Vyper 不可重入锁漏洞技术事后分析报告

Vyperlang 团队特别感谢 Omniscia 团队[1]

2023 年 7 月 30 日，由于 Vyper 编译器（特别是版本、和 ）中的潜在漏洞，多个Curve.Fi流动性池被利用。虽然该版本已识别并修补了错误，但当时并未意识到对使用易受攻击的编译器的协议的影响，也没有明确通知他们。该漏洞本身是一个未正确实施的重入防护，在某些情况下可能会被绕过，我们将在本报告中深入研究这一点。`0.2.15 0.2.16 0.3.0 v0.3.1`

vyper影响

虽然黑客本身已经在其他事后分析中得到了充分的报道，包括 Curve.Fi 的官方事后分析，但我们希望深入研究 Vyper 编译器本身到底出了什么问题，以及为什么该漏洞很难被发现，以及整个生态系统可以从这些事件中学到什么。

如果您熟悉区块链领域以及 Vyper 存在的原因，**我们建议您跳过背景**部分，因为它包含您很可能知道的非常基本的信息。

背景

维珀

Vyper 是一种面向合约、特定领域的 Python 编程语言，针对以太坊虚拟机 (EVM)。其目标包括简单性、Python性、安全性和可审计性。

EVM：单线程非并发机器

*EVM 上部署的代码所关心的一个常见问题是可重入*的概念。与传统程序相比，"区块链程序"的控制流将让位于任何给定时刻正在执行的"活动"程序。从此，"区块链程序"将被称为合约。

详细来说，我们可以将所有区块链程序视为在单线程上运行，不支持并发。每当一个程序调用另一个程序时，整个控制流都会传递给被调用的程序。

重入：一个广泛存在的 Web 3.0 问题

这意味着原始调用者的执行基本上被及时冻结，直到被调用的程序结束为止，此时调用者将在他们离开的确切位置恢复。虽然这种行为可能会产生不同类型的漏洞，但最著名的一个是是一个*重入*。

由于控制流被放弃给被调用的合约，被调用的合约可以在冻结时*重新进入原始调用者*。容易受到此类攻击的合约将在外部合约调用后包含状态更新，这意味着当它们被冻结时，**它们的状态已过时且不正确**。

解决方案

该生态系统提出了两种方法来对抗重入攻击，并从本质上使其无效；检查-效果-交互模式 (CEI) 和重入防护。

检查-效果-交互 (CEI) 模式

CEI 模式是一种编程方法，它规定函数的代码应首先执行其安全检查，然后在其存储中执行任何影响，最后在函数结束时执行与外部合约的交互。

如果严格遵循这种模式，"交互"（即控制流放弃）期间合约的状态将是最新且正确的，无论合约如何重新输入，任何可能的利用都是不可能的。

重入防护

在大多数情况下，CEI 模式就足够了，但是，DeFi 生态系统是多方面的，函数通常依赖于**外部调用的结果来继续执行自己的操作**。在这种情况下，CEI 模式不适用，**必须设置重入防护**。

由于 Vyper 语言的核心原则之一是*安全性*，Vyper 决定通过特殊函数装饰器直接在语言级别引入可重入防护 `@nonreentrant`。`v0.1.0-beta.9` 自Vyper 最早版本之一发布以来，可重入防护一直是该语言的核心功能。

从本质上讲，两种实现的功能相同；它们设置两种状态（激活、非激活）之间的存储值。`@nonreentrant` 当调用标记为的函数时，该标志为：

- 确保不处于活动状态
- 设置为活动状态

简单来说在代码中的用法是这样的
@external
@nonreentrant("lock")
def make_a_call(_addr: address):
    pass

一旦函数调用结束，标志为：

- 设置为非活动状态

通过这种机制，`@nonreentrant` 用户可以确保函数只有**在结束后**才能重新调用，这意味着**无论执行什么外部调用都不会发生重入**。存在更复杂形式的重入攻击（即 `view` 仅重入、跨合约重入），但就该漏洞而言，基本情况才是重要的。

Vyper 漏洞历史时间表

`@nonreentrant`：基于标签的重入防护

自从引入以来，`@nonreentrant` 装饰器始终支持 `<key>` 设置，与仅在合约级别全局应用的不可重入锁相比，它提供了更大的灵活性。

一个简单的实现可能是使用 `mapping` 并 `key` 在其上设置相关的重入标志，但是，由于查找的 `keccak256` Gas 成本，这种方法会产生额外的成本 `mapping`。

由于 Vyper 是一种不向用户提供原始存储访问的语言，因此在编译合约时它将完全了解合约使用的所有存储槽。因此，它承担分配存储槽的工作，其中包括确保存储变量和可重入键锁的槽不会彼此重叠。

PR#1264在 Vyper 版本中引入了此功能，`v0.1.0-beta.9` 使用了一种简单的方法来确保不重叠，将重入标志存储在距离合约原始插槽的特定偏移处（`0xFFFFFF` 准确地说）。

重构编译器

随着新功能的开发，从 2018 年开始，Vyper 编译器开始了多年 的 努力，将当时的单通道架构重构为多通道架构，该架构将类型检查和语义分析的关注点分离到前端，与代码生成后端不同。与大多数大型重构项目一样，这项工作是渐进且零碎的，与其他错误修复和功能开发一起进行，直到最终在 2023 年以PR#3390达到顶峰。

位置优化：存储槽位更智能分配

PR#2308是 Vyper `v0.2.9` 版本的一部分，旨在通过在处理合约的常规存储变量的所有插槽后利用第一个可用的未分配存储插槽，而不是从常量开始，使存储分配更加智能 `0xFFFFFF`。这将节省字节码空间，因为在字节码中，`PUSH` 通过将不可重入键的位置推送到堆栈而在存储槽的任何加载或存储之前的指令可以使用更少的字节。

避免腐败：正确的抵消计算

上述版本的 PR `v0.2.9` 运行良好，并且只要在存储布局的（物理）前端顺序分配的变量不跨越多个顺序槽，就可以保证可重入标志槽和存储槽之间不会重叠。

由于 Vyper 语言和代码库当时正在进行重大重构，PR#2361（版本的一部分 `v0.2.13`）引入了一种更有效的方法来存储可以在合约中跨越 1 个以上存储槽（32 字节）的变量。作为更大的重构工作的一部分，它还将常规存储变量的槽计算从现有代码生成传递移至新的前端传递，但保留了可重入键的槽计算。由于可重入键的槽计算取决于常规存储变量的分配结果，因此最终为前端和代码生成过程之间的常规存储变量分配保留了两种不同的分配器实现。这导致了PR#2308的偏移计算不正确并需要更新。

该更新是在PR#2379（发行版的一部分）中引入的 `v0.2.14`，旨在通过考虑存储中声明的变量的正确大小来正确计算可重入标志的存储偏移量，而不是假设所有变量都占用一个单独的存储空间。`slot`（这在早期的实现中是正确的）。然而，第二次更新仍然存在一个错误，该错误源于前端和 codegen 分配器实现之间的差异，我们将在下面描述。

由于这些错误，这两个 `v0.2.13` 版本在发布后不久 `v0.2.14` 就被"猛拉"[2]。

决定性事件：重新进入警卫腐败 `v0.2.14`

~~`v0.2.14` 发布后不久，一名 Vyper 用户在 Vyper GitHub 存储库中打开了~~问题 #2393 `0.2.14`，表示当他们将 YearnVault 代码升级到.

当用户打开问题时，拍摄Yearn 最新可用版本的快照，继续使用 进行编译 `v0.2.14`，并使用EtherVM 反编译器检查**反编译的字节码**，将发现伪代码存储偏移量被用作"标志"文件的和实例中的关键字。`storage[0x2e] @nonreentrant("withdraw") def deposit def withdraw Vault.vy`

然而，相同的存储偏移量用于合约级 `managementFee` 变量。`managementFee()` 这可以通过评估getter 函数以及setter 函数的反编译函数来验证 `setManagementFee`，后者将重用相同的存储偏移量。

编译相同的代码库 `v0.2.13` 表明，可重入防护按预期工作，并且在存储中没有重叠。然而，PR#2379 的 `v0.2.14` 发布并没有完全解决腐败问题。

`v0.2.14` 为装饰器分配存储槽的发行版代码 `@nonreentrant` 仍然会在新的前端代码和现有的 codegen 分配器之间产生不正确的交互。由于前端和代码生成分配器之间映射类型的分配策略不同，可重入槽最终仍然与常规存储变量重叠。**数据损坏代码** `v0.2.14` **如下**：

```
def get_nonrentrant_counter(self, key):
    """
    Nonrentrant locks use a prefix with a counter to minimise deployment cost of a contract
    We're able to set the initial re-entrant counter using the sum of the sizes
    of all the storage slots because all storage slots are allocated while parsing
    the module-scope, and re-entrancy locks aren't allocated until later when parsing
    individual function scopes. This relies on the deprecated _globals attribute
    because the new way of doing things (set_data_positions) doesn't expose the
    next unallocated storage location.
    """
    if key in self._nonrentrant_keys:
        return self._nonrentrant_keys[key]
    else:
        counter = (
            sum(v.size for v in self._globals.values() if not isinstance(v.typ, MappingType
            + self._nonrentrant_counter
        )
        self._nonrentrant_keys[key] = counter
        self._nonrentrant_counter += 1
        return counter
```

获取重入锁

如果这个重入锁已经分配过了

没有分配过的重入锁就需要重新计算全局slot位置

将其与当时计算常规存储变量的存储布局的前端代码进行比较。

```
available_slot = 0
for node in vyper_module.get_children(vy_ast.AnnAssign):
    type_ = node.target._metadata["type"]
    type_.set_position(StorageSlot(available_slot))
    available_slot += math.ceil(type_.size_in_bytes / 32)
```

虽然此代码可以正确使用该值并为同一值 `key` 生成相同的存储偏移量，但它错误地计算了存储偏移量。`@nonreentrant key`

具体来说，旧的分配器没有为 `MappingType` 条目（即 `HashMap`）分配存储槽，而新的分配器则分配了。`MappingType` 存储条目永远不会被写入，但会被保留，无论编译器如何（参考：Issue 2436）。这导致不可重入密钥分配器和前端分配器之间不一致，从而导致报告的存储损坏。

引入漏洞：重入锁故障 `v0.2.15`

在 `v0.2.14` 被撤下后，为了纠正该 `v0.2.14` 版本对重入防护的损坏，该版本中包含的 PR#2391 `v0.2.15` 旨在通过将重入密钥移至物理分配来修复前面提到的PR#2379引入的错误在常规存储变量前面。此外，为了减少再次引入此类问题的机会，它通过将逻辑移动到前端中与常规存储变量相同的函数中，完成了从代码生成过程中删除存储槽分配逻辑的操作分配。然而，这样做时，它删除了旧的 `self._nonreentrant_keys` 数据结构，并且最重要的是，删除了确保每个不可重入键仅分配一个锁的随附逻辑：

```
if key in self._nonrentrant_keys:
    # --> SAFE. only allocate one slot per key <--
    return self._nonrentrant_keys[key]
```

实际的漏洞是在以下代码中引入的 `v0.2.15`：

```
# Allocate storage slots from 0
# note storage is word-addressable, not byte-addressable
storage_slot = 0

for node in vyper_module.get_children(vy_ast.FunctionDef):
    type_ = node._metadata["type"]
    if type_.nonreentrant is not None:
        # --> BUG! should check nonreentrant key not already allocated <--
        type_.set_reentrancy_key_position(StorageSlot(storage_slot))
        # TODO use one byte - or bit - per reentrancy key
        # requires either an extra SLOAD or caching the value of the
        # location in memory at entrance
        storage_slot += 1
```

该漏洞源于 `storage_slot` 可重入键的偏移量如何忽略装饰器 `<key>` 的实际情况，并且无论使用什么"键"，`@nonreentrant(<key>)` 都只是为每个看到的装饰器保留一个新插槽。`@nonreentrant`

潜伏期：`v0.2.15`、`v0.2.16` 和 `v0.3.0`

由于当时 Vyper 代码库中的测试不足（2021 年 7 月 21 日至 2021 年 11 月 30 日之间的 4 个月期间），该漏洞在 `v0.2.15` 临时版本期间未被检测到。`v0.2.16 v0.3.0`

所有使用版本编译的 Vyper 合约 `v0.2.15` **都** `v0.2.16` **容易** `v0.3.0` **受到重入防护故障的影响。**

补救措施：`v0.3.1` 发布

该 `v0.3.1` 版本通过调整编译器为合约中的每个变量分配数据槽的方式解决了此漏洞。**该漏洞已在两个不同的 PR**中修复。

PR#2439：修复未使用的存储插槽

第一个部分修复该漏洞的 PR 是PR#2439，其中包含以下描述：

> 这不是语义错误，而是优化错误，因为我们分配的
> 插槽比实际需要的多，导致插槽
> 分配器中出现"漏洞"——已分配但未使用的插槽。

这个描述实际上并没有清楚地描述这个问题。"漏洞"的描述来自于检查编译输出如何为每个可重入键 `layout` 生成单个值。`slot` 为了更好地理解发生了什么，让我们看一下中的数据分配函数 `v0.3.0`：

```
for node in vyper_module.get_children(vy_ast.FunctionDef):
    type_ = node._metadata["type"]
    if type_.nonreentrant is not None:
        type_.set_reentrancy_key_position(StorageSlot(storage_slot))

        # TODO this could have better typing but leave it untyped until
        # we nail down the format better
        variable_name = f"nonreentrant.{type_.nonreentrant}"
        ret[variable_name] = {
            "type": "nonreentrant lock",
            "location": "storage",
            "slot": storage_slot,
        }

        # TODO use one byte - or bit - per reentrancy key
        # requires either an extra SLOAD or caching the value of the
        # location in memory at entrance
        storage_slot += 1
```

`type_` 此代码的问题在于，它将每个（即单个键）的可重入键位置设置 `@nonreentrant` 为 的最新值 `storage_slot`，并在每次迭代时递增。这意味着 的唯一实例 `@nonreentrant(<key>)` 都使用不同的 `storage_slot` 值，但是，每次迭代时 `ret` 的条目都会被覆盖。`variable_name`

因此，`layout` 编译器的输出包含单个条目和单个存储偏移量，这意味着根据PR 的原始原理，`nonreentrant.<key>` 检查编译器的输出似乎只是"跳过"连续声明的存储槽。`@nonreentrant(<key>)`

版本中部分修补该漏洞的非漏洞代码 `v0.3.1` 如下：

```
for node in vyper_module.get_children(vy_ast.FunctionDef):
    type_ = node._metadata["type"]
    if type_.nonreentrant is None:
        continue

    variable_name = f"nonreentrant.{type_.nonreentrant}"

    # a nonreentrant key can appear many times in a module but it
    # only takes one slot. ignore it after the first time we see it.
    if variable_name in ret:
        continue

    type_.set_reentrancy_key_position(StorageSlot(storage_slot))

    # TODO this could have better typing but leave it untyped until
    # we nail down the format better
    ret[variable_name] = {
        "type": "nonreentrant lock",
        "location": "storage",
        "slot": storage_slot,
    }

    # TODO use one byte - or bit - per reentrancy key
    # requires either an extra SLOAD or caching the value of the
    # location in memory at entrance
    storage_slot += 1
```

现在，代码 `storage_slot` 第一次识别到重复的可重入密钥时将正确分配单个密钥。但是，它不会在具有相同偏移量的 `set_reentrancy_key_position` 每个条目上调用该函数，这意味着第一个条目之外的任何条目都将具有"未定义"的存储偏移量可供使用。`type_ @nonreentrant(<key>)`

当尝试使用装饰器编译合约时，这会导致编译器恐慌[3] `@nonreentrant`。为了纠正这个问题，需要进行进一步的更改，以确保所有 `@nonreentrant` 装饰器都正确了解他们需要操作的存储槽。

PR#2514：修复不可重入键的代码生成失败

最终完成 `@nonreentrant` 漏洞缓解的 PR 是PR#2514。具体来说，它扩展了上述代码段，以确保 `set_reentrancy_key_position` 使用为给定 `@nonreentrant` 锁分配的正确槽正确调用该函数。

Vyper版本最终的无漏洞代码 `v0.3.1` **如下**：

```
for node in vyper_module.get_children(vy_ast.FunctionDef):
    type_ = node._metadata["type"]
    if type_.nonreentrant is None:
        continue
    variable_name = f"nonreentrant.{type_.nonreentrant}"

    # 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.
    if variable_name in ret:
        _slot = ret[variable_name]["slot"]
        type_.set_reentrancy_key_position(StorageSlot(_slot))
        continue

    type_.set_reentrancy_key_position(StorageSlot(storage_slot))
    # TODO this could have better typing but leave it untyped until
    # we nail down the format better
    ret[variable_name] = {
        "type": "nonreentrant lock",
        "location": "storage",
        "slot": storage_slot,
    }
    # TODO use one byte - or bit - per reentrancy key
    # requires either an extra SLOAD or caching the value of the
    # location in memory at entrance
    storage_slot += 1
```

正如我们在上面的片段中可以观察到的，现在可以为每个条目 `set_reentrancy_key_position` 正确调用，只要在装饰器中指定了相同的条目，该条目就会正确地使用相同的条目。`type_ nonreentrant storage_slot key @nonreentrant(<key>)`

此外，除了上述修复之外，PR 还包括Vyper 存储库中**急需且缺失的测试；评估跨功能重入性的**专用单元测试：

```
@external
@nonreentrant('protect_special_value')
def protected_function(val: String[100], do_callback: bool) -> uint256:
    self.special_value = val

    if do_callback:
        self.callback.updated_protected()
        return 1
    else:
        return 2

@external
@nonreentrant('protect_special_value')
def protected_function2(val: String[100], do_callback: bool) -> uint256:
    self.special_value = val
    if do_callback:
        # call other function with same nonreentrancy key
        # --> (revert expected here) <--
        Self(self).protected_function(val, False)
        return 1
    return 2
```

然而，虽然该错误在编译器代码库中被识别、修复和测试，但当时并没有意识到对生产合同的影响，并且可能使用相关编译器版本的协议也没有被明确通知。

`<key>` 可以跨装饰器重用的唯一值的概念 `@nonreentrant` 仅出于单一目的而存在；跨功能重入保护。事实上，Vyper 存储库在其 `0.3.1` 发布之前就缺少这样的测试，这是导致该漏洞首先被引入并一直未被检测到的一个因素。

漏洞总结

- 受影响的版本：`v0.2.15`，`v0.2.16`，`v0.3.0`
- 根本原因：对重入防护数据损坏问题的补救措施不当 `v0.2.13`
- 漏洞简介：`@nonreentrant` Vyper 合约中的所有装饰器都将使用**唯一的存储偏移量，无论其 是多少** `key`，这意味着在使用易受影响的版本编译的所有合约上都可以进行跨功能重入

有利可图的漏洞的条件

虽然漏洞本身很容易识别，并且已在各种实时合约中观察到，但其盈利能力来自于需要满足的一组非常具体的条件。具体来说：

- `.vy` 使用以下任一版本编译的合约 `vyper`：`0.2.15`、`0.2.16`、`0.3.0`
- `@nonreentrant` 使用具有特定装饰器且不 `key` 严格遵循 CEI 模式的主要函数（即在存储更新之前包含对不受信任方的外部调用）
- 使用相同功能的次要功能 `key` 会受到主要功能引起的不正确状态的影响

不幸的是，这些条件正是 Curve.Fi 流动性池中出现的情况，它们被利用是因为它们需要执行本机的外部分发 `ETH`（在 EVM 上，只能通过执行上下文传输 CALL 来完成[4]）在敏感存储更新之前在函数中执行，否则这些函数将受到正常运行的 `@nonreentrant` 防护的保护。

总结和要点

对于任何大型生产软件项目来说，错误都是一个不幸且严峻的现实。我们能做的就是最大程度地尝试减少错误及其相关风险。

我们可以采取几个实际步骤来提高使用 Vyper 编译的智能合约的正确性：

1. 改进了编译器的测试，包括继续提高覆盖率、将编译器输出与语言规范进行比较，以及利用形式验证 (FV) 工具进行编译器字节码验证。
2. 为开发人员提供工具，使他们能够更轻松地采用多方面的方法来测试他们的代码，包括源代码和字节码级别的测试。
3. 使用 Vyper 协议实现更严格的双向反馈

但仅仅关注最新版本编译器的正确性是不够的；由于智能合约的不可变性，使用过去版本的 Vyper 编写的合约可能会获得大量资金。

因此，确保 Vyper 过去版本的安全**是我们未来将投入大量资源的另一个重要的新焦点轴**。它与引入新功能、为最新版本提供错误修复和重构一样重要。

最终，**我们希望从最近发生的事件中汲取教训，确保 Vyper 成为世界上最坚如磐石、最安全的智能合约语言和编译器项目**。因此，这些目标将得到我们团队内外各种新的安全相关举措的支持，包括：

- 与Codehawks合作**进行短期竞争性审核**，重点关注最新版本的 Vyper
- 与Immunefi合作，涵盖所有版本的 Vyper 编译器的**短期和长期（开放式）错误赏金计划**
- **Vyper 安全联盟**，一个协调的多协议赏金计划，用于帮助发现当前和过去影响 Vyper 版本保护实时 TVL 的编译器错误
- 与ChainSecurity、OtterSec、Statemind和Certora等**多家审计公司合作**，审查 Vyper 的过去版本，确保大量实时 TVL，并帮助持续审查编译器的未来
- **扩大团队**；包括专门的安全工程角色，旨在全面改进 Vyper 的安全工具，包括内部和面向用户的安全工具
- 与Solidity 提供的**现有安全工具包的协作将使 Vyper 生态系统受益匪浅**
- **设计语言规范**，这将有助于正式验证并帮助测试编译器本身的工作

我们希望很快看到您使用 Vyper :)。请继续关注未来几周有关这些举措的更多公告！要了解进一步的公告，请关注Vyper 官方推特。为了帮助贡献，请参阅Vyper Github！如果您对 Vyper 感到兴奋并希望在资金方面提供帮助 – 或者只是想聊天 – 请通过Vyper Discord与我们联系，我们将随时欢迎您加入社区。

---

1. 特别感谢Omniscia团队，虽然与 Vyper 没有直接关系，但为本事后报告提供了大量的共同作者、反馈和审查 ↩
2. 简而言之，"复制"意味着存储库中的标签可用于历史用途，但版本不会发布以供下载。有关更多信息，请参阅PEP-592。↩
3. 也就是说，编译器只会出错，而不生成任何代码。虽然编译器恐慌对用户来说很烦人，但它被认为是"安全"错误，因为它不会发出使其进入生产环境的代码。↩
4. 从技术上讲，还有其他方法可以传输以太币，但在撰写本文时它们并不适用。EIP-5920可能是一个积极的发展。↩