SMART CONTRACT AUDIT REPORT

for

PSTAKE

Prepared By: Yiqun Chen

PeckShield
July 9, 2021

Document Properties

Client	Persistence
Title	Smart Contract Audit Report
Target	pStake
Version	1.0
Author	Xuxian Jiang
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 9, 2021	Xuxian Jiang	Final Release
1.0-rc1	June 20, 2021	Xuxian Jiang	Release Candidate #1
0.2	June 14, 2021	Xuxian Jiang	Additional Findings #1
0.1	May 31, 2021	Xuxian Jiang	Initial Draft

Contact

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

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

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `pStake` 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 pStake

`pStake` is a liquid staking solution which helps in unlocking the liquidity of the staked assets. The `pStake` solution is built up using ERC20 contracts over `Ethereum` blockchain. It is designed as an inter-chain DeFi product of `Persistence` to enable enable liquid staking of PoS chains. In particular, it works with `pBridge` to allow for transfer of value between multiple disparate blockchains like `Ethereum`, `Cosmos` , `Persistence` etc. Unlike other bridges available in the blockchain ecosystem which only facilitates creating peg tokens, the bridge solution can also perform inter-chain transactions pertaining to PoS staking and unstaking, which are fulfilled at the protocol level in the native chain.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of pStake

Item	Description
Target	pStake
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 9, 2021

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

- https://github.com/persistenceOne/pStakeSmartContracts.git (6435cbd)

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

- https://github.com/persistenceOne/pStakeSmartContracts.git (ee0618d)

1.2 About PeckShield

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

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 [10]:

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

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

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 accordingly classified into four categories, 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 contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

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

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

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

1.4 Disclaimer

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

Table 1.4: 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.

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 design and implementation of `pStake`. 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	2	🟦 🟦
Medium	1	🟦
Low	3	🟦 🟦 🟦
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 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 high-severity vulnerabilities, 1 medium-severity vulnerability, and 3 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Logic Of getTotalUnbondedTokens()	Business Logic	Fixed
PVE-002	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time And State	Fixed
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-004	Low	Improved Sanity Checks For System Parameters	Coding Practices	Fixed
PVE-005	High	Improved Business Logic in withdrawUTokens()	Business Logic	Mitigated
PVE-006	High	ABI Mismatch in getHolderAttributes()	Business Logic	Fixed

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

3 | Detailed Results

3.1 Improved Logic Of getTotalUnbondedTokens()

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

- Target: LiquidStaking
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The pStake protocol is a liquid staking solution which helps in unlocking the liquidity of the staked assets. It also works closely with pBridge to efficiently transfer assets, including ERC20, ERC721, ERC1155 and many other token standards, between multiple disparate blockchains like Ethereum, Cosmos, Persistence etc. While examining the locking and unlocking logic, we notice the current implementation can be improved.

To elaborate, we show below the getTotalUnbondedTokens() function from the LiquidStaking contract. This function imposes an unnecessary restriction in only allowing the caller to be the given staker. As a public view function, such restriction is considered unnecessary.

```
/**
 * @dev get Total Unbonded Tokens
 * @param staker: account address
 *
 */
function getTotalUnbondedTokens(address staker) public view virtual override
    whenNotPaused returns (uint256 unbondingTokens) {
    if(staker == _msgSender()){
        uint256 _unstakingExpirationLength = _unstakingExpiration[staker].length;
        for (uint256 i=0; i<_unstakingExpirationLength; i=i.add(1)) {
            if (block.timestamp > _unstakingExpiration[staker][i]) {
                unbondingTokens = unbondingTokens.add(_unstakingAmount[staker][i]);
            }
        }
    }
```

```
        return unbondingTokens;
    }
```

Listing 3.1: LiquidStaking::getTotalUnbondedTokens()

Moreover, this function contains an internal for-loop, which iterates the entire array of _unstakingExpiration to compute the total unbonded tokens for retrieval. However, the first counter[staker] members are already zeroed out, which can be taken into account to further optimize this routine.

Recommendation Apply the above two optimizations to simplify the getTotalUnbondedTokens() logic. An example revision is shown below:

```
    /**
     * @dev get Total Unbonded Tokens
     * @param staker: account address
     *
     */
    function getTotalUnbondedTokens(address staker) public view virtual override
        whenNotPaused returns (uint256 unbondingTokens) {
            uint256 _unstakingExpirationLength = _unstakingExpiration[staker].length;
            for (uint256 i=_counters[staker]; i<_unstakingExpirationLength; i=i.add(1))
                {
                if (block.timestamp > _unstakingExpiration[staker][i]) {
                    unbondingTokens = unbondingTokens.add(_unstakingAmount[staker][i]);
                }
            }
        }
        return unbondingTokens;
    }
```

Listing 3.2: Revised LiquidStaking::getTotalUnbondedTokens()

Status The issue has been fixed by this commit: 18ded94.

3.2 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: STokens
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

Description

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

We notice there are several occasions where the `checks-effects-interactions` principle is violated. Using the `STokens` as an example, the `_calculateRewards()` function (see the code snippet below) is provided to calculate rewards for the provided. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 132) starts before effecting the update on the internal state (line 135), 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.

```
121     /**
122      * @dev Calculate rewards for the provided 'address'
123      * @param to: account address
124      */
125     function _calculateRewards(address to) internal returns (uint256){
126         // Calculate the rewards pending
127         uint256 _reward = calculatePendingRewards(to);
128         // mint uTokens only if reward is greater than zero
129         if(_reward>0) {
130             // Mint new uTokens and send to the callers account
131             emit CalculateRewards(to, _reward, block.timestamp);
132             _uTokens.mint(to, _reward);
133         }
134         // Set the new stakedBlock to the current
135         _stakedBlocks[to] = block.number;
136         return _reward;
137     }
```

Listing 3.3: `STokens::_calculateRewards()`

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier or follow the best practice of `Checks-Effects-Interactions` to block possible `re-entrancy`.

Status The issue has been fixed by this commit: `18ded94`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

The pStake protocol has a privileged admin account that plays a critical role in governing and regulating the protocol-wide operations (e.g., adding various roles, setting token contracts, and configuring various system parameters). In the following, we show representative privileged operations in the protocol's core TokenWrapper contract.

```
    function generateUTokens(address to, uint256 amount) public virtual override
        whenNotPaused {
        require(amount>0, "TokenWrapper: Number of tokens should be greater than 0");
        require(hasRole(BRIDGE_ADMIN_ROLE, _msgSender()), "TokenWrapper: Only bridge
            admin can mint new tokens for a user");
        emit GenerateUTokens(to, amount, block.timestamp);
        _uTokens.mint(to, amount);
    }
```

Listing 3.4: TokenWrapper::generateUTokens()

We emphasize that the privilege assignment may be necessary and consistent with the token design. However, it is worrisome if the admin is not governed by a DAO-like structure. Note that a compromised admin account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the entire PoS bridge design.

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 and partially mitigated. Specifically, the team confirms that these privileged operations are handled by a quorum which will be instituted using the persistenceBridge.

3.4 Improved Sanity Checks For System Parameters

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `pStake` protocol is no exception. Specifically, if we examine the `LiquidStaking` contract, it has a public `setFees()` function that allows for reconfiguring the stake and unstake fee.

```
    /**
     * @dev Set 'fees', called from admin
     * @param stakeFee: stake fee
     * @param unstakeFee: unstake fee
     *
     * Emits a {SetFees} event with 'fee' set to the stake and unstake.
     *
     */
    function setFees(uint256 stakeFee, uint256 unstakeFee) public virtual returns (bool
        success) {
        require(hasRole(DEFAULT_ADMIN_ROLE, _msgSender()), "LiquidStaking: User not
            authorised to set fees");
        _stakeFee = stakeFee;
        _unstakeFee = unstakeFee;
        emit SetFees(stakeFee, unstakeFee);
        return true;
    }
```

Listing 3.5: `LiquidStaking::setFees()`

Our result shows the update logic on these fee parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of a large `_stakeFee/_unstakeFee` parameters will hurt protocol users. Note that the `TokenWrapper` contract can also be benefited from improving its `setFees()` function.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

Status The issue has been fixed by this commit: `18ded94`.

3.5 Improved Business Logic in withdrawUTokens()

- ID: PVE-005
- Severity: High
- Likelihood: High
- Impact: High
- Target: `TokenWrapper`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned earlier, the `pStake` protocol is a liquid staking solution which helps in unlocking the liquidity of the staked assets. It has a core `TokenWrapper` contract that allows for generating and withdrawing `UTokens`. While examining the withdraw logic, we notice a flawed handling.

To elaborate, we show below the `withdrawUTokens()` function from the `TokenWrapper` contract. This function has a straightforward logic in burning `UTokens` for the provided `address` and `tokens`. However, the actual burn amount in the current logic is the computed `finalTokens` (line 213), not the requested `tokens`!

```
204     function withdrawUTokens(address from, uint256 tokens, string memory toChainAddress)
             public virtual override whenNotPaused {
205         require(tokens>_minWithdraw, "TokenWrapper: Requires a min withdraw amount");
206         //check if toChainAddress is valid address
207         bool isAddressValid = toChainAddress.isBech32AddressValid(hrpBytes,
                controlDigitBytes, dataBytesSize);
208         require(isAddressValid == true, "TokenWrapper: Invalid chain address ");
209         uint256 _currentUTokenBalance = _uTokens.balanceOf(from);
210         require(_currentUTokenBalance>=tokens, "TokenWrapper: Insuffcient balance for
                account");
211         require(from == _msgSender(), "TokenWrapper: Withdraw can only be done by Staker
                ");
212         uint256 finalTokens = (((tokens.mul(100)).mul(_valueDivisor)).sub(_withdrawFee))
                .div(_valueDivisor.mul(100));
213         _uTokens.burn(from, finalTokens);
214         emit WithdrawUTokens(from, finalTokens, toChainAddress, block.timestamp);
215     }
216 }
```

Listing 3.6: `TokenWrapper::withdrawUTokens()`

Recommendation Revise the `withdrawUTokens()` logic to burn the correct amount of `UTokens`.

Status This issue has been confirmed and additional improvement has been made in the following commit: `18ded94`.

3.6 ABI Mismatch in getHolderAttributes()

- ID: PVE-006
- Severity: High
- Likelihood: High
- Impact: High
- Target: STokens
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The pStake protocol has an internal helper routine getHolderAttributes() that is defined to retrieve various attributes associated with a holder. However, this routine is not used properly with mis-interpreted return results.

In particular, if we pay attention to the return values of this function, it is defined to return three values in the following order: lpBalance, lpSupply, and sTokenSupply. However, the function is used with the expectation of returning values in a different order: sTokenSupply, lpBalance, and lpSupply!

```
    function getHolderAttributes(address whitelistedAddress, address userAddress) public
         view returns (uint256 lpBalance, uint256 lpSupply, uint256 sTokenSupply){
        // copy all holder logic attributes to local variables
        address _lpTokenERC20ContractAddress = _holderContractAddresses[
            whitelistedAddress][1];
        address _sTokenReserveContractAddress = _holderContractAddresses[
            whitelistedAddress][2];
        ...
    }
```

Listing 3.7: LiquidStaking::getTotalUnbondedTokens()

```
    function generateHolderRewards(address whitelistedAddress, address userAddress)
        internal returns (bool){

        // CALCULATE TOTAL REWARD OF WHITELISTED CONTRACT USING STOKEN RESERVE TOTAL
            SUPPLY::

        uint256 _sTokenReserveSupply;
        uint256 _lpTokenBalance;
        uint256 _lpTokenSupply;
        ...
    }
```

Listing 3.8: LiquidStaking::getTotalUnbondedTokens()

Recommendation Ensure the consistency between the definition and use of getHolderAttributes ().

Status The issue has been fixed by this commit: 18ded94.

4 Conclusion

In this audit, we have analyzed the design and implementation of the pStake protocol. The system presents a unique offering as a liquid staking solution which helps in unlocking the liquidity of the staked assets. It also works closely with pBridge to efficiently transfer assets, including ERC20, ERC721, ERC1155 and many other token standards, between multiple disparate blockchains like Ethereum, Cosmos , Persistence etc. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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

References

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

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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

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

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

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

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

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

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