last epoch of `providerEpochs[provider]`.

```
387    function _updateProviderEpochs(int256 deltaShares) internal {
388        for (uint256 i = 0; i < providers.length; i++) {
389            IInsuranceProvider provider = providers[i];
390            EpochInfo[] storage epochs = providerEpochs[address(provider)];
391
392            // Create first EpochInfo, if needed
```

```
393             if (epochs.length == 0) {
394                 epochs.push(EpochInfo(provider.nextEpoch(), 0, 0, 0));
395             }
396             EpochInfo storage epochInfo = epochs[epochs.length - 1];
397             uint256 totalShares = epochInfo.totalShares;
398
399             // Add new EpochInfo's, if needed
400             uint256 id = provider.followingEpoch(epochInfo.epochId);
401             while (id != 0) {
402                 epochs.push(EpochInfo(id, totalShares, 0, 0));
403                 epochInfo = epochs[epochs.length - 1];
404                 id = provider.followingEpoch(id);
405             }
406
407             epochInfo.totalShares = deltaShares > 0
408                 ? epochInfo.totalShares + uint256(deltaShares)
409                 : epochInfo.totalShares - uint256(-deltaShares);
410         }
411     }
```

Listing 2.6: SelfInsuredVault.sol::_updateProviderEpochs

If the last epoch has unclaimed payouts, this mistaken update could be leveraged by malicious users to amplify their ratio of shares in the last epoch and claim all payouts with a flash loan. Such an attack can be achieved by following the steps below:

1. Borrow a large number of tokens via flash loan.
2. Invoke the function `deposit` to update shares in the last epoch. Due to a large amount of deposit, the malicious user can amplify his/her share ratio to approximately one.
3. Invoke the functions `claimVaultPayouts` and `claimPayouts` to collect unclaimed payouts in the last epoch. Based on the manipulated share ratio, the malicious user could loot nearly all payouts to his/herself.
4. Invoke the function `withdraw` to get out all deposited assets.
5. Repay the flash loan borrowed in step 1.

It is worth noting that the above steps can be done within a single transaction with a customized contract implementing steps 2-3. Moreover, the malicious user does not need to actually own a large number of tokens to launch this attack due to the nature of the flash loan.

Impact The malicious users can amplify their share ratio in the last epoch and loot all unclaimed payouts with a flash loan.

Suggestion Only allow users to deposit when there is a pending epoch.

2.3 NFT Security

2.4 Additional Recommendation

2.4.1 Remove unused code from the function '_purchaseForNextEpoch'

Status Fixed in `Version 2`

Introduced by Version 1

Description

The variable assigned at line 601 of listing 2.7 is unused.

```
589   function _purchaseForNextEpoch(uint256 i, uint256 amount) internal {
590       IInsuranceProvider provider = providers[i];
591       require(provider.isNextEpochPurchasable(), "SIV: not purchasable");
592
593       uint256 nextEpochId = provider.nextEpoch();
594       EpochInfo[] storage epochs = providerEpochs[address(provider)];
595       if (epochs.length == 0 || epochs[epochs.length - 1].epochId != nextEpochId) {
596           epochs.push(EpochInfo(nextEpochId, 0, 0, 0));
597       }
598       EpochInfo storage epochInfo = epochs[epochs.length - 1];
599       require(epochInfo.premiumPaid == 0, "SIV: already purchased");
600
601       uint256 weight = weights[i];
602
603       IERC20(provider.paymentToken()).approve(address(provider), amount);
604       provider.purchaseForNextEpoch(amount);
605       epochInfo.premiumPaid = amount;
606   }
```

Listing 2.7: SelfInsuredVault.sol::_purchaseForNextEpoch

Impact Extra gas consumption.

Suggestion Remove the redundant code (i.e., line 601).

2.4.2 Remove redundant check for the function 'claimRewards'

Status Fixed in Version 2

Introduced by Version 1

Description

The first require check in the function `claimRewards` (i.e., line 456 of listing 2.8) is redundant, since the variable `owed` is constructed based on `rewardTokens.length` in the function `previewClaimRewards` (i.e., line 440 of listing 2.9.

```
451   function claimRewards() external returns (uint256[] memory) {
452       _harvest();
453
454       uint256[] memory owed = _previewClaimRewards(msg.sender);
455
456       require(owed.length == rewardTokens.length, "SIV: claim length");
457
458       for (uint8 i = 0; i < uint8(owed.length); i++) {
459           address t = address(rewardTokens[i]);
460
461           _updateYield(msg.sender, t);
462           require(owed[i] == userYieldInfos[msg.sender][t].accumulatedYield, "SIV: claim acc");
463
464           userYieldInfos[msg.sender][t].accumulatedYield = 0;
```

```
465
466             IERC20(rewardTokens[i]).safeTransfer(msg.sender, owed[i]);
467             globalYieldInfos[t].claimedYield += owed[i];
468         }
469
470         return owed;
471     }
```

Listing 2.8: SelfInsuredVault.sol::claimRewards

```
439     function _previewClaimRewards(address who) internal view returns (uint256[] memory) {
440         uint256[] memory result = new uint256[](rewardTokens.length);
441         for (uint256 i = 0; i < result.length; i++) {
442             result[i] = _calculatePendingYield(who, rewardTokens[i]);
443         }
444         return result;
445     }
```

Listing 2.9: SelfInsuredVault.sol::_previewClaimRewards

Impact Extra gas consumption.

Suggestion Remove the redundant code (i.e., line 456 of listing 2.8).

2.4.3 Remove unused code regarding the function '_projectEpochYield'

Status Fixed in `Version 2`

Introduced by `Version 1`

Description

As shown in listing 2.10, the function `_projectEpochYield` is not used in the contract anymore. Therefore, the function `_projectEpochYield` can be removed. Correspondingly, the functions and variables regarding `YieldOracle` can be removed too.

```
616     function purchaseInsuranceForNextEpoch(uint256 minBps, uint256 projectedYield) external
            onlyAdmin {
617         /* uint256 projectedYield = _projectEpochYield(); */
618
619         // Get epoch's yield upfront via Delorean
620         uint256 sum = 0;
621         for (uint256 i = 0; i < weights.length; i++) {
622             sum += (projectedYield * weights[i]) / WEIGHTS_PRECISION;
623         }
624         uint256 minOut = (sum * minBps) / 100_00;
625         ...
```

Listing 2.10: SelfInsuredVault.sol::purchaseInsuranceForNextEpoch

Impact Extra gas consumption.

Suggestion Remove the redundant code.

2.4.4 Remove unused variable 'nextEpochId' from the function '_computeAccumulatePayouts'

Status Fixed in `Version 2`

Introduced by `Version 1`

Description

The variable `nextEpochId` at line 354 of listing 2.11 is not used and has not effect to the logic.

```
345    function _computeAccumulatePayouts(address user) internal view returns (uint256) {
346        UserEpochTracker storage tracker = userEpochTrackers[user];
347        if (tracker.startEpochId == 0) return 0;
348        if (tracker.shares == 0) return 0;
349
350        uint256 deltaAccumulatedPayouts;
351
352        for (uint256 i = 0; i < providers.length; i++) {
353            IInsuranceProvider provider = providers[i];
354            uint256 nextEpochId = provider.nextEpoch();
355            uint256 currentEpochId = provider.currentEpoch();
356            EpochInfo[] storage infos = providerEpochs[address(provider)];
357
358            // TODO: GAS: Use nextId field + mapping instead of list
359            for (uint256 j = 0; j < infos.length; j++) {
360                EpochInfo storage info = infos[j];
361                if (info.epochId < tracker.startEpochId) continue;
362                if (info.epochId >= tracker.nextEpochId) break;
363                if (info.epochId == currentEpochId) break;
364                deltaAccumulatedPayouts += (tracker.shares * info.payout) / info.totalShares;
365            }
366        }
367
368        return deltaAccumulatedPayouts;
369    }
```

Listing 2.11: SelfInsuredVault.sol::_computeAccumulatePayouts

Impact Extra gas consumption.

Suggestion Remove the redundant code.

2.4.5 Improper update for 'claimedEpochIndex'

Status Fixed in `Version 2`

Introduced by `Version 1`

Description

The function `claimPayouts` (as shown in listing 2.12) possibly invokes `_claimPayoutForEpoch` in multiple times. As a result, the variable `claimedEpochIndex` will be updated repeatedly in the function `_claimPayoutForEpoch` (i.e., line 137 of listing 2.13).

```
141    function claimPayouts() external override returns (uint256) {
142        uint256 amount = 0;
```

```
143         uint256 len = vault.epochsLength();
144         // TODO: Double check this logic, as it may not be 100% right.
145         // Does it correctly handle the current epoch?
146         for (uint256 i = claimedEpochIndex; i < len; i++) {
147             amount += _claimPayoutForEpoch(vault.epochs(i));
148         }
149
150         paymentToken.safeTransfer(beneficiary, amount);
151         return amount;
152     }
```

Listing 2.12: Y2KEarthquakeV1InsuranceProvider.sol::claimPayouts

```
134     function _claimPayoutForEpoch(uint256 epochId) internal returns (uint256) {
135         uint256 assets = vault.balanceOf(address(this), epochId);
136         uint256 amount = vault.withdraw(epochId, assets, address(this), address(this));
137         claimedEpochIndex = vault.epochsLength();
138         return amount;
139     }
```

Listing 2.13: Y2KEarthquakeV1InsuranceProvider.sol::_claimPayoutForEpoch

Impact Extra gas consumption.

Suggestion Remove the update of `claimedEopchIndex` from the internal function `_claimPayoutForEpoch` and update `claimedEopchIndex` externally in the function `claimPayouts`.