AlphaX Audit Report

Signature: b8cd7973d0b36a5e2f27d09243959692ab101ec34f511b68b0026e116d93281c

AlphaX Audit Report

Overview

This document details our collaborative engineering effort with AlphaX team regarding their protocol.

Project Name	AlphaX
Repository Link	AlphaX-Protocol-Contract
Commit	Revert "add to check" 5966ac4
Language	Solidity - Ethereum
Scope	contracts/**/*.sol

Disclaimer

The audit does not ensure that it has identified every security issue in the smart contracts, and it should not be seen as a confirmation that there are no more vulnerabilities. The audit is not exhaustive, and we recommend further independent audits and setting up a public bug bounty program for enhanced security verification of the smart contracts. Additionally, this report should not be interpreted as personal financial advice or recommendations.

Auditing Process

- Static Analysis: We perform static analysis using our internal tools and Slither to identify potential vulnerabilities and coding issues.
- Fuzz Testing: We execute fuzz testing with our internal fuzzers to uncover potential bugs and logic flaws.
- Invariant Development: We convert the project into Foundry project and develop Foundry invariant tests for the project based on the code semantics and documentations.
- Invariant Testing: We run multiple fuzz testing tools, including Foundry and ItyFuzz, to identify violations of invariants we developed.
- Formal Verification: We develop individual tests for critical functions and leverage Halmos to prove the functions in question are not vulnerable.
- Manual Code Review: Our engineers manually review code to identify potential vulnerabilities not captured by previous methods.

Vulnerability Severity

We divide severity into four distinct levels: high, medium, low, and info. This classification helps prioritize the issues identified during the audit based on their potential impact and urgency.

- **High Severity Issues** represent critical vulnerabilities or flaws that pose a significant risk to the system's security, functionality, or performance. These issues can lead to severe consequences such as fund loss, or major service disruptions if not addressed immediately. High severity issues typically require urgent attention and prompt remediation to mitigate potential damage and ensure the system's integrity and reliability.
- **Medium Severity Issues** are significant but not critical vulnerabilities or flaws that can impact the system's security, functionality, or performance. These issues might not pose an immediate threat but have the potential to cause considerable harm if left unaddressed over time. Addressing medium severity issues is important to maintain the overall health and efficiency of the system, though they do not require the same level of urgency as high severity issues.
- **Low Severity Issues** are minor vulnerabilities or flaws that have a limited impact on the system's security, functionality, or performance. These issues generally do not pose a significant risk and can be addressed in the regular maintenance cycle. While low severity issues are not critical, resolving them can help improve the system's overall quality and user experience by preventing the accumulation of minor problems over time.
- **Informational Severity Issues** represent informational findings that do not directly impact the system's security, functionality, or performance. These findings are typically observations or recommendations for potential improvements or optimizations. Addressing info severity issues can enhance the system's robustness and efficiency but is not necessary for the system's immediate operation or security. These issues can be considered for future development or enhancement plans.

Below is a summary of the vulnerabilities with their current status, highlighting the number of issues identified in each severity category and their resolution progress.

	Number	Resolved
High Severity Issues	0	0
Medium Severity Issues	0	0
Low Severity Issues	2	2
Informational Severity Issues	2	2

Findings

[Low] Potential Gas Loss

Description:

Since `to` can be controlled by the attackers, the attackers can set it to a contract that mints gas tokens (CHI tokens) or uses massive gas. If the backend signs and sends transactions blindly based on the `eth_estimateGas` result for the gas fee, the sender account may lose a significant amount of ETH due to gas consumption.

```
(bool success, ) = to.call{value: amount}("");
require(
    success,
    "Address: unable to send value, recipient may have reverted"
);
```

Recommendation:

- Add a codesize check to ensure `to` is not a contract; or
- Add a threshold to the maximum amount of gas spent for each transaction.

Status: Acknowledged

[Low] Potential Risks in the Withdrawal Function

Description:

There is a potential risk of duplicate signatures in the current `withdrawETH/withdrawERC20` function implementation. In some cases, such as transaction failure, the signer may be induced to re-sign the same transaction. Yet, the signature may already be leaked onchain, and users can use that signature to withdraw the assets again. This may lead to repeated withdrawal of funds (i.e., double spending) or other unexpected behaviors, threatening the security and integrity of the contract.

1. *Retry after transaction failure*
When an on-chain transaction fails for various reasons (such as insufficient gas), the backend may try to re-initiate the transaction.
2. *Network delay*
Due to network delay, the backend may not be able to obtain the transaction status in time, resulting in repeated transaction submissions.

3. *Manipulation of `to` parameters*

By setting `to` to a malicious contract, the attacker can force the ETH transfer to fail, subsequently making the whole transaction fail.

```
(bool success, ) = to.call{value: amount}("");
require(
    success,
    "Address: unable to send value, recipient may have reverted"
);
```

Recommendation:

- Invalidate previous signature: Create another mapping to record the revoked signature. The signer signs again only after the revoke transaction is mined; or
- Implement a global/per-user nonce mechanism: The signer maintains a local nonce, signs with the nonce, and the contract checks the nonce. If the nonce in the transaction is smaller than the contract's current nonce, revert the transaction; or
- Do not re-sign upon failure: Retry with an updated gas price, then manually resolve if the transaction still fails.

Status: Acknowledged

[Info] `withdrawETHByOwner` Function Uses `transfer` Instead of `call`

Description:

In the DEXVault contract, the `withdrawETHByOwner` function uses the `transfer` method (max 2300 gas) to send ETH instead of the safer `call` method. This transfer would fail if `to` is a contract (e.g., transferring to a gnosis-safe contract).

```
payable(to).transfer(address(this).balance);
```

Recommendation:

Replace the `transfer` method with the `call` method and ensure proper return value handling. For example:

```
(bool success, ) = payable(to).call{value: amount}("");
require(success, "ETH transfer failed");
```

Status: Acknowledged

[Info] Unused `USDT_ADDRESS` State Variable

Description:

In the `DEXVault` contract, a public state variable named `USDT_ADDRESS` is declared but never used throughout the contract.

```
address public USDT_ADDRESS;
```

Recommendation:
If `USDT_ADDRESS` is indeed not needed, it should be removed.

Status: Acknowledged