BLOCKSEC

Security Audit

Report for AlphaX-Protocol-Contract-Tron

Date: April 30, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Chapter 1 Introduction **1**

1.1 About the Audit Target . . . 1
1.2 Disclaimer . . . 2
1.3 Procedure of Auditing . . . 2
1.3.1 Security Issues . . . 2
1.3.2 Additional Recommendation . . . 3
1.4 Security Model . . . 3

Chapter 2 Findings **5**

2.1 Security Issue . . . 6
2.1.1 Lack of verification for `permit.gasFreeAddress` . . . 6
2.1.2 Lack of critical signature field . . . 7
2.1.3 Lack of TRC20 transfer result handling in `withdrawERC20()` . . . 9
2.1.4 Lack of whitelist check in the function `depositETH()` . . . 9
2.1.5 Improper owner attribution in withdrawal events . . . 10
2.1.6 Lack of constructor parameters in `getGasFreeAccountCreationCodeHash()` 12
2.2 Recommendation . . . 12
2.2.1 Return the prefixed hash from function `calcSigHash()` . . . 12
2.2.2 Use consistent TRON native asset naming and unit semantics . . . 14
2.2.3 Add `_disableInitializers()` in the function `constructor()` . . . 14
2.2.4 Remove redundant code . . . 15
2.2.5 Adopt two-step `transferOwnership()` ownership handover . . . 15
2.3 Note . . . 16
2.3.1 TRC-10 asset compatibility . . . 16
2.3.2 TRON address format conversion . . . 16
2.3.3 Predicted CREATE2 addresses depend on the variable `controller` . . . 17
2.3.4 Assumption of supported token . . . 17
2.3.5 TRON typed-data signing depends on address normalization . . . 18
2.3.6 Potential centralization risks . . . 18
2.3.7 Centralized delay mechanism . . . 19
2.3.8 Permanent controller binding for accounts . . . 19
2.3.9 Ensure the correct declaration of `permit.firstTime` . . . 19
2.3.10 Ethereum-style signing prefix and naming on TRON . . . 20
2.3.11 Asymmetric gasless support between TRX and TRC20 . . . 20

Report Manifest

Item	Description
Client	AlphaX-Protocol
Target	AlphaX-Protocol-Contract-Tron

Version History

Version	Date	Description
1.0	April 30, 2026	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-04-30 08:38:09+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About the Audit Target

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The audit target (hereinafter referred to as the Target) of this audit is the code repository [1] of AlphaX-Protocol-Contract-Tron of AlphaX-Protocol.

This project is a TRON-based custody vault and gasless transfer system. The contract `DEXVaultV1` is an upgradeable custody vault that allows users to deposit assets and permits withdrawals when predefined risk controls are satisfied. The gasless wallet layer consists of the contract `GasFreeController`, the contract `GasFreeAccount`, and the contract `GasFreeFactory`. Users can store funds in their own `GasFreeAccount` contracts, sign transactions off-chain, and rely on a relayer to pay for transaction costs during execution. The contract `GasFreeController` handles EIP-712-based permit execution, while the contract `GasFreeFactory` deploys deterministic accounts through CREATE2. Note that for contract `DEXVaultV1`, we only audit its compatibility with Tron.

Note this audit only focuses on the smart contracts in the following directories/files:

- DEXVaultV1.sol
- GasFreeController.sol
- GasFreeAccount.sol
- GasFreeFactory.sol

Other files are not within the scope of the audit. Additionally, all dependencies of the Target are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
AlphaX-Protocol-Contract-Tron	`Version 1`	`6e874e8337955b5053a5cd53c86a6af22002b23a`
	`Version 2`	`e10c87af2ed6f46623d29800e162f590ad78b4b6`

[1] https://github.com/AlphaX-Protocol/AlphaX-Protocol-Contract-Tron

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the Target, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of the Target.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan the Target with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of the Target and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness

- Oracle security
- Business logic correctness
- Semantic and functional consistency
- Emergency mechanism
- Economic and incentive impact

1.3.2 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **six** potential security issues. Besides, we have **five** recommendations and **eleven** notes.

- Medium Risk: 2
- Low Risk: 4
- Recommendation: 5
- Note: 11

ID	Severity	Description	Category	Status
1	Medium	Lack of verification for `permit.gasFreeAddress`	Security Issue	Fixed
2	Medium	Lack of critical signature field	Security Issue	Fixed
3	Low	Lack of TRC20 transfer result handling in `withdrawERC20()`	Security Issue	Confirmed
4	Low	Lack of whitelist check in the function `depositETH()`	Security Issue	Confirmed
5	Low	Improper owner attribution in withdrawal events	Security Issue	Fixed
6	Low	Lack of constructor parameters in `getGasFreeAccountCreationCodeHash()`	Security Issue	Confirmed
7	-	Return the prefixed hash from function `calcSigHash()`	Recommendation	Confirmed
8	-	Use consistent TRON native asset naming and unit semantics	Recommendation	Confirmed
9	-	Add `_disableInitializers()` in the function `constructor()`	Recommendation	Confirmed
10	-	Remove redundant code	Recommendation	Confirmed
11	-	Adopt two-step `transferOwnership()` ownership handover	Recommendation	Confirmed
12	-	TRC-10 asset compatibility	Note	-
13	-	TRON address format conversion	Note	-
14	-	Predicted CREATE2 addresses depend on the variable `controller`	Note	-
15	-	Assumption of supported token	Note	-
16	-	TRON typed-data signing depends on address normalization	Note	-
17	-	Potential centralization risks	Note	-
18	-	Centralized delay mechanism	Note	-
19	-	Permanent controller binding for accounts	Note	-

20	-	Ensure the correct declaration of `permit.firstTime`	Note	-
21	-	Ethereum-style signing prefix and naming on TRON	Note	-
22	-	Asymmetric gasless support between TRX and TRC20	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Lack of verification for `permit.gasFreeAddress`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `GasFreeController`, the function `executePermitDepositVault()` deposits `permit.value` into the vault after instructing the input account `permit.gasFreeAddress` to transfer TRX. However, account validation relies solely on `IGasFreeAccount(permit.gasFreeAddress).owner()`. The function does not verify that the account is factory-issued and the controller balance delta is greater than `permit.value` after invoking `transferMainCoin()`. As a result, an attacker can provide a forged account contract to spend residual TRX held by the controller, causing the vault deposit to be funded by the controller rather than by the claimed account.

```
185   function executePermitDepositVault(PermitTransfer calldata permit, bytes calldata signature)
          external nonReentrant {
186       require(vault != address(0), "GasFreeController: vault not set");
187       require(permit.deadline >= block.timestamp, "GasFreeController: Permit expired");
188
189       bytes32 structHash = _hashPermit(permit);
190       bytes32 digest = _hashTypedDataV4(structHash);
191       address signer = ECDSA.recover(digest, signature);
192
193       require(signer == permit.user && signer != address(0), "GasFreeController: Invalid
              signature");
194       require(nonces[permit.user] == permit.nonce, "GasFreeController: Invalid nonce");
195       require(IGasFreeAccount(permit.gasFreeAddress).owner() == permit.user, "GasFreeController:
              account not owned by user");
196
197       nonces[permit.user]++;
198
199       uint256 totalFee = permit.token == address(0) ? transferFeeTRX : transferFee;
200       if (permit.firstTime) {
201           totalFee += permit.token == address(0) ? activateFeeTRX : activateFee;
202       }
```

Listing 2.1: contracts/GasFreeController.sol

```
208     if (permit.token == address(0)) {
209         require(permit.gasFreeAddress.balance >= totalCost, "GasFreeController: Insufficient
                TRX balance");
210         IGasFreeAccount account = IGasFreeAccount(permit.gasFreeAddress);
211         account.transferMainCoin(permit.serviceProvider, totalFee);
212         account.transferMainCoin(address(this), permit.value);
213         IDEXVault(vault).depositETH{value: permit.value}(permit.receiver);
214     } else {
215         if (!tokenApprovals[permit.gasFreeAddress][permit.token]) {
216             IGasFreeAccount(permit.gasFreeAddress).approveToken(permit.token);
217             tokenApprovals[permit.gasFreeAddress][permit.token] = true;
218         }
219         IERC20 token = IERC20(permit.token);
220         require(token.balanceOf(permit.gasFreeAddress) >= totalCost, "GasFreeController:
                Insufficient balance");
221         token.transferFrom(permit.gasFreeAddress, permit.serviceProvider, totalFee);
222         token.transferFrom(permit.gasFreeAddress, address(this), permit.value);
223         token.approve(vault, permit.value);
224         IDEXVault(vault).depositERC20(permit.token, permit.value, permit.receiver);
225     }
```

Listing 2.2: contracts/GasFreeController.sol

Impact An attacker can spend residual TRX held by the controller, causing the vault deposit to be funded by the controller rather than by the claimed account.

Suggestion Validate that `permit.gasFreeAddress` is a legitimate account created by `GasFreeFactory`, and verify the controller balance delta before forwarding TRX into the vault.

Clarification from BlockSec The function `executePermitDepositVault()` now verifies the TRX balance delta, but does not verify that `permit.gasFreeAddress` is a legitimate account deployed by the contract `GasFreeFactory`. This leaves a residual risk of untrusted account execution, though the practical impact is limited.

2.1.2 Lack of critical signature field

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Functions `executePermitTransfer()` and `executePermitDepositVault()` verify the permit signature, then perform TRC20 transfers or deposits into the vault, respectively. These functions allow anyone to invoke and perform actions on behalf of the gasless account.

However, the signed payload does not include an operation type or execution path identifier. This allows the signature intended for vault deposits to be valid to perform direct transfers, and vice versa. Therefore, attackers can perform an attack by fetching a valid signature from pending transactions, front-running it by a different function invocation. violating the intended user operation.

```
145   function executePermitTransfer(PermitTransfer calldata permit, bytes calldata signature)
          external nonReentrant {
146       require(permit.deadline >= block.timestamp, "GasFreeController: Permit expired");
147       require(permit.token != address(0), "GasFreeController: Use TRC20 tokens only");
148       bytes32 structHash = _hashPermit(permit);
149       bytes32 digest = _hashTypedDataV4(structHash);
150       address signer = ECDSA.recover(digest, signature);
151
152       require(signer == permit.user && signer != address(0), "GasFreeController: Invalid
              signature");
153       require(nonces[permit.user] == permit.nonce, "GasFreeController: Invalid nonce");
154       require(IGasFreeAccount(permit.gasFreeAddress).owner() == permit.user, "GasFreeController:
              account not owned by user");
155
156       nonces[permit.user]++;
```

Listing 2.3: contracts/GasFreeController.sol

```
185   function executePermitDepositVault(PermitTransfer calldata permit, bytes calldata signature)
          external nonReentrant {
186       require(vault != address(0), "GasFreeController: vault not set");
187       require(permit.deadline >= block.timestamp, "GasFreeController: Permit expired");
188
189       bytes32 structHash = _hashPermit(permit);
190       bytes32 digest = _hashTypedDataV4(structHash);
191       address signer = ECDSA.recover(digest, signature);
192
193       require(signer == permit.user && signer != address(0), "GasFreeController: Invalid
              signature");
194       require(nonces[permit.user] == permit.nonce, "GasFreeController: Invalid nonce");
195       require(IGasFreeAccount(permit.gasFreeAddress).owner() == permit.user, "GasFreeController:
              account not owned by user");
```

Listing 2.4: contracts/GasFreeController.sol

```
30    struct PermitTransfer {
31        address token;
32        address serviceProvider;
33        address user;
34        address receiver;
35        address gasFreeAddress;
36        bool firstTime;
37        uint256 value;
38        uint256 maxFee;
39        uint256 deadline;
40        uint256 version;
41        uint256 nonce;
42    }
```

Listing 2.5: contracts/GasFreeController.sol

Impact The signature intended for an operation can be used in another function, violating the intended user behavior.

Suggestion Add a field in the signature to designate the operation type.

2.1.3 Lack of TRC20 transfer result handling in `withdrawERC20()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the contract `DEXVaultV1`, the function `withdrawERC20()` settles TRC20 withdrawals through the function `_transferERC20Out()`. However, the function `_transferERC20Out()` only verifies low-level call success and does not require the returned data to decode to `true` when present. As a result, the function `withdrawERC20()` may advance the withdrawal request state and the variable `dailyWithdrawals` even when a token signals a failed transfer by returning `false`.

Additionally, function `depositERC20()` does not verify the actual balance change before and after token transfer.

```
311         // Try to insert the request ID. Will revert if the request id was invalid
312         tryInsertRequestId(block.chainid, requestId, to, amount, token);
313
314         dailyWithdrawals[token] += amount;
315         // Success, send ERC20 token
316         _transferERC20Out(token, to, amount);
317         emit Withdraw(owner, msg.sender, to, token, amount, requestId);
318     }
```

Listing 2.6: contracts/DEXVaultV1.sol

Impact This flaw can cause the function `withdrawERC20()` to treat an unsuccessful token transfer as successful, diverging the withdrawal state and the variable `dailyWithdrawals` from the actual settlement outcome.

Suggestion Add balance-delta checks around the relevant token transfers.

Feedback from the project The project team acknowledged the issue and considered the impact to be under control.

2.1.4 Lack of whitelist check in the function `depositETH()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the contract `DEXVaultV1` , the function `depositETH()` allows anyone to deposit native tokens. However, unlike the restrictions in the function `depositERC20()`, this function does not enforce the token whitelist check (`tokenWhitelist(address(0))`). As a result, users can directly deposit ETH into the `DEXVaultV2` contract after initialization, which may not align with the intended design of only allowing specific tokens for deposits.

```
84  // check if token is allowed
85  modifier tokenWhitelist(address token) {
86      require(tokenWithdrawLimit[token] > 0, "Token not allowed");
87      _;
88  }
```

Listing 2.7: contracts/DEXVaultV1.sol

```
151 function depositETH(
152     address receiver
153 ) public payable whenNotPaused nonReentrant {
154     require(msg.value > 0, "Deposit amount must be greater than zero");
155     emit Deposit(msg.sender, receiver, address(0), msg.value);
156 }
```

Listing 2.8: contracts/DEXVaultV1.sol

Impact Users can always deposit ETH without any restrictions.

Suggestion Add a `tokenWhitelist(address(0))` check to the `depositETH()` function.

Feedback from the project The project team acknowledged the issue and considered the impact to be under control.

2.1.5 Improper owner attribution in withdrawal events

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `DEXVaultV1`, functions `withdrawETH()` and `withdrawERC20()` both accept an externally supplied `owner` parameter. However, the signature verification logic does not include this `owner` value in the signed message. The same unauthenticated value is then emitted through the event `Withdraw`. As a result, the fund movement remains protected by the signer checks, but the `owner` field in the event is not a trustworthy attribution field.

```
187 function withdrawETH(
188     address owner,
189     address payable to,
190     uint256 amount,
191     uint256 expireTime,
192     uint256 requestId,
193     address[] memory allSigners,
194     bytes[] memory signatures
195 )
196     public
197     whenNotPaused
198     nonReentrant
199     dailyLimitNotExceeded(address(0), amount)
200 {
201     require(allSigners.length >= 2, "invalid allSigners length");
202     require(
```

```
203             allSigners.length == signatures.length,
204             "invalid signatures length"
205         );
206         require(allSigners[0] != allSigners[1], "can not be same signer"); // must be different
                signer
207         require(expireTime >= block.timestamp, "expired transaction");
208
209         require(
210             amount <= tokenWithdrawLimit[address(0)],
211             "exceed withdraw eth limit"
212         );
213
214         bytes32 operationHash = keccak256(
215             abi.encodePacked(
216                 "ETHER",
217                 block.chainid,
218                 to,
219                 amount,
220                 expireTime,
221                 requestId,
222                 address(this)
223             )
224         );
225         operationHash = MessageHashUtils.toEthSignedMessageHash(operationHash);
226
227         for (uint8 index = 0; index < allSigners.length; index++) {
228             address signer = ECDSA.recover(operationHash, signatures[index]);
229             require(signer == allSigners[index], "invalid signer");
230             require(isAllowedSigner(signer), "not allowed signer");
231         }
232
233         // Try to insert the request ID. Will revert if the request id was invalid
234         tryInsertRequestId(block.chainid, requestId, to, amount, address(0));
235
236         // send ETHER
237         require(
238             address(this).balance >= amount,
239             "Address: insufficient balance"
240         );
241
242         dailyWithdrawals[address(0)] += amount;
243
244         (bool success, ) = to.call{value: amount}("");
245         require(
246             success,
247             "Address: unable to send value, recipient may have reverted"
248         );
249
250         emit Withdraw(owner, msg.sender, to, address(0), amount, requestId);
251     }
```

Listing 2.9: contracts/DEXVaultV1.sol

Impact A caller may emit withdrawal events with an arbitrary `owner` value.

Suggestion Remove the unauthenticated `owner` parameter from the event data.

2.1.6 Lack of constructor parameters in `getGasFreeAccountCreationCodeHash()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the contract `GasFreeFactory`, the function `getGasFreeAccountCreationCodeHash()` returns the creation-code hash of `GasFreeAccount` for CREATE2 address prediction. CREATE2 uses the keccak256 hash of `initcode` to calculate the address, and `initcode` is the combination of creation code and constructor parameters. However, the function hashes only `type(GasFreeAccount).creationCode` and omits the constructor parameters `user` and `controller`. As a result, off-chain computations based on this hash derive incorrect deterministic addresses.

```
81   /**
82    * @notice Returns the keccak256 hash of the GasFreeAccount's creation code.
83    * @dev This is used by off-chain tools to ensure consistent CREATE2 address prediction.
84    */
85   function getGasFreeAccountCreationCodeHash() public pure returns (bytes32) {
86       return keccak256(type(GasFreeAccount).creationCode);
87   }
```

Listing 2.10: contracts/GasFreeFactory.sol

Impact Off-chain tools attempting to pre-compute account addresses using this hash will generate incorrect results.

Suggestion Modify the function to accept `user` and `controller` as parameters and return `keccak256(abi.encodePacked(type(GasFreeAccount).creationCode, abi.encode(user, controller)))` to align with the actual deployed bytecode hash.

Feedback from the project The project team acknowledged the issue and considered the impact to be under control.

2.2 Recommendation

2.2.1 Return the prefixed hash from function `calcSigHash()`

Status Confirmed

Introduced by `Version 1`

Description In contract `DEXVaultV1`, function `calcSigHash()` returns the raw `keccak256` hash, while functions `withdrawETH()` and `withdrawERC20()` verify signatures against `toEthSignedMessageHash(operationHash)`. It is recommended to align the return value of the function `calcSigHash()` with the hash format used by the signature verification logic.

```
479   function calcSigHash(
480       address to,
481       uint256 amount,
482       address token,
483       uint256 expireTime,
484       uint256 requestId
485   ) public view returns (bytes32) {
486       bytes32 operationHash;
487
488       if (token == address(0)) {
489           operationHash = keccak256(
490               abi.encodePacked(
491                   "ETHER",
492                   block.chainid,
493                   to,
494                   amount,
495                   expireTime,
496                   requestId,
497                   address(this)
498               )
499           );
500       } else {
501           operationHash = keccak256(
502               abi.encodePacked(
503                   "ERC20",
504                   block.chainid,
505                   to,
506                   amount,
507                   token,
508                   expireTime,
509                   requestId,
510                   address(this)
511               )
512           );
513       }
514       return operationHash;
515   }
```

Listing 2.11: contracts/DEXVaultV1.sol

```
291       bytes32 operationHash = keccak256(
292           abi.encodePacked(
293               "ERC20",
294               block.chainid,
295               to,
296               amount,
297               token,
298               expireTime,
299               requestId,
300               address(this)
301           )
302       );
303       operationHash = MessageHashUtils.toEthSignedMessageHash(operationHash);
```

Listing 2.12: contracts/DEXVaultV1.sol

Suggestion Return `toEthSignedMessageHash(operationHash)` from the function `calcSigHash()`.

2.2.2 Use consistent TRON native asset naming and unit semantics

Status Confirmed

Introduced by `Version 1`

Description In the contract `DEXVaultV1`, the native asset and its unit are still named with `ETH`, `ETHER`, and `Wei` terminology, e.g., in the function `depositETH()`, the function `withdrawETH()`, and the function `withdrawETHByOwner()`. This is not a direct security vulnerability, but it may increase ambiguity in the TRON environment, especially when off-chain services must handle `TRX`, `sun` units, and TRON address formats. It is recommended to use consistent TRON-native naming and unit semantics across the codebase and related integrations.

```
446     * @param requestId the unique request id
447     * @param to      the destination address to send an outgoing transaction
448     * @param amount   the amount in Wei to be sent
449     * @param token   the address of the ERC20 contract
```

Listing 2.13: contracts/DEXVaultV1.sol

Suggestion Rename native asset references to `TRX` or `Native`, and use `sun` semantics consistently in events, comments, and scripts.

2.2.3 Add `_disableInitializers()` in the function `constructor()`

Status Confirmed

Introduced by `Version 1`

Description In contract `DEXVaultV1`, the function `_disableInitializers()` is not invoked in the constructor. Invoking this function prevents the contract itself from being initialized, thereby avoiding unexpected behavior.

```
113   function initialize(
114       address[] memory allowedSigners,
115       address usdt,
116       uint256 _withdrawUSDTLimit,
117       uint256 _withdrawETHLimit
118   ) public initializer {
119       __Ownable_init(msg.sender);
120       __UUPSUpgradeable_init();
121       __ReentrancyGuard_init();
122       __Pausable_init();
123
124       require(allowedSigners.length == 3, "invalid allSigners length");
125       require(
126           allowedSigners[0] != allowedSigners[1],
127           "must be different signers"
```

```
128         );
129         require(
130             allowedSigners[0] != allowedSigners[2],
131             "must be different signers"
132         );
133         require(
134             allowedSigners[1] != allowedSigners[2],
135             "must be different signers"
136         );
137         require(usdt != address(0), "invalid usdt address");
138
139         signers = allowedSigners;
140         tokenWithdrawLimit[usdt] = _withdrawUSDTLimit;
141         tokenWithdrawLimit[address(0)] = _withdrawETHLimit;
142     }
```

Listing 2.14: contracts/DEXVaultV1.sol

Suggestion Invoke the function `_disableInitializers()` in the constructor.

2.2.4 Remove redundant code

Status Confirmed

Introduced by `Version 1`

Description There are several unused variables `USDT_ADDRESS` and `ZERO_ADDRESS`. It is recommended to remove them for better code readability. Specifically, the following code should be removed or revised.

```
55  // Public fields
56  address public USDT_ADDRESS; // USDT contract address
57  address[] public signers; // The addresses that can co-sign transactions on the wallet
58
59  // Mapping of request IDs to requests
60  mapping(uint256 => request) requests;
61
62  IERC20 private constant ZERO_ADDRESS = IERC20(address(0));
```

Listing 2.15: contracts/DEXVaultV1.sol

Suggestion Remove the redundant code.

2.2.5 Adopt two-step `transferOwnership()` ownership handover

Status Confirmed

Introduced by `Version 1`

Description The contract `GasFreeController` and the contract `GasFreeFactory` each expose the function `transferOwnership()` as a single-step ownership reassignment that immediately updates the variable `owner` upon invocation. This design is operationally fragile because any misconfiguration is irreversible, resulting in permanent loss of administrative control over the contracts.

A two-step ownership transfer pattern mitigates this risk by requiring the current owner to nominate a new owner via the function `transferOwnership()`, followed by an explicit confirmation via the function `acceptOwnership()` from the nominated address.

```
111   function transferOwnership(address newOwner) external onlyOwner {
112       require(newOwner != address(0), "GasFreeController: New owner is the zero address");
113       owner = newOwner;
114       emit OwnerUpdated(newOwner);
115   }
```

Listing 2.16: contracts/GasFreeController.sol

```
44   function transferOwnership(address newOwner) external onlyOwner {
45       require(newOwner != address(0), "GasFreeFactory: New owner is the zero address");
46       owner = newOwner;
47       emit OwnerUpdated(newOwner);
48   }
```

Listing 2.17: contracts/GasFreeFactory.sol

Suggestion Implement a two-step ownership transfer mechanism in contracts `GasFreeController` and `GasFreeFactory`.

2.3 Note

2.3.1 TRC-10 asset compatibility

Introduced by `Version 1`

Description The contract `GasFreeAccount` does not provide a withdrawal path for `TRC-10` assets. If `TRC-10` assets are transferred to the contract by mistake, those assets may remain inaccessible because no dedicated recovery interface is available.

2.3.2 TRON address format conversion

Introduced by `Version 1`

Description Users should note that the function `getAddress()` returns a Solidity-level `address` value. However, in TRON interactions (e.g., TRON-native wallets and TronWeb), addresses are commonly represented in Base58Check format or as hex values with the `41` prefix. An incorrect address format is usually rejected. Therefore, users may need to convert the returned address before using it for transfer or query operations.

```
77   function getAddress(address user, uint256 salt) public view returns (address) {
78       return accounts[user][salt];
79   }
```

Listing 2.18: contracts/GasFreeFactory.sol

2.3.3 Predicted CREATE2 addresses depend on the variable `controller`

Introduced by Version 1

Description In the contract `GasFreeFactory`, any off-chain address derivation from the real CREATE2 inputs must use the current value of the variable `controller`. Once the variable `controller` changes, previously derived addresses should be treated as stale. The effective init code for future `GasFreeAccount` deployments changes with it.

```
38  // Added setController function
39  function setController(address _controller) external onlyOwner {
40      require(_controller != address(0), "GasFreeFactory: _controller cannot be zero address");
41      controller = _controller;
42  }
```

Listing 2.19: contracts/GasFreeFactory.sol

```
57  function createAccount(address user, uint256 salt) external returns (address accountAddress) {
58      require(controller != address(0), "GasFreeFactory: Controller not set");
59      require(!isAccountCreated[user][salt], "GasFreeFactory: Account already exists for this
            user and salt"); // New check
60
61      bytes32 create2Salt = keccak256(abi.encodePacked(user, salt));
62
63      GasFreeAccount newAccount = new GasFreeAccount{salt: create2Salt}(user, controller);
64      accountAddress = address(newAccount);
65
66      isAccountCreated[user][salt] = true; // Set mapping to true
67      accounts[user][salt] = accountAddress;
68      emit AccountCreated(user, accountAddress, salt);
69  }
```

Listing 2.20: contracts/GasFreeFactory.sol

2.3.4 Assumption of supported token

Introduced by Version 1

Description The contracts `DEXVaultV1` and `GasFreeAccount` accept arbitrary token addresses in their settlement logic and do not enforce a USDT-only restriction at the contract level. The project has confirmed that only USDT is intended to be supported in practice. The actual token support scope is therefore enforced at the operational layer rather than at the contract layer. Integrators and users should be aware that no on-chain safeguard prevents other tokens from being submitted.

```
167 function depositERC20(
168     IERC20 token,
169     uint256 amount,
170     address receiver
171 ) public tokenWhitelist(address(token)) whenNotPaused nonReentrant {
172     require(amount > 0, "Deposit amount must be greater than zero");
173     token.safeTransferFrom(msg.sender, address(this), amount);
174     emit Deposit(msg.sender, receiver, address(token), amount);
```

```
175   }
```

Listing 2.21: contracts/DEXVaultV1.sol

```
55    function withdraw(address token, address to, uint256 amount) external {
56        require(msg.sender == owner, "GasFreeAccount: Caller is not the owner");
57        if (token == address(0)) {
58            require(address(this).balance >= amount, "GasFreeAccount: Insufficient TRX balance");
59            (bool ok,) = payable(to).call{value: amount}("");
60            require(ok, "GasFreeAccount: TRX transfer failed");
61        } else {
62            uint256 balance = IERC20(token).balanceOf(address(this));
63            require(balance >= amount, "GasFreeAccount: Insufficient balance");
64            IERC20(token).transfer(to, amount);
65        }
66    }
```

Listing 2.22: contracts/GasFreeAccount.sol

2.3.5 TRON typed-data signing depends on address normalization

Introduced by Version 1

Description In this project, the typed-data signing flow depends on SDK-side address normalization. The signed fields in the contract `GasFreeController` are encoded as Solidity `address` values, while TRON user-facing addresses commonly appear in Base58 form or in hex form with the `41` prefix. In this signing flow, off-chain address inputs must be normalized into standard 20-byte `0x`-prefixed values before signing. If the signing side uses Base58 addresses or `41`-prefixed TRON hex addresses directly, the signature may no longer match the on-chain digest, and the flow may fail closed.

```
23contract GasFreeController is EIP712, ReentrancyGuard {
24    /// @dev EIP712 type hash for the PermitTransfer struct.
25    bytes32 private constant PERMIT_TRANSFER_TYPEHASH = keccak256(
26        "PermitTransfer(address token,address serviceProvider,address user,address receiver,address
              gasFreeAddress,bool firstTime,uint256 value,uint256 maxFee,uint256 deadline,uint256
             version,uint256 nonce)"
27    );
```

Listing 2.23: contracts/GasFreeController.sol

```
108 console.log("Message to sign:", JSON.stringify(message, null, 2));
109
110 const signature = tronWebUser.trx.signTypedData(domain, types, message, USER_PRIVATE_KEY);
111 console.log("Generated Signature:", signature);
```

Listing 2.24: script/executeGaslessDepositUups.cjs

2.3.6 Potential centralization risks

Introduced by Version 1

Description In this project, several privileged roles (e.g., the owners of contracts `DEXVaultV1`, `GasFreeController`, and `GasFreeFactory`) can conduct sensitive operations, which introduces potential centralization risks. For example, the `DEXVaultV1` owner can unilaterally replace all multi-sig signers, upgrade the vault implementation, or invoke emergency withdrawal functions to drain vault assets. The `GasFreeController` owner can redirect all gasless deposit flows to an arbitrary address or inflate transfer fees, and the `GasFreeFactory` owner can replace the bound controller, exposing all user `GasFreeAccount` funds to an attacker-controlled contract. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

2.3.7 Centralized delay mechanism

Introduced by `Version 1`

Description The gasless transfer design relies on a privileged `GasFreeController` contract as an intermediary execution layer between the service provider and each `GasFreeAccount`. Each `GasFreeAccount` is initialized with an immutable controller address, and the controller is permitted to approve tokens and move native funds on behalf of the account. Additionally, the `GasFreeFactory` owner can update the controller address used for newly created accounts through the function `setController()`. Users must trust the designated controller to execute only authorized actions, and must also trust the factory owner not to repoint future accounts to a malicious or compromised controller.

2.3.8 Permanent controller binding for accounts

Introduced by `Version 1`

Description The contract `GasFreeFactory` allows the owner to update the controller address for future account deployments through the function `setController()`. However, each `GasFreeAccount` stores the controller as an immutable variable at deployment time, meaning the binding cannot be changed after the account is created. As a result, accounts created at different points in time may remain permanently bound to different controller versions. Integrators and users should be aware of this persistent design constraint. Off-chain tools deriving CREATE2 addresses must also use the controller value that was active at the time of deployment, as a subsequent controller update will affect the init code of newly created accounts and render previously derived addresses stale.

2.3.9 Ensure the correct declaration of `permit.firstTime`

Introduced by `Version 1`

Description In the contract `GasFreeController`, both functions `executePermitTransfer()` and `executePermitDepositVault()` derive the activation fee directly from the caller-supplied field `permit.firstTime`. This field is included in the signed permit message and is treated as a user-declared fee switch rather than a chain-verifiable first-activation indicator. The contract does not validate `permit.firstTime` against any on-chain state to confirm whether the account is genuinely being activated for the first time. Integrators should be aware that the activation fee

is applied solely based on the value declared in the signed message, and off-chain scripts are responsible for setting this field correctly.

2.3.10 Ethereum-style signing prefix and naming on TRON

Introduced by Version 1

Description The contract DEXVaultV1 applies the Ethereum-standard prefix "\x19Ethereum Signed Message:\n32" when hashing withdrawal operation data, rather than the TRON-native TIP-191 prefix "\x19TRON Signed Message:\n32". Additionally, the signing fields use "ETHER" and "ERC20" identifiers, and function names such as withdrawETH() and depositETH() retain Ethereum-style naming. The project has evaluated this and confirmed that the implementation functions correctly on the TRON network under the current design. Readers should note that this is an intentional design choice and does not constitute a security risk in the project's assessed context.

2.3.11 Asymmetric gasless support between TRX and TRC20

Introduced by Version 1

Description The contract GasFreeController supports gasless transfers only for TRC20 tokens through the function executePermitTransfer(). Native TRX held in contract GasFreeAccount cannot be transferred via the gasless path. It can only be withdrawn directly by the account owner via the function withdraw(), or deposited into the vault through the function executePermitDepositVa
This asymmetry is an intentional design choice confirmed by the project. Users and integrators should be aware that the gasless relayer flow does not cover native TRX peer-to-peer transfers.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS