PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Preliminary Report

For SmarDex (Update)

23 November 2023

paladinsec.co

info@paladinsec.co

Table of Contents

Disclaimer

Paladin Blockchain Security ("Paladin") has conducted an independent audit to verify the integrity of and highlight any vulnerabilities or errors, intentional or unintentional, that may be present in the codes that were provided for the scope of this audit. This audit report does not constitute agreement, acceptance or advocation for the Project that was audited, and users relying on this audit report should not consider this as having any merit for financial advice in any shape, form or nature. The contracts audited do not account for any economic developments that may be pursued by the Project in question, and that the veracity of the findings thus presented in this report relate solely to the proficiency, competence, aptitude and discretion of our independent auditors, who make no guarantees nor assurance that the contracts are completely free of exploits, bugs, vulnerabilities or deprecation of technologies. Further, this audit report shall not be disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Paladin.

All information provided in this report does not constitute financial or investment advice, nor should it be used to signal that any persons reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report. Information is provided 'as is', and Paladin is under no covenant to the completeness, accuracy or solidity of the contracts audited. In no event will Paladin or its partners, employees, agents or parties related to the provision of this audit report be liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this audit report.

Cryptocurrencies and any technologies by extension directly or indirectly related to cryptocurrencies are highly volatile and speculative by nature. All reasonable due diligence and safeguards may yet be insufficient, and users should exercise considerable caution when participating in any shape or form in this nascent industry.

The audit report has made all reasonable attempts to provide clear and articulate recommendations to the Project team with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts provided. It is the sole responsibility of the Project team to sufficiently test and perform checks, ensuring that the contracts are functioning as intended, specifically that the functions therein contained within said contracts have the desired intended effects, functionalities and outcomes of the Project team.

Paladin retains the right to re-use any and all knowledge and expertise gained during the audit process, including, but not limited to, vulnerabilities, bugs, or new attack vectors. Paladin is therefore allowed and expected to use this knowledge in subsequent audits and to inform any third party, who may or may not be our past or current clients, whose projects have similar vulnerabilities. Paladin is furthermore allowed to claim bug bounties from third-parties while doing so.

1 Overview

This report has been prepared for SmarDex's updated contracts on the Ethereum network. Paladin provides a user-centred examination of the smart contracts to look for vulnerabilities, logic errors or other issues from both an internal and external perspective.

This audit focuses on a diff-based evaluation of the SmarDex Protocol, and specifically targets the updated smart contracts. Paladin will analyze the changes introduced in the recent update, assessing any new vulnerabilities or security implications arising from these modifications.

Please refer to Section 1.2 below to see a list of all contracts that were assessed in the scope of this audit.

1.1 Summary

Project Name	Smardex.io
URL	https://smardex.io/
Network	Ethereum
Language	Solidity
Preliminary	https://github.com/petra-foundation/smardex-paladin-audit-2/tree/ff6016bea6ebb197b12a8bdf0c1d91d7e0a59fab/contracts

1.2 Methodology

The audit will treat the previously audited contracts as a black box. Our primary focus will be on the differences between the old and updated versions, examining how these changes interact with the existing codebase.

We will assess:

- the scurity vulnerabilities introduced by the updates,
- any logical inconsistencies or flaws resulting from the changes, and
- the impact of the modifications on the overall protocol architecture and functionality.

This audit will conclude with a comprehensive report that outlines any identified risks, potential vulnerabilities, and recommendations for improvements to ensure the integrity and security of SmarDex.

The commit that was previously audited is https://github.com/SmarDex-Dev/smart-contracts-updatable-fees/tree/dc05e390fbc86cd5ca9919a44f14dabd300389c4/contracts

The commit which includes the updates is https://github.com/petra-foundation/smardex-paladin-audit-2/tree/ff6016bea6ebb197b12a8bdf0c1d91d7e0a59fab/contracts

1.3 Contracts Assessed

Name	Contract	Live Code Match
SmardexFactory	0xB878DC600550367e14220d4916Ff678fB284214F	✓ MATCH
SmardexPair	0x007597aAF6A4E4652e7843aF5C44c359321368c2 Also dependency (SmardexFactory)	✓ MATCH
SmardexLibrary	Dependency (SmardexFactory, SmardexPair, AutoSwapper)	✓ MATCH
SmardexRouter	0xC33984ABcAe20f47a754eF78f6526FeF266c0C6F	✓ MATCH
Autoswapper	0x346c0cA93354383a31d78d4944290D51F3b3F920	✓ MATCH
FarmingRange	Not deployed	N/A
RewardManagerWithdrawable	Not deployed	N/A
PoolAddress	Dependency (SmardexRouter)	✓ MATCH
PoolHelpers	Dependency (SmardexRouter)	✓ MATCH
Staking	Not deployed	N/A

1.4 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
● Governance	2	-	-	2
● High	0	-	-	-
● Medium	1	-	-	1
● Low	3	-	-	3
● Informational	0	-	-	-
Total	**6**	**0**	**0**	**6**

Classification of Issues

Severity	Description
● Governance	Issues under this category are where the governance or owners of the protocol have certain privileges that users need to be aware of, some of which can result in the loss of user funds if the governance's private keys are lost or if they turn malicious, for example.
● High	Exploits, vulnerabilities or errors that will certainly or probabilistically lead towards loss of funds, control, or impairment of the contract and its functions. Issues under this classification are recommended to be fixed with utmost urgency.
● Medium	Bugs or issues that may be subject to exploit, though their impact is somewhat limited. Issues under this classification are recommended to be fixed as soon as possible.
● Low	Effects are minimal in isolation and do not pose a significant danger to the project or its users. Issues under this classification are recommended to be fixed nonetheless.
● Informational	Consistency, syntax or style best practices. Generally pose a negligible level of risk, if any.

1.4.1 SmardexFactory

ID	Severity	Summary	Status
01	GOV	Invalid/malicious pair can be pushed into the factory storage	ACKNOWLEDGED
02	LOW	Pairs that were unintentionally not added cannot be added once whitelist is closed	ACKNOWLEDGED

1.4.2 SmardexPair

No issues found.

1.3.3 SmardexLibrary

No issues found.

1.3.4 SmardexRouter

ID	Severity	Summary	Status
03	MEDIUM	Inconsistency when fetching pairs can result in issues	ACKNOWLEDGED
04	LOW	Safeguard might overflow	ACKNOWLEDGED
05	LOW	Potential mismatch in frontend and Solidity calculation can result in transaction revert	ACKNOWLEDGED

1.3.5 Autoswapper

ID	Severity	Summary	Status
06	GOV	Contract owner can benefit from own swap	ACKNOWLEDGED

1.3.6 FarmingRange

No issues found.

1.3.7 RewardManagerWithdrawable

No issues found.

1.3.8 PoolAddress

No issues found.

1.3.9 PoolHelpers

No issues found.

1.3.10 Staking

No issues found.

2 Findings

2.1 SmardexFactory

The following diffchecker link highlights the changes which have been introduced: https://www.diffchecker.com/D9WRS5v8/

Most notably, a whitelist mechanism was introduced in the updated contract: modifiers have been aded to check if the whitelist is open or closed (`onlyIfWhitelistIsOpen`, `onlyIfWhitelistIsClosed`).

Normal pair creation via the `createPair` function can now only be done if the whitelist is closed. A `closeWhitelist` function allows the owner to close the whitelist, and once the whitelist is closed, it can no longer be reopened. If the whitelist is open, a pair assignment can only be done via the `addPair` function, which basically pushes an already existing pair into the factory storage.

This logic was implemented to migrate existing pairs into the new factory storage.

2.1.1 Issues & Recommendations

Issue #01	Invalid/malicious pair can be pushed into the factory storage
Severity	● GOVERNANCE
Description	Since the only input parameter for the `addPair` function is the `_pair` address, the owner can simply input an invalid or malicious pair into the factory's storage. This can potentially result in issues down-stream, and ultimately lost funds. Furthermore, any migrated pairs should be added into the router's whitelist mechanism such that the `PoolAddress.pairFor` function fetches the correct pair. After reviewing the Router, an edge-case scenario was identified where the governance is able to steal tokens that users have approved to the router. Consider this PoC: 1. Call the `addPair` function and add a malicious contract which exposes the correct interface. 2. Call the `addPairToWhitelist` function in the router with the desired tokens, and it will now fetch the assigned address from point 1 and assign this to the whitelist. 3. This contract can now call various functions within the Router, including but not limited to `smardexSwapCallback` and `smardexMintCallback`, potentially allowing this contract to steal tokens from users which have approved them to the router.
Recommendation	Consider strictly keeping an eye on the validity of the pairs that are added.
Resolution	● ACKNOWLEDGED

Issue #02	Pairs that were unintentionally not added cannot be added once whitelist is closed
Severity	● LOW SEVERITY
Description	If the owner closes the whitelist without having all necessary pairs added, it is impossible to add an existing pair to the factory storage afterwards, which will then again result in down-stream issues.
Recommendation	Consider being very careful with the migration and double-check if all pairs have been migrated.
Resolution	● ACKNOWLEDGED

2.2 SmardexPair

The following diffchecker link highlights the changes which have been introduced: https://www.diffchecker.com/9lUfKALr/

The only change to the pair contract was the addition of a `skim` function which allows anyone to withdraw `token0` and `token1` from the pair as long as it is uninitialized (`totalSupply = 0`).

2.2.2 Issues & Recommendations

No issues found.

2.3 SmardexLibrary

The following diffchecker link highlights the changes which have been introduced: https://www.diffchecker.com/GQ9mBjGP/

The `approximateEqual` function has been modified to immediately return `true` if `x=y`. Within the contracts normal business logic, the previous functionality did not have any issues; however, since the approximation value is 1/1 000 000, this function will return `false` for values below 1 000 000. The reason for this can be illustrated in the following example.

Assuming `x = 100 000` and `y = 100 000`, the function should in fact return `true` because the intent is to evaluate if both values are equal with a deviation of 0.0001%, which is in fact true here. However, in that scenario, the function will move in to the `else` block, which executes the following calculation:

```
return _y < (_x + (_x * APPROX_PRECISION) / APPROX_PRECISION_BASE);
```

In actual fact, this function will return false, since `100 000 * 1 / 1 000 000 = 0`, hence x is not larger than y.

2.3.1 Issues & Recommendations

No issues found.

2.4 SmardexRouter

The following diffchecker link highlights the changes which have been introduced: https://www.diffchecker.com/oGUErdTb/

The following changes have been implemented:

- Whitelist functionality: As already mentioned within the `SmardexFactory` contract, the contract owner can migrate old pairs to the new factory. It is important to understand that the router contract usually fetches a corresponding pair for a swap or liquidity execution directly from the factory or uses the pre-deterministic nature of the deployment by using the factory address, salt and init-code hash. However, as the pair is migrated and deployed via the old factory, the pre-deterministic approach will return the pair which has or will be deployed by the new factory. To counter this issue, the `addPairToWhitelist` function has been implemented, which basically pushes the old pair address to the `tokenHash` of the new pair address from the corresponding new factory. Whenever a swap or liquidity execution is executed, the router will call the `pairFor` function from the `PoolAddress` library, which in the first step checks if the corresponding `tokenHash` has an old pair assigned and then returns the address of the migrated pair.

1. Adjustment of the liquidity addition logic: The user is now allowed to pass the following parameters to the function call:
   1. `tokenA`
   2. `tokenB`
   3. `amountADesired`
   4. `amountBDesired`
   5. `amountAMin`
   6. `amountBMin`
   7. `fictiveReserveB`
   8. `fictiveReserveAMin`

9. `fictiveReserveAMax`

These parameters allow the user to prevent any potential loss from a malicious frontrunning transaction via the `_product` check within the `_addLiquidity` function.

The Smardex team described the mechanism as follows.

Case 1:

1. The pool is currently empty.
2. Alice wants to add 100 USDT and 100 USDC of liquidity to the pool.
3. `amountADesired = 100e18` and `amountBDesired = 100e18`, with a slippage of 1%.
4. The frontend calculates the following parameters:

    `amountAMin = 99e18`

    `amountBMin = 99e18`

    `fictiveReservesB = 100e18` (the frontend will use `amountBDesired`)

    `fictiveReserveAMin = 99e18`

    `fictiveReserveAMax = 101e18`

If there is a front-running manipulation, the transaction will revert. This aligns with the code snippet in the router.

Case 2:

1. The pool has liquidity.
2. The current reserves are at a 1:1 ratio of 100 USDT and 100 USDC.
3. `fictiveReserveA = 50e18`, `fictiveReserveB = 50e18`
4. Alice wants to add 1 USDT and 1 USDC with a slippage of 1%.
5. The frontend calculates the following parameters:

    `amountAMin = 0.99`

    `amountBMin = 0.99`

    `fictiveReserveB = 50e18`

```
        fictiveReserveAMax = 49.5e18
        fictiveReserveAMax = 50.5e18
```

If there is a front-running manipulation, the transaction will revert. This aligns with the code snippet in the router.

It is important to mention that if a pool is already in such a state and users willingly accept any price, this can still result in a loss of funds as shown in the above two PoCs.

- Additionally, the following functions were included with `permit` logic: `swapExactTokensForETHWithPermit`, `swapTokensForExactETHWithPermit`, `swapTokensForExactTokensWithPermit`, `swapExactTokensForTokensWithPermit`. The permit call was also wrapped into a try/catch block.

2.4.1 Issues & Recommendations

Issue #03	Inconsistency when fetching pairs can result in issues
Severity	● MEDIUM SEVERITY
Description	Whenever liquidity is added, the `_addLiquidity` function is invoked, which fetches the pair from the factory and then fetches the corresponding reserves for the pair to execute a slippage check The slight issue here is that these two pair-fetching methodologies are different — in the first method, the pair is fetched directly from the factory and in the second one, the pair is fetched from the `PoolHelpers` library. In the scenario of a migrated but non-whitelisted pair, this will fetch the address of the migrated pair in the first step but fetch the address of the pair which will be deployed from the new factory in the second step (deterministic approach). This can result in issues depending on which pair the frontend uses to fetch the reserves from. In the scenario where the frontend is using the `pairFor` method to calculate the slippage parameters but the user will in fact add liquidity to the migrated pair, then the slippage check is totally void.
Recommendation	Firstly, it must be ensured that all migrated pairs are whitelisted in the router, and secondly, it might make sense to reconsider the fetching logic. While it is great that the `getPair` function is used instead of the `pairOf` function in the first step (`pairFor` would not work here since it always returns a valid address even if the pair is non-existent, which would prevent the creation of a new pair), it must be ensured that no mismatch can occur here. Furthermore, instead of updating two whitelists (one in the factory and one in the router) consider using only one to prevent edge cases. Consider removing the whitelist of the router and in `PoolAddress` library and simply use the `getPair` function from the factory. Note that this would only increase the gas cost slightly as a storage read is already needed if using the router's whitelist.

	This would also make it easier for the SmarDex team to update the pairs as currently, the pairs need to be whitelisted in the router first, then in the factory as the router checks that the `getPair` function of the factory returns the zero address. **<u>Do note that for significant code changes in the resolution round, a revalidation fee may apply.</u>**
Resolution	● ACKNOWLEDGED

Issue #04	**Safeguard might overflow**
Severity	● LOW SEVERITY
Location	`require(_product <= _params.fictiveReserveAMax * _reserveBFic, "SmarDexRouter: PRICE_TOO_HIGH")`
Description	The above line might result in an overflow in the scenario of specific tokens in the pair, such as tokens with 21 decimals or a very large supply.
Recommendation	Since this is a very specific edge-case, we recommend the SmarDex team to undergo specific tests, especially tests with already existing exotic pairs. Further considerations should be done if this scenario in fact happens during these tests.
Resolution	● ACKNOWLEDGED

Issue #05	Potential mismatch in frontend and Solidity calculation can result in transaction revert
Severity	● LOW SEVERITY
Description	Since the frontend presumably calculates values using JavaScript, there might be issues if the pair is in such a state that a very low amount of liquidity was added as potential calculations on the Solidity side then round down and a mismatch occurs between the JavaScript and Solidity calculations. For small slippages, this might eventually revert when it should be going through.
Recommendation	Consider carefully examining this issue and eventually adjusting the calculation on the frontend side to align with Solidity's calculations.
Resolution	● ACKNOWLEDGED

2.5 Autoswapper

The following diffchecker link highlights the changes which have been introduced: https://www.diffchecker.com/utJ65u9e/

The following changes have been implemented:

- The Ownable library has been inherited.

- A `swapTokensWithPath` function was implemented, which allows the owner to swap any arbitrary token within the Autoswapper contract to the SmarDex token with the `StakingContract` as recipient.

2.5.1 Issues & Recommendations

Issue #06	Contract owner can benefit from own swap
Severity	● GOVERNANCE
Description	Whenever `swapTokenWithPath` is called, the owner can pass a `_amountOutMin` parameter towards this function. This allows the owner to potentially sandwich-attack their own swap and benefit from it.
Recommendation	Consider keeping a strong governance architecture.
Resolution	● ACKNOWLEDGED The Smardex team is a well-known and responsible team with doxxed members and have also had a long track-record. Therefore, we do not expect any malicious behavior.

2.6 FarmingRange

The following diffchecker link highlights the changes which have been introduced: https://www.diffchecker.com/A7ybYQHZ/

The following changes were introduced:

- The `addRewardInfo` function has been modified to not allow the distribution of rewards retroactively and preventing rewards from being allocated to time ranges that have already passed.

- A `removeLastRewardInfoMultiple` function has been implemented which allows the owner to remove multiple rewardInfos from a campaign.

- The `deposit` function was modified to allow depositing on behalf of other users.

- The `withdraw` function was modified and now allows withdrawing rewards and the staked amount to different addresses.

2.6.1 Issues & Recommendations

No issues found.

2.7 RewardManagerWithdrawable

The following diffchecker link highlights the changes which have been introduced: https://www.diffchecker.com/IcZJUilj/

In summary, `RewardManagerWithdrawable` is the exact same contract as `RewardManagerL2`, with the only difference being that the owner of the farming contract can withdraw funds.

2.7.1 Issues & Recommendations

No issues found.

2.8 PoolAddress

The following diffchecker link highlights the changes which have been introduced: https://www.diffchecker.com/L6IFhfgj/

The `PoolAddress` contract has been modified to handle the migration logic correctly. The difference to the previous implementation is that the `pairFor` function allows for a whitelist parameter which simply checks if the `tokenHash` has already an assigned pair in the whitelist mapping. In that scenario, it means that the returned pair address is the migrated pair which was deployed by the old factory. If that is not the case, the deterministic address based on the new factory will be returned.

It is important to double check the new init code hash — `c762a0f9885cc92b9fd8eef224b75997682b634460611bc0f2138986e20b653f` —to ensure that it is in fact the same as the `initcode` from the deployed factory, as the init code can change based on compiler settings.

2.8.1 Issues & Recommendations

No issues found.

2.9 PoolHelpers

The following diffchecker link highlights the changes which have been introduced: https://www.diffchecker.com/6TxVmRUy/

The changes are as follows:

- The `getReserves` function was modified to support the migrated pairs. The change is the same as that applied to the router: the `pairFor` function within the `PoolAddress` library is now called with a whitelist parameter.

- The same change has been applied to the `getFictiveReserves` function.

- A `getAllReserves` function has been implemented, which fetches both the real and the fictive reserves and returns them, while ensuring that the return order is the same as the provided tokens. This function is used within the liquidity addition of the router to ensure that liquidity is added for a correct pair state (correct price/`fictiveReserve` ratio)

- The `priceAverage` function was modified to support migrated pairs.

2.9.1 Issues & Recommendations

No issues found.

2.10 Staking

The following diffchecker link highlights the changes which have been introduced: https://www.diffchecker.com/PyRSM0zm/

The only change is the modification of the `deposit` and `harvest` call to the `FarmingRange` contract which now includes a third parameter for the `to` address.

2.10.1 Issues & Recommendations

No issues found.

PALADIN
BLOCKCHAIN SECURITY