SMART CONTRACT AUDIT REPORT

for

EllipsisV2 Staking

Prepared By: Patrick Lou

PeckShield

March 26, 2022

Document Properties

Client	Ellipsis Finance
Title	Smart Contract Audit Report
Target	EllipsisV2 Staking
Version	1.0-rc
Author	Xuxian Jiang
Auditors	Luck Hu, Jing Wang, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0-rc	March 26, 2022	Xuxian Jiang	Release Candidate #1
1.0-rc	March 26, 2022	Xuxian Jiang	Release Candidate

Contact

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

Name	Patrick Lou
Phone	+86 156 0639 2692
Email	contact@peckshield.com

Contents

1 Introduction **4**
- 1.1 About Ellipsis 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 Proper totalMigrated Booking in EllipsisToken2 11
- 3.2 Proper Pending Reward Calculation in LPStaking::deposit()/withdraw() 12
- 3.3 Proper Claimable Reward Calculation on LPStaking::claimableReward() 13
- 3.4 Proper Pool adjustedSupply Adjustment on emergencyWithdraw() 14
- 3.5 Suggested Adherence Of Checks-Effects-Interactions Pattern 15
- 3.6 Trust Issue Of Admin Keys 17

4 Conclusion **18**

References **19**

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the staking and emissions contracts for the EllipsisV2 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 contract can be further improved due to the presence of several issues related to business logic, security or performance. This document outlines our audit results.

1.1 About Ellipsis

The Ellipsis Finance is officially launched in March 2021 as an authorized fork of Curve Finance and shares the core values as a trustless and decentralized architecture with zero deposit or withdrawal fees, no lock ups on liquidity, and extremely efficient stable coin exchanges. The audited staking and emissions contracts incentivize protocol users to stake supported tokens to receive a portion of the rewards (including trade fees generated when users perform exchanges). The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of the EllipsisV2 protocol

Item	Description
Name	Ellipsis Finance
Website	https://ellipsis.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 26, 2022

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

- https://github.com/mashup-artist/ellipsis-v2.git (9ef052)

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

- https://github.com/mashup-artist/ellipsis-v2.git (1dce372)

1.2 About PeckShield

PeckShield Inc. [9] 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 \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- 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.

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

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 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) [7], 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 `staking` and `emissions` contracts for the `EllipsisV2` 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 logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	1	■
Low	4	■ ■ ■ ■
Informational	0	
Total	6	

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 some 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, this smart contract is well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 1 medium-severity vulnerability, and 4 low-severity vulnerabilities.

Table 2.1: Key EllipsisV2 Staking Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Proper totalMigrated Booking in Ellipsis-Token2	Business Logic	Fixed
PVE-002	Low	Proper Pending Reward Calculation in LP-Staking::deposit()/withdraw()	Business Logic	Fixed
PVE-003	Low	Proper Claimable Reward Calculation on LPStaking::claimableReward()	Business Logic	Fixed
PVE-004	High	Proper Pool adjustedSupply Adjustment on emergencyWithdraw()	Business Logic	Fixed
PVE-005	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed
PVE-006	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

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 Proper totalMigrated Booking in EllipsisToken2

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `EllipsisToken2`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In the `EllipsisV2` protocol, there is a `EllipsisToken2` contract that is designed to migrate from the existing protocol token. This token contract is fully ERC20-compliant and has additional logic for migration. While analyzing the current migration logic, we notice the current implementation can be improved.

To elaborate, we show below the related `migrate()` function, which has a rather straightforward logic in swapping the old protocol token with the new one based the specified migration ratio. However, it comes to our attention that this contract defines a state `totalMigrated` to keep track of the total migrated amount and this state is not properly updated in `migrate()`. To keep track of the migration status, there is a need to properly update the storage state!

```
    function migrate(uint256 _amount) external returns (bool) {
        oldToken.transferFrom(msg.sender, address(0), _amount);
        uint256 newAmount = _amount * migrationRatio;
        balanceOf[msg.sender] += newAmount;
        emit Transfer(address(0), msg.sender, newAmount);
        emit TokensMigrated(msg.sender, _amount, newAmount);
    }
```

Listing 3.1: `EllipsisToken2::migrate()`

Recommendation Revise the above `migrate()` routine properly update the storage state.

Result The issue has been fixed by this commit: `ad2ddcf`.

3.2 Proper Pending Reward Calculation in LPStaking::deposit()/withdraw()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LPStaking`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The `EllipsisV2` protocol has a core contract `LPStaking` that shares a `MasterChef`-like approach to incentivize protocol users to stake the supported assets. Naturally, the contract requires proper accounting of pending rewards for staking users. While reviewing the reward calculation logic, we notice the current implementation is flawed and needs to be corrected.

In particular, we show below the related code snippet from the `deposit()` function. The logic is rather straightforward in validating the staking user, next computing the pending reward, then transferring the staked amount in, and finally bookkeeping the user stakes. It comes to our attention that when the pending reward is computed and claimed, the reward amount is computed as `pending + user.claimable` (line 272). However, the intermediate `pending` state already includes the `user.claimable`. Note the `withdraw()` function shares the same issue.

```
260     function deposit(address _receiver, address _token, uint256 _amount, bool _claimRewards) external {
261         require(_amount > 0, "Cannot deposit zero");
262         if (msg.sender != _receiver) {
263             require(!blockThirdPartyActions[_receiver], "Cannot deposit on behalf of this account");
264         }
265         uint256 accRewardPerShare = _updatePool(_token);
266         UserInfo storage user = userInfo[_token][_receiver];
267         if (user.adjustedAmount > 0) {
268             uint256 pending = user.adjustedAmount * accRewardPerShare / 1e12 - user.rewardDebt;
269             if (_claimRewards) {
270                 pending += user.claimable;
271                 user.claimable = 0;
272                 _mintRewards(_receiver, pending + user.claimable);
273             } else if (pending > 0) {
274                 user.claimable += pending;
275             }
276         }
277         IERC20(_token).safeTransferFrom(
278             address(msg.sender),
279             address(this),
```

```
            _amount
        );
        uint256 depositAmount = user.depositAmount + _amount;
        user.depositAmount = depositAmount;
        _updateLiquidityLimits(_receiver, _token, depositAmount, accRewardPerShare);
        emit Deposit(msg.sender, _receiver, _token, _amount);
    }
```

Listing 3.2: `LPStaking::deposit()`

Recommendation Adjust the above-mentioned functions to properly compute the pending rewards for claims.

Result The issue has been fixed by this commit: `2ddc75b`.

3.3 Proper Claimable Reward Calculation on LPStaking::claimableReward()

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `LPStaking`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, the `EllipsisV2` protocol has a core contract `LPStaking` that shares a `MasterChef` -like approach to incentivize protocol users to stake the suggested assets. While reviewing the `claimableReward()` function in this contract, we notice this function makes use of a wrong state to compute the claimable reward.

To elaborate, we show below the full implementation of this function. Apparently, the current implementation uses the `user.depositAmount` (line 175) to compute the deserved reward. It comes to our attention that the deserved reward needs to correctly apply the boost with the `user.adjustedAmount` !

```
163    function claimableReward(address _user, address[] calldata _tokens)
164        external
165        view
166        returns (uint256[] memory)
167    {
168        uint256[] memory claimable = new uint256[](_tokens.length);
169        for (uint256 i = 0; i < _tokens.length; i++) {
170            address token = _tokens[i];
171            PoolInfo storage pool = poolInfo[token];
```

```
        UserInfo storage user = userInfo[token][_user];
        (uint256 accRewardPerShare,) = _getRewardData(token);
        accRewardPerShare += pool.accRewardPerShare;
        claimable[i] = user.claimable + user.depositAmount * accRewardPerShare / 1
            e12 - user.rewardDebt;
    }
    return claimable;
}
```

Listing 3.3: `LPStaking::claimableReward()`

Recommendation Properly revise the `claimableReward()` function to compute the right reward amount.

Result The issue has been fixed by this commit: `02d7de5`.

3.4 Proper Pool adjustedSupply Adjustment on emergencyWithdraw()

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `LPStaking`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

Based on the `SushiSwap`'s `MasterChef`, the `LPStaking` contract implements the incentive mechanisms that reward the staking of supported assets with certain reward tokens. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. In the meantime, it also provides an `emergencyWithdraw()` function that allows the staking user to exit the current pool without taking the accumulated rewards.

In the process of analyzing the `emergencyWithdraw()` function, we notice the current implementation is flawed in not timely updating the pool-wide state, i.e., `poolInfo[_token].adjustedSupply`. If the pool-wide `adjustedSupply` state is not udpated, it is possible for current staking users to receive less staking rewards if a malicious user keeps iterating the `deposit()-and-emergencyWithdraw()` operations.

```
function emergencyWithdraw(address _token) external {
    UserInfo storage user = userInfo[_token][msg.sender];

    uint256 amount = user.depositAmount;
    delete userInfo[_token][msg.sender];
    IERC20(_token).safeTransfer(address(msg.sender), amount);
    emit EmergencyWithdraw(_token, msg.sender, amount);
```

```
337     }
```

Listing 3.4: LPStaking::emergencyWithdraw()

Recommendation Properly update the pool-wide `adjustedSupply` state in the above `emergencyWithdraw()` routine.

Result The issue has been fixed by this commit: `2d075a7`.

3.5 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LPStaking`
- Category: Time and State [6]
- CWE subcategory: CWE-663 [2]

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` [11] exploit, and the recent `Uniswap/Lendf.Me` hack [10].

We notice there are a number of occasion where the `checks-effects-interactions` principle is violated. Using the `LPStaking` 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 277) starts before effecting the update on internal states (lines 283−284), 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.

```
260     function deposit(address _receiver, address _token, uint256 _amount, bool
            _claimRewards) external {
261         require(_amount > 0, "Cannot deposit zero");
262         if (msg.sender != _receiver) {
```

```
            require(!blockThirdPartyActions[_receiver], "Cannot deposit on behalf of this account");
        }
        uint256 accRewardPerShare = _updatePool(_token);
        UserInfo storage user = userInfo[_token][_receiver];
        if (user.adjustedAmount > 0) {
            uint256 pending = user.adjustedAmount * accRewardPerShare / 1e12 - user.rewardDebt;
            if (_claimRewards) {
                pending += user.claimable;
                user.claimable = 0;
                _mintRewards(_receiver, pending + user.claimable);
            } else if (pending > 0) {
                user.claimable += pending;
            }
        }
        IERC20(_token).safeTransferFrom(
            address(msg.sender),
            address(this),
            _amount
        );
        uint256 depositAmount = user.depositAmount + _amount;
        user.depositAmount = depositAmount;
        _updateLiquidityLimits(_receiver, _token, depositAmount, accRewardPerShare);
        emit Deposit(msg.sender, _receiver, _token, _amount);
    }
```

Listing 3.5: LPStaking::deposit()

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`. The adherence of `checks-effects-interactions` best practice in these routines is strongly recommended.

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

Status The issue has been fixed by this commit: `a99aa4e`.

3.6 Trust Issue Of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the audited staking and emissions contracts, there exist certain privileged accounts that play critical roles in governing and regulating the protocol-wide operations. In the following, we show the privileged owner and the related privileged accesses in current contracts.

```
    function setTokenApprovalQuorum(uint256 _quorumPct) external onlyOwner {
        emit ApprovalQuorumSet(msg.sender, tokenApprovalQuorumPct, _quorumPct);
        tokenApprovalQuorumPct = _quorumPct;
    }


    /**
        @dev Modify the approval for a token to receive incentives.
        This can only be called on tokens that were already voted in, it cannot
        be used to bypass the voting process. It is intended to block emissions in
        case of an exploit or act of maliciousness from a token within an approved pool.
     */

    function setTokenApproval(address _token, bool _isApproved) external onlyOwner {
        if (!isApproved[_token]) {
            (,,uint256 lastRewardTime,) = lpStaking.poolInfo(_token);
            require(lastRewardTime != 0, "Token must be voted in");
        }
        isApproved[_token] = _isApproved;
    }
```

Listing 3.6: Example Privileged Operations in IncentiveVoting

There are also some other privileged functions not listed above. And we understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the list of extra privileges granted to owner explicit to the protocol users.

Result The issue has been mitigated by this commit: 51a171a.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `staking` and `emissions` contracts for the `EllipsisV2` protocol, which is officially launched in March 2021 as an authorized fork of `Curve Finance` and shares the core values as a trustless and decentralized architecture with zero deposit or withdrawal fees. The audited `staking and emissions` contracts incentivize protocol users to stake with supported tokens to receive a portion of the rewards (including trade fees generated when users perform exchanges). 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[6] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.

[10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[11] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.