



# SMART CONTRACT AUDIT REPORT

for

## Gearbox Protocol



Prepared By: Yiqun Chen

PeckShield  
August 10, 2021

## Document Properties

Client	Gearbox
Title	Smart Contract Audit Report
Target	Gearbox
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Jing Wang, Shulin Bie, Xiaotao Wu
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

## Version Info

Version	Date	Author(s)	Description
1.0	August 10, 2021	Xuxian Jiang	Final Release
1.0-rc1	July 22, 2021	Xuxian Jiang	Release Candidate #1

## Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	About Gearbox . . . . .	4
1.2	About PeckShield . . . . .	5
1.3	Methodology . . . . .	5
1.4	Disclaimer . . . . .	6
<b>2</b>	<b>Findings</b>	<b>10</b>
2.1	Summary . . . . .	10
2.2	Key Findings . . . . .	11
<b>3</b>	<b>Detailed Results</b>	<b>12</b>
3.1	Proper Liquidity Limit Enforcement in PoolService . . . . .	12
3.2	Duplicate Avoidance in connectCreditManager() . . . . .	13
3.3	Proper MAX_INT_4 Calculation . . . . .	14
3.4	Improved Precision By Multiplication And Division Reordering . . . . .	15
3.5	Reentrancy Protection in CreditAccount Lending/Repayment . . . . .	17
<b>4</b>	<b>Conclusion</b>	<b>19</b>
	<b>References</b>	<b>20</b>

# 1 | Introduction

Given the opportunity to review the **Gearbox** design document and related smart contract source code, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

## 1.1 About Gearbox

**Gearbox** protocol is aiming to enhance capital efficiency in the DeFi space with the introduction of new DeFi primitives for under-collateralized interactions with other DeFi lego blocks which can be used for decentralized margin trading, leveraged yield farming, etc. This is made possible with **Credit Accounts**, which are agent-oriented isolated smart contracts, and liquidation thresholds protecting the bottom line of the deposited capital. Overall, **Gearbox** does not only give users and other decentralized protocols access to leverage their trading and farming positions (and therefore maximize their profit), but also ensures a non-custodial, transparent, and composable design of the protocol.

The basic information of Gearbox is as follows:

Table 1.1: Basic Information of Gearbox

Item	Description
Issuer	Gearbox
Website	<a href="https://gearbox.finance/">https://gearbox.finance/</a>
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 10, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used

in this audit. Note that Gearbox assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit.

- <https://github.com/Gearbox-protocol/gearbox-contracts.git> (2600a60)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- <https://github.com/Gearbox-protocol/gearbox-contracts.git> (cee91b9)

## 1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (<https://t.me/peckshield>), Twitter (<http://twitter.com/peckshield>), or Email ([contact@peckshield.com](mailto:contact@peckshield.com)).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

## 1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
<b>Configuration</b>	Weaknesses in this category are typically introduced during the configuration of the software.
<b>Data Processing Issues</b>	Weaknesses in this category are typically found in functionality that processes data.
<b>Numeric Errors</b>	Weaknesses in this category are related to improper calculation or conversion of numbers.
<b>Security Features</b>	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
<b>Time and State</b>	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
<b>Error Conditions, Return Values, Status Codes</b>	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
<b>Resource Management</b>	Weaknesses in this category are related to improper management of system resources.
<b>Behavioral Issues</b>	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
<b>Business Logic</b>	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
<b>Initialization and Cleanup</b>	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
<b>Arguments and Parameters</b>	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
<b>Expression Issues</b>	Weaknesses in this category are related to incorrectly written expressions within code.
<b>Coding Practices</b>	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.






comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.



## 2 | Findings

### 2.1 Summary

Here is a summary of our findings after analyzing the implementation of the Gearbox protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	
Low	3	
Informational	1	
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in [Section 3](#).

## 2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Gearbox Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Proper Liquidity Limit Enforcement in PoolService	Coding Practices	Fixed
PVE-002	Low	Duplicate Avoidance in connectCreditManager()	Coding Practices	Fixed
PVE-003	Low	Proper MAX_INT_4 Calculation	Numeric Errors	Fixed
PVE-004	Low	Improved Precision By Multiplication And Division Reordering	Coding Practices	Fixed
PVE-005	Informational	Reentrancy Protection in CreditAccount Lending/Repayment	Time and State	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

## 3 | Detailed Results

### 3.1 Proper Liquidity Limit Enforcement in PoolService

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Low
- Target: PoolService
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

#### Description

In the Gearbox protocol, there is a `PoolService` contract that allows any one to provide liquidity and support credit accounts for margin trading and leveraged farming. While examining the `PoolService` contract, we notice the guarded launch feature with the `expectedLiquidityLimit` enforcement can be improved.

To elaborate, we show below the `addLiquidity()` routine. It implements a rather straightforward logic in transferring the intended asset to the pool, then minting the proportional pool tokens (i.e., `diesel`) with current `diesel` rate, and finally updating the expected pool liquidity and the pool borrow rate. Note the enforcement of `expectedLiquidityLimit` needs to be performed by taking into account the new liquidity just provided, i.e., `require(expectedLiquidity()+amount < expectedLiquidityLimit)`, instead of current `require(expectedLiquidity() < expectedLiquidityLimit)` (line 145).

```
134     function addLiquidity(  
135         uint256 amount,  
136         address onBehalfOf,  
137         uint256 referralCode  
138     )  
139     external  
140     override  
141     whenNotPaused // T:[PS-4]  
142     nonReentrant  
143     {  
144         require(  
145             expectedLiquidity() < expectedLiquidityLimit,
```

```

146         Errors.POOL_MORE_THAN_EXPECTED_LIQUIDITY_LIMIT
147     ); // T:[PS-31]

149     IERC20(underlyingToken).safeTransferFrom(
150         msg.sender,
151         address(this),
152         amount
153     ); // T:[PS-2, 7]

155     DieselToken(dieselToken).mint(onBehalfOf, toDiesel(amount)); // T:[PS-2, 7]

157     _expectedLiquidityLU = _expectedLiquidityLU.add(amount); // T:[PS-2, 7]
158     _updateBorrowRate(); // T:[PS-2, 7]

160     emit AddLiquidity(msg.sender, onBehalfOf, amount, referralCode); // T:[PS-2, 7]
161 }

```

Listing 3.1: PoolService::addLiquidity()

**Recommendation** Revise the above addLiquidity() routine to properly enforce the expectedLiquidityLimit invariant.

**Status** The issue has been fixed by this commit: a2ec6e4.

## 3.2 Duplicate Avoidance in connectCreditManager()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: PoolService
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

### Description

As mentioned in Section 3.1, the Gearbox protocol has a PoolService contract that allows any one to provide liquidity and support credit accounts for margin trading and leveraged farming. The funds in PoolService may be borrowed by credit accounts via the trusted credit manager. To facilitate the management, the PoolService contract provides a privileged function connectCreditManager(), which connects a new credit manager to the pool.

```

415     /// @dev Connects new Credif manager to pool
416     /// @param _creditManager Address of credif manager
417     function connectCreditManager(address _creditManager)
418         external
419         configuratorOnly // T:[PS-9]
420     {

```

```

421     require(
422         address(this) == ICreditManager(_creditManager).poolService(),
423         Errors.POOL_INCOMPATIBLE_CREDIT_ACCOUNT_MANAGER
424     ); // T:[PS-10]

426     creditManagersCanBorrow[_creditManager] = true; // T:[PS-11]
427     creditManagersCanRepay[_creditManager] = true; // T:[PS-11]
428     creditManagers.push(_creditManager); // T:[PS-11]
429     emit NewCreditManagerConnected(_creditManager); // ToDo: ADD CHECK HERE
430 }

```

Listing 3.2: PoolService::connectCreditManager()

To elaborate, we show above the `connectCreditManager()` routine. The logic is rather straightforward in saving the new credit manager in its internal array `creditManagers`. It comes to our attention that the current implementation does not deal with possible duplicate that may already exist in the `creditManagers` array.

**Recommendation** Revise the above `connectCreditManager()` routine to prevent an existing entry from being added again.

**Status** The issue has been fixed by this commit: `a2ec6e4`.

### 3.3 Proper MAX\_INT\_4 Calculation

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Constants
- Category: Numeric Errors [8]
- CWE subcategory: CWE-190 [2]

#### Description

The Gearbox aims to enhance capital efficiency in DeFi with the introduction of `Credit Accounts` - a new primitive for under-collateralized interactions with other DeFi protocols. One common need behind the interaction with various DeFi protocols is the efficient management of token allowance of a credit account on an external contract.

To facilitate such need, the `Credit Manager` contract provides a helper routine `_provideCreditAccountAllowance()` to grant necessary allowance to a given external contract `toContract`. Our analysis shows that if the current allowance is less than `Constants.MAX_INT_4`, a maximum allowance is then granted.

```

736     function _provideCreditAccountAllowance(
737         address creditAccount,
738         address toContract,
739         address token

```

```

740     ) internal {
741         // Get 10x reserve in allowance
742         if (
743             IERC20(token).allowance(creditAccount, toContract) <
744             Constants.MAX_INT_4
745         ) {
746             ICreditAccount(creditAccount).approveToken(token, toContract); // T:[CM-35]
747         }
748     }

```

Listing 3.3: CreditManager::\_provideCreditAccountAllowance()

It comes to our attention that `Constants.MAX_INT_4` is defined as 25% of `MAX_INT`, which should be `0x3fff`, instead of current `0x4fff`.

**Recommendation** Update the `Constants.MAX_INT_4` constant to reflect the intended amount.

**Status** The issue has been fixed by this commit: `ad31b1d`.

### 3.4 Improved Precision By Multiplication And Division Reordering

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: CreditManager
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

#### Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

In particular, if we examine the `CreditManager::liquidateCreditAccount()` routine, this routine is designed to liquidate an underwater credit account.

```

38     function liquidateCreditAccount(address borrower, address to)
39         external
40         override
41         whenNotPaused // T:[CM-39]

```

```

42     nonReentrant
43     {
44         address creditAccount = getCreditAccountOrRevert(borrower);

46         // send assets to "to" address and compute total value (tv) & threshold weighted
            value (twv)
47         (uint256 totalValue, uint256 twv) = _transferAssetsTo(
48             creditAccount,
49             to
50         );

52         // Checks that current Hf < 1
53         require(
54             twv.div(PercentageMath.PERCENTAGE_FACTOR) <
55             creditFilter.calcCreditAccountAccruedInterest(creditAccount),
56             Errors.CM_CAN_LIQUIDATE_WITH_SUCH_HEALTH_FACTOR
57         ); // T:[CM-13, 16, 17]

59         // Liquidate credit account
60         (, uint256 remainingFunds) = _closeCreditAccountImpl(
61             creditAccount,
62             Constants.OPERATION_LIQUIDATION,
63             totalValue,
64             borrower,
65             msg.sender,
66             to
67         ); // T:[CM-13]

69         emit LiquidateCreditAccount(borrower, msg.sender, remainingFunds); // T:[CM-13]
70     }

```

Listing 3.4: CreditManager::liquidateCreditAccount()

We notice the liquidation validation of the given virtual account (lines 310–315) is performed with mixed multiplication and division. For improved precision, it is better to calculate the equation without involving the division, i.e., `twv < creditFilter.calcCreditAccountAccruedInterest(creditAccount).mul(PercentageMath.PERCENTAGE_FACTOR)`. Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

**Recommendation** Revise the above calculations to better mitigate possible precision loss.

**Status** The issue has been fixed by this commit: `ad31b1d`.



## 3.5 Reentrancy Protection in CreditAccount Lending/Repayment

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: PoolService
- Category: Time and State [7]
- CWE subcategory: CWE-682 [3]

### Description

A common coding best practice in Solidity is the adherence of checks-effects-interactions principle. This principle is effective in mitigating a serious attack vector known as re-entrancy. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the DAO [13] exploit, and the recent Uniswap/Lendf.Me hack [12].

We notice there is an occasion where the checks-effects-interactions principle is violated. Using the PoolService as an example, the lendCreditAccount() function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above re-entrancy.

Apparently, the interaction with the external contract (line 248) starts before effecting the update on internal states (line 254), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching re-entrancy via the same entry function.

```

237     function lendCreditAccount(uint256 borrowedAmount, address creditAccount)
238     external
239     override
240     whenNotPaused // T:[PS-4]
241     {
242         require(
243             creditManagersCanBorrow[msg.sender],
244             Errors.POOL_CREDIT MANAGERS_ONLY
245         ); // T:[PS-12, 13]
246
247         // Transfer funds to credit account
248         IERC20(underlyingToken).safeTransfer(creditAccount, borrowedAmount); // T:[PS
249             -14]
250
251         // Update borrow Rate
252         _updateBorrowRate(); // T:[PS-17]

```

```
252
253     // Increase total borrowed amount
254     totalBorrowed = totalBorrowed.add(borrowedAmount); // T:[PS-16]
255
256     emit Borrow(msg.sender, creditAccount, borrowedAmount); // T:[PS-15]
257 }
```

Listing 3.5: PoolService::lendCreditAccount()

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for re-entrancy. However, it is important to take precautions in making use of `nonReentrant` to block possible re-entrancy. Note the similar issue is also present in another routine `repayCreditAccount()` from the same contract.

**Recommendation** Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible re-entrancy.

**Status** The issue has been fixed by this commit: [cee91b9](#).



## 4 | Conclusion

In this audit, we have analyzed the Gearbox design and implementation. The system presents a unique, robust offering as a decentralized non-custodial protocol that enables decentralized margin trading and leveraged yield farming. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Moreover, we need to emphasize that [Solidity](#)-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.



## References

- [1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. <https://cwe.mitre.org/data/definitions/1126.html>.
- [2] MITRE. CWE-190: Integer Overflow or Wraparound. <https://cwe.mitre.org/data/definitions/190.html>.
- [3] MITRE. CWE-682: Incorrect Calculation. <https://cwe.mitre.org/data/definitions/682.html>.
- [4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [5] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [6] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [7] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. <https://cwe.mitre.org/data/definitions/389.html>.
- [8] MITRE. CWE CATEGORY: Numeric Errors. <https://cwe.mitre.org/data/definitions/189.html>.
- [9] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.

- [10] OWASP. Risk Rating Methodology. [https://www.owasp.org/index.php/OWASP\\_Risk\\_Rating\\_Methodology](https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology).
- [11] PeckShield. PeckShield Inc. <https://www.peckshield.com>.
- [12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.
- [13] David Siegel. Understanding The DAO Attack. <https://www.coindesk.com/understanding-dao-hack-journalists>.

