Optimism Audit Report:

ECDSA Wallet

dapp.org

fv@dapp.org.uk

last updated: 12.01.2021

Table of Contents

Summary

From November 18th to November 27th, a team of four engineers spent a total of 5 person weeks reviewing the ECDSA Smart Wallet contracts for the OVM optimistic rollup.

This work was carried out against the following git repository:

- `ethereum-optimism/contracts-v2` at `bb3539bbd10c15a72a46cf4fb8d2472ef68f6322`

The team found 12 issues, of which 10 were high severity, and 7 were both high severity and high likelihood.

The team additionally identified 7 potential improvements to gas efficiency or code clarity.

The discovered issues can be broadly grouped into the following categories:

- Insufficient validation of transaction parameters

- Exploits in the relayer compensation mechanisms
- Memory handling errors in low level libraries
- Insufficient restrictions on user actions
- Unsafe math

Scope

The team reviewed the code contained within `OVM_ECDSAContractAccount.sol` and `OVM_ProxyEOA.sol` with the aim of validating the following security properties:

- The account will not execute any message which was not authenticated by a user
- The account will always execute a message which conforms to the standard Ethereum EOA specification
- The account's implementation cannot be upgraded unless the user consents to it
- The account cannot be destructed or otherwise caused to permanently brick

The OVM smart contracts are a large and complex system, and the team made the following assumptions to allow the analysis of the contract wallet in isolation from the system as a whole:

- The Execution Manager "correctly implements its EVM equivalents", i.e., each `ovmOPCODE` behaves as you would expect the OPCODE to behave in the EVM, except its inputs and outputs are call/returndata.
- The `ovmGETNONCE` and `ovmSETNONCE` opcodes work properly as a way for OVM contracts to access and update their own nonce
- The `Lib_SafeExecutionManagerInteraction` contract correctly allows for the above functionalities to be preserved, i.e., not only is the EVM correctly implemented, but using `Lib_SafeExecutionManagerInteraction` to access the EVM will not violate that correctness.

Tests

To facilitate the analysis, the team implemented a suite of integration and property tests using the dapptools toolbox, which can be found at https://github.com/dapp-org/optimism-tests/. This includes demonstrations of most of the vulnerabilities described in this document.

Team

The review was carried out by the following members of the dapp.org collective:

- David Terry
- Denis Erfurt
- Jenny Pollack
- Martin Lundfall

Changelog

A revision history for this document can be found here

System Overview

The OVM implements full account abstraction. End user accounts are represented by a smart contract wallet (`OVM_ProxyEOA`), which sits at the same address in the OVM as the users account does in L1.

This contract implements a user upgradable `delegatecall` based proxy that forwards all calls (except for `upgrade(address)` and `getImplementation()`) to the address stored at a hardcoded `IMPLEMENTATION_KEY`.

The `OVM_ExecutionManager` implements a new OVM specific opcode `ovmCREATEEOA`, which accepts a message hash and signature over that hash, and creates an `OVM_ProxyEOA` for the message signer. This opcode sets the implementation of the deployed proxy to the `OVM_ECDSAContractAccount` precompile (`0x4200000000000000000000000000000000000003`).

The `OVM_ECDSAContractAccount` has one method:

```
execute(
    bytes memory _transaction,
    Lib_OVMCodec.EOASignatureType _signatureType,
    uint8 _v,
    bytes32 _r,
    bytes32 _s
)
```

This takes a serialized transaction, a flag denoting its encoding, and a signature over that transaction. Two kinds of transaction encoding are supported:

1. The RLP encoding of:

```
(nonce, gasPrice, gasLimit, to, value, data, chainId)
```

1. An eth signed message (prefix: `\x19Ethereum Signed Message:\n32`) consisting of the abi encoding of:

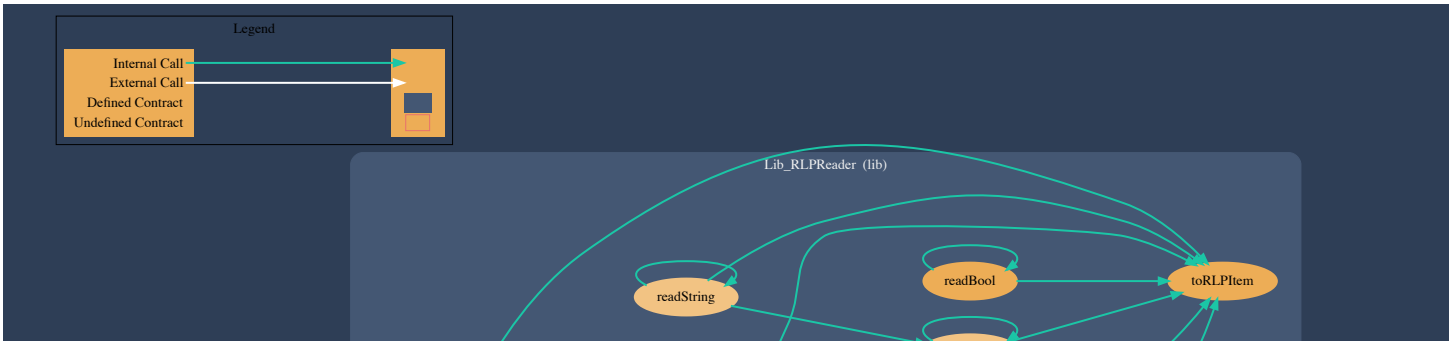
```
(nonce, gasLimit, gasPrice, chainId, to, data)
```

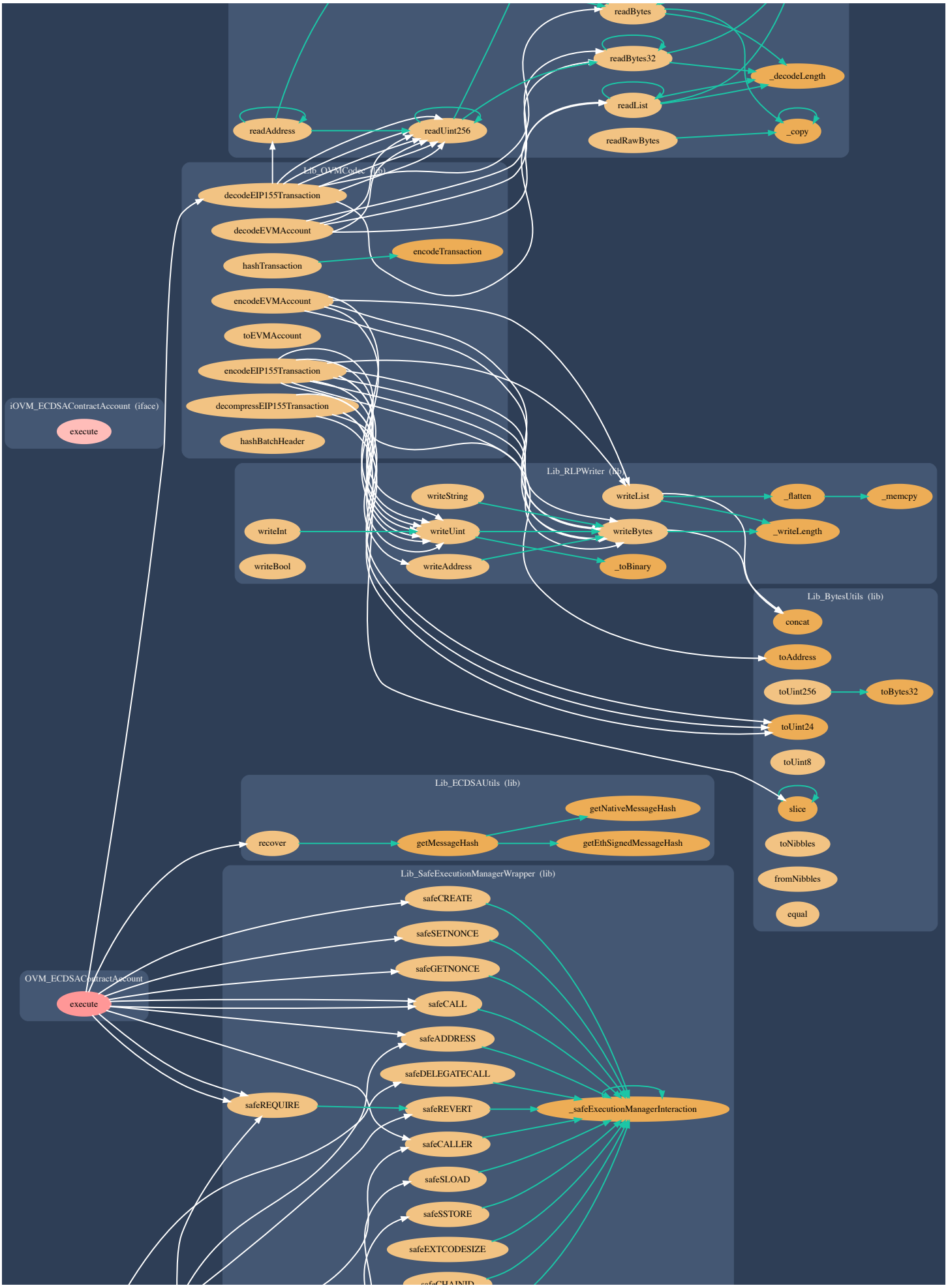
The `execute` method takes this transaction, checks that it was signed by the L2 address of the account, transfers an amount of the L2 `WETH` to the relayer (the caller of the `execute` method), and then executes the call specified in the transaction.

All calls to the `execute` method must be wrapped in a native OVM transaction, and included in the L2 transaction chain, either trustlessly via the L1 transaction queue (`OVM_CanonicalTransactionChain.appendBatch`) or by the OVM sequencer on behalf of the user (`OVM_CanonicalTransactionChain.appendSequencerBatch`).

Transactions submitted by the sequencer will by convention begin with a call into the `OVM_ProxySequencerEntrypoint` precompile (`0x4200000000000000000000000000000000000004`), which has it's initial implementation set to the `OVM_SequencerEntrypoint` precompile (`0x4200000000000000000000000000000000000005`), which uses a more compressed transaction representation and also creates EOA contract accounts as needed.

Contract Map




readBytes
readBytes32
_decodeLength
readList
readAddress
readUint256
readRawBytes
_copy
Lib_OVMCodec (lib)
decodeEIP155Transaction
decodeEVMAccount
encodeTransaction
hashTransaction
encodeEVMAccount
toEVMAccount
encodeEIP155Transaction
decompressEIP155Transaction
hashBatchHeader
iOVM_ECDSAContractAccount (iface)
execute
Lib_RLPWriter (lib)
writeString
writeList
_flatten
_memcpy
writeInt
writeUint
writeBytes
_writeLength
writeBool
writeAddress
_toBinary
Lib_BytesUtils (lib)
concat
toAddress
toUint256
toBytes32
toUint24
toUint8
slice
toNibbles
fromNibbles
equal
Lib_ECDSAUtils (lib)
getNativeMessageHash
recover
getMessageHash
getEthSignedMessageHash
Lib_SafeExecutionManagerWrapper (lib)
safeCREATE
safeSETNONCE
safeGETNONCE
OVM_ECDSAContractAccount
execute
safeCALL
safeADDRESS
safeDELEGATECALL
safeREQUIRE
safeREVERT
_safeExecutionManagerInteraction
safeCALLER
safeSLOAD
safeSSTORE
safeEXTCODESIZE

Findings

Our findings are separated into three sections:

- **Bugs**: issues that impact the security or correctness of the system
- **Improvements**: changes that could improve the clarity, functionality, or efficiency of the system, but that do not impact security or correctness
- **Notes and Miscellanea**: points of interest that do not merit an explicit recommendation for change

Bugs

Finding	Severity	Likelihood	Addressed
B01 L1 transactions can be replayed on L2	High	High	`0aa6e3a`
B02 Relayer can provide insufficient gas for inner transaction	High	High	`6a4d48a`
B03 Account overcharges fees if the tx uses less gas then specified in gasLimit	High	High	No
B04 Arithmetic overflow in relayer fee calculation	High	High	No
B05 Arithmetic underflow in gas limit calculation	High	High	`6a4d48a`
B06 Transactions can be executed despite failing gas transfer	High	High	`46e2f65`
B07 Incorrect casting in `LibBytes32.fromAddress`	High	High	`244424f`
B08 Arithmetic overflow in input validation for `LibBytesUtils.slice`	High	Low	`6712904`
B09 Memory corruption in `Lib_RLPWriter.writeAddress`	High	Low	`96baeba`

Finding	Severity	Likelihood	Addressed
B10 Memory corruption in `LibBytesUtils.slice`	High	Low	`ca84d45`
B11 `ovmSETNONCE` and `ovmCREATE` allows users to overflow their nonce	Medium	High	No
B12 The `value` of a transaction is silently ignored in `OVM_ECDSAContractAccount`	Low	High	No

B01 L1 transactions can be replayed on L2

The `chainID` field in the transaction passed to the `execute` method in the `ECDSAContractAccount` is not checked, and as such L1 messages can be replayed by anyone on L2.

When combined with B02, this allows an attacker to steal from any user that reuses their L1 address on L2 by replaying L1 transactions with insufficient gas and pocketing the excess.

The amount stolen is limited by the total amount ever spent on gas by that account on L1, and the attacker must also pay to include the replayed transactions on L2. Nevertheless, it seems reasonable to suggest that the attacker could profit to the tune of several hundred or thousand dollars for an active L1 account.

B02 Relayer can provide insufficient gas for inner transaction

The gas specified in the transaction to be executed by the `ECDSAContractAccount` can be higher than the gas provided by the relayer. This allows the relaying party to force any transactions to run out of gas at will, while still incrementing the nonce and marking the transaction as executed.

As noted below, the relayer always receives `gasLimit * gasPrice` L2 WETH as payment, independent of the actual gas used during execution. This means that relayers can profit by forcing an out of gas in the execution of the relayed transaction, while still collecting the full gas payment.

In order to guarantee that the gas provided by the relayer is sufficient to execute the transaction with `decodedTx.gasLimit` gas forwarded to the inner call, we recommend to add a check:

```
Lib_SafeExecutionManagerWrapper.safeREQUIRE(
    gasleft() >= safeAdd(decodedTx.gasLimit, buffer)
)
```

where `buffer` is sufficient to cover the gas costs of all of the transactions up to and including the `CALL/CREATE` which forms the entrypoint of the transaction.

B03 Account overcharges fees if the tx uses less gas then specified in gasLimit

The fee that pays gas for the transaction in the `OVM_ECDSAContractAccount` is computed directly by `gasLimit * gasPrice` and is not dependent on the actual gas used.

This leads to users overpaying for transactions when supplying more gas then necessary.

The audit team recommends that the remaining gas should be returned back to the user after the call is performed.

B04 Arithmetic overflow in relayer fee calculation

The calculation of the fee paid to the relayer in the `ECDSAContractAccount` is made using unchecked arithmetic (`uint256 fee = decodedTx.gasLimit * decodedTx.gasPrice`), where both `gasLimit` and `gasPrice` are user provided. This allows users to craft transactions that have a very high gas price or gas limit which do not result in a corresponding fee payment to the relayer.

B05 Arithmetic underflow in gas limit calculation

A similar issue exists in the calculation of the gas limit to be used when creating a new contract. In this case, setting a gas limit lower than `2000` results in an arithmetic underflow, and a huge gas limit will be passed to the call to `LibSafeExecutionManagerWrapper.safeCREATE`.

This allows users to execute transactions at a very high cost without sufficient compensation to the relayer.

B06 Transactions can be executed despite failing gas transfer

The call to `transfer` L2 WETH to pay for gas usage `ECDSAContractAccount` can REVERT, and the transaction will still execute despite the relayer not being compensated for the gas usage.

The audit team recommends to require a successful `transfer` before executing the transaction.

B07 Incorrect casting in `LibBytes32.fromAddress`

In `Lib_Bytes32Utils`, the `address _in` is cast to a `bytes32` as: `bytes32(bytes20(_in))`. The `bytes20` cast right pads its argument, which makes this casting inconsistent with the corresponding `toAddress` cast: `address(uint160(uint256(_in)))` which assumes the argument to be left padded.

B08 Arithmetic overflow in input validation for `LibBytesUtils.slice`

A lack of safemath on L168 and L100 in `LibBytesUtils` can cause malformed input data to bypass the out of bounds check, which leads to `slice` trying to allocate `2 ^ 256 -1` bytes of memory, for example given the input `LibBytesUtils.slice(bytes(""), 1)`.

This will clearly always fail and consume all available gas.

Recommendation: use safemath here, and elsewhere.

As of version `0.6.0`, Solidity supports slices of bytestrings natively. However, only calldata bytestrings are currently supported. The audit team recommends using native slices wherever possible instead of the custom implementation in `LibBytesUtils`.

B09 Memory corruption in `Lib_RLPWriter.writeAddress`

There is a memory corruption issue in `Lib_RLPWriter` that can cause unexpected division by zero, resulting in an assertion violation and unexpected transaction failures.

The error (as well as the related problem in `LibBytesUtils.slice`) stems from an incorrect usage of the Solidity free memory pointer.

By convention, Solidity stores the currently allocated memory size at memory locations `0x40-0x5f`, which can be retrieved by `mload(0x40)` in assembly. However, memory is not guaranteed to be empty at this location as:

> *Solidity always places new objects at the free memory pointer and memory is never freed (this might change in the future).*

Recommendation: always make sure to clear memory before writing.

B10 Memory corruption in `LibBytesUtils.slice`

The assembly code in `LibBytesUtils.slice` suffers from the same problem as `Lib_RLPWriter.writeAddress` wherein memory is not cleared before it is being used, leading to incorrect output and OOG errors.

The code here is copied from solidity-bytes-utils by Gonçalo Sá. The audit team notified the author who promptly addressed the issue with this fix.

B11 `ovmSETNONCE` and `ovmCREATE` allows users to overflow their nonce

The `ovmSETNONCE` opcode allows users to set their nonce to an arbitrary value, including `uint(-1)`. It also contains a check ensuring that the new nonce is greater than the current nonce. This check is not present if the nonce is incremented during a contract deployment with `ovmCREATE`.

If users set their nonce to `uint(-1)` and call `ovmCREATE`, the nonce will overflow and will end up 0.

This allows for users to replay their own transactions, which means that the OVM chain can have multiple state transitions triggered by the same transaction hash, invalidating the assumption made by ethereum clients that transaction hashes are sufficient to identify transactions.

B12 The `value` of a transaction is silently ignored in `OVM_ECDSAContractAccount`

The `value` field in the transaction passed to `OVM_ECDSAContractAccount` is silently ignored. Since native ether does not exist as such on L2, a nonzero `value` doesn't have any effect. Regardless, accepting nonzero values can lead to confusion and the risk of social engineering attacks depending on how these are displayed by chain explorers, etc.

The audit team recommends to add a check that will call `safeREVERT` if the `_transaction.value` field is non-zero.

Improvements

Recommendation	Implemented
I01 Use safemath	No
I02 Replace usage of `Lib_BytesUtils.concat` with `abi.encodePacked`	No
I03 Use `EIP-1967` for administering proxy implementation slots	No
I04 Make use of `immutable`	No
I05 Redundant casts in `OVM_SequencerEntrypoint`	No
I06 Missing documentation for location of `v` parameter in calldata	No
I07 Avoid duplication of transaction type enum	No

I01 Use safemath

The OVM contracts make extensive use of unchecked arithmetic, even in calculations involving values that can be controlled by end users. This makes reasoning about the code significantly more challenging and unnecessarily increases the risk of an unintentional overflow (several such issues were found as part of this engagement).

The audit team recommends replacing all uses of unchecked arithmetic with a safe math abstraction that calls `OVM_ExecutionManager.safeRevert` if an overflow is detected.

I02 Replace usage of `Lib_BytesUtils.concat` with `abi.encodePacked`

`Lib_BytesUtils.concat` is a complex piece of hand written assembly. The same result can be achieved by using `abi.encodePacked`.

The audit team recommends relying on the standard and well tested implementation in the `solc` compiler and replacing all usage of `Lib_BytesUtils.concat` with `abi.encodePacked`.

I03 Use `EIP-1967` for administering proxy implementation slots

EIP-1967 specifies a standard storage slot to be used for proxy implementation addresses.

Usage of this standardized storage slot would enable easier integration of the OVM into block explorers or other external tooling.

I04 Make use of `immutable`

The following storage variables do not change after construction and can be made immutable:

- `OVM_StateTransitioner.preStateRoot`
- `OVM_StateTransitioner.stateTransitionIndex`
- `OVM_StateTransitioner.transactionhash`
- `OVM_CanonicalTransactionChain.forceInclusionPeriodSeconds`
- `OVM_ExecutionManager.safetyChecker`

I05 Redundant casts in `OVM_SequencerEntrypoint`

Lines 61 and 71 of `OVM_SequqncerEntrypoint` contain redundant casts to `uint8` for the `v` value of the signature. `v` is already given type `uint8` on line 46.

These can be safely removed.

I06 Missing documentation for location of `v` parameter in calldata

The documentation of the expected calldata layout for the `OVM_SequencerEntrypoint` is missing an entry for the `v` parameter of the signature.

I07 Avoid duplication of transaction type enum

Both `Lib_OVMCodec` and `OVM_SequencerEntryPoint` contain separate definitions of semantically equivalent transaction type enums.

The audit team recommends removing the enum from `OVM_SequencerEntryPoint` and using the implementation from `Lib_OVMCodec` throughout.

Notes and Miscellanea

Use of `ABIEncoderV2`

The OVM contracts use the ABIEncoderV2. Although recently moved out of "experimental" status, the V2 encoder is still less tested than the V1 encoder and has been the cause of many recent `solc` bugs (ref). Usage of the V2 encoder increases the risk that a vulnerability will be introduced into the contracts by `solc` during compilation.

Very high values accepted for transaction parameters

The `ECDSAContractAccount` accepts transactions with `gasLimit` and `nonce` of `uint256`, whereas `geth` caps these values as the max value of a `uint64`.

Inconsistent ordering of transaction fields

The ordering of fields in the two transaction encodings supported by the `ECDSAContractAccount` differs. This may make life slightly harder for those integrating with the OVM.

Unchecked EOA creation

The `ovmCREATEOA` opcode does perform any checks on the contents of the message it has been passed, and as such allows anyone to create an EOA on the OVM on behalf of any possible public key, by passing `messageHash=0` hash.

Relayers must maintain an implementation allowlist

EOA accounts can be arbitrarily upgraded by their users, including to an implementation that does not pay a relayer fee. Relayers should maintain a client side allowlist of known good EOA implementations.

Use of `address(0)` as a sentinel value may interfere with trading workflows

Traders often use transactions to the zero address as a way to cancel pending orders.

The usage of transactions to zero as a sentinel value to indicate contract creation within the `ECDSAContractAccount` may interfere with these workflows.

Appendix A. Bug Classifications

Severity	
informational	The issue does not have direct implications for functionality, but could be relevant for understanding.
low	The issue has no security implications, but could affect some behaviour in an unexpected way.
medium	The issue affects some functionality, but does not result in economically significant loss of user funds.
high	The issue can cause loss of user funds.
Likelihood	
low	The system is unlikely to be in a state where the bug would occur or could be made to occur by any party.
medium	It is fairly likely that the issue could occur or be made to occur by some party.
high	It is very likely that the issue could occur or could be exploited by some parties.