



SMART CONTRACT AUDIT REPORT

for

THORSwap Aggregators



Prepared By: Patrick Lou

PeckShield
March 24, 2022

Document Properties

Client	THORSwap
Title	Smart Contract Audit Report
Target	THORSwap Aggregators
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 24, 2022	Jing Wang	Final Release
1.0-rc	March 18, 2022	Jing Wang	Release Candidate

Contact

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

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About THORSwap Aggregators	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Possible Fund Stealing From Approving Users	11
3.2	Accommodation of approve() Idiosyncrasies	12
3.3	Trust Issue of Admin Keys	14
3.4	Possible Costly LPs From Improper vTHOR Initialization	15
3.5	Reentrancy Risk in vTHOR::deposit()	16
4	Conclusion	18
	References	19

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the THORSwap Aggregators protocol, 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 THORSwap Aggregators

The THORSwap Aggregators is a decentralized aggregator protocol and contains contracts to integrate a variety of AMMs/DEXs together with THORChain in order to enable swapping from any asset on supported AMMs to any asset on THORChain in either direction. With the THORSwap Aggregators platform, THORChain could support aggregator contracts, which are deployed on Ethereum, on any EVM compatible chain that THORChain supports. The basic information of the THORSwap Aggregators protocol is as follows:

Table 1.1: Basic Information of The THORSwap Aggregators Protocol

Item	Description
Issuer	THORSwap
Website	https://thorswap.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 24, 2022

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

- <https://github.com/thorswap/thorswap-contract-v2.git> (00a69b3)

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

- <https://github.com/thorswap/thorswap-contract-v2.git> (8303249)

1.2 About PeckShield

PeckShield Inc. [13] 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).

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 OWASP Risk Rating Methodology [12]:

- 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 list of check 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

Table 1.3: The Full List of Check Items

Category	Check Item
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

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) [11], 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.

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 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.




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

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	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.
Error Conditions, Return Values, Status Codes	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.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	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.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the THORSwap Aggregators implementation. 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 logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	1	
High	0	
Medium	1	
Low	3	
Informational	0	
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 that 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 critical-severity vulnerability, 1 medium-severity vulnerability, and 3 low-severity vulnerabilities.

Table 2.1: Key THORSwap Aggregators Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Critical	Possible Fund Stealing From Approving Users	Business Logic	Fixed
PVE-002	Low	Accommodation of approve() Idiosyncrasies	Coding Practices	Fixed
PVE-003	Medium	Trust Issue of Admin Keys	Business Logics	Fixed
PVE-004	Low	Possible Costly LPs From Improper vTHOR Initialization	Time and State	Confirmed
PVE-005	Low	Reentrancy Risk in vTHOR::deposit()	Time and State	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Possible Fund Stealing From Approving Users

- ID: PVE-001
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: TSAggregatorGeneric
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

To facilitate the token swap from Ethereum to THORChain, the TSAggregatorGeneric protocol has a helper routine `swapIn()`. This routine is developed to transfer user funds into this contract, next use an external router to swap the funds to ETH, and then use the THORChain router to deposit the ETH with a memo with the information about the THORChain asset and the recipient we want to swap to.

To elaborate, we show below the `swapIn()` routine implementation. This routine allows the user to provide an arbitrary router and `callData` that is used directly in `router.call(data)` (line 29). However, the arbitrary router and `callData` may be exploited to transfer all funds from the users to the hacker.

```
16     function swapIn(  
17         address tcRouter,  
18         address tcVault,  
19         string calldata tcMemo,  
20         address token,  
21         uint amount,  
22         address router,  
23         bytes calldata data,  
24         uint deadline  
25     ) public nonReentrant {  
26         token.safeTransferFrom(msg.sender, address(this), amount);  
27         token.safeApprove(address(router), amount);  
  
29         (bool success,) = router.call(data);  
30         require(success, "failed to swap");
```

```

32     uint256 amountOut = address(this).balance;
33     amountOut = skimFee(amountOut);
34     IThorchainRouter(tcRouter).depositWithExpiry{value: amountOut}(
35         payable(tcVault),
36         address(0), // ETH
37         amountOut,
38         tcMemo,
39         deadline
40     );
41 }

```

Listing 3.1: TSAggregatorGeneric::swapIn()

Specifically, users usually need to give allowance to the TSAggregatorGeneric contract to facilitate the token swap. To save gas fee, sometimes the user may give a maximum allowance as `type(uint256).max` to the TSAggregatorGeneric contract! In this case, if the router value is changed to the token address and the callData is encoded to call function `token.transferFrom(userAddress, hackerAddress, amount)` where the `amount = token.balanceOf(userAddress)`, all token of the user may be withdrawn by the malicious actor.

Recommendation Apply necessary rigorous validity checks on the untrusted user input. Also raise the community awareness of not giving token allowance to the TSAggregatorGeneric contract.

Status The issue has been fixed by adding a separate contract to actually hold the approvals at this commit: 8303249. Note that there is also a need to raise community awareness of not giving token allowance to the TSAggregatorGeneric contract.

3.2 Accommodation of approve() Idiosyncrasies

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., USDT, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need

of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (<https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729>).

```

194  /**
195  * @dev Approve the passed address to spend the specified amount of tokens on behalf
      of msg.sender.
196  * @param _spender The address which will spend the funds.
197  * @param _value The amount of tokens to be spent.
198  */
199  function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

201      // To change the approve amount you first have to reduce the addresses'
202      // allowance to zero by calling 'approve(_spender, 0)' if it is not
203      // already 0 to mitigate the race condition described here:
204      // https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
205      require(!(_value != 0) && (allowed[msg.sender][_spender] != 0));

207      allowed[msg.sender][_spender] = _value;
208      Approval(msg.sender, _spender, _value);
209  }

```

Listing 3.2: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use the `TSAggregatorUniswapV2::swapIn()` routine as an example. This routine is designed to swap tokens from Ethereum to THORChain. To accommodate the specific idiosyncrasy, for each `safeApprove()` (line 33), there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

```

23  function swapIn(
24      address tcRouter,
25      address tcVault,
26      string calldata tcMemo,
27      address token,
28      uint amount,
29      uint amountOutMin,
30      uint deadline
31  ) public nonReentrant {
32      token.safeTransferFrom(msg.sender, address(this), amount);
33      token.safeApprove(address(swapRouter), amount);

35      address[] memory path = new address[](2);
36      path[0] = token;
37      path[1] = weth;
38      swapRouter.swapExactTokensForETH(
39          amount,
40          amountOutMin,
41          path,
42          address(this),
43          deadline

```

```

44     );
45
46     uint amountOut = skimFee(address(this).balance);
47     IThorchainRouter(tcRouter).depositWithExpiry{value: amountOut}(
48         payable(tcVault),
49         address(0), // ETH
50         amountOut,
51         tcMemo,
52         deadline
53     );
54 }

```

Listing 3.3: TSAggregatorUniswapV2::swapIn()

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related approve().

Status This issue has been fixed in this commit: cbb6afc.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: PledgeFactory
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the THORSwap Aggregators protocol, there is a privileged owner account that plays a critical role in governing and regulating the system-wide operations (e.g., privileged account setting and fee adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```

23     function setFee(uint256 _fee, address _feeRecipient) public isOwner {
24         fee = _fee;
25         feeRecipient = _feeRecipient;
26         emit FeeSet(_fee, _feeRecipient);
27     }

```

Listing 3.4: TSAggregator::setFee()

If the privileged owner account is a plain EOA account, this may be worrisome and pose counterparty risk to the exchange users. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key

concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation. Moreover, it should be noted if current contracts are to be deployed behind a proxy, there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been fixed by this commit: 381a256.

3.4 Possible Costly LPs From Improper vTHOR Initialization

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: vTHOR
- Category: Time and State [7]
- CWE subcategory: CWE-362 [3]

Description

The vTHOR contract allows the users to stake their funds to receive vTHOR token as shares. The staking users will get their pro-rata share based on their staked amount. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the share extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `deposit()` routine. This `deposit()` routine is used for participating users to deposit the supported asset (e.g., token) and get respective shares in return. The issue occurs when the vTHOR contract is being initialized under the assumption that the current contract is empty.

```

17     function deposit(uint256 amount) public {
18         uint256 totalShares = totalSupply;
19         uint256 totalDeposits = IERC20(token).balanceOf(address(this));
20         token.safeTransferFrom(msg.sender, address(this), amount);
21         if (totalShares == 0 || totalDeposits == 0) {
22             _mint(msg.sender, amount);
23         } else {
24             _mint(msg.sender, (amount * totalShares) / totalDeposits);
25         }
26     }

```

Listing 3.5: `valut::_issueSharesForAmount()`

Specifically, when the contract is being initialized, the share value directly takes the value of `amount` (line 22), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `shares = 1 WEI`. With that, the actor can further deposit a huge amount of `token` with the goal of making the share extremely expensive.

An extremely expensive share can be very inconvenient to use as a small number of 1 `Wei` may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when `totalSupply` is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to `address(0)`). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current execution logic of share calculation to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure guarded launch that safeguards the first deposit to avoid being manipulated.

Status The issue has been confirmed by the team and they clarify there would be a guarded launch that safeguards the first deposit to avoid being manipulated.

3.5 Reentrancy Risk in `vTHOR::deposit()`

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `vTHOR`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [4]

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` [15] exploit, and the recent `Uniswap/Lendf.Me` hack [14].

We notice there is an occasion where the checks-effects-interactions principle is violated. Using the vTHOR as an example, the `deposit()` 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 20) starts before effecting the update on the internal state (lines 21-25), 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.

```
17     function deposit(uint256 amount) public {
18         uint256 totalShares = totalSupply;
19         uint256 totalDeposits = IERC20(token).balanceOf(address(this));
20         token.safeTransferFrom(msg.sender, address(this), amount);
21         if (totalShares == 0 || totalDeposits == 0) {
22             _mint(msg.sender, amount);
23         } else {
24             _mint(msg.sender, (amount * totalShares) / totalDeposits);
25         }
26     }
```

Listing 3.6: vTHOR::deposit()

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

Status The issue has been fixed by this commit: [34ef363](#).

4 | Conclusion

In this audit, we have analyzed the THORSwap Aggregators protocol design and implementation. The THORSwap Aggregators protocol provides a decentralized aggregator protocol and contains contracts that integrate AMMs/DEXs together with THORChain to enable swapping from any asset on supported AMMs to any asset on THORChain in either direction. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that 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-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [3] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). <https://cwe.mitre.org/data/definitions/362.html>.
- [4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. <https://cwe.mitre.org/data/definitions/663.html>.
- [5] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [6] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [7] MITRE. CWE CATEGORY: 7PK - Time and State. <https://cwe.mitre.org/data/definitions/361.html>.
- [8] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [9] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.

- [10] MITRE. CWE CATEGORY: Concurrency. <https://cwe.mitre.org/data/definitions/557.html>.
- [11] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [12] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [13] PeckShield. PeckShield Inc. <https://www.peckshield.com>.
- [14] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.
- [15] David Siegel. Understanding The DAO Attack. <https://www.coindesk.com/understanding-dao-hack-journalists>.

