# Smart Contracts and Vyper

Vyper is an experimental, contract-oriented programming language for the Ethereum Virtual Machine that strives to provide superior auditability, by making it easier for developers to produce intelligible code. In fact, one of the principles of Vyper is to make it virtually impossible for developers to write misleading code.

In this chapter we will look at common problems with smart contracts, introduce the Vyper contract programming language, and compare it to Solidity, demonstrating the differences.

# Vulnerabilities and Vyper

A recent study analyzed nearly one million deployed Ethereum smart contracts and found that many of these contracts contained serious vulnerabilities. During their analysis, the researchers outlined three basic categories of trace vulnerabilities:

**Suicidal contracts**

Smart contracts that can be killed by arbitrary addresses

**Greedy contracts**

Smart contracts that can reach a state in which they cannot release ether

**Prodigal contracts**

Smart contracts that can be made to release ether to arbitrary addresses

Vulnerabilities are introduced into smart contracts via code. It may be strongly argued that these and other vulnerabilities are not intentionally introduced, but regardless, undesirable smart contract code evidently results in the unexpected loss of funds for Ethereum users, and this is not ideal. Vyper is designed to make it easier to write secure code, or equally to make it more difficult to accidentally write misleading or vulnerable code.

# Comparison to Solidity

One of the ways in which Vyper tries to make unsafe code harder to write is by deliberately *omitting* some of Solidity's features. It is important for those considering developing smart contracts in Vyper to understand what features Vyper does *not* have, and why. Therefore, in this section, we will explore those features and provide justification for why they have been omitted.

## Modifiers

As we saw in the previous chapter, in Solidity you can write a function using modifiers. For example, the following function, `changeOwner`, will run the code in a modifier called `onlyBy` as part of its execution:

```
function changeOwner(address _newOwner)
    public
    onlyBy(owner)
{
    owner = _newOwner;
}
```

This modifier enforces a rule in relation to ownership. As you can see, this particular modifier acts as a mechanism to perform a pre-check on behalf of the `changeOwner` function:

```
modifier onlyBy(address _account)
{
    require(msg.sender == _account);
    _;
}
```

But modifiers are not just there to perform checks, as shown here. In fact, as modifiers, they can significantly change a smart contract's environment, in the context of the calling function. Put simply, modifiers are *pervasive*.

Let's look at another Solidity-style example:

```
enum Stages {
    SafeStage,
    DangerStage,
    FinalStage
}

uint public creationTime = now;
Stages public stage = Stages.SafeStage;

function nextStage() internal {
    stage = Stages(uint(stage) + 1);
}

modifier stageTimeConfirmation() {
    if (stage == Stages.SafeStage &&
                now >= creationTime + 10 days)
        nextStage();
    _;
}

function a()
    public
    stageTimeConfirmation
    // More code goes here
{
}
```

On the one hand, developers should always check any other code that their own code is calling. However, it is possible that in certain situations (like when time constraints or exhaustion result in lack of concentration) a developer may overlook a single line of code. This is even more likely if the developer has to jump around inside a large file while mentally keeping track of the function call hierarchy and committing the state of smart contract variables to memory.

Let's look at the preceding example in a bit more depth. Imagine that a developer is writing a public function called `a`. The developer is new to this contract and is utilizing a modifier written by someone else. At a glance, it appears that the `stageTimeConfirmation` modifier is simply performing some checks regarding the age of the contract in relation to the calling function. What the developer may *not* realize is that the modifier is also calling another function, `nextStage`. In this simplistic demonstration scenario, simply calling the public function `a` results in the smart contract's `stage` variable moving from `SafeStage` to `DangerStage`.

Vyper has done away with modifiers altogether. The recommendations from Vyper are as follows: if only performing assertions with modifiers, then simply use inline checks and asserts as part of the function; if modifying smart contract state and so forth, again make these changes explicitly part of the function. Doing this improves auditability and readability, as the reader doesn't have to mentally (or manually) "wrap" the modifier code around the function to see what it does.

## Class Inheritance

Inheritance allows programmers to harness the power of prewritten code by acquiring preexisting functionality, properties, and behaviors from existing software libraries. Inheritance is powerful and promotes the reuse of code. Solidity supports multiple inheritance as well as polymorphism, but while these are key features of object-oriented programming, Vyper does not support them. Vyper maintains that the implementation of inheritance requires coders and auditors to jump between multiple files in order to understand what the program is doing. Vyper also takes the view that multiple inheritance can make code too complicated to understand—a view tacitly admitted by the Solidity documentation, which gives an example of how multiple inheritance can be problematic.

## Inline Assembly

Inline assembly gives developers low-level access to the Ethereum Virtual Machine, allowing Solidity programs to perform operations by directly accessing EVM instructions. For example, the following inline assembly code adds 3 to memory location 0x80:

```
3 0x80 mload add 0x80 mstore
```

Vyper considers the loss of readability to be too high a price to pay for the extra power, and thus does not support inline assembly.

## Function Overloading

Function overloading allows developers to write multiple functions of the same name. Which function is used on a given occasion depends on the types of the arguments supplied. Take the following two functions, for example:

```
function f(uint _in) public pure returns (uint out) {
    out = 1;
}

function f(uint _in, bytes32 _key) public pure returns (uint out) {
    out = 2;
}
```

The first function (named f) accepts an input argument of type uint; the second function (also named f) accepts two arguments, one of type uint and one of type bytes32. Having multiple function definitions with the same name taking different arguments can be confusing, so Vyper does not support function overloading.

## Variable Typecasting

There are two sorts of typecasting: *implicit* and *explicit*

Implicit typecasting is often performed at compile time. For example, if a type conversion is

semantically sound and no information is likely to be lost, the compiler can perform an implicit conversion, such as converting a variable of type uint8 to uint16. The earliest versions of Vyper allowed implicit typecasting of variables, but recent versions do not.

Explicit typecasts can be inserted in Solidity. Unfortunately, they can lead to unexpected behavior. For example, casting a uint32 to the smaller type uint16 simply removes the higher-order bits, as demonstrated here:

```
uint32 a = 0x12345678;
uint16 b = uint16(a);
// Variable b is 0x5678 now
```

Vyper instead has a convert function to perform explicit casts. The convert function (found on line 82 of *convert.py*):

```
def convert(expr, context):
    output_type = expr.args[1].s
    if output_type in conversion_table:
        return conversion_table[output_type](expr, context)
    else:
        raise Exception("Conversion to {} is invalid.".format(output_type))
```

Note the use of conversion_table (found on line 90 of the same file), which looks like this:

```
conversion_table = {
    'int128': to_int128,
    'uint256': to_unint256,
    'decimal': to_decimal,
    'bytes32': to_bytes32,
}
```

When a developer calls convert, it references conversion_table, which ensures that the appropriate conversion is performed. For example, if a developer passes an int128 to the convert function, the to_int128 function on line 26 of the same (*convert.py*) file will be executed. The to_int128 function is as follows:

```
@signature(('int128', 'uint256', 'bytes32', 'bytes'), 'str_literal')
def to_int128(expr, args, kwargs, context):
    in_node = args[0]
    typ, len = get_type(in_node)
    if typ in ('int128', 'uint256', 'bytes32'):
        if in_node.typ.is_literal
            and not SizeLimits.MINNUM <= in_node.value <= SizeLimits.MAXNUM:
            raise InvalidLiteralException(
                "Number out of range: {}".format(in_node.value), expr
            )
        return LLLnode.from_list(
            ['clamp', ['mload', MemoryPositions.MINNUM], in_node,
            ['mload', MemoryPositions.MAXNUM]], typ=BaseType('int128'),
            pos=getpos(expr)
        )
    else:
        return byte_array_to_num(in_node, expr, 'int128')
```

As you can see, the conversion process ensures that no information can be lost; if it could be, an exception is raised. The conversion code prevents truncation as well as other anomalies that would ordinarily be allowed by implicit typecasting.

Choosing explicit over implicit typecasting means that the developer is responsible for performing all casts. While this approach does produce more verbose code, it also improves the safety and auditability of smart contracts.

## Infinite Loop

Although there is no merit because of gaslimit, developers can write an endless loop processing in Solidity. Infinite loop makes it impossible to set an upper bound on gas limits, opening the door for gas limit attacks. Therefore, Vyper doesn't permit you to write the processing and has the following three features:

**The `while` statement**

you can use `while` statement in Solidity, but Vyper doesn't have the statement.

**Deterministic number of iterations of `for` statement**

Vyper has a `for` statement, but the upper limit of the number of iterations must be determinate, and `range ()` can only accept integer literals.

**Recursive calling**

Recursive calling can be written in Solidity, but not in Vyper.

## Preconditions and Postconditions

Vyper handles preconditions, postconditions, and state changes explicitly. While this produces redundant code, it also allows for maximal readability and safety. When writing a smart contract in Vyper, a developer should observe the following three points:

**Condition**

What is the current state/condition of the Ethereum state variables?

**Effects**

What effects will this smart contract code have on the condition of the state variables upon execution? That is, what *will* be affected, and what *will not* be affected? Are these effects congruent with the smart contract's intentions?

**Interaction**

After the first two considerations have been exhaustively dealt with, it is time to run the code. Before deployment, logically step through the code and consider all of the possible permanent outcomes, consequences, and scenarios of executing the code, including interactions with other contracts.

Ideally, each of these points should be carefully considered and then thoroughly documented in the code. Doing so will improve the design of the code, ultimately making it more readable and auditable.

# Decorators

The following decorators may be used at the start of each function:

**@private**

The `@private` decorator makes the function inaccessible from outside the contract.

**@public**

The `@public` decorator makes the function both visible and executable publicly. For example, even the Ethereum wallet will display such functions when viewing the contract.

**@constant**

Functions with the `@constant` decorator are not allowed to change state variables. In fact, the compiler will reject the entire program (with an appropriate error) if the function tries to change a state variable.

**@payable**

Only functions with the `@payable` decorator are allowed to transfer value.

Vyper implements the logic of decorators explicitly. For example, the Vyper compilation process will fail if a function has both a `@payable` decorator and a `@constant` decorator. This makes sense because a function that transfers value has by definition updated the state, so cannot be `@constant`. Each Vyper function must be decorated with either `@public` or `@private` (but not both!).

# Function and Variable Ordering

Each individual Vyper smart contract consists of a single Vyper file only. In other words, all of a given Vyper smart contract's code, including all functions, variables, and so forth, exists in one place. Vyper requires that each smart contract's function and variable declarations are physically written in a particular order. Solidity does not have this requirement at all. Let's take a quick look

at a Solidity example:

```
pragma solidity ^0.4.0;

contract ordering {

    function topFunction()
    external
    returns (bool) {
        initiatizedBelowTopFunction = this.lowerFunction();
        return initiatizedBelowTopFunction;
    }

    bool initiatizedBelowTopFunction;
    bool lowerFunctionVar;

    function lowerFunction()
    external
    returns (bool) {
        lowerFunctionVar = true;
        return lowerFunctionVar;
    }

}
```

In this example, the function called topFunction is calling another function, lowerFunction. topFunction is also assigning a value to a variable called initiatizedBelowTopFunction. As you can see, Solidity does not require these functions and variables to be physically declared before being called upon by the excecuting code. This is valid Solidity code that will compile successfully.

Vyper's ordering requirements are not a new thing; in fact, these ordering requirements have always been present in Python programming. The ordering required by Vyper is straightforward and logical, as illustrated in this next example:

```
# Declare a variable called theBool
theBool: public(bool)

# Declare a function called topFunction
@public
def topFunction() -> bool:
    # Assign a value to the already declared variable called theBool
    self.theBool = True
    return self.theBool

# Declare a function called lowerFunction
@public
def lowerFunction():
    # Call the already declared function called topFunction
    assert self.topFunction()
```

This shows the correct ordering of functions and variables in a Vyper smart contract. Note how the variable theBool and the function topFunction are declared before they are assigned a value and called, respectively. If theBool was declared below topFunction or if topFunction was declared below lowerFunction this contract would not compile.

# Compilation

Vyper has its own online code editor and compiler, which allows you to write and then compile your smart contracts into bytecode, ABI, and LLL using only your web browser. The Vyper online compiler has a variety of prewritten smart contracts for your convenience, including contracts for a simple open auction, safe remote purchases, ERC20 tokens, and more. This tool, offers only one version of the compilation software. It is updated regularly but does not always guarantee the latest version. Etherscan has an online Vyper compiler which allows you to select the compiler version. Also Remix, originally designed for Solidity smart contracts, now has a Vyper plugin available in the settings tab.

**NOTE**

Vyper implements ERC20 as a precompiled contract, allowing these smart contracts to be easily used out of the box. Contracts in Vyper must be declared as global variables. An example for declaring the ERC20 variable is as follows:

```
token: address(ERC20)
```

You can also compile a contract using the command line. Each Vyper contract is saved in a single file with the *.vy* extension. Once installed, you can compile a contract with Vyper by running the following command:

```
vyper ~/hello_world.vy
```

The human-readable ABI description (in JSON format) can then be obtained by running the

following command:

```
vyper -f json ~/hello_world.v.py
```

# Protecting Against Overflow Errors at the Compiler Level

<a id="__indexterm-42" type="indexterm"></a><a id="__indexterm-43" type="indexterm"></a><a id="__indexterm-44" type="indexterm"></a>Overflow errors in software can be catastrophic when dealing with real value. For example, one <a href="http://bit.ly/2yHfvoF">transaction from mid-April 2018</a> shows the malicious transfer of over 57,896,044,618,658,100,000,000,000,000,000,000,000,000, 000,000,000,000,000,000 BEC tokens. This transaction was the result of an integer overflow issue in BeautyChain’s ERC20 token contract (BecToken.sol). Solidity developers do have access to libraries like <a href="http://bit.ly/2ABhb4l">SafeMath</a> as well as Ethereum smart contract security analysis tools like <a href="http://bit.ly/2CQRoGU">Mythril OSS</a>. However, developers are not forced to use the safety tools. Put simply, if safety is not enforced by the language, developers can write unsafe code that will successfully compile and later on "successfully" execute.

Vyper has built-in overflow protection, implemented in a two-pronged approach. Firstly, Vyper provides a SafeMath equivalent that includes the necessary exception cases for integer arithmetic. Secondly, Vyper uses clamps whenever a literal constant is loaded, a value is passed to a function, or a variable is assigned. Clamps are implemented via custom functions in the Low-level Lisp-like Language (LLL) compiler, and cannot be disabled. (The Vyper compiler outputs LLL rather than EVM bytecode; this simplifies the development of Vyper itself.)

# Reading and Writing Data

While it is costly to store, read, and modify data, these storage operations are a necessary component of most smart contracts. Smart contracts can write data to two places:

**Global state**

The state variables in a given smart contract are stored in Ethereum's global state trie; a smart contract can only store, read, and modify data in relation to that particular contract's address (i.e., smart contracts cannot read or write to other smart contracts).

**Logs**

A smart contract can also write to Ethereum's chain data through log events. While Vyper initially employed the `__log__` syntax for declaring these events, an update has been made that brings its event declaration more in line with Solidity's original syntax. For example, Vyper's declaration of an event called `MyLog` was originally `MyLog: __log__({arg1: indexed(bytes[3])})`. The syntax has now become `MyLog: event({arg1: indexed(bytes[3])})`. It is important to note that the execution of the log event in Vyper was, and still is, as follows: `log.MyLog("123")`.

While smart contracts can write to Ethereum's chain data (through log events), they are unable to read the on-chain log events they've created. Notwithstanding, one of the advantages of writing to

Ethereum’s chain data via log events is that logs can be discovered and read, on the public chain, by light clients. For example, the logsBloom value in a mined block can indicate whether or not a log event is present. Once the existence of log events has been established, the log data can be obtained from a given transaction receipt.

# Conclusions

Vyper is a powerful and interesting new contract-oriented programming language. Its design is biased toward "correctness," at the expense of some flexibility. This may allow programmers to write better smart contracts and avoid certain pitfalls that cause serious vulnerabilities to arise. Next, we will look at smart contract security in more detail. Some of the nuances of Vyper design may become more apparent once you read about all the possible security problems that can arise in smart contracts.