SMART CONTRACT AUDIT REPORT

for

DEGO FINANCE

Prepared By: Shuxiao Wang

PeckShield
May 20, 2021

Document Properties

Client	Dego Finance
Title	Smart Contract Audit Report
Target	Dego-Ino
Version	1.0
Author	Xuxian Jiang
Auditors	Yiqun Chen, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 20, 2021	Xuxian Jiang	Final Release
1.0-rc1	May 18, 2021	Xuxian Jiang	Release Candidate #1
0.2	May 14, 2021	Xuxian Jiang	Add More Findings #1
0.1	May 5, 2021	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the `Dego Finance`'s **dego-ino** 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 Dego Finance

`Dego Finance` is a `NFT+DeFi` protocol and infrastructure with two main functions: First, it acts as an independent and open `NFT` ecosystem with services covering the full `NFT` lifecycle, enabling anyone to issue `NFTs`, participate in auctions, and trade `NFTs`. Second, it is also building an `NFT` protocol to provide a cross-chain layer 2 infrastructure. By building on multiple blockchains such as `Binance Smart Chain`, `Ethereum`, and `Polkadot`, `Dego Finance` enables blockchain projects to acquire users, distribute tokens and develop more diverse `NFT` applications. The audited `dego-ino` protocol implements the much-needed initial NFC offering platform.

The basic information of Dego-Ino is as follows:

Table 1.1: Basic Information of Dego-Ino

Item	Description
Issuer	Dego Finance
Website	https://dego.finance
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 20, 2021

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

- https://github.com/dego-labs/ino.git (30c1d38)

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

- https://github.com/dego-labs/ino.git (TBD)

1.2 About PeckShield

PeckShield Inc. [12] 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*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [11]:

- 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) [10], 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
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 Logic	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.

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 Dego-Ino 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	2	■ ■
Low	2	■ ■
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 2 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Dego-Ino Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved refund() Logic	Business Logics	Fixed
PVE-002	Medium	Reentrancy Risk in raise()/split()	Time and State	Fixed
PVE-003	Low	Improved Sanity Checks Of System/Function Parameters	Coding Practices	Fixed
PVE-004	Informational	Avoided Storage Use For Constant State Variables	Coding Practices	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Improved refund() Logic

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `RaisedPool`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `dego-ino` protocol has a central contract `RaisedPool` that provides a number of key functions, e.g., `raise()`, `vest()`, `split()`, `withdraw()`, and `refund()`, to facilitate various NFC operations. In the following, we examine one specific `refund()` function.

To elaborate, we show below the `refund()` logic. It basically returns remaining target tokens from the pool back to its creator. We note that the `refund()` logic has two conditions `require(block.timestamp > _endRaisedTime)` (line 413) and `require(_remainAmount > 0)` (line 414). Note these two conditions are a subset of `canRefund()`, which additionally requires `require(!hasRefund)`. With that, it is suggested to simplify the logic of `refund()` by enforcing `require(canRefund())`. Moreover, the `_remainAmount` state variable needs to be zeroed out before returning from `refund()`.

```
409    /*
410     * get target token from pool
411     */
412    function refund() public onlyOwner{
413        require(block.timestamp > _endRaisedTime, "wait to end");
414        require(_remainAmount > 0, "sale out");
415        TransferHelper.safeTransfer(_targetToken, msg.sender, _remainAmount);
416
417        hasRefund = true;
418        emit Refund(_targetToken, _remainAmount);
419    }
```

Listing 3.1: RaisedPool::refund()

Recommendation Improve the `refund()` by enforcing `require(canRefund())` and zeroing out `_remainAmount`. An example revision is show below:

```
    /*
     * get target token from pool
     */
    function refund() public onlyOwner{
        require(canRefund(), "!refund");
        TransferHelper.safeTransfer(_targetToken, msg.sender, _remainAmount);
        _remainAmount = 0;
        hasRefund = true;
        emit Refund(_targetToken, _remainAmount);
    }
```

Listing 3.2: RaisedPool::refund()

Status The issue has been fixed in this commit: `fe16a77`.

3.2 Reentrancy Risk in raise()/split()

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `RaisedPool`
- Category: Time and State [9]
- 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` [14] exploit, and the recent `Uniswap/Lendf.Me` hack [13].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `RaisedPool` as an example, the `raise()` 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 171) starts before effecting the update on internal states (lines 172−174), 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 very same `raise()` function.

```
    function raise(uint amount)  public {
        uint beforeBalance = IERC20(_costToken).balanceOf(address(this));
        TransferHelper.safeTransferFrom(_costToken, msg.sender, address(this), amount);
        uint endBalance = IERC20(_costToken).balanceOf(address(this));

        uint refund = doRaise(msg.sender, endBalance.sub(beforeBalance));
        if(refund > 0){
            require( refund <= IERC20(_costToken).balanceOf(address(this)), "not enough
                refund token");
            TransferHelper.safeTransfer(_costToken, msg.sender, refund);
        }
    }
```

Listing 3.3: RaisedPool::raise()

It should be mentioned that the internal `doRaise()` handler implements the needed non-reentrancy protection. However, this protection should be moved to cover its caller, i.e., `raise()`. Another similar violation can be found in the `split()` routine within the same contract.

In the meantime, we should mention that the supported tokens in the protocol may need to be whitelisted to avoid unwanted risks of reentrancy. Current standard ERC20-compliant tokens without any extra functionality are not vulnerable or exploitable for `re-entrancy`.

Recommendation Apply necessary reentrancy prevention by making use of the common `nonReentrant` modifier.

Status The issue has been fixed in this commit: `fe16a77`.

3.3 Improved Sanity Checks Of System/Function Parameters

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [2]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `dego-ino` protocol is no exception. In the following, we examine a number of routines that can benefit from improved validation.

The first example is the `RaisedPool::initialize()` function. As the name indicates, this routine sets up necessary parameters for the associated pool. One specific set of arguments is `times` to initialize `startRaisedTime` and `endRaisedTime`. It is helpful to enforce the following condition, i.e., `require(startRaisedTime < endRaisedTime)`

```
    function initialize(address controller, address[2] memory tokens, uint[2] memory priceRatioes,
        uint saleAmount, uint vsType,  uint[2] memory times, uint[4] memory args, uint maxCeiling,
        bool isPrivate,uint index, address factory)  public
    {
        require(!initialized_, "initialize: Already initialized!");

        _controller = controller;
        _targetToken = tokens[0];
        _poolIndex = index;
        _costToken = tokens[1];

        _targetPriceRatio = priceRatioes[0];
        _costPriceRatio = priceRatioes[1];

        _totalSaleAmount = saleAmount;
        _remainAmount = _totalSaleAmount;
        _startRaisedTime = times[0];
        _endRaisedTime = times[1];

        if(vsType == 0){
            //do nothing
        }else if(vsType == 1){
            require(args[0] > _endRaisedTime, "vest after end");
            require(args[1] > args[0], "end larger then start");
        }else if(vsType == 2){
            require(args[0] > _endRaisedTime, "vest after end");
            require(args[1] > 0, "not zero");
            require(args[2] < BASE_PERCENT_10K, "too big");
            require(args[3] == 0, "phase start from zero");
        }else{
            require(false, "error vsType");
        }

        _isPrivate = isPrivate;

        _factory = factory;
        _vestType = vsType;
        _args = args;
        _maxCeiling = maxCeiling;

        initialized_ = true;
    }
```

Listing 3.4: RaisedPool:: initialize ()

The second example is about the `RaisedPool::withdraw()` logic. It currently validates `require(block.timestamp > _endRaisedTime)`, which should be replaced with `require(canWithdraw(), "wait to end")`.

```
    function withdraw()  public  onlyOwner{
```

```
        require(block.timestamp > _endRaisedTime, "wait to end");

        uint balance;
        if(_costToken == address(0x0)){
            balance = address(this).balance;
        }else{
            balance = IERC20(_costToken).balanceOf(address(this));
        }

        address teamWallet = IRaisedController(_controller).getTeamWallet();
        uint withdrawFee = IRaisedController(_controller).getWithdrawFee();
        uint feebase = IRaisedController(_controller).getFeeBase();

        uint raisedFund = (feebase - withdrawFee) * balance /  feebase;

        if( _costToken == address(0x0) ){
            TransferHelper.safeTransferETH(msg.sender, raisedFund);
            TransferHelper.safeTransferETH(teamWallet, balance.sub(raisedFund));
        }else{
            TransferHelper.safeTransfer(_costToken, msg.sender, raisedFund);
            TransferHelper.safeTransfer(_costToken, teamWallet, balance.sub(raisedFund))
                ;
        }
        hasWidthdraw = true;
        emit Withdraw(_targetToken, _costToken, raisedFund, balance.sub(raisedFund),
            teamWallet);
    }
```

Listing 3.5: RaisedPool::withdraw()

The last example is about the `INOFactory::forceChangeId()` routine. It is helpful to ensure `require(value > _inoId, "wrong value")` before the system-wide parameter `_inoId` is updated.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. Also, consider emitting related events for external monitoring and analytics tools.

Status The issue has been fixed in this commit: `fe16a77`.

3.4 Avoided Storage Use For Constant State Variables

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `RaisedController`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1099 [1]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

In the following, we show the key state variables defined in `RaisedController`. If there is no need to dynamically update these key state variables, they can be declared as `immutable` for gas efficiency.

```
contract RaisedController is IRaisedController, Governance {
    using Address for address;
    using SafeMath for uint;
    // config
    address public _poolFactory;
    address public _inoFactory;

    uint public _withdrawFee = 100;
    uint constant public FEE_BASE = 10000;
    ...
}
```

Listing 3.6: RaisedController.sol

Recommendation Revisit the state variable definition and make extensive use of `immutable` states.

Status The issue has been fixed in this commit: `fe16a77`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `RaisedController`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the `dego-ino` protocol, the governance account plays a critical role in governing and regulating the system-wide operations (e.g., add/remove minters and set system parameters). It also has the privilege to regulate or govern the flow of assets among the involved components in the protocol.

We emphasize that current privilege assignment is necessary and required for proper protocol operation. However, it is worrisome if the `governance` is not governed by a `DAO`-like structure. The discussion with the team has confirmed that the `governance` will be managed by a multi-sig account. Note that a compromised `governance` account is capable of modifying current protocol configuaration with adverse consequences on user funds. In the following, we show representative privileged operations in the `dego-ino` protocol.

```
    function setSplitCostDego(uint newValue) public
        onlyGovernance
    {

        require(_splitCostDego != newValue, "invalid args");
        emit ChangeSplitCostDego(_splitCostDego, newValue);
        _splitCostDego = newValue;
    }


    function getDegoToken() external view override returns (address){
        return _degoToken;
    }

    function setDegoToken(address token) public
        onlyGovernance
    {

        require(_degoToken != token, "invalid args");
        emit ChangeDego(_degoToken, token);
        _degoToken = token;
    }


    function getWithdrawFee() external view override  returns (uint){
        return _withdrawFee;
```

```
    }

    function getFeeBase() external view override  returns (uint){
        return FEE_BASE;
    }

    function setWithdrawFee(uint withdrawFee) public
        onlyGovernance
    {

        require(withdrawFee != _withdrawFee && withdrawFee < FEE_BASE,"invalid args");
        emit ChangeWithdrawFee(_withdrawFee, withdrawFee);
        _withdrawFee = withdrawFee;
    }
```

Listing 3.7: Various Getters/Setters in RaisedController

Recommendation Promptly transfer the `governance` privilege to the intended `DAO`-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed and partially mitigated with a multi-sig account to regulate the governance/controller privileges.

4 Conclusion

In this audit, we have analyzed the Dego-Ino design and implementation. The system presents a much-needed `initial NFC offering` platform. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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

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

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

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

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

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

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