Vyper Builtins

Security Assessment

March 21st, 2024 — Prepared by OtterSec

Naoya Okanami minaminao@osec.io

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

Vyper engaged OtterSec to assess the `builtins` program. This assessment was conducted between November 26th and December 30th, 2023. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 5 findings throughout this audit engagement.

In particular, we identified a memory overflow vulnerability in the application binary interface decoding (OS-VYP-ADV-00) and several instances of compiler panic, including when calculating the square root recursively or within a for loop (OS-VYP-ADV-01) and when a valid Ethereum function selector starts with zero (OS-VYP-ADV-02). Furthermore, we highlighted a scenario where the function responsible for recovering the address associated with the provided signature returns zero for valid signatures if the internally utilized precompiled contract fails due to running out of gas (OS-VYP-ADV-03).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/vyperlang/vyper. This audit was performed against commit b334218.

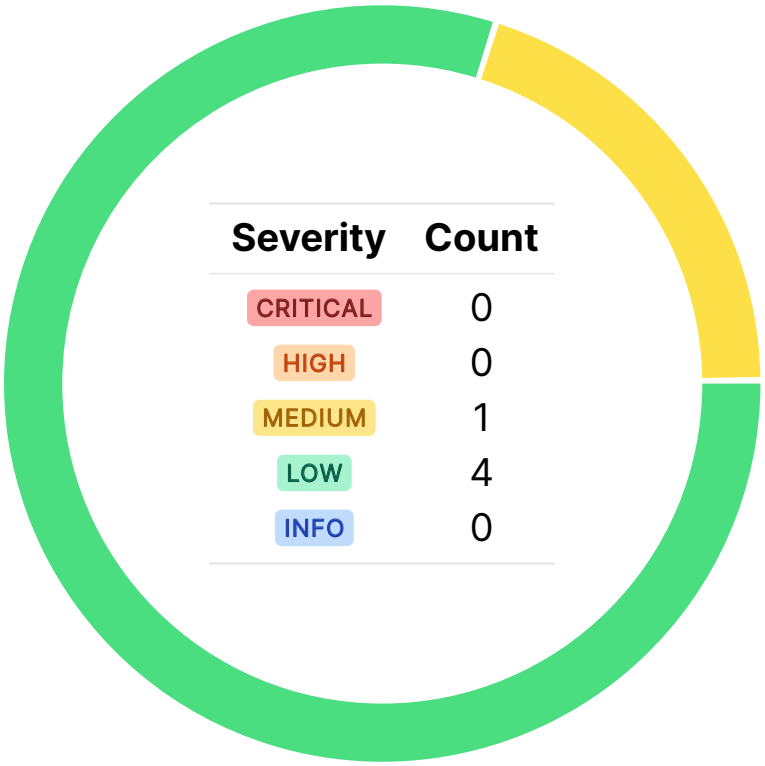
A brief description of the programs is as follows:

Name	Description
builtins	The program defines various built-in functions for the Vyper programming language.

03 — Findings

Overall, we reported 5 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	0
MEDIUM	1
LOW	4
INFO	0

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-VYP-ADV-00	MEDIUM	TODO	Memory overflow in application binary interface (ABI) decoding, within `_abi_decode` and external calls.
OS-VYP-ADV-01	LOW	TODO	Compiler panic when utilizing `sqrt` recursively or within a `for` loop.
OS-VYP-ADV-02	LOW	RESOLVED	The `MethodID` class in Vyper may encounter a compiler panic for valid Ethereum function selectors starting with zero.
OS-VYP-ADV-03	LOW	TODO	`ecrecover` returns zero for valid signatures if the internally utilized `ecrecover` precompiled contract fails due to running out of gas.
OS-VYP-ADV-04	LOW	TODO	In `raw_log`, specifying memory or storage variables as topics results in incorrect values being logged.

Memory Overflow In ABI Decoding MEDIUM OS-VYP-ADV-00

Description

A potential integer overflow risk exists within `_abi_decode`, specifically when the starting index for an array is excessively large. This overflow causes the read position to go beyond the intended array bounds during decoding, which may result in potential exploitations in contracts utilizing arrays within `_abi_decode`. This overflow issue also extends to return data, where the absence of boundary checking for the start index in the variable designated to hold the function's return data renders it susceptible to overflow in the start index.

Proof of Concept

The following example contract demonstrates the overflow:

>_ poc.vy vyper

```vyper
event Pwn:
    pass

@external
def f(x: Bytes[32 * 3]):
    a: Bytes[32] = b"foo"
    y: Bytes[32 * 3] = x

    decoded_y1: Bytes[32] = _abi_decode(y, Bytes[32])
    a = b"bar"
    decoded_y2: Bytes[32] = _abi_decode(y, Bytes[32])

    if decoded_y1 != decoded_y2:
        log Pwn()
```

With calldata as:

>_ calldata text

```
0xd45754f8
0000000000000000000000000000000000000000000000000000000000000020
0000000000000000000000000000000000000000000000000000000000000060
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffa0
```

- The contract has a function `f` that receives a `Bytes[32 * 3]` parameter `x`.
- It decodes two `Bytes[32]` values (`decoded_y1` and `decoded_y2`) from `x` utilizing `_abi_decode`.
- The critical part is the change in `a` between the two `_abi_decode` calls.
- The starting index is excessively large, resulting in an integer overflow in `_abi_decode`.
- The overflowed index leads to unintended array element decoding, resulting in the value of `a` being read.

Remediation

Ensure that the array's starting index is within range and implement comprehensive bounds checking when assigning return data to variables.

Compiler Panic Due To Variable Shadowing LOW OS-VYP-ADV-01

Description

Upon recursive invocation of `sqrt`, `generate_inline_function` creates a temporary variable named `range_ix0`. However, `generate_inline_function` generates multiple instances of `range_ix0s` (one for each invocation), duplicating `range_ix0s`.

Consequently, this results in scope issues due to the repeated generation of the same variable, causing `range_ix0s` to be shadowed. This shadowing triggers a panic in the compiler, resulting in the compiler throwing the following error:
`vyper.exceptions.CompilerPanic: shadowed loop variable range_ix0`.

Similar to the above scenario, when `sqrt` is called within a `for` loop, the loop iterator is converted to `range_ix0`. This, in turn, results in a variable collision analogous to the previously described case, prompting the compiler to throw the same error.

Remediation

Employ a different variable naming strategy for loop variables and temporary variables within the compiler to avoid naming conflicts.

Inconsistency In Function Selector LOW OS-VYP-ADV-02

Description

The issue involves a compiler panic for all functions whose function selector starts with zero when utilizing the `MethodID` class. The code assesses whether the inferred return type (`return_type`) matches the type `BYTES4_T` . If this condition holds, it endeavors to convert the value to a hexadecimal representation using `vy_ast.Hex.from_node` .

>_ *functions.py* python

```python
class MethodID(FoldedFunctionT):
    _id = "method_id"

    def evaluate(self, node):
        validate_call_args(node, 1, ["output_type"])

        args = node.args
        if not isinstance(args[0], vy_ast.Str):
            raise InvalidType("method id must be given as a literal string", args[0])
        if " " in args[0].value:
            raise InvalidLiteral("Invalid function signature - no spaces allowed.")

        return_type = self.infer_kwarg_types(node)
        value = method_id_int(args[0].value)

        if return_type.compare_type(BYTES4_T):
            return vy_ast.Hex.from_node(node, value=hex(value))
        else:
            return vy_ast.Bytes.from_node(node, value=value.to_bytes(4, "big"))
```

The vulnerability arises from the code checking for an exact match (`compare_type`) with `BYTES4_T` . Due to this, there is a 1/16 (or one in 16) probability that a valid Ethereum function selector starts with zero (i.e., one out of 16 possible hexadecimal values).

Remediation

Handle function selectors that start with zero correctly instead of checking for an exact match with `BYTES4_T` .

Patch

Fixed in 06fa46a

Function Gas Failure LOW

OS-VYP-ADV-03

Description

`ecrecover` recovers the address associated with the provided signature. The issue arises when the internally used `ecrecover` precompiled contract is called, and the static call fails due to running out of gas. Unlike Solidity's `ecrecover` implementation, Vyper's `ecrecover` does not explicitly check for a failed call to the internal `ecrecover` precompiled contract. If the internal call fails due to running out of gas, `ecrecover` will return zero, even for a valid signature.

Remediation

Explicitly check the result of the static call after calling the internal `ecrecover` precompiled contract or document the behavior of Vyper's `ecrecover`, highlighting the fact that it may return zero for valid signatures if the internal static call fails due to running out of gas.

Incorrect Topic Logging LOW OS-VYP-ADV-04

Description

The vulnerability concerns the potential incorrect logging of values when utilizing memory or storage variables. In `raw_log`, if memory or storage variables are employed as topics, `build_IR` fails to properly unwrap these variables. Consequently, incorrect values are logged as topics, potentially resulting in unexpected behavior in client-side applications reliant on these logs for processing or decision-making.

>_ functions.py python

```python
@process_inputs
def build_IR(self, expr, args, kwargs, context):
    [...]
    if data.typ == BYTES32_T:
        placeholder = context.new_internal_variable(BYTES32_T)
        return IRnode.from_list(
            [
                "seq",
                # TODO use make_setter
                ["mstore", placeholder, unwrap_location(data)],
                ["log" + str(topics_length), placeholder, 32] + topics,
            ]
        )
    [...]
    return IRnode.from_list(
        [
            "with",
            "_sub",
            input_buf,
            ["log" + str(topics_length), ["add", "_sub", 32], ["mload", "_sub"], *topics],
        ]
    )
```

In `build_IR`, when assembling the log operation, the topics are anticipated to be literals (immediate values). However, if memory or storage variables are provided as topics, they are not appropriately unwrapped to extract their actual values before logging. Consequently, the log operation may utilize incorrect values as topics, resulting in inconsistencies in the logged data.

Proof of Concept

1. The example contract provided below defines `f`, which executes two log operations using `raw_log`, each with a different variable (`self.x` and `y`) as the topic.
2. In both cases, the variable (`self.x` and `y`) is initialized with the same value (`0x1234567890123456789012345678901234567890123456789012345678901234`).
3. However, due to the vulnerability, the resulting logged topics are incorrect. For `self.x`, the topic is logged as `0x00`, and for `y`, the topic is logged as `0x40`.
4. This inconsistency in the logged topics may result in unexpected behavior in client-side applications relying on these logs, especially if they are used for critical operations such as tracking token transfers.

>_ poc.vy vyper

```vyper
x: bytes32

@external
def f():
    self.x = 0x1234567890123456789012345678901234567890123456789012345678901234
    raw_log([self.x], b"") # LOG1(offset:0x60, size:0x00, topic1:0x00)

    y: bytes32 = 0x1234567890123456789012345678901234567890123456789012345678901234
    raw_log([y], b"") # LOG1(offset:0x80, size:0x00, topic1:0x40)
```

Remediation

Ensure that memory or storage variables used as topics in log operations are properly unwrapped to extract their actual values before logging.

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.

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.

INFO

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.