NEX Audit Report

Aug 05, 2022

Table of Contents

Summary

This report has been prepared for NEX Audit Report smart contract, to discover issues and vulnerabilities in the source code of their Smart Contract as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Static Analysis and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

Overview

Project Summary

Project Name	**NEX**
Codebase	**https://github.com/NEX-market/nex-contracts**
Commit	**216c8d00ab7071cad06909646595ae86daac3528**
Language	**Solidity**

Audit Summary

Delivery Date	**Aug 05, 2022**
Audit Methodology	**Static Analysis, Manual Review**
Total Isssues	**8**

[H-1] Direct call Vault.sol#sell() will cause fund loss to all the users of the vault

High

Issue Description

Based on the context, we believe this function should only be called by the `NitManager` . However, the current implementation makes it possible for anyone to call it directly without a cost and it will transfer funds out to the `_receiver` , and cause fund loss to all the users of the vault.

https://github.com/NEX-market/nex-contracts/blob/e1e8e826637ec66d6d747467e397c2c10f2abcc4/contracts/core/NitManager.sol#L200

```
200   uint256 amountOut = vault.sell(_tokenOut, _receiver, usdAmount);
```

https://github.com/NEX-market/nex-contracts/blob/e1e8e826637ec66d6d747467e397c2c10f2abcc4/contracts/core/Vault.sol#L450-L472

```
450   function sell(address _token, address _receiver, uint256 _usdAmount) external
      override nonReentrant returns (uint256) {
451       _validateManager();
452       _validate(whitelistedTokens[_token], 19);
453       useSwapPricing = true;
454
455       updateCumulativeFundingRate(_token, _token);
456
457       uint256 redemptionAmount = getRedemptionAmount(_token, _usdAmount);
458       _validate(redemptionAmount > 0, 21);
459
460       _decreasePoolAmount(_token, redemptionAmount);
461
462       uint256 feeBasisPoints = vaultUtils.getSellUsdFeeBasisPoints(_token,
      _usdAmount);
463       uint256 amountOut = _collectSwapFees(_token, redemptionAmount,
      feeBasisPoints);
464       _validate(amountOut > 0, 22);
```

```

        _transferOut(_token, amountOut, _receiver);

        emit Sell(_receiver, _token, _usdAmount, amountOut, feeBasisPoints);

        useSwapPricing = false;
        return amountOut;
    }
```

Recommendation

Consider only allowing the function to be called by the `NitManager` .

Status

✓ Fixed

[H-2] VaultPriceFeed.sol#getPairPrice() can be manipulated

High

Issue Description

It’s well known that on-chain decentralized oracle based on a AMM is prone to be manipulatable, the ecosystem has witnessed numerous high-profile hacks where the oracle implementation is the primary attack vector. Some of these vulnerabilities are discussed here.

Specifically, we found that the `ethNear` pool on Trisolaris has a relatively low liquidity (about $3M worth of ETH and NEAR tokens combined), an attacker can take a 1000 ETH (worth about 1.5M) flash-loan and swap through the pool back-and-forth and use the manipulated price in between.

https://github.com/NEX-market/nex-contracts/blob/e1e8e826637ec66d6d747467e397c2c10f2abcc4/contracts/core/VaultPriceFeed.sol#L334-L344

```
334  // if divByReserve0: calculate price as reserve1 / reserve0
335  // if !divByReserve1: calculate price as reserve0 / reserve1
336  function getPairPrice(address _pair, bool _divByReserve0) public view returns
     (uint256) {
337      (uint256 reserve0, uint256 reserve1, ) = ITrisolarisPair(_pair).getReserves();
338      if (_divByReserve0) {
339          if (reserve0 == 0) { return 0; }
340          return reserve1.mul(PRICE_PRECISION).div(reserve0);
341      }
342      if (reserve1 == 0) { return 0; }
343      return reserve0.mul(PRICE_PRECISION).div(reserve1);
344  }
```

Recommendation

Consider creating a TWAP price oracle based on Uniswap V2’s official implementation:

https://github.com/Uniswap/v2-periphery/blob/master/contracts/examples/ExampleOracleSimple.sol

Uniswap V2 includes several improvements for supporting manipulation-resistant public price feeds. First, every pair measures (but does not store) the market price at the beginning of each block, before any trades take place. This price is expensive to manipulate because it is set by the last transaction, whether it is a mint, swap, or burn, in a previous block.

See also:

1. *Building an Oracle* from Uniswap v2 Docs: https://docs.uniswap.org/protocol/V2/guides/smart-contract-integration/building-an-oracle
2. *Oracles* from Uniswap v2 Docs: https://docs.uniswap.org/protocol/V2/concepts/core-concepts/oracles

Status

✓ **Fixed**

[M-3] Users/contracts who direct call Vault.sol#buy() mistakenly can lose their funds

Medium

Issue Description

The original implementation of this function in GMX will mint USDG for the caller: `IUSDG(usdg).mint(_receiver, mintAmount);` , the current implementation removed that line and there are also no other ways to update the accounting for the caller.

https://github.com/NEX-market/nex-contracts/blob/e1e8e826637ec66d6d747467e397c2c10f2abcc4/contracts/core/Vault.sol#L421-L448

```
function buy(address _token, address _receiver) external override nonReentrant
returns (uint256) {
    _validateManager();
    _validate(whitelistedTokens[_token], 16);
    useSwapPricing = true;

    uint256 tokenAmount = _transferIn(_token);
    _validate(tokenAmount > 0, 17);

    updateCumulativeFundingRate(_token, _token);

    uint256 price = getMinPrice(_token);

    uint256 usdAmount = tokenAmount.mul(price).div(PRICE_PRECISION);
    usdAmount = usdAmount.mul(PRICE_PRECISION).div(10 ** tokenDecimals[_token]);
    _validate(usdAmount > 0, 18);

    uint256 feeBasisPoints = vaultUtils.getBuyUsdFeeBasisPoints(_token,
usdAmount);
    uint256 amountAfterFees = _collectSwapFees(_token, tokenAmount,
feeBasisPoints);
    uint256 mintAmount = amountAfterFees.mul(price).div(PRICE_PRECISION);
    mintAmount = mintAmount.mul(PRICE_PRECISION).div(10 ** tokenDecimals[_token]);

    _increasePoolAmount(_token, amountAfterFees);

    emit Buy(_receiver, _token, tokenAmount, mintAmount, feeBasisPoints);

    useSwapPricing = false;
    return mintAmount;
}
```

This means that whoever calls this function directly will lose all the funds immediately.

Recommendation

If this is designed to work as an internal function, consider adding access control and only allow the function to be called by the `NitManager` .

Status

✓ Fixed

[M-4] Vault.sol#maxUsdgAmounts and related functions were removed

Medium

Issue Description

The original implementation of `_increaseUsdgAmount()` , `_decreaseUsdgAmount()` in GMX will check and put a constains to the upper limit of the max USDG debt for a token:

`_increaseUsdgAmount()` :

- checks for the upper limit of `usdgAmounts[_token]` (when there is a `maxUsdgAmounts[_token]`)

https://github.com/gmx-io/gmx-contracts/blob/787d767e033c411f6d083f2725fb54b7fa956f7e/contracts/core/Vault.sol#L1152-L1159

```
    function _increaseUsdgAmount(address _token, uint256 _amount) private {
        usdgAmounts[_token] = usdgAmounts[_token].add(_amount);
        uint256 maxUsdgAmount = maxUsdgAmounts[_token];
        if (maxUsdgAmount != 0) {
            _validate(usdgAmounts[_token] <= maxUsdgAmount, 51);
        }
        emit IncreaseUsdgAmount(_token, _amount);
    }
```

`_decreaseUsdgAmount()` :

- deduct from `usdgAmounts[_token]`
- resets `usdgAmounts[_token] = 0;` when `_amount > usdgAmounts[_token]`

https://github.com/gmx-io/gmx-contracts/blob/787d767e033c411f6d083f2725fb54b7fa956f7e/contracts/core/Vault.sol#L1161-L1173

```
    function _decreaseUsdgAmount(address _token, uint256 _amount) private {
        uint256 value = usdgAmounts[_token];
        // since USDG can be minted using multiple assets
        // it is possible for the USDG debt for a single asset to be less than
zero
        // the USDG debt is capped to zero for this case
        if (value <= _amount) {
            usdgAmounts[_token] = 0;
            emit DecreaseUsdgAmount(_token, value);
            return;
        }
        usdgAmounts[_token] = value.sub(_amount);
        emit DecreaseUsdgAmount(_token, _amount);
    }
```

The current implementation removed the `maxUsdgAmounts` and both `_increaseUsdgAmount()` , `_decreaseUsdgAmount()` , making it unable to limit the max debt for a certain token.

This can be dangerous, especially when a token is not a stable major asset and is still being whitelisted, in which case, when the price of such token drops quickly, the market is incentivized to take more loans with such token and exit all other assets, sacrifice the stability of the whole protocol.

Status

✓ **Fixed**

[L-5] VaultPriceFeed.sol#getAmmPrice() is not absolutely safe from overflow

Low

Issue Description

`getPairPrice()` will return the price amplified by `1e30`.

At L328, once `price0.mul(price1)` exceeds `type(uint256).max`, the transaction reverts due to overflow and malfunctions all the dependent features.

https://github.com/gmx-io/gmx-contracts/blob/master/contracts/core/VaultPriceFeed.sol#L338-L352

While the original version is using `bnbBusd` as `price0` and `ethBnb` / `btcBnb` as `price1`, it's relatively safe.

However, in the current implementation, `price0.mul(price1)` (current value: `~0.75e69`) can be pretty close to the overflow threshold `~1e77`:

Pair	token0	token1	current price
nearUsdc	USDC	NEAR	2.5e41
ethNear	NEAR	ETH	3e27

While it's not an immediate threat given the current market price of Near and ETH, but we could still say that it's not absolutely safe from overflow.

https://github.com/NEX-market/nex-contracts/blob/e1e8e826637ec66d6d747467e397c2c10f2abcc4/contracts/core/VaultPriceFeed.sol#L319-L344

```
319  function getAmmPrice(address _token) public override view returns (uint256) {
320      if (_token == near) {
321          return getPairPrice(nearUsdc, true);
322      }
323
324      if (_token == eth) {
325          uint256 price0 = getPairPrice(ethNear, true);
326          uint256 price1 = getPairPrice(nearUsdc, true);
327          // this calculation could overflow if (price0 / 10**30) * (price1 /
     10**30) is more than 10**17
328             return price0.mul(price1).div(PRICE_PRECISION);
329         }
330 
331         return 0;
332     }
333 
334     // if divByReserve0: calculate price as reserve1 / reserve0
335     // if !divByReserve1: calculate price as reserve0 / reserve1
336     function getPairPrice(address _pair, bool _divByReserve0) public view returns (uint256) {
337         (uint256 reserve0, uint256 reserve1, ) = ITrisolarisPair(_pair).getReserves();
338         if (_divByReserve0) {
339             if (reserve0 == 0) { return 0; }
340             return reserve1.mul(PRICE_PRECISION).div(reserve0);
341         }
342         if (reserve1 == 0) { return 0; }
343         return reserve0.mul(PRICE_PRECISION).div(reserve1);
344     }
```

Recommendation

Consider changing L328 to:

```
return price0.div(PRICE_PRECISION).mul(price1);
```

Status

✓ **Fixed**

[G-6] Unnecessary usage of .div() and .mul() adds gas cost and precision loss

Gas

Issue Description

https://github.com/NEX-market/nex-contracts/blob/e1e8e826637ec66d6d747467e397c2c10f2abcc4/contracts/core/Vault.sol#L433-L434

```
uint256 usdAmount = tokenAmount.mul(price).div(PRICE_PRECISION);
usdAmount = usdAmount.mul(PRICE_PRECISION).div(10 ** tokenDecimals[_token]);
```

The unnecessary `.div()` and `.mul()` calculation adds gas cost. They can be removed to save gas and make the code cleaner.

Also, the first `.div()` can cause precision loss when the token is very cheap.

Recommendation

Change to:

```
uint256 usdAmount = tokenAmount.mul(price).div(10 ** tokenDecimals[_token]);
```

Status

✓ Fixed

[G-7] Using immutable variable can save gas

Gas

Issue Description

https://github.com/NEX-market/nex-contracts/blob/e1e8e826637ec66d6d747467e397c2c10f2abcc4/contracts/core/VaultUtils.sol#L27

```
INitManager public nitManager;
```

https://github.com/NEX-market/nex-contracts/blob/e1e8e826637ec66d6d747467e397c2c10f2abcc4/contracts/core/VaultUtils.sol#L34

```
constructor(IVault _vault, INitManager _nitManager) {
    vault = _vault;
    nitManager = _nitManager;
}
```

Considering that `nitManager` will never change, changing it to immutable variable instead of storage variable can save gas.

Status

✓ Fixed

[G-8] Unnecessary storage write to useSwapPricing in multiple functions of Vault.sol

Gas

Issue Description

The current implementation of `getMaxPrice()` , `getMinPrice()` removed the reference of `useSwapPricing` .

https://github.com/NEX-market/nex-contracts/blob/e1e8e826637ec66d6d747467e397c2c10f2abcc4/contracts/core/Vault.sol#L712-L718

```
function getMaxPrice(address _token) public override view returns (uint256) {
    return IVaultPriceFeed(priceFeed).getPrice(_token, true, includeAmmPrice);
}

function getMinPrice(address _token) public override view returns (uint256) {
    return IVaultPriceFeed(priceFeed).getPrice(_token, false, includeAmmPrice);
}
```

The original implementation from GMX:

https://github.com/gmx-io/gmx-contracts/blob/787d767e033c411f6d083f2725fb54b7fa956f7e/contracts/core/Vault.sol#L761-L767

```
function getMaxPrice(address _token) public override view returns (uint256) {
    return IVaultPriceFeed(priceFeed).getPrice(_token, true, includeAmmPrice, useSwapPricing);
}

function getMinPrice(address _token) public override view returns (uint256) {
    return IVaultPriceFeed(priceFeed).getPrice(_token, false, includeAmmPrice, useSwapPricing);
}
```

Therefore, the storage writes of `useSwapPricing` at L424, L446, L453, L470, L480, L510 in `buy()` , `sell()` , and `swap()` is no longer needed.

https://github.com/NEX-market/nex-contracts/blob/e1e8e826637ec66d6d747467e397c2c10f2abcc4/contracts/core/Vault.sol#L421-L512

```
421  function buy(address _token, address _receiver) external override nonReentrant returns (uint256) {
422      _validateManager();
423      _validate(whitelistedTokens[_token], 16);
424      useSwapPricing = true;
425
426      uint256 tokenAmount = _transferIn(_token);
427      _validate(tokenAmount > 0, 17);
428
429      updateCumulativeFundingRate(_token, _token);
430
431      uint256 price = getMinPrice(_token);
432
433      uint256 usdAmount = tokenAmount.mul(price).div(PRICE_PRECISION);
434      usdAmount = usdAmount.mul(PRICE_PRECISION).div(10 ** tokenDecimals[_token]);
435      _validate(usdAmount > 0, 18);
436
437      uint256 feeBasisPoints = vaultUtils.getBuyUsdFeeBasisPoints(_token, usdAmount);
438      uint256 amountAfterFees = _collectSwapFees(_token, tokenAmount, feeBasisPoints);
439      uint256 mintAmount = amountAfterFees.mul(price).div(PRICE_PRECISION);
440      mintAmount = mintAmount.mul(PRICE_PRECISION).div(10 ** tokenDecimals[_token]);
441
442      _increasePoolAmount(_token, amountAfterFees);
443
444      emit Buy(_receiver, _token, tokenAmount, mintAmount, feeBasisPoints);
445
446      useSwapPricing = false;
447      return mintAmount;
448  }
449
450  function sell(address _token, address _receiver, uint256 _usdAmount) external override nonReentrant returns (uint256) {
451      _validateManager();
452      _validate(whitelistedTokens[_token], 19);
453      useSwapPricing = true;
454
```

```
    updateCumulativeFundingRate(_token, _token);

    uint256 redemptionAmount = getRedemptionAmount(_token, _usdAmount);
    _validate(redemptionAmount > 0, 21);

    _decreasePoolAmount(_token, redemptionAmount);

    uint256 feeBasisPoints = vaultUtils.getSellUsdFeeBasisPoints(_token,
_usdAmount);
    uint256 amountOut = _collectSwapFees(_token, redemptionAmount,
feeBasisPoints);
    _validate(amountOut > 0, 22);

    _transferOut(_token, amountOut, _receiver);

    emit Sell(_receiver, _token, _usdAmount, amountOut, feeBasisPoints);

    useSwapPricing = false;
    return amountOut;
}

function swap(address _tokenIn, address _tokenOut, address _receiver) external
override nonReentrant returns (uint256) {
    _validate(isSwapEnabled, 23);
    _validate(whitelistedTokens[_tokenIn], 24);
    _validate(whitelistedTokens[_tokenOut], 25);
    _validate(_tokenIn != _tokenOut, 26);

    useSwapPricing = true;

    updateCumulativeFundingRate(_tokenIn, _tokenIn);
    updateCumulativeFundingRate(_tokenOut, _tokenOut);

    uint256 amountIn = _transferIn(_tokenIn);
    _validate(amountIn > 0, 27);

    uint256 priceIn = getMinPrice(_tokenIn);
    uint256 priceOut = getMaxPrice(_tokenOut);

    uint256 amountOut = amountIn.mul(priceIn).div(priceOut);
    amountOut = amountOut.mul(10 ** tokenDecimals[_tokenOut]).div(10 **
tokenDecimals[_tokenIn]);

```

```
    uint256 usdAmount = amountIn.mul(priceIn).div(PRICE_PRECISION);
    usdAmount = usdAmount.mul(PRICE_PRECISION).div(10 ** tokenDecimals[_tokenIn]);

    uint256 feeBasisPoints = vaultUtils.getSwapFeeBasisPoints(_tokenIn, _tokenOut, usdAmount);
    uint256 amountOutAfterFees = _collectSwapFees(_tokenOut, amountOut, feeBasisPoints);


    _increasePoolAmount(_tokenIn, amountIn);
    _decreasePoolAmount(_tokenOut, amountOut);

    _validateBufferAmount(_tokenOut);

    _transferOut(_tokenOut, amountOutAfterFees, _receiver);

    emit Swap(_receiver, _tokenIn, _tokenOut, amountIn, amountOut, amountOutAfterFees, feeBasisPoints);

    useSwapPricing = false;
    return amountOutAfterFees;
}
```

Status

✓ Fixed

Appendix

Timeliness of content

The content contained in the report is current as of the date appearing on the report and is subject to change without notice, unless indicated otherwise by WatchPug; however, WatchPug does not guarantee or warrant the accuracy, timeliness, or completeness of any report you access using the internet or other means, and assumes no obligation to update any information following publication.

Disclaimer