



# Diamond Swap

Smart Contract Security Audit

Prepared by ShellBoxes

Nov 12<sup>th</sup>, 2022 - Nov 29<sup>th</sup>, 2022

[Shellboxes.com](https://shellboxes.com)

[contact@shellboxes.com](mailto:contact@shellboxes.com)

# Document Properties

<b>Client</b>	Diamond Swap
<b>Version</b>	1.0
<b>Classification</b>	Public

# Scope

<b>Repository</b>	<b>Commit Hash</b>
<a href="https://github.com/DiamondDevTeam/Diamond_Swap_Contracts">https://github.com/DiamondDevTeam/Diamond_Swap_Contracts</a>	aff196707fa532b0ad488a64a4fd9480c3fd1084

# Re-Audit

<b>Repository</b>	<b>Commit Hash</b>
<a href="https://github.com/DiamondDevTeam/Diamond_Swap_Contracts">https://github.com/DiamondDevTeam/Diamond_Swap_Contracts</a>	f15125e685007b63f47584d1f40a6f845f948f04

# Contacts

<b>COMPANY</b>	<b>EMAIL</b>
ShellBoxes	contact@shellboxes.com

# Contents

1	Introduction	5
1.1	About Diamond Swap	5
1.2	Approach & Methodology	5
1.2.1	Risk Methodology	6
2	Findings Overview	7
2.1	Disclaimer	7
2.2	Summary	7
2.3	Key Findings	7
3	Finding Details	9
SHB.1	Loss of Precision Can Lead To All Contributors Not Claiming Their Ethers	9
SHB.2	<code>contribute</code> Function Not Protected	13
SHB.3	Possible DoS Can Lead To Preventing Users From Buying Tokens	15
SHB.4	Non Fixed Price Fixed to <code>5000000</code>	19
SHB.5	Missing Access Control On <code>diamondTransfer</code>	21
SHB.6	The Upgrading Mechanism Is Not Protected	23
SHB.7	Pool Owner Can Cancel The Pool And Retrieve Tokens At Any Moment	24
SHB.8	Executing Multiple Operations On Non-Existent Pools	26
SHB.9	Overriding The Social Handle Is Possible	33
SHB.10	Privacy Issues For Users	34
SHB.11	Admin Can Drain The DiamondSwap Contract	36
SHB.12	Admin Can Add a Duplicate Twitter User	38
SHB.13	The Buyer Can Withdraw Double The Authorized Amount	39
SHB.14	Centralization Power Of The Admin	44
SHB.15	Pool Owner Can Change Visibility Of A Canceled Pool	45
SHB.16	Ether Transfer Failure Can Lead To DoS	47
SHB.17	Race Condition	52
SHB.18	Missing Percentage Check	53
SHB.19	Loss Of Precision	55
SHB.20	Functions Not Existing In The Interface Or Missing Parameters	56
SHB.21	The <code>initialize</code> function Can Be Front Run	57
SHB.22	For Loop Over Dynamic Array	59

SHB.23	Missing Value Verification . . . . .	60
SHB.24	Missing Address Verification . . . . .	61
SHB.25	No Verification OffChain Done For The Price . . . . .	63
4	Best Practices . . . . .	66
BP.1	Use The <code>Pausable</code> Contract Instead Of <code>AllowClaim</code> . . . . .	66
BP.2	Remove <code>IsVerified</code> From <code>UserInfo</code> Struct . . . . .	66
BP.3	Remove Modifier From <code>getFeeData</code> Function . . . . .	67
BP.4	Redundant Verification On Price Of Sent Ether . . . . .	69
BP.5	Wrong Function Name <code>contributeToOwnedPool</code> . . . . .	70
BP.6	Unnecessary Payable Function <code>claimETH</code> . . . . .	71
BP.7	Redundant/Unnecessary Code . . . . .	71
BP.8	Remove Dead Code . . . . .	72
BP.9	No Need To Add The Token Parameter In The <code>contribute</code> function . . . . .	74
5	Tests . . . . .	77
6	Conclusion . . . . .	78
7	Scope Files . . . . .	79
7.1	Audit . . . . .	79
7.2	Re-Audit . . . . .	79
8	Disclaimer . . . . .	81

# 1 Introduction

Diamond Swap engaged ShellBoxes to conduct a security assessment on the Diamond Swap beginning on Nov 12<sup>th</sup>, 2022 and ending Nov 29<sup>th</sup>, 2022. 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 Diamond Swap

DiamondSwap is a first-of-its-kind utility that provides all crypto investors with a new and lucrative way to buy and sell tokens without affecting the project's chart. When one can sell without adversely affecting the chart, the likelihood of growth is amplified.

Issuer	Diamond Swap
Website	<a href="https://www.diamondswap.co/">https://www.diamondswap.co/</a>
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 Disclaimer

Aside from the issues listed in the findings section, the audit team has encountered multiple compilation errors in the contracts during the audit. Furthermore, the project lacks any unit, integration, or end-to-end testing methodologies that ensure the correctness of the contracts' functionalities, these tests are extremely critical and can help discover multiple bugs before deployment which can save potentially lost funds. In addition, the auditors' team was not given detailed documentation that could have helped in the discovery of further concerns.

## 2.2 Summary

The following is a synopsis of our conclusions from our analysis of the Diamond Swap 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

The smart contracts' implementation might be improved by addressing the discovered flaws, which include **13** critical-severity, **2** high-severity, **5** medium-severity, **4** low-severity, **1** undetermined-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Loss of Precision Can Lead To All Contributors Not Claiming Their Ethers	CRITICAL	Fixed
SHB.2. <code>contribute</code> Function Not Protected	CRITICAL	Fixed
SHB.3. Possible DoS Can Lead To Preventing Users From Buying Tokens	CRITICAL	Fixed

SHB.4. Non Fixed Price Fixed to 5000000	CRITICAL	Fixed
SHB.5. Missing Access Control On diamondTransfer	CRITICAL	Fixed
SHB.6. The Upgrading Mechanism Is Not Protected	CRITICAL	Fixed
SHB.7. Pool Owner Can Cancel The Pool And Retrieve Tokens At Any Moment	CRITICAL	Fixed
SHB.8. Executing Multiple Operations On Non-Existent Pools	CRITICAL	Fixed
SHB.9. Overriding The Social Handle Is Possible	CRITICAL	Fixed
SHB.10. Privacy Issues For Users	CRITICAL	Acknowledged
SHB.11. Admin Can Drain The DiamondSwap Contract	CRITICAL	Acknowledged
SHB.12. Admin Can Add a Duplicate Twitter User	CRITICAL	Fixed
SHB.13. The Buyer Can Withdraw Double The Authorized Amount	CRITICAL	Mitigated
SHB.14. Centralization Power Of The Admin	HIGH	Acknowledged
SHB.15. Pool Owner Can Change Visibility Of A Canceled Pool	HIGH	Fixed
SHB.16. Ether Transfer Failure Can Lead To DoS	MEDIUM	Partially Fixed
SHB.17. Race Condition	MEDIUM	Acknowledged
SHB.18. Missing Percentage Check	MEDIUM	Fixed
SHB.19. Loss Of Precision	MEDIUM	Fixed
SHB.20. Functions Not Existing In The Interface Or Missing Parameters	MEDIUM	Fixed
SHB.21. The initialize function Can Be Front Run	LOW	Acknowledged
SHB.22. For Loop Over Dynamic Array	LOW	Fixed
SHB.23. Missing Value Verification	LOW	Fixed
SHB.24. Missing Address Verification	LOW	Partially Fixed
SHB.25. No Verification OffChain Done For The Price	UNDETERMINED	Fixed



# 3 Finding Details

## SHB.1 Loss of Precision Can Lead To All Contributors Not Claiming Their Ethers

- Severity: **CRITICAL**
- Likelihood: 3
- Status: Fixed
- Impact: 3

### Description:

In the `diamondTransfer` function of the `publicPool` contract, each contributor/seller's `claimableETH` will be calculated using the `contributorPercent`, however the calculation is affected by a loss of precision. If `_contributorAmount * 100` is less than `amount` the `contributorPercent` will be equal to 0 ;therefore, the `contributorETH` will also be equal to 0 and the `claimableETH` will not be incremented. Thus, sellers will not collect any ether from this operation.

### Exploit Scenario:

#### 1<sup>st</sup> scenario

1. The list of contributor's amount is the following `[10,60,50,2000,8000]`
2. A buyer wants to buy `10_000` tokens.
3. The contributor percent will be the following `[0,0,0,20,78]` and the sum of this is different than `100%`.

#### 2<sup>nd</sup> scenario

1. The list of contributor's amount is the following `[9,9,9,....,9]`
2. A buyer wants to buy `1_000` tokens.
3. The contributor percent will be the following `[0,0,0,....,0]` and the sum of this is different than `100%`. Thus, all the contributors will not claim any ethers.

## Files Affected:

### SHB.1.1: publicPool.sol

```
77  function diamondTransfer(  
78      address to, // Buyer  
79      uint256 amount, // Amount of Tokens  
80      uint256 price, // Price in GWEI  
81      uint256 range // acceptable range in %  
82  ) external payable returns(  
83      uint256 newPrice,  
84      bool success  
85  ) {  
86      require(range <= 100, "Range must be a valid percent");  
87      require(amount <= _balance, "Amount exceeds available tokens");  
88      require(!isHidden, "Pool is currently not available");  
89      checkFeeData();  
90      checkAmounts(price, range, amount);  
91      newPrice = takeFee(price);  
92  
93      uint256 transferAmount = amount; //9_948  
94      uint256 _contributorAmount;  
95      uint256 contributorPercent;  
96      uint256 contributorETH;  
97      address contributorAddress;  
98      while(transferAmount != 0) {  
99          contributorAddress = contributors[counter];  
100         _contributorAmount = contributorAmounts[contributorAddress];  
101         if(_contributorAmount > transferAmount) {  
102             _contributorAmount = transferAmount;  
103             transferAmount = 0;  
104             contributorAmounts[contributorAddress] -=  
105                 ↔ _contributorAmount;  
106         } else {  
107             transferAmount -= _contributorAmount;
```

```

107         contributorAmounts[contributorAddress] = 0;
108         poolContributor[contributorAddress] = false;
109     }
110     contributorPercent = ((_contributorAmount * 100) / amount);
111     contributorETH = ((contributorPercent * newPrice) / 100);

```

## Recommendation:

A change in the architecture may be possible to solve the issue, to mitigate the risk we recommend first having a big number by multiplying the `contributorAmount` by `1018`.

## Updates

The DiamondSwap team resolved the issue by implementing a `pricePerToken` method to calculate contributor ETH earned to avoid rounding errors.

### SHB.1.2: publicPool.sol

```

86 function diamondTransfer(
87     address payable to, // Buyer
88     uint256 amount, // Amount of Tokens
89     string memory resellerCode,
90     uint256 _diamondFee,
91     address _spotAggregator,
92     address payable _DiamondSwapFeeReceiver
93 ) public payable onlyRole(DIAMOND_ADMIN) nonReentrant LockThePool
    ↪ returns(
94     uint256 newPrice,
95     uint256 _amount
96 ) {
97     require(amount <= _balance, "Amount exceeds available tokens");
98     require(!isHidden, "Pool is currently not available");
99     DiamondFee = _diamondFee;
100    spotAggregator = IOracle(address(_spotAggregator));
101    DiamondSwapFeeReceiver = payable(_DiamondSwapFeeReceiver);
102    checkFeeData(resellerCode, to);

```

```

103     address _to = to;
104     checkAmounts(msg.value, amount);
105     newPrice = takeFee(msg.value);
106
107     uint256 transferAmount = amount;
108     uint256 totalETHSent;
109     uint256 pricePerToken = (newPrice * 10**18) / amount;
110     uint256 returnETH;
111
112     for(uint loopCounter = 0; transferAmount != 0 && loopCounter <= 100;
113         ↪ loopCounter++) {
114         uint256 _contributorAmount;
115         uint256 contributorETH;
116         address contributorAddress = contributors[counter];
117         _contributorAmount = contributorAmounts[contributorAddress];
118         if(_contributorAmount > transferAmount) {
119             _contributorAmount = transferAmount;
120             transferAmount = 0;
121             contributorAmounts[contributorAddress] -= _contributorAmount;
122         } else {
123             transferAmount -= _contributorAmount;
124             contributorAmounts[contributorAddress] = 0;
125             poolContributor[contributorAddress] = false;
126         }
127         contributorETH = _contributorAmount * pricePerToken;
128         totalETHSent += contributorETH;
129         _balance -= _contributorAmount;
130         _diamondSwap.deposit(contributorAddress, contributorETH, address(
131             ↪ _token), address(this));
132         _diamondSwap.updatePublicAmount(_contributorAmount, address(
133             ↪ _token), address(this), _to, contributorAddress);
134         _diamondSwap.updateBuyerSeller(contributorETH, address(_token),
135             ↪ _to, address(this));
136         if(contributorAmounts[contributorAddress] == 0) {

```

```

133         contributors[counter] = address(0);
134         counter++;
135     }
136     if(loopCounter == 100) {
137         returnETH = msg.value - totalETHSent;
138     }
139 }
140 _amount = amount - transferAmount;
141 (bool success, ) = payable(_DiamondInterface).call{value: newPrice
    ↵ }("");
142     require(success, "Failed to send ETH");
143     if(returnETH > 0) {
144         (success, ) = payable(_to).call{value: (returnETH)}("ETH Returned
    ↵ to Sender");
145         require(success, "Failed to send ETH");
146     }
147     IERC20(_token).safeTransfer(_to, (_amount));
148     if(_balance == 0) {
149         isHidden = true;
150     }
151
152     return(
153         totalETHSent,
154         _amount
155     );
156 }

```

## SHB.2 **contribute** Function Not Protected

- Severity: **CRITICAL**
- Likelihood: 3
- Status: Fixed
- Impact: 3

## Description:

In the `publicPool` contract, the `contribute` function is an external function that any user can call, it is not protected by any access control, and it does not transfer any tokens from the user to the pool.

## Exploit Scenario:

1. A malicious user will call `contribute` function with his address and a huge `tokenAmount`.
2. A malicious user can sell tokens because he is added to the list of contributors without sending any tokens.

## Files Affected:

### SHB.2.1: publicPool.sol

```
214     function contribute(  
215         uint256 tokenAmount,  
216         address token,  
217         address user  
218     ) external {  
219         require(address(token) == address(_token), "Must deposit the  
           ↪ correct token into this pool");  
220         _balance += tokenAmount;  
221         if(poolContributor[user]) {  
222             contributorAmounts[user] += tokenAmount;  
223         } else {  
224             poolContributor[user] = true;  
225             contributors.push(user);  
226             contributorCounter[user] = placeInLine;  
227             contributorAmounts[contributors[placeInLine]] += tokenAmount;  
228             placeInLine++;  
229         }  
230         isHidden = false;  
231     }
```

## Recommendation:

It is recommended to add access control to the `contribute` function.

## Updates

The DiamondSwap team resolved the issue by adding access control to the `contribute` function using the `onlyRole(DIAMOND_ADMIN)` modifier.

### SHB.2.2: publicPool.sol

```
254 function contribute(  
255     uint256 tokenAmount,  
256     address user  
257 ) external onlyRole(DIAMOND_ADMIN) {
```

## SHB.3 Possible DoS Can Lead To Preventing Users From Buying Tokens

- Severity: **CRITICAL**
- Status: Fixed
- Likelihood: 3
- Impact: 3

## Description:

When buying an amount of tokens using the `buyFromPool` function, the tokens are sold using a FIFO system between contributors. The protocol loops over all contributors until the order is fulfilled. The issue here is that if we loop many contributors until we reach the gas limit, the transaction will fail. A malicious user can exploit this weakness to prevent any user buying from a pool.

## Exploit Scenario:

1. Malicious user calls the `contributeToOwnedPool` with a `tokenAmount` of 1 multiple times (ex. more than 100).

2. A legit user wants to buy from the pool, assuming that he wants to buy more than `100/10decimals` he will loop over the 100 first contributors, reach the gas limit and the transaction will fail.

The worst-case scenario of this is that by contributing 0 tokens to the pool, the contract doesn't validate the amount, and we can inject our array with multiple 0 therefore causing the DoS without spending any tokens.

## Files Affected:

### SHB.3.1: publicPool.sol

```
77  function diamondTransfer(  
78      address to, // Buyer  
79      uint256 amount, // Amount of Tokens  
80      uint256 price, // Price in GWEI  
81      uint256 range // acceptable range in %  
82  ) external payable returns(  
83      uint256 newPrice,  
84      bool success  
85  ) {  
86      require(range <= 100, "Range must be a valid percent");  
87      require(amount <= _balance, "Amount exceeds available tokens");  
88      require(!isHidden, "Pool is currently not available");  
89      checkAmounts(price, range, amount);  
90      newPrice = takeFee(price);  
91  
92      uint256 transferAmount = amount;  
93      uint256 _contributorAmount;  
94      uint256 contributorPercent;  
95      uint256 contributorETH;  
96      address contributorAddress;  
97      while(transferAmount != 0) {  
98          contributorAddress = contributors[counter];
```



## SHB.3.2: manyToMany.sol

```
80     function diamondTransfer(  
81         address to, // Buyer  
82         uint256 amount, // Amount of Tokens  
83         uint256 price, // Price in GWEI  
84         uint256 range // acceptable range in %  
85     ) external payable nonReentrant onlyRole(DIAMOND_ADMIN) returns(  
86         uint256,  
87         bool  
88     ) {  
89         require(range <= 100, "Range must be a valid percent");  
90         require(amount <= _balance, "Transfer amount must be equal to the  
           ↪ contract balance");  
91         checkFeeData();  
92         amount *= 10**Decimals;  
93         uint256 transferAmount = amount;  
94  
95         checkAmounts(price, range, amount);  
96         uint256 newPrice = takeFee(price);  
97  
98         if(contributorAmounts[contributors[counter]] < amount) {  
99  
100             while(contributorAmounts[contributors[counter]] <=  
                   ↪ transferAmount) {
```

### Recommendation:

A change in the architecture may be required to solve the issue, however to mitigate the risk consider limiting the number of loops.

### Updates

The DiamondSwap team resolved the issue by limiting the number of loop iterations to **100**.

### SHB.3.3: publicPool.sol

```
112 for(uint loopCounter = 0; transferAmount != 0 && loopCounter <= 100;
    ↪ loopCounter++) {
113     uint256 _contributorAmount;
114     uint256 contributorETH;
115     address contributorAddress = contributors[counter];
116     _contributorAmount = contributorAmounts[contributorAddress];
117     if(_contributorAmount > transferAmount) {
118         _contributorAmount = transferAmount;
119         transferAmount = 0;
120         contributorAmounts[contributorAddress] -= _contributorAmount;
121     } else {
122         transferAmount -= _contributorAmount;
123         contributorAmounts[contributorAddress] = 0;
124         poolContributor[contributorAddress] = false;
125     }
126     contributorETH = _contributorAmount * pricePerToken;
127     totalETHSent += contributorETH;
128     _balance -= _contributorAmount;
129     _diamondSwap.deposit(contributorAddress, contributorETH, address(
        ↪ _token), address(this));
130     _diamondSwap.updatePublicAmount(_contributorAmount, address(_token),
        ↪ address(this), _to, contributorAddress);
131     _diamondSwap.updateBuyerSeller(contributorETH, address(_token), _to,
        ↪ address(this));
132     if(contributorAmounts[contributorAddress] == 0) {
133         contributors[counter] = address(0);
134         counter++;
135     }
136     if(loopCounter == 100) {
137         returnETH = msg.value - totalETHSent;
138     }
139 }
```

## SHB.4 Non Fixed Price Fixed to 5000000

- Severity: **CRITICAL**
- Likelihood: 3
- Status: Fixed
- Impact: 3

### Description:

In the `checkAmounts` function, the contract is verifying if there is no `fixedPrice` for the pool, however after doing so, we are fixing the price `weiPerToken` which contradicts the logic of `fixedPrice`.

### Files Affected:

#### SHB.4.1: ownedPool.sol

```
165     function checkAmounts(  
166         uint256 price,  
167         uint256 range,  
168         uint256 amount  
169     ) internal view {  
170  
171         uint256 weiPerToken;  
172         uint256 tokenAmount;  
173         uint256 rangeAmount;  
174  
175         if(!fixedPrice) {  
176             // Comparing spot price to passed price  
177             //(uint256 weiPerToken, ) = spotAggregator.getRate(IERC20(WETH),  
178                 ↪ IERC20(_token), IERC20(zeroAddress));  
179             weiPerToken = 5000000;
```

### Recommendation:

Consider removing `weiPerToken = 5000000` and having the correct logic of the price.

## Updates

The DiamondSwap team resolved the issue by getting the `weiPerToken` from the `spotAggregator` instead of hard-coding it in the contract.

### SHB.4.2: ownedPool.sol

```
168 function checkAmounts(  
169     uint256 price,  
170     uint256 amount  
171 ) internal view {  
172     uint256 weiPerToken;  
173     uint256 tokenAmount;  
174     uint256 rangeAmount;  
175  
176     if(!fixedPrice) {  
177         // Comparing spot price to passed price  
178         (weiPerToken, ) = spotAggregator.getRate(IERC20(WETH), IERC20(_token  
179             ↪ ), IERC20(address(0)));  
179         weiPerToken -= ((weiPerToken * discountPercent) / 1000);
```

### SHB.4.3: publicPool.sol

```
158 function checkAmounts(  
159     uint256 price,  
160     uint256 amount  
161 ) internal pure {  
162     uint256 weiPerToken;  
163     uint256 tokenAmount;  
164     uint256 rangeAmount;  
165     // Comparing spot price to passed price  
166     (weiPerToken, ) = spotAggregator.getRate(IERC20(WETH), IERC20(_token  
167         ↪ ), IERC20(address(0)));  
167     tokenAmount = price / weiPerToken;  
168     tokenAmount *= 10**18;  
169     rangeAmount = ((amount * 5) / 100);  
170     require((amount - rangeAmount) <= tokenAmount && tokenAmount <= (
```

```
↔ amount + rangeAmount), "Invalid amount, adjust range");
```

## SHB.5 Missing Access Control On `diamondTransfer`

- Severity: **CRITICAL**
- Likelihood: 3
- Status: Fixed
- Impact: 3

### Description:

In the `buyFromPool` function, the `diamondSwap` contract calls the `diamondTransfer` for the specified pool. However, this call in the `publicPool` is not protected and can be called by any user therefore spoofing the `price` variable.

### Exploit Scenario:

1. Malicious user calls the `diamondTransfer` from a `publicPool` with a spoofed price.
2. The malicious user will receive an amount of tokens without paying any fees.

### Files Affected:

#### SHB.5.1: `publicPool.sol`

```
77 function diamondTransfer(  
78     address to, // Buyer  
79     uint256 amount, // Amount of Tokens  
80     uint256 price, // Price in GWEI  
81     uint256 range // acceptable range in %  
82 ) external payable returns(  
83     uint256 newPrice,  
84     bool success  
85 ) {  
86     require(range <= 100, "Range must be a valid percent");
```

```
87     require(amount <= _balance, "Amount exceeds available tokens");
88     require(!isHidden, "Pool is currently not available");
```

## Recommendation:

Consider adding the access control mechanism `onlyRole(DIAMOND_ADMIN)` to `diamondTransfer`.

### SHB.5.2: publicPool.sol

```
77 function diamondTransfer(
78     address to, // Buyer
79     uint256 amount, // Amount of Tokens
80     uint256 price, // Price in GWEI
81     uint256 range // acceptable range in %
82 ) external payable onlyRole(DIAMOND_ADMIN) nonReentrant returns(
83     uint256 newPrice,
84     bool success
85 ) {
86     require(range <= 100, "Range must be a valid percent");
87     require(amount <= _balance, "Amount exceeds available tokens");
88     require(!isHidden, "Pool is currently not available");
```

## Updates

The DiamondSwap team resolved the issue by adding access control to the `diamondTransfer` function using the `onlyRole(DIAMOND_ADMIN)` modifier.

### SHB.5.3: publicPool.sol

```
86 function diamondTransfer(
87     address payable to, // Buyer
88     uint256 amount, // Amount of Tokens
89     string memory resellerCode,
90     uint256 _diamondFee,
91     address _spotAggregator,
92     address payable _DiamondSwapFeeReceiver
```

```

93 ) public payable onlyRole(DIAMOND_ADMIN) nonReentrant LockThePool
    ↪ returns(
94     uint256 newPrice,
95     uint256 _amount
96 ) {

```

## SHB.6 The Upgrading Mechanism Is Not Protected

- Severity: **CRITICAL**
- Likelihood: 3
- Status: Fixed
- Impact: 3

### Description:

Upgrading the implementation of the proxy to `newImplementation` is done using the `upgradeTo` function, this function calls the `_authorizeUpgrade` internal function which validates the access control, based on the documentation:

Function that should revert when `msg.sender` is not authorized to upgrade the contract.

However, in the `ownedPoolContract` the function was overridden without adding any access control, giving the ability to any user to upgrade the contract.

### Files Affected:

#### SHB.6.1: publicPoolContract.sol

```

24     function _authorizeUpgrade(
25         address newImplementation
26     ) internal override {
27     }

```

#### SHB.6.2: ownedPoolContract.sol

```
23     function _authorizeUpgrade(  
24         address newImplementation  
25     ) internal override {  
26     }
```

## Recommendation:

Consider adding an access control mechanism to `_authorizeUpgrade` function.

## Updates

The DiamondSwap team resolved the issue by adding access control to the `_authorizeUpgrade` function using the `onlyOwner` modifier.

### SHB.6.3: publicPoolContract.sol

```
24     function _authorizeUpgrade(  
25         address newImplementation  
26     ) internal override onlyOwner {  
27     }
```

### SHB.6.4: ownedPoolContract.sol

```
23     function _authorizeUpgrade(  
24         address newImplementation  
25     ) internal override onlyOwner {  
26     }
```

## SHB.7 Pool Owner Can Cancel The Pool And Retrieve Tokens At Any Moment

- Severity: **CRITICAL**
- Likelihood: 3
- Status: Fixed
- Impact: 3



## Description:

When a new pool is created through the `createPool` function, based on the `isVestedIsFixedPricelsTwitterIsHiddenPreventCancel`, the `preventCancel` is set to `True` to prevent canceling the pool. However, this check is easy to bypass, therefore a malicious pool owner can cancel his pool and transfer all the tokens from it.

## Exploit Scenario:

1. Pool owner calls the `updatePoolOwner` with the `preventCancel` false.
2. Pool owner calls the `cancelOwnedPool` to retrieve all tokens.

## Files Affected:

### SHB.7.1: DiamondSwap.sol

```
627     function cancelOwnedPool(  
628         address _token,  
629         address poolAddress  
630     ) external {  
631         IDiamondContract(poolAddress).cancelPool(msg.sender);  
632         pools[_token].hidden[poolAddress] = true;  
633  
634         emit DiamondEvents.poolCanceled(poolAddress, msg.sender);  
635     }
```

## Recommendation:

Prevent the pool owner from changing the `preventCancel` variable in the `updatePoolOwner`.

## Updates

The DiamondSwap team resolved the issue by removing the ability to cancel a pool during ownership transfer ,unless `preventCancel` is equal to `false`.

## SHB.7.2: ownedPool.sol

```
350 function updateOwner(  
351     address owner,  
352     address newOwner,  
353     bool _preventCancel  
354 ) external onlyRole(DIAMOND_ADMIN) nonReentrant LockThePool returns(  
355     uint256 amount  
356 ) {  
357     require(address(owner) == address(poolOwner), "Must be pool owner  
        ↳ to update ownership");  
358     poolOwner = newOwner;  
359     if (!preventCancel) {  
360         preventCancel = _preventCancel;  
361     }  
362  
363     return(_balance);  
364 }
```

## SHB.8 Executing Multiple Operations On Non-Existent Pools

- Severity: **CRITICAL**
- Likelihood: 3
- Status: Fixed
- Impact: 3

### Description:

Any external user can call different functions without having any additional permission on non-existent pools, this will have a huge impact on the contract by injecting false information on the contract.

### Exploit Scenario:

1<sup>st</sup> scenario:

1. A malicious user will deploy a malicious contract with a fake payable `diamondTransfer` function that will return `(100*1018,true)`.
2. The malicious user will call the `buyFromPool` with the address of the fake pool recently deployed.
3. The `DiamondSwap` contract will emit an event with fake information `DiamondEvents.tokensPurchased(pool, amount, msg.sender, 100*1018)`.

2<sup>nd</sup> scenario:

1. A malicious user will deploy a malicious contract with a fake `contribute` function that will return `true`.
2. The malicious user will call the `contributeToOwnedPool` with the address of the fake pool recently deployed and the address of a token not existing in `publicTokens`.
3. The `DiamondSwap` contract will add the fake pool to `publicPools` and emit an event with fake information `DiamondEvents.poolContribution(pools[token],publicPoolAddress,tokenAmount)`.
4. A legit user will call the `buyFromPool` using the fake pool address, the malicious user will list his fake pool with an attractive `pricePerTokenWei`.
5. ethers will be transferred directly to the pool and the user won't receive any tokens.

Same issue in `cancelOwnedPool`,`updatePoolVisibility`,`updatePoolOwner` functions.

## Files Affected:

### SHB.8.1: DiamondSwap.sol

```
465     function buyFromPool(  
466         address payable pool,  
467         uint256 amount,  
468         uint256 price,  
469         uint256 range  
470     ) external payable {  
471         require(msg.value == price, "!ETH");
```

```

472
473     (uint256 newPrice, bool success) = IDiamondContract(payable(pool)
        ↪ ).diamondTransfer{value: msg.value}(msg.sender, amount,
        ↪ price, range);
474     require(success, "!GAS");
475
476     emit DiamondEvents.tokensPurchased(pool, amount, msg.sender,
        ↪ newPrice);
477 }

```

### SHB.8.2: DiamondSwap.sol

```

479     function contributeToOwnedPool(
480         uint256 tokenAmount,
481         address token,
482         address pool
483     ) external {
484         IERC20(token).safeTransferFrom(msg.sender, pool, tokenAmount);
485         IDiamondContract(payable(pool)).contribute(tokenAmount, token,
            ↪ msg.sender);
486         pools[token].PublicSaleAmount[pool] += tokenAmount;
487         users[msg.sender].createdPoolAmountContributed[token][pool] +=
            ↪ tokenAmount;
488         pools[token].hidden[pool] = false;
489         if(!publicTokens.contains(token)) {
490             publicTokens.add(token);
491             publicPools.add(pool);
492         }
493
494         emit DiamondEvents.poolContribution(pools[token].
            ↪ publicPoolAddress, tokenAmount);
495     }

```

### SHB.8.3: DiamondSwap.sol

```

627     function cancelOwnedPool(

```

```

628     address _token,
629     address poolAddress
630 ) external {
631     IDiamondContract(poolAddress).cancelPool(msg.sender);
632     pools[_token].hidden[poolAddress] = true;
633
634     emit DiamondEvents.poolCanceled(poolAddress, msg.sender);
635 }

```

#### SHB.8.4: DiamondSwap.sol

```

637     function updatePoolVisibility(
638         address _token,
639         address poolAddress,
640         bool isHidden
641     ) external {
642         IDiamondContract(poolAddress).updateVisibility(msg.sender,
643             ↪ isHidden);
644         pools[_token].hidden[poolAddress] = isHidden;
645     }

```

#### SHB.8.5: DiamondSwap.sol

```

646     function updatePoolOwner(
647         address _token,
648         address _pool,
649         address newOwner,
650         bool preventCancel
651     ) external {
652         uint256 _amount = IDiamondContract(_pool).updateOwner(msg.sender,
653             ↪ newOwner, preventCancel);
654         UpdateDiamondStruct._transferPoolOwner(_token, _pool, _amount,
655             ↪ users[msg.sender], users[newOwner]);
656
657         emit DiamondEvents.poolOwnershipUpdated(_pool, msg.sender,
658             ↪ newOwner);

```

656

```
}
```

## Recommendation:

Have a mapping of verified pools in the `DiamondSwap` contract, in each call/operation verify if the pool exists.

### SHB.8.6: DiamondSwap.sol

```
function buyFromPool(
    address payable pool,
    uint256 amount,
    uint256 price,
    uint256 range
) external payable {
    require(poolsVerified[pool] != address(0), "Inexistent Pool");
    require(msg.value == price, "!ETH");
    (uint256 newPrice, bool success) = IDiamondContract(payable(pool)
        ↪ ).diamondTransfer{value: msg.value}(msg.sender, amount,
        ↪ price, range);
    require(success, "!GAS");

    emit DiamondEvents.tokensPurchased(pool, amount, msg.sender,
        ↪ newPrice);
}
```

## Updates

The `DiamondSwap` team resolved the issue by adding the `isPool` mapping and implementing the existence verification in the pool operations.

### SHB.8.7: DiamondSwap.sol

```
392 function buyFromPool(
393     address payable pool,
394     uint256 amount,
395     string memory resellerCode
```

```

396 ) external payable {
397     require(isPool[pool]);
398
399     (uint256 newPrice, uint256 _amount) = IDiamondContract(payable(pool)
    ↪ ).diamondTransfer{value: msg.value}(msg.sender, amount,
    ↪ resellerCode, PlatformInfo.DiamondFee, PlatformInfo.
    ↪ priceOracle, PlatformInfo.DiamondSwapFeeReceiver);
400
401     emit DiamondEvents.tokensPurchased(pool, _amount, msg.sender,
    ↪ newPrice);
402 }

```

### SHB.8.8: DiamondSwap.sol

```

404 function contributeToPool(
405     uint256 amount,
406     address pool,
407     address contributionOwner
408 ) public payable {
409     require(isPool[pool]);
410     IERC20Upgradeable(IDiamondContract(pool)._token.address).
    ↪ safeTransferFrom(msg.sender, pool, amount);
411     IDiamondContract(payable(pool)).contribute(amount, msg.sender);
412     UpdateDiamondStruct._updatePoolCreator(pools[IDiamondContract(pool).
    ↪ _token.address], IDiamondContract(pool)._token.address, pool,
    ↪ amount, users[contributionOwner], PublicSets, false);
413
414     emit DiamondEvents.poolContribution(pools[IDiamondContract(pool).
    ↪ _token.address].publicPoolAddress, amount);
415 }

```

### SHB.8.9: DiamondSwap.sol

```

501 function cancelOwnedPool(
502     address pool
503 ) external payable {

```

```

504     require(isPool[pool]);
505     IDiamondContract(pool).cancelPool(payable(msg.sender));
506
507     emit DiamondEvents.poolCanceled(pool, msg.sender);
508 }

```

#### SHB.8.10: DiamondSwap.sol

```

510 function updatePoolVisibility(
511     address _token,
512     address pool,
513     bool isHidden
514 ) external payable {
515     require(isPool[pool]);
516     IDiamondContract(pool).updateVisibility(msg.sender, isHidden);
517     pools[_token].hidden[pool] = isHidden;
518 }

```

#### SHB.8.11: DiamondSwap.sol

```

520 function updatePoolOwner(
521     address _token,
522     address _pool,
523     address newOwner
524 ) external payable{
525     require(isPool[_pool]);
526     uint256 _amount = IDiamondContract(_pool).updateOwner(msg.sender,
527         ↪ newOwner, false);
528     UpdateDiamondStruct._transferPoolOwner(_token, _pool, _amount, users
529         ↪ [msg.sender], users[newOwner]);
530
531     emit DiamondEvents.poolOwnershipUpdated(_pool, msg.sender, newOwner)
532         ↪ ;
533 }

```



## SHB.9 Overriding The Social Handle Is Possible

- Severity: **CRITICAL**
- Likelihood : 3
- Status : Fixed
- Impact : 3

### Description:

When calling the `socialAuth` function, an admin can verify a user and also add his handle. The issue here is that there is no verification on the handle if it's already existing, by calling the `socialAuth` with an existing handle first it will override the `userAddress` to the new user address and the second issue is that the old user will still have the `isVerified` attribute and the `verifiedUser` with the same handle ;therefore, having two users with the same handle.

### Files Affected:

#### SHB.9.1: DiamondSwap.sol

```
135     function socialAuth(  
136         string memory handle,  
137         address account  
138     ) external onlyRole(DEFAULT_ADMIN_ROLE) {  
139         userHandles[handle].userAddress = account;  
140         users[account].isVerified = true;  
141         users[account].verifiedUser = handle;
```

### Recommendation:

Consider verifying if `userHandles[handle]` is equal to `address(0)` first, then verify the user.

#### SHB.9.2: DiamondSwap.sol

```
1173     function socialAuth(  
1174         string memory handle,  
1175         address account  
1176     ) external onlyRole(DEFAULT_ADMIN_ROLE) {
```

```

1177     require(userHandles[handle].userAddress == address(0), "Handle
        Already Exist!");
1178     userHandles[handle].userAddress = account;
1179     users[account].isVerified = true;
1180     users[account].verifiedUser = handle;

```

## Updates

The DiamondSwap team resolved the issue by verifying the `userAddress` to be the `address(0)` before setting the `account`.

### SHB.9.3: DiamondSwap.sol

```

125 function socialAuth(
126     string memory handle,
127     address account
128 ) external onlyRole(DEFAULT_ADMIN_ROLE) payable {
129     require(userHandles[handle].userAddress == address(0) && account !=
        ↪ address(0));
130
131     userHandles[handle].userAddress = account;
132     users[account].verifiedUser = handle;
133
134     emit DiamondEvents.influencerVerified(handle, account);
135 }

```

## SHB.10 Privacy Issues For Users

- Severity: **CRITICAL**
- Status: Acknowledged
- Likelihood: 3
- Impact: 3

## Description:

In the DiamondSwap protocol, users can verify their account by providing their twitter handle or other social network and the admin can then manually validate the account and call the `socialAuth` function. This presents a huge risk for influencers that don't want their public address to be leaked.

## Exploit Scenario:

An attacker can listen to `DiamondEvents.influencerVerified(handle, account)` event and map all verified users' handles with their public address.

## Files Affected:

### SHB.10.1: DiamondSwap.sol

```
135 function socialAuth(  
136     string memory handle,  
137     address account  
138 ) external onlyRole(DEFAULT_ADMIN_ROLE) {  
139     userHandles[handle].userAddress = account;  
140     users[account].isVerified = true;  
141     users[account].verifiedUser = handle;  
142     emit DiamondEvents.influencerVerified(handle, account);
```

## Recommendation:

Consider storing only the hash of handle as a bytes32, therefore only the hash will be stored in the contract. If someone wants to transfer to an influencer with the handle, the hash function will be used offchain in the dAPP and the function will receive only the hash, therefore protecting also the identity of the influencer.

## Updates

The DiamondSwap team acknowledged the risk, stating that the goal of the functionality is to provide transparency about influencers' holdings.

## SHB.11 Admin Can Drain The DiamondSwap Contract

- Severity: **CRITICAL**
- Likelihood : 3
- Status : Acknowledged
- Impact : 3

### Description:

The admin can call the `_withdrawLockedFunds` to retrieve the locked funds in the DiamondSwap contract. By having this logic a malicious admin can drain the DiamondSwap using two different methods:

- The first method is calling `_withdrawLockedFunds` for all existing users, therefore sending all funds to the `FraudFundsWallet` and the `claimableETH` of users will be equal to 0.
- The second method is using an existing re-entrancy attack, the `.call` transfer is executed before changing the `claimableETH`, therefore the admin can call the `_withdrawLockedFunds` multiple times until the contract is drained.

### Exploit Scenario:

#### 1<sup>st</sup> Scenario :

1. Admin will retrieve all users that deposited in the contract by searching in the `DiamondEvents.ETHDeposited` event.
2. Admin will call `_lockUser` first then the `_withdrawLockedFunds` and do the same for all users until the contract is drained.

#### 2<sup>nd</sup> Scenario :

1. Admin will deploy a contract which contains in its `fallback` or `receive` functions a call to the `_withdrawLockedFunds`.
2. Admin will add the new contract as an admin of this contract.
3. The new contract will call the `_withdrawLockedFunds` and the transaction will continue until the contract is drained.

## Files Affected:

### SHB.11.1: DiamondSwap.sol

```
87     function _withdrawLockedFunds(  
88         address userWallet  
89     ) external onlyRole(DEFAULT_ADMIN_ROLE) {  
90         require(users[userWallet].locked, "!LOCKED");  
91         (bool success, ) = address(FraudFundsWallet).call{value: users[  
            ↪ userWallet].claimableETH}("");  
92         require(success, "!SEND");  
93         users[userWallet].claimableETH = 0;  
94     }
```

## Recommendation:

- As mentioned in the previous issue, for the centralization issue it's recommended to have a multisig wallet or a DAO.
- For the re-entrancy attack use the checks-effects-interactions design; in this case it's advised to first set the `claimableETH` to 0, then send to the `FraudFundsWallet` or use the Re-Entrancy guard from OpenZeppelin.

### SHB.11.2: DiamondSwap.sol

```
function _withdrawLockedFunds(  
    address userWallet  
) external onlyRole(DEFAULT_ADMIN_ROLE) {  
    require(users[userWallet].locked, "!LOCKED");  
    users[userWallet].claimableETH = 0;  
    (bool success, ) = address(FraudFundsWallet).call{value:  
        ↪ users[userWallet].claimableETH}("");  
    require(success, "!SEND");  
}
```

## Updates

The DiamondSwap team acknowledged the risk, stating that the **ADMIN** role will be held by a multi-sig wallet.

## SHB.12 Admin Can Add a Duplicate Twitter User

- Severity: **CRITICAL**
- Likelihood : 3
- Status : Fixed
- Impact : 3

### Description:

An admin can add a twitter user using the `setTwitterUsers`, however there is no protection to prevent adding duplicate twitter accounts, by doing so the twitter handle will be overridden and the `twitterCounter` will be increased.

### Files Affected:

#### SHB.12.1: ownedPool.sol

```
365     function setTwitterUsers(  
366         string[] memory handles,  
367         uint256[] memory amounts  
368     ) external onlyRole(DIAMOND_ADMIN) {  
369         for(uint256 i = 0; i < handles.length; i++) {  
370             twitterHandle[handles[i]] = amounts[i];  
371             twitterCounter++;  
372         }  
373         twitterReserved = true;  
374     }
```

### Recommendation:

Verify if the `twitterHandle[handles[i]]` is different from 0 then increment the `twitterCounter`.

## Updates

The DiamondSwap team resolved the issue by incrementing the `twitterCounter` only when the handle's amount is equal to zero.

### SHB.12.2: ownedPool.sol

```
365 function setTwitterUsers(  
366     string[] memory handles,  
367     uint256[] memory amounts  
368 ) external onlyRole(DIAMOND_ADMIN) {  
369     for(uint256 i = 0; i < handles.length; i++) {  
370         if(twitterHandle[handles[i]] > 0) {  
371             twitterHandle[handles[i]] += amounts[i];  
372         } else {  
373             twitterHandle[handles[i]] = amounts[i];  
374             twitterCounter++;  
375         }  
376     }  
377     twitterReserved = true;  
378 }
```

## SHB.13 The Buyer Can Withdraw Double The Authorized Amount

- Severity: **CRITICAL**
- Likelihood: 3
- Status: Mitigated
- Impact: 3

### Description:

The `range` functionality allows the buyer to specify an accepted slippage in order to prevent unfavorable trades and unexpected outputs. However, this functionality allows the buyer to buy double the authorized amount associated to the paid price.

## Exploit Scenario:

Let's consider the case where the pool contains  $2 * 10^{18}$  unit of the token for 2ETH, this implies that the price of the token will be 1Wei per unit ( $\text{pricePerTokenWei} = 10^{18}$ ). The attacker can call the `buyFromPool` function with the following arguments:

- $10^{18}$  as the price argument
- 100 as the range argument
- $2 * 10^{18}$  as the amount argument

The function responsible for verifying the price, the amount, and the range argument is `checkAmounts`. Considering the injected arguments, the `tokenAmount` variable will be equal to  $10^{18}$  and the `rangeAmount` will be equal to the amount argument which is  $10^{18}$ , the function verifies the following statement:  $\text{amount} - \text{rangeAmount} \leq \text{tokenAmount} \leq \text{amount} + \text{rangeAmount}$  which will be equivalent in our case to :  $0 \leq 10^{18} \leq 4 * 10^{18}$  Therefore, the require statement verification will pass and the `buyFromPool` function will transfer `amount` which is  $2 * 10^{18}$  units of the token to the buyer. 1ETH is supposed to give the buyer  $10^{18}$  units of the token meanwhile in this scenario he was able to get double the authorized amount.

## Files Affected:

### SHB.13.1: ownedPool.sol

```
155 function checkAmounts(  
156     uint256 price,  
157     uint256 range,  
158     uint256 amount  
159 ) internal view {  
160  
161     uint256 weiPerToken;  
162     uint256 tokenAmount;  
163     uint256 rangeAmount;  
164  
165     if(!fixedPrice) {  
166         // Comparing spot price to passed price
```



```

167     //(uint256 weiPerToken, ) = spotAggregator.getRate(IERC20(WETH),
        ↪ IERC20(_token), IERC20(zeroAddress));
168     weiPerToken = 5000000;
169     weiPerToken -= ((weiPerToken * discountPercent) / 1000);
170     tokenAmount = price / weiPerToken;
171     tokenAmount *= 10**18;
172     rangeAmount = ((amount * range) / 100);
173     require((amount - rangeAmount) <= tokenAmount && tokenAmount <= (
        ↪ amount + rangeAmount), "Invalid amount, adjust range");
174 } else if(fixedPrice) {
175     tokenAmount = (price / pricePerTokenWei);
176     tokenAmount *= 10**18;
177     rangeAmount = ((amount * range) / 100);
178     require ((amount - rangeAmount) <= tokenAmount && tokenAmount <=
        ↪ (amount + rangeAmount), "Invalid amount, adjust range");
179 }
180 }

```

### SHB.13.2: manyToMany.sol

```

129 function checkAmounts(
130     uint256 price,
131     uint256 range,
132     uint256 amount
133 ) internal view {
134
135     // Comparing spot price to passed price
136     //(uint256 EthPerToken, ) = spotAggregator.getRate(IERC20(WETH),
        ↪ IERC20(_token), IERC20(zeroAddress));
137     uint256 EthPerToken = 10000000000000000;
138     uint256 tokenAmount = price / EthPerToken;
139     tokenAmount *= 10**Decimals;
140     uint256 rangeAmount = ((amount * range) / 100);
141     require((amount - rangeAmount) <= tokenAmount && tokenAmount <= (
        ↪ amount + rangeAmount), "Invalid amount, adjust range");

```

```
142 }
```

### SHB.13.3: publicPool.sol

```
131 function checkAmounts(  
132     uint256 price,  
133     uint256 range,  
134     uint256 amount  
135 ) internal pure {  
136     uint256 weiPerToken;  
137     uint256 tokenAmount;  
138     uint256 rangeAmount;  
139     // Comparing spot price to passed price  
140     //(uint256 EthPerToken, ) = spotAggregator.getRate(IERC20(WETH),  
141         ↪ IERC20(_token), IERC20(zeroAddress));  
142     weiPerToken = 5000000;  
143     tokenAmount = price / weiPerToken;  
144     tokenAmount *= 10**18;  
145     rangeAmount = ((amount * range) / 100);  
146     require((amount - rangeAmount) <= tokenAmount && tokenAmount <= (  
147         ↪ amount + rangeAmount), "Invalid amount, adjust range");  
148 }
```

## Recommendation:

Consider using the `tokenAmount` variable as the transferred amount to the buyer, as it is the accurate value calculated based on the price and the price per token.

## Updates

The DiamondSwap team mitigated the risk by fixing the slippage to 5%, this allows the buyer to get 5% more of the authorized amount.

### SHB.13.4: publicPool.sol

```
158 function checkAmounts(  
159     uint256 price,
```

```

160     uint256 amount
161 ) internal pure {
162     uint256 weiPerToken;
163     uint256 tokenAmount;
164     uint256 rangeAmount;
165     // Comparing spot price to passed price
166     //(uint256 EthPerToken, ) = spotAggregator.getRate(IERC20(WETH),
        ↪ IERC20(_token), IERC20(address(0)));
167     weiPerToken = 5000000;
168     tokenAmount = price / weiPerToken;
169     tokenAmount *= 10**18;
170     rangeAmount = ((amount * 5) / 100);
171     require((amount - rangeAmount) <= tokenAmount && tokenAmount <= (
        ↪ amount + rangeAmount), "Invalid amount, adjust range");
172 }

```

### SHB.13.5: ownedPool.sol

```

168 function checkAmounts(
169     uint256 price,
170     uint256 amount
171 ) internal view {
172     uint256 weiPerToken;
173     uint256 tokenAmount;
174     uint256 rangeAmount;
175
176     if(!fixedPrice) {
177         // Comparing spot price to passed price
178         //(uint256 weiPerToken, ) = spotAggregator.getRate(IERC20(WETH),
            ↪ IERC20(_token), IERC20(address(0)));
179         weiPerToken = 5000000;
180         weiPerToken -= ((weiPerToken * discountPercent) / 1000);
181         tokenAmount = price / weiPerToken;
182         tokenAmount *= 10**18;
183         rangeAmount = ((amount * 5) / 100);

```

```

184     require((amount - rangeAmount) <= tokenAmount && tokenAmount <= (
        ↪ amount + rangeAmount), "Invalid amount, adjust range");
185 } else if(fixedPrice) {
186     tokenAmount = (price / pricePerTokenWei);
187     tokenAmount *= 10**18;
188     rangeAmount = ((amount * 5) / 100);
189     require ((amount - rangeAmount) <= tokenAmount && tokenAmount <=
        ↪ (amount + rangeAmount), "Invalid amount, adjust range");
190 }
191 }

```

## SHB.14 Centralization Power Of The Admin

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

### Description:

Many functions in multiple contracts give power to the **ADMIN** role, including upgrading a contract, locking users, withdrawing locked funds ... This can have a serious problem if somehow the private key of the admin is exposed.

### Files Affected:

#### SHB.14.1: DiamondSwap.sol

```

77     function _authorizeUpgrade(
78         address newImplementation
79     ) internal onlyRole(DEFAULT_ADMIN_ROLE) override {
80     }

```

#### SHB.14.2: DiamondSwap.sol

```

80     function _lockUser(

```

```

81     address userWallet,
82     bool isLocked
83 ) external onlyRole(DEFAULT_ADMIN_ROLE) {
84     users[userWallet].locked = isLocked;
85 }

```

### SHB.14.3: DiamondSwap.sol

```

87     function _withdrawLockedFunds(
88         address userWallet
89     ) external onlyRole(DEFAULT_ADMIN_ROLE) {
90         require(users[userWallet].locked, "!LOCKED");
91         (bool success, ) = address(FraudFundsWallet).call{value: users[
92             ↪ userWallet].claimableETH}("");
93         require(success, "!SEND");
94         users[userWallet].claimableETH = 0;
95     }

```

### Recommendation:

Consider having a multisig wallet or a DAO that will have control over these functions.

### Updates

The DiamondSwap team acknowledged the risk stating that the **ADMIN** role will be controlled by a multi-sig wallet.

## SHB.15 Pool Owner Can Change Visibility Of A Canceled Pool

- Severity: **HIGH**
- Likelihood: 2
- Status: Fixed
- Impact: 3

## Description:

The pool owner has the ability to set the visibility of a pool to be not hidden by calling the `updatePoolVisibility` function, even if the pool was already canceled and hidden.

## Exploit Scenario:

1. Pool owner calls the `cancelOwnedPool` and the `pools[_token].hidden[poolAddress]` is set to `true`.
2. Pool owner calls the `updatePoolVisibility` with `isHidden` as `false`.

## Files Affected:

### SHB.15.1: DiamondSwap.sol

```
637 function updatePoolVisibility(  
638     address _token,  
639     address poolAddress,  
640     bool isHidden  
641 ) external {  
642     IDiamondContract(poolAddress).updateVisibility(msg.sender,  
        ↪ isHidden);  
643     pools[_token].hidden[poolAddress] = isHidden;  
644 }
```

## Recommendation:

Consider verifying if the pool was already canceled, if it's the case revert the transaction.

## Updates

The DiamondSwap team resolved the issue by adding a check that verifies the pool to not be cancelled before changing the visibility.

### SHB.15.2: ownerPool.sol

```
398 function updateVisibility(  
        ↪
```

```

399     address user,
400     bool visibility
401 ) external onlyRole(DIAMOND_ADMIN) LockThePool {
402     require(address(user) == address(poolOwner), "Only pool owner can
        ↳ call this function");
403     require(!isCancelled, "This pool no longer exists");
404
405     isHidden = visibility;
406 }

```

## SHB.16 Ether Transfer Failure Can Lead To DoS

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

### Description:

The `takeFee` internal function transfers to `ProjectFeeReceiver` and `ResellerFeeReceiver` the fees associated to the price. A malicious admin will insert the `ProjectFeeReceiver` or the `ResellerFeeReceiver` as a contract and revert on the `fallback/receive` functions. Therefore, causing the transaction to fail completely.

### Files Affected:

#### SHB.16.1: publicPool.sol

```

161     function takeFee(
162         uint256 price
163     ) internal returns(
164         uint256 newPrice
165     ) {
166         uint256 diamondFee = ((price * DiamondFee) / 100);
167         uint256 projectFee = ((diamondFee * ProjectFee) / 100);

```

```

168     uint256 resellerFee = ((diamondFee * ResellerFee) / 100);
169     newPrice = price - diamondFee;
170     diamondFee -= (projectFee + resellerFee);
171     bool success;
172     //Take and distribute fees
173     if(isVerified && isReseller) {
174         (success, ) = payable(DiamondSwapFeeReceiver).call{value:
            ↪ diamondFee}("");
175         require(success, "Failed to send Diamond Fee");
176         (success, ) = payable(ProjectFeeReceiver).call{value:
            ↪ projectFee}("");
177         require(success, "Failed to send Project Fee");
178         (success, ) = payable(ResellerFeeReceiver).call{value:
            ↪ resellerFee}("");
179         require(success, "Failed to send Reseller Fee");
180         (success, ) = payable(_DiamondInterface).call{value: newPrice
            ↪ }("");
181         require(success, "Failed to send ETH");

```

## SHB.16.2: ownedPool.sol

```

235 function takeFee(
236     uint256 price
237 ) internal returns(
238     uint256 newPrice
239 ) {
240     uint256 diamondFee = ((price * DiamondFee) / 100);
241     uint256 projectFee = ((diamondFee * ProjectFee) / 100);
242     uint256 resellerFee = ((diamondFee * ResellerFee) / 100);
243     newPrice = price - diamondFee;
244     diamondFee -= (projectFee + resellerFee);
245     bool success;
246     //Take and distribute fees
247     if(isVerified && isReseller) {
248         (success, ) = payable(DiamondSwapFeeReceiver).call{value:

```



```

        ↪ diamondFee}("");
249     require(success, "Failed to send Diamond Fee");
250     (success, ) = payable(ProjectFeeReceiver).call{value:
        ↪ projectFee}("");
251     require(success, "Failed to send Project Fee");
252     (success, ) = payable(ResellerFeeReceiver).call{value:
        ↪ resellerFee}("");
253     require(success, "Failed to send Reseller Fee");
254     (success, ) = payable(_DiamondInterface).call{value: newPrice
        ↪ }("");
255     require(success, "Failed to send ETH");

```

### SHB.16.3: manyToMany.sol

```

158 function takeFee(
159     uint256 price
160 ) internal returns(
161     uint256
162 ) {
163     uint256 diamondFee = ((price * DiamondFee) / 100);
164     uint256 projectFee = ((diamondFee * ProjectFee) / 100);
165     uint256 resellerFee = ((diamondFee * ResellerFee) / 100);
166     uint256 newPrice = price - diamondFee;
167
168     diamondFee -= (projectFee + resellerFee);
169
170     bool success;
171
172     //Take and distribute fees
173     if(isVerified && isReseller) {
174         (success, ) = payable(DiamondSwapFeeReceiver).call{value:
            ↪ diamondFee}("");
175         require(success, "Failed to send Diamond Fee");
176         (success, ) = payable(ProjectFeeReceiver).call{value:
            ↪ projectFee}("");

```

```

177         require(success, "Failed to send Project Fee");
178         (success, ) = payable(ResellerFeeReceiver).call{value:
            ↪ resellerFee}("");
179         require(success, "Failed to send Reseller Fee");
180         (success, ) = payable(_DiamondInterface).call{value: newPrice
            ↪ }("");
181         require(success, "Failed to send ETH");

```

## Recommendation:

Consider having this logic of claiming ether instead of sending the amounts directly, this will reduce the gas cost for the user and also prevent the issue.

## Updates

The DiamondSwap team resolved the issue for the [ProjectFeeReceiver](#) and [ResellerFeeReceiver](#) by sending the fees to the [DiamondSwapFeeReceiver](#) when the calls fail. However, the issue is still valid for the [DiamondSwapFeeReceiver](#).

### SHB.16.4: publicPool.sol

```

211 if(isVerified && isReseller) {
212
213     (success, ) = payable(ProjectFeeReceiver).call{value: projectFee
        ↪ }("");
214     if(!success) {
215         diamondFee += projectFee;
216     }
217     (success, ) = payable(ResellerFeeReceiver).call{value: resellerFee
        ↪ }("");
218     if(!success) {
219         diamondFee += resellerFee;
220     }
221     (success, ) = payable(DiamondSwapFeeReceiver).call{value: diamondFee
        ↪ }("");
222     require(success, "Failed to send Diamond Fee");

```

```

223
224     return newPrice;
225 } else if(isVerified isReseller) {
226     if(isVerified) {
227         (success, ) = payable(ProjectFeeReceiver).call{value: projectFee
                ↪ }("");
228         if(!success) {
229             diamondFee += projectFee;
230         }
231         (success, ) = payable(DiamondSwapFeeReceiver).call{value:
                ↪ diamondFee}("");
232         require(success, "Failed to send Diamond Fee");
233
234         return newPrice;
235     } else {
236         (success, ) = payable(ResellerFeeReceiver).call{value:
                ↪ resellerFee}("");
237         if(!success) {
238             diamondFee += resellerFee;
239         }
240         (success, ) = payable(DiamondSwapFeeReceiver).call{value:
                ↪ diamondFee}("");
241         require(success, "Failed to send Diamond Fee");
242
243         return newPrice;
244     }
245 } else {
246     (success, ) = payable(DiamondSwapFeeReceiver).call{value: diamondFee
                ↪ }("");
247     require(success, "Failed to send Diamond Fee");
248
249     return newPrice;
250 }

```

## SHB.17 Race Condition

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

### Description:

A race condition vulnerability occurs when the code depends on the order of the transactions submitted to it. The project has certain modifiable variables that may be affected by the transaction's execution sequence.

### Exploit Scenario:

The buyer calls the `buyFromPool` function from the `DiamondSwap` contract using a specific value of the `DiamondFee`, then the default admin changes the `DiamondFee`. If the default admin's transaction gets mined first, the buyer's transaction will be executed using the new value of `DiamondFee` generating an unexpected output.

### Files Affected:

#### SHB.17.1: DiamondSwap.sol

```
105 function updateDiamondFee(  
106     uint256 feePercent,  
107     address payable diamondFeeReceiver  
108 ) external onlyRole(DEFAULT_ADMIN_ROLE) {  
109     DiamondFee = feePercent;  
110     DiamondSwapFeeReceiver = payable(diamondFeeReceiver);  
111  
112     emit DiamondEvents.DiamondFeeUpdated(feePercent);  
113 }
```

## Recommendation:

It is recommended to add the diamond fee as an argument to the `buyFromPool` function, then verify that it is the same as the one that is stored in the contract. Also, consider notifying the community of any changes to the fee structure.

## Updates

The DiamondSwap team acknowledged the risk stating that they are planning to use a robust multi-sig process so the admins will change the fees only when it is needed.

## SHB.18 Missing Percentage Check

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

## Description:

There is no implemented measure to check the `DiamondFee` as it could be easily set to a value that exceeds 100, which would result in a negative impact on the logic of the contract. Same issue in the `ownedPool` for the `initialDistributionPercent`.

## Files Affected:

### SHB.18.1: DiamondSwap.sol

```
109     function updateDiamondFee(  
110         uint256 feePercent,  
111         address payable diamondFeeReceiver  
112     ) external onlyRole(DEFAULT_ADMIN_ROLE) {  
113         DiamondFee = feePercent;  
114         DiamondSwapFeeReceiver = payable(diamondFeeReceiver);
```

### SHB.18.2: ownedPool.sol

```
309     function updateVested(  
310         bool vested,  
311         uint256 vestingInfo  
312     ) external onlyRole(DIAMOND_ADMIN) {  
313         isVested = vested;  
314         initialDistributionPercent = vestingInfo;  
315     }
```

### Recommendation:

To solve the issue, a check should be placed in the function to make sure that the **Diamond-Fee** is always less than 100%.

### Updates

The DiamondSwap team resolved the issue by adding a percentage check in the **updateDiamondFee** function.

### SHB.18.3: publicPool.sol

```
92     function updateDiamondFee(  
93         uint256 feePercent,  
94         address payable diamondFeeReceiver  
95     ) external onlyRole(DEFAULT_ADMIN_ROLE) payable {  
96         require(feePercent <= 100 && diamondFeeReceiver != address(0));  
97  
98         PlatformInfo.DiamondFee = feePercent;  
99         PlatformInfo.DiamondSwapFeeReceiver = payable(diamondFeeReceiver);  
100  
101         emit DiamondEvents.DiamondFeeUpdated(feePercent);  
102     }
```

## SHB.19 Loss Of Precision

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

### Description:

In `createPool` function, `(fixedEthPrice * 10**18)` is divided by `tokenAmount`, the issue here is that if we have the `(fixedEthPrice * 10**18)` less than `tokenAmount` the `pools[token].fixedPricePerToken[newPool]` will be equal to 0 due to a loss of precision.

### Files Affected:

#### SHB.19.1: DiamondSwap.sol

```
345 if(isVestedIsFixedPriceIsTwitterIsHiddenPreventCancel[1]) {
346     IDiamondContract(newPool).setFixedPrice(fixedEthPrice, tokenAmount);
347     pools[token].fixedPricePerToken[newPool] = (fixedEthPrice * 10**18)
        ↪ / tokenAmount;
348 }
```

### Recommendation:

Consider verifying that `(fixedEthPrice * 10**18)` is greater than `tokenAmount`.

### Updates

The DiamondSwap team resolved the issue by requiring the `fixedEthPrice * 10**18` to be greater than `amount`.

#### SHB.19.2: publicPool.sol

```
261 if(dataChecks[1]) {
262     require((fixedEthPrice * 10**18) > amount);
263     IDiamondContract(newPool).setFixedPrice(fixedEthPrice, amount);
```

```

264     pools[token].fixedPricePerToken[newPool] = (fixedEthPrice * 10**18)
        ↪ / amount;
265 }

```

## SHB.20 Functions Not Existing In The Interface Or Missing Parameters

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

### Description:

Certain functions don't exist in the interface, or they have some missing parameters, causing the contracts to not compile correctly.

### Files Affected:

#### SHB.20.1: DiamondSwap.sol

```

325 if(isVestedIsFixedPriceIsTwitterIsHiddenPreventCancel[4]) {
326     IDiamondContract(newPool). preventCancellation (true);
327 }

```

#### SHB.20.2: manyToMany.sol

```

119     } else {
120         contributorAmounts[contributors[counter]] -= amount;
121
122         _diamondSwap.deposit(contributors[counter], newPrice,
            ↪ address(_token), address(this));
123     }
124     _diamondSwap. updatePublicAmount (amount, address(_token),
        ↪ address(this), to);

```



```
125     _balance -= amount;
126     _token.safeTransfer(to, amount);
```

### SHB.20.3: DiamondSwap.sol

```
652     uint256 _amount = IDiamondContract(_pool).updateOwner (msg.sender
    ↪ ,newOwner, preventCancel);
```

## Recommendation:

- Consider adding the `preventCancellation` function to `IDiamondContract` interface
- Consider fixing the `updatePublicAmount` parameters by adding the `owner` address in the last argument.
- Consider adding a third argument to `updateOwner` function in `IDiamondContract` interface.

## Updates

The DiamondSwap team resolved the issue by fixing the parameters in the interface.

### SHB.20.4: IDiamondContract.sol

```
40 function preventCancellation(
41     bool _preventCancel
42 ) external payable;
```

## SHB.21 The `initialize` function Can Be Front Run

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

## Description:

The `DiamonSwap` contract is an upgradable contract that contains the `initilize` function, having the fact the function is protected by the `initializer` modifier will only protect the function from being called more than once, however it doesn't protect the function from being called by another entity. This is due to not having the deployment and the `initialize` function in the same transaction. This is marked as a low issue because it's unlikely to have an attacker listen to all mempool transactions and front-run the `initialize` call.

Same issue in `publicPoolContract` contract.

## Files Affected:

### SHB.21.1: DiamondSwap.sol

```
49     function initialize(address publicPoolCreator, address
        ↪ ownedPoolCreator) initializer public {
50         __ERC20_init("DIAMOND SWAP MAIN", "DIAMOND SWAP MAIN");
51         __AccessControl_init();
```

### SHB.21.2: publicPoolContract.sol

```
18     function initialize() initializer public {
19         __Ownable_init();
20         __UUPSUpgradeable_init();
21
22     }
```

## Recommendation:

Consider calling the `initialize` and the deployment of the contract in the same transaction, this can be done by using another contract, it can be either a proxy or a new contract.

## Updates

The `DiamondSwap` team acknowledged the risk.

## SHB.22 For Loop Over Dynamic Array

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

### Description:

When smart contracts are deployed, or their associated functions are invoked, the execution of these operations always consumes a certain quantity of gas, according to the amount of computation required to accomplish them. Modifying an unknown-sized array that grows over time can result in a Denial Of Service. Simply by having an excessively large array, users can exceed the gas limit, therefore preventing the transaction from ever succeeding.

### Files Affected:

#### SHB.22.1: DiamondSwap.sol

```
455     } else if(userAmounts[0] > 0) {
456         pools[_token].isReserved[poolAddress] = true;
457         for (uint256 i = 0; i < userAddresses.length; i++) {
458             updateReservedPool(_token, poolAddress, userAmounts[i]
459                 ↪ ], userAddresses[i], "", false);
460             emit DiamondEvents.tokensReserved(userAddresses[i],
461                 ↪ userAmounts[i], poolAddress);
462         }
463     }
```

### Recommendation:

We recommend avoiding any actions that involve looping across the entire data structure. If you really must loop over an array of unknown size, you will need to arrange for it to consume many blocks and thus multiple transactions.

## Updates

The DiamondSwap team resolved the issue by removing the `addSpecificUsers` function.

## SHB.23 Missing Value Verification

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

### Description:

Certain functions lack a value safety check, the values of the arguments should be verified to allow only the ones that comply with the contract's logic. In the constructor of `publicPool`, `_balance` should be greater than 0.

### Files Affected:

#### SHB.23.1: publicPool.sol

```
52 constructor(address token, uint256 amount, address DiamondInterface,  
    ↪ address owner) payable ERC20 ("Public Pool", "Public Pool") {  
53  
54     _DiamondInterface = payable(DiamondInterface);  
55     _token = token;  
56     _balance = amount;
```

#### SHB.23.2: diamondSwap.sol

```
479     function contributeToOwnedPool(  
480         uint256 tokenAmount,  
481         address token,  
482         address pool  
483     ) external {  
484         IERC20(token).safeTransferFrom(msg.sender, pool, tokenAmount);
```

```

485     IDiamondContract(payable(pool)).contribute(tokenAmount, token,
        ↪ msg.sender);
486     pools[token].PublicSaleAmount[pool] += tokenAmount;
487     users[msg.sender].createdPoolAmountContributed[token][pool] +=
        ↪ tokenAmount;
488     pools[token].hidden[pool] = false;
489     if(!publicTokens.codiamondTransferntains(token)) {
490         publicTokens.add(token);
491         publicPools.add(pool);
492     }

```

## Recommendation:

We recommend that you verify the values provided in the arguments. The issue can be addressed by utilizing a [require](#) statement.

## Updates

The DiamondSwap team resolved the issue by adding a [require](#) statement that makes sure the [amount](#) is greater than zero.

### SHB.23.3: publicPool.sol

```

56     constructor(address token, uint256 amount, address DiamondInterface,
        ↪ address owner) payable ERC20 ("Public Pool", "Public Pool") {
57         require(amount > 0);

```

## SHB.24 Missing Address Verification

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

## Description:

Certain functions lack a safety check in the address, the address-type arguments should include a zero-address test, otherwise, the contract's functionality may become inaccessible.

## Files Affected:

### SHB.24.1: DiamondSwap.sol

```
60     IPublicPoolContract _PublicPoolContract = IPublicPoolContract(  
        ↪ payable(address(publicPoolCreator)));  
61     PublicPoolContract = _PublicPoolContract;  
62     IOwnedPoolContract payable(address(ownedPoolCreator));  
63     OwnedPoolContract = _OwnedPoolContract;
```

### SHB.24.2: DiamondSwap.sol

```
110    function updateDiamondFee(  
111        uint256 feePercent,  
112        address payable diamondFeeReceiver  
113    ) external onlyRole(DEFAULT_ADMIN_ROLE) {  
114        DiamondFee = feePercent;  
115        DiamondSwapFeeReceiver = payable(diamondFeeReceiver);
```

### SHB.24.3: publicPoolContract.sol

```
35    function updateInterfaceAddress(address newInterface) external  
        ↪ onlyOwner {  
36        diamondInterface = newInterface;  
37        transferOwnership(newInterface);  
38    }
```

### SHB.24.4: publicPool.sol

```
47    constructor(address token, uint256 amount, address DiamondInterface,  
        ↪ address owner) payable ERC20 ("Public Pool", "Public Pool") {  
48
```

```
49     _DiamondInterface = payable(DiamondInterface);
50     _token = token;
```

#### SHB.24.5: ownedPoolContract.sol

```
34     function updateInterfaceAddress(address newInterface) external
        ↪ onlyOwner {
35         diamondInterface = newInterface;
36         transferOwnership(newInterface);
37     }
```

### Recommendation:

We recommend that you make sure the addresses provided in the arguments are different from the `address(0)`.

### Updates

The `DiamondSwap` team partially resolved the issue by verifying the `publicPoolCreator`, the `ownedPoolCreator` and the `diamondFeeReceiver` to be different from the `address(0)`.

#### SHB.24.6: DiamondSwap.sol

```
50     function __Diamond_init_unchained(address publicPoolCreator, address
        ↪ ownedPoolCreator) internal initializer {
51         require(publicPoolCreator != address(0) && ownedPoolCreator !=
            ↪ address(0));
```

## SHB.25 No Verification OffChain Done For The Price

- Severity: **UNDETERMINED**
- Status: Fixed
- Likelihood: 1
- Impact: -

## Description:

In the `checkAmounts` function, no verification off-chain is done in order to get the spot price, and no comparison between the spot price and the `weiPerToken` in the function.

## Files Affected:

### SHB.25.1: ownedPool.sol

```
155     function checkAmounts(  
156         uint256 price,  
157         uint256 range,  
158         uint256 amount  
159     ) internal view {  
160  
161         uint256 weiPerToken;  
162         uint256 tokenAmount;  
163         uint256 rangeAmount;
```

## Recommendation:

Consider adding a call to validate the spot price of the token.

## Updates

The DiamondSwap team resolved the issue by getting the `weiPerToken` from the `spotAggregator`.

### SHB.25.2: ownedPool.sol

```
168     function checkAmounts(  
169         uint256 price,  
170         uint256 amount  
171     ) internal view {  
172         uint256 weiPerToken;  
173         uint256 tokenAmount;  
174         uint256 rangeAmount;
```



```

175
176     if(!fixedPrice) {
177         // Comparing spot price to passed price
178         (weiPerToken, ) = spotAggregator.getRate(IERC20(WETH), IERC20(_token
            ↪ ), IERC20(address(0)));
179         weiPerToken -= ((weiPerToken * discountPercent) / 1000);
180         tokenAmount = price / weiPerToken;
181         tokenAmount *= 10**18;
182         rangeAmount = ((amount * 5) / 100);
183         require((amount - rangeAmount) <= tokenAmount && tokenAmount <= (
            ↪ amount + rangeAmount), "Invalid amount, adjust range");
184     } else if(fixedPrice) {
185         tokenAmount = (price / pricePerTokenWei);
186         tokenAmount *= 10**18;
187         rangeAmount = ((amount * 5) / 100);
188         require ((amount - rangeAmount) <= tokenAmount && tokenAmount <=
            ↪ (amount + rangeAmount), "Invalid amount, adjust range");
189     }

```

## 4 Best Practices

### BP.1 Use The **Pausable** Contract Instead Of **Allow-Claim**

#### Description:

In the `diamondSwap` contract, the `updateClaimable` is used to disable a contract in case of exploitation. It's recommended to use the `pausable` contract since it's standard and you can import the library and use the modifier `whenNotPaused` in all critical functions.

#### Files Affected:

##### BP.1.1: DiamondSwap.sol

```
97     function updateClaimable(  
98         bool _enable  
99     ) external onlyRole(DEFAULT_ADMIN_ROLE) {  
100         allowClaim = _enable;  
101  
102         emit DiamondEvents.claimingAllowed(_enable);  
103     }
```

#### Status - Fixed

The `DiamondSwap` team implemented the best practice by using the `Pausable` contract.

### BP.2 Remove **IsVerified** From **UserInfo** Struct

#### Description:

In the `DiamondStructs` library, the `isVerified` attribute stores a boolean value that indicates if the user has been verified or not. This variable is not needed since we can use the `verifiedUser` string and verify if it's empty then it means that the user is still not verified.

## Files Affected:

### BP.2.1: DiamondSwap.sol

```
11 struct UserInfo {
12     uint256 claimableETH; // ETH in users account
13     uint256 totalETHEarned; // Total amount of ETH earned by user
14     bool isVerified; // Verified address
15     bool locked; // Lock specific user from withdrawing funds
16     string verifiedUser; // List users kyc'd name, i.e. twitter or tg
    ↪ @'s
```

## Status - Fixed

The DiamondSwap team followed the best practice by removing the `isVerified` attribute from the `UserInfo` struct.

## BP.3 Remove Modifier From `getFeeData` Function

### Description:

The `getFeeData` function located in the DiamondSwap contract is a view function protected by the modifier `onlyRole(DIAMOND_CONTRACT)`. This access control is useless since all data in the blockchain can be accessed and viewed by any user even when adding the `private` keyword to the variable.

Same issue in the `getReservedAmount` function.

## Files Affected:

### BP.3.1: DiamondSwap.sol

```
156     function getFeeData(
157         address _token
158     ) external onlyRole(DIAMOND_CONTRACT) view returns(
159         address payable diamondSwapFeeReceiver,
160         address payable projectReceiver,
```

```

161     address payable resellerReceiver,
162     uint256 diamondFee,
163     uint256 projectFee,
164     uint256 resellerFee,
165     bool isVerified,
166     bool isReseller
167 ) {
168     address token = _token;
169     return(
170         payable(DiamondSwapFeeReceiver),
171         payable(pools[token].projectFeeReceiver),
172         payable(pools[token].resellerFeeReceiver),
173         DiamondFee,
174         pools[token].projectFee,
175         pools[token].resellerFee,
176         pools[token].isVerified,
177         pools[token].isReseller
178     );
179 }

```

### BP.3.2: DiamondSwap.sol

```

192     function getReservedAmount(
193         string memory handle,
194         address user,
195         address pool,
196         address token,
197         bool isTwitter
198     ) external onlyRole(DIAMOND_CONTRACT) view returns(
199         uint256 reservedTokens
200     ){
201         if(isTwitter) {
202             return pools[token].SpecificUserAmountTwitter[pool][handle];
203         } else {
204             return pools[token].SpecificUserAmount[pool][user];

```

```
205     }
206 }
```

## Status - Fixed

The DiamondSwap team followed the best practice by removing the modifier from the `get-FeeData` function.

## BP.4 Redundant Verification On Price Of Sent Ether

### Description:

In the `buyFromPool`, a user can call this function to buy tokens from the pool by sending an amount of ETH to the pool, however the `msg.value` is validated against the `price` variable which is redundant, it's advised to use directly the `msg.value`.

### Files Affected:

#### BP.4.1: DiamondSwap.sol

```
1465     function buyFromPool(
1466         address payable pool,
1467         uint256 amount,
1468         uint256 price,
1469         uint256 range
1470     ) external payable {
1471         require(msg.value == price, "!ETH");
1472         (uint256 newPrice, bool success) = IDiamondContract(payable(pool)
            ↪ ).diamondTransfer{value: msg.value}(msg.sender, amount,
            ↪ price, range);
1473         require(success, "!GAS");
```

## Status - Fixed

The DiamondSwap team followed the best practice by removing the redundant verification.

## BP.5 Wrong Function Name

### contributeToOwnedPool

#### Description:

Using the `contributeToOwnedPool` we can contribute to any pool, whether it's public or owned, it's advised to change the naming of the function to reflect more the logic of the function, ex. `contributeToPool`.

#### Files Affected:

##### BP.5.1: DiamondSwap.sol

```
479 function contributeToOwnedPool(  
480     uint256 tokenAmount,  
481     address token,  
482     address pool  
483 ) external {  
484     IERC20(token).safeTransferFrom(msg.sender, pool, tokenAmount);  
485     IDiamondContract(payable(pool)).contribute(tokenAmount, token,  
         ↪ msg.sender);  
486     pools[token].PublicSaleAmount[pool] += tokenAmount;  
487     users[msg.sender].createdPoolAmountContributed[token][pool] +=  
         ↪ tokenAmount;  
488     pools[token].hidden[pool] = false;  
489     if(!publicTokens.contains(token)) {  
490         publicTokens.add(token);  
491         publicPools.add(pool);  
492     }  
493  
494     emit DiamondEvents.poolContribution(pools[token].  
         ↪ publicPoolAddress, tokenAmount);  
495 }
```

## Status - Fixed

The DiamondSwap team followed the best practice by changing the function name to `contributeToPool`.

## BP.6 Unnecessary Payable Function `claimETH`

### Description:

The `claimETH` is a payable function, however this function doesn't receive any ethers, it's advised to remove the `payable` keyword.

### Files Affected:

#### BP.6.1: DiamondSwap.sol

```
615     function claimETH(  
616     ) external payable nonReentrant {  
617         require(allowClaim && !users[msg.sender].locked, "!CLAIMABLE");  
618         uint256 amount = users[msg.sender].claimableETH;
```

## Status - Fixed

The DiamondSwap team followed the best practice by removing the `payable` keyword from the `claimETH` function.

## BP.7 Redundant/Unnecessary Code

### Description:

When claiming ethers using the `claimETH` function, the `claimableETH` is sent back to the user. It's advised to set the attribute to 0 rather than reducing it from the `amount` value.

### Files Affected:

### BP.7.1: DiamondSwap.sol

```
615     function claimETH(  
616     ) external payable nonReentrant {  
617         require(allowClaim && !users[msg.sender].locked, "!CLAIMABLE");  
618         uint256 amount = users[msg.sender].claimableETH;  
619  
620         users[msg.sender].claimableETH -= amount;  
621         (bool success, ) = address(msg.sender).call{value: amount}("");  
622         require(success, "!SEND");  
623  
624         emit DiamondEvents.ETHClaimed(msg.sender, amount);  
625     }
```

### Status - Fixed

The DiamondSwap team followed the best practice by setting `users[msg.sender].claimableETH` to zero instead of decrementing it.

## BP.8 Remove Dead Code

### Description:

Remove the dead code from the `checkAmounts` function located in the `publicPool` and `ownedPool` contracts.

### Files Affected:

#### BP.8.1: publicPool.sol

```
140     function checkAmounts(  
141         uint256 price,  
142         uint256 range,  
143         uint256 amount  
144     ) internal pure {  
145         uint256 weiPerToken;
```



```

146     uint256 tokenAmount;
147     uint256 rangeAmount;
148     // Comparing spot price to passed price
149     //(uint256 EthPerToken, ) = spotAggregator.getRate(IERC20(WETH),
        ↪ IERC20(_token), IERC20(zeroAddress));
150     weiPerToken = 5000000;
151     tokenAmount = price / weiPerToken;
152     tokenAmount *= 10**18;
153     rangeAmount = ((amount * range) / 100);
154     require((amount - rangeAmount) <= tokenAmount && tokenAmount <= (
        ↪ amount + rangeAmount), "Invalid amount, adjust range");
155 }

```

## BP.8.2: ownedPool.sol

```

155     function checkAmounts(
156         uint256 price,
157         uint256 range,
158         uint256 amount
159     ) internal view {
160
161         uint256 weiPerToken;
162         uint256 tokenAmount;
163         uint256 rangeAmount;
164
165         if(!fixedPrice) {
166             // Comparing spot price to passed price
167             //(uint256 weiPerToken, ) = spotAggregator.getRate(IERC20(WETH),
                ↪ IERC20(_token), IERC20(zeroAddress));
168             weiPerToken = 5000000;
169             weiPerToken -= ((weiPerToken * discountPercent) / 1000);
170             tokenAmount = price / weiPerToken;
171             tokenAmount *= 10**18;
172             rangeAmount = ((amount * range) / 100);
173             require((amount - rangeAmount) <= tokenAmount && tokenAmount <= (

```

```

        ↪ amount + rangeAmount), "Invalid amount, adjust range");
174     } else if(fixedPrice) {
175         tokenAmount = (price / pricePerTokenWei);
176         tokenAmount *= 10**18;
177         rangeAmount = ((amount * range) / 100);
178         require ((amount - rangeAmount) <= tokenAmount && tokenAmount
        ↪ <= (amount + rangeAmount), "Invalid amount, adjust
        ↪ range");
179     }
180 }

```

### BP.8.3: manyToMany.sol

```

136     function checkAmounts(
137         uint256 price,
138         uint256 range,
139         uint256 amount
140     ) internal view {
141
142         // Comparing spot price to passed price
143         //(uint256 EthPerToken, ) = spotAggregator.getRate(IERC20(WETH),
        ↪ IERC20(_token), IERC20(zeroAddress));
144         uint256 EthPerToken = 10000000000000000;

```

## Status - Fixed

The DiamondSwap team followed the best practice by removing the dead code.

## BP.9 No Need To Add The Token Parameter In The `contribute` function

### Description:

Remove the parameter `token` from the `contribute` function. Instead, directly use the `_token` variable located in the `publicPool` contract.

## Files Affected:

### BP.9.1: publicPool.sol

```
214     function contribute(  
215         uint256 tokenAmount,  
216         address token,  
217         address user  
218     ) external  
219         require(address(token) == address(_token), "Must deposit the  
           ↪ correct token into this pool");  
220         _balance += tokenAmount;  
221         if(poolContributor[user]) {  
222             contributorAmounts[user] += tokenAmount;  
223         } else {  
224             poolContributor[user] = true;  
225             contributors.push(user);  
226             contributorCounter[user] = placeInLine;  
227             contributorAmounts[contributors[placeInLine]] += tokenAmount;  
228             placeInLine++;  
229         }  
230         isHidden = false;  
231     }
```

### BP.9.2: ownedPool.sol

```
295     function contribute(  
296         uint256 tokenAmount,  
297         address token,  
298         address user  
299     ) external onlyRole(DIAMOND_ADMIN) nonReentrant {  
300         require(!singleSale, "Invalid pool");  
301         require(address(user) == address(poolOwner), "Only pool owner can  
           ↪ call this function");  
302         require(address(token) == address(_token), "Must deposit the  
           ↪ correct token into this pool");
```

```
303     require(tokenAmount > 0, "No tokens sent to pool");
304     _balance += tokenAmount;
305     publicAmount += tokenAmount;
306     isHidden = false;
307 }
```

## Status - Fixed

The DiamondSwap team followed the best practice by removing the token parameter.

# 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 Diamond Swap contract and discovered several issues of varying severity. Diamond Swap team addressed 17 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 Diamond Swap 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
DiamondSwap.sol	fa4f464ed684c7c8d6f69df3f18671c6
contracts/manyToMany.sol	2d0e89c95906f1f3fb3937ec5c92bfbd
contracts/ownedPool.sol	3a12d7404213e5fc89138448d81d49a3
contracts/ownedPoolContract.sol	d5dea9e0939a7f57228f3564cc429c0d
contracts/publicPool.sol	164f3a7c85538f2e090c98669b4d8d93
contracts/publicPoolContract.sol	b717bcb06e791c9a94c3dcfdccc64df6
contracts/libraries/DiamondEvents.sol	6a8804e55aa09a6575722509aec6d4a6
contracts/libraries/DiamondSearch.sol	2ce29ac8b8a3efab75a9c32d54ab3cd9
contracts/libraries/DiamondStructs.sol	59f8069c35ccbfaafa0a0fd69df85c4
contracts/libraries/UpdateDiamondStruct.sol	5e2c54474c1695fcd0e7aa90eecf8150

## 7.2 Re-Audit

Files	MD5 Hash
DiamondSwap.sol	a4e62a57348c8e8268cddfc6febc69dd
contracts/ownedPool.sol	fe67b9415575395dd02a17f810b60f81
contracts/ownedPoolContract.sol	79de865b898b68a0b34217fb3a9bd4eb
contracts/publicPool.sol	76d7b9c0eb648ccb4204309b41d5aca3

contracts/publicPoolContract.sol	637e8c912d18579fce57b5d0af09b38e
contracts/libraries/DiamondEvents.sol	c664a41a1b13a03e1545b6c006ad87dc
contracts/libraries/DiamondSearch.sol	aceff48220668bfe40fd174d45d5d67b
contracts/libraries/DiamondStructs.sol	049080801f9de7a0060772ba1a4dce2e
contracts/libraries/UpdateDiamondStruct.sol	f528bff3d6fc0f4059f34dd042c84fd2
contracts/interface/IDiamondContract.sol	9fb37863c1d9c51829cab32403d48274
contracts/interface/IDiamondEvents.sol	768e41e135dacbe8d4b393f59308e2e2
contracts/interface/IDiamondSwap.sol	788af61ded86c8985934e32c39aae69b
contracts/interface/IOracle.sol	5525cc18092d45e08a8f72a8256993fc
contracts/interface/IOwnedPoolContract.sol	9412b043de1c84243d124d6ef7780871
contracts/interface/IPublicPoolContract.sol	84d736bedc9726c9e58e45ee4f3194dd



## 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.



For a Contract Audit, contact us at [contact@shellboxes.com](mailto:contact@shellboxes.com)