Pashov Audit Group

Turnkey Security Review

October 27th 2025 - November 5th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Turnkey

Turnkey enables gasless transactions by letting delegated wallets execute user-signed operations. It combines TKGasStation and TKGasDelegate to handle execution, batching, and signature validation for off-chain authorized actions.

5. Executive Summary

A time-boxed security review of the **tkhq/gas-station** repository was done by Pashov Audit Group, during which **ast3ros**, **unforgiven**, **zark**, **aslanbek** engaged to review **Turnkey**. A total of **20** issues were uncovered.

Protocol Summary	
Project Name	Turnkey
Protocol Type	Gas Station
Timeline	October 27th 2025 - November 5th 2025

Review commit hash:
- 7e5da9855fc4655f6be256950788ea007b960646 (tkhq/gas-station)

Fixes review commit hash:
- e66103a5400c31a388e86c2ee04f0c33ea87722e (tkhq/gas-station)

Scope

`TKGasDelegate.sol` `TKGasStation.sol` `interfaces/`

6. Findings

Findings count

Severity	Amount
High	1
Medium	2
Low	17
Total findings	**20**

Summary of findings

ID	Title	Severity	Status
[H-01]	Authorized sender can bypass output contract validation in batch session execution	High	**Resolved**
[M-01]	Non-standard batch hash breaks EIP712 compliance	Medium	**Resolved**
[M-02]	Bytes wrapper misdecodes batch calls	Medium	**Resolved**
[L-01]	`TKGasDelegate` lacks EIP-1271 support	Low	**Resolved**
[L-02]	`TKGasDelegate` missing ERC-165 implementation violates EIP-1155	Low	**Resolved**
[L-03]	`hashBatchExecution` missing deadline parameter	Low	**Resolved**
[L-04]	Incorrect hash computation in `_executeSessionArbitraryNoReturn`	Low	**Resolved**
[L-05]	Unsafe decoding of the calls array data	Low	**Resolved**
[L-06]	Reverted transactions prevent nonce increase causing temporary DoS	Low	Acknowledged
[L-07]	Inconsistent `ethAmount` byte size	Low	Acknowledged
[L-08]	Contract `TKGasDelegate` have non-EOA behavior	Low	Acknowledged
[L-09]	Batch execution can cause unexpected results due to unspecified gas	Low	Acknowledged

ID	Title	Severity	Status
[L-10]	`TKGasStation` allows for calling defined session functions	Low	Resolved
[L-11]	Inconsistent free memory pointer reset	Low	Resolved
[L-12]	Dead function helpers should be removed	Low	Resolved
[L-13]	Redundant receive guard in `TKGasStation`	Low	Resolved
[L-14]	Return path and `noReturn` share signature, causing DoS to paymaster	Low	Acknowledged
[L-15]	Session burn ignores caller	Low	Acknowledged
[L-16]	ERC-20 `approve` success not validated	Low	Resolved
[L-17]	Storage collision allows nonce manipulation, unusable `TKGasDelegate`	Low	Resolved

High findings

[H-01] Authorized sender can bypass output contract validation in batch session execution

Severity

Impact: High

Likelihood: Medium

Description

The `_executeBatchSessionNoReturn` variant fails to validate that each `execution.to` matches the signed `outputContract`. This allows an authorized paymaster to execute calls to arbitrary contracts, completely bypassing the intended access control.

```
    function _executeBatchSessionNoReturn(
        bytes calldata _signature,
        bytes calldata _counterBytes,
        bytes calldata _deadlineBytes,
        bytes calldata _outputContractBytes, // Signed but not enforced
        IBatchExecution.Call[] calldata _calls
    ) internal {
        ...
        for (uint256 i = 0; i < length;) {
            IBatchExecution.Call calldata execution = _calls[i];
            uint256 ethAmount = execution.value;
>>>         address outputContract = execution.to; // No validation!
            bytes calldata _callData = execution.data;
            assembly {
                let ptr := mload(0x40)
                calldatacopy(ptr, _callData.offset, _callData.length)
                if iszero(call(gas(), outputContract, ethAmount, ptr, _callData.length, 0, 0))
{ revert(0, 0) }
            }
        ...
```

Consider a scenario:

A user signs a session allowing their paymaster to interact with a safe NFT game contract. The paymaster can then call `n executeBatchSession(bytes calldata data)` to drain all user funds by calling token contracts, DEXes, or any other contract.

Recommendations

Add the output contract validation check before executing each call.

Medium findings

[M-01] Non-standard batch hash breaks EIP712 compliance

Severity

Impact: Medium

Likelihood: Medium

Description

`TKGasDelegate::hashBatchExecution` derives the batch digest with `keccak256(abi.encode(_calls))`. EIP-712 requires hashing each `Call` struct with its own type hash and then folding the array as `keccak256(abi.encodePacked(hashCall1, hashCall2, ...))`. Because the contract omits those per-item hashes (and the shared `Call` type hash), signatures produced by generic EIP-712 tooling will disagree with on-chain verification.

```
function hashBatchExecution(uint128 _nonce, uint32 _deadline, IBatchExecution.Call[] calldata
_calls)
    external
    view
    returns (bytes32)
{
    bytes32 executionsHash = keccak256(abi.encode(_calls));
    bytes32 hash;
    assembly {
        let ptr := mload(0x40)
        mstore(ptr, BATCH_EXECUTION_TYPEHASH)
        mstore(add(ptr, 0x20), _nonce)
        mstore(add(ptr, 0x40), _deadline)
        mstore(add(ptr, 0x60), executionsHash)
        hash := keccak256(ptr, 0x80)
        mstore(0x40, add(ptr, 0x80)) // Update free memory pointer
    }
    return _hashTypedData(hash);
}
```

You can find more about the EIP712 spec and how it must hash arrays here.

Recommendations

Rework the batch hashing to follow the EIP-712 array pattern: hash each `IBatchExecution.Call` with its type hash, store the 32-byte results in memory, and take the keccak256 of that packed list before embedding it in the outer struct.

[M-02] Bytes wrapper misdecodes batch calls

Severity

Impact: Medium

Likelihood: Medium

Description

Several bytes-only entry points, such as `TKGasDelegate::executeBatchSession(bytes calldata)` and `TKGasDelegate::executeBatchSessionArbitrary(bytes calldata)`, decode the `Call[]` payload using hard-coded offsets (`add(105, 0x40)`, etc.) instead of basing them on `data.offset`. When these helpers are invoked, the decoded array points to garbage, causing reverts or malformed batches despite a valid signature.

```
function executeBatchSessionArbitrary(bytes calldata data) external {
    IBatchExecution.Call[] calldata calls;
    assembly {
        calls.offset := add(85, 0x40)
        calls.length := calldataload(add(85, 0x20))
    }
    _executeBatchSessionArbitraryNoReturn(data[0:65], data[65:81], data[81:85], calls);
}
```

Recommendations

Derive offsets from `data.offset` (e.g., `calls.offset := add(data.offset, 0x69)`), ensuring the array is decoded relative to the actual calldata location before forwarding to the internal executor.

Low findings

[L-01] `TKGasDelegate` lacks EIP-1271 support

TKGasDelegate does not implement the EIP-1271 standard signature validation interface. When an EOA delegates to TKGasDelegate via EIP-7702, the account appears as a smart contract (with `extcodesize == 23`). Many protocols and libraries detect this and attempt to verify signatures using EIP-1271's `isValidSignature(bytes32 hash, bytes memory signature)` method rather than traditional ECDSA recovery.

For example, OpenZeppelin's `SignatureChecker` library uses contract code size to determine the verification method:

```
    function isValidSignatureNow(address signer, bytes32 hash, bytes memory signature) internal
view returns (bool) {
        if (signer.code.length == 0) {
            (address recovered, ECDSA.RecoverError err, ) = ECDSA.tryRecover(hash, signature);
            return err == ECDSA.RecoverError.NoError && recovered == signer;
        } else {
            return isValidERC1271SignatureNow(signer, hash, signature);
        }
    }
```

Without EIP-1271 implementation, external protocols attempting off-chain signature verification will fail when interacting with TKGasDelegate enabled accounts.

[L-02] `TKGasDelegate` missing ERC-165 implementation violates EIP-1155

The TKGasDelegate contract implements ERC-1155 receiver functions (`onERC1155Received` and `onERC1155BatchReceived`) but fails to implement ERC-165's `supportsInterface` function.

According to EIP-1155 specification:

```
    Smart contracts MUST implement the ERC-165 supportsInterface function and signify support
for the ERC1155TokenReceiver interface to accept transfers. See “ERC1155TokenReceiver ERC-165
rules” for further details.
```

It's recommended to implement the `supportsInterface` function to comply with EIP-1155.

[L-03] `hashBatchExecution` missing deadline parameter

The `BATCH_EXECUTION_TYPEHASH` is defined as:

```
    bytes32 private constant BATCH_EXECUTION_TYPEHASH =
        0x14007e8c5dd696e52899952d0c28098ab95c056d082adc0d757f91c1306c7f55;
    // Original: keccak256("BatchExecution(uint128 nonce,uint32 deadline,Call[]
calls)Call(address to,uint256 value,bytes data)")
```

However, `function hashBatchExecution(uint128 _nonce, IBatchExecution.Call[] calldata _calls)` omits the deadline parameter entirely:

```
    function hashBatchExecution(uint128 _nonce, IBatchExecution.Call[] calldata _calls)
        external
        view
        returns (bytes32)
    {
        // Keep abi.encode for complex types (abi.encodePacked doesn't support Call[] arrays)
        bytes32 executionsHash = keccak256(abi.encode(_calls));
        bytes32 hash;
        assembly {
            let ptr := mload(0x40)
            mstore(ptr, BATCH_EXECUTION_TYPEHASH)
            mstore(add(ptr, 0x20), _nonce)
            mstore(add(ptr, 0x40), executionsHash)
            hash := keccak256(ptr, 0x60)
            mstore(0x40, add(ptr, 0x60)) // Update free memory pointer
        }
        return _hashTypedData(hash);
    }
```

This mismatch means signatures generated will fail validation during execution, as internal functions like `_executeBatch` correctly include the `deadline` in the hash computation.

[L-04] Incorrect hash computation in `_executeSessionArbitraryNoReturn`

The `_executeSessionArbitraryNoReturn` function incorrectly computes the EIP-712 hash by including the `_outputContract` parameter, which is not part of the `ARBITRARY_SESSION_EXECUTION_TYPEHASH` definition.

The type hash is correctly defined as:

```
    bytes32 private constant ARBITRARY_SESSION_EXECUTION_TYPEHASH =
        0x37c1343675452b4c8f9477fbedff7bcc1e7fa8b3bc97a1e58d4e371c86bd64bb;
    // Original: keccak256("ArbitrarySessionExecution(uint128 counter,uint32 deadline,address
sender)")
```

However, the hash computation incorrectly includes a 4th field:

```
    function _executeSessionArbitraryNoReturn(
        bytes calldata _signature,
        bytes calldata _counterBytes,
        bytes calldata _deadlineBytes,
        address _outputContract,
        bytes calldata _arguments
    ) internal {
```

```
        ...
            mstore(ptr, ARBITRARY_SESSION_EXECUTION_TYPEHASH)
            let counterValue := shr(128, calldataload(_counterBytes.offset))
            mstore(add(ptr, 0x20), counterValue)
            mstore(add(ptr, 0x40), deadline)
            mstore(add(ptr, 0x60), caller())
>>>         mstore(add(ptr, 0x80), _outputContract) // Should not be included
            hash := keccak256(ptr, 0xa0) // Should be 0x80
        ...
    }
```

It's recommended to remove the incorrect `outputContract` field from the hash computation.

[L-05] Unsafe decoding of the calls array data

Functions `_executeBatch()` receives a calls array as bytes and decodes it while checking the signature. The issue is that signature check is done by hashing the raw bytes, and decoding is done without safety checks, and code may try to decode a malformed encoding, which will result in executing transactions that aren't users intention to execute them.

```
function _executeBatch(
        bytes calldata _signature,
        bytes calldata _nonceBytes,
        bytes calldata _deadlineBytes,
        bytes calldata _calls
    ) internal returns (bytes[] memory) {
        // Hash the raw encoded calls slice to match the off-chain preimage exactly
        bytes32 executionsHash = keccak256(_calls);
...
        IBatchExecution.Call[] calldata calls;
        uint256 length;
        assembly {
            calls.offset := add(_calls.offset, 0x40)
            calls.length := calldataload(add(_calls.offset, 0x20))
            length := calls.length
        }
```

There are two issue: 1. Code assumes that `offset` of the calls array would be exactly 0x40. 2. Code doesn't check that call array doesn't go beyond `msg.data.length`.

As a result of #1, it may be possible to trick user to sign a malformed encoding that contain 2 transactions, but the `calls.length` is 1 in the encoding, and `calls.offset` is pointing to transaction 1. When this encoding is decoded in the `_executeBatch()` code first transaction would execute even so the encoding points to the second tx. The signature check would pass because user signed the whole encoding.

As a result of #2, it may be possible to trick user to sign a malformed encoding that `calls.length` is 2, but the signed data includes one call data. Later attacker can put the second call bytes after the `msg.data.length`. As code decodes the calls with low level assembly, it would allow looping through the second call, which is outside the `msg.data`. The signature check would bypass because it would only check the `msg.data` part of the calls.

This unsafe decoding without checking the offset value and length of the call opens an attack vector that can be exploited in some scenarios. It would be better if the code decode the calls and then encode them and check the signature with this encoded value instead of hashing the raw bytes for signature check.

[L-06] Reverted transactions prevent nonce increase causing temporary DoS

In normal EOA even when user's transaction reverts, the nonce will be increased, but in the TKGasDelegate when ever the signed call reverts, the higher level transaction is reverted too, and nonce won't be increased, and it would cause a revert for the next transactions too, as the nonce wouldn't match:

```
        if iszero(call(gas(), outputContract, _ethAmount, ptr, _arguments.length, 0, 0)) {
revert(0, 0) }
```

This behavior isn't consistent with normal EOA behavior, and it creates a temporary DOS, as when one transaction reverts, then the next transactions would stuck too. Also, it won't be clear that the transaction is supposed to revert, and the app should ignore it or it should be executed again as the revert was temporary. So it's not easy to decide between retrying the transaction or just ignoring it and signing the next transaction with the same nonce. It would be better for user or app to choose if they want a silent failure and increase the nonce even when the inner call reverts.

[L-07] Inconsistent `ethAmount` byte size

There's different function ways to execute a signed transaction. The fallback function use custom encoded messages and normal solidity functions with parameters. The issue is that for fallback method, the ethAmount is 10 byte variable, and it's inconsistent with 32 byte value that is used in other methods:

```
                arguments.offset := add(107, 10) // Skip the 10-byte ethAmount
...
            // value is 32 bytes immediately after address
            value := calldataload(add(data.offset, 105))
```

Also, in some other chains and L2 the 10 byte integer would be around ~65k native token, which will not be enough to perform all the actions, specially in networks where the native token value is low (less than $1).

[L-08] Contract `TKGasDelegate` have non-EOA behavior

It's possible to call a normal EOA address with custom data and send ETH too. In the current implementation, the fallback function isn't payable, and it reverts if it was called with arbitrary data:

```
    fallback(bytes calldata) external returns (bytes memory) {
        ..........
        revert UnsupportedExecutionMode();
```

When an EOA delegates to the TKGasDelegate and executes a transaction, during that transaction the EOA address won't behave exactly like, and EOA account and it won't accept ETH with data calls, and as a result, some projects won't be compatible with TKGasDelegate (If they call EOA to send ETH with data).

[L-09] Batch execution can cause unexpected results due to unspecified gas

In the batch executions, code sends the remaining gas for the inner calls:

```
        for (uint256 i = 0; i < length;) {
            IBatchExecution.Call calldata execution = calls[i];
            uint256 ethAmount = execution.value;
            address outputContract = execution.to;
            bytes calldata _callData2 = execution.data;
            assembly {
                let ptr := mload(0x40)
                calldatacopy(ptr, _callData2.offset, _callData2.length)
                if iszero(call(gas(), outputContract, ethAmount, ptr, _callData2.length, 0, 0))
{ revert(0, 0) }
            }
            unchecked {
                ++i;
            }
        }
```

The issue is that the transaction result may differ based on gas amount (for example, for finalizing bridge transactions), and the gas consumption of the first transactions in the batch can be different than what has been simulated off-chain. As a result, the later calls in the batch will get less or higher gas, and the result of some calls in the batch will be different than what was user/app intended. This allows attacker to be able to manipulate transactions in some scenarios by changing gas amount. Attacker can change the state of the target contracts before executing the batch to change the gas consumption of calls and change the result of later calls.

[L-10] `TKGasStation` allows for calling defined session functions

TKGasStation contract doesn't allow calling session functions through fallback by checking custom function selector byte:

```
        bytes1 functionSelector = bytes1(data[22] & 0xf0); // mask the last nibble
        // only allow execute functions, no session functions
        if (functionSelector == 0x00 || functionSelector == 0x10 || functionSelector == 0x20) {
            (bool success, bytes memory result) = target.call(data[21:]);
```

The issue is that TKGasDelegate contract has defined functions that can pass the function selector byte. For example function `executeBatchSessionArbitrary(bytes calldata data)` has function signature `0xbb2e8fc8`, so it would be possible to call it from TKGasStation's fallback by setting `calldata` as `<target_adderss>bb2e8fc8<data_argument>`. This will pass the `TKGasStation.fallback()` check (because code check hex of the byte 22's first 4 bit to be 0, 1 or 2) and it would call `TKGasDelegate.executeBatchSessionArbitrary()`. This can happen to other functions that their signature passes the check.

[L-11] Inconsistent free memory pointer reset

`TKGasDelegate` builds multiple EIP-712 hashes by borrowing the free-memory pointer, but in some cases it doesn't advance it afterward. This is resulting in the remaining of dirty bytes in memory and an asynced free memory pointer, while in some functions the fmp is indeed being forwarded. The "asynced" functions are : `_executeNoValue`, `_executeWithValue`, `_executeNoValueNoReturn`, `_executeWithValueNoReturn`, `_approveThenExecuteWithParams`, `_approveThenExecuteNoReturnWithParams`, `_approveThenExecute`, `_approveThenExecuteNoReturn`. It is recommended to use a consistent pattern and call `mstore(0x40, add(ptr, size))` after deriving each hash in order to keep the allocator state predictable everywhere.

[L-12] Dead function helpers should be removed

Several internal helpers in `TKGasDelegate` are never referenced, yet they duplicate executable logic and reserve deployment bytecode:

- `_executeSessionArbitraryNoReturn(bytes, bytes, bytes, bytes, bytes)`
- `_executeSessionArbitraryNoReturn(bytes, bytes, bytes, address, bytes)`
- `_executeWithValueNoReturn(bytes, bytes, bytes, bytes, uint256, bytes)`.

We recommend removing these functions so that only reachable code ships.

[L-13] Redundant receive guard in `TKGasStation`

`TKGasStation::receive` only reverts, yet the contract's `fallback(bytes calldata)` is non-payable, so any stray ETH already reverts without this hook. The extra function adds bytecode and deployment cost but provides no real protection. Drop the receive function and keep the implicit revert behavior from the non-payable fallback.

```
receive() external payable {
    revert NoEthAllowed();
}
```

[L-14] Return path and `noReturn` share signature, causing DoS to paymaster

`TKGasDelegate::execute` and `TKGasDelegate::executeReturns` both rely on the identical EIP-712 struct, so anyone holding the user's signature can call the no-return path first, consume the nonce, and make a paymaster's subsequent `executeReturns` revert. The issue with this is that the paymaster can't fetch return data even though it broadcast a valid intent. As a result, a paymaster that relies on the returned bytes to continue its own transaction flow is blocked SINCE the intent is consumed before `executeReturns` can deliver the data it needs.

```
bytes32 private constant APPROVE_THEN_EXECUTE_TYPEHASH =
 0x5307a057487d127f168eaec165127bc70635758316af883df210876a14cac22a;
// Original: keccak256("ApproveThenExecute(uint128 nonce,uint32 deadline,address
erc20Contract,address spender,uint256 approveAmount,address outputContract,uint256
ethAmount,bytes arguments)")

function executeReturns(address _to, uint256 _ethAmount, bytes calldata _data) external returns
(bytes memory) {
    bytes memory result = _ethAmount == 0
        ? _executeNoValue(_data[0:65], _data[65:81], _data[81:85], _to, _data[85:])
        : _executeWithValue(_data[0:65], _data[65:81], _data[81:85], _to, _ethAmount,
_data[85:]);
    return result;
}

function execute(address _to, uint256 _ethAmount, bytes calldata _data) external {
    _ethAmount == 0
        ? _executeNoValueNoReturn(_data[0:65], _data[65:81], _data[81:85], _to, _data[85:])
        : _executeWithValueNoReturn(_data[0:65], _data[65:81], _data[81:85], _to, _ethAmount,
_data[85:]);
}
```

[L-15] Session burn ignores caller

`TKGasDelegate::burnSessionCounter` validates the user's EIP-712 signature, but it never compares the supplied `_sender` with `msg.sender`. Anyone who obtains the signature can submit it, even if the caller doesn't match the user's intent. While this only accelerates session expiry (no funds at risk), checking that `_sender == msg.sender` would align the on-chain guard with the signed payload.

```
function burnSessionCounter(bytes calldata _signature, uint128 _counter, address _sender)
external {
    bytes32 hash;
    assembly {
        let ptr := mload(0x40) // Get free memory pointer
        mstore(ptr, BURN_SESSION_COUNTER_TYPEHASH)
        mstore(add(ptr, 0x20), _counter)
        mstore(add(ptr, 0x40), _sender)
        hash := keccak256(ptr, 0x60)
        mstore(0x40, add(ptr, 0x60)) // Update free memory pointer
```

```
    }
    hash = _hashTypedData(hash);

    _requireCounter(_counter);
    if (ECDSA.recoverCalldata(hash, _signature) != address(this)) {
        revert NotSelf();
    }
    expiredSessionCounters[_counter] = true;
}
```

[L-16] ERC-20 `approve` success not validated

In all `approveThenExecute*` paths, the contract performs a low-level call to `token.approve(spender, amount)` and treats **any non-reverting call as success**. It does not check the return value:

- Some tokens legitimately return `false` on failure without reverting (weird tokens)
- Current logic proceeds to execute subsequent calls (e.g., relying on the approval) even if the token actually **returned** `false`.

This can cause downstream operations to run under the false assumption that an allowance was set, potentially breaking flows or enabling logic that depends on that allowance to misbehave

Recommendations

Validate the returndata from `approve`:

- If `returndatasize() == 0` → **accept** (old tokens).
- If `returndatasize() == 32` → **require** the decoded bool to be `true`.
- Otherwise → **revert**.

[L-17] Storage collision allows nonce manipulation, unusable `TKGasDelegate`

TKGasDelegate stores `nonce` and `expiredSessionCounters` in standard storage slots 0 and 1. Under EIP-7702, when users delegate between different contracts, these storage slots persist with the user's address and can be overwritten by other contracts.

```
    uint128 public nonce;

    mapping(uint128 => bool) public expiredSessionCounters;
```

This creates two issues:

- Nonce wrap around: If a user previously delegated to a contract that wrote a value near `uint128.max` to slot 0, the nonce will wrap around to 0, which can confuse users.
- Transaction replay: If a user delegates to another contract that resets or modifies slot 0 or slot `keccak256(abi.encode(key, 1))` (by having a mapping in slot 1 with uint128 as a key, it can write the exact same storage addresses), then re-delegates to TKGasDelegate, the `nonce` and `expiredSessionCounters` could be reset to a previous value. This allows replay of old signed transactions if their deadlines haven't expired.

The team acknowledges the issue:

```
    This is accepted because we have a deadline transactions and on step 2, if you delegate to
a malicious contract the attacker already has control.
```

While the documentation acknowledges this issue for malicious contracts, the problem also occurs with legitimate contracts that happen to use the same storage slots. Users switching between different EIP-7702 delegates for various purposes may inadvertently corrupt these critical security values.

Reference: https://eips.ethereum.org/EIPS/eip-7702#storage-management

Recommendations

It's recommended to use `ERC-7201: Namespaced Storage Layout` to prevent storage collisions regardless of which contracts users delegate to before or after `TKGasDelegate`.