CONSENSYS
Diligence

Vyper Preliminary Security Review

OCTOBER 28, 2019 BY STEVE MARX

A few weeks ago, we conducted a preliminary review of the Vyper compiler. Note that this wasn't an *audit*. An audit is typically a last step in the development process before a production release, and in an audit we attempt to be exhaustive in our analysis.

This was a one-week review, and our goal was to identify what work remains *before* Vyper is ready for a full security audit.

What Is Vyper?

Vyper, currently in beta, is a Python-like programming language for building smart contracts targeting the Ethereum Virtual Machine (EVM). Per the documentation, the goals of the language are:

> - **Security:** It should be possible and natural to build secure smart-contracts in Vyper.

> - **Language and compiler simplicity:** The language and the compiler implementation should strive to be simple.
> - **Auditability:** Vyper code should be maximally human-readable. Furthermore, it should be maximally difficult to write misleading code. Simplicity for the reader is more important than simplicity for the writer, and simplicity for readers with low prior experience with Vyper (and low prior experience with programming in general) is particularly important.

Our review focused on the middle bullet, helping to ensure the language and compiler could be easily audited.

Summary of Our Recommendations

At a high level, we had two major recommendations for Vyper:

1. Specify the language completely. Without knowing what the compiler is *supposed* to do, it's hard to verify that it's working correctly.
2. Decompose the compiler into smaller passes with well-defined interfaces. The current compiler architecture is difficult to audit because of how much happens all at once. As always, breaking a complex process into smaller, simpler components makes it much easier to get right.

If that second point intrigues you, I encourage you to read the full report. It goes into some detail about how compilers are typically architected and specifically where the Vyper compiler can be improved.

Notable Bugs

Although it was not our goal to find bugs in the Vyper compiler, we did include the bugs we saw in our report. You can find a list of them in section 10 of the report.

Below are a few of the more interesting/surprising bugs we found.

Ordering Dependencies

Vyper `struct` literals depend on the order of their fields:

```
struct Foo:
    a: uint256
    b: uint256

@public
@constant
def test() -> uint256:
    foo: Foo = Foo({a: 1, b: 2})
    return foo.a # returns 1

@public
@constant
def test2() -> uint256:
    foo: Foo = Foo({b: 2, a: 1})
    return foo.a # BUG: returns 2
```

This is due to the following code:

```
return LLLnode.from_list(
    ["multi"] + [o[key] for key in (list(o.keys()))],
    typ=StructType(members, name, is_literal=True),
    pos=getpos(expr),
)
```

Note that different versions and implementations of Python have different behavior when it comes to listing dictionary keys, so this Vyper behavior is actually dependent on *the version of Python* used to run the compiler.

For a completely different reason, `min` and `max` can give you the wrong answer depending on the order of the arguments supplied:

```
@public
@constant
def test1() -> uint256:
    return min(0, 2**255) # returns 2**255

@public
@constant
def test2() -> uint256:
    return min(2**255, 0) # returns 0
```

This is due to the combination of two things:

1. Vyper decides the type of a numeric literal based on its value. Small enough values are considered to be signed integers, while larger values are treated as unsigned.
2. The compiler decides whether to use a signed or unsigned comparison based on the *left* argument only.

In `test1()`, the first argument is small enough to be a signed integer, so a signed comparison is used. `2**255` treated as a signed value is a very large negative number.

An Infinite Loop

From the Vyper documentation:

> Following the principles and goals, Vyper **does not** provide the following features:
>
> …
>
> - **Infinite-length loops:** Similar to recursive calling, infinite-length loops make it impossible to set an upper bound on gas limits,

> opening the door for gas limit attacks.

We were able to create an infinite loop by making use of a compiler-generated variable:

```
@public
def infinite_loop():
    for n in [1,2,3]:
        _index_for_n = 0 # BUG
```

Overflows

One of my favorite Vyper features is that you don't need something like `SafeMath`. In Vyper, all integer math is checked automatically for overflows, and if an overflow happens, the transaction is reverted.

We did manage to find a few ways to skip these checks:

```
@public
@constant
def overflow_unsigned(x: uint256, y: uint256) -> uint256:
    return x**y # BUG: test(10, 78) returns 73663286101470436611432119930496737173840

@public
@constant
def overflow_signed(x: int128, y: int128) -> int128:
    return x**y # BUG: test(2, 256) returns 0

@constant
@public
def underflow_negation(x: uint256) -> uint256:
    return -x # BUG: returns 2**256 - x
```

Another Vyper Honeypot

My colleague constructed a clever honeypot last month using a Vyper bug relating to function selector collisions. (That bug has already been fixed.)

I wanted to build something similarly devious, so I took advantage of an active bug I had encountered a while back:

```
owner: public(address)
balances: public(map(address, uint256(wei)))
expiration: public(timestamp)

@public
def __init__():
    self.owner = msg.sender
    self.expiration = block.timestamp + (60 * 60 * 24 * 7)

@public
@payable
def __default__():
    if msg.value > 0:
        assert msg.value >= as_wei_value(1, "ether")
        self.deposit(msg.sender, msg.value)

    if msg.value == 0:
        self.withdraw(msg.sender)

@public
def kill():
    assert block.timestamp > self.expiration
    selfdestruct(self.owner)

@private
def deposit(account: address, amount: uint256(wei)) -> uint256(wei):
    self.balances[account] += amount
    return self.balances[account]

@private
def withdraw(account: address):
    send(account, self.balances[account])
```

The idea here is that the contract is seeded with a small amount of ether, and you should be able to get that ether out by first sending some more. A classic honeypot! But it's quite hard to see why you won't actually be able to get the ether out.

The issue has to do with stack height manipulation. Surprisingly, it's impossible to reach line 18 because a *stack underflow* occurs if the first conditional isn't entered.

The GitHub issue has more details for the curious.

Should You Use Vyper?

Vyper is an interesting language, and it's certainly worth a look if you're curious how language design can improve smart contracts.

However, Vyper is still in beta. Until Vyper has been fully audited and has a stable (non-beta) release, **we don't recommend using it in production**.

. . .

We *love* doing this type of preliminary review. It gives us a chance to suggest design changes early, when it's still cheap to address them. If you think your project could benefit from this type of feedback, please get in touch with us!

More posts >