SMART CONTRACT AUDIT REPORT

for

Merlin Protocol

Prepared By: Xiaomi Huang

PeckShield
February 20, 2023

Document Properties

Client	Merlin Protocol
Title	Smart Contract Audit Report
Target	HashNFT
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 20, 2023	Luck Hu	Final Release
1.0-rc	December 12, 2022	Luck Hu	Release Candidate

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the HashNFT 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 is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About HashNFT

Merlin is an infrastructure protocol for DeFi mining hash power and its derivatives. It digitizes real-world cryptocurrency hash power assets and then introduces them into the DeFi ecosystem through a hash power oracle and a decentralized settlement system. The audited HashNFT is the first Real-World Assets from Merlin to manage the investment of hash power assets. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of HashNFT

Item	Description
Name	Merlin Protocol
Website	https://merlinprotocol.org/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 20, 2023

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

- https://github.com/merlinprotocol/HashNFT.git (f6d517d)

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

- https://github.com/merlinprotocol/HashNFT.git (672d15e)

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 the 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.

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

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

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. 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 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 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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `HashNFT` smart contracts. 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	0	
Total	4	

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 and 2 low-severity vulnerabilities.

Table 2.1: Key HashNFT Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited Logic in BitcoinEarningsOracle	Business Logic	Fixed
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Fixed
PVE-003	Low	Suggested Adherence of Checks-Effects-Interaction Pattern	Time and State	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

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 Revisited Logic in BitcoinEarningsOracle

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

Description

In the `RiskControl` contract, the issuer delivers mining earnings to users to unlock the funds. The mining earnings of each day is obtained from the `BitcoinEarningsOracle` contract. While examining the logic to update the daily earnings in the `BitcoinEarningsOracle` contract, we notice the meaning of the day index is inconsistent with what it is expected in the `RiskControl` contract.

To elaborate, we show below the code snippets from the `BitcoinEarningsOracle/RiskControl` contracts. As the name indicates, the `trackDailyEarnings()` routine is used for the `TRACK_ROLE` to update the daily earnings. Note the `day` here (line 69) indicates the days since the epoch time (line 62). However, in the `RiskControl::deliver()` routine where the daily earnings are queried, the provided `desDay` (line 149) means the days since the end of the collection period (lines 135−136). The inconsistence between the meanings of the day index makes the returned daily earnings unexpected.

```
61     function _today() private view returns (uint256) {
62         return block.timestamp / 1 days;
63     }
64
65     function trackDailyEarnings(
66         uint256[] memory earnings_,
67         uint256[] memory hashrates_
68     ) public onlyRole(TRACK_ROLE) {
69         uint256 day = _today();
70         _makerDailyEarnings(day, earnings_, hashrates_);
71         emit TrackDailyEarnings(day, _dailyEarnings[day]);
```

```
    }
```

Listing 3.1: BitcoinEarningsOracle :: trackDailyEarnings ()

```
130     function dayNow() public view override returns (uint256) {
131         require(
132             _currentStage() > Stage.CollectionPeriod,
133             "RiskControl: error stage"
134         );
135         uint256 duration = block.timestamp -
136             (startTime + collectionPeriodDuration);
137         return duration / 1 days;
138     }
139 
140     function deliver() public {
141         require(
142             deliverAllowed() && dayNow() > 0,
143             "RiskControl: deliver not allowed"
144         );
145         require(
146             initialPayment != 0  _currentStage() == Stage.ObservationPeriod,
147             "RiskControl: must generate initial payment"
148         );
149         uint256 desDay = dayNow() - 1;
150         require(deliverRecords[desDay] == 0, "RiskControl: already deliver");
151         uint256 earnings = earningsOracle.getRound(desDay);
152         if (earnings == 0) {
153             (, uint256 lastEarnings) = earningsOracle.lastRound();
154             earnings = lastEarnings;
155         }
156         ...
157     }
```

Listing 3.2: RiskControl :: deliver ()

What is more, in the `BitcoinEarningsOracle::getRound()` routine, which is used in the `RiskControl::deliver()` routine to get the earnings for the input day, it reverts the transaction (line 48) if the daily earnings are not set. That is to say it is impossible for the routine to return 0. However, in the `RiskControl::deliver()` routine, it has a special handling when the returned earning is 0 (line 152) which could never be reached. Our analysis shows that the `BitcoinEarningsOracle::getRound()` routine shall return 0 if the daily earnings are not set.

```
35      function getRound(uint256 day)
36          public
37          view
38          virtual
39          override
40          returns (uint256)
41      {
42          for (uint256 i = 0;i<7;++i) {
43              uint256 earning = _dailyEarnings[day-i];
```

```
            if (earning != 0) {
                return earning;
            }
        }
        revert("!round");
    }
```

Listing 3.3: BitcoinEarningsOracle :: getRound()

Recommendation Revisit the above mentioned logic in the `BitcoinEarningsOracle` contact to make it consistent with the using from the `RiskControl` contact.

Status This issue has been fixed in these commits: `612b423` and `cfda163`.

3.2 Accommodation of Non-ERC20-Compliant Tokens

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

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 the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
```

```
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.4: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `deliver()` routine in the `RiskControl` contract. If the `ZRX` token is supported as `funds`, the unsafe version of `funds.transfer(issuer, deliverReleaseAmount)` (line 150) may return `false` while not revert. Without a validation on the return value, the transaction can proceed even when the transfer fails.

The same issue is applicable to the `HashNFT::payForMint()/RiskControl::deliver()` routines, where the call to `transferFrom()` is suggested to use the safe version, i.e., `safeTransferFrom()`.

```
140    function deliver() public {
141        require(...);
142        uint256 desDay = dayNow() - 1;
143        require(deliverRecords[desDay] == 0, "RiskControl: already deliver");
144        uint256 earnings = earningsOracle.getRound(desDay);
145        if (earnings == 0) {...}
146        uint256 amount = earnings.mul(hashnft.sold());
147        rewards.transferFrom(issuer, hashnft.dispatcher(), amount);
148        deliverRecords[desDay] = amount;
149        if (deliverReleaseAmount > 0) {
150            funds.transfer(issuer, deliverReleaseAmount);
151        }
152        emit Deliver(address(issuer), hashnft.dispatcher(), amount);
153    }
```

Listing 3.5: `RiskControl::deliver()`

Recommendation Accommodate the above-mentioned idiosyncrasies with safe-version implementation of ERC20-related `transfer()/transferFrom()`.

Status This issue has been fixed in this commit: `612b423`.

3.3 Suggested Adherence of Checks-Effects-Interactions Pattern

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- 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, as well as the `Uniswap/Lendf.Me` hack [10].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `RiskControl` as an example, the `claimTax()` 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 264) starts before effecting the update on internal states (line 265), 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 `claimTax()` function.

```
256     function claimTax(address to)
257         public
258         onlyRole(ADMIN_ROLE)
259         afterStage(Stage.ObservationPeriod)
260     {
261         uint256 tax = hashnft.sold().mul(cost).mul(taxPercent).div(10000);
262         require(taxClaimed < tax, "RiskControl: already tax claimed");
263         uint256 amount = tax.sub(taxClaimed);
264         funds.safeTransfer(to, amount);
265         taxClaimed = tax;
266         emit ClaimTax(to, amount);
267     }
```

Listing 3.6: `RiskControl::claimTax()`

Note the same issue can be found in the following routines: `RiskControl::claimOption()/HashNFT ::payForMint()`, etc.

Recommendation Apply necessary re-entrancy prevention by following the checks-effects-interactions best practice. An example revision on the BaseShareField::_mint() routine is shown below:

```
function claimTax(address to)
    public
    onlyRole(ADMIN_ROLE)
    afterStage(Stage.ObservationPeriod)
{
    uint256 tax = hashnft.sold().mul(cost).mul(taxPercent).div(10000);
    require(taxClaimed < tax, "RiskControl: already tax claimed");
    uint256 amount = tax.sub(taxClaimed);
    taxClaimed = tax;
    funds.safeTransfer(to, amount);
    emit ClaimTax(to, amount);
}
```

Listing 3.7: RiskControl::claimTax

Status This issue in the RiskControl contract has been fixed in this commit: f52732d, and this issue in the HashNFT::payForMint() routine has been mitigated as the team confirmed that the funds token is USDT which has no hook to reenter the protocol.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the HashNFT protocol, there are certain privileged accounts, e.g., owner/ADMIN_ROLE/TRACK_ROLE, that play critical roles in governing and regulating the system-wide operations (e.g., set daily mining earnings). Our analysis shows that these privileged accounts needs to be scrutinized. In the following, we use the BitcoinEarningsOracle contract as an example and show the representative functions potentially affected by the privileges of the DEFAULT_ADMIN_ROLE/TRACK_ROLE accounts.

Specifically, the privileged trackDailyEarnings() function in BitcoinEarningsOracle allows for the TRACK_ROLE to set the daily mining earnings. And the privileged complementDailyEarnings() function allows for the DEFAULT_ADMIN_ROLE to complement the daily earnings for some passed day.

```
function complementDailyEarnings(
    uint256 day_,
    uint256[] memory earnings_,
```

```
        uint256[] memory hashrates_
    ) public onlyRole(DEFAULT_ADMIN_ROLE) {
        require(
            day_ < _today(),
            "BitcoinYeildOracle: can't to complement on today"
        );
        require(
            _dailyEarnings[day_] == 0,
            "BitcoinYeildOracle: complement only missing earning"
        );
        _makerDailyEarnings(day_, earnings_, hashrates_);
        emit ComplementDailyEarnings(day_, _dailyEarnings[day_]);
    }

    function trackDailyEarnings(
        uint256[] memory earnings_,
        uint256[] memory hashrates_
    ) public onlyRole(TRACK_ROLE) {
        uint256 day = _today();
        _makerDailyEarnings(day, earnings_, hashrates_);
        emit TrackDailyEarnings(day, _dailyEarnings[day]);
    }
```

Listing 3.8: Example Privileged Operations in the `BitcoinEarningsOracle` Contract

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the privileged accounts may also be a counter-party risk to the protocol users. It is worrisome if the privileged accounts are plain EOA accounts. 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.

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 This issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the design and implementation of the HashNFT protocol. Merlin is an infrastructure protocol for DeFi mining hash power and its derivatives. It digitizes real-world cryptocurrency hash power assets and then introduces them into the DeFi ecosystem through a hash power oracle and a decentralized settlement system. The audited HashNFT is the first Real-World Assets launched by Merlin to manage the investment of hash power assets. 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-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.