SpartaDex - Staking Dex Audit Report

Prepared by bytes032

Contents

1 About bytes032

bytes032 is an independent smart contract security researcher.

His knack for identifying smart contract security vulnerabilities in a range of protocols is more than a skill; it's a passion. Committed to enhancing the blockchain ecosystem, he invests his time and energy into thorough security research and reviews. Want to know more or collaborate? bytes's always up for a chat about all things security.

Feel free to reach out on X.

2 Protocol Summary

SpartaDEX is a combination of real-time strategy game set in the realities of ancient Greece and a decentralized cryptocurrency exchange. They call it a **gamified DEX**. The main goal is to provide the exchange with user engagement known from video games, which builds loyalty and consistency in providing liquidity.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of {X} according to the specific commit. Any modifications to the code will require a new security review.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Overview

Project	SpartaDex - Staking Dex
Repository	sdex-smart-contracts
Commit	c4bee5399024. . .
Date	June 2023

Issues Found

Severity	**Count**
Critical Risk	1
High Risk	2
Medium Risk	1
Low Risk	2
Informational	0
Gas Optimizations	0
Total Issues	**6**

Summary of Findings

Title	**Status**
[C-1] Replay attack in PolisManagers upgradeWithSignature	Resolved
[H-1] Unclaimed tokens remain stuck in SpartaStaking smart contract	Resolved
[H-2] Unintended excess funds retention in WithFees contract	Resolved
[M-1] EIP712 compliance issue in PaymentReceiver.sol	Resolved
[L-1] Mismatch in UPGRADE TYPE hash parameters in PolisManagers upgradeHash function	Resolved
[L-2] Unhandled failures in token transfer operations	Resolved

5 Findings

5.1 Critical Risk

[C-1] Replay attack in PolisManagers upgradeWithSignature

Context: PolisManager.sol

Impact: The lack of checks for signature reuse in the `upgradeWithSignature` function presents a vulnerability that can lead to indefinite token upgrades.

Description: The `upgradeWithSignature` function in PolisManager contract is used to upgrade a locked token. However, this function doesn't enforce any checks to prevent the reuse of a previously used signature.

```
 function upgradeWithSignature(
       uint256 tokenId,
       uint8 level,
       bytes calldata signature
   ) external override {
       bytes32 messageHash = upgradeHash(tokenId, level);
       address signer = messageHash.toEthSignedMessageHash().recover(
           signature
       );
       _ensureHasUpgradeRole(signer);
       // @audit-issue signature replay attack
       if (polis.lockerOf(tokenId) != address(this)) {
           revert OnlyIfLocked();
       }
       polis.upgrade(tokenId);
   }
```

This presents a significant risk, as once a signature has been used, it can be reused indefinitely to perform multiple upgrades on a token.

Additionally, the lack of an expiry mechanism for the signature is a minor concern, as it leaves the signature indefinitely valid, thereby extending the period of vulnerability.

Recommendation

1. **Implement Signature Tracking**: To mitigate the risk of replay attacks, implement a mechanism to track used signatures. For instance, you could use a mapping to store used signatures and check against this mapping whenever the `upgradeWithSignature` function is called. If a signature is found in the mapping, the function should revert to prevent the upgrade.

```
mapping (bytes => bool) usedSignatures;

function upgradeWithSignature(
    uint256 tokenId,
    uint8 level,
    bytes calldata signature
) external override {
    require(!usedSignatures[signature], "Signature has already been used");
    // ...
    usedSignatures[signature] = true;
}
```

2. **Add Expiry to Signature**: Introduce a mechanism to set an expiry for the signatures. The `upgradeHash` could be modified to include an expiry timestamp and the `upgradeWithSignature` function should check whether the signature has expired.

5.2 High Risk

[H-1] Unclaimed tokens remain stuck in SpartaStaking smart contract

Context: SpartaStaking.sol

Impact

A significant amount of tokens (1 token per second over the duration of 1 hour in the provided example) remains stuck in the contract indefinitely. This unclaimed portion could grow substantial over time if there are delays in staking after the `initialize(...)` function is called. This **will** result in loss of funds for the stakeholders.

Description: In the current implementation of the `SpartaStaking` contract, a vulnerability exists where tokens meant for staking rewards may become irretrievable and stuck within the contract.

Let's consider that you have a `SpartaStaking` contract with a reward duration of one month seconds (2592000):

You call `initialize(...)` with a reward of one month seconds (2592000) only. **The intention** is for a period of a month, 1 reward token per second should be distributed to stakers.

The initial state is: `rewardRate = 1` `duration = 2592000` `start = 1686483210` `updatedAt = start` `finishAt() = 1689075210 = (1686483210 + 2592000)`

Then, 1 hour (3600) passes and the first staker calls `stake(...)` to stake some amount.

At this call, `rewardPerToken is 0`, because there there are no other stakers.

https://github.com/SpartaDEX/sdex-smart-contracts/blob/0369f48e3b2a655616dc75afdf87b9915f64d720/contracts/staking/SpartaStaking.sol#L80-L82

```
        if (totalSupply == 0 || block.timestamp < start) {
            return rewardPerTokenStored;
        }
```

And now `updatedAt = 1686486810` (1 hour after start)

Hence, for this staker, the clock starts from now, and he or she will accumulate rewards from this point.

```
(rewardRate * (lastTimeRewardApplicable() - updatedAt) * 1e18) /
```

As a result 2592000-3600= 2588400 tokens will be distributed, and these 3600 tokens will be stuck in the contract **forever** because there's no token recovery function.

Recommendation: To mitigate this issue, I recommend one of the two approaches:

1. **Setting an "End" Time:** Instead of starting the rewards distribution immediately upon initialization, consider a model where the "end" time is defined and the rewards distribution starts only when the first `stake(...)` function is called. This ensures that no tokens will be left unclaimed.
2. **Implement a Recovery Function:** If the first solution doesn't fit into the overall design, an alternative is to implement a recovery function that allows the contract owner to retrieve unclaimed tokens after a certain period. This would ensure the stuck tokens can be put back into circulation.

[H-2] Unintended excess funds retention in WithFees contract

Context: WithFees.sol

Impact: Users who send more funds than the required fee are susceptible to an unintended loss of their excess funds. The contract does not refund the surplus funds, effectively causing the users to pay higher fees than necessary.

Description: The `WithFee's` contract in question employs a modifier called `onlyWithFees`, which is used to enforce native coin fees on certain functions across Staking/Vesting contracts. The modifier employs a local variable named `value` that is compared against `msg.value`, which represents the amount of native coins sent by the sender.

```
    modifier onlyWithFees() {
        if (value > msg.value) {
            revert OnlyWithFees();
        }

        // @audit-issue doesn't refund
        _;
    }
```

As seen in the code above, if the sender sends an amount less than the value, the modifier will revert the transaction.

However, if the sender sends an amount greater than the required fee (`value`), the contract does not handle the excess funds sent, and they are effectively locked within the contract. This means that a user who inadvertently sends too much will suffer a loss.

Recommendation: To rectify this issue, the contract should either refund the excess amount sent by the sender or enforce strict equality between the `value` and `msg.value`. Implementing either of these solutions will prevent users from unintentionally losing funds.

Option 1: Returning the excess funds

```
modifier onlyWithFees() {
    if (value > msg.value) {
        revert OnlyWithFees();
    }
    if (value < msg.value) {
        // Refund the excess amount
        payable(msg.sender).transfer(msg.value - value);
    }

    _;
}
```

Option 2: Requiring strict equality

```
modifier onlyWithFees() {
    if (value != msg.value) {
        revert OnlyWithFees();
    }

    _;
}
```

5.3 Medium Risk

[M-1] EIP712 compliance issue in PaymentReceiver.sol

Context: PaymentReceiver.sol

Impact: The non-compliance with EIP712 standards in PaymentReceiver.sol could lead to improper domain separation and incorrect hashing of variables. This may result in unexpected behaviour

Description: PaymentReceiver.sol's buyGemsHash function is designed to allow users buy wallet tolkens if the signature is signed by a wallet with GAMES_TRADER role.

It also incorporates EIP712. However, its current implementation is **NOT** EIP712 compliant.

```
function buyGemsHash(
       address _wallet,
       uint256 _tokenId,
       uint256 _amount,
       uint256 _price,
       uint256 _deadline
   ) public view override returns (bytes32) {
       return
           keccak256(
               abi.encodePacked(
                   uint16(0x1901),
                   keccak256(
                       abi.encode(
                           keccak256(
                               "EIP712Domain(string name, string version, address verifyingContract, uint256 signedAt)"
                           ),
                           keccak256(bytes("SPARTA")),
                           keccak256(bytes("1")),
                           _chainId(),
                           address(this),
                           keccak256(bytes("SPARTA_GENS"))
                       )
                   ),
                   _wallet,
                   _tokenId,
                   _amount,
                   _price,
                   _deadline
               )
           );
   }
```

There are two problems. First, the domain separator is wrong, its supposed to be `"EIP712Domain(string name, string version, address verifyingContract, uint256 signedAt)"`

but the variables that are passed are "string(name), string(version), uint(chainId), address, string(salt)"

Whereas, as per EIP712 the ordering is different.

The second problem is that the gems hash variables are hashed incorrectly. These https://github.com/SpartaDEX/sdex-smart-contracts/blob/2588db7d10bec32bf9b178107c923899154d1c98/contracts/payment-receiver/PaymentReceiver.sol#L94-L98

```
                   _wallet,
                   _tokenId,
                   _amount,
                   _price,
                   _deadline
```

Note: `PolisManager` has the same issue related to the domain separator.

Recommendation:

1. **Fix the Domain Separator**: The domain separator should be corrected to match the EIP712 standard. The corrected keccak256 function should be:

```
keccak256( "EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)" );
```

2. **Fix the Encoded Data**: The hashing of the gems variables should be corrected to match the EIP712 standard. An example of the correct implementation can be found here

5.4 Low Risk

[L-1] Mismatch in UPGRADE TYPE hash parameters in PolisManagers upgradeHash function

Context: PolisManager.sol

Impact: This issue may lead to improper calculation of the hash and can affect the intended functionality of the contract. It might not have immediate severe consequences, but if users hash on their own instead of using "upgradeHash", this would result in invalid hashes.

Description: There is a discrepancy between the parameters defined in the `UPGRADE_TYPE` hash and the actual parameters passed to the `upgradeHash` function in the PolisManager contract. The `UPGRADE_TYPE` hash is defined to include three variables: `tokenId`, `level`, and `signature`.

```
    string constant UPGRADE_TYPE =
        "upgrade(uint256 tokenId, uint8 level, bytes signature)";
```

However, the `upgradeHash` function, which is supposed to use this type hash, is only called with two parameters - `tokenId` and `level`.

```
    function upgradeHash(
        uint256 tokenId,
        uint8 level
    ) public view override returns (bytes32) {
```

Recommendation: The `UPGRADE_TYPE` constant should be updated to reflect the actual parameters that are passed to the `upgradeHash` function. Specifically, the `signature` parameter should be removed from the `UPGRADE_TYPE` constant. This will ensure that the type hash correctly represents the parameters being passed.

```
string constant UPGRADE_TYPE = "upgrade(uint256 tokenId, uint8 level)";
```

[L-2] Unhandled failures in token transfer operations

Context: PolisMinter.sol, SpartaRewardLPLinearStaking.sol, LinearStaking.sol

Impact: This vulnerability has a low severity rating. While it does not lead to the direct loss of funds or manipulation of contract logic, it can hinder expected contract behavior when interacting with certain ERC20 tokens, like USDT.

Description: There are several instances in the `PolisMinter.sol`, `SpartaRewardLPLinearStaking.sol`, and `LinearStaking.sol` contracts where `transfer` and `transferFrom` are utilized to move tokens. These operations check if the transfer operation returns true or false and revert the transaction if the operation returns false. For example:

```
     if (
            !paymentToken.transferFrom(
                msg.sender,
                address(paymentReceiver),
                paymentValue
            )
        ) {
            revert PaymentFiled();
        }
```

```
 if (!rewardToken.transfer(msg.sender, toTransfer)) {
            revert TransferFailed();
        }
```

```
        if (!rewardToken.transfer(msg.sender, reward)) {
            revert TransferFailed();
        }
```

However, this method is incompatible with certain tokens, like USDT. The reason is that the USDT contract does not return a boolean value after a transfer operation, and thus does not comply with the common ERC20 interface. Therefore, the aforementioned code segments would always revert when dealing with USDT, which would result in the failure of legitimate operations.

Recommendation: In order to avoid this issue and provide compatibility with a wider range of tokens, it is recommended to use the `safeTransfer` and `safeTransferFrom` functions provided by the OpenZeppelin library. These functions ensure that the transfer operation behaves as expected, by reverting the transaction if the transfer fails, even if the token contract does not return a boolean value. Thus, these functions would work correctly with all tokens that follow the ERC20 standard, including those that do not strictly comply with the common interface.