Competitive Security Assessment

Glyph_Exchange

Mar 19th, 2024

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3′s Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

• Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.

• Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.

• Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.

• Verify the code base is compliant with the most up-to-date industry standards and security best practices.

• Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

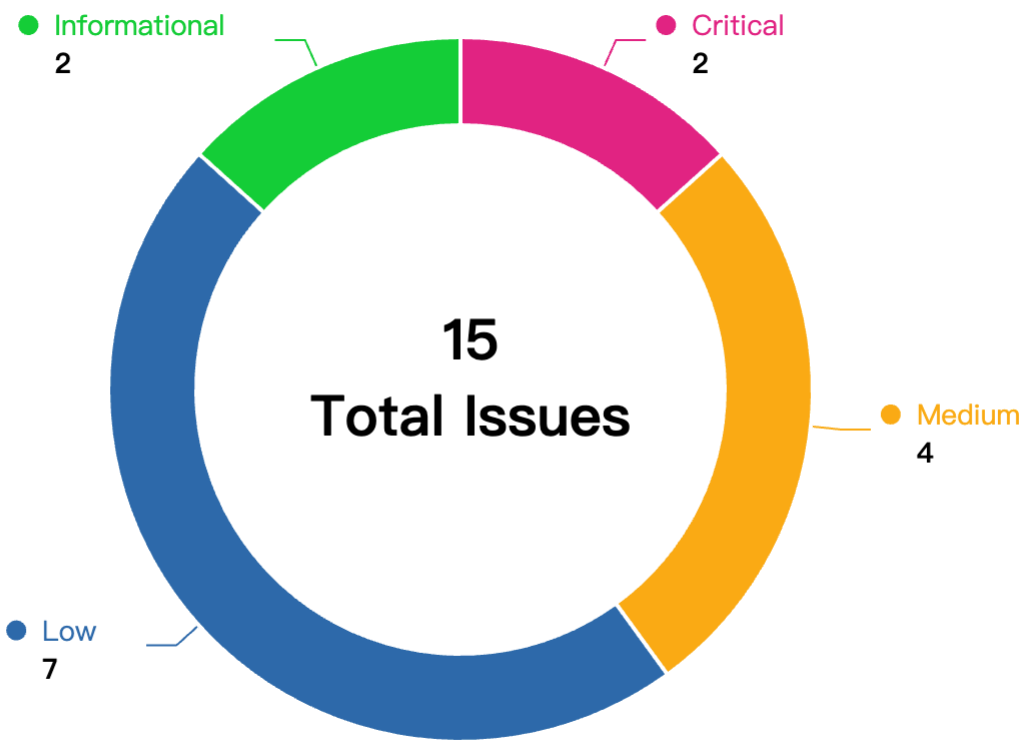
Overview

Project Name	Glyph_Exchange
Language	Solidity
Codebase	• https://github.com/glyph-exchange/glyph-stake-contracts • audit version - 0a9561e710ad57197da727b5e6879b0ce5e4e2a9 • final version - 9c3022e47293549d9e1abc984d6d792b4b269b2a
Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

Audit Scope

File	SHA256 Hash
./contracts/swap/SwapRouter.sol	46276db1eb36df9d4a5dea15daea899a77c1382821c327900d94f16cdec7d9c7
./contracts/swap/SwapPair.sol	f8e0e11e2637326dba73be99e2b122dc87f7b663bf55ca00986db23de96344b4
./contracts/swap/SwapFactory.sol	d86ae2ef7ef5260c897caf6b7f1d3c8f93634002de87db97f947af51fff6655c
./contracts/swap/SwapERC20.sol	c21f37e61263e01f7bb6add53bde98735c838b40ff853b21fb18515a0affcead

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
GEX-1	Signature Malleability of EVM's `ecrecover()`	Signature Forgery or Replay	Critical	Fixed	0xzoobi
GEX-2	Incorrect Identification of WETH as a Fairc20 Token Leading to Pair Creation Reversion	DOS	Critical	Fixed	arno
GEX-3	Unverified Return Values from External Calls Lead to Potentially Unsafe Token Interactions	Logical	Medium	Fixed	BradMoonUESTC
GEX-4	Unbounded Liquidity Minting Vulnerability in DEX Liquidity Pool Migration Logic	Logical	Medium	Fixed	BradMoonUESTC
GEX-5	Replay attack in case of hard fork	Signature Forgery or Replay	Medium	Fixed	0xzoobi
GEX-6	Lack of check for minting complete in function `mint()`	Logical	Medium	Fixed	Yaodao

GEX-7	Potential Exploit in GlyphDEX Swap Function Allowing Manipulation of Token Balances	Logical	Low	Fixed	BradMoonUESTC
GEX-8	Ownership change should use two-step process	Logical	Low	Fixed	Yaodao
GEX-9	Ownable: Does not implement 2-Step-Process for transferring ownership	Logical	Low	Fixed	0xzoobi
GEX-10	Missing Sender-Recipient Address Check in Token Transfer Function	Logical	Low	Acknowledged	0xzoobi
GEX-11	Mint and Burn Allowed when the freeze is set to true	Privilege Related	Low	Fixed	0xzoobi
GEX-12	Lack of the check for address (0)	Logical	Low	Fixed	Yaodao
GEX-13	Lack check of `MINIMUM_LIQUIDITY`	Logical	Low	Fixed	Yaodao
GEX-14	No need to use SafeMath in solidity version which is above 0.8	Language Specific	Informational	Acknowledged	ginlee, Yaodao
GEX-15	Liquidity Migration Allows Multiple Executions, Potentially Diluting the Liquidity Pool	Logical	Informational	Acknowledged	BradMoonUESTC

GEX-1:Signature Malleability of EVM's `ecrecover()`

Category	Severity	Client Response	Contributor
Signature Forgery or Replay	Critical	Fixed	0xzoobi

Code Reference

- code/contracts/swap/SwapERC20.sol#L90

```
90: address recoveredAddress = ecrecover(digest, v, r, s);
```

Description

0xzoobi: The function currently utilizes the Solidity **`ecrecover()`** function directly for signature verification. However, it's important to note that the **`ecrecover()`** EVM opcode can lead to malleable (non-unique) signatures, making the contract vulnerable to replay attacks.
While a replay attack may not appear feasible for this contract at the moment, I highly recommend adopting the battle-tested OpenZeppelin ECDSA library for enhanced security.
For further details, refer to the Siilae reports from the **C4 contest**:

- 2021-04-meebits#h-01-signature-malleability-of-evms-ecrecover-in-verify
- 2022-11-non-fungible#n-09-signature-malleability-of-evms-ecrecover

Recommendation

0xzoobi: Utilize the **`recover`** function provided by OpenZeppelin's ECDSA library for signature verification. Make sure to use the latest version of OpenZeppelin contracts.

Client Response

0xzoobi: Fixed.fix details
Utilize the recover function provided by OpenZeppelin's ECDSA library for signature verification
commit_url
https://github.com/glyph-exchange/glyph-stake-contracts/commit/c808f8dde6cd1602f5db3a2e80d8822ac8828227

GEX-2:Incorrect Identification of WETH as a Fairc20 Token Leading to Pair Creation Reversion

Category	Severity	Client Response	Contributor
DOS	Critical	Fixed	arno

Code Reference

- code/contracts/swap/SwapFactory.sol#L41-L48

```
41: if (isFairc20PoolWhitelistEnabled) {
42:             //only whitelist can create fairc20 pool
43:             (bool token0IsFairc20, ) = token0.call(abi.encodeWithSelector(bytes4(keccak256("mint
Complete()"))));
44:             (bool token1IsFairc20, ) = token1.call(abi.encodeWithSelector(bytes4(keccak256("mint
Complete()"))));
45:
46:             if (token0IsFairc20) require(isFairc20PoolWhitelist[msg.sender], "GlyphDEX Swap: Not
Whitelist");
47:             if (token1IsFairc20) require(isFairc20PoolWhitelist[msg.sender], "GlyphDEX Swap: Not
Whitelist");
48:         }
```

Description

arno: The SwapFactory contract is designed to facilitate the creation of token pairs for a decentralized exchange. A significant issue arises when users attempt to create a pair involving Wrapped Ether (WETH) and any other token, due to WETH's behavior of accepting arbitrary calls and returning **`true`**. The contract includes a mechanism intended to restrict pair creation to only whitelisted addresses for certain tokens identified as **`Fairc20`**. However, because WETH on the Ethereum mainnet will return **`true`** for any call, including the **`mintComplete()`** check used to identify **`Fairc20`** tokens, attempts to create pairs with WETH and another token (e.g., USDT) will inadvertently be rejected if **`isFairc20PoolWhitelistEnabled`** is enabled. By default, this setting is **`true`** upon deployment, leading to an unexpected behavior where valid pair creation transactions are reverted despite the creator having no intention of establishing a **`Fairc20`** pair.

Recommendation

arno: 1. **Adjust the `Fairc20` Detection Method:** Implement a more robust and specific method for identifying **`Fairc20`** tokens that does not rely on a function call's success status. This could involve checking for a specific return value or using a registry of known **`Fairc20`** tokens.

Client Response

arno: Fixed： Incorrect Identification of WETH as a Fairc20 Token
Commit_url:
https://github.com/glyph-exchange/glyph-stake-contracts/commit/1c1fd3cc4ac629bf44092a3e0bb3128af9b9bfbf#diff-e77643ac234cfc8c268749153657142b367ddf8a981cf5811c325b6eef53586b
https://github.com/glyph-exchange/glyph-stake-contracts/commit/b2470810f6ffa5fb3cfb2bd48faac72ff4180d5a#diff-e85d5d030e1369010e290364042a0ecef2ea134c95b58e4feb21c709c0944e46

description:
we adjust isFairc20() method for identifying Fairc20

GEX-3:Unverified Return Values from External Calls Lead to Potentially Unsafe Token Interactions

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	BradMoonUESTC

Code Reference

- code/contracts/swap/SwapPair.sol#L209-L216

```
209: {
210:             //can not swap before minting complete if token is fairc20
211:             (bool token0IsFairc20, ) = token0.call(abi.encodeWithSelector(bytes4(keccak256("min
tComplete()"))));
212:             (bool token1IsFairc20, ) = token1.call(abi.encodeWithSelector(bytes4(keccak256("min
tComplete()"))));
213:
214:             if (token0IsFairc20) require(IFairc20(token0).mintComplete(), "GlyphDEX Swap: Minti
ng");
215:             if (token1IsFairc20) require(IFairc20(token1).mintComplete(), "GlyphDEX Swap: Minti
ng");
216:         }
```

Description

BradMoonUESTC: The contract contains a vulnerability in the section where it checks whether the minting process of tokens (token0 and token1) involved in a swap operation has been completed. This check is performed by making external calls to the tokens' **`mintComplete()`** functions and proceeds based on the assumption that the returned results are trustworthy. The code in question is as follows:

```
{
    // cannot swap before minting complete if token is FairC20
    (bool token0IsFairc20, ) = token0.call(abi.encodeWithSelector(bytes4(keccak256("mintComplete
()"))));
    (bool token1IsFairc20, ) = token1.call(abi.encodeWithSelector(bytes4(keccak256("mintComplete
()"))));

    if (token0IsFairc20) require(IFairc20(token0).mintComplete(), "GlyphDEX Swap: Minting");
    if (token1IsFairc20) require(IFairc20(token1).mintComplete(), "GlyphDEX Swap: Minting");
}
```

This approach introduces a risk because it does not verify whether the external call to **`mintComplete()`** succeeded or validate the truthfulness of the returned boolean value. A malicious token contract could exploit this by always returning **`true`** for **`mintComplete()`**, even if the minting process has not actually been completed, or by manipulating the return value based on the caller's address.

Recommendation

BradMoonUESTC: it's advisable to adopt a validation strategy based on the token's minted quantity and total supply, instead of solely relying on the return value of the **`mintComplete()`** function. This approach involves

verifying that the total supply of the token has reached a predetermined threshold that indicates minting completion. The recommendation includes implementing an additional check to verify the token's total supply against a known, expected value that signifies the end of the minting process. Here's how you could implement this revised strategy:

1. Define a constant or configurable threshold that represents the expected total supply upon completion of minting for both tokens involved in the swap.
2. Replace the direct reliance on the `**mintComplete()**` return value with a function that checks if the token's total supply meets or exceeds this threshold.

Below is an example code snippet implementing these recommendations:

```
// Example thresholds for demonstration purposes. Adjust according to your tokenomics.
uint256 private constant TOKEN0_MINT_COMPLETE_THRESHOLD = 1000000 * 10**18; // Example token0 thre
shold
uint256 private constant TOKEN1_MINT_COMPLETE_THRESHOLD = 500000 * 10**18; // Example token1 thres
hold

{
    // Ensuring minting completion from a supply perspective
    require(_isMintingComplete(token0, TOKEN0_MINT_COMPLETE_THRESHOLD), "GlyphDEX Swap: Token0 min
ting incomplete");
    require(_isMintingComplete(token1, TOKEN1_MINT_COMPLETE_THRESHOLD), "GlyphDEX Swap: Token1 min
ting incomplete");
}

// Function to check if minting is complete based on token supply
function _isMintingComplete(address token, uint256 threshold) internal view returns (bool) {
    uint256 tokenSupply = IERC20(token).totalSupply();
    return tokenSupply >= threshold;
}
```

This revised recommendation mitigates the risk by not depending on potentially manipulable boolean return values from external contract calls. Instead, it bases the validation on the observable state of the blockchain, specifically the total supply of tokens, which is less susceptible to manipulation. This method assumes that the threshold values are determined with a clear understanding of the token's minting logic and total supply cap, ensuring they accurately reflect the completion of the minting process.

Client Response

BradMoonUESTC: fixed
We decided to create the pool and add liquidity at the end of fairc20 mint to avoid causing various unexpected problems.
commit url
https://github.com/glyph-exchange/glyph-stake-contracts/commit/0e6d4cf49770c2db13df0d70f45722bfa49c2974

GEX-4:Unbounded Liquidity Minting Vulnerability in DEX Liquidity Pool Migration Logic

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	BradMoonUESTC

Code Reference

- code/contracts/swap/SwapPair.sol#L163-L166

```
163: if (msg.sender == migrator) {
164:                 liquidity = IMigrator(migrator).desiredLiquidity();
165:                 require(liquidity > 0 && liquidity != type(uint256).max, "Bad desired liquidit
y");
166:             } else {
```

Description

BradMoonUESTC: from one pool to another, a feature intended to facilitate upgrades or transitions to new contract versions. This logic is executed when the contract's total supply (`**_totalSupply**`) is zero, indicating the pool is newly deployed or has no existing liquidity. Specifically, the contract allows a designated `**migrator**` contract to specify the amount of liquidity tokens it desires to mint (`**desiredLiquidity**`) for the migration process.
However, there's no upper limit enforced on the `**desiredLiquidity**` amount, potentially allowing the `**migrator**` to mint an arbitrary amount of liquidity tokens. This could lead to significant security risks, including dilution of real liquidity providers' shares, price manipulation, and potential loss of funds if the `**migrator**` withdraws a disproportionate amount of assets from the pool without equivalent value contribution.
The problematic code section is:

```
if (_totalSupply == 0) {
    address migrator = ISwapFactory(factory).migrator();
    if (msg.sender == migrator) {
        liquidity = IMigrator(migrator).desiredLiquidity();
        require(liquidity > 0 && liquidity != type(uint256).max, "Bad desired liquidity");
    } else {
        require(migrator == address(0), "Must not have migrator");
        liquidity = Math.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY);
        _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY toke
ns
    }
}
```

Recommendation

BradMoonUESTC: it is recommended to implement an upper limit for the `**desiredLiquidity**` amount that can be requested by the `**migrator**`. This limit could be based on a reasonable estimation of the pool's expected liquidity or tied to the assets being migrated to ensure that the minted liquidity tokens are backed by equivalent value. Furthermore, the process of setting the `**migrator**` address should be strictly controlled, with only trusted or thoroughly audited contracts allowed to fulfill this role.

A possible code adjustment could include:

```
// Define a maximum liquidity multiplier (e.g., 2x of the total assets being migrated)
uint256 constant MAX_LIQUIDITY_MULTIPLIER = 2;

if (_totalSupply == 0) {
    address migrator = ISwapFactory(factory).migrator();
    if (msg.sender == migrator) {
        uint256 maxLiquidity = Math.sqrt(amount0.mul(amount1)).mul(MAX_LIQUIDITY_MULTIPLIER);
        liquidity = IMigrator(migrator).desiredLiquidity();
        require(liquidity > 0 && liquidity <= maxLiquidity, "Invalid desired liquidity");
    } else {
        require(migrator == address(0), "Must not have migrator");
        liquidity = Math.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY);
        _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY toke
ns
    }
}
```

Client Response

BradMoonUESTC: fixed
Add upper limit for the desiredLiquidity amount that can be requested by the migrator.
commit_url
https://github.com/glyph-exchange/glyph-stake-contracts/commit/9c3022e47293549d9e1abc984d6d792b4b269b2a

GEX-5:Replay attack in case of hard fork

Category	Severity	Client Response	Contributor
Signature Forgery or Replay	Medium	Fixed	0xzoobi

Code Reference

- code/contracts/swap/SwapERC20.sol#L29-L36
- code/contracts/swap/SwapERC20.sol#L86

```
29: DOMAIN_SEPARATOR = keccak256(
30:             abi.encode(
31:                 keccak256('EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)'),
32:                 keccak256(bytes(name)),
33:                 keccak256(bytes('1')),
34:                 chainId,
35:                 address(this)
36:             )
```

```
86: DOMAIN_SEPARATOR,
```

Description

0xzoobi: The 'DOMAIN_SEPARATOR' is not recalculated in the case of a hard fork. The variable DOMAIN_SEPARATOR in the contract SwapERC20 is cached in the contract storage and will not change after being initialized in the constructor. However, if a hard fork occurs after the contract deployment, the domain may become invalid on one of the forked chains due to changes in the block.chainId.
Similar Issues from C4 contests
https://code4rena.com/reports/2022-07-golom#m-05-replay-attack-in-case-of-hard-fork
https://github.com/code-423n4/2021-10-ambire-findings/issues/55

Recommendation

0xzoobi: An elegant solution that you may consider applying is from Sushi Trident:
https://github.com/sushiswap/trident/blob/concentrated/contracts/pool/concentrated/TridentNFT.sol#L47-L62

Client Response

0xzoobi: fixed
Use Sushi Trident solution:
https://github.com/sushiswap/trident/blob/concentrated/contracts/pool/concentrated/TridentNFT.sol#L47-L62
commit_url :https://github.com/glyph-exchange/glyph-stake-contracts/commit/8e0d5cebb8aae7d4d0898d2ccebfff6051a780af

GEX-6:Lack of check for minting complete in function `mint()`

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	Yaodao

Code Reference

- code/contracts/swap/SwapPair.sol#L209-216

```
209: {
210:             //can not swap before minting complete if token is fairc20
211:             (bool token0IsFairc20, ) = token0.call(abi.encodeWithSelector(bytes4(keccak256("min
tComplete()"))));
212:             (bool token1IsFairc20, ) = token1.call(abi.encodeWithSelector(bytes4(keccak256("min
tComplete()"))));
213:
214:             if (token0IsFairc20) require(IFairc20(token0).mintComplete(), "GlyphDEX Swap: Minti
ng");
215:             if (token1IsFairc20) require(IFairc20(token1).mintComplete(), "GlyphDEX Swap: Minti
ng");
216:         }
```

Description

Yaodao: The check for minting complete is used in the function **`SwapPair.swap()`**, which is called by the **`SwapRou ter.swap()`** and will transfer tokens into the pair.

```
        {
            //can not swap before minting complete if token is fairc20
            (bool token0IsFairc20, ) = token0.call(abi.encodeWithSelector(bytes4(keccak256("mintCo
mplete()"))));
            (bool token1IsFairc20, ) = token1.call(abi.encodeWithSelector(bytes4(keccak256("mintCo
mplete()"))));

            if (token0IsFairc20) require(IFairc20(token0).mintComplete(), "GlyphDEX Swap: Mintin
g");
            if (token1IsFairc20) require(IFairc20(token1).mintComplete(), "GlyphDEX Swap: Mintin
g");
        }
```

According to the comments and codes, the check is used to limit the swap of **`fairc20`** tokens before minting is complete.
However, in the function **`SwapPair.mint()`** which is called by **`SwapRouter.addLiquidity()`**, it will transfer tokens into the pair but not check whether minting is complete when the token is fairc20.
As a result, the protocol allows the users to add liquidity before the fairc20 token minting is complete and limits the users to swap before the fairc20 token minting is complete.

Recommendation

Yaodao: Recommend adding the check for minting complete in the function **`mint()`**.

Client Response

Yaodao: fixed
We decided to create the pool and add liquidity at the end of fairc20 mint to avoid causing various unexpected problems.
commit url
https://github.com/glyph-exchange/glyph-stake-contracts/commit/0e6d4cf49770c2db13df0d70f45722bfa49c2974

GEX-7:Potential Exploit in GlyphDEX Swap Function Allowing Manipulation of Token Balances

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	BradMoonUESTC

Code Reference

- code/contracts/swap/SwapPair.sol#L209-L216

```
209: {
210:             //can not swap before minting complete if token is fairc20
211:             (bool token0IsFairc20, ) = token0.call(abi.encodeWithSelector(bytes4(keccak256("min
tComplete()"))));
212:             (bool token1IsFairc20, ) = token1.call(abi.encodeWithSelector(bytes4(keccak256("min
tComplete()"))));
213:
214:             if (token0IsFairc20) require(IFairc20(token0).mintComplete(), "GlyphDEX Swap: Minti
ng");
215:             if (token1IsFairc20) require(IFairc20(token1).mintComplete(), "GlyphDEX Swap: Minti
ng");
216:         }
```

Description

BradMoonUESTC: The GlyphDEX swap function contains a section intended to ensure that tokens cannot be swapped before their minting process is complete, particularly for tokens adhering to the Fairc20 standard. This is achieved through external calls to the **`token0`** and **`token1`** contracts, querying their **`mintComplete`** status. However, these external calls introduce a vulnerability that could be exploited in a rug pull scenario. Specifically, during these calls, a malicious token contract could interact with the pair contract, either transferring additional tokens to it without using the **`skim`** function or decreasing the pair's token balance through the token contract directly. Such actions would not trigger the reentrancy guard ("lock") and could artificially inflate or deflate the **`balance0`** and **`balance1`** values within the swap function.
This manipulation of balances can provide avenues for attackers to exploit the token by either drawing more assets from the pool than intended (if balances are inflated) or by decreasing the pool's assets without a corresponding swap (if balances are deflated), leading to unexpected outcomes for users. This is particularly concerning in the absence of a callback call, as the token could leverage these modified balances to either grant users more tokens than they should receive or diminish their received amount, deviating from expected swap outcomes.

Recommendation

BradMoonUESTC: To mitigate this risk, it is advisable to limit the gas provided to external calls within the swap function, thereby restricting the amount of computation and state changes that these calls can perform. This limitation can help prevent malicious contracts from executing complex operations that could manipulate balances or engage in reentrancy not guarded by the existing lock.
A possible code modification to implement this recommendation is shown below. This involves specifying a gas limit for the external calls querying the **`mintComplete`** status of **`token0`** and **`token1`**. By imposing a gas limit, the ability of these tokens to perform actions that could impact the balances within the swap function is significantly reduced.

Client Response

BradMoonUESTC: fixed
We decided to create the pool and add liquidity at the end of fairc20 mint to avoid causing various unexpected problems.
commit url
https://github.com/glyph-exchange/glyph-stake-contracts/commit/0e6d4cf49770c2db13df0d70f45722bfa49c2974

GEX-8:Ownership change should use two-step process

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	Yaodao

Code Reference

- code/contracts/swap/SwapFactory.sol#L8

```
8: contract SwapFactory is Ownable, ISwapFactory {
```

Description

Yaodao: The contract **`SwapFactory`** uses the openzeppelin's **`Ownable`** contract to control the owner role which will transfer the ownership directly.

```
contract SwapFactory is Ownable, ISwapFactory
```

It is possible that the **`onlyOwenr`** role mistakenly transfers ownership to the wrong address, resulting in the loss of the onlyOnwer role.

Recommendation

Yaodao: Recommend implementing a two-step process where the owner nominates an account and the nominated account needs to call an acceptOnwership() function for the transfer of the ownership to fully succeed.

Client Response

Yaodao: fixed:
All Ownable has been used instead of Ownable2Step
commit url:
https://github.com/glyph-exchange/glyph-stake-contracts/commit/19bf6c3b2cd517245fd6dbbcd56c0c638983c288

GEX-9:Ownable: Does not implement 2-Step-Process for transferring ownership

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	0xzoobi

Code Reference

- code/contracts/swap/SwapFactory.sol#L4

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

Description

0xzoobi: The contracts `SwapFactory.sol` does not implement a 2-Step-Process for transferring ownership.
So ownership of the contract can easily be lost when making a mistake when transferring ownership.
Since the privileged roles have critical function roles assigned to them. Assigning the ownership to a wrong user can be disastrous.
So Consider using the Ownable2Step contract from OZ (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable2Step.sol) instead.
The way it works is there is a `transferOwnership` to transfer the ownership and `acceptOwnership` to accept the ownership. Refer the above Ownable2Step.sol for more details.

Recommendation

0xzoobi: Implement 2-Step-Process for transferring ownership via `Ownable2Step`.

Client Response

0xzoobi: fixed:
All Ownable has been used instead of Ownable2Step
commit url:
https://github.com/glyph-exchange/glyph-stake-contracts/commit/19bf6c3b2cd517245fd6dbbcd56c0c638983c288

GEX-10:Missing Sender-Recipient Address Check in Token Transfer Function

Category	Severity	Client Response	Contributor
Logical	Low	Acknowledged	0xzoobi

Code Reference

- code/contracts/swap/SwapERC20.sol#L57-L61

```
57: function _transfer(address from, address to, uint value) private {
58:         balanceOf[from] = balanceOf[from].sub(value);
59:         balanceOf[to] = balanceOf[to].add(value);
60:         emit Transfer(from, to, value);
61:     }
```

Description

0xzoobi: The **`_transfer`** function is a private function used for transferring tokens from one address to another within the contract. It subtracts the transferred amount from the sender's balance and adds it to the recipient's balance. However, there's a crucial check missing to ensure that the sender (**`from`**) and recipient (**`to`**) addresses are not the same. This check is important for maintaining the integrity of token transfers and preventing unintended behavior, such as self-transfers or accidental loss of tokens.

Recommendation

0xzoobi: It is recommended to include a check in the **`_transfer`** function to ensure that the sender (**`from`**) and recipient (**`to`**) addresses are different. This can be achieved by adding a simple **`require`** statement at the beginning of the function to enforce this invariant:

```
function _transfer(address from, address to, uint value) private {
    require(from != to, "Sender and recipient cannot be the same address");
    balanceOf[from] = balanceOf[from].sub(value);
    balanceOf[to] = balanceOf[to].add(value);
    emit Transfer(from, to, value);
}
```

Client Response

0xzoobi: Acknowledged.

GEX-11:Mint and Burn Allowed when the freeze is set to true

Category	Severity	Client Response	Contributor
Privilege Related	Low	Fixed	0xzoobi

Code Reference

- code/contracts/swap/SwapPair.sol#L152
- code/contracts/swap/SwapPair.sol#L183

```
152: function mint(address to) external lock returns (uint liquidity) {
```

```
183: function burn(address to) external lock returns (uint amount0, uint amount1) {
```

Description

0xzoobi: The `swap` function in the SwapPair contract is equipped with the `onlyNotFrozen` modifier, which prevents swaps when the `freeze` variable is set to true. However, equivalent modifier checks are absent in the mint and burn functions. This oversight allows users to call these functions without any restrictions, even when the pair is frozen, which is not desirable behavior

Recommendation

0xzoobi: Add the missing modifier `onlyNotFrozen` to functions that are not allowed to be called when the pair is frozen.

Client Response

0xzoobi: fixed:
abandon freeze function, delete freeze code
commit url:
https://github.com/glyph-exchange/glyph-stake-contracts/commit/4c1b2985b6305a2ba4da4c797190de3660fa640a

GEX-12:Lack of the check for address(0)

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	Yaodao

Code Reference

- code/contracts/swap/SwapFactory.sol#L66-69
- code/contracts/swap/SwapFactory.sol#L71-74
- code/contracts/swap/SwapFactory.sol#L76-79
- code/contracts/swap/SwapFactory.sol#L81-83

```
66: function setFeeTo(address _feeTo) external override {
67:         require(msg.sender == feeToSetter, "GlyphDEX Swap: FORBIDDEN");
68:         feeTo = _feeTo;
69:     }
```

```
71: function setMigrator(address _migrator) external override {
72:         require(msg.sender == feeToSetter, "GlyphDEX Swap: FORBIDDEN");
73:         migrator = _migrator;
74:     }
```

```
76: function setFeeToSetter(address _feeToSetter) external override {
77:         require(msg.sender == feeToSetter, "GlyphDEX Swap: FORBIDDEN");
78:         feeToSetter = _feeToSetter;
79:     }
```

```
81: function setFreezerSetter(address _freezerSetter) external override onlyOwner {
82:         freezerSetter = _freezerSetter;
83:     }
```

Description

Yaodao: These privileged functions lack checks for **`address(0)`**. If the given parameter is address(0), it will cause the **`feeTo/migrator/feeToSetter/freezerSetter`** address to be address(0) and the **`feeToSetter`** can't gain the privileged role again.

Recommendation

Yaodao: Recommend adding the check of **`address(0)`**.

Client Response

Yaodao: fixed:
we allow feeTo/migrator as address(o) to control the opening or closing of related functions.
check address(o) when call setFeeToSetter function.
commit url:
https://github.com/glyph-exchange/glyph-stake-contracts/commit/4c1b2985b6305a2ba4da4c797190de3660fa640a

GEX-13:Lack check of `MINIMUM_LIQUIDITY`

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	Yaodao

Code Reference

- code/contracts/swap/SwapPair.sol#L163-166

```
163: if (msg.sender == migrator) {
164:                 liquidity = IMigrator(migrator).desiredLiquidity();
165:                 require(liquidity > 0 && liquidity != type(uint256).max, "Bad desired liquidit
y");
166:             } else {
```

Description

Yaodao: According to the following codes, **`IMigrator(migrator).desiredLiquidity()`** will be called when the **`msg.sender`** is **`migrator`**. The **`liquidity`** returned is only checked over 0 and is not **`type(uint256).max`**.
As a result, when the **`liquidity`** is very low, it has the potential to be attacked.
Reference: https://ethereum.stackexchange.com/questions/132491/why-minimum-liquidity-is-used-in-dex-like-uniswap

Recommendation

Yaodao: Recommend adding the following check:.

```
        require(liquidity >= MINIMUM_LIQUIDITY, "Too low desired liquidity");
```

Client Response

Yaodao: fixed:
alter the condition of require, require(liquidity >= MINIMUM_LIQUIDITY && liquidity != type(uint256).max, "Bad desired liquidity");
commit url:
https://github.com/glyph-exchange/glyph-stake-contracts/commit/0db6764da574523c18ca814c16a63457a19af2dd

GEX-14:No need to use SafeMath in solidity version which is above 0.8

Category	Severity	Client Response	Contributor
Language Specific	Informational	Acknowledged	ginlee, Yaodao

Code Reference

- code/contracts/swap/SwapERC20.sol#L4
- code/contracts/swap/SwapERC20.sol#L7

```
4: import '../libraries/SafeMath.sol';
```

```
7: using SafeMath for uint;
```

Description

ginlee: Solidity 0.8.0 introduced built-in arithmetic overflow checks, eliminating the need to use the SafeMath library to prevent overflows
Yaodao: The library `SafeMath` is used to deal with the overflow in calculations. However, the contract `SwapERC20` uses the solidity version `^0.8.19` which will automatic dealing with of overflow in calculations.
As a result, the use of `SafeMath` is unnecessary.

Recommendation

ginlee: Just remove SafeMath library when building with solidity version which is above 0.8.0
Yaodao: Recommend removing the use of `SafeMath`.

Client Response

ginlee: Acknowledged.
Yaodao: Acknowledged.

GEX-15:Liquidity Migration Allows Multiple Executions, Potentially Diluting the Liquidity Pool

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	BradMoonUESTC

Code Reference

- code/contracts/swap/SwapPair.sol#L163-L166

```
163: if (msg.sender == migrator) {
164:                 liquidity = IMigrator(migrator).desiredLiquidity();
165:                 require(liquidity > 0 && liquidity != type(uint256).max, "Bad desired liquidit
y");
166:             } else {
```

Description

BradMoonUESTC: within the **`mint`** function, there's a conditional branch that handles liquidity migration through a designated migrator contract. This mechanism is intended to facilitate the transfer of liquidity from one pool to another, a common feature for upgrading or migrating between versions of decentralized finance (DeFi) protocols.
The problematic section:

```
if (_totalSupply == 0) {
    address migrator = ISwapFactory(factory).migrator();
    if (msg.sender == migrator) {
        liquidity = IMigrator(migrator).desiredLiquidity();
        require(liquidity > 0 && liquidity != type(uint256).max, "Bad desired liquidity");
    } else {
        require(migrator == address(0), "Must not have migrator");
        liquidity = Math.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY);
        _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY toke
ns
    }
}
```

The issue arises in the scenario where **`msg.sender`** is the designated migrator. The contract fetches the desired amount of liquidity to migrate through **`IMigrator(migrator).desiredLiquidity()`**. However, the code does not check whether migration has already been performed. This omission could allow the migrator to execute the migration multiple times, potentially diluting the liquidity pool by repeatedly minting new liquidity tokens without equivalent asset backing, assuming the external migrator contract allows for such a scenario.

Recommendation

BradMoonUESTC: To mitigate this risk and ensure liquidity migration can only occur once, a state variable should be introduced to flag whether migration has already taken place. This flag would be checked upon attempting migration, and set after successful execution, thus preventing multiple migrations.
Here's a suggested code modification to implement this safeguard:

1. Add a state variable to track migration status:

```
bool private migrationCompleted = false;
```

2. Modify the liquidity migration condition to check this variable:

```
if (_totalSupply == 0) {
    require(!migrationCompleted, "Migration already completed");
    address migrator = ISwapFactory(factory).migrator();
    if (msg.sender == migrator) {
        liquidity = IMigrator(migrator).desiredLiquidity();
        require(liquidity > 0 && liquidity != type(uint256).max, "Bad desired liquidity");
        migrationCompleted = true; // Set the flag after successful migration
    } else {
        require(migrator == address(0), "Must not have migrator");
        liquidity = Math.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY);
        _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY toke
ns
    }
}
```

Client Response

BradMoonUESTC: Acknowledged.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3's prior written consent in each instance.

This report is not an "endorsement" or "disapproval" of any particular project or team. This report is not an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3's position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.