#Hashlock.

Surge (Protocol)

SMART CONTRACT

Security Audit

Performed on Contracts:

AccountV3TBD.sol

MD5 Hash:cc1bcf51eeac7f2e39833cb4c265141f

DealFactory.sol

MD5 Hash:b7e4d45cec51bf701478d3144f04174c

DealNFT.sol

MD5 Hash: 7ab6df1634c3fd994a9ef940c4587d11

Platform

ETH

hashlock.com.au

June 2024

Table of Contents

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE THAT COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR THE USE OF THE CLIENT.

Executive Summary

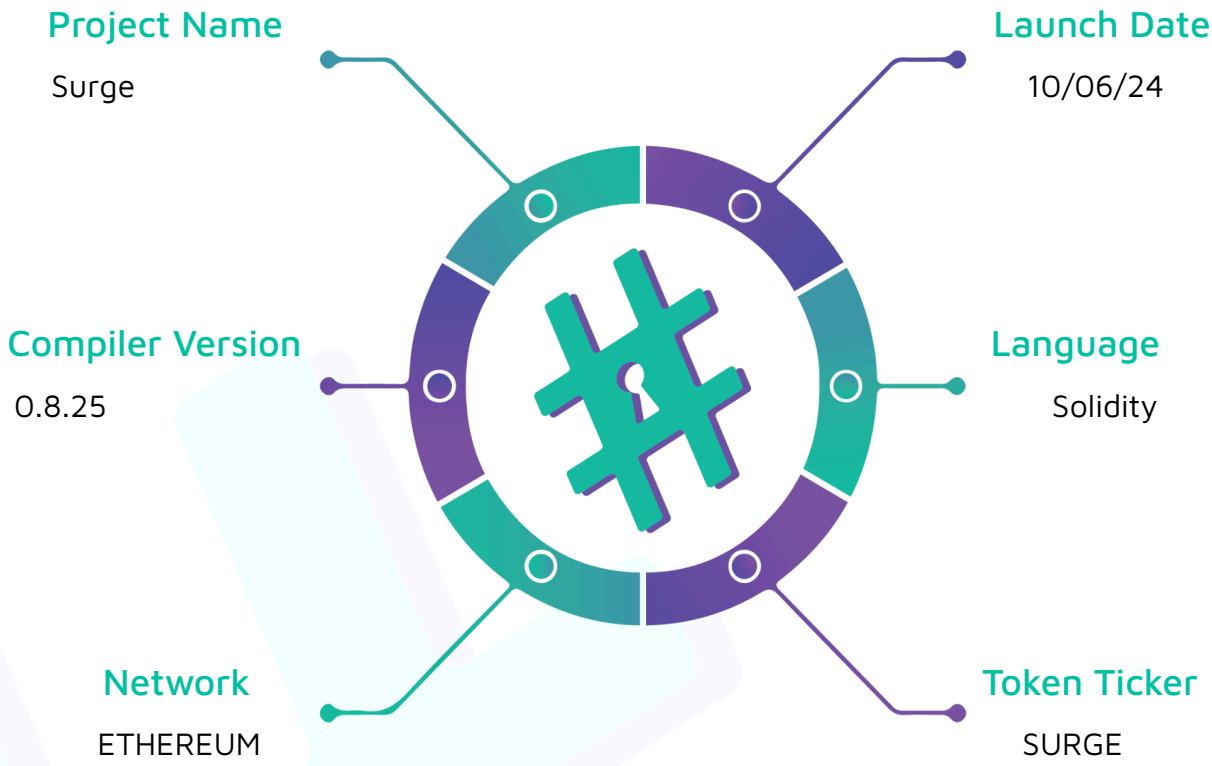
The Surge team partnered with Hashlock to conduct a security audit of their DealFactory and DealNFT smart contracts. Hashlock manually and proactively reviewed the code to ensure the project's team and community that the deployed contracts were secure.

Project Context

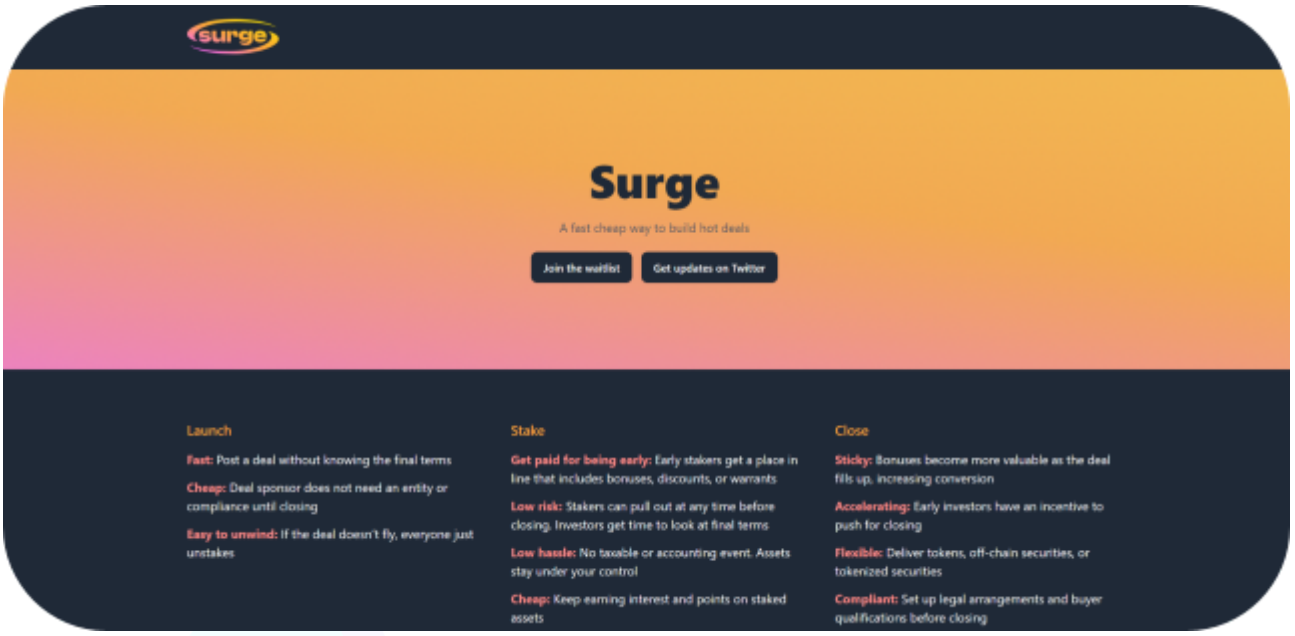
The Surge project aims to audit and enhance the security of a "Deal" object, which is part of a system where users can stake coins to secure a position in buying deals, potentially with bonuses.
This system utilizes a unique implementation of Token Bound Accounts (TBA) to manage stakes through NFTs, ensuring that each stake is linked to a specific NFT and can be managed accordingly.

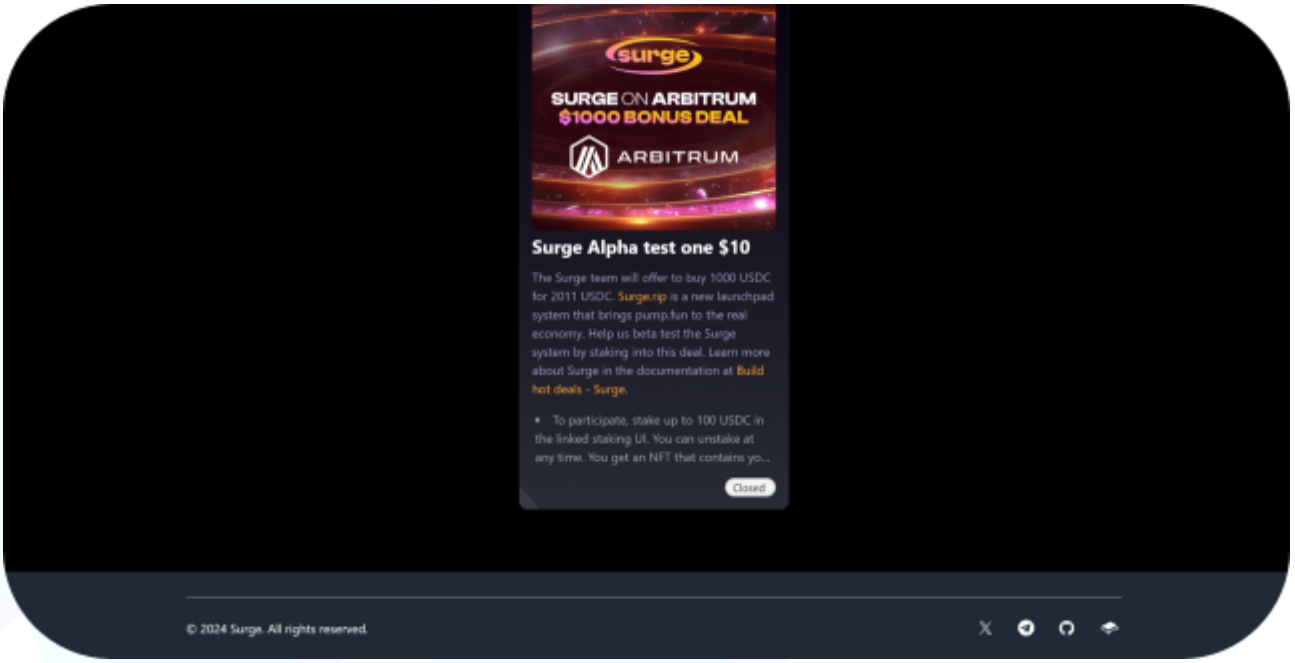
Project Name: Sweep and Surge
Compiler Version: 0.8.25
Website: https://surge.rip/
Logo:

Visualised Context:



Project Visuals:





Audit scope

We at Hashlock audited the solidity code within the Surge project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	**Surge Protocol Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**June, 2024**
Contract 1	AccountV3TBD.sol
Contract 1 MD5 Hash	cc1bcf51eeac7f2e39833cb4c265141f
Contract 2	DealFactory.sol
Contract 2 MD5 Hash	b7e4d45cec51bf701478d3144f04174c
Contract 2	DealNFT.sol
Contract 3 MD5 Hash	7ab6df1634c3fd994a9ef940c4587d11

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses token-bound contracts. We identified some vulnerabilities which will be required to be addressed before deploying to the public network.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on-chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section.

All vulnerabilities initially identified have yet to be resolved or acknowledged.

Hashlock found:

0 High-severity vulnerabilities

3 Medium-severity vulnerabilities

9 Low-severity vulnerabilities

2 Gas Optimisations

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Behaviours

Claimed Behaviour	Actual Behaviour

AccountV3TBD.sol
- Approve Tokens:
  - Allows the contract to approve the maximum possible amount of escrow tokens for a specific token contract.
- Execute Transactions:
  - Ensures that only allowed tokens can be used in transactions.
  - Executes transactions on behalf of the contract, adhering to the rules set by the token contract.
	Contract achieves this functionality.

DealNFT.sol
- Setup and Initialization:
  - Initialize deal parameters such as escrow token, closing delay, unstaking fee, and associated web details.
  - Activate the deal after setting up all necessary parameters.
  - Configure deal details including description, closing time, deal minimum and maximum, and arbitrator.
  - Set whether the NFTs are transferrable or not.
  - Cancel the deal if it is in the setup or active state.
- Staker Management:
  - Approve a staker to participate in the deal with a specified amount.
	Contract achieves this functionality.

- Approve a staker to be claimed by the sponsor. - Configure whitelists for staking and claiming. - Token Operations: - Stake tokens into the deal, minting a new NFT for the staker. - Unstake tokens from the deal, transferring them back to the staker minus any fees. - Recover tokens if the deal is canceled or closed. - Claim tokens from the deal by the sponsor. - State Management: - Retrieve the current state of the deal. - Get the total amount of tokens staked in the deal. - Get the next available token ID. - Get the account bound to a specific NFT. - Create an account bound to the NFT.	
DealFactory.sol - Create Deal: - Create a new DealNFT contract with specified parameters such as sponsor, name, and symbol.	**Contract achieves this functionality.**

Code Quality

This Audit scope involves the smart contracts of the Surge project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring is required.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the Surge projects' smart contract code in the form of GitHub access.

As mentioned above, code parts are well-commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments help us understand the overall architecture of the protocol.

Dependencies

Per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry-standard open-source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies

Audit Findings

Medium

[M-01] AccountV3TBD#approve - Unlimited Approval

Description

The `approve` function in the `AccountV3TBD` contract sets an unlimited approval (`type(uint256).max`) for the `tokenContract`. This means that the `tokenContract` can transfer any amount of tokens from the `AccountV3TBD` contract's balance. If the `tokenContract` is compromised or malicious, it can drain all tokens from the `AccountV3TBD` contract's balance.

Additionally, some tokens (like the very popular USDT) do not work when changing the allowance from an existing non-zero allowance value (it will revert if the current approval is not zero to protect against front-running changes of approvals). These tokens must first be approved for zero and then the actual allowance can be approved. Furthermore, OpenZeppelin's implementation of `safeApprove` would throw an error if an approve is attempted from a non-zero value ("SafeERC20: approve from non-zero to non-zero allowance").

Vulnerability Details

- **Unlimited Approval**: Granting unlimited approval (`type(uint256).max`) allows the approved contract to transfer any amount of tokens from the `AccountV3TBD` contract's balance. This poses a significant security risk if the approved contract is compromised or contains vulnerabilities.
- **Approval Reset**: If the `approve` function is called multiple times without resetting the current allowance to zero first, it might lead to potential race conditions. This can happen if the token contract does not handle approvals correctly, allowing an attacker to exploit the approval process.

Impact

A compromised or malicious `tokenContract` can drain all approved tokens from the `AccountV3TBD` contract, leading to a complete loss of funds.

Granting unlimited approvals diminishes the overall security of the contract and opens up potential attack vectors.

Recommendation

1. **Reset Approval**: Before setting a new approval, reset the current approval to zero. This prevents race conditions and ensures that the approval amount is always controlled.
2. **Limit Approval Amount**: Instead of using type(uint256).max, set a reasonable limit for the approval amount.

Status

Resolved

[M-02] AccountV3TBD#approve - Deprecated safeApprove Function Usage

Description

The `approve` function in the `AccountV3TBD` contract uses the deprecated `safeApprove` function from OpenZeppelin's SafeERC20 library. The `safeApprove` function is known to be vulnerable to race conditions and front-running attacks.

Vulnerability Details

The `safeApprove` function is deprecated due to its vulnerability to race conditions. More information can be found in the SWC Registry under SWC-114.

Impact

Using the deprecated `safeApprove` function can lead to unexpected behavior and potential loss of funds due to race conditions and front-running attacks.

Recommendation

Use `safeIncreaseAllowance` and `safeDecreaseAllowance`: Instead of using the deprecated `safeApprove` function, use `safeIncreaseAllowance` and `safeDecreaseAllowance` to manage token allowances safely.

Status

Resolved

[M-03] DealNFT#sponsor - Inability to Change Sponsor Address

Description

The `DealNFT` contract currently lacks a mechanism to change the sponsor address securely, which poses a significant security risk if the sponsor's private key is compromised or lost.

Vulnerability Details

Functions like `setWhitelists`, `approveStaker`, `approveBuyer`, and `cancel` are protected by the `onlySponsor` modifier, but there is no mechanism to change the sponsor if needed. This means that if the sponsor's private key is compromised or lost, there is no way to change the sponsor or control the contract anymore.

Impact

- **Security Risk:** If the sponsor's private key is compromised, an attacker can manipulate critical functions, leading to unauthorized actions and potential loss of funds.
- **Inflexibility:** The inability to change the sponsor after deploying the contract reduces the flexibility and adaptability of the contract to changing circumstances. If the sponsor's key is lost, the contract becomes uncontrollable.

Recommendation

Implement a mechanism to change the sponsor address securely. This can be achieved through a multi-signature wallet or a governance mechanism. Additionally, a two-step transfer of role process should be implemented to ensure secure and deliberate changes.

```
 /**
     * @notice Initiate the transfer of the sponsor role to a new address
     * @param newSponsor The address of the new sponsor
     */
    function initiateSponsorTransfer(address newSponsor) external onlySponsor {
        require(newSponsor != address(0), "new sponsor cannot be zero address");
        _pendingSponsor = newSponsor;
    }

    /**
     * @notice Complete the transfer of the sponsor role to the new address
     */
    function completeSponsorTransfer() external {
         require(msg.sender == _pendingSponsor, "only pending sponsor can complete the
transfer");
        address oldSponsor = sponsor;
        sponsor = _pendingSponsor;
        _pendingSponsor = address(0);
        emit SponsorChanged(oldSponsor, sponsor);
    }
```

Status

Resolved

Low

[L-01] DealFactory#Sponsor - Unused Sponsor Event

Description

The `Sponsor` event is defined in the `DealFactory` contract but is never emitted. This might be an oversight or an incomplete implementation. Events in Solidity are used to log and notify external entities about specific occurrences within a smart contract. They serve as a mechanism for emitting and recording data onto the blockchain, making it transparent and easily accessible.

Recommendation

1. **Implement Sponsor Approval Logic:** Implement the logic for sponsor approval and emit the `Sponsor` event accordingly.
2. **Remove Unused Event:** If sponsor approval is not required, consider removing the event definition to avoid confusion.

```
function approveSponsor(address sponsor, bool approved) external onlyOwner {

    // Logic to approve the sponsor

    emit Sponsor(sponsor, approved);

}
```

Status

Resolved

[L-02] DealFactory#constructor - Hardcoded Addresses

Description

The constructor of the `DealFactory` contract requires specific addresses (`registry_`, `implementation_`, `treasury_`) to be provided at deployment. If these addresses need to be updated in the future, the contract would need to be redeployed, which is not practical and reduces the flexibility of the contract.

Recommendation

1. **Implement Setter Functions:** Implement setter functions to allow updating these addresses if necessary. Ensure these functions are protected by access control to prevent unauthorized changes.
2. **Access Control:** Use the `onlyOwner` modifier or a similar mechanism to ensure that only authorized users can update these addresses.

```
import "@openzeppelin/contracts/access/Ownable.sol";


contract DealFactory is Ownable {

    // ... existing code ...


    function setRegistry(address newRegistry) external onlyOwner {

        require(newRegistry != address(0), "registry is the zero address");

        _registry = newRegistry;

    }


    function setImplementation(address newImplementation) external onlyOwner {

            require(newImplementation != address(0), "implementation is the zero
address");

        _implementation = newImplementation;

    }
```

```
    function setTreasury(address newTreasury) external onlyOwner {
        require(newTreasury != address(0), "treasury is the zero address");
        _treasury = newTreasury;
    }
}
```

Status

Acknowledged

[L-03] DealNFT#CLAIMING_FEE&CLOSING_PERIOD - Hardcoded Values in DealNFT Contract

Description

The CLAIMING_FEE and CLOSING_PERIOD are hardcoded and cannot be changed after deployment. These values are set as constants in the contract, which means they are immutable and cannot be adjusted to adapt to future requirements or changes in the ecosystem.

Recommendation

Implement setter functions to update these fees and periods, protected by access control. This will allow the contract to adapt to changing conditions while ensuring that only authorized users can make these changes.

Status

Acknowldeged

[L-04] DealNFT#configure - Incorrect Handling of Closing Time Validation

Description

The configure function contains the following lines for validating the closingTime_ parameter:

```
require(closingTime_ == 0 || closingTime_ > block.timestamp + closingDelay, "invalid closing time");

require(closingTime_ < block.timestamp + 52 weeks, "invalid closing time");
```

According to EIP-2612, signatures used on exactly the deadline timestamp are supposed to be allowed. While the signature may or may not be used for the exact EIP-2612 use case (transfer approvals), for consistency's sake, all deadlines should follow this semantic. If the timestamp is an expiration rather than a deadline, consider whether it makes more sense to include the expiration timestamp as a valid timestamp, as is done for deadlines.

Recommendation

Modify the validation logic to include the exact boundary values as valid timestamps. This ensures consistency with EIP-2612 and other standards that allow exact deadlines.

```
function configure(

    string memory description_,

    uint256 closingTime_,

    uint256 dealMinimum_,

    uint256 dealMaximum_,

    address arbitrator_

) external nonReentrant onlySponsor {

      require(closingTime_ == 0 || closingTime_ >= block.timestamp + closingDelay, "invalid closing time");

    require(closingTime_ <= block.timestamp + 52 weeks, "invalid closing time");



    require(dealMinimum_ <= dealMaximum_, "wrong deal range");

    require(state() < State.Closed, "cannot configure anymore");



    if (state() == State.Claiming) {
```

```
        require(totalStaked() < dealMinimum, "minimum stake reached");
    }

    description = description_;
    closingTime = closingTime_;
    dealMinimum = dealMinimum_;
    dealMaximum = dealMaximum_;
    arbitrator = arbitrator_;

        emit Configure(sponsor, description, closingTime, dealMinimum, dealMaximum,
arbitrator);
}
```

Status

Resolved

[L-05] DealNFT#constructor - Inefficient String Concatenation Using abi.encodePacked

Description

In the constructor of the DealNFT contract, the _base variable is initialized using abi.encodePacked for string concatenation:

```
_base = string(abi.encodePacked(
    baseURI_,
    "/chain/",
    block.chainid.toString(),
    "/deal/",
    address(this).toHexString(),
```

```
    "/token/"
));
```

Solidity version 0.8.4 introduces `bytes.concat()` and version 0.8.12 introduces `string.concat()`, which are more efficient and catch concatenation errors (in the event of a bytes data mixed in the concatenation).

Recommendation

Update the code to use `string.concat()` for string concatenation to improve efficiency and error handling.

Status

Resolved

[L-06] DealNFT - Use of Magic Numbers Instead of Constants

Description

The `DealNFT` contract uses magic numbers directly in the code, which reduces readability and maintainability. Examples include:

```
require(closingDelay < 52 weeks, "closing delay too big");

require(unstakingFee <= 100000, "cannot be bigger than 10%");

require(closingTime_ < block.timestamp + 52 weeks, "invalid closing time");

escrowToken.safeTransferFrom(tokenBoundAccount, treasury, fee / 2);
```

Even assembly can benefit from using readable constants instead of hex/numeric literals.

Recommendation

Define constants for these values to improve readability, maintainability, and consistency.

Status

Resolved

[L-07] DealNFT - Use Modifiers Instead of require Statements for Access Control

Description

The DealNFT contract uses require statements for access control in several functions, which reduces readability and maintainability. Examples include:

```
function approveClaim() external nonReentrant {

    require(msg.sender == arbitrator, "not the arbitrator");

    claimApproved = true;

    emit ClaimApproved(sponsor, arbitrator);

}


function unstake(uint256 tokenId) external nonReentrant {

    require(msg.sender == ownerOf(tokenId), "not the nft owner");

    require(state() <= State.Active, "cannot unstake after claiming/closed/canceled");


    uint256 amount = stakedAmount[tokenId];

    address tokenBoundAccount = getTokenBoundAccount(tokenId);


    approvalOf[msg.sender] += amount;

    stakedAmount[tokenId] = 0;


    uint256 fee = amount.mulDiv(unstakingFee, 1e6);

    escrowToken.safeTransferFrom(tokenBoundAccount, msg.sender, amount - fee);

    escrowToken.safeTransferFrom(tokenBoundAccount, sponsor, fee.ceilDiv(2));
```

```
    escrowToken.safeTransferFrom(tokenBoundAccount, treasury, fee / 2);


    emit Unstake(msg.sender, tokenBoundAccount, tokenId, amount);
}


function recover(uint256 tokenId) external nonReentrant {
    require(msg.sender == ownerOf(tokenId), "not the nft owner");
    require(state() >= State.Closed, "cannot recover before closed/canceled");


    address tokenBoundAccount = getTokenBoundAccount(tokenId);
    uint256 balance = escrowToken.balanceOf(tokenBoundAccount);


    escrowToken.safeTransferFrom(tokenBoundAccount, msg.sender, balance);


    emit Recover(msg.sender, tokenBoundAccount, tokenId, balance);
}
```

Recommendation

Define modifiers for access control and use them in the relevant functions to improve readability, maintainability, and consistency.

```
  /**
     * @notice Modifier to check the caller is the arbitrator
     */
    modifier onlyArbitrator() {
        require(msg.sender == arbitrator, "not the arbitrator");
        _;
```

```
    }

    /**
     * @notice Modifier to check the caller is the owner of the token
     */
    modifier onlyTokenOwner(uint256 tokenId) {
        require(msg.sender == ownerOf(tokenId), "not the nft owner");
        _;
    }
```

Status

Resolved

[L-08] DealNFT#activate - Use Scientific Notation for Readability of Large Numbers

Description

The `DealNFT` contract uses large numbers directly in the code, which reduces readability and makes it harder to verify the correctness of these values. An example includes:

```
require(unstakingFee <= 100000, "cannot be bigger than 10%");
```

This is found in the `activate` function:

```
function activate() external nonReentrant onlySponsor {
    require(address(escrowToken) != address(0), "sponsor cannot be zero");
    require(closingDelay > 0, "closing delay cannot be zero");
    require(closingDelay < 52 weeks, "closing delay too big");
    require(unstakingFee <= 100000, "cannot be bigger than 10%");
    require(bytes(web).length > 0, "web cannot be empty");
```

```
    require(bytes(twitter).length > 0, "twitter cannot be empty");

    require(bytes(image).length > 0, "image cannot be empty");



    _active = true;



    emit Activate(sponsor);

}
```

Recommendation

Use scientific notation for large numbers to improve readability and ease of verification.

Status

Resolved

[L-09] DealNFT#configure - Missing Zero Address Validation in configure Function

Description

In the `configure` function within the `DealNFT` contract, there is a lack of validation to ensure that the `arbitrator` parameter is not the zero address (`address(0)`). This omission poses a security risk, allowing anyone to inadvertently set the `arbitrator` to the zero address, which could lead to unexpected behavior or loss of functionality.

Recommendation

Add a validation check to ensure that the `arbitrator` address is not the zero address before assigning it.

Status

Acknowledged

Gas

[G-01] DealNFT#totalStaked - Superfluous Initialization of Variables to Zero

Description

In the `totalStaked` function of the `DealNFT` contract, the variable `total` is explicitly initialized to zero. This is unnecessary because Solidity automatically initializes variables to their default values, which for `uint256` is zero.

Recommendation

Remove the explicit initialization of the `total` variable to zero. Solidity will automatically initialize it to zero.

Status

Resolved

[G-02] DealFactory - State Variables Should Be Declared Immutable

Description

In the `DealFactory` contract, certain state variables are set only once during the contract's deployment and are never modified afterward. These variables should be declared as `immutable` to save gas and improve code clarity. The variables in question are:

- `_registry`
- `_implementation`
- `_treasury`

Recommendation

Add the immutable keyword to the state variables that are set only in the constructor and never modified afterward.

Status

Resolved

Centralisation

The Surge project values decentralization as a contributor to the security and predictability of deals for stakers.

A Surge deal has no owner executable functions. The smart contract enforces a relationship between stakers, a deal sponsor, and an optional arbitrator.

A staker who deposits assets can unstake, and get those assets back, in all circumstances except for a successful closing and claim by a sponsor. The claiming period is time limited. Surge protocol cannot interfere with the rights to unstake or change the terms of a deal.

Conclusion

After Hashlocks analysis, the Surge project seems to have a sound and well-tested code base, now that our findings have been resolved/acknowledged in order to achieve security. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits are to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we still need to verify the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown not to represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contracts details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au


#Hashlock.