SHELLBOXES

Sukiyaki Finance

Smart Contract Security Audit

Prepared by ShellBoxes

Feb 13th, 2023 – Feb 14th, 2023

Shellboxes.com

contact@shellboxes.com

Document Properties

Client	Sukiyaki Finance
Version	1.0
Classification	Public

Scope

Contract Name	Contact Address
Sukiyaki	0xD212046F89680aC9F106B9c63f314cc9808e18d5

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1 Introduction

Sukiyaki Finance engaged ShellBoxes to conduct a security assessment on the Sukiyaki Finance beginning on Feb 13th, 2023 and ending Feb 14th, 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. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About Sukiyaki Finance

Sukiyaki - The world's first AI powered DEX aggregator, uses custom built AI to route for best prices on chain and cross chain for users, thereby gaining points ahead of the competition over 65% of the time on Ethereum, Binance Smart Chain, Arbitrum and Polygon.

Issuer	Sukiyaki Finance
Website	`https://sukiyaki.finance`
Type	Solidity Smart Contract
Whitepaper	Sukiyaki Official White paper
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 Disclaimer

The Sukiyaki Finance team has decided to acknowledge our findings and not proceed with the fixes due to the fact that the contract is renounced and the liquidity pool is locked. The contract renouncing is based on their community's demand post the Fair-launch with the goal of building their community's trust.

2.2 Summary

The following is a synopsis of our conclusions from our analysis of the Sukiyaki Finance 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.

2.3 Key Findings

In general, these smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include , 2 high-severity, 7 medium-severity, 4 low-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Missing critical return value check	HIGH	Acknowledged
SHB.2. The tokensForBurn is set to zero without burning tokens	HIGH	Acknowledged
SHB.3. Missing fee and max transaction exclusion update in the transferOwnership	MEDIUM	Acknowledged
SHB.4. The owner has total control over the contract funds	MEDIUM	Acknowledged

SHB.5. Perfoming divisions before multiplications reduces precision	MEDIUM	Acknowledged
SHB.6. Missing fee limitation	MEDIUM	Acknowledged
SHB.7. Centralization Risk	MEDIUM	Acknowledged
SHB.8. Transaction Order Dependency	MEDIUM	Acknowledged
SHB.9. Centralized token allocation	MEDIUM	Acknowledged
SHB.10. Approve race condition	LOW	Acknowledged
SHB.11. Using .transfer() to transfer Ether	LOW	Acknowledged
SHB.12. The early buy penalty is not documented	LOW	Acknowledged
SHB.13. The owner can renounce ownership	LOW	Acknowledged

3 Finding Details

SHB.1 Missing critical return value check

- Severity : HIGH
- Status : Acknowledged

- Likelihood : 2
- Impact : 3

Description:

The swapBack function can be called either by the owner or by the transfer function if certain conditions are met. This function distributes the contract funds that were collected from the transfer fees to the devAddress, the marketingAddress, and the liquidity pools. This function is missing a critical check over the return value of the call function that is used for transferring the eth value. Therefore, if the call fails the transaction will not revert and both the variables tokensForDev and tokensForMarketing will be set to zero, which implies that the devAddress and the marketingAddress will permanently lose those funds.

Files Affected:

SHB.1.1: Sukiyaki.sol

```
tokensForLiquidity = 0;
tokensForMarketing = 0;
tokensForDev = 0;
tokensForBurn = 0;

if(liquidityTokens > 0 && ethForLiquidity > 0){
    addLiquidity(liquidityTokens, ethForLiquidity);
}

(success,) = address(devAddress).call{value: ethForDev}("");

```

```
(success,) = address(marketingAddress).call{value: address(this).balance
    ↪ }("");
```

Recommendation:

The trivial recommendation will be to use require statements to assure the success variable is equal to true. However, this can introduce a new risk where the devAddress and the marketingAddress can revert any call to the swapBack function, causing a DoS. Consider using the _safeTransferETHWithFallback function for transferring ether to avoid all the risks:

SHB.1.2: Sukiyaki.sol

```
function _safeTransferETHWithFallback(address to, uint256 amount)
    ↪ internal {
    if (!_safeTransferETH(to, amount)) {
        WETH.deposit{value: amount}();
        WETH.transfer(to, amount);
    }
}
function _safeTransferETH(address to, uint256 amount)
    internal
    returns (bool)
{
    (bool success, ) = to.call{value: amount}(new bytes(0));
    return success;
}
```

SHB.2 The tokensForBurn is set to zero without burning tokens

- Severity : HIGH
- Status : Acknowledged
- Likelihood : 3
- Impact : 2

Description:

In the swapBack function, the tokensForBurn are supposed to be burned whenever the value is greater than zero. When the contract balance is lower than the tokensForBurn, the variable is set to zero without burning any tokens, which represents a critical innaccuracy in the contract logic.

Files Affected:

SHB.2.1: Sukiyaki.sol

```
if(tokensForBurn > 0 && balanceOf(address(this)) >= tokensForBurn) {
    _burn(address(this), tokensForBurn);
}
tokensForBurn = 0;
```

Recommendation:

Consider setting the tokensForBurn to zero only when there are enough tokens in the contract. When the balance is lower than the tokensForBurn, consider either burning the available balance and decrementing the variable or skipping the burning step without updating the variable and the tokensForBurn will be burned in the future calls when there is enough balance to fulfill the requirement.

SHB.3 Missing fee and max transaction exclusion update in the transferOwnership

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

Description:

The Sukiyaki contract makes use of the Ownable contract to allow the owner to perform privileged actions. By default, the Sukiyaki contract excludes the owner from the fees and the restriction over the max transaction amount. However, the transferOwnership function is not adapted to this behavior, changing the owner will keep the old owner excluded and the new owner will not have these exclusions.

Files Affected:

SHB.3.1: Sukiyaki.sol

```
function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero
       ↪ address");
    emit OwnershipTransferred(_owner, newOwner);
    _owner = newOwner;
}
```

Recommendation:

Consider overriding the transferOwnership function and implementing the exclusion update by including the old owner back in the fees ,the max transaction amount restriction ,and excluding the new owner from these restrictions.

SHB.4 The owner has total control over the contract funds

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

Description:

The transferForeignToken and the withdrawStuckETH functions allows the owner to withdraw any amount of tokens or ether from the contract. This represents a significant centralization risk over the contract's funds, where the owner have total control over the contract balance which is a shared ressource between the dev, marketing, and the liquidity pools.

Files Affected:

SHB.4.1: Sukiyaki.sol

```
function transferForeignToken(address _token, address _to) external
    ↪ onlyOwner returns (bool _sent) {
    require(_token != address(0), "_token address cannot be 0");
    require(_token != address(this), "Can't withdraw native tokens");
    uint256 _contractBalance = IERC20(_token).balanceOf(address(this));
    _sent = IERC20(_token).transfer(_to, _contractBalance);
    emit TransferForeignToken(_token, _contractBalance);
}
```

SHB.4.2: Sukiyaki.sol

```
function withdrawStuckETH() external onlyOwner {
    bool success;
    (success,) = address(msg.sender).call{value: address(this).balance
        ↪ }("");
}
```

Recommendation:

Consider removing the following functions or setting some limitation over them to reduce the risk.

SHB.5 Perfoming divisions before multiplications reduces precision

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

Description:

The result of integer division in solidity is an integer value. As a result, dividing before multiplying will result in inaccurate results, and loss of precision.

Files Affected:

SHB.5.1: ContractName.sol

```
if(earlyBuyPenaltyInEffect() && automatedMarketMakerPairs[from] && !automatedMarketMakerPairs[to] && buyTotalFees > 0){

    if(!boughtEarly[to]){
        boughtEarly[to] = true;
        botsCaught += 1;
        emit CaughtEarlyBuyer(to);
    }

    fees = amount * 99 / 100;
    tokensForLiquidity += fees * buyLiquidityFee / buyTotalFees;
    tokensForMarketing += fees * buyMarketingFee / buyTotalFees;
    tokensForDev += fees * buyDevFee / buyTotalFees;
    tokensForBurn += fees * buyBurnFee / buyTotalFees;
}

// on sell
else if (automatedMarketMakerPairs[to] && sellTotalFees > 0){
    fees = amount * sellTotalFees / 100;
    tokensForLiquidity += fees * sellLiquidityFee / sellTotalFees;
    tokensForMarketing += fees * sellMarketingFee / sellTotalFees;
    tokensForDev += fees * sellDevFee / sellTotalFees;
    tokensForBurn += fees * sellBurnFee / sellTotalFees;
}

// on buy
else if(automatedMarketMakerPairs[from] && buyTotalFees > 0) {
    fees = amount * buyTotalFees / 100;
    tokensForLiquidity += fees * buyLiquidityFee / buyTotalFees;
    tokensForMarketing += fees * buyMarketingFee / buyTotalFees;
    tokensForDev += fees * buyDevFee / buyTotalFees;
    tokensForBurn += fees * buyBurnFee / buyTotalFees;
}
```

Recommendation:

Consider performing multiplication operations before divisions to improve the calculation's precision.

SHB.6 Missing fee limitation

- Severity : MEDIUM
- Status : Acknowledged

- Likelihood : 1
- Impact : 3

Description:

The owner is the one responsible for modifying the buy and sell fees using the updateBuyFees and the updateSellFees respectively. These functions lack a limitation over the fee value, this allows the owner to specify any amount as a fee, which can break the structure of the contract and result in unexpected results for the users.

Files Affected:

SHB.6.1: Sukiyaki.sol

```
function updateBuyFees(uint256 _marketingFee, uint256 _liquidityFee,
   ↪ uint256 _DevFee, uint256 _burnFee) external onlyOwner {
    buyMarketingFee = _marketingFee;
    buyLiquidityFee = _liquidityFee;
    buyDevFee = _DevFee;
    buyBurnFee = _burnFee;
    buyTotalFees = buyMarketingFee + buyLiquidityFee + buyDevFee +
       ↪ buyBurnFee;
    require(buyTotalFees <= 35, "Must keep fees at 35% or less");
}
```

SHB.6.2: Sukiyaki.sol

```
function updateSellFees(uint256 _marketingFee, uint256 _liquidityFee,
   ↪ uint256 _DevFee, uint256 _burnFee) external onlyOwner {
    sellMarketingFee = _marketingFee;
    sellLiquidityFee = _liquidityFee;
    sellDevFee = _DevFee;
    sellBurnFee = _burnFee;
    sellTotalFees = sellMarketingFee + sellLiquidityFee + sellDevFee +
       ↪ sellBurnFee;
    require(sellTotalFees <= 35, "Must keep fees at 35% or less");
}
```

Recommendation:

It is recommended to limit the fees to a reasonable amount in order to provide a guarantee for the users and to prevent any unexpected outputs.

SHB.7 Centralization Risk

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

Description:

Using the onlyOwner modifier on almost all functions creates a centralization problem by allowing the owner to have complete control over the functionality of the contract which can potentially lead to misuse or abuse of power.

Recommendation:

To address this issue, it's important to implement more decentralized and democratic approaches to decision-making, such as multi-signature control or community governance models that distribute power more evenly.

SHB.8 Transaction Order Dependency

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

Description:

A race condition vulnerability occurs when code depends on the order of the transactions submitted to it. The project contains some modifiable variables that might be impacted by the execution order of the transaction.

Files Affected:

SHB.8.1: Sukiyaki.sol

```
function updateBuyFees(uint256 _marketingFee, uint256 _liquidityFee,
    ↪ uint256 _DevFee, uint256 _burnFee) external onlyOwner {
    buyMarketingFee = _marketingFee;
    buyLiquidityFee = _liquidityFee;
    buyDevFee = _DevFee;
    buyBurnFee = _burnFee;
    buyTotalFees = buyMarketingFee + buyLiquidityFee + buyDevFee +
        ↪ buyBurnFee;
    require(buyTotalFees <= 35, "Must keep fees at 35% or less");
}
```

SHB.8.2: Sukiyaki.sol

```
function updateSellFees(uint256 _marketingFee, uint256 _liquidityFee,
    ↪ uint256 _DevFee, uint256 _burnFee) external onlyOwner {
    sellMarketingFee = _marketingFee;
    sellLiquidityFee = _liquidityFee;
    sellDevFee = _DevFee;
    sellBurnFee = _burnFee;
    sellTotalFees = sellMarketingFee + sellLiquidityFee + sellDevFee +
        ↪ sellBurnFee;
    require(sellTotalFees <= 35, "Must keep fees at 35% or less");
}
```

Recommendation:

Consider adding the fees as arguments then adding a require statement to verify the arguments to be equal to the fees stored in the contract, or consider notifying the community with any change in terms of the fees to mitigate the risk.

SHB.9 Centralized token allocation

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

Description:

In the constructor, the owner of the contract mints all the total supply to his address. This represents a significant centralization risk where the deployer has too much power over the total supply of the token.

Files Affected:

SHB.9.1: Sukiyaki.sol

```
_createInitialSupply(newOwner, totalSupply);
```

Recommendation:

It is recommended to use a DAO or a multisig as the deployer of the contract to include multiple parties in the supply allocation.

SHB.10 Approve race condition

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

Description:

The standard ERC20 implementation contains a widely known racing condition in its approve function.

Exploit Scenario:

A spender can witness the token owner broadcast a transaction altering their approval and quickly sign and broadcast a transaction using transferFrom to move the current approved amount from the owner's balance to the spender. If the spender's transaction is validated before the owner's, the spender will be able to get both approval amounts of both transactions.

Files Affected:

SHB.10.1: Sukiyaki.sol

```
function approve(address spender, uint256 amount) public virtual
   ↪ override returns (bool) {
    _approve(_msgSender(), spender, amount);
    return true;
}
```

Recommendation:

We recommend using increaseAllowance and decreaseAllowance functions to modify the approval amount instead of using the approve function to modify it.

SHB.11 Using .transfer() to transfer Ether

- Severity : LOW
- Status : Acknowledged

- Likelihood : 1
- Impact : 2

Description:

Although transfer() and send() are recommended as a security best-practice to prevent reentrancy attacks because they only forward 2300 gas, the gas repricing of opcodes may break deployed contracts.

Files Affected:

SHB.11.1: Sukiyaki.sol

```
function transferForeignToken(address _token, address _to) external
    ↪ onlyOwner returns (bool _sent) {
        require(_token != address(0), "_token address cannot be 0");
        require(_token != address(this), "Can't withdraw native tokens");
        uint256 _contractBalance = IERC20(_token).balanceOf(address(this)
            ↪ );
        _sent = IERC20(_token).transfer(_to, _contractBalance);
        emit TransferForeignToken(_token, _contractBalance);
    }
```

Recommendation:

Consider using .call value: ... ("") instead, without hardcoded gas limits along with checks-effects-interactions pattern or reentrancy guards for reentrancy protection.

SHB.12 The early buy penalty is not documented

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

Description:

The contract implements a penalty for the bots/snipers where it takes 99% of their transferred amount as a fee if they perform a transfer before a certain blockForPenaltyEnd. This behavior is not documented, this can result in a loss to legitimate users.

Files Affected:

SHB.12.1: Sukiyaki.sol

```
if(earlyBuyPenaltyInEffect() && automatedMarketMakerPairs[from] && !automatedMarketMakerPairs[to] && buyTotalFees > 0){

    if(!boughtEarly[to]){
        boughtEarly[to] = true;
        botsCaught += 1;
        emit CaughtEarlyBuyer(to);
    }

    fees = amount * 99 / 100;
    tokensForLiquidity += fees * buyLiquidityFee / buyTotalFees;
    tokensForMarketing += fees * buyMarketingFee / buyTotalFees;
    tokensForDev += fees * buyDevFee / buyTotalFees;
    tokensForBurn += fees * buyBurnFee / buyTotalFees;
}
```

Recommendation:

It is recommend to document this behavior to notify the community and inform them about this risk.

SHB.13 The owner can renounce ownership

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

Description:

Typically, the account that deploys the contract is also its owner. Consequently, the owner is able to engage in certain privileged activities in his own name. In the Ownable contract, the renounceOwnership function is used to renounce ownership, which means that if the contract's ownership has never been transferred, it will never have an Owner, rendering some owner-exclusive functionality unavailable.

Files Affected:

SHB.13.1: Sukiyaki.sol

```
function renounceOwnership() external virtual onlyOwner {
    emit OwnershipTransferred(_owner, address(0));
    _owner = address(0);
}
```

Recommendation:

We recommend that you prevent the owner from calling renounceOwnership without first transferring ownership to a different address. Additionally, if you decide to use a multi-signature wallet, then the execution of the renounceOwnership will require for at least two or more users to be confirmed. Alternatively, you can disable Renounce Ownership functionality by overriding it.

4 Best Practices

BP.1 The functions and state variables lack documentation

Description:

The contract's state variables lack documentation which makes it harder for a reader to understand the logic and leaves the room for multiple false assumptions about the intended behaviors. It is recommended to improve the code's documentation to have a more readable and well structured code.

Files Affected:

BP.1.1: Sukiyaki.sol

```
uint256 public buyTotalFees;
uint256 public buyMarketingFee;
uint256 public buyLiquidityFee;
uint256 public buyDevFee;
uint256 public buyBurnFee;

uint256 public sellTotalFees;
uint256 public sellMarketingFee;
uint256 public sellLiquidityFee;
uint256 public sellDevFee;
uint256 public sellBurnFee;

uint256 public tokensForMarketing;
uint256 public tokensForLiquidity;
uint256 public tokensForDev;
uint256 public tokensForBurn;
```

Status - Acknowledged

BP.2 Remove unnecessary transferOwnership call

Description:

In the constructor, the Ownable contract already sets the owner as the deployer of the contract, therefore transferring the ownership to the deployer is redundant. It is recommended to remove this redundant call.

Files Affected:

BP.2.1: Sukiyaki.sol

```
transferOwnership(newOwner);
```

Status - Acknowledged

BP.3 Remove unnecessary initializations

Description:

In solidity, there is no need to initialize a variable with its default value, this is done automatically after the variable declaration.

Files Affected:

BP.3.1: Sukiyaki.sol

```
uint256 public tradingActiveBlock = 0; // 0 means trading is not active
uint256 public blockForPenaltyEnd = 0;
```

BP.3.2: Sukiyaki.sol

```
bool public tradingActive = false;
bool public swapEnabled = false;
```

BP.3.3: Sukiyaki.sol

```
buyLiquidityFee = 0;
```

BP.3.4: Sukiyaki.sol

```
buyBurnFee = 0;
```

BP.3.5: Sukiyaki.sol

```
uint256 fees = 0;
```

Status - Acknowledged

BP.4 Use revert statements instead of require

Description:

It is recommended to use revert statements to throw errors when there are invalid conditions as it costs less gas than the require statements.

Status - Acknowledged

BP.5 Use pre-increment instead of post-increment

Description:

i++ is generally more expensive because it must increment a value and "return" the old value, so it may require holding two numbers in memory. ++i only ever uses one number in memory. Therfore, ++i consumes less Gas than i++.

Files Affected:

BP.5.1: Sukiyaki.sol

```
function massManageBoughtEarly(address[] calldata wallets, bool flag)
    ↪ external onlyOwner {
    for(uint256 i = 0; i < wallets.length; i++){
        boughtEarly[wallets[i]] = flag;
    }
}
```

Status - Acknowledged

BP.6 Emiting events should be done after state modifications

Description:

In the renounceOwnership and transferOwnership functions, the OwnershipTransferred event is emmited before the _owner variable is initialized.

Files Affected:

BP.6.1: Sukiyaki.sol

```
function renounceOwnership() external virtual onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
}

function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero
        ↪ address");
    emit OwnershipTransferred(_owner, newOwner);
    _owner = newOwner;
}
```

Status - Acknowledged

BP.7 Remove Unnecessary require statements

Description:

The require statements in the returnToNormalTax function are redundant knowing that the sellTotalFees and the buyTotalFees will be equal to 5 anyway ,since the fee values are hard-coded, therefore; the verification of the values is not necessary.

Files Affected:

BP.7.1: Sukiyaki.sol

```
function returnToNormalTax() external onlyOwner {
    sellMarketingFee = 4;
    sellLiquidityFee = 0;
    sellDevFee = 1;
    sellBurnFee = 0;
    sellTotalFees = sellMarketingFee + sellLiquidityFee + sellDevFee +
        ↪ sellBurnFee;
    require(sellTotalFees <= 10, "Must keep fees at 10% or less");

    buyMarketingFee = 4;
    buyLiquidityFee = 0;
    buyDevFee = 1;
    buyBurnFee = 0;
    buyTotalFees = buyMarketingFee + buyLiquidityFee + buyDevFee +
        ↪ buyBurnFee;
    require(buyTotalFees <= 10, "Must keep fees at 10% or less");
}
```

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

In this audit, we examined the design and implementation of Sukiyaki Finance contract and discovered several issues of varying severity. Sukiyaki Finance team addressed 0 issues raised in the initial report and implemented the necessary fixes, while classifying the rest as a risk with low-probability of occurrence. Shellboxes' auditors advised Sukiyaki Finance Team to maintain a high level of vigilance and to keep those findings in mind in order to avoid any future complications.

7 Scope Files

7.1 Audit

Files	MD5 Hash
Sukiyaki.sol	7a2bfe13dff3386b3071c6c1f880411c

8 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