Highload Wallet Contract V3 (secp256k1) Implementation Security Audit

TONTECH VERIFIED ON 24 APR 2025

01 Summary

This audit report focuses on compliance with the TON Documentation and TL-B Scheme for the highload wallet contract v3 implementation.

Total findings	Resolved	Acknowledged	Declined	Unresolved
5	5	0	0	0

The findings were classified by the severity levels as following:

Critical	Major	Medium	Minor	Informational
0	0	0	3	2

02 Severity levels

Critical

These are the most severe security flaws in a smart contract. They can lead to immediate and significant risks, allowing attackers to gain unauthorized access, bypass security mechanisms, or cause significant financial losses. Critical vulnerabilities require immediate attention and remediation.

Major

These vulnerabilities are significant but may not pose an immediate threat or have a lower probability of exploitation compared to critical vulnerabilities. However, they still have the potential to result in significant damage or security breaches if left unaddressed. They should be resolved in a timely manner to ensure the overall security of the smart contract.

Medium

Medium vulnerabilities indicate security weaknesses that may not have a high likelihood of exploitation or immediate serious consequences. However, they can still impact the overall security posture of the smart contract and should be fixed to enhance its resilience against potential attacks or vulnerabilities.

Minor

These vulnerabilities are less critical in nature and generally have a limited impact on the security of the smart contract. While they may not pose an immediate threat, they should not be overlooked, as they can compound with other vulnerabilities or serve as a stepping stone for potential attackers. Addressing them is important to maintain a robust security posture.

Informational

Suggestions for code style, architecture, and optimization to improve quality and efficiency. Not security risks, but enhance resilience.

03 Codebase

Repository (`ton-blockchain/highload-wallet-secp256k1`)

The following commit contains code that was audited:

Target commit (`a084c2866dfae45792edcf9d7f08f78b83dd40e6`)

Code and contract hashes of the commit are provided in the next section.

04 Scope

Source code files (inside of `contracts/` directory):

File paths	Rollup SHA-256 hash of the files
`contracts/imports/stdlib.fc`	`4b77d3114254ce410485701457b54da9dd28596aa56d5b136020a97ad3665b49`
`contracts/highload-wallet-v3.func`	`2e8728bb076e271fd72f0fe0dca73428e8c8f0cd630dcb0058913254754c0cfc`
`contracts/secp256k1.func`	`134875a3faf122b0f4ef402a5bb6c5ded743027ca59f4b9c46b056ba554eb76d`

Rollup is obtained by calling `sha256sum contracts/imports/stdlib.fc contracts/highload-wallet-v3.func contracts/secp256k1.func`

Resulting hashes of the compiled smart contracts:

SC compile definition name	Compiled code cell hash
`HighloadWalletV3`	`0202d7b28d511a2a290cb8097f41a59d1593a282e28475dfec6d4008b87d6a1e`

`wrappers/HighloadWalletV3.compile.ts` definitions were used for compiling contracts and getting their hashes. Hash is equal to hash provided in `build/HighloadWalletV3.compiled.json`.

05 Evaluation of compliance to conduct

The TON Documentation defines behaviour and TL-B Scheme defines storage and messages schemes for the highload wallet contract v3 implementation. Our assessment confirms that the implementation correctly follows all requirements of both standards. The implementation includes all required message handlers, and follows the specified data structures and message formats.

Storage layout

Category	Severity	Location	Status
Compliance	Minor	contracts/highload-wallet-v3.func	Verified

Description

Storage schema describes storage layout and defined in the TL-B Scheme.

Implementation Details

The implementation follows the exact TL-B schema from the standard:

```
_ shift:uint13 bit_number:(## 10) {bit_number >= 0} { bit_number <= 1022 } =
QueryId;

storage$_ parity:uint1 public_key:bits256 subwallet_id:uint32 old_queries:
(HashmapE 13 ^Cell)
         queries:(HashmapE 13 ^Cell) last_clean_time:uint64 timeout:uint22
         = Storage;
```

Compliance Note

The implementation of contract storage follows the TL-B Scheme.

External Message Receiver

Category	Severity	Location	Status
Compliance	Minor	contracts/highload-wallet-v3.func	Verified

Description

The external message receiver implements the mechanism of sending transactions at very high rates defined in the TON Documentation and augmented in the TL-B Scheme. This operation allows wallet owners to send messages from wallet at a very high rate due to query replay protection.

Implementation Details

The implementation follows the exact TL-B Scheme:

```
_ shift:uint13 bit_number:(## 10) {bit_number >= 0} { bit_number <= 1022 } =
QueryId;

_ subwallet_id:uint32 message_to_send:^Cell send_mode:uint8 query_id:QueryId
created_at:uint64 timeout:uint22 = MsgInner;

signature$_ recovery_bit:uint1 r:uint256 s:uint256 = Secp256k1Signature;

msg_body$_ signature:Secp256k1Signature message:^(MsgInner) = ExternalInMsgBody;
```

The external message receiver correctly handles all required parameters:

1. `signature` `(Secp256k1Signature)` - signature of `MsgInner`
2. `subwallet_id` `(uint32)` - unique wallet id
3. `message_to_send` `(^Cell)` - serialized internal message that will be sent
4. `send_mode` `(uint8)` - send mode to send message with
5. `query_id` `(QueryId)` - query id used for replay protection
6. `created_at` `(uint64)` - message timestamp
7. `timeout` `(uint22)` - timeout between queries clean up

The implementation validates message params and verifies signature, securely protects against replay attacks, validates and sends an outbound message.

Compliance Note

The implementation follows the conduct described in the TON Documentation and TL-B Scheme. This operation enables the wallet to send one internal message — optionally to itself for batch actions — while ensuring high throughput and security.

secp256k1 Signature Verification

Category	Severity	Location	Status
Compliance	Minor	contracts/secp256k1.func	Verified

Description

The signature verification logic ensures that messages processed by the highload wallet are authentic and non-malleable. This is a critical part of the external message processing.

Implementation Details

The implementation uses a **public key recovery** mechanism via the `ECRECOVER` opcode. This process allows the contract to reconstruct the signer's public key from the signature and the message hash, enabling authentication without explicitly storing the full key on-chain.

Elliptic curve public keys on the `secp256k1` curve are points `(x, y)` that satisfy the equation $y^2 = x^3 + 7 \bmod p$, where `p` is a large prime number defining the field. For a given x-coordinate, there are always exactly two possible y-values that satisfy the equation: `y` and `p - y`. These two values are additive inverses modulo `p`, and because `p` is odd, the two y-values will always have opposite parity — one will be even, and the other will be odd. This property enables a public key to be stored in a compressed format: only the x-coordinate (256 bits) is stored, along with a single bit indicating the parity (the least significant bit) of the y-coordinate. This 1-bit value is enough to distinguish between the two valid y solutions during decompression, allowing full reconstruction of the public key point on the curve.

Reference: SEC 1 - Elliptic Curve Cryptography, §2.3.3

Compliance Note

The implementation uses this optimized variant of the public key format to minimize on-chain data storage and preserving gas efficiency without compromising security.

Internal Message Receiver

Category	Severity	Location	Status
Compliance	Informational	contracts/highload-wallet-v3.func	Verified

Description

The internal message receiver performs security checks and handles `internal_transfer#ae42e5a4` message defined in the TON Documentation. This operation allows wallet owners to execute up to 254 actions.

Implementation Details

The implementation follows the exact TL-B schema from the standard:

```
internal_transfer#ae42e5a4 {n:#} query_id:uint64 actions:^(OutList n) =
InternalMsgBody n;
```

The `internal_transfer` operation correctly processes all required parameters:

1. `query_id` `(uint64)` - arbitrary request number
2. `actions` `(^(OutList n))` - serialized actions list

The implementation performs several critical validations: it ensures that the body contains the correct number of bits and references, verifies that the message is not bounced, and checks that the sender is the contract's own address. After completing these validations, the contract saves the current code, updates the action list with the actions provided in the message, and sets the code to ensure it remains unchanged.

Compliance Note

The implementation follows the conduct described in the TON Documentation and TL-B Scheme. This operation enables the wallet to execute up to 254 actions while ensuring security.

Get-Methods Implementation

Category	Severity	Location	Status
Compliance	Informational	contracts/highload-wallet-v3.func	Verified

Description

The get-methods implement the data retrieval interfaces defined in TON Documentation. These methods allow to access wallet data, such as the public key, subwallet ID, timeout settings, and query processing history.

Implementation Details

The Highload Wallet v3 implements 5 get methods:

```
int get_public_key();
int get_subwallet_id();
int get_last_clean_time();
int get_timeout();
int processed?(int query_id, int need_clean);
```

Get Public Key

The `get_public_key()` method returns:

1. `public_key` `(int)` - the compressed public key. Module of value is `x` point. If the value is positive then `parity = 0`, otherwise `parity = 1`.

Get Subwallet Id

The `get_subwallet_id()` method returns:

1. `subwallet_id` `(int)` - the subwallet id of the contract

Get Last Clean Time

The `get_last_clean_time()` method returns:

1. `last_clean_time` `(int)` - the time of the last cleaning

Get Timeout

The `get_timeout()` method returns:

1. `timeout` `(int)` - the timeout value

Processed

The `processed?(int query_id, int need_clean)` method accepts:

1. `query_id` `(int)` - id of query
2. `need_clean` `(int)` - flag when set to true does query cleanup based on `last_clean_time` and `timeout`

And returns:

1. `is_processed` `(int)` - flag whether the `query_id` has been processed

Compliance Note

The implementation fulfills all TON Documentation requirements for data retrieval interfaces. The get-methods provide essential chain analysis capabilities. These methods enable data verification and query deduplication.

06 Security Analysis

Our security assessment found that the implementation demonstrates a high level of security awareness and follows TON smart contract best practices. Key security aspects examined include:

1. Replay protection mechanism ensures that message will not be accepted twice by smart contract
2. Access control mechanisms properly restrict functions and validate sender ownership
3. Economic security measures prevent overflow/underflow issues
4. Message handling correctly processes incoming messages, including bounced messages
5. ECDSA algorithm with public key recovery on curve secp256k1 is used
6. Smart contract patterns are implemented properly, with no reentrancy vulnerabilities or unexpected code modifications

07 Issues

During our audit of the TON Blockchain Highload Wallet Contract V3, we identified the following findings.

Incorrect use of createdAt in Message Construction

Category	Severity	Location	Status
Integration risk	Informational	contracts/highload-wallet-v3.func	Resolved

Description

When constructing messages to send to the wallet contract, the `createdAt` field must be set slightly in the past rather than using the current timestamp. This is required because the destination node may not have produced a block yet at the time the message is received. If `createdAt` is set to the current or future time, the contract may reject the message as invalid.

Impact

Failure to set `createdAt` correctly can result in message rejection even if all other parameters are valid.

Recommendation

Ensure that `createdAt` is set to a recent but slightly earlier timestamp (e.g., the last block timestamp) to guarantee compatibility with network latency and block timing.

Risk of Execution Failure with mode = 128

Category	Severity	Location	Status
Execution risk	Minor	contracts/highload-wallet-v3.func	Resolved

Description

If the value for transfer to itself is not explicitly set, the wrapper uses `mode = 128`, sending the entire balance. When multiple external messages are processed rapidly, timing issues can arise. A message might send all funds just before another arrives. If the contract balance changes due to incoming commissions between these steps, later internal messages may fail due to insufficient funds, even though the request is marked as processed.

Impact

This leads to silent failures in transfers. The contract behaves as if it succeeded, but the intended actions never complete.

Recommendation

Avoid using `mode = 128`. Always set the `value` field explicitly and allocate at least 1 TON to ensure successful internal processing.

Initialization Parameters Mismatch

Category	Severity	Location	Status
Security risk	Minor	contracts/highload-wallet-v3.func	Resolved

Description

A contract's address is derived from its `state_init`, meaning any change in initial parameters affects the resulting address. Fields like `old_queries`, `queries`, and `last_clean_time` not included in the external message, but contribute to `state_init`. If these are not set to their default values (`null`, `null`, `0`) during deployment, it becomes possible to create multiple contracts with different addresses but the same public key, subwallet ID, and timeouts. This opens a risk where a signed message intended for one contract can be reused on another.

Impact

A message signed for one instance of a wallet could be accepted by another, allowing unintended execution of transactions across similar contracts with mismatched init parameters.

Recommendation

Ensure all deployments use default values for `old_queries`, `queries`, and `last_clean_time` fields in `state_init`, to guarantee consistent contract addresses and prevent message replay across wallets.

Timeout Configuration Sensitivity

Category	Severity	Location	Status
Logic issue	Minor	contracts/highload-wallet-v3.func	Resolved

Description

Timeouts used for cleaning query IDs must be carefully selected. If the timeout is too large, the wallet state may never be cleaned, causing the queries dictionary to grow indefinitely. The wallet stores queries in a limited-size dictionary with a hard cap of 8,380,415 entries per timeout period. Once full, new messages will be rejected until the timeout expires and the dictionary is cleared. Short timeout may result in messages not reaching the wallet in time before they expire, leading to their loss.

Impact

Improper timeout configuration can cause either message loss or indefinite message retention.

Recommendation

Set the timeout value to a duration between 1 and 24 hours. This range provides a balance between reliability and timely state cleanup.

Message Limitations

Category	Severity	Location	Status
Design limitation	Informational	contracts/highload-wallet-v3.func	Resolved

Description

Highload Wallet V3 can include up to 254 actions in a single `internal_transfer` call, with one action slot reserved to prevent unauthorized code updates. When batching more than 254 messages, the wallet nests the overflow batch into the 254th action, triggering another `internal_transfer` to continue processing. This mechanism ensures eventual execution of all messages across multiple calls.

Impact

Although the wrapper handles message splitting automatically, it assumes the entire packed structure fits within the 64 KB limit for an external message. Sending overly large batches increases the risk of exceeding this limit, causing processing failures or message rejection.

Recommendation

It is recommended to limit the number of actions per batch to 150. This helps ensure that even complex messages remain within the size constraints and are processed.

08 Conclusion

The TON Blockchain Highload Wallet Contract V3 implementation demonstrates full compliance with the TON Documentation behaviour, augmented with the TL-B Scheme. The code is mature and well-structured.

From a security perspective, the implementation correctly follows almost all requirements. The codebase implements necessary security measures, handles most edge cases properly, and applies appropriate access control across all operations.