PUBLIC

Limited Code Review

of the Vyper Compiler ABI decoder and v0.4.0 pull requests

June 21, 2024

Produced for

by

Contents

1 Executive Summary 3
2 Review Overview 5
3 Limitations and use of report 10
4 Terminology 11
5 Findings 12
6 Informational 18

1 Executive Summary

Dear Vyper team,

Thank you for trusting us to help Vyper with this review. Our executive summary provides an overview of subjects covered in our review of the latest version of Vyper Compiler according to Scope to support you in forming an opinion on their security risks.

Limited code reviews are best-effort checks and don't provide assurance comparable to a non-limited code assessment. This review was not conducted as an exhaustive search for bugs, but rather as a best-effort sanity check for the pull requests of interests. The review was executed by one engineer over two weeks. Given the large scope and codebase and the limited time, the findings aren't exhaustive.

This review concentrated on multiple pull requests of the to-be-released version 0.4.0 of the Vyper compiler. The review focused on the ABI decode routine, recent fixes and new features such as function exports or transient storage integration.

The most critical subjects covered in our review are the functional correctness of the ABI decode routine, invalid memory and storage reads as well as correct handling of function exports. Several issues were found in the ABI decoding routine as shown in the issues ABI-decode incorrect checks for complex types head and ABI-decode incorrect checks for Dynamic array head and fixed in subsequent pull requests. Additionally make_setter overlaps with static call presents an issue with an invalid read due to a read-after-write pattern.

It is important to note that security reviews are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	7

2 Review Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The review was performed on the source code files inside the Vyper Compiler repository:

The review consisted of a review of specific pull requests and was conducted within the available time constraints.

The following pull requests were in scope:

- https://github.com/vyperlang/vyper/pull/3925
- https://github.com/vyperlang/vyper/pull/4060
- https://github.com/vyperlang/vyper/pull/3789
- https://github.com/vyperlang/vyper/pull/4037
- https://github.com/vyperlang/vyper/pull/4059
- https://github.com/vyperlang/vyper/pull/3786
- https://github.com/vyperlang/vyper/pull/3919
- https://github.com/vyperlang/vyper/pull/4015
- https://github.com/vyperlang/vyper/pull/4033
- https://github.com/vyperlang/vyper/pull/4040
- https://github.com/vyperlang/vyper/pull/3949
- https://github.com/vyperlang/vyper/pull/4007
- https://github.com/vyperlang/vyper/pull/4091
- https://github.com/vyperlang/vyper/pull/4144
- https://github.com/vyperlang/vyper/pull/3941

In all pull requests, changes related to Venom IR were not reviewed.

Additionally, both the ABI decoding routine as well the overall transition to transient storage in the whole codebase were reviewed as of `a72488ce68125a65813199f9b1188ce60a987feb`.

This review was not conducted as an exhaustive search for bugs, but rather as a best-effort sanity check. The issues already documented on the Vyper Compiler repository at the time of the review were not included in this report.

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	June 02, 2024	1f6b9433fbd52447d0737fb2eee16f42a01308f4	Initial Version

2.2 System Overview

The Vyper language is a pythonic smart-contract-oriented language, targeting the Ethereum Virtual Machine (EVM). The Vyper compiler translates the Vyper language into the EVM bytecode. The compilation process is performed in multiple phases:

1. Vyper Abstract Syntax Tree (AST) is generated from Vyper source code.
2. Literal nodes in the AST are validated.
3. The semantics of the program are validated and the namespace is populated. The structure and the types of the program are checked and type annotations are added to the AST. Module imports are resolved and several related properties are checked.
4. Positions in storage and code are allocated for storage and immutable variables.
5. The Vyper AST is turned into a lower-level intermediate representation language (IR).
6. Various optimizations are applied to the IR.
7. The IR is turned into EVM assembly.
8. Assembly is turned into bytecode, essentially resolving symbolic locations to concrete values.

We now give a brief overview of the two main components we are interested in; semantic validation and code generation.

2.2.1 Semantic Validation

Once the Abstract Syntax Tree has been generated from the source code, it is analyzed to ensure that it is a valid AST regarding Vyper semantics and annotated with types and various metadata so that the code generation module can properly generate IR nodes from the AST.

The semantics validation starts with the `ModuleAnalyzer` which iterates over the various Module-level statements of the contract. For each statement, after performing various checks, the compiler updates the namespace to add a new entry if needed. For example, for a variable declaration, the namespace will be updated to map the variable's name to some data structure with relevant information such as its type, whether it is public or constant for example. If an `Import` or an `ImportFrom` statement is visited, the imported module's AST is produced and analyzed by the `ModuleAnalyzer`.

The `FunctionAnalyzer` is then used to validate the content of each function one by one. It iterates over all the statements in the body of the function, and, for each statement, validates that it respects Vyper semantics and, if needed, calls functions like `validate_expected_type` or `get_possible_types_from_node` to perform some type-checking. Each statement's sub-expressions are visited by the `ExprVisitor` which annotates the node with its type and performs more semantic validation and analysis on the expression.

Once all module-level statements and functions have been properly analyzed, the compiler adds getters for public variables to the AST and checks some properties related to modules imported, such as ensuring that each initialized module's `__init__` function is called or that modules annotated as used are actually used in the contract.

The `_ExprAnalyser` is used when functions such `validate_expected_type` or `get_possible_types_from_node` are called to infer one or multiple types for a given expression. Such a process can be recursive in the case of complex expressions and mostly acts as a type checker.

The `FunctionAnalyzer` is also in charge of validating several properties of the functions, for example, that its body respects the function's mutability, or that there is eventually a terminus node at the end of the function if it has a return type. Module-level statements are also annotated with their type by the analyzer.

2.2.2 Code Generation

After the Abstract Syntax Tree has been type-checked, storage slots have been assigned to storage variables, and data locations have been assigned to immutable variables, the resulting AST is forwarded to the code generation phase to be turned into an intermediate representation of the code (IR). The intermediate representation is a lower-level description of the same program, where the operations performed are more similar to the EVM primitives. As such, it handles pointers directly, explicitly performs memory and storage stores and loads, and translates every high-level Vyper concept into an EVM-compatible equivalent. It differs from the assembly because it has some high-level convenience functionality, such as performing conditional jumps with the `if` operator, looping with the `repeat` operator, defining and caching stack variables with the `with` operator and setting new values for them with the `set` operator or marking jump locations with the `label` operator. Furthermore, it has some convenience functions such as `sha3_32` and `sha3_64`, which use the keccak hash function to compute the hashes of stack variables and the `deploy` function, which copies the runtime bytecode to memory and returns it at the end of the constructor execution.

The code generation is accessed from the function `vyper.codegen.module.generate_ir_for_module()`, which accepts a `ModuleT` containing the annotated AST as well as some properties such as the list of function definitions of the module or its variable declarations. Code is generated for the runtime (code that will be returned by the smart contract constructor and stored in the smart contract) and for the constructor/deployment. Functions are sorted topologically according to the call graph, code is generated first for functions that don't call other functions, then for functions that depend on those, and so on. The memory allocation strategy for a function is to reserve a memory frame large enough for every callee at the beginning of the function memory frame, and then allocate the variables after the biggest memory offset used by any of the callees. This is possible because there are no dynamic types in Vyper and all variable size is known at compile time. The compiler performs reachability analyses to avoid including unreachable code in the final bytecode.

2.2.2.1 Deployment code and constructor

In the case there is no explicitly defined constructor in the contract, the deployment code is rather straightforward, it simply copies the runtime bytecode from the the deployment code itself to the memory before returning both the offset in memory where the runtime bytecode starts and its length. This operation is abstracted away by the IR using the `deploy` pseudo-opcode.

If there is a constructor defined, the compiler assumes that the compiler arguments, if there are any, will be appended to the deployment bytecode. At the IR level, the `dload` pseudo opcode can be used to read such arguments. The immutables assigned in the constructor, if any, must be returned by the deployment bytecode together with the runtime bytecode to be accessible at runtime. To append them at the end of the runtime bytecode, the IR offers the `istore/iload` abstractions which essentially perform `mstore/mload` at the index corresponding to the end of the buffer for the runtime bytecode in the memory. The runtime bytecode can access them at the IR level with the pseudo-opcode `dload`, which, similarly to its use in the deployment context with constructor arguments, performs `codecopy` from the bytecode's immutable section (at the very end of the bytecode) to the memory. Once the logic of the constructor has been executed, the runtime bytecode concatenated with the immutables is returned using the `deploy` pseudo-opcode.

2.2.2.2 External function and entry points

In the following, an entry point can be seen as an external function if the function has no default arguments. If the function has some, there exists one entry point for each combination of calldata arguments/default overridden argument that is possible. In other words, A function with `D` default argument will have `D+1` entry points. An external function with keyword arguments will generate several entry points, each setting the default values for keyword arguments, and then calling a common function body.

The structure of the runtime bytecode depends on the optimization that has been specified when compiling the contract.

2.2.2.3 Linear selector section

When no optimization is chosen (`optimize=none`), the compiler will generate a linear function selector as it used to do in its previous versions. This selector section acts as follows:

- If the `calldatasize` is smaller than 4 bytes, the execution goes to the fallback.
- For each entry point `F` defined in the contract, the bytecode checks whether the given method id `m` in the calldata matches the method id of `F`.
  - If it does, several checks are performed, such as ensuring `callvalue` is null if the function is non-payable or making sure that `calldatasize` is large enough for `F`. If all the checks pass, the body of the entry-point is executed, otherwise, the execution reverts.
  - If it does not, the iteration goes to the next entry point and, if it was the last one, to the fallback.

2.2.2.4 Sparse selector section

When the gas optimization is set (`optimize=gas`), at compile time, the entry points are sorted by buckets. The amount of bucket is roughly equal to the amount of entry points but can differ a bit as the compiler tries to minimize the maximum bucket size. An entry point with a method id `m` belongs to the bucket `i = m % n` where `n` is the number of buckets. Once all the buckets are generated, a special data section is appended to the runtime bytecode, it contains as many rows as there are buckets, all rows have the same size and row `k` contains the code location of the handler for the bucket `k`. In the case that the bucket was to be empty, its corresponding location is the fallback function.

When the contract is called, the method id `m` is extracted from the calldata. Given the code location of the data section `d`, the size of its rows (2 bytes) and the bucket to look for (`i = m % n`), the location of the handler for the bucket `i` is stored at location `d + i * 2`. The execution can then jump to the bucket handler location. Very similarly to the non-optimized selector table described above, each bucket handler essentially is an iteration over the entry points it contains, trying to match the method ID. If one of them matches, some `calldatasize` and `callvalue` checks are performed before executing the entry point's body. In the case none of the bucket's entry points match `m`, the execution goes to the `fallback` section.

2.2.2.5 Dense selector section

If the code size is optimized (`optimize=codesize`), the selector section is organized around a two-step lookup. First, very similarly to the sparse selector section, the method ID can be mapped to some bucket ID `i` which can be used as an index of the `Bucket Headers` data section to read `i`'s' metadata. For a given bucket `b` with id `i` of size `n`, the row `i` of the `Bucket Headers` data section contains `b`'s magic number, `b`'s' data section's location as well as `n`. The bucket magic `bm` is a number that has been computed at compile time such that `(bm * m) >> 24 % n` is different for every method ID `m` of entry-point contained in `b`. Having this unique identifier for methods belonging to `b` means that we can index another data section, specific to `b`. For each entry-point `m` of `b`, this data section contains `m` (its method ID), the location of the entry point's handler, the minimum `calldatasize` it requires accepts and whether or not the entry point is non-payable.

Given all those meta-data, the necessary checks can be performed and the execution can jump to the entry point's body code.

2.2.2.6 Internal and external functions arguments and return values

Function arguments for internal functions are allocated as memory variables at the beginning of the function memory frame. The caller will set their value, accessing the callee memory frame. For external functions, calldata is copied to memory if clamping is needed or if the internal Vyper representation is different than the ABI encoding. Clamping is necessary for types that could exceed their allowed range, such as `uint128`, and ABI encoding and Vyper memory representation differ for dynamic types, for which the ABI encoding includes relative pointers. Return values for internal functions are copied to a buffer allocated in the caller function memory frame. The caller passes on the stack the address of the return buffer to the callee function. The return program counter, for internal functions, is also pushed on the stack by the caller.

2.2.2.7 Function body IR generation

Code for the function body is then generated by calling `vyper.codegen.stmt.parse_body()`. It generates the codes for every statement in a function. Sub-expressions in every statement are recursively parsed. For every type of AST node representing a statement, a function `parse_{NodeType}` is present in `stmt.py`, which generates the IR for a node of type `NodeType` (e.g. `parse_Return`, `parse_Assign`). Expressions contained in statements are recursively parsed in the `vyper.codegen.expr` module. Code is generated for the innermost expressions, and the output of the generated code is used to evaluate the containing expressions.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe our findings. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	7

- ABI-decode Incorrect Checks for Complex Types Head
- ABI-decode Incorrect Checks for Dynamic Array Head
- ABI-decode min_size Check Too Strict
- Make_Setter Overlaps With Static Call
- Misleading Error Message
- AttributeError When Wrongly Exporting a Variable From the Current Module
- CodegenPanic When Assigning to a Tuple of Tuple

5.1 ABI-decode Incorrect Checks for Complex Types Head

Correctness Low Version 1

CS-VYPER_JUNE_2024-001

In `make_setter(left, right, hi)`, When `right` is ABI encoded and in memory (there is a dirty read risk), in the case that value of type `T` to be copied and decoded is either a static array, a tuple or a struct, `_complex_make_setter()` is called and no specific check is performed to ensure that the words to be copied are within bounds of the source buffer.

In the case, `T` is static according to the ABI specifications (no sub-element is itself dynamic), the lack of check is not an issue given that either:

- The call to `make_setter()` was done from `abi_decode()`, `external_function.py` or `external_call.py`: a check ensuring that the buffer is larger than `T.min_size()` was performed there.
- The call to `make_setter()` was done as part of a recursion of `make_setter()`, the size of the buffer was checked by the caller or one of its callers.

However, in the case the type to be copied is dynamic according to the ABI specifications, the lack of check is problematic given that it could be that the heads to be read are out of bounds of the source buffer.

The following contract and script illustrate the issue.

```
@nonpayable
@external
def foo(x:Bytes[320]):
    x_mem: Bytes[320] = x
    y:DynArray[uint256, 2][2] = _abi_decode(x_mem,DynArray[uint256, 2][2])

@nonpayable
@external
def bar(x:Bytes[320]):
    if True:
        a: Bytes[160] = b''
        # make the word following the x_mem payload dirty to make a potential OOB revert
        fake_head: uint256 = 32
    x_mem: Bytes[320] = x
    y:DynArray[uint256, 2][2] = _abi_decode(x_mem,DynArray[uint256, 2][2])
```

```
import boa
foo = boa.load("foo.vy")

def abi_payload_from_tuple(payload: tuple[int | bytes, ...]) -> bytes:
    return b"".join(
        p.to_bytes(32, "big") if isinstance(p, int) else p for p in payload
    )

encoded = (
    0x80,  # head of the static array
    0x00,  # garbage to pass the ABI min_size check
    0x00,  # garbage to pass the ABI min_size check
    0x00,  # garbage to pass the ABI min_size check
)
inner = (
    0x00,  # head of the first dynarray
    # 0x00,  # head of the second dynarray
)

encoded = abi_payload_from_tuple(encoded + inner)
foo.foo(encoded) # succeed (y == [[],[]])
foo.bar(encoded) # revert
```

5.2 ABI-decode Incorrect Checks for Dynamic Array Head

Correctness **Low** **Version 1**

CS-VYPER_JUNE_2024-002

In `make_setter(left, right, hi)`, When `right` is ABI encoded and in memory (there is a dirty read risk), in the case that the type to be copied and decoded is a Dynamic array, `clamp_dyn_array()` is called to ensure that the given data to be copied is within the bounds of the source buffer. However, in the following check:

```
["assert", ["le", item_end, hi]]
```

`item_end` is computed as:

```
["add", OFFSET, ["mul", get_dyn_array_count(ir_node), ir_node.typ.value_type.abi_type.static_size()]]
```

In the case the value type of the array is a byte array or a dynamic array, one would expect `item_end` to be equal to `32 + get_dyn_array_count(ir_node) * 32` since there should be a word for the length and a 32-byte head for each sub-element. However, for the two types, `T.static_size()` is equal to `0`, which means that the check is too weak and it could be that the head of the inner array is read out of bounds if the size of the payload is not large enough. It should be noted that such out-of-bound reads could access some dirty memory and treat it as the head of the inner array.

The following contract and script illustrate the issue.

```
@nonpayable
@external
def foo(x:Bytes[320]):
    if True:
        a: Bytes[320-32] = b''
        # make the word following the x_mem payload dirty to make a potential OOB revert
        fake_head: uint256 = 32
    x_mem: Bytes[320] = x
    y: DynArray[DynArray[uint256, 2], 2] = _abi_decode(x_mem,DynArray[DynArray[uint256, 2], 2])

@nonpayable
@external
def bar(x:Bytes[320]):
    x_mem: Bytes[320] = x
    y:DynArray[DynArray[uint256, 2], 2] = _abi_decode(x_mem,DynArray[DynArray[uint256, 2], 2])
```

```
import boa
foo = boa.load("foo.vy")

def abi_payload_from_tuple(payload: tuple[int | bytes, ...]) -> bytes:
    return b"".join(
        p.to_bytes(32, "big") if isinstance(p, int) else p for p in payload
    )

encoded = (
    0xE0,  # head of the dynarray
    0x00,  # 0x20
    0x00,  # 0x40
    0x00,  # 0x60
    0x00,  # 0x80
    0x00,  # 0xA0
    0x00,  # 0xC0
    0x02,  # len of outer
)
inner = (
    0x0,  # head
    # 0x0,  # head2
)

encoded = abi_payload_from_tuple(encoded + inner)

print(foo.foo(encoded)) # revert
print(foo.bar(encoded)) # succeed (y == [[],[]])
```

5.3 ABI-decode `min_size` Check Too Strict

Design | Low | Version 1

CS-VYPER_JUNE_2024-003

The ABI-decode routine is used in the following location in the codebase:

- The `_abi_decode` builtin.
- Function arguments decoding in `external_function.py`.
- Return value decoding in `external_call.py`.

In all three cases, the to-be-decoded payload is checked against a minimum size `Type.min_size()`. This check is too strict, as it does not take into account payloads that could have been encoded not using the strict encoding algorithm but would still be valid according to the ABI specifications.

The following Payload would not be decoded to `(DynArray[DynArray[uint256,2], 1],)` given that the type's `min_size` is greater than the actual payload size although it is a valid encoding for `([[],[]],)`

```
0x00: 0x20 # head of the outer dynamic array
0x20: 0x02 # length of the outer dynamic array
0x40: 0x00 # head of the 1st inner dynamic array, length of both inner dynamic arrays.
0x60: 0x00 # head of the 2nd inner dynamic array.
```

5.4 Make_Setter Overlaps With Static Call

Correctness | Low | Version 1

CS-VYPER_JUNE_2024-004

In the past, `make_setter` has been having issues with dependencies between the left-hand side and right-hand side as shown in issues #2418, #3503 and #4056. Although PR 3410, PR 4037 and PR 4059 provided fixes, the issue persists when the right-hand side reads from the left-hand side with a static call.

```
interface A:
    def boo() -> uint256 : view
interface B:
    def boo() -> uint256 : nonpayable

a: DynArray[uint256, 10]

@external
def foo() -> DynArray[uint256, 10]:
    self.a = [0,0,0]
    self.a = [1, 2, staticcall A(self).boo(), 4]
    return self.a # returns [1, 2, 1, 4]

@external
def bar() -> DynArray[uint256, 10]:
    self.a = [0,0,0]
    self.a = [1, 2, extcall B(self).boo(), 4]
    return self.a # returns [1, 2, 0, 4]

@external
@view
# @nonpayable
```

```
def boo() -> uint256:
    return self.a[0]
```

5.5 Misleading Error Message

Design Low Version 1

CS-VYPER_JUNE_2024-005

The displayed error when a module is assigned to itself is misleading.

```
import lib1

@external
@nonreentrant
def foo():
    lib1 = lib1
```

```
vyper.exceptions.ImmutableViolation: Environment variable cannot be written to
```

5.6 `AttributeError` When Wrongly Exporting a Variable From the Current Module

Design Low Version 1

CS-VYPER_JUNE_2024-006

When exporting a public variable defined in the current module, the compiler raises an `AttributeError` instead of a more meaningful error message.

```
exports: self.y

y:public(uint256)
```

```
vyper.exceptions.CompilerPanic: unhandled exception 'NoneType' object has no attribute '_metadata'
```

5.7 `CodegenPanic` When Assigning to a Tuple of Tuple

Correctness Low Version 1

CS-VYPER_JUNE_2024-007

When having an assignment to a tuple containing a tuple, the compiler panics.

```
@internal
def bar() -> (uint256, (uint256, uint256)):
    return (1,(2,3))
```

```
@external
@nonreentrant
def foo():
    a:uint256 = 0
    c:uint256 = 0
    d:uint256 = 0
    (a,(c,d)) = self.bar()
```

```
vyper.exceptions.CodegenPanic: unhandled exception , parse_Assign
```

6 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

6.1 Misleading Function Names, Types and Documentation

Informational **Version 1**

CS-VYPER_JUNE_2024-008

- The function `build_layout_output()` is commented with the following although it now returns non-storage layouts:

```
# in the future this might return (non-storage) layout,
# for now only storage layout is returned.
```

- The function name `CompilerData.storage_layout()` is misleading given that it returns non-storage layout.
- Both `build_layout_output()` and `CompilerData.storage_layout()` are marked to return `StorageLayout` although they also return non-storage layouts.
- The following comment in `_set_nonreentrant_keys()` is outdated:

```
# a nonreentrant key can appear many times in a module but it
# only takes one slot. after the first time we see it, do not
# increment the storage slot.
```