PUBLIC

Code Assessment

of the Enso-Weiroll Smart Contracts

January 26, 2023

Produced for

by

Contents

1 Executive Summary 3
2 Assessment Overview 5
3 Limitations and use of report 8
4 Terminology 9
5 Findings 10
6 Resolved Findings 11
7 Notes 15

1 Executive Summary

Dear Connor,

Thank you for trusting us to help Enso with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Enso-Weiroll according to Scope to support you in forming an opinion on their security risks.

Enso implements Enso-Weiroll - a virtual machine that is capable of grouping a chain of smart contract function calls into a single transaction. This chain of operations, or scripts, can perform arbitrary calls with user-defined data and allow the output of one command to be used as the input for the subsequent commands.

The most critical subjects covered in our audit are functional correctness and memory consistency. Security regarding all the aforementioned subjects is high.

The general subjects covered are a check of the specification and error handling. The specification is improvable, e.g. examples of encoded data can be added. Error handling is improved, after the fix of Assumptions on output from unsuccessful call.

In summary, we find that the codebase provides a good level of security. The remaining unfixed Complexity of Commands Effect Evaluation issue is fundamentally linked to the same risks as any other Ethereum transaction - however, the novelty of Enso-Weiroll requires additional tooling and user education to minimize this risk.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	4
• Code Corrected	4
Low-Severity Findings	4
• Code Corrected	3
• Risk Accepted	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the `CommandBuilder.sol` and `VM.sol` code files inside the contracts folder of the Enso-Weiroll repository based on the README.md file as documentation. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	16 November 2022	0d658b5a6432d849c92c1ef3bcb9710b0004292e	Initial Version
2	10 January 2023	0318cc5187fd52534560ccaf6dc1244b96a31423	Version with fixes
3	17 January 2023	46ac1d111bc2f93a559d91b7e0dd987d536039b2	Version with fixes
4	24 January 2023	900250114203727ff236d3f6313673c17c2d90dd	Merge into main branch

For the solidity smart contracts, the compiler version `0.8.16` was chosen.

2.1.1 Excluded from scope

Since all the contracts in scope are abstract, we don't know how exactly they will be used. Any contract that will extend and use the assessed functionality might introduce integration bugs and thus is out of the scope.

2.2 Explicit Assumptions

The Enso-Weiroll heavily relies on the encoding of input parameters. Any party that will use the functionality of the Enso-Weiroll is assumed to understand and be aware of the consequences of the execution. They will need to be aware of the state changes that execution of the commands sequence will cause. While the weiroll encoded data is different from regular Ethereum transactions, we assume that the users have comprehensive tools for analysis of the command sequence.

2.3 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Enso implemented Weiroll - a virtual machine built on Ethereum that enables the execution of user defined scripts.

Weiroll allows the chaining of Ethereum transactions within a single transaction. Financial operations of arbitrary complexity can therefore be executed atomically, without the need for ad-hoc contract deployments.

The Weiroll architecture is split into a Virtual Machine (`VM.sol`), which performs external calls to execute the provided commands, and a CommandBuilder (`CommandBuilder.sol`), a library that implements ABI encoding at runtime for the arguments provided by the virtual machine.

2.3.1 The Virtual Machine

The Virtual Machine (VM) architecture is composed of an *execution loop*, a list of *bytes32[] commands* that are consumed by the execution loop, and a *bytes[] state* from which the loop reads inputs and to which it writes outputs of commands.

A command is a 32 bytes value with the following structure:

```
|0      | |          1| |                2                   3  |
|0 1 2 3|4|5 6 7 8 9 0|1|2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1|
|  sel  |f|    in     |o|              target                   |
```

The fields consist of:

1. `sel`: a function selector
2. `f`: a set of flags
3. `in`: a list of six input indices
4. `o`: an output index.
5. `target`: a target address

When `FLAG_EXTENDED_COMMAND` is set in the flags, the next command in the command list is treated as a list of 32 input indices, which substitutes the six input indices of a command.

The execution loop takes care of calling the target address with calldata consisting of the given function selector and an ABI encoded payload, defined by the list of input indices. The list of input indices encodes which elements of the state are to be included in the ABI encoded input, and in which position. With the use of special indices, arbitrarily nested tuples and variable length arrays can be encoded, up to a size limit.

The state consists of a dynamic array of up to 128 bytes strings, each bytes string being one state element. When the input for an external call is built, state elements are read and encoded according to the 8 bits index values in the input indices list. The lowest 7 order bits represent the offset of the element in the state array, the highest bit represents whether the indexed value is to be interpreted as a static value (constant length, one word in size), or a string (variable length, where the first word is the size).

The output index specifies the treatment for the value returned by the external call to `target`. It can be stored in a state slot as a static or dynamic variable, it can be saved as a tuple in a state slot, or it can be ABI decoded as an array of bytes strings and replace the whole state.

The VM doesn't implement any logic itself on how the output is produced from the inputs. Its mechanism of action is the calling of external `target` contracts with the supplied inputs and the optional writing of the output to the state. According to which flags are set in the command, a `CALL` with zero ether, `DELEGATECALL`, `STATICCALL`, or `CALL` with non-zero ether value will be performed. When a `CALL` with value is performed, the first input index is used to retrieve the ether value from the state. The remaining indices are used to build the ABI encoded input for the call as usual.

Finally, the special flag `FLAG_DATA` allows passing the content of a state element as the calldata to the external call without prepending the selector nor undergoing any encoding.

2.3.2 Command Builder

The library `CommandBuilder` implements an ABI encoder that can produce arbitrary payloads at runtime in the EVM. The function `buildInputs` takes as arguments the state, an array of bytes strings, a 4 bytes selector, and at most 32 8bits indices for inputs to be encoded.

Each index either points to a state variable, as a static or dynamic size variable, or it is a special index that indicates special actions for the command builder. The value `IDX_END_OF_ARGS` is used to terminate the encoding, and all indices beyond the current one are ignored. `IDX_USE_STATE` is used to encode at the current position the whole state array. `IDX_TUPLE_START` and `IDX_ARRAY_START` serve as opening parenthesis for the recursive encoding of dynamic tuples or dynamic arrays (arrays are encoded as tuples prepended with their number of elements). `IDX_DYNAMIC_END` closes the nested dynamic tuple (or array). This set of primitives can express the full range of encodings allowed by the Ethereum ABI.

Indexes that don't have special values are interpreted according to their higher order bit. If it is not set, the state at that index is treated as a static variable, 32 bytes in size, and is encoded in-place in the abi encoding. When the higher order bit is set, the state element at the index (after masking away the higher order bit) is treated as a variable length value: a string, bytes, array or tuple. An offset pointer is written in the abi encoded output, and the string is written at that offset.

The list of indices of the command needs to be terminated by `IDX_END_OF_ARGS`, and every opened dynamic tuple or array needs to be closed with `IDX_DYNAMIC_END`.

The code of `CommandBuilder` contract is structured such, that there are two passes over the indices. The first pass is used to assess the total bytes length of the encoding, and to count the number of elements of every nested dynamic tuple or array. The second pass uses the information from the first pass to write the encoded output.

2.3.3 Changes in Version 1

Following changes affect the system overview:

The `DELEGATECALL` operation was removed from the `VM`.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	1

- Complexity of Commands Effect Evaluation Risk Accepted

5.1 Complexity of Commands Effect Evaluation

Design Low Version 1 Risk Accepted

Due to the novelty and non-standard encoding of Weiroll, the end user will need to sign a transaction, without knowing full details about the execution consequences. Standard hardware and software wallets won't be able to decode the content of the commands. As a result, users will need to perform blind-signing - signing without verifying the full transaction details. Phishing attacks can be performed on users to trick them to sign commands that will impact the token balances in an undesired way. Users should be notified about this risk and only sign transactions from trusted sources and ideally after careful inspection.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	1
Medium-Severity Findings	4
Low-Severity Findings	3

- Function writeOutputs Can Corrupt Memory Code Corrected

- Assumptions on Output From Unsuccessful Call Code Corrected
- Dynamic Variable Encoding Is Assumed to Be Correct Code Corrected
- The Index Is Not Masked Code Corrected
- Value for the Call Can Be Loaded From Wrong Memory Location Code Corrected

- IDX_USE_STATE Case Not Handled Inside Tuples and Arrays Code Corrected
- Non-terminated Indices Fail Silently Code Corrected
- Unbalanced Tuple Starts and Ends Cause Silent Failure Code Corrected

6.1 Function writeOutputs Can Corrupt Memory

Correctness High Version 1 Code Corrected

To store the pointer of the return data the `writeOutputs` function performs a write to memory at the index `state + 32 + (idx & IDX_VALUE_MASK) * 32` . However, a check that this location still belongs to the `state` array of pointers is not performed. This effectively permits writing to locations in memory that can contain other variables, including data of other `state` elements. The command (maliciously or accidentally) can trigger such writing and cause unexpected results.

Code corrected:

A check was introduced that verifies that `idx & IDX_VALUE_MASK < state.length`.

6.2 Assumptions on Output From Unsuccessful Call

Design Medium Version 1 Code Corrected

Unsuccessful calls are assumed to revert with no output data, with output data of the type `Panic()` (4 bytes selector, empty payload), or with output data of the type `Error(string)` (4 bytes selector, 32 bytes pointer, 32 bytes string size, string content).

Errors can however have arbitrary signatures, which are up to the contract implementors to define.

For example, an error of type `Error(uint256,uint256)` will have its second integer interpreted as a string length in the VM error handling, potentially causing a memory expansion that will consume all the gas, if the `uint256` value is big enough.

Code corrected:

Additional checks have been introduced to interpret the return data of the error as a string only when it is appropriate to do so.

6.3 Dynamic Variable Encoding Is Assumed to Be Correct

Correctness | Medium | Version 1 | Code Corrected

When `CommandBuilder` builds inputs from the state, the variable length case for bytes and strings (not array, not tuple) does not verify that the state element at the given index is correct abi encoded data. The following consequences are possible:

The case when `state[idx & IDX_VALUE_MASK].length == 0` is not handled correctly. During the encode loop, this is executed `free += state[idx & IDX_VALUE_MASK].length` at line 113, or `offset += state[idx & IDX_VALUE_MASK].length` at line 320. However if the state element is the empty bytes sequence "", the `free` or `offset/pointer` pointer does not change. The encoding of such a state element will write a pointer to unallocated free memory. If any other dynamic variable is allocated afterward the pointer of the empty state element will point to the same location. The checks performed in `setupDynamicVariable()` requires `state[idx & IDX_VALUE_MASK].length % 32 == 0` - this does not prevent the empty state case.

Code corrected:

The following fix was done to address the issue:

A constraint was added for the dynamic variable case in the `setupDynamicVariable` function: in addition to `state[idx & IDX_VALUE_MASK].length % 32 == 0` check, a check that this lengths does not equal 0 was added. This resolves the issue.

Enso responded:

> Added check to revert if argLen == 0 (weiroll.js already encodes `0x` as a full bytes32 value, so the state generated with weiroll.js will be unaffected). Also, we now check the variable's encoded size is the same as the content size.

Note:

The `weiroll.js` library is out of scope for this assessment, however, encoding of an empty string as full bytes32 value does not fully comply with abi encoding. Such behavior was considered a bug in solidity. Some contracts with strict decoding rules might not accept empty strings encoded this way.

6.4 The Index Is Not Masked

Correctness | Medium | Version 1 | Code Corrected

The IDX_VALUE_MASK is not applied to index values at certain places:

1. Mask is not applied on the index in `VM` smart contract at line 94.

2. Mask is not applied on the index in `CommandBuilder` smart contract at line 396.

Code corrected:

The appropriate index masking has been applied.

6.5 Value for the Call Can Be Loaded From Wrong Memory Location

Correctness Medium Version 1 Code Corrected

When a call with value is performed in `VM` the first index is treated as an index for the state element. The read from this memory location is done via assembly instruction.

```
bytes memory v = state[uint8(bytes1(indices))];
assembly {
    callEth := mload(add(v, 0x20))
}
```

This mload skips 1 word - the length of the state element. However, the state element can be empty. In this case, the mload will read memory allocated for other data. Since `callEth` should be a `uint256` typed argument, it should be treated the same way as any other static variable.

Code corrected:

Enso responded:

> We now validate that the state element's length is 32 bytes and convert it into a uint256.

6.6 IDX_USE_STATE Case Not Handled Inside Tuples and Arrays

Correctness Low Version 1 Code Corrected

Indices with the value `IDX_USE_STATE` behave differently according to whether they belong to dynamic tuples or not. Inside dynamic tuples, the `0xfe == IDX_USE_STATE` index is treated as variable length data (bytes or string). 0xfe value is masked and used as an index to 126 state bytes element. Outside of dynamic tuples, it causes the whole state to be ABI encoded at that position. This difference in behavior is not mentioned in the specification.

Code corrected:

`IDX_USE_STATE` now explicitly reverts when used inside a dynamic tuple or array.

6.7 Non-terminated Indices Fail Silently

Design Low Version 1 Code Corrected

If `FLAG_EXTENDED_COMMAND` is used, and no `FF` index is included in the indices list, an invalid input will be produced by `CommandBuilder.buildInputs()` instead of reverting, causing the external contract call to have invalid data.

Code corrected:

A new variable, `indicesLength`, keeps track of the number of indices that needs to be considered by `commandBuilder.buildInputs()`

6.8 Unbalanced Tuple Starts and Ends Cause Silent Failure

Design | Low | Version 1 | Code Corrected

In `CommandBuilder.buildInputs()` every dynamic array or tuple start should be matched by an index of value `IDX_DYNAMIC_END`. Failing to match opening and closing structures causes `offsets` not to be updated, and invalid output to be produced. The invalid result risks being passed to arbitrary external calls.

Since functions `setupDymamicTuple` and `encodeDynamicTuple` need to encounter an index `IDX_DYNAMIC_LENGTH` to exit correctly, the alternative return statements at lines 226 and 343 are superfluous and should never happen.

Code corrected:

Function `setupDynamicTuple` now reverts if no terminating index is found for a dynamic tuple or array.

6.9 DELEGATECALL and SELFDESTRUCT

Informational | Version 1 | Code Corrected

The `VM` abstract contract allows `DELEGATECALL` to the address specified in the command. Any contract that will inherit this functionality must not perform a call to an arbitrary user-specified address. The delegate call to an address that has a bytecode with `SELFDESTRUCT` opcode will cause permanent destruction of the smart contract. Thus, it is important to allow `DELEGATECALL` only to trusted smart contracts.

Code corrected:

Enso responded:

> We have removed delegate calls from the VM entirely as there was both risks to the contract via self destruct as well as the ability to change storage values of the importing contract, potentially bricking the contract.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Calls to Addresses With No Code

Note Version 1

The low-level delegatecall, call, and staticcall operations will succeed when used with addresses with no code, however, in the VM there seems to be no reason to use them on addresses with no code, excluding the precompiled contracts. Only VALUECALL has a reason for being used on an address with no code.

7.2 Floating Pragma

Note Version 1

Enso-Weiroll uses the floating pragma `^0.8.16`. Several assumptions about the layout of memory are made in the code, which could potentially change without a major version upgrade. Any solidity compiler version needs to be carefully tested before the deployment of the code to ensure stable functionality.