SHELLBOXES

AutoAirdrop NFT Minter

Smart Contract Security Audit

Prepared by ShellBoxes

May 31st, 2023 – June 2nd, 2023

Shellboxes.com

contact@shellboxes.com

Document Properties

Client	AutoAirdrop
Version	1.0
Classification	Public

Scope

- Audit

Contract Name	Contract Address
MyERC1155	0x708fda40d3609d763369919263697b9b1e7c305e

- Re-Audit

Contract Name	Contract Address
MyERC1155	0x142a79A9fF89E5E6d1beA5FAdc0948E30Ae9A028

Scope Files

- Audit

Files	MD5 Hash
MyERC1155.sol	f8d9673097717941b94bd7bcc077c91e

- Re-Audit

Files	MD5 Hash
MyERC1155.sol	03206b2756698d365448ee1fc1b6db44

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1 Introduction

AutoAirdrop engaged ShellBoxes to conduct a security assessment on the AutoAirdrop NFT Minter beginning on May 31st, 2023 and ending June 2nd, 2023. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code.

This document summarizes the findings of our audit.

1.1 About AutoAirdrop

An innovative venture in the decentralized finance (DeFi) space, providing a unique interaction model for enthusiasts and investors. Our primary business activity is airdrop farming, an innovative method that seeks to earn rewards from promising blockchain platforms. Our operations begin with a 48-hour 'minting' period for our unique non-fungible tokens (NFTs), which are then sold to our community. The revenue from these sales is then divided between our operations and farming wallets, funding both our regular business operations and our airdrop farming activities. Our software performs various transactions on the blockchain to mimic regular user activity, increasing our chances of receiving an airdrop, which is a reward for using a particular platform. Profits from successful airdrops are then shared with our community, providing a unique opportunity for passive income in the DeFi space. In addition to this, our website serves as an information hub and a decentralized app (dapp) that allows users to interact with our smart contracts for minting, staking, and reward distribution.

Issuer	AutoAirdrop
Type	Solidity Smart Contract
Audit Method	Whitebox

1.2 Approach & Methodology

ShellBoxes used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

1.2.1 Risk Methodology

Vulnerabilities or bugs identified by ShellBoxes are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low

Likelihood

2 Findings Overview

2.1 Summary

The following is a synopsis of our conclusions from our analysis of the AutoAirdrop NFT Minter implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.2 Key Findings

The contract is used to perform a tiered minting process for an ERC-1155 token, it also implements a referral system, when the referral system is enabled and a valid referral address is provided, the contract provides a discount to the minter and adds referral points to the referral's account. This smart contract can be improved by addressing the discovered flaws, which include , 3 medium-severity, 4 low-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Potential Misappropriation of Funds by the Owner	MEDIUM	Fixed
SHB.2. Uninitialized mintPrices and referPoints in The Constructor	MEDIUM	Fixed
SHB.3. Concerns Regarding Excessive Owner Control and Authority	MEDIUM	Acknowledged
SHB.4. Potential Rounding Errors in Discount and Referral Calculations	LOW	Fixed
SHB.5. Missing Value Verification	LOW	Fixed
SHB.6. Renounce Ownership Risk	LOW	Fixed
SHB.7. Missing Address Verification in ownerMint Function	LOW	Fixed

3 Finding Details

SHB.1 Potential Misappropriation of Funds by the Owner

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 2
- Impact : 2

Description:

The current design of the platform's funds withdrawal mechanisms presents a potential security risk, wherein the owner could misuse their permissions to misappropriate funds. The smart contract code has two withdrawal functions: withdrawFunds() and manualWithdrawFunds().

The withdrawFunds() function is designed to distribute funds to the treasury and operations addresses in pre-determined proportions, which is a standard practice that ensures proper allocation of resources for the platform's maintenance and development. This function ensures that the funds in the contract are distributed transparently and consistently.

However, the manualWithdrawFunds() function grants the owner the ability to manually withdraw any amount of funds to any address, including their own, which presents a significant risk, especially if the owner's keys are compromised. This function does not have the checks and balances present in the withdrawFunds() function, such as the predefined allocation percentages or the requirement to send funds only to specific, intended addresses (i.e., treasury and operations addresses). As a result, it could be exploited to withdraw funds improperly.

Files Affected:

SHB.1.1: MyERC1155.sol

```
function withdrawFunds() external onlyOwner{
    require(treasuryAddress != address(0) && operationsAddress !=
       ↪ address(0),"set addresses");
    uint256 total = IERC20(paymentToken).balanceOf(address(this));
    require(total > 1000, "low funds");
    uint256 treasuryAmount = (total*250)/1000;
    uint256 operationalAmount = (total - treasuryAmount);
    IERC20(paymentToken).safeTransfer(treasuryAddress, treasuryAmount);
    IERC20(paymentToken).safeTransfer(operationsAddress,
       ↪ operationalAmount);
    emit fundsWithdrawn(total);
}
```

SHB.1.2: MyERC1155.sol

```
function manualWithdrawFunds(address _account, uint256 _amount) external
    ↪  onlyOwner{
    require(_account != address(0) && _amount != 0,"zero address");
    require(_amount <= IERC20(paymentToken).balanceOf(address(this)),"
       ↪ low balance");
    IERC20(paymentToken).safeTransfer(_account, _amount);
    emit fundsWithdrawn(_amount);
}
```

Recommendation:

To mitigate this risk, it is recommended that the manualWithdrawFunds() function be revised or removed altogether. If there is a legitimate need for manual withdrawals, these should be subjected to stricter requirements, such as multi-signature approval or predefined withdrawal limits, to prevent any potential misuse. Also, consider implementing a more robust audit trail for fund withdrawals to ensure transparency and accountability.

Updates

The team has resolved the issue by removing the manualWithdrawFunds() function ,and now manage withdrawals using the withdrawFunds function.

SHB.2 Uninitialized mintPrices and referPoints in The Constructor

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 1
- Impact : 3

Description:

The mintPrices and referPoints mappings should be initialized in the constructor to avoid operating with default values. If these arrays are not initialized, the contract may rely on default values, which can negatively impact the business logic and result in unexpected behaviors. It is crucial to ensure that the initial values for mintPrices and referPoints are explicitly set during contract deployment to reflect the intended pricing and referral point configurations.

Files Affected:

SHB.2.1: MyERC1155.sol

```
mapping(uint256 => uint256) public mintPrices;
```

SHB.2.2: MyERC1155.sol

```
mapping(uint256 => uint256) public referPoints;
```

Recommendation:

To address this issue, modify the constructor of the contract to include the initialization of the mintPrices and referPoints mappings. Set the appropriate values for each tier to align with the desired pricing and referral point structure. By explicitly initializing these arrays during deployment, you ensure that the contract operates with the intended values from the beginning, eliminating any reliance on default values and minimizing the risk of unexpected behaviors.

Updates

The team has resolved the issue by initializing the mintPrices and referPoints in the constructor.

SHB.2.3: MyERC1155.sol

```
constructor(
    string memory _uri,
 uint256 _saleCap,
  address _paymentToken,
   uint256[] memory tier,
    uint256[] memory price,
     uint256[] memory points
     ) ERC1155(_uri) {
    require(_paymentToken != address(0), "set payment Token");
    require(_saleCap > 0, "0 sale cap");
    require(tier.length == price.length && tier.length == points.length
       ↪ && tier.length < 5, "length misMatched");
    totalSaleCap = _saleCap;
    paymentToken = _paymentToken;
    for(uint256 i = 0; i < tier.length; i++) {
    require(tier[i] >= 1 && tier[i] <= 4 , "Invalid tier");
    require(points[i] > 0 && price[i] >=1000, "0 points/price<1000");
    mintPrices[tier[i]] = price[i];
    referPoints[tier[i]] = points[i];
    }
}
```

SHB.3 Concerns Regarding Excessive Owner Control and Authority

- Severity : MEDIUM
- Status : Acknowledged

- Likelihood : 1
- Impact : 3

Description:

The contract design grants the owner extensive control and authority over various critical variables and functions, which raises concerns about the concentration of power. While it is common for the contract owner to have certain privileges and responsibilities, excessive control can introduce potential risks and vulnerabilities.

Files Affected:

Function:

- ownerMint
- awardReferalPoints
- setMintPrice
- setReferPoints
- setTotalSaleCapp
- addWhitelist
- removeWhitelist
- blacklistNft
- removeBlacklistedNft
- setAllowedPerMint
- setDiscountPercentage

- setReferalDiscount
- setReferalClaimPoints
- setTier4MaxCap
- setAddresses
- setPaymentToken
- toggleSale
- toggleWhitelist
- toggleDiscount
- toggleRefer
- withdrawFunds
- manualWithdrawFunds

Recommendation:

To address the concentration of control and mitigate the risks associated with the extensive power granted to the owner, it is strongly recommended to implement a decentralized governance model, such as a Decentralized Autonomous Organization (DAO) or a multisignature (multisig) mechanism.

Updates

The team has acknowledged the issue, stating that they will be implementing some security practices to protect the owner's wallet.

SHB.4 Potential Rounding Errors in Discount and Referral Calculations

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

In the current smart contract implementation, the discount and referral reward calculations could potentially lead to inaccuracies due to rounding errors, especially for small amounts.

The discount is calculated as a percentage of the mint price of the token. However, since Solidity does not support floating-point arithmetic, the division operation rounds down to the nearest integer. This means that if the discount is supposed to be a small fraction of the mint price, it may be rounded down to zero. This issue affects both the discountEnabled and referEnabled parts of the code.

Files Affected:

SHB.4.1: MyERC1155.sol

```
if(discountEnabled){
    uint256 discAmount = ((mintPrices[tier]*amount) * discountPercentage
        ↪ )/ 1000;
    amountToPay = (mintPrices[tier]*amount - discAmount);
}else if(referEnabled){
    require(refferalAddress != msg.sender, "cannot refer your self");
    uint256 refDiscAmount = ((mintPrices[tier]*amount) * referDiscount)/
        ↪  1000;
    amountToPay = (mintPrices[tier]*amount - refDiscAmount);
    if(refferalAddress!= address(0)){
        require(referPoints[tier] > 0, "set referPoints");
        referReward[refferalAddress] += referPoints[tier] * amount;
    }
}else{
    amountToPay = mintPrices[tier]*amount;
}
```

Recommendation:

To mitigate the risk of rounding errors, take into consideration this edge case when choosing the mint prices for each tier as it needs to be at least greater than 1000.

Updates

The team has resolved the issue by requiring the prices to be greater than 1000.

SHB.4.2: MyERC1155.sol

```
for(uint256 i = 0; i < tier.length; i++) {
require(tier[i] >= 1 && tier[i] <= 4 , "Invalid tier");
require(points[i] > 0 && price[i] >=1000, "0 points/price<1000");
mintPrices[tier[i]] = price[i];
referPoints[tier[i]] = points[i];
}
```

SHB.4.3: MyERC1155.sol

```
    for(uint256 i = 0; i < tier.length; i++) {
    require(tier[i] >= 1 && tier[i] <= 4 && price[i] >=1000, "Invalid
        ↪ tier/price<1000");
    mintPrices[tier[i]] = price[i];
    }
}
```

SHB.5 Missing Value Verification

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The constructor and setters in the contract are not verifying the correctness of the provided values. This could lead to unexpected behavior if incorrect values are set. In particular, this could lead to security vulnerabilities if incorrect values are set into the contract. Additionally, a lack of value checks could lead to erroneous contract behavior that may disrupt the normal operation of the contract.

Files Affected:

SHB.5.1: MyERC1155.sol

```
constructor(string memory _uri, uint256 _saleCap, address _paymentToken)
    ↪ ERC1155(_uri) {
    require(_paymentToken != address(0), "set payment Token");
    totalSaleCap = _saleCap;
    paymentToken = _paymentToken;
}
```

SHB.5.2: MyERC1155.sol

```
function setMintPrice(uint256[] memory tier, uint256[] memory price)
    ↪ external onlyOwner {
    require(tier.length == price.length && tier.length < 5, "length
        ↪ misMatched");
    for(uint256 i = 0; i < tier.length; i++) {
    require(tier[i] >= 1 && tier[i] <= 4, "Invalid tier");
    mintPrices[tier[i]] = price[i];
    }
    emit mintPricesEvent(tier, price);
}
```

SHB.5.3: MyERC1155.sol

```
function setReferPoints(uint256[] memory tier, uint256[] memory _points)
    ↪  external onlyOwner {
    require(tier.length == _points.length && tier.length < 5, "length
        ↪ misMatched");
    for(uint256 i = 0; i < tier.length; i++) {
    require(tier[i] >= 1 && tier[i] <= 4, "Invalid tier");
    referPoints[tier[i]] = _points[i];
    }
    emit referPointsEvent(tier, _points);
}
```

Recommendation:

To improve the code, it is recommended to implement value verification in several areas. In the constructor, add checks to ensure that the _saleCap parameter is greater than zero. In the setMintPrice function, verify that the price is greater than zero. Similarly, in the setReferPoints function, require the referral points to be greater than zero. These checks will help ensure that critical variables are properly initialized, reducing the risk of potential issues.

Updates

The team has resolved the issue by adding value verification in the constructor, setMintPrice, and setReferPoints function.

SHB.5.4: MyERC1155.sol

```
constructor(
    string memory _uri,
  uint256 _saleCap,
   address _paymentToken,
    uint256[] memory tier,
    uint256[] memory price,
     uint256[] memory points
     ) ERC1155(_uri) {
    require(_paymentToken != address(0), "set payment Token");
    require(_saleCap > 0, "0 sale cap");
    require(tier.length == price.length && tier.length == points.length
       ↪ && tier.length < 5, "length misMatched");
    totalSaleCap = _saleCap;
    paymentToken = _paymentToken;
    for(uint256 i = 0; i < tier.length; i++) {
    require(tier[i] >= 1 && tier[i] <= 4 , "Invalid tier");
    require(points[i] > 0 && price[i] >=1000, "0 points/price<1000");
    mintPrices[tier[i]] = price[i];
    referPoints[tier[i]] = points[i];
    }
}
```

SHB.5.5: MyERC1155.sol

```
function setMintPrice(uint256[] memory tier, uint256[] memory price)
    ↪ external onlyOwner {
    require(tier.length == price.length && tier.length < 5, "length
       ↪ misMatched");
    for(uint256 i = 0; i < tier.length; i++) {
    require(tier[i] >= 1 && tier[i] <= 4 && price[i] >=1000, "Invalid
       ↪ tier/price<1000");
    mintPrices[tier[i]] = price[i];
    }
    emit mintPricesEvent(tier, price);
}
```

SHB.5.6: MyERC1155.sol

```
function setReferPoints(uint256[] memory tier, uint256[] memory _points)
    ↪  external onlyOwner {
    require(tier.length == _points.length && tier.length < 5, "length
    ↪ misMatched");
    for(uint256 i = 0; i < tier.length; i++) {
    require(tier[i] >= 1 && tier[i] <= 4 && _points[i] > 0, "Invalid
    ↪ tier/0 points");
    referPoints[tier[i]] = _points[i];
    }
    emit referPointsEvent(tier, _points);
}
```

SHB.6 Renounce Ownership Risk

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The contract inherits from the Ownable pattern, which includes a renounceOwnership function. This function, if called, can result in the contract having no owner, causing a Denial of Service (DoS) for the functions with the onlyOwner modifier.

In the current implementation, the contract is ownable, and the renounceOwnership function allows the contract owner to permanently relinquish ownership. If the ownership is renounced, the contract will not have an owner, and any function with the onlyOwner modifier will become unreachable. This scenario could lead to a Denial of Service (DoS) on these functions, as no one would be able to execute them, effectively rendering them useless.

Files Affected:

SHB.6.1: MyERC1155.sol

```
contract MyERC1155 is ERC1155URIStorage , Ownable, ReentrancyGuard {
```

Recommendation:

To mitigate this risk, consider either removing the renounceOwnership function or replacing it with a safer alternative, such as allowing ownership transfer to a predefined address, like a multisig wallet or a timelock contract. This approach will maintain control over the contract and prevent a potential DoS on the functions with the onlyOwner modifier.

Updates

The team has resolved the issue by removing the renounceOwnership to disable the functionality.

SHB.7 Missing Address Verification in ownerMint Function

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The ownerMint function in the smart contract allows the contract owner to mint tokens directly to a specified account. However, there is a missing address verification check in the function, which does not ensure that the provided account address is different from the zero address (address(0)). This poses a potential vulnerability as tokens could unintentionally be minted to the zero address, resulting in an irretrievable loss of tokens.

Files Affected:

SHB.7.1: MyERC1155.sol

```
function ownerMint(uint256 tier, uint256 amount, address _account)
    ↪ external onlyOwner {

    require(tier >= 1 && tier <= 4, "Invalid tier");

    uint256 tokenId;

    for(uint256 i=0; i<amount; i++){
        tokenCounts[tier]++;
        tokenId = tokenCounts[tier];
        uint256 id = tier * 10**uint256(digit(tokenId)) + tokenId;
        _mint(_account, id, 1, "");
    }

}
```

Recommendation:

To mitigate the risk of inadvertently minting tokens to the zero address, it is recommended to add an address verification check in the ownerMint function. Before executing the minting operation, ensure that the provided account address (_account) is not equal to the zero address (address(0)). If the address is the zero address, revert the transaction and throw an appropriate error message indicating that the zero address is not allowed.

Updates

The team has resolved the issue by adding a verification over the _account address and requiring it to be different from the address(0).

SHB.7.2: MyERC1155.sol

```
function ownerMint(uint256 tier, uint256 amount, address _account)
    ↪ external onlyOwner {

    require(tier >= 1 && tier <= 4, "Invalid tier");
    require(_account != address(0) && amount != 0, "zero address/amount
        ↪ ");
```

4 Best Practices

BP.1 Performance Optimizations Needed in Token Minting Logic

Description:

The contract code could benefit from some performance optimizations to minimize storage operations and consequently, reduce the gas costs associated with transactions.

In the current design, the tokenCounts[tier]++; operation is performed inside a loop, which leads to multiple storage writes. Each storage write operation in Ethereum incurs a cost of 20,000 gas. Therefore, repetitive writes can significantly increase the transaction gas cost.

Moreover, the contract code utilizes the _mint function inside the loop to mint tokens one by one. ERC1155 contract includes a function called _mintBatch which allows for minting multiple tokens in a single transaction, which can be more gas efficient when dealing with large numbers of tokens.

Files Affected:

BP.1.1: MyERC1155.sol

```
for(uint256 i=0; i<amount; i++){

    tokenCounts[tier]++;
    tokenId = tokenCounts[tier];
    uint256 id = tier * 10**uint256(digit(tokenId)) + tokenId;
    _mint(msg.sender, id, 1, "");
}
```

To optimize gas cost, you can move the tokenCounts[tier]++; operation out of the loop and replace it with tokenCounts[tier] += amount;. This would perform a single storage write operation instead of multiple.

Additionally, consider using the _mintBatch function for minting multiple tokens. You would need to construct the array of token IDs and their corresponding amounts before calling _mintBatch.

The optimized code might look like this:

BP.1.2: MyERC1155.sol

```
    tokenCounts[tier] += amount;
    uint256[] memory ids = new uint256[](amount);
    uint256[] memory amounts = new uint256[](amount);
    for(uint256 i=0; i<amount; i++){
        tokenId = tokenCounts[tier] + i + 1;
        uint256 id = tier * 10**uint256(digit(tokenId)) + tokenId;
        ids[i] = id;
        amounts[i] = 1;
    }
    _mintBatch(msg.sender, ids, amounts, "");
```

Status - Fixed

BP.2 Utilize Merkle Trees for Whitelisting

Description:

Consider implementing Merkle trees for whitelisting instead of relying solely on mappings. Merkle trees offer several advantages for managing large-scale whitelists efficiently. By leveraging Merkle trees, you can reduce the gas costs associated with storing and verifying whitelist entries, especially when the number of addresses becomes significant.

Merkle trees provide a compact representation of the whitelist data, allowing for efficient inclusion and exclusion checks. Additionally, they offer a secure and tamper-proof way to validate the authenticity of a specific address within the whitelist.

To implement Merkle trees for whitelisting, you need to:

- Generate a Merkle root hash by combining the hashes of all whitelisted addresses. - Store the Merkle root hash in the contract. - When checking if an address is whitelisted, provide the necessary Merkle proof along with the address. - Verify the Merkle proof against the Merkle root hash in the contract to determine the address's whitelist status. By adopting Merkle trees for whitelisting, you can optimize gas costs and enhance the security and scalability of your contract's whitelist management.

Files Affected:

BP.2.1: MyERC1155.sol

```
if(whitelistEnabled){
require(isWhitelisted[msg.sender] , " Not whiteListed");
}
```

Status - Acknowledged

BP.3 Use Custom Solidity Errors with if and revert Instead of require Statements

Description:

In the current implementation, the contract uses require statements for various validation checks. While this approach works, using custom Solidity errors with if and revert statements can provide more informative and specific error messages. This makes it easier for developers and users to understand the reasons behind failed transactions, and it allows for better error handling.

To implement this best practice, consider replacing the existing require statements with if and revert statements that include custom error messages. Define custom error types using the error keyword and provide descriptive names and parameters to convey the nature of the error. Then, use these custom error types in combination with revert statements in your validation checks.

Status - Acknowledged

5 Tests

Because the project lacks unit, integration, and end-to-end tests, we recommend establishing numerous testing methods covering multiple scenarios for all features in order to ensure the correctness of the smart contracts.

6 Conclusion

We examined the design and implementation of AutoAirdrop NFT Minter in this audit and found several issues of various severities. We advise AutoAirdrop team to implement the recommendations contained in all 7 of our findings to further enhance the code's security. It is of utmost priority to start by addressing the most severe exploit discovered by the auditors then followed by the remaining exploits, and finally we will be conducting a re-audit following the implementation of the remediation plan contained in this report.

We would much appreciate any constructive feedback or suggestions regarding our methodology, audit findings, or potential scope gaps in this report.

7 Disclaimer

Shellboxes reports should not be construed as ”endorsements” or ”disapprovals” of particular teams or projects. These reports do not reflect the economics or value of any ”product” or ”asset” produced by any team or project that engages Shellboxes to do a security evaluation, nor should they be regarded as such. Shellboxes Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the examined technology, nor do they provide any indication of the technology’s proprietors, business model, business or legal compliance. Shellboxes Reports should not be used in any way to decide whether to invest in or take part in a certain project. These reports don’t offer any kind of investing advice and shouldn’t be used that way. Shellboxes Reports are the result of a thorough auditing process designed to assist our clients in improving the quality of their code while lowering the significant risk posed by blockchain technology. According to Shellboxes, each business and person is in charge of their own due diligence and ongoing security. Shellboxes does not guarantee the security or functionality of the technology we agree to research; instead, our purpose is to assist in limiting the attack vectors and the high degree of variation associated with using new and evolving technologies.

SHELLBOXES

For a Contract Audit, contact us at contact@shellboxes.com