About

0xkmmm & dyoff are a team of independent smart contract security researchers and developers. With a wealth of experience conducting smart contract security reviews and uncovering numerous vulnerabilities in live smart contracts. For inquiries about security reviews, feel free to contact us via Telegram or Twitter at @auditsbydanny and @0xkmmm.

Disclaimer

Audits are a time-, resource-, and expertise-bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can reveal the presence of vulnerabilities **but cannot guarantee their absence**.

Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - involves a small loss of funds or affects a core functionality of the protocol.
- **Low** - encompasses any unexpected behavior that is non-critical.

Likelihood

- **High** - a direct attack vector; the cost is relatively low compared to the potential loss of funds.
- **Medium** - only a conditionally incentivized attack vector, with a moderate likelihood.
- **Low** - involves too many or unlikely assumptions; offers little to no incentive.

Actions required by severity level

- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

Executive summary

0xkmmm & dyoff were engaged by Titan Legends to review `TitanLegendsMarketplaceV2` over the period from 24.09.2024 to 27.09.2024.

Project Name: Titan Legends
Repository: Link
Commit hash: `dfa1443055cba09abdd9faed1aa6c65af6f363ff`
Methods: Manual review

High Risk	**2**
Medium Risk	3
Low Risk	1
Informational	6

Vulnerabilities

High Risk

[H-01] `TitanLegendsMarketplaceV2::noContract` Modifier Can Be Bypassed by Constructor Call, Leading to Potential Exploitation

Severity: High Risk
Context: `TitanLegendsMarketplaceV2#L39`

Description: In the `TitanLegendsMarketplaceV2` contract, the function `buyListingEth` allows buyers to purchase listings using Ether (ETH) instead of TitanX tokens. To calculate the amount of ETH required, the contract uses the current spot price from the TitanX/WETH pool.

However, the protection against potential flash loan attacks and other forms of automated exploitation relies on the `noContract` modifier:

```
modifier noContract {
    require(address(msg.sender).code.length == 0, "Contracts are
prohibited");
    _;
}
```

This modifier checks that the `msg.sender` is not a smart contract by ensuring the length of its code is zero. While this prevents direct contract interactions, it can be circumvented by attackers using the following approach:

- An attacker deploys a malicious contract.
- Within the contract's constructor, they manipulate the TitanX/WETH spot price.
- In the same constructor call, they execute `buyListingEth` to purchase listings at a heavily discounted price due to the manipulated exchange rate.
- The contract then self-destructs, removing any trace of the attack.

This attack vector arises because contracts in their constructor phase have a `code.length` of zero, bypassing the `noContract` modifier.

Impact: The attacker can manipulate the TitanX/WETH ratio in a single transaction, allowing them to purchase listings at an artificially low price.

Recommendation: Consider switching from using the pool's spot price for converting TitanX to WETH. A more robust pricing mechanism, such as time-weighted average price (TWAP) oracles, can mitigate the risk of flash loan attacks.

Resolution: Fixed

[H-02] Incorrect Division in `getCurrentEthPrice` Causes Users to Pay Significantly More

Severity: High Risk
Context: `TitanLegendsMarketplaceV2#L79`

Description:
The function `buyListingEth` in `TitanLegendsMarketplaceV2` is responsible for converting a listing's price in TitanX tokens to Ether (ETH). This conversion uses the `getCurrentEthPrice` function, which returns the ratio of 1 TitanX to WETH (Wrapped Ether). However, an incorrect division in the price calculation results in users paying significantly more ETH than intended.

Consider the following example:

- **Listing Price:** 200 TitanX
- **TitanX to ETH Price:** 0.0000000002795 (as of Sept 24th)

The current approach for calculating the ETH price is incorrect:

```
uint256 priceInEth = price / getCurrentEthPrice();
```

Which causes incorrect results:
0.000000055911525000
0.000000715416007701

- Incorrect calculation:

```
200e18 / 279557625 = 715416007701 ( 0.000000715416007701 ETH )
```

- Correct calculation:

```
(200e18 * 279557625) / 1e18 = 55911525000 ( 0.000000055911525000 ETH )
```

The incorrect calculation makes users pay significantly more ETH for listings.

Impact: Users can pay substantially less/more than the intended amount for listings due to incorrect division in the ETH price calculation.

Recommendation: Replace the incorrect division-based calculation with a multiplication-based approach, ensuring the conversion of TitanX to ETH is done properly. The correct formula should be:

```
uint256 priceInEth = (price * getCurrentEthPrice()) / 1e18;
```

Resolution: Fixed

Medium Risk

[M-01] Division Before Multiplication Causes `TitanLegendsMarketplaceV2::getCurrentEthPrice` to return an inflated price

Severity: Medium Risk
Context: `TitanLegendsMarketplaceV2#L141`

Description:
The function `getCurrentEthPrice` in `TitanLegendsMarketplaceV2` calculates the price ratio of 1 TitanX to 1 WETH using the square of the pool's spot price (`sqrtPriceX96`). However, the current implementation divides before multiplying, leading to a loss of precision in the result.

Consider the following code snippet:

```
    function getCurrentEthPrice() public view returns (uint256) {
        IUniswapV3Pool pool = IUniswapV3Pool(TITANX_WETH_POOL);
        (uint160 sqrtPriceX96, , , , , , ) = pool.slot0();

        uint256 priceX192 = uint256(sqrtPriceX96) * uint256(sqrtPriceX96);
        uint256 price = (priceX192 / (1 << 192)) * 1 ether;
        if (WETH9 < address(titanX)) {
            price = (1 ether * 1 ether) / price;
        }

        return price;
    }
```

The issue arises because the division (`priceX192 / (1 << 192)`) is performed before the multiplication by `1 ether`, leading to a truncation of significant digits. The result is an inflated price due to this loss of precision.

Since this value is then used as a denominator when inverting the price (`price = (1 ether * 1 ether) / price`), the loss of precision could lead to `getCurrentEthPrice` returning an inflated price.

Example Calculation:

Given a sample value of `sqrtPriceX96` (as of September 24th):

- **`sqrtPriceX96` Value:**
  `4671143369717780267387307041098717`

The first step of the calculation is to square this value:

- **`priceX192` (squared value):**
  `4671143369717780267387307041098717 * 4671143369717780267387307041098717 = 21819580380458379234325740991841158769806786867319521259342539046089`

Next, the contract divides `priceX192` by `2^192` (a left shift of 192 bits) and multiplies the result by `1e18` to get the ETH price.

- **Current Approach (Incorrect):**
  - Perform the division first:
    `21819580380458379234325740991841158769806786867319521259342539046089 / 2^192 = 3476059700`
  - Multiply by `1e18`:
    `3476059700 * 1e18 = 3,476,059,700,000,000,000,000,000,000`
- **Correct Approach:**
  - Multiply `priceX192` by `1e18` first:
    `21819580380458379234325740991841158769806786867319521259342539046089 * 1e18 = 2181958038045837923432574099184115876980678686731952125934253904608900000000000000000`
  - Then divide by `2^192`:
    `2181958038045837923432574099184115876980678686731952125934253904608900000000000000000 / 2^192 = 3,476,059,700,841,259,953,558,110,830`
- **Inverting the price**:

- ***Current (Incorrect) Result:*** `1e36 / 347605970000000000000000000 = 287682055.633279256`
- ***Correct Result:*** `1e36 / 3,476,059,700,841,259,953,558,110,830 = 287682055.56365577`

By dividing before multiplying, the contract loses 19 decimal places of precision, resulting in an inflated price.

Impact:

- This loss of precision leads to an inflated conversion price of TitanX to WETH, causing users to overpay for purchases made using Ether.
- Such inaccuracies could lead to significant financial losses for buyers and disrupt the fairness of marketplace transactions.

Recommendation: To ensure the price calculation maintains precision, consider changing to:

```
-  uint256 price = (priceX192 / (1 << 192)) * 1 ether;
+  uint256 price = FullMath.mulDiv(priceX192,1e18, 1 << 192);
```

This will ensure that the price calculation reflects the correct value without truncating any significant decimal places.

Resolution: Fixed

[M-02] Vulnerability to Sandwich Attacks Allowing Discounts on NFTs

Severity: Medium Risk
Context: `TitanLegendsMarketplaceV2` - Function `getCurrentEthPrice`

Description:
The `getCurrentEthPrice` function in the `TitanLegendsMarketplaceV2` contract calculates the TITANX/WETH ratio by reading the current spot price from the Uniswap V3 pool's `slot0`. This reliance on the current price introduces a vulnerability to sandwich attacks, which malicious actors can exploit to purchase NFTs at a discount.

Mechanism of the Attack:

A malicious actor can execute a series of transactions to manipulate the price and obtain NFTs at a reduced rate:

1. **Transaction 1: Manipulate the Pool Price**
    - The attacker interacts with the Uniswap V3 pool to adjust the price ratio between TitanX and WETH. This could involve making a large swap that alters the pool's current spot price.
2. **Transaction 2: Purchase the NFT**
    - The attacker then calls the `buyListingEth` function to buy the NFT using the manipulated price from `getCurrentEthPrice`. Since this function relies on the already changed price, the attacker effectively buys the NFT at a discount.
3. **Transaction 3: Return Funds to the Pool**
    - Finally, the attacker can return any excess funds back to the pool, solidifying their profit from the transaction while retaining ownership of the NFT purchased at the inflated discount.

Impact:

- This exploit allows attackers to purchase NFTs at prices significantly lower than intended, undermining the integrity of the marketplace.
- It could lead to financial losses for sellers and damage the overall trust in the platform, as prices are no longer reflective of true market value.

Recommendation:
To mitigate the risk of sandwich attacks, consider implementing the following measures:

1. **Transaction Limitations:** Introduce limits on price changes or restrict the frequency of price updates to reduce the window of opportunity for attackers to manipulate prices.
2. **Slippage Tolerance:** Implement slippage checks within the `buyListingEth` function to ensure that the price has not changed beyond a certain threshold since the transaction was initiated.

Resolution: Fixed

[M-03] Denial of Service (DoS) Risk if `sqrtPriceX96` Exceeds `type(uint128).max`

Severity: Medium Risk
Context: `TitanLegendsMarketplaceV2#L140`

Description:
The `getCurrentEthPrice` function in the `TitanLegendsMarketplaceV2` contract retrieves the current exchange rate between TitanX and WETH using the `sqrtPriceX96` value from a Uniswap V3 pool. However, if the value of `sqrtPriceX96` exceeds the maximum value that can be stored in a `uint128` type (`type(uint128).max`), this can lead to unexpected behavior or a denial of service (DoS).

Potential Vulnerability:

The `sqrtPriceX96` value is used to calculate the price as follows:

1. **Calculation of `priceX192`:**
   - If `sqrtPriceX96` exceeds `type(uint128).max`, the multiplication used to compute `priceX192` will result in an overflow, producing an incorrect value or causing a revert.
2. **Example of Overflow:**
   - The calculation for `priceX192` is performed as:

     ```
     uint256 priceX192 = uint256(sqrtPriceX96) * uint256(sqrtPriceX96);
     ```

   - If `sqrtPriceX96` exceeds `2^64`, the resulting `priceX192` will overflow, leading to reverts due to using solidty > 0.8.

Impact: If an overflow occurs due to `sqrtPriceX96` exceeding `type(uint128).max`, it can lead to complete DoS of `buyListingEth` function.
Recommendation: To mitigate this risk, consider changing the implementation to handle this case:

```
if (sqrtRatioX96 <= type(uint128).max) {
            uint256 ratioX192 = uint256(sqrtRatioX96) * sqrtRatioX96;

            uint256 price = FullMath.mulDiv(ratioX192, 1e18, 1 << 192);

             if (WETH9 < address(titanX)) {
```

```
                price = (1 ether * 1 ether) / price;
            }

        } else {
            uint256 ratioX128 = FullMath.mulDiv(sqrtRatioX96, sqrtRatioX96,
1 << 64);
            uint256 price = FullMath.mulDiv(ratioX128, 1e18, 1 << 128);

             if (WETH9 < address(titanX)) {
                price = (1 ether * 1 ether) / price;
            }
        }
```

Resolution: Fixed

Low Risk

[L-01] Marketplace Fees Can Be Avoided Due to Rounding Down

Severity: Low Risk
Context: `TitanLegendsMarketplaceV2#L65`, `TitanLegendsMarketplaceV2#L81`

Description: The system currently employs a marketplace fee which is deducted from the amount the owner of the listing will receive when the listing is bought, it is calculated as:

```
uint256 _marketplaceFee = (listing.price * marketplaceFee) / 10000;
```

However, if for some reason the listing price is too small the protocol will not receive any fees, due to rounding it down to 0.

Impact: The protocol will not receive any dust fees
Recommendation: Consider rounding up the fees in benefit of the protocol.

Resolution: Fixed

Gas Optimizations

- **Consider using custom errors instead of string errors**
- **Redundant check in `getCurrentEthPrice()`, since we already know that WETH < TITAN_X:**
  - In the WETH/TITAN X pool the titanX is `token1`, which means TITAN X > WETH
- **`uint 2`costs less compared to uint * uint`**

Informational

- **Missing 0 address checks in constructors**
- **`TitanLegendsMarketplaceV2::addListing` logic can be simplified**

```
    function addListing(uint256 tokenId, uint256 price) external
nonReentrant noContract {
        require(price > 0, "Price must be greater than zero");
-       uint256 listingId = currentListingId;
+       uint256 listingId = currentListingId++; //@note -> This way
`listingId`, will be assigned to currentListingId before incrementing it
        collection.safeTransferFrom(msg.sender, address(this), tokenId);

        listings[listingId] = Listing(tokenId, price, msg.sender);
        activeListings.add(listingId);
-        currentListingId++;

        emit ListingAdded(listingId, tokenId, msg.sender, price);
    }
```

- **Due to the `TitanLegendsMarketplaceV2::noContract` modifier users with smart wallets will not be able to interact with the contract**