Vyper

Audit

Presented by:

OtterSec contact@osec.io

Nicholas R. Putra nicholas@osec.io

Robert Chen r@osec.io

Contents

01 | Executive Summary

Overview

Vyper engaged OtterSec to perform an assessment of the `vyper` program. This assessment was conducted between September 22nd and October 6th, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 6 findings in total.

In particular, we identified a faulty optimization operation code that incorrectly merged sequential memory storage instructions with an overlap in the addresses (OS-VPR-ADV-00), highlighted a memory overflow issue from specifying excessively large array sizes (OS-VPR-ADV-01), and discovered a flaw in the function responsible for converting unsigned integers into corresponding string values, where it also converted signed integers (OS-VPR-ADV-02).

Additionally, we made recommendations for optimizing jump tables (OS-VPR-SUG-00), suggested the implementation of a check to avoid the collision of label names (OS-VPR-SUG-01), and suggested utilizing a compiler check to handle a specific instance where both the range and bound are numeral integers, and the range exceeds the bound (OS-VPR-SUG-02).

02 | Scope

The source code was delivered to us in a git repository at github.com/vyperlang/vyper. This audit was performed against release v0.3.10rc1.

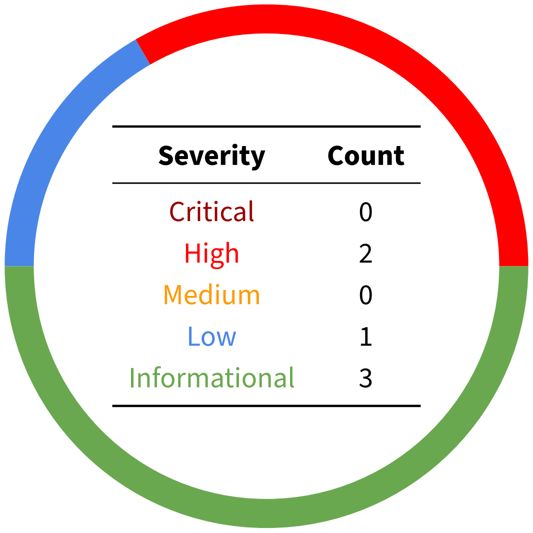
A brief description of the programs is as follows:

Name	Description
vyper	The core compiler code for the Vyper language is a contract-centric programming language with a Pythonic syntax designed for the Ethereum Virtual Machine (EVM).

03 | Findings

Overall, we reported 6 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
Critical	0
High	2
Medium	0
Low	1
Informational	3

04 | Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-VPR-ADV-00	High	Resolved	`mcopy` incorrectly merges the `mstore` instructions when there is an overlap of the addresses.
OS-VPR-ADV-01	High	Resolved	Specifying a large array size exceeding 2^{256} results in memory overflow.
OS-VPR-ADV-02	Low	TODO	`uint2str` incorrectly converts signed integers into strings.

OS-VPR-ADV-00 [high]| Incorrect Optimization

Description

`optimizer` contains `_merge_load`, an optimization function that aims to reduce redundant load operations in the intermediate representation code by merging consecutive load operations into a single copy operation when possible. In Ethereum, memory operations such as `mload` and `mstore` read and write data in memory. The `mload` operation reads a 32-byte word from memory, and the `mstore` operation writes a 32-byte word to memory.

irr/optimizer.py PYTHON

```python
def _merge_mload(argz):
    if not version_check(begin="cancun"):
        return False
    return _merge_load(argz, "mload", "mcopy")

def _merge_load(argz, _LOAD, _COPY):
    # look for sequential operations copying from X to Y
    # and merge them into a single copy operation
    changed = False
    [...]
    for i, ir_node in enumerate(argz):
        [...]
        # if we get this far, the current node is a different operation
        # it's time to apply the optimization if possible
        if len(mstore_nodes) > 1:
            changed = True
            new_ir = IRnode.from_list(
                [_COPY, initial_dst_offset, initial_src_offset, total_length],
                source_pos=mstore_nodes[0].source_pos,
            )
            # replace first copy operation with optimized node and remove the rest
            argz[idx] = new_ir
            # note: del xs[k:l] deletes l - k items
            del argz[idx + 1 : idx + len(mstore_nodes)]

        initial_dst_offset = 0
        initial_src_offset = 0
        total_length = 0
        mstore_nodes.clear()

    return changed
```

However, `_merge_load` incorrectly merges `mload`/`mstore` sequences into a single opcode: `mcopy`, when the source and destination buffers overlap, and the destination buffer is ahead of (i.e., has a greater offset than) the source buffer. This may result in incorrect behavior in the code from an incorrect optimization that should not be applied in such instances.

Proof of Concept

1. Consider the following sequence of instructions:
   - `mstore(32, mload(0))`: This instruction copies a 32-byte value from memory location zero and stores it in memory location 32.
   - `mstore(64, mload(32))`: This instruction copies a 32-byte value from memory location 32 and stores it in memory location 64.
2. In this scenario, we have two consecutive `mstore` instructions that appear to copy data from one location in memory to another, but they are not directly copied from source to destination. Instead, there is an overlap in memory locations. The second `mstore` operation utilizes the result of the first `mstore` as its source.
3. `_merge_load` tries to optimize this sequence by merging these two `mstore` instructions into a single `mcopy` instruction: `mcopy(32,0,64)`. The `mcopy` operation typically expects a non-overlapping source and destination, but in this instance, the destination buffer (64) is ahead of the source buffer (32), and they overlap.
4. Thus in `mcopy(32,0,64)`, while storing the value in memory location 64, instead of using the updated value in location 32 (loaded from location zero) as above, it will utilize the old value in 32, resulting in this optimization returning an incorrect value in the end.

Remediation

Ensure optimization does not occur if the `mcopy` regions overlap with each other.

Patch

Fixed in 2191fc3.

OS-VPR-ADV-01 [high] | Memory Overflow

Description

The memory overflow issue involves situations where Vyper may have inconsistent behavior with numbers exceeding the maximum representable value in Ethereum, 2^{256}. Ethereum's Ethereum Virtual Machine operates with 256-bit numbers, so exceeding this limit results in unexpected behavior. This issue may be particularly challenging in memory allocation and storage operations within smart contracts.

Proof of Concept

poc.vy VYPER

```vyper
@external
def zzz(x: DynArray[uint256, 3618502788666131106986593281521497120414
                     687020801267626233049500247285301213 9]):
    y: uint256[7] = [0, 0, 0, 0, 0, 0, 0]
    y[6] = y[5]
```

- `zzz` takes an extremely large dynamic array x as an argument, defined to exceed Ethereum's size limit.
- Inside the function, an array y of type `uint256[7]` is defined and initialized with seven zeros.
- The line `y[6] = y[5]` attempts to copy the value at index five of the y array into index six. However, due to the extremely large size of x, the behavior of this assignment is unpredictable.

Remediation

Ensure the size that may be allocated is not greater than a pre-determined value.

Patch

Fixed in 9c71339.

OS-VPR-ADV-02 [low] | Incorrect Conversion To String

Description

`uint2str` is a built-in function in Vyper language, which allows users to convert unsigned integers to string representations. It calculates the number of digits required to represent the integer as a string (`n_digits`), and in a loop, it iterates over each digit of the unsigned integer to convert it into its corresponding string value, building the string representation.

builtins/function.py PYTHON

```python
class Uint2Str(BuiltinFunction):
    _id = "uint2str"
    _inputs = [("x", IntegerT.unsigneds())]

    def fetch_call_return(self, node):
        arg_t = self.infer_arg_types(node)[0]
        bits = arg_t.bits
        len_needed = math.ceil(bits * math.log(2) / math.log(10))
        return StringT(len_needed)

    def evaluate(self, node):
        validate_call_args(node, 1)
        if not isinstance(node.args[0], vy_ast.Int):
            raise UnfoldableNode

        value = str(node.args[0].value)
        return vy_ast.Str.from_node(node, value=value)

    def infer_arg_types(self, node):
        self._validate_arg_types(node)
        input_type = get_possible_types_from_node(node.args[0]).pop()
        return [input_type]

    @process_inputs
    def build_IR(self, expr, args, kwargs, context):
        [...]
        return b1.resolve(IRnode.from_list(ret, location=MEMORY, typ=return_t))
```

However, the behavior of `uint2str` in Vyper, specifically when utilized with negative numbers as literals, is incorrect as it is intended to convert unsigned integers to strings. However, due to the lack of type checking in the function, it also accepts negative numbers as input, successfully converting signed integers into their corresponding string representations.

Proof of Concept

Below is an example of a Vyper snippet that showcases the issue:

vyper/ir/optimizer.py PYTHON

```python
@external
def test():
    a: String[78] = uint2str(-1)
    pass
```

In this code, a Vyper function called `test` attempts to use the built-in `uint2str` to convert the value `-1` to a string and assign it to a string array called a.

The issue is that `-1` is a negative number, but `uint2str` should only accept unsigned (non-negative) integers. Negative numbers do not have valid representations as unsigned integers, so attempting to convert `-1` in this context would result in an error.

Remediation

Implement a check to reject the conversion of signed integers in `uint2str`.

05 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-VPR-SUG-00	Optimizations for dense and sparse tables to improve efficiency.
OS-VPR-SUG-01	Collision of label names during generation of label names in `ir_identifier`.
OS-VPR-SUG-02	Addition of a compiler check to handle when both the range and bound are numeral integers.

OS-VPR-SUG-00 | Jump Table Optimizations

Description

In `_selector_section_dense`, for dense tables, a check is performed to ensure that the `calldatasize` is greater than or equal to four (indicating that there are sufficient bytes in the `calldata` to read the method ID).

codgen/module.py PYTHON

```python
def _selector_section_dense(external_functions, global_ctx):
    [...]
    with func_info.cache_when_complex("func_info") as (b1, func_info):
        [....]
        # check method id is right. If not, then fallback.
        # need to check calldatasize >= 4 in case there are
        # trailing 0s in the method id.
        calldatasize_valid = ["gt", "calldatasize", 3]
        [...]

    return ret
```

However, if the method identifier does not end with a zero, it indicates that the method identifier may have trailing zero bytes, and checking for a minimum `calldatasize` of four bytes may be unnecessary, and the `calldatasize_valid` check may be skipped.

codegen/module.py PYTHON

```python
def _selector_section_sparse(external_functions, global_ctx):
    for bucket_id, bucket in buckets.items():
        [...]
        for method_id in bucket:
            [...]
            expected_calldatasize = entry_point.min_calldatasize

            dispatch = ["seq"]  # code to dispatch into the function
            skip_callvalue_check = func_t.is_payable
            skip_calldatasize_check = expected_calldatasize == 4
            bad_callvalue = [0] if skip_callvalue_check else ["callvalue"]
            bad_calldatasize = (
                [0] if skip_calldatasize_check else ["lt", "calldatasize",
                    ↪  expected_calldatasize]
            )
    [...]
    return ret
```

Additionally, in `_selector_section_sparse`, for sparse tables, there is a check to ensure that the size of the `calldata` matches the expected `calldata` size for the selected function. This check may be skipped if the method identifier does not end with a zero or the expected `calldata` size for the selected function does not equal four.

Remediation

- For dense tables, skip the `calldatasize_valid` check if the selector does not end with zero.
- For sparse tables, skip the `calldata` size check if the selector does not end with zero or if `expected_calldatasize != 4`.

OS-VPR-SUG-01 | Label Name Collision

Description

In common, `ir_identifier` generates a unique and human-readable identifier for a contract function based on its visibility, name, and argument types. However, there is a possibility of a label collision in the logic for generating the label name in the current code. An example of such a collision would be as follows: `def zzz(x: uint8[3], y: uint8)` and `def zzz__uint8_3(y:uin8)`.

codegen/function_definitions/common.py PYTHON

```python
@cached_property
   def ir_identifier(self) -> str:
       argz = ",".join([str(argtyp) for argtyp in self.func_t.argument_types])
       return mkalphanum(f"{self.visibility} {self.func_t.name} ({argz})")
```

Remediation

Ensure unique label names are given for each function.

OS-VPR-SUG-02 | Compiler Check

Description

For the maximum bound in a range, in situations where both the range and the bound are numerical integers, and the range surpasses the bound, it would be advisable to incorporate a compiler check. When both values are numerical integers, and the range exceeds the bound, this check will trigger an exception in the compiler. This exception serves to indicate that the range exceeds the specified bound.

semantics/analysis/local.py PYTHON

```python
def visit_For(self, node):
        if isinstance(node.iter, vy_ast.Subscript):
            raise StructureException("Cannot iterate over a nested list",
                ↪  node.iter)

        if isinstance(node.iter, vy_ast.Call):
            [...]
            if len(args) == 1:
                # range(CONSTANT)
                n = args[0]
                bound = kwargs.pop("bound", None)
                validate_expected_type(n, IntegerT.any())
        [...]
```

Remediation

Add a compiler check to handle the above scenario.

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings section.

Critical	Vulnerabilities that immediately result in a loss of user funds with minimal preconditions. Examples: • Misconfigured authority or access control validation. • Improperly designed economic incentives leading to loss of funds.
High	Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions. • Exploitation involving high capital requirement with respect to payout.
Medium	Vulnerabilities that may result in denial of service scenarios or degraded usability. Examples: • Computational limit exhaustion through malicious input. • Forced exceptions in the normal user flow.
Low	Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions.
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants. • Improved input validation.

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.