BLOCKSEC

Security Audit

Report for ETH Rust SDK

Date: April 1, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Bitget Wallet
Target	ETH Rust SDK

Version History

Version	Date	Description
1.0	April 1, 2026	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-04-14 10:09:29+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	Rust
Approach	Semi-automatic and manual verification

The audit target (hereinafter referred to as the Target) of this audit is provided as a **ZIP** archive.

The ETH Rust SDK is a Rust-based Ethereum wallet library exposed to Go applications through a unified FFI interface. It provides comprehensive wallet capabilities including address derivation, public key generation, message signing, and transaction signing. Developers interact with the SDK via a single entry point, tee_wallet_call, supplying function names and JSON-formatted parameters. The SDK supports multiple Ethereum standards such as EIP-55 addresses, EIP-712 typed data signing, EIP-1559 and EIP-2930 transactions, and advanced EIP-7702 authorization-based transactions. Designed for flexibility and security, it allows seamless integration of Ethereum wallet functionality into Go-based systems.

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

- ethereum.rs
- lib.rs
- mod.rs
- types.rs
- utils.rs

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 MD5 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	MD5 SHA
ETH Rust SDK	`Version 1`	`08f935c3c6c7722cdd4c98aa2e1da319`
	`Version 2`	`40cd22a65644ba34140415f6832e0d3c`

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 [1] and Common Weakness Enumeration [2]. 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:

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

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

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **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 **twenty eight** potential security issues. Besides, we have **five** recommendations and **three** notes.

- High Risk: 11
- Medium Risk: 7
- Low Risk: 10
- Recommendation: 5
- Note: 3

ID	Severity	Description	Category	Status
1	High	DoS due to panic in the function `hash_eth_message()`	Security Issue	Fixed
2	High	Incorrect value parsing for hexadecimal strings	Security Issue	Fixed
3	High	Incorrect unit scaling	Security Issue	Fixed
4	High	DoS due to integer overflow via unsafe type casting	Security Issue	Fixed
5	High	Potential DoS when signing `TypedDataV1` or `TypedDataV3` message	Security Issue	Fixed
6	High	Incorrect match logic when handling array types	Security Issue	Fixed
7	High	Incorrect removal logic in function `sign_transaction()`	Security Issue	Fixed
8	High	Incorrect condition checks in the function `convert_to_tx_param()`	Security Issue	Fixed
9	High	Signed integer types can not handle negative values	Security Issue	Fixed
10	High	Lack of support for numeric values in JSON	Security Issue	Fixed
11	High	Lack of support for fixed-size bytes types	Security Issue	Fixed
12	Medium	Default handling of null and invalid inputs for `chain_id` and `nonce` breaks authorization signature semantics	Security Issue	Fixed
13	Medium	Incorrect implementation in function `hash_typed_data_v4()`	Security Issue	Fixed
14	Medium	Precision loss or panic due to intermediate `u128` type casting	Security Issue	Fixed
15	Medium	Unchecked unwraps in function `sign_transaction()` may panic	Security Issue	Fixed
16	Medium	Incorrect version of ethers-signers	Security Issue	Fixed
17	Medium	Incorrect handling of the field `chain_id`	Security Issue	Fixed

18	Medium	Potential signature collision in the function `process_bit_sign_message()`	Security Issue	Confirmed
19	Low	Empty access list prevents gas optimization	Security Issue	Confirmed
20	Low	Non-string JSON fields are silently dropped in the function `process_bit_sign_message()`	Security Issue	Confirmed
21	Low	Improper handling logic of `__type` field in function `parse_message_format()`	Security Issue	Confirmed
22	Low	Improper handling logic for `message` in function `sign_personal_message()`	Security Issue	Confirmed
23	Low	Potential encryption key generation failure due to inconsistent hexadecimal prefix handling	Security Issue	Fixed
24	Low	Lack of support for BIP39 passphrase	Security Issue	Confirmed
25	Low	Hardcoded default derivation path can derive incorrect private key	Security Issue	Confirmed
26	Low	Inappropriate use of signed integer parsing for transaction values	Security Issue	Fixed
27	Low	Default handling logic in the transaction parsing process alters signature semantics	Security Issue	Fixed
28	Low	Inconsistent naming conventions for JSON parsing fields	Security Issue	Fixed
29	-	Remove redundant code	Recommendation	Fixed
30	-	Unify mnemonic validation logic	Recommendation	Fixed
31	-	Ensure parameter naming consistency in logic	Recommendation	Fixed
32	-	Replace unsafe type casting in the the function `ecdsa_sign()` to prevent potential overflow	Recommendation	Fixed
33	-	Unify `chain_id` parse logic	Recommendation	Fixed
34	-	Deprecated EIP-1024 due to security concern	Note	-
35	-	Blob transaction not supported	Note	-
36	-	Input parameters from external callers must be verified and secure	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 DoS due to panic in the function `hash_eth_message()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `hash_eth_message()` handles the processing of messages for the type `EthSign`. It enforces that the message length must be exactly 32 bytes. However, it uses `panic!` to enforce this constraint instead of returning a proper error.

If a user or an attacker submits a message with the `EthSign` prefix (e.g., `EthSign:0x1234`) that decodes to a byte array with a length other than 32, the panic is triggered. Since this SDK is used as a core component for backend services, an unhandled panic will crash the entire application process, leading to a DoS.

```
1501     if message.len() != 32 {
1502         panic!("ETH_SIGN message must be 32 bytes");
1503     }
```

Listing 2.1: ethereum.rs

Impact An attacker can intentionally send a malformed `EthSign` request to crash the service relying on this SDK.

Suggestion Replace the macro `panic!` with proper error handling logic.

2.1.2 Incorrect value parsing for hexadecimal strings

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description According to the `DEVELOPER_GUIDE.md`, some fields (e.g., `value`, `nonce`, `gas_limit`, `gas_price`, `max_fee_per_gas` and `max_priority_fee_per_gas`) should support both hex and decimal format.

However, the function `sign_transaction()` does not handle hexadecimal strings properly. The current implementation attempts to parse fields `value` and `gas_price` via `parse::<f64>()`. Since function `parse::<f64>()` does not support the prefix "0x", it returns an error and uses `.unwrap_or(0.0)` to mask this error and set the value as 0. This leads to valid input with hex values assigned with zero values.

Meanwhile, `U256::from_str()` is intended to parse decimal strings into U256, not hexadecimal strings, which is incorrectly used.

```
317         for key in ["value", "gas_price"].iter() {
318             if let Some(val) = transaction_data.get(*key) {
319                 let converted_value = if let Some(s) = val.as_str() {
320                     let parsed_value = s.parse::<f64>().unwrap_or(0.0);
321                     (parsed_value * 1_000_000_000_000_000_000.0) as u128
```

Listing 2.2: ethereum.rs

```
1037    let nonce = U256::from_str(&tx_params.nonce.trim_start_matches("0x")).unwrap_or_default();
1038    let gas_price =
1039        U256::from_str(&tx_params.gas_price.trim_start_matches("0x")).unwrap_or_default();
1040    let gas_limit = if tx_params.gas_limit.starts_with("0x") {
1041        U256::from_str(&tx_params.gas_limit.trim_start_matches("0x")).unwrap_or_default()
1042    } else {
1043        if let Ok(decimal_val) = tx_params.gas_limit.parse::<u64>() {
1044            U256::from(decimal_val)
1045        } else {
1046            U256::from_str(&tx_params.gas_limit).unwrap_or_default()
1047        }
1048    };
1049    let value = U256::from_str(&tx_params.value.trim_start_matches("0x")).unwrap_or_default();
```

Listing 2.3: ethereum.rs

```
1101    let nonce = U256::from_str(&tx_params.nonce.trim_start_matches("0x")).unwrap_or_default();
```

Listing 2.4: ethereum.rs

```
1105    let mut max_priority_fee_per_gas =
1106        U256::from_str(&tx_params.max_priority_fee_per_gas.trim_start_matches("0x"))
1107            .unwrap_or_default();
1108    let mut max_fee_per_gas =
1109        U256::from_str(&tx_params.max_fee_per_gas.trim_start_matches("0x")).unwrap_or_default();
```

Listing 2.5: ethereum.rs

```
1117        let gas_price = if tx_params.gas_price.starts_with("0x") {
1118            U256::from_str(&tx_params.gas_price.trim_start_matches("0x")).unwrap_or_default()
```

Listing 2.6: ethereum.rs

```
1135    let gas_limit = if tx_params.gas_limit.starts_with("0x") {
1136        U256::from_str(&tx_params.gas_limit.trim_start_matches("0x")).unwrap_or_default()
1137    } else {
1138        if let Ok(decimal_val) = tx_params.gas_limit.parse::<u64>() {
1139            U256::from(decimal_val)
1140        } else {
1141            U256::from_str(&tx_params.gas_limit.trim_start_matches("0x")).unwrap_or_default()
1142        }
1143    };
1144    let value = U256::from_str(&tx_params.value.trim_start_matches("0x")).unwrap_or_default();
```

Listing 2.7: ethereum.rs

```
1742        let nonce_u256 = U256::from_str(nonce.trim_start_matches("0x")).unwrap_or_default();
```

Listing 2.8: ethereum.rs

```
1757        let max_priority_fee_per_gas_u256 =
1758            U256::from_str(max_priority_fee_per_gas.trim_start_matches("0x"))
1759                .unwrap_or_default();
```

Listing 2.9: ethereum.rs

```
1775        let max_fee_per_gas_u256 =
1776            U256::from_str(max_fee_per_gas.trim_start_matches("0x")).unwrap_or_default();
```

Listing 2.10: ethereum.rs

```
1793        } else if gas_limit.starts_with("0x") {
1794            U256::from_str(gas_limit.trim_start_matches("0x")).unwrap_or_default()
```

Listing 2.11: ethereum.rs

```
1799                U256::from_str(gas_limit.trim_start_matches("0x")).unwrap_or_default()
```

Listing 2.12: ethereum.rs

```
1826        let value_u256 = U256::from_str(value.trim_start_matches("0x")).unwrap_or_default();
```

Listing 2.13: ethereum.rs

```
1971        let nonce_u256 = U256::from_str(nonce.trim_start_matches("0x")).unwrap_or_default();
```

Listing 2.14: ethereum.rs

```
1978    // 3. maxPriorityFeePerGas
1979    let max_priority_fee_per_gas_u256 =
1980        U256::from_str(max_priority_fee_per_gas.trim_start_matches("0x")).unwrap_or_default();
1981    stream.append(&self.to_unpadded_buffer(&max_priority_fee_per_gas_u256));
1982    // 4. maxFeePerGas
1983    let max_fee_per_gas_u256 =
1984        U256::from_str(max_fee_per_gas.trim_start_matches("0x")).unwrap_or_default();
1985    stream.append(&self.to_unpadded_buffer(&max_fee_per_gas_u256));
1986    // 5. gasLimit
1987    let gas_limit_u256 = if gas_limit.starts_with("0x") {
1988        U256::from_str(gas_limit.trim_start_matches("0x")).unwrap_or_default()
1989    } else {
1990        if let Ok(decimal_val) = gas_limit.parse::<u64>() {
1991            U256::from(decimal_val)
1992        } else {
1993            U256::from_str(gas_limit.trim_start_matches("0x")).unwrap_or_default()
1994        }
1995    };
```

Listing 2.15: ethereum.rs

```
2008        let value_u256 = U256::from_str(value.trim_start_matches("0x")).unwrap_or_default();
```

Listing 2.16: ethereum.rs

Impact Hexadecimal strings cannot be parsed correctly.

Suggestion It is suggested to follow the Ethereum JSON-RPC standard to ensure the correct parsing of hexadecimal strings.

2.1.3 Incorrect unit scaling

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `DEVELOPER_GUIDE.md` states that the `value`, `gas_price` and all the gas relative fields unit should be Wei.

However, the current implementation multiplies the value by `1e18`, which assumes these numeric inputs are in Ether rather than Wei. If a DApp or user passes a value in Wei, the amount is multiplied by `1e18` again. This leads to astronomically high values, causing transaction failure or unintended massive transfers of funds.

Moreover, numbers without quotes are parsed into the `serde_json::Number` type, which internally often relies on `f64` (IEEE 754 double precision) for non-integer formats or large integers. However, the safe integer limit for IEEE 754 is $2^{53} - 1$. Ethereum values (Wei) regularly exceed this (i.e., about 9.007 Ether). Any numeric value exceeding this limit will lose its least bits during the conversion process.

```
2//    introduction
3"introduction": {
```

Listing 2.17: .happyAuditor/info.jsonc

```
317         for key in ["value", "gas_price"].iter() {
318             if let Some(val) = transaction_data.get(*key) {
319                 let converted_value = if let Some(s) = val.as_str() {
320                     let parsed_value = s.parse::<f64>().unwrap_or(0.0);
321                     (parsed_value * 1_000_000_000_000_000_000.0) as u128
322                 } else if let Some(n) = val.as_number() {
323                     if let Some(f) = n.as_f64() {
324                         (f * 1_000_000_000_000_000_000.0) as u128
325                     } else if let Some(u) = n.as_u64() {
326                         (u as f64 * 1_000_000_000_000_000_000.0) as u128
327                     } else if let Some(i) = n.as_i64() {
328                         (i as f64 * 1_000_000_000_000_000_000.0) as u128
329                     } else {
330                         0
331                     }
332                 } else {
333                     0
334                 };
```

Listing 2.18: ethereum.rs

Impact This leads to astronomically high values, causing transaction failure or unintended massive transfers of funds.

Suggestion Revise the logic accordingly.

Clarification from BlockSec The project team has acknowledged this issue and implemented a fix in `Version 2` by treating any value containing a '.' as an Ether-denominated value. The

project team is aware of the associated risks and has confirmed that upstream components will be informed to carefully validate parameters passed to the SDK. Additionally, when Ethereum values (in Wei) are represented using `serde_json::Number`, any value exceeding $2^{53}-1$ may lose precision during the conversion process. The project team has taken note of this limitation.

2.1.4 DoS due to integer overflow via unsafe type casting

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `sign_eip1559_transaction()`, transaction parameters (i.e., `value`, `nonce`, `max_fee_per_gas`, and `gas_limit`) are stored as type `U256` types but are converted using `.as_u64()` before being cast to `u128` for the structure `FeeMarketTransaction`.

However, the `as_u64()` function will get panic when overflow occurs. If a user attempts to sign a transaction where any of these fields exceed `u64::MAX` (i.e., 18.44 Ether in Wei), the program will panic, causing a DoS for the wallet service.

```
995     let tx = FeeMarketTransaction {
996         chain: chain_value,
997         nonce: nonce.as_u64() as u128,
998         max_priority_fee_per_gas: max_priority_fee_per_gas.as_u64() as u128,
999         max_fee_per_gas: max_fee_per_gas.as_u64() as u128,
1000        gas: gas_limit.as_u64() as u128,
1001        to,
1002        value: value.as_u64() as u128,
1003        data,
1004        access_list: ethereum_tx_sign::AccessList(vec![]), // Empty access list
1005    };
```

Listing 2.19: ethereum.rs

Impact If a user attempts to sign a transaction where any of these fields exceed `u64::MAX` (i.e., 18.44 Ether in Wei), the program will panic, causing a DoS for the service.

Suggestion Revise the logic accordingly.

2.1.5 Potential DoS when signing `TypedDataV1` or `TypedDataV3` message

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `parse_message_format()`, the message type parsing logic is designed to differentiate between different versions of Ethereum typed data signing standards based on the `__v4` boolean flag in the input JSON (i.e., the parameter `message`).

However, when this flag is set to `false`, current implementation unconditionally returns `TypedDataV1`, which is designed for array-based legacy format. This creates a conflict with the design intent, as `TypedDataV3` uses the same EIP-712 structured object format as `TypedDataV4`.

```
1574                return match msg_type {
1575                    "personal" => (MessageType::PersonalSign, data),
1576                    "eth_sign" | "ethSign" => {
1577                        let processed_data = self.preprocess_eth_sign_message(&data);
1578                        (MessageType::EthSign, processed_data)
1579                    }
1580                    "typed_data" | "typedData" => {
1581                        if is_v4 {
1582                            (MessageType::TypedDataV4, data)
1583                        } else {
1584                            (
1585                                MessageType::TypedDataV1,
1586                                self.process_legacy_typed_data(&data),
1587                            )
1588                        }
1589                    }
1590                    _ => (MessageType::PersonalSign, data),
1591                };
```

Listing 2.20: ethereum.rs

Furthermore, in the file `ethereum.rs` line 1597 - 1608, the function `parse_message_format()` returns `TypedDataV4` if the `message` is not a JSON type input and starts with string "TypedData:". This also creates a conflict when the `message` is `TypedDataV1`.

```
1597    if message.starts_with("Personal:") {
1598        (MessageType::PersonalSign, message[9..].to_string())
1599    } else if message.starts_with("EthSign:") {
1600        let message_content = message[8..].to_string();
1601        let processed_data = self.preprocess_eth_sign_message(&message_content);
1602        (MessageType::EthSign, processed_data)
1603    } else if message.starts_with("TypedData:") {
1604        (MessageType::TypedDataV4, message[10..].to_string())
1605    } else {
1606        //
1607        (MessageType::PersonalSign, message.to_string())
1608    }
```

Listing 2.21: ethereum.rs

Impact This creates a potential conflict between `TypedDataV1` and `TypedDataV3`.

Suggestion Revise the message parsing logic.

Clarification from BlockSec The project team has acknowledged that the function `parse_message_format()` has limitations in handling `TypedDataV1` messages in plain text format. The project team is aware of this behavior and has confirmed that upstream components will be informed to wrap all `TypedDataV1` messages in JSON format to ensure proper functionality.

2.1.6 Incorrect match logic when handling array types

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `encode_field_raw()`, the match logic incorrectly treats array types (e.g., uint[], int[]) as scalar types when the field type matches via `starts_with("uint")` or `starts_with("int")`.

This causes array values to be treated as single strings, defaulting `num_str` to "0" if no valid string is present, and subsequently encoding a zero value instead of properly encoding the array elements.

```
374     _ if field_type.starts_with("uint") => {
375         let num_str = value.and_then(|v| v.as_str()).unwrap_or("0");
376         let num = parse_uint256(num_str)?;
377         let mut bytes = vec![0u8; 32];
378         num.to_big_endian(&mut bytes);
379         Ok((field_type.to_string(), bytes))
380     }
381     _ if field_type.starts_with("int") => {
382         let num_str = value.and_then(|v| v.as_str()).unwrap_or("0");
383         let num = parse_uint256(num_str)?;
384         let mut bytes = vec![0u8; 32];
385         num.to_big_endian(&mut bytes);
386         Ok((field_type.to_string(), bytes))
387     }
388     _ if field_type.ends_with("[]") => {
```

Listing 2.22: utils.rs

Impact Integer arrays (e.g., uint[], int[]) are incorrectly encoded as the integer zero.

Suggestion Reorder the match branches to prioritize specific types, ensuring array types are handled correctly.

2.1.7 Incorrect removal logic in function `sign_transaction()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `sign_transaction()`, transaction data is parsed and processed. When the function determines that the transaction is not the EIP-1559 type, it directly uses `remove("type")` to remove the `type` field from the transaction data, and then invokes the function `sign_transaction_internal()`.

In the internal invocation to the function `convert_to_tx_param()`, the transaction type is automatically detected, and the missing `type` field is forcibly mapped to 0. This causes the `type` field provided by the user to not be processed correctly.

If the user passes a `type` value of 1, since the transaction is not EIP-1559 type, the `type` will be removed in the function `sign_transaction()` and then be forcibly mapped to 0. Consequently, the function `sign_eip2930_transaction()` branch will not be triggered. This may result in missing function logic and semantic errors in the transaction.

```
471     if is_eip1559 {
472         transaction_data
473             .as_object_mut()
474             .unwrap()
475             .remove("gas_price");
476         transaction_data["type"] = json!(2);
477     } else {
478         transaction_data
479             .as_object_mut()
480             .unwrap()
481             .remove("max_fee_per_gas");
482         transaction_data
483             .as_object_mut()
484             .unwrap()
485             .remove("max_priority_fee_per_gas");
486         transaction_data.as_object_mut().unwrap().remove("type");
487     }
```

Listing 2.23: ethereum.rs

```
1316     let mut tx_type_value = obj.get("type").and_then(|v| v.as_u64()).unwrap_or(0);
```

Listing 2.24: ethereum.rs

Impact The branches of the function `sign_eip2930_transaction()` will not be triggered, which may lead to missing logic and semantic errors in transaction processing.

Suggestion Revise the logic accordingly.

2.1.8 Incorrect condition checks in the function `convert_to_tx_param()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `convert_to_tx_param()` processes EIP-1559 transactions separately by checking relevant fields (i.e., `maxFeePerGas`, `max_fee_per_gas`, `maxPriorityFeePerGas` and `max_priority_fee_per_gas`). Specifically, when EIP-1559 related fields are detected, it unconditionally overwrites the transaction type to 2. However, the same fields are also used in EIP-7702 transactions. If the user intends to submit an EIP-7702 transaction, it will be directly overwritten as an EIP-1559 type, which is incorrect.

```
1316     let mut tx_type_value = obj.get("type").and_then(|v| v.as_u64()).unwrap_or(0);
1317     if obj.get("maxFeePerGas").is_some()
1318         || obj.get("max_priority_fee_per_gas").is_some()
1319         || obj.get("maxPriorityFeePerGas").is_some()
1320         || obj.get("max_fee_per_gas").is_some()
1321     {
1322         tx_type_value = 2; // EIP-1559
1323     }
```

Listing 2.25: ethereum.rs

Impact The type of the EIP-7702 transaction can not be identified correctly.

Suggestion Revise the logic accordingly.

2.1.9 Signed integer types can not handle negative values

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In file `utils.rs`, the function `encode_field_raw()` uses function `parse_uint256()` to parse signed integer (e.g., int, int256) values. However, the function `parse_uint256()` uses `U256` (unsigned 256-bit integer), which cannot represent negative numbers. When a negative value (e.g., -1) is provided, the function returns an error instead of correctly encoding it.

```
381     _ if field_type.starts_with("int") => {
382         let num_str = value.and_then(|v| v.as_str()).unwrap_or("0");
383         let num = parse_uint256(num_str)?;
384         let mut bytes = vec![0u8; 32];
385         num.to_big_endian(&mut bytes);
386         Ok((field_type.to_string(), bytes))
387     }
```

Listing 2.26: utils.rs

```
512 pub fn parse_uint256(value: &str) -> Result<U256, Box<dyn std::error::Error>> {
513     if value.is_empty() {
514         return Ok(U256::zero());
515     }
516
517     if value.starts_with("0x") || value.starts_with("0X") {
518         U256::from_str_radix(&value[2..], 16)
519             .map_err(|e| format!("Failed to parse hex uint256 '{}': {}", value, e).into())
520     } else {
521         U256::from_dec_str(value)
522             .map_err(|e| format!("Failed to parse decimal uint256 '{}': {}", value, e).into())
523     }
524 }
```

Listing 2.27: utils.rs

Impact Any EIP-712 TypedData message containing negative integer values will fail to be encoded correctly.

Suggestion Implement proper signed integer parsing logic.

2.1.10 Lack of support for numeric values in JSON

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `encode_field_raw()`, numeric values are extracted using `value.and_then(|v| v.as_str()).unwrap_or("0")`. The function `.as_str()` only returns `Some` for `Value::String`, not for `Value::Number`. When the JSON contains a numeric value without quotes (e.g., "amount": 1000000000000000000), the function `.as_str()` returns `None` and the value will be set to "0" accordingly.

```
374     _ if field_type.starts_with("uint") => {
375         let num_str = value.and_then(|v| v.as_str()).unwrap_or("0");
376         let num = parse_uint256(num_str)?;
377         let mut bytes = vec![0u8; 32];
378         num.to_big_endian(&mut bytes);
379         Ok((field_type.to_string(), bytes))
380     }
381     _ if field_type.starts_with("int") => {
382         let num_str = value.and_then(|v| v.as_str()).unwrap_or("0");
383         let num = parse_uint256(num_str)?;
384         let mut bytes = vec![0u8; 32];
385         num.to_big_endian(&mut bytes);
386         Ok((field_type.to_string(), bytes))
387     }
```

Listing 2.28: utils.rs

Impact Numeric values in JSON may not be parsed correctly.

Suggestion Handle both string and numeric values in JSON properly.

2.1.11 Lack of support for fixed-size bytes types

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `encode_field_raw()`, only `bytes32` is explicitly handled. Other fixed-size byte types (`bytes1` through `bytes31`) fall into the default branch (line 406), where they are recognized as primitive types by `is_primitive_type()`, causing the function to return an "Unsupported primitive type" error.

```
361     "bytes" => {
362         let bytes_str = value.and_then(|v| v.as_str()).unwrap_or("0x");
363         let bytes_val = hex::decode(bytes_str.trim_start_matches("0x"))?;
364         let hash = keccak256(&bytes_val);
365         Ok(("bytes32".to_string(), hash.to_vec()))
366     }
367     "bytes32" => {
368         let bytes_str = value
369             .and_then(|v| v.as_str())
370             .unwrap_or("0x0000000000000000000000000000000000000000000000000000000000000000");
371         let bytes_val = hex::decode(bytes_str.trim_start_matches("0x"))?;
372         Ok(("bytes32".to_string(), bytes_val))
373     }
```

Listing 2.29: utils.rs

```
406     _ => {
407         if is_primitive_type(field_type) {
408             return Err(format!(
409                 "Unsupported primitive type in encode_field_raw: {}",
410                 field_type
411             )
412             .into());
413         }
```

Listing 2.30: utils.rs

Impact Any EIP-712 TypedData message using `bytes1` to `bytes31` types will fail with an error.

Suggestion Add proper logic to support the bytesN types.

2.1.12 Default handling of null and invalid inputs for `chain_id` and `nonce` breaks authorization signature semantics

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `sign_eip7702_authorization()`, the fields `chain_id` and `nonce` are parsed using function `parse_uint256()` before participating in the construction of the signed message. If the parsing fails or the input is empty, function `unwrap_or_default()` is used to set the value to 0. Subsequently, these zero values are encoded as empty byte arrays.

This design causes different inputs (e.g., empty strings, 0x, 0, or failed parsing results) to be treated as equivalent at the signature level. These inputs produce identical RLP encoding and message hashes and are included in the ECDSA signature. Since this logic occurs in the authorization message signing path, it directly alters the semantics of the signed message, making the signature unable to accurately reflect the user's original input. Similar logical problems also exist in other parts of the `ethereum.rs`.

```
2110     // Encode chainId
2111     let chain_id_bytes = if chain_id.is_empty() || chain_id == "0x" {
2112         vec![]
2113     } else {
2114         let chain_id_u256 = utils::parse_uint256(chain_id).unwrap_or_default();
2115         self.to_unpadded_buffer(&chain_id_u256)
2116     };
```

Listing 2.31: ethereum.rs

```
2124     // Encode nonce
2125     let nonce_u256 = utils::parse_uint256(nonce).unwrap_or_default();
2126     let nonce_bytes = if nonce_u256.is_zero() {
2127         vec![]
2128     } else {
2129         self.to_unpadded_buffer(&nonce_u256)
2130     };
```

Listing 2.32: ethereum.rs

Impact It directly alters the semantics of the signed message, making the signature unable to accurately reflect the user's original input.

Suggestion Revise the logic accordingly.

2.1.13 Incorrect implementation in function `hash_typed_data_v4()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `hash_typed_data_v4()` does not support the recursive processing of fixed-size arrays (e.g., `bytes[2]`) and multi-dimensional arrays (e.g., `bool[2][1]`).

Additionally, custom type arrays are not handled properly. If a custom type `Struct` is defined, and a field declared as `Struct[]`, then in the parse process, it tries to look up `Struct[]` as a type name instead of recognizing it as an array of `Struct`.

```
388     _ if field_type.ends_with("[]") => {
389         let base_type = &field_type[..field_type.len() - 2];
390         let empty_vec = Vec::new();
391         let array_value = value.and_then(|v| v.as_array()).unwrap_or(&empty_vec);
392 
393         let mut element_types = Vec::new();
394         let mut element_values = Vec::new();
395 
396         for item in array_value {
397             let (element_type, element_value) = encode_field_raw(base_type, Some(item), types)?;
398             element_types.push(element_type);
399             element_values.push(element_value);
400         }
401 
402         let encoded_elements = raw_encode(&element_types, &element_values)?;
403         let hash = keccak256(&encoded_elements);
404         Ok(("bytes32".to_string(), hash.to_vec()))
405     }
```

Listing 2.33: utils.rs

Impact This leads to an incorrect hash calculation that differs from the standard EIP-712 specification for valid typed data.

Suggestion Revise the logic accordingly.

2.1.14 Precision loss or panic due to intermediate `u128` type casting

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description While Ethereum natively supports 256-bit unsigned integers, the current implementation casts some fields into type `u128`.

Specifically, function `.as_u128()` or statement `as u128` are used in the conversion process, which can result in precision loss or panic.

```
902     let tx = LegacyTransaction {
903         nonce: nonce.as_u128(),
904         gas_price: gas_price.as_u128(),
905         gas: gas_limit.as_u128(),
906         to,
907         value: value.as_u128(),
908         data,
909         chain: chain_id,
910     };
```

Listing 2.34: ethereum.rs

```
1067    let tx = AccessListTransaction {
1068        chain: chain_id,
1069        nonce: nonce.as_u128(),
1070        gas_price: gas_price.as_u128(),
1071        gas: gas_limit.as_u128(),
1072        to,
1073        value: value.as_u128(),
1074        data,
1075        access_list: ethereum_tx_sign::AccessList(vec![]), // Empty access list
1076    };
```

Listing 2.35: ethereum.rs

```
319             let converted_value = if let Some(s) = val.as_str() {
320                 let parsed_value = s.parse::<f64>().unwrap_or(0.0);
321                 (parsed_value * 1_000_000_000_000_000_000.0) as u128
322             } else if let Some(n) = val.as_number() {
323                 if let Some(f) = n.as_f64() {
324                     (f * 1_000_000_000_000_000_000.0) as u128
325                 } else if let Some(u) = n.as_u64() {
326                     (u as f64 * 1_000_000_000_000_000_000.0) as u128
327                 } else if let Some(i) = n.as_i64() {
328                     (i as f64 * 1_000_000_000_000_000_000.0) as u128
```

Listing 2.36: ethereum.rs

```
376                     format!("0x{:x}", f as u128)
```

Listing 2.37: ethereum.rs

```
395         let converted_value = if let Some(s) = gas_price.as_str() {
396             let parsed_value = s.parse::<f64>().unwrap_or(0.0);
397             (parsed_value * 1_000_000_000_000_000_000.0) as u128
398         } else if let Some(n) = gas_price.as_number() {
399             if let Some(f) = n.as_f64() {
400                 (f * 1_000_000_000_000_000_000.0) as u128
```

```
401             } else if let Some(u) = n.as_u64() {
402                 (u as f64 * 1_000_000_000_000_000_000.0) as u128
403             } else if let Some(i) = n.as_i64() {
404                 (i as f64 * 1_000_000_000_000_000_000.0) as u128
```

Listing 2.38: ethereum.rs

```
1121            let wei_val = (float_val * 1e18) as u128;
```

Listing 2.39: ethereum.rs

Impact This conversion causes data loss or panic for any value exceeding `u128::MAX`.

Suggestion Revise the logic accordingly.

2.1.15 Unchecked unwraps in function `sign_transaction()` may panic

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `sign_transaction()` parses `data` into `transaction_data` for transaction signing. However, it invokes function `unwrap()` on functions `pk.as_str()`, `mn.as_str()`, and `transaction_data.as_object_mut()` without validating their types. As a result, malformed or unexpected JSON causes a runtime panic, leading to denial of service during signing.

```
272         pk.as_str().unwrap().to_string()
```

Listing 2.40: ethereum.rs

```
276             mnemonic: mn.as_str().unwrap().to_string(),
```

Listing 2.41: ethereum.rs

```
467         transaction_data.as_object_mut().unwrap().remove("gas");
```

Listing 2.42: ethereum.rs

Impact Malformed or unexpected JSON causes a runtime panic, leading to denial of service during signing.

Suggestion Revise the logic accordingly.

2.1.16 Incorrect version of crate ethers-signers

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description File `Cargo.toml` specifies the version of crate ethers-signers as 2.0. However, the implementation of `LocalWallet::from_str()` varies across its different sub-versions.

Specifically, in version 2.0.0, `LocalWallet::from_str()` allows for 0x and 0X prefixes. However, in newer versions (e.g., 2.0.14), `LocalWallet::from_str()` only accepts 0X as a prefix.

In the current implementation, the parameter `private_key` has a 0x prefix added before `LocalWallet::from_str()` is invoked, which is incompatible with newer versions of crate ethers-signers.

```
831     let private_key = private_key.trim_start_matches("0x");
832     let wallet = LocalWallet::from_str(&format!("0x{}", private_key))
833         .map_err(|e| WalletError::InvalidPrivateKey(e.to_string()))?;
```

Listing 2.43: ethereum.rs

Impact Function `sign_transaction_internal_impl()` may not work as expected.

Suggestion Revise the logic accordingly.

2.1.17 Incorrect handling of the field `chain_id`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The parameter `chain_id` in the function `sign_transaction()` has inconsistent type handling compared to other transaction parameters. While fields (e.g., `nonce`) support both numeric and string formats, the parameter `chain_id` only accepts numeric types. When provided as a string (e.g., "0x1", "1337"), the parsing fails and the code silently falls back to the default value "1" (i.e., Ethereum Mainnet). This creates a vulnerability where users who intend to sign transactions on testnets or alternative chains may unknowingly sign transactions intended for mainnet.

```
1327         chain_id: obj
1328             .get("chainId")
1329             .or_else(|| obj.get("chain_id"))
1330             .and_then(|v| v.as_u64())
1331             .unwrap_or(1)
1332             .to_string(),
```

Listing 2.44: ethereum.rs

Similar issue exists in the `authorization_list`. The parameter `chainId` only supports string format and silently defaults to "0x0" when provided as a number.

```
1273                     chain_id: auth_obj
1274                         .get("chainId")
1275                         .unwrap_or(&json!("0x"))
1276                         .as_str()
1277                         .unwrap_or("0x0")
1278                         .to_string(),
```

Listing 2.45: ethereum.rs

In addition, in the current implementation, the default chain id logic at line 308 can not work as expected as it will be overwritten by line 1331.

```
307     if transaction_data.get("chain_id").is_none() {
308         transaction_data["chain_id"] = json!("1");
309     }
```

Listing 2.46: ethereum.rs

Impact This leads to mismatch of the user intended chain id and the signed chain id.

Suggestion Revise the logic accordingly.

2.1.18 Potential signature collision in the function `process_bit_sign_message()`

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description The function `process_bit_sign_message()` is designed to process JSON messages by sorting keys alphabetically and concatenating their values, but it completely discards the key names themselves, retaining only the values in sorted order.

This design can lead to signature collisions for different JSON objects. For instance, assume there are three messages: `{"a": "hello", "b": "world"}`,`{"x": "hellow", "y": "orld"}` and `{"p": "hell", "q": "oworld"}`, all of the ouputs are "helloworld".

This means different JSON structures can produce identical concatenated strings, which will subsequently be used to sign.

```
1622 fn process_bit_sign_message(&self, message: &str) -> String {
1623     //      JSON    message
1624     let message_content = if message.starts_with('{') {
1625         if let Ok(json_value) = serde_json::from_str::<Value>(message) {
1626             //
1627             let mut keys: Vec<_> = json_value.as_object().unwrap().keys().collect();
1628             keys.sort();
1629             keys.into_iter()
1630                 .map(|key| json_value[key].as_str().unwrap_or(""))
1631                 .collect::<Vec<_>>()
1632                 .join("")
1633         } else {
1634             message.to_string()
1635         }
1636     } else {
1637         message.to_string()
1638     };
1639
1640     //      bitsign888$#@5!{message_content}$bitend@1*
1641     format!("bitsign888$#@5!{}$bitend@1*", message_content)
1642 }
```

Listing 2.47: ethereum.rs

Impact Potential signature collision when using the function `process_bit_sign_message()`.

Suggestion Revise the code logic accordingly. Confirmed and this is by design

Feedback from the project The project states that this is an acceptable design.

2.1.19 Empty access list prevents gas optimization

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the EIP-2930, EIP-1559, and EIP-7702 transaction signing implementations, the code creates transactions with `access_list` support but always initializes the `access_list` as an empty vector, completely negating the gas optimization benefits that EIP-2930 was designed to provide. This prevents users from optimizing their transactions' gas cost by supplying `access_list` data, potentially wasting thousands of gas per transaction in DeFi scenarios involving multiple storage reads/writes.

```
1004        access_list: ethereum_tx_sign::AccessList(vec![]), // Empty access list
```

Listing 2.48: ethereum.rs

```
1075        access_list: ethereum_tx_sign::AccessList(vec![]), // Empty access list
```

Listing 2.49: ethereum.rs

```
1845    // 9. accessList (empty for 7702)
1846    stream.begin_list(0);
```

Listing 2.50: ethereum.rs

Impact Preventing users from optimizing gas cost.

Suggestion Revise the logic accordingly.

Feedback from the project The project states that they ignore access list gas optimization in a short term.

2.1.20 Function `process_bit_sign_message()` skips non-string JSON fields

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The function `process_bit_sign_message()` implements a custom normalization strategy for JSON-formatted inputs prior to signature generation. When the input string starts with `'{'`, the function attempts to parse it using `serde_json::from_str::<Value>()`, extract all top-level keys, sort them alphabetically, and concatenate the corresponding values to form the message payload to be signed.

However, the implementation uses `json_value[key].as_str().unwrap_or("")` to extract each value. The function `.as_str()` only returns a value when the JSON field is of type string. For all other valid JSON types, including numbers, booleans, nulls, objects, and arrays, it returns None, which is then coerced into an empty string via `.unwrap_or("")`.

As a result, any non-string fields present in the input JSON (such as amounts, identifiers, nonces, or other numeric parameters) are silently excluded from the generated message payload.

```
1621 ///
1622 fn process_bit_sign_message(&self, message: &str) -> String {
1623     //      JSON    message
1624     let message_content = if message.starts_with('{') {
1625         if let Ok(json_value) = serde_json::from_str::<Value>(message) {
1626             //
1627             let mut keys: Vec<_> = json_value.as_object().unwrap().keys().collect();
1628             keys.sort();
1629             keys.into_iter()
1630                 .map(|key| json_value[key].as_str().unwrap_or(""))
1631                 .collect::<Vec<_>>()
1632                 .join("")
1633         } else {
1634             message.to_string()
1635         }
1636     } else {
1637         message.to_string()
1638     };
1639
1640     //      bitsign888$#@5!{message_content}$bitend@1*
1641     format!("bitsign888$#@5!{}$bitend@1*", message_content)
1642 }
```

Listing 2.51: ethereum.rs

Impact This may cause JSON inputs containing non-string primitives to be incomplete or incorrect signatures.

Suggestion Revise the logic accordingly.

Feedback from the project The project states that upstream will pass JSON object with string field only.

2.1.21 Improper handling logic of `__type` field in function `parse_message_format()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In file `ethereum.rs`, the function `parse_message_format()` determines the message signing method by inspecting the input format. If the input is a JSON object, the function attempts to read the `__type` field to identify the intended signing method (e.g., personal, eth_sign, or typed_data).

However, when the input message is a valid JSON object but does not contain the `__type` field, the function does not return an error. Instead, it silently falls through to the default behavior and returns `MessageType::PersonalSign` (line 1607).

As a result, a structured JSON message that was intended to be signed using another signing scheme (e.g., eth_sign or typed_data) may be incorrectly interpreted as a personal-sign message, producing an unintended signature type.

```
1547    if message.starts_with('{') {
1548        if let Ok(json_value) = serde_json::from_str::<Value>(message) {
1549            if let Some(msg_type) = json_value.get("__type").and_then(|v| v.as_str()) {
```

Listing 2.52: ethereum.rs

```
1605    } else {
1606        //
1607        (MessageType::PersonalSign, message.to_string())
1608    }
```

Listing 2.53: ethereum.rs

Impact This design may produce an unintended signature type, leading to incorrect signature results.

Suggestion Revise the logic accordingly.

Feedback from the project The project states that this is intended behavior.

2.1.22 Improper handling logic for `message` in function `sign_personal_message()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In file `ethereum.rs`, the function `sign_personal_message()` treats the input `message` as a hexadecimal string when it begins with `0x`. However, this prefix-based heuristic can misclassify a plain string that starts with `0x` (e.g., an address) and decode it as hexadecimal bytes.

```
737    // Handle hex messages for PersonalSign
738    let message_bytes = if message.starts_with("0x") {
739        hex::decode(message.trim_start_matches("0x"))
740            .unwrap_or_else(|_| message.as_bytes().to_vec())
741    } else {
742        message.as_bytes().to_vec()
743    };
```

Listing 2.54: ethereum.rs

Impact As a result, legitimate messages may be signed over unintended bytes, leading to incorrect signatures and verification failures.

Suggestion Revise the logic accordingly.

Feedback from the project The project states that for the plain text start with `0x`, upstream will encode it into hexadecimal bytes first.

2.1.23 Potential encryption key generation failure due to inconsistent hexadecimal prefix handling

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Current implementation handles uppercase `0X` and lowercase `0x` prefixes inconsistently when processing private keys. Some functions correctly support both formats by invoking function `trim_start_matches()` twice.

However, the function `get_encryption_public_key()` (line 631) only removes the lowercase `0x`. As a result, it fails to decode private keys that use an uppercase `0X` prefix.

```
626 async fn get_encryption_public_key(&self, private_key: String) -> WalletResult<String> {
627     use base64;
628     use x25519_dalek;
629
630     // Decode private key from hex
631     let private_key_bytes = hex::decode(private_key.trim_start_matches("0x"))
632         .map_err(|e| WalletError::InvalidPrivateKey(e.to_string()))?;
```

Listing 2.55: ethereum.rs

```
124     if format == "enc" {
125         //   Curve25519     x25519-xsalsa20-poly1305
126         self.get_encryption_public_key(final_private_key).await
127     } else if format == "secp256k1" || format.is_empty() {
128         //   secp256k1
129         let private_key_clean = final_private_key
130             .trim_start_matches("0x")
131             .trim_start_matches("0X")
132             .to_lowercase();
```

Listing 2.56: ethereum.rs

Impact Potential encryption key generation failure.

Suggestion Align prefix handling across all functions by adopting the same trim logic.

2.1.24 Lack of support for BIP39 passphrase

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The current implementation of the Ethereum wallet SDK does not support the BIP39 passphrase feature.

In the function `get_derived_private_key()`, it invokes the `to_seed()` function of the BIP39 library with a hardcoded empty string. This ignores any possibility of using a user-supplied passphrase for seed generation.

```rust
/// Get derived private key
async fn get_derived_private_key(&self, params: DerivePriKeyParams) -> WalletResult<String> {
    use bip32::XPrv;
    use bip39::{Language, Mnemonic};

    // Parse the mnemonic
    let mnemonic_str = &params.mnemonic;
    let mnemonic = Mnemonic::parse_in_normalized(Language::English, &mnemonic_str)
        .map_err(|e| WalletError::Unknown(format!("Invalid mnemonic: {}", e)))?;

    // Generate seed from mnemonic
    let seed = mnemonic.to_seed("");
```

Listing 2.57: ethereum.rs

Impact Users who have created wallets using a BIP39 passphrase on other platforms (e.g., Ledger, Trezor, MetaMask) may cannot import their wallets into this SDK correctly.

Suggestion Revise the logic accordingly.

Feedback from the project The project states that not supporting this feature is intended design.

2.1.25 Hardcoded default derivation path can derive incorrect private key

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The function `get_derived_private_key()` derives a private key when the user provides a mnemonic phrase for authentication. However, a hardcoded derivation path is used and might not match the original path used by the user. This mismatch may produce an incorrect account address.

```rust
    let final_private_key = if is_mnemonic {
        //
        self.get_derived_private_key(DerivePriKeyParams {
            mnemonic: private_key.to_string(),
            path: Some("m/44'/60'/0'/0/0".to_string()),
            r#type: None,
        })
        .await?
```

Listing 2.58: ethereum.rs

Impact This mismatch may produce an incorrect account address.

Suggestion Revise the logic accordingly.

Feedback from the project The project states that not supporting this feature is intended design.

2.1.26 Inappropriate use of signed integer parsing for transaction values

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `sign_transaction()` incorrectly implements data parsing logic that allows signed integers (e.g., `as_i64()`) to be used for strictly unsigned Ethereum transaction fields (i.e., `nonce`, `value`, `gas_price`, `max_fee_per_gas`, and `max_priority_fee_per_gas`). It is recommended to not support signed integers in unsigned fields to prevent potential logic errors, accidental underflows, or the processing of invalid negative values.

```
327                 } else if let Some(i) = n.as_i64() {
```

Listing 2.59: ethereum.rs

```
349             } else if let Some(n) = nonce_val.as_i64() {
```

Listing 2.60: ethereum.rs

```
366                 } else if let Some(i) = n.as_i64() {
```

Listing 2.61: ethereum.rs

```
430                 } else if let Some(i) = n.as_i64() {
```

Listing 2.62: ethereum.rs

```
451             } else if let Some(n) = nonce_val.as_i64() {
```

Listing 2.63: ethereum.rs

Impact This can lead to the execution of transactions with unintended value.

Suggestion Revise the logic accordingly.

2.1.27 Default handling logic in the transaction parsing process alters signature semantics

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In several functions (e.g., `sign_transaction()`, `sign_legacy_transaction()` and `sign_eip1559_transaction()`), parameters that are parsed incorrectly or fields that do not meet expectations are set to zero by default without raising an error. This may downgrade an originally invalid or user-unconfirmed transaction into a valid signature with completely different semantics.

```
318             if let Some(val) = transaction_data.get(*key) {
319                 let converted_value = if let Some(s) = val.as_str() {
320                     let parsed_value = s.parse::<f64>().unwrap_or(0.0);
321                     (parsed_value * 1_000_000_000_000_000_000.0) as u128
322                 } else if let Some(n) = val.as_number() {
323                     if let Some(f) = n.as_f64() {
324                         (f * 1_000_000_000_000_000_000.0) as u128
325                     } else if let Some(u) = n.as_u64() {
326                         (u as f64 * 1_000_000_000_000_000_000.0) as u128
327                     } else if let Some(i) = n.as_i64() {
328                         (i as f64 * 1_000_000_000_000_000_000.0) as u128
329                     } else {
330                         0
331                     }
332                 } else {
333                     0
334                 };
```

Listing 2.64: ethereum.rs

```
858     let nonce = if tx_params.nonce.starts_with("0x") {
859         U256::from_str_radix(&tx_params.nonce.trim_start_matches("0x"), 16).unwrap_or_default()
860     } else {
861         U256::from_str_radix(&tx_params.nonce, 10).unwrap_or_default()
862     };
863     let gas_price = if tx_params.gas_price.starts_with("0x") {
864         U256::from_str_radix(&tx_params.gas_price.trim_start_matches("0x"), 16)
865             .unwrap_or_default()
866     } else {
867         U256::from_str_radix(&tx_params.gas_price, 10).unwrap_or_default()
868     };
869     let mut gas_limit = if tx_params.gas_limit.starts_with("0x") {
870         U256::from_str_radix(&tx_params.gas_limit.trim_start_matches("0x"), 16)
871             .unwrap_or_default()
872     } else {
873         U256::from_str_radix(&tx_params.gas_limit, 10).unwrap_or_default()
874     };
```

Listing 2.65: ethereum.rs

```
942     let nonce = if tx_params.nonce.starts_with("0x") {
943         U256::from_str_radix(&tx_params.nonce.trim_start_matches("0x"), 16).unwrap_or_default()
944     } else {
945         U256::from_str_radix(&tx_params.nonce, 10).unwrap_or_default()
946     };
947
948     let max_priority_fee_per_gas = if tx_params.max_priority_fee_per_gas.starts_with("0x") {
949         U256::from_str_radix(
950             &tx_params.max_priority_fee_per_gas.trim_start_matches("0x"),
951             16,
952         )
953         .unwrap_or_default()
954     } else {
```

```
955         U256::from_str_radix(&tx_params.max_priority_fee_per_gas, 10).unwrap_or_default()
956     };
```

Listing 2.66: ethereum.rs

Impact This may result in unexpected behavior or execution logic.

Suggestion Revise the logic accordingly.

2.1.28 Inconsistent naming conventions for JSON parsing fields

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `convert_to_tx_param()`, when manually parsing JSON objects, some fields use only hardcoded strings in camelCase as keys. However, this is inconsistent with the naming conventions in `DEVELOPER_GUIDE.md`, where the exported JSON keys should only use snake_case.

This leads to confusion for callers when using the API. Furthermore, if callers follow the snake_case naming conventions specified in the `DEVELOPER_GUIDE.md` API documentation, it will result in missing parameters or parsing errors.

In the function `convert_to_tx_param()`, the current implementation only checks for the field `max_fee_per_gas` and `max_priority_fee_per_gas` at line 312, while the line 1317 checks both snake_case and camelCase (e.g., `maxFeePerGas` and `max_fee_per_gas`). In this case, if the user inputs valid `maxFeePerGas` and `maxPriorityFeePerGas`, but these fields are not matched at line 312, thus the function enters the Legacy branch rather than the EIP-1559 branch. This will lead to ambiguity.

```
1268     if let Some(auth_list) = obj.get("authorizationList") {
```

Listing 2.67: ethereum.rs

```
1274                         .get("chainId")
```

Listing 2.68: ethereum.rs

```
1292                         .get("yParity")
```

Listing 2.69: ethereum.rs

```
312     let is_eip1559 = transaction_data.get("max_fee_per_gas").is_some()
313         || transaction_data.get("max_priority_fee_per_gas").is_some();
```

Listing 2.70: ethereum.rs

```
1317     if obj.get("maxFeePerGas").is_some()
1318         || obj.get("max_priority_fee_per_gas").is_some()
1319         || obj.get("maxPriorityFeePerGas").is_some()
1320         || obj.get("max_fee_per_gas").is_some()
1321     {
```

Listing 2.71: ethereum.rs

Impact This leads to confusion for callers, and may result in parameter parsing errors or ambiguity.

Suggestion Keep the naming conventions consistent.

2.2 Recommendation

2.2.1 Remove redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There are some redundant code in the current codebase.

1. In the function `get_public_key()`, the public key is calculated using different cryptographic algorithms based on the `format`, including `Curve25519` and `secp256k1`. However, for unknown `format` values, the function does not revert directly but defaults to using `secp256k1`, which results in a large amount of redundant code. Additionally, if distinguishing unknown `format` values is required, an event should be triggered to ensure the logical difference between the two cases.

```
159     } else {
160         //    secp256k1
161         let private_key_clean = final_private_key
162             .trim_start_matches("0x")
163             .trim_start_matches("0X")
164             .to_lowercase();
165 
166         if private_key_clean.len() != 64
167             || !private_key_clean.chars().all(|c| c.is_ascii_hexdigit())
168         {
169             return Err(WalletError::InvalidPrivateKey(
170                 "Invalid private key format".to_string(),
171             ));
172         }
173 
174         let private_key_bytes = hex::decode(&private_key_clean)
175             .map_err(|e| WalletError::InvalidPrivateKey(format!("Hex decode error: {}", e)))?;
176 
177         let private_key_array: [u8; 32] = private_key_bytes.try_into().map_err(|_| {
178             WalletError::InvalidPrivateKey("Invalid private key length".to_string())
179         })?;
180 
181         let field_bytes: FieldBytes = *FieldBytes::from_slice(&private_key_array);
182         let signing_key = SigningKey::from_bytes(&field_bytes).map_err(|e| {
183             WalletError::InvalidPrivateKey(format!("Invalid private key: {}", e))
184         })?;
185 
186         let verifying_key = signing_key.verifying_key();
```

```
187         let public_key_bytes = verifying_key.to_encoded_point(false).as_bytes().to_vec();
188
189         Ok(format!("0x{}", hex::encode(public_key_bytes)))
190     }
```

Listing 2.72: ethereum.rs

2. In the function `rlp_encode_eip7702_transaction()`, the gas limit parsing logic contains a special case that checks `if gas_limit == "55000"` before falling through to the general decimal parsing logic. While the current implementation does not introduce functional bugs, this special case is redundant as the general parsing branch would handle this value correctly. The unnecessary condition reduces code readability and clarity.

```
1791         let gas_limit_u256 = if gas_limit == "55000" {
1792             U256::from(55000u64)
```

Listing 2.73: ethereum.rs

3. Several parameter definitions are present in the codebase but are not referenced or used by the Ethereum related implementation. The corresponding logic in the Ethereum module only relies on transaction construction, signing, and hashing components, while these parameters are never instantiated or consumed. It is recommended to remove parameter definitions that are not used by any active logic.

```
113 #[derive(Debug, Clone, Serialize, Deserialize)]
114 pub struct ValidAddressParams {
115     pub address: String,
116 }
117
118 /// Result of address validation
119 #[derive(Debug, Clone, Serialize, Deserialize)]
120 pub struct ValidAddressResult {
121     pub is_valid: bool,
122     pub address: String,
123 }
124
125 /// Parameters for validating private keys
126 #[derive(Debug, Clone, Serialize, Deserialize)]
127 pub struct ValidPrivateKeyParams {
128     pub private_key: String,
129 }
130
131 /// Result of private key validation
132 #[derive(Debug, Clone, Serialize, Deserialize)]
133 pub struct ValidPrivateKeyResult {
134     pub is_valid: bool,
135     pub private_key: String,
136 }
```

Listing 2.74: types.rs

```
199 #[derive(Debug, Clone, Serialize, Deserialize)]
200 pub struct MpcRawTransactionParam {
```

```rust
    pub private_key: String,
    pub transaction: serde_json::Value,
}

/// Parameters for MPC operations
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct MpcTransactionParam {
    pub private_key: String,
    pub transaction: serde_json::Value,
}

/// Parameters for MPC message operations
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct MpcMessageParam {
    pub private_key: String,
    pub message: String,
}

/// Parameters for hardware wallet operations
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct HardwareRawTransactionParam {
    pub private_key: String,
    pub transaction: serde_json::Value,
}

/// Parameters for calculating transaction hash
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct CalcTxHashParams {
    pub data: String,
}
```

Listing 2.75: types.rs

Suggestion It is recommended to remove redundant code to improve maintainability.

2.2.2 Unify mnemonic validation logic

Status Fixed in `Version 2`

Introduced by `Version 1`

Description File `ethereum.rs` uses two different approaches to detect whether input is a BIP-39 mnemonic phrase. Although the current implementation does not introduce functional bugs, the inconsistent logic reduces code maintainability and clarity.

```rust
    let private_key = if private_key_input.split_whitespace().count() >= 12 {
```

Listing 2.76: ethereum.rs

```rust
fn is_mnemonic(&self, input: &str) -> bool {
    let words: Vec<&str> = input.trim().split_whitespace().collect();
    //      12-24         3  BIP39
    words.len() >= 12 && words.len() <= 24 && words.len() % 3 == 0
}
```

Listing 2.77: ethereum.rs

Suggestion Unify mnemonic validation by reusing the existing function `is_mnemonic()`.

2.2.3 Replace unsafe type casting in the the function `ecdsa_sign()` to prevent potential overflow

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The field `v` is calculated as `v = recovery_id + (chain_id * 2 + 35) as u8` in the function `ecdsa_sign()`. However, using the `u8` type here means that if this signing logic is improperly applied to other chains, when `chain_id * 2 + 35 > 255`, an overflow occurs. While the current use cases for this function do not trigger an overflow, it is recommended to ensure the correct implementation of the function.

```
1455 /// ECDSA sign
1456 fn ecdsa_sign(
1457     &self,
1458     message_hash: &[u8; 32],
1459     private_key: &[u8],
1460     chain_id: u64,
1461 ) -> WalletResult<Signature> {
1462     use secp256k1::{Message, Secp256k1, SecretKey};
1463     let secp = Secp256k1::new();
1464     let msg = Message::from_digest_slice(message_hash)
1465         .map_err(|e| WalletError::Unknown(format!("Invalid message hash: {e}")))?;
1466     let sk = SecretKey::from_slice(private_key)
1467         .map_err(|e| WalletError::InvalidPrivateKey(e.to_string()))?;
1468     let recoverable_sig = secp.sign_ecdsa_recoverable(&msg, &sk);
1469     let (recovery_id, sig) = recoverable_sig.serialize_compact();
1470     let r = &sig[0..32];
1471     let s = &sig[32..64];
1472     let recovery_id = recovery_id.to_i32() as u8; // 0 or 1
1473     let v = if chain_id > 0 {
1474         recovery_id + (chain_id * 2 + 35) as u8
1475     } else {
1476         recovery_id + 27
1477     };
1478     Ok(Signature {
1479         r: r.to_vec(),
1480         s: s.to_vec(),
1481         v,
1482         recovery_id,
1483     })
1484 }
```

Listing 2.78: ethereum.rs

Suggestion It is recommended to perform the calculation using `u64` and safely convert it. If the value exceeds the allowed range, an error should be reported rather than being truncated.

2.2.4 Unify `chain_id` parse logic

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `chain_id` parsing logic varies across the functions `sign_legacy_transaction()`, `sign_eip1559_transaction()`, `sign_eip2930_transaction()`, and `sign_eip7702_transaction()`. To improve code readability, maintainability, and consistency, it is recommended to unify the handling of `chain_id` in these functions.

```
852     let chain_id = Self::parse_u64_safe(&tx_params.chain_id, "chain_id")?;
```

Listing 2.79: ethereum.rs

```
928     let chain_id = if tx_params.chain_id.starts_with("0x") {
929         U256::from_str_radix(&tx_params.chain_id.trim_start_matches("0x"), 16)
930             .unwrap_or_default()
931             .as_u64()
932     } else {
933         U256::from_str_radix(&tx_params.chain_id, 10)
934             .unwrap_or_default()
935             .as_u64()
936     };
```

Listing 2.80: ethereum.rs

```
1023    let chain_id = if tx_params.chain_id.starts_with("0x") {
1024        U256::from_str_radix(&tx_params.chain_id.trim_start_matches("0x"), 16)
1025            .unwrap_or_default()
1026            .as_u64()
1027    } else {
1028        U256::from_str_radix(&tx_params.chain_id, 10)
1029            .unwrap_or_default()
1030            .as_u64()
1031    };
```

Listing 2.81: ethereum.rs

```
1095    let chain_id = Self::parse_u64_safe(&tx_params.chain_id, "chain_id")?;
```

Listing 2.82: ethereum.rs

Suggestion Unify `chain_id` parsing logic in the functions mentioned above.

2.3 Note

2.3.1 Deprecated EIP-1024 due to security concern

Introduced by `Version 1`

Description The EIP-1024 Cross-client Encrypt/Decrypt has been deprecated due to security concerns and a lack of standardization. The primary issue identified was the cryptographic

misuse of using the same Ethereum key (on the secp256k1 curve) for both signing transactions and encrypting data on a different curve (x25519). Upstream software and users should be aware of associated risks when using this SDK.

Feedback from the project The project has removed the corresponding implementation in `Version 2`.

2.3.2 Blob transaction not supported

Introduced by `Version 1`

Description In file `ethereum.rs`, the function `sign_transaction_internal_impl()` does not support blob transactions (type 0x3), introduced by EIP-4844. Upstream software and users should be aware of associated risks when using this function for post-Dencun Ethereum networks.

Feedback from the project The project states that blob transaction will not be supported by this SDK at this time.

2.3.3 Input parameters from external callers must be verified and secure

Introduced by `Version 1`

Description This ETH Rust SDK exposes interfaces for external consumption. As input data originates from external sources, the security model relies in part on the integrity of the calling program. For instance, it is the responsibility of the external program to ensure the user's private key is not used to sign unintended transactions.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS