hexens × polygon zkEVM

安全审查报告

多边形 ZKEVM

内容

内容

关于六角星

六角形是一家网络安全公司，致力于提升 Web 3.0 的安全标准，为用户创造更安全的环境，并确保大规模采用 Web 3.0。

六角形拥有多个专注于信息安全不同领域的一流审计团队，在最具挑战性和技术复杂性的任务中展现出极致的表现，包括但不限于：基础设施审计、零知识证明/新型密码学、DeFi 和 NFT。Hexens 不仅使用广为人知的方法和流程，而且注重在日常工作中发现和引入新的方法和流程。

2022 年，我们的团队宣布结束由领先的 Web 3.0 风险投资机构 IOSG Ventures 领投的 420 万美元种子轮融资。其他投资者包括 Delta Blockchain Fund、Chapter One、Hash Capital、ImToken Ventures、Tenzor Capital，以及来自 Polygon 和其他区块链项目的天使投资人。

自 Hexens 于 2021 年成立以来，它在业界拥有令人印象深刻的业绩记录和认可度：Mudit Gupta - Polygon Technology 的首席信息安全官 - 最大的 EVM 生态系统，仅在完成一次合作迭代后就加入了公司顾问委员会。Polygon Technology、1inch、Lido、Hats Finance、Quickswap、Layerswap、4K、RociFi 以及数十个 DeFi 协议和桥接器已经成为我们的客户，并采取积极措施保护他们的资产。

审计
由...领着

瓦赫
卡拉佩蒂安

联合创始人/首席技术官 | 六角形

审计开始日期
17.12.2022

审计完成日期
27.02.2023

---

hexens × polygon zkEVM

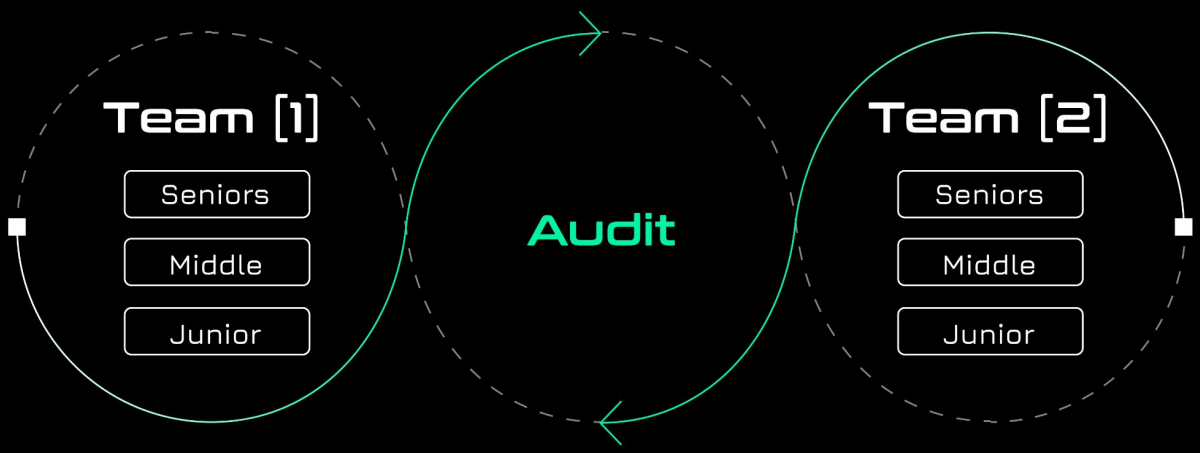
方法

共同审计程序

公司通常只指派一名工程师进行一项没有指定级别的安全评估。尽管指定的工程师可能拥有无可挑剔的技能，但它会带来可能影响产品生命周期的人为因素风险。

十六进制方法论

Hexens 方法涉及 2 个团队，包括多名不同资历的审计员，至少有 5 名安全工程师。这种独特的交叉检查机制有助于我们提供市场上最好的质量。



严重性结构

漏洞严重性是根据两个组件计算的

- 漏洞的影响
- 漏洞概率

影响	可能性			
	稀有的	不太可能	有可能	很可能
低/信息	低/信息	低/信息	中等的	中等的
中等的	低/信息	中等的	中等的	高的
高的	中等的	中等的	高的	批判的
批判的	中等的	高的	批判的	批判的

严重性特征

漏洞的严重程度和影响各不相同，了解它们的严重程度以便确定解决方案的优先级非常重要。以下是不同类型的漏洞严重程度：

批判的

这种严重程度的漏洞可能会导致重大的经济损失或声誉受损。它们通常允许攻击者获得对合约的完全控制，直接窃取或冻结合约或用户的资金，或者永久阻止协议的功能。例子包括无限铸币和治理操纵。

高的

这种严重程度的漏洞可能会导致一些经济损失或声誉受损。它们通常允许攻击者直接从合约或用户那里窃取收益，或者暂时冻结资金。示例包括不适当的访问控制整数上溢/下溢或逻辑错误。

中等的

这种严重程度的漏洞可能会对协议或用户造成一定程度的损害，而攻击者却无利可图。它们通常允许攻击者利用合约造成伤害，但影响可能有限，例如暂时阻止协议的功能。示例包括未初始化的存储指针和无法检查外部调用。

低的

这种严重程度的漏洞可能不会导致经济损失或重大伤害。但是，它们可能会影响合同的可用性或可靠性。示例包括滑点和抢先交易，或较小的逻辑错误。

信息性的

这种严重程度的漏洞与气体优化和
代码风格。它们通常涉及文档问题、EIP 标准的不正确使用、节省气体的最佳实践或合同的总体设计。示例包括不符合 ERC20，或文档和代码之间存在分歧。

在评估项目的安全性时，重要的是要考虑所有类型的漏洞，包括信息漏洞。全面的安全审计应考虑所有类型的漏洞，以确保最高级别的安全性和可靠性。

管理人员概括

概述

Polygon zkEVM 是一项新技术，是第一个与 EVM 等效的零知识扩展解决方案，现有的智能合约、开发人员工具和钱包可以在其中无缝工作。Polygon zkEVM 利用零知识证明来降低交易成本并提高吞吐量，同时继承以太坊的安全性。

zkEVM 网络 (L2) 中的交易被编译成批次，然后这些批次在以太坊智能合约中排序，之后它们的状态转换在以太坊上被证明和验证，达到可信状态。

zkEVM 有多个操作层：

- 网络层：Sequencer 和 Aggregators 运行的地方。
- ROM 层：zkEVM 使用一种称为 zkASM 的新语言来实现 EVM，从而使 EVM 状态交易可证明。
- 硬件层：zkEVM 使用一种称为 PIL 的新语言来创建多项式恒等式、约束，以保证 zkASM ROM 的完整和良好执行。
- L1 以太坊智能合约：桥接网络之间的资产并实施 PoE（效率证明）共识，确保批次的正确状态转换。

在安全审查期间，Hexens 团队涵盖了最关键的攻击面，包括智能合约、PIL 硬件和 zkASM ROM、EVM 和 zkEVM 之间的意外差异等。

由于其复杂性和可组合性，审查需要网络安全不同领域的广泛专业知识和思维方式。

我们已经设法在一些主要组件中找到严重漏洞，以及一个严重程度高且主要是小问题的建议。

此外，在审核期间，该团队还为 PIL 语言开发了一个静态分析工具，该工具在整个审核过程中为我们提供了帮助。

最后，我们报告的所有问题都由开发团队修复并由我们验证。

我们得出的结论是，在审查完成后，整体安全性和代码质量有所提高。

范围

分析的资源位于：
https://github.com/0xPolygonHermez/zkevm-contracts
https://github.com/0xPolygonHermez/zkevm-proverjs https://
github.com/0xPolygonHermez/zkevm-rom https://
github.com/0xPolygonHermez/zkevm -存储-ROM

报告中描述的问题已在以下提交中修复：

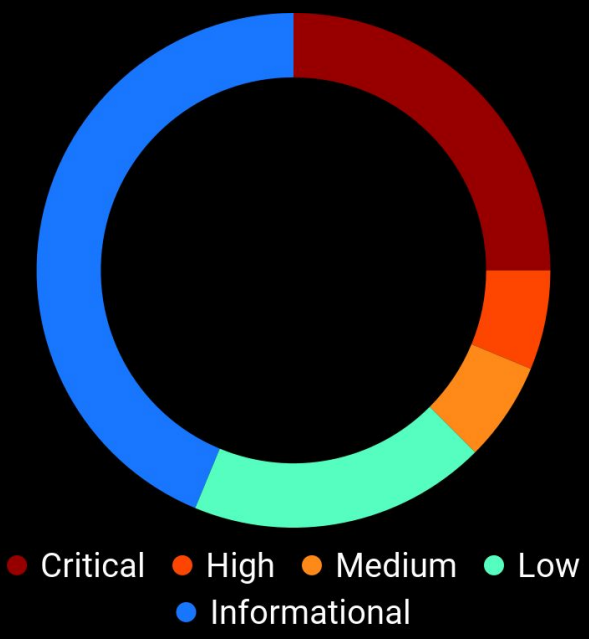
https://github.com/0xPolygonHermez/zkevm-contracts/commit/ec421a4499f07b65d2242f39bb039476ec1cf5e1
https://github.com/0xPolygonHermez/zkevm-proverjs/commit/ab3dbf24172b828e3ff4bbb0238f866199f0c834
https://github.com/0xPolygonHermez/zkevm-rom/commit/2ddeffbed7c022e04032e6d56ed6c6fb14cc38dc
https://github.com/0xPolygonHermez/zkevm-storage-rom/commit/97af71cd372ae6715e818795266d02a5a854cfa6

概括

严重性	调查结果数
批判的	4个
高的	1个
中等的	1个
低的	3个
信息性的	7

总计：16

严重性



地位

弱点

本节包含已发现弱点的列表。

1. ERC777重入攻击

严重性：批判的

小路：PolygonZkEvmBridge.sol

整治：为桥接合约中的桥接/声明功能实施重入锁

地位：菲固定的

描述：

在函数 bridgeAsset() 中，任何用户都可以桥接他的 ERC20 代币，指定目标网络和目标地址，如果它不是包装的代币，它将调用 transferFrom (SafeERC20.safeTransferFrom) 并更新合约的余额并最终创建一个在 Merkle 树中存放叶子。这座桥为桥接任何事物提供了自然的机会

一种ERC20代币，也包括ERC20的扩展版本，比如

ERC777。

ERC777 令牌（一些示例是 Verasity、imBTC、AMP、p.Network 令牌等，例如 Polygon Bridge 甚至在
他们的控制列表）是一种 ERC20 令牌，它通过调用挂钩等几个特性扩展了它的功能。

鉴于此以及余额计算如何在
合同中，任何人都有可能通过微妙的重入问题将所有这些代币从桥中耗尽。

攻击分解：
该漏洞的出现是因为余额更新是通过以下方式完成的：

```
// 为了支持费用代币检查收到的金额，而不是转移的金额
uint256平衡前 =IERC20可升级（令牌）。余额(地址(这));
IERC20可升级（令牌）。安全转移自(消息.发送者,地址(这）， 数量）；
uint256平衡后 =IERC20可升级（令牌）。余额(地址(这));

// 用收到的金额覆盖 leafAmount
leafAmount = balanceAfter - balanceBefore;
```

因此 leafAmount 正在计算调用前后的余额差额，以符合转账收费代币的要求。虽然乍一看似乎不可能重新进入 transferFrom 调用，因为接收方是桥接合约并且不受攻击者控制，但有

一个鲜为人知的调用挂钩，称为“ERC777TokensSender”，它将被调用
在余额实际转移到目的地之前的“发件人”地址。为了注册钩子，攻击者需要在
_ERC1820_REGISTRY（这是一个已知地址）上调用 setInterfaceImplementer

在以太坊中）并将自己注册为自己的“ERC777TokensSender”
接口实现者。

此时，无论何时调用 transferFrom (safeTransferFrom)，攻击合约都能够接收到 tokensToSend() 回调调用
“发件人”设置为攻击者地址。

为了利用可重入性并产生不公平的存款，攻击者
需要按以下方式重新进入 bridgeAsset() 函数：除了最后一次之外，所有重新进入
的调用都调用了 amount = 0（攻击
也完全有可能 amount=1 ，所以零检查不会
减轻它），并且只有在最后一次重新进入的呼叫中它应该发送一些数量
他想要放大的代币；为了简单起见，我们将采取
10**18（一个令牌）。
这种设计的重入将作为**uint256 balanceBefore =
IERC20Upgradeable(token).balanceOf(地址(this));**已经设置好了，即使我们
重新进入 bridgeAsset 调用也不会改变。并且存款会被重入级别放大。

重入分解：

- bridgeAsset（金额= 0）：
- LEVEL = 1，balanceBefore = 0：
  - token.safeTransferFrom() -> bridgeAsset(金额 = 0):
  - LEVEL = 2，balanceBefore = 0
    - token.safeTransferFrom() -> bridgeAsset(金额 = 0):
    - LEVEL = 3，balanceBefore = 0
      - token.safeTransferFrom() -> bridgeAsset(金额 = 10^{18}):
      - LEVEL = 4，balanceBefore = 0，balanceAfter = leafAmount = 10^{18}
    - LEVEL = 3，balanceAfter = leafAmount = 10^{18}
  - LEVEL = 2，balanceAfter = leafAmount = 10^{18}
- LEVEL = 1，balanceAfter = leafAmount = 10^{18}

可以看出，所有的重入层级最终都会计算
leafAmount = 10*18 并进行正确的存款并发出相应的事件（具有相应的
depositCount）。
因此在这种情况下存入的金额将是 3 * (10**18)，这是可以做到的
对于任意数量的令牌数量和任意数量的重入级别，在
一般有存款**级别 * 数量**
此外，攻击只需要执行一笔交易，并且可以在使用主动桥之前完成，因为攻击者可以创建存款
离开并等待合适的时机来领取它们。

2. PIL 中缺少约束导致证明 SMT 中存在虚假包含

严重性：批判的

小路：存储.pil

整治：添加缺少的二进制约束

地位：菲固定的

描述：

Storage状态机使用SMT（Sparse Merkle Tree）并实现
可证明的 CRUD 操作与存储 ROM 结合使用。

为了证明SMT中包含(Key, Value)，状态机
将密钥表示为位串。它使用密钥的 LSB（最低有效位）从根向下遍历树：该位的 0/1
值
对应左/右边缘遍历。

由于树是稀疏的，叶子层级不一定等于关键位
长度，这意味着一旦叶子被插入到树中，键的剩余部分 (rkey) 就会被编码到叶
子值中。

包含检查算法由两部分组成（参考链接：*https://wiki.polygon.technology/docs/zkEVM/zkProver/basic-smt-ops*）：

1. 检查根 - 作为通用的默克尔树根检查完成

使用兄弟散列从叶子爬到根。

2. 检查密钥：

I. 为了检查密钥，状态机预先准备下一个路径位

到剩余的密钥（rkey），例如**密钥||b1||b0**(对于叶级别 = 2)。

二。运行结束，可以在Storage ROM中区分

通过 LATCH_GET 命令，iLatchGet 选择器被设置为 1。

三、在主 SM 中使用置换约束以确保密钥

从 zkEVM ROM 传递过来的与在步骤 (I) 中构造的匹配：

```
sRD{
   SR0+2^32*SR1,SR2+2^32*SR3,SR4+2^32*SR5,SR6+2^32*SR7,
   密钥[0],密钥[1个],密钥[2个],密钥[3个
   ], op0,op1,op2,op3,
   op4,op5,op6,op7,
   计数器
}是
贮存.iLatchGet{
   贮存.老根0,贮存.老根1,贮存.老根2,贮存.老根3,
   贮存.rkey0,贮存.密钥1,贮存.密钥2,贮存.密钥3,
   贮存.值低0,贮存.值低1,贮存.值低2,贮存.值低3,
   贮存.值高0,贮存.值高1,贮存.值高2,贮存.值高3,
   贮存.计数器+2个
};
```

因此，部分（1）用于证明SMT中存在Value，并且
第（2）部分用于证明Value实际上与正确的Key绑定。

出现此问题的原因是下一位多项式 rkeyBit 缺少二进制
存储 SM 中的约束，以及存储 ROM 的事实，
很自然地，不断言下一位来自免费
输入，不能是 0 或 1 以外的任何其他值，并使用 JMPZ。

storage_sm_get.zkasm:

```
; 如果下一个关键位为零，则兄弟散列必须在右边（兄弟的关键位为 1）
${GetNextKeyBit()} =>RKEY_BIT
RKEY_BIT                    :JMPZ(Get_SiblingIsRight)
```

尽管如此，它不能被直接滥用；必须克服一些限制。由于密钥重构算法的特殊性以及第（1）部分中的值包含检查需要同时成立的事实。

限制概述：

在 Storage SM 中，密钥被分解为四个寄存器：rkey0,..,rkey3 并且路径是通过循环该寄存器的连续位来构建的
寄存器：

路径 = [rKey0_0, rKey1_0, rKey2_0, rKey3_0, rKey0_1, ...]

（参考链接：
https://wiki.polygon.technology/docs/zkEVM/zkProver/construct-key-path）

因此，为了从位中重建密钥，相应的 rkey 多项式需要加上该位：

rkey[level % 4] ||= rkeyBit

值得一提的是，密钥实际上是
账户地址、存储槽和查询密钥。

为了避免**模 4**操作时，Storage SM引入了LEVEL
寄存器，它也由 4 部分组成：level0,..,level3 和
存储 ROM 中的 ROTATE_LEVEL 操作码。

LEVEL 首先设置为**叶级 % 4**，然后使用 ROTATE_LEVEL
每次证明者需要爬树时：

存储 sm-get.zkasm：

```
; 更新剩余密钥
                    :旋转水平
                    :CLIMB_RKEY

                    :JMP(Get_ClimbTree)
```

水平旋转（storage.pil）：

```
波尔旋转Level0=iRotateLevel*(1级-0级) +0级;
波尔旋转Level1=iRotateLevel*(level2-1级) +1级;
波尔旋转Level2=iRotateLevel*(3级-level2) +level2;
波尔旋转Level3=iRotateLevel*(0级-3级) +3级;
```

最后，当使用 CLIMB_RKEY 操作码时，rkey 将被修改

存储.pil：

```
波尔已爬Key0= (0级*(rkey0*2个+密钥位-rkey0) +rkey0);
波尔爬上Key1= (1级*(密钥1*2个+密钥位-密钥1) +密钥1);
波尔爬上Key2= (level2*(密钥2*2个+密钥位-密钥2) +密钥2);
波尔爬上Key3= (3级*(密钥3*2个+密钥位-密钥3) +密钥3);
```

为了保持上述置换约束所有
rkey0-3 寄存器必须在操作结束时修改（当使用 LATCH_GET 操作码时）。以及存储 ROM 将重新排列
左/右节点通过匹配下一位进行散列。鉴于一个事实
只能滥用下一位的非零值，限制可以是
通过使用具有的键将任意值插入 SMT 来克服**1111**

作为它的 LSB：。

密钥 = ***1111, 这是需要有机会改变所有 4 rkey
寄存器。

这意味着从帐户地址和存储槽号（除了存储查询密钥）派生的 POSEIDON 哈希需要将其结果寄存器 hash0、..、hash3 的最低有效位设置为 1：

```
散列0= ***1个
散列1= ***1个
哈希2= ***1个
哈希3= ***1个
```

由于每个 POSEIDON 哈希寄存器只有 1 位是固定的，因此很容易做到
克服 4 位熵并找到一个存储槽（对于任何给定的帐户
地址）满足攻击先决条件。

另一个限制是被插入的叶子必须具有大于 4 的级别，在现实世界的场景中保证是这种情况（使用
相反的概率可以忽略不计），因为将插入数百万片叶子
到树上。即使不是这种情况，攻击者也只需要按照相同的规则预先计算两个存储槽，然后将它们都插入以保证最低级别。

插入后 (**KeyPrecomputed, ValueArbitrary)**进入SMT使用

opSSTORE 过程，从而满足攻击者可以伪造任何密钥绑定的先决条件**伪造密钥**与价值**值任意**， 经过

将自由输入的最后 4 个下一位值设置为：

```
密钥位[0] =      rkeyToFake[0] -     rkey0*2个
密钥位[1个] =    rkeyToFake[1个] -   密钥1*2个
密钥位[2个] =    rkeyToFake[2个] -   密钥2*2个
密钥位[3个] =    rkeyToFake[3个] -   密钥3*2个
```

由于 Storage ROM 使用 JMPZ 来区分爬升路径，因此尽管 rkeyBit 大于 1，但将被视为与设置为 1 相同的方式，并且将成功绕过根检查（值包含）。

有利于攻击者的主要影响将是伪造包含 **（KeyAttackerBalance，ArbitraryAmount）**在贴片机。

3. 错误的 CTX 分配导致随机数量的以太币添加到

音序器平衡

严重性：批判的

小路：process-tx.zkasm，预编译/identity.zkasm

整治：在跳转到 handleGas 标签之前更改 identity.zkasm 代码以将 originCTX 保存在寄存器中

地位：菲固定的

描述：

zkEVM ROM 架构使用 Contexts (CTX) 来划分和仿真
一个事务内的调用上下文之间的虚拟地址到物理地址的转换。一个 CTX 地址空间用于确定动态
在调用上下文之间变化的内存空间，以及堆栈和 CTX 变量（例如 msg.sender、msg.value、active storage account
等等。）。上下文切换是使用辅助变量完成的，例如 originCTX，它指的是创建当前上下文的原始 CTX 作为
以及当前的 CTX。有一个特殊的 CTX(0) 用于存储全局变量，例如 tx.origin 或旧状态根，第一个上下文批处理
transaction 以 CTX(1) 开始，并随着新调用、上下文切换或正在处理的事务而递增。

漏洞在于“身份”（0x4）预编译合约
执行。如果没有设置 originCTX，这实际上意味着 EOA 直接调用预编译合约，而不是在
内部合约调用，预编译的合约应该消耗内在的gas
并结束交易执行。虽然做了上下文切换
正确地在 ecrecover (0x1) 预编译中，身份预编译是
上下文切换错误。要检查交易是否直接调用合约，使用 originCTX 变量并检查它是否是
等于 0：
https://github.com/0xPolygonHermez/zkevm-rom/blob/develop/main/precompiled/identity.zkasm#L21

```
$=>CTX        :负载(产地CTX),JMPZ(处理气体)
```

虽然它立即将 originCTX 加载到 CTX 寄存器中，但所有内存操作都将为 CTX(0) 完成。

由于 GLOBAL 和 CTX 上下文之间的上下文切换是通过 useCTX 完成的：
https://github.com/0xPolygonHermez/zkevm-proverjs/blob/develop/pil/main.pil#LL203-L204C85

```
波尔地址关系=工业*E0+indRR*RR+抵消;
波尔地址=使用CTX*CTX*2^18+是堆栈*2个^16+是堆栈*SP+是内存*2^17+地址关系;
```

全球 -> 使用 CTX = 0，CTX -> 使用 CTX = 1

如果 CTX 寄存器设置为 0，最终地址实际上是相同的。鉴于变量是通过它们的偏移量寻址的，ROM 的全局
变量将被其适当的 CTX 变量双重引用
相同的偏移量。

例子：

OFFSET(0): VAR GLOBAL oldStateRoot <--> VAR CTX txGasLimit

OFFSET(1): VAR GLOBAL oldAccInputHash <--> VAR CTX txDestAddr

…

OFFSET(17): VAR GLOBAL nextHashPId <--> VAR CTX gasRefund

从而使 GLOBAL 和 CTX 变量偏移发生冲突。

攻击分解：

- 用户（EOA）创建一个目标地址设置为身份预编译合约（0x4）的交易
- 当执行到达

  $ => CTX :MLOAD(originCTX), JMPZ(handleGas)

  CTX 将被设置为 0，并且跳转到 handleGas 标签。

handleGas 将检查退款（一个重要的细节是在
当前 VAR 配置，gasRefund 变量与 nextHashPId 冲突，在这种情况下将为 0，尽管
如果它与另一个 VAR 冲突
绝对值越大，那么调用者将有机会“凭空打印钱”给自己），在退还给发送者之
后
继续到需要计算定序器地址消耗的气体的程度。

```
;; 将消耗的 gas 发送到 sequencer 发
送气体序列:
    $=>A          :负载(交易气体限制)
    A-气体=>A

    $=>乙         :负载(txGas价格)
    ; Arith 乘法运算
    A             :商城(arithA)
    乙            :商城(算术B),称呼(多维斯)
    $=>丁         :负载(arithRes1); 在 D 中支付定序器的价值
```

由于 txGasLimit 引用了 oldStateRoot，它是状态的哈希值
树并且有一个非常大的绝对值，**MLOAD(交易气体限制)**将返回 oldStateRoot 值。
通过将 gasPrice 设置为 1（或任意小的值而不溢出乘法），定序器将是

记入一个巨大的余额。

攻击要求和概率：

对于任何能够将大量以太币余额归功于自己的用户，他需要
成为排序它的人。最方便的方法是在 L1 PolygonZkEVM 合约中强制进行批处理。

与当前配置一样，可信排序器忽略强制
批次；它将它们存储在数据库中单独的 state.forced_batch 表中：
https://github.com/0xPolygonHermez/zkevm-node/blob/develop/state/pgstatestorage.go#L316-L32

以及当排序器将查询待定批次时
在 getSequencesToSend() 函数中排序：
https://github.com/0xPolygonHermez/zkevm-node/blob/develop/sequencer/sequencesender.go#L114

它只会查询 state.batch 表中的批次：
https://github.com/0xPolygonHermez/zkevm-node/blob/develop/state/pgstatestorage.go#L535-L539

因此攻击者将需要强制批处理然后等待超时
传递并对其进行排序，将排序器设置为任意地址。在当前配置中，如果结合使用其他“虚拟”交易混淆交易并在某处添加 bridgeAsset() 调用，则攻击为任何人提供了强制执行此类批处理的机会，并在超时期限后记入大量以太币余额同一批次，一旦批次被验证，攻击者将获得任意以太币数量的存款叶，并且可以耗尽桥持有的所有以太币。

4. PIL 中缺少约束导致执行流劫持

严重性：批判的

小路：utils.zkasm, main.pil

整治：为 inNeg 多项式添加一个约束，以确保它的计算结果仅为 0 或 1。例如 isNeg * (1-isNeg) = 0

地位：菲固定的

描述：

zkEVM ROM 中的自由输入检查和 main.pil 中缺少约束的组合导致执行劫持，并有可能跳转到 ROM 中的任意地址。

其中一个影响是任意增加任何调用者的余额。

在文件 utils.zkasm 中，一些过程使用自由输入调用来进行小的计算，例如，computeSendGasCall：

```
; C = [c7, c6, ..., c0]
; JMPN 指令确保 c0 在 [0, 2^32 - 1] 范围内 ${气体 >> 6} =>C
                              :JMPN(失败断言)
${GAS & 0x3f} =>丁

; 因为 D 肯定小于 0x40
; 强制 [c7, c6, ..., c1] 为 0，因为没有值乘以 64
; 等于字段
; 由于 e0 保证小于 32 位，因此 c0 * 64 + d0 不会溢出字段
C*64+丁                :断言
```

在这种情况下，使用 JMPN 来确保自由输入的有效性。

JMPN 将有效检查寄存器 C 中设置的自由输入是否在范围 $[0,2^{32}-1]$。这是一个安全假设，可确保寄存器在断言阶段不会溢出：

```
C*64+丁                :断言
```

JMPN 约束：

https://github.com/0xPolygonHermez/zkevm-proverjs/blob/develop/pil/main.pil#L209-L228

```
波尔jmpn条件值=JMPN*(是负数*2个^32+op0);
```

通过检查 jmpnCondValue 是一个 32 位数字，我们确保 op0 在 $[-2^{32},2^{32})$ 范围，从而防止溢出。跳转目的地以及 zkPC 约束因此基于

负数：

https://github.com/0xPolygonHermez/zkevm-proverjs/blob/develop/pil/main.pil#L322-L336

```
zkPC' =多JMP* (最终跳转地址-下一个NoJmpZkPC) +其他JMP* (最后的其他地址-
下一个NoJmpZkPC) +下一个NoJmpZkPC;
```

尽管如此，缺少一个约束来确保 isNeg 仅评估为
1 或 0。对于 utils.zkasm 过程，没有指定 elseAddr，并且具有：

```
最后的其他地址=下一个NoJmpZkPC
doJMP=是负数
其他JMP= (1个-是负数)
```

zkPC 约束可以简化为：

```
zkPC' =是负数* (最终跳转地址-下一个NoJmpZkPC) +下一个NoJmpZkPC
```

其中 finalJmpAddr 和 nextNoJmpZkPC 都是 ROM 程序编译阶段的已知值。

为了能够跳转到任意zkPC，攻击者需要计算isNeg和op0对应的值；这可以使用派生公式来完成：

isNeg = (zkPC_arbitrary - nextNoJmpZkPC) * (finalJmpAddr - nextNoJmpZkPC)-1 mod P

op0 = - isNeg * 2^{32}模型 P

攻击分解：

此时攻击者有一个跳转到任意地址的原语；这
下一步将是找到一个合适的小工具跳转到，目标的主要要求是：

- 不要破坏/恢复 zkEVM 执行
- 对攻击者有利的影响

发现的跳转链之一是使用 *CALL 操作码之一作为
攻击链的开始调用 computeSendGasCall 并随后跳转到 refundGas 标签的代码：

https://github.com/0xPolygonHermez/zkevm-rom/blob/develop/main/process-tx.zkasm#L496-L498

```
$=>A                    :负载(txSrcOriginAddr)
0=>乙,C                  ; 平衡键smt
$=>SR                   :商店
```

这会将 txSrcOriginAddr balance 设置为寄存器 D 中包含的值
并完成交易执行。要滥用 SSTORE 指令设置的值，攻击者需要在寄存器 D 中设置巨大的值，为此可以使用 DELEGATECALL 操作码，因为在实现中它在调用 computeSendGasCall 之前设置寄存器 D：

```
$=>丁        :负载(存储地址)
...
乙          :商城(retCallLength),称呼(计算气体发送呼叫); in: [gasCall: gas sent to call] out: [A: min(
requested_gas , all_but_one_64th(63/64))]
```

所以D寄存器将设置绝对值很大的storageAddr。

攻击的附加设置：

- 应使用启动对任何地址的 delegatecall() 调用的函数（或回退）来部署合约。
- 事务应在 gasPrice 设置为 0 的情况下启动，以便在将其发送到定序器时不会溢出气体，并且这将
  支持攻击者证明批处理启动的事实。
- 应该预先计算 gasLimit 以在最后以 0 gas 结束
  执行的原因，这是出于与上述相同的原因

5. MAXMEM 处理中的 BUG 可以停止批处理验证

严重性：高的

小路：main.zkasm，创建-终止-context.zkasm

整治：解决此问题的方法之一是使用 BITS17 更改 diffMem 查询，或将 MAX_MEM_EXPANSION_BYTES 缩小为 2_{21}-32，这样 MAXMEM 的最大值可以是 2_{16}-1

地位：菲固定的

描述：

在 zkEVM ROM 中，用于设置所用内存最大偏移量的 MAXMEM 寄存器仅被设置为零一次——用于执行跟踪的第一步。

如果引用的内存的相对地址高于

当前设置的一个：

```
波尔地址关系=工业*E0+indRR*RR+抵消;

波尔最大内存关系=是内存*地址关系;

波尔最大内存计算=最大内存*(地址关系-最大内存) +最大内存;

最大内存' =设置最大内存* (op0-最大内存计算) +最大内存计算;
```

对当前 MAXMEM 和

相对地址：

```
波尔差异内存=最大内存* ( (最大内存关系-最大内存) -
            (最大内存-最大内存关系) ) +
      (最大内存-最大内存关系);
最大内存* (1个-最大内存) =0;
...
差异内存在全球的.字节2;
```

这有效地检查了差异不是负数，以及 isMaxMem 是否正确设置，并且它限制了

差异为：$|maxMemRel - MAXMEM| < 2^{16}$

另一方面，zkEVM ROM 对相关内存有限制检查

中完成的抵消**utils.zkasm:saveMem**程序：

```
$=>乙                :负载(lastMemOffset)
; 如果二进制文件有进位，意味着内存扩展非常大。我们可以直接跳转到oog
; B 中的偏移量 + 长度
$=>乙                :添加,JMPC(没气了)
; 检查新内存长度是否低于 2**22 - 31 - 1（最大支持的内存扩展
%TX_GAS_LIMIT 气体）
%MAX_MEM_EXPANSION_BYTES=>A
$                  :LT,JMPC(没气了)
```

虽然区别在于**MAX_MEM_EXPANSION_BYTES**等于 2^{22}- 32. 作为 ROM 有效用作相对值的偏移量

地址将是偏移量/32，这意味着合约可以将内存偏移量推到最大值（2^{22}-32) / 32 = 2^{17}- 1.

由于 diffMem 应该在 BYTE2 范围内，攻击者将需要两次 MLOAD 或 MSTORE 操作来将 MAXMEM 推到大于 BYTE2 的值

范围，例如**opMSTORE(1000) + opMSTORE(2^{16}+ 999)**.

由于 MAXMEM 永远不会复位，并且**Global.BYTE2 中的 diffMem；**才不是有一个选择器——对于使用 GLOBAL/CTX 或堆栈变量的内存操作，maxMemRel 将等于 0（因为 isMem 将为 0）：

```
波尔最大内存关系=是内存*地址关系;
```

这意味着 diffMem：

```
波尔差异内存=最大内存* ( (最大内存关系-最大内存) -
                (最大内存-最大内存关系) ) +
        (最大内存-最大内存关系);
```

要么等于**maxMemRel - 最大内存**--负值或 **MAXMEM - maxMemRel**这将是 > 2_{16} -1，因此不会满足 BYTE2 的连续查找。

这使任何非法行为者都能够将交易发送到受信任的排序器或强制执行它，并且一旦对批次进行排序，就不可能证明下一个状态转换。

6. ZKASM ECRECOVER 中的错 误限制检查

执行

严重性： 中等的

小路：
https://github.com/0xPolygonHermez/zkevm-rom/blob/develop/main/ecrecover/ecrecover.zkasm#L61C8-L67

整治： FNEC_DIV_TWO 应该等于
57896044618658097711785492504343953926418782139537452191302581570759080747168 (Fp/2)

地位： 菲固定的

描述：

在ecrecover函数的zkASM实现中，必须有一个
检查 ECDSA 签名延展性。检查应在验证交易签名时进行，而在验证时省略

调用预编译的 ecrecover。为了使ECDSA签名不可延展，S值不应大于Fp/2（S <= Fp/2）；FP/2 = **57896044618658097711785492504343953926418782139537452191302581570759080747168**此检查在 EVM (Go-ethereum) 中实现
https://github.com/ethereum/go-ethereum/blob/f53ff0ff4a68ffc56004ab1d5cc244bcb64d3277/crypto/crypto.go#L268-L270

尽管在 zkASM 的情况下 ecrecover 再次检查 Fp/2 + 1
(**57896044618658097711785492504343953926418782139537452191302581570759080747169**)，而不是 Fp/2，因此 S 值的允许范围
还包括 Fp/2 + 1。这种差异可以被滥用来生成证明
对于不符合 EVM 的交易。

```
CONSTL%FNEC_DIV_TWO=
57896044618658097711785492504343953926418782139537452191302581570759080747169n
...
ecrecover_tx:
    %FNEC_DIV_TWO:商城(ecrecover_s_upperlimit)
...

    ; s 在 [1, ecrecover_s_upperlimit]
    $=>A       :负载(ecrecover_s_upperlimit)
    $=>乙      :负载(ecrecover_s)
    $         :LT,JMPC(ecrecover_s_is_too_big)
    0n=>A
    $          .情商,JMPC(ecrecover_s_is_zero)
```

7. ZKEVM 和 EVM 之间交易 RLP 解码的差异

严重性：低的

小路：
https://github.com/0xPolygonHermez/zkevm-rom/blob/develop/main/load-tx-rlp.zkasm#L164-L206

整治：添加对字符串和列表解码的检查，以确保对超过 55 字节的大小进行长格式编码

地位：菲固定的

描述：

在 load-tx-rlp.zkasm 交易 RLP 解码代码标签 dataREAD 的实现中代表解码 DATA 字段的部分
交易。数据字段被编码为 RLP 字符串，并且可以是短的和长的大小，如 RLP 格式所述
(*https://ethereum.org/en/developers/docs/data-structures-and-encoding/rlp#definition*）字符串可以根据长度以两种方式表示
（0-55 字节和 55+ 字节）：

- ……
- *否则，如果字符串的长度为 0-55 个字节，则 RLP 编码由值为 0x80（dec.128）的单个字节加上字符串的长度组成*
  其次是字符串。因此第一个字节的范围是 [0x80, 0xb7]
  （十二月 [128, 183]）。
- *如果字符串的长度超过 55 个字节，则 RLP 编码包含一个*
  值为 0xb7（183 年 12 月）的单字节加上二进制形式的字符串长度的字节长度，后跟字符串的长度，
  其次是字符串。例如，一个 1024 字节长的字符串将被编码为 |xb9|x04|x00 (dec. 185, 4, 0) 后跟字符串。这里，0xb9 (183 + 2 = 185) 作为第一个字节，后面跟着 2 个字节 0x0400 (dec. 1024)，表示实际字符串的长度。因此第一个字节的范围是 [0xb8, 0xbf] (dec. [184, 191])。

- ……

当一个人想要用一个
短数据字段（小于 55 字节）但以长字符串格式编码
（0xb801AA = [“AA”]，1 字节字符串的示例，但采用长字符串格式）。EVM 实施了这些检查以避免此类情况：
*https://github.com/ethereum/go-ethereum/blob/master/rlp/decode.go#L1008-L1011*因此，由于 zkEVM-node 继承了 lib，它具有
检查也是如此。
然而，zkEVM ROM 的 RLP 解码机制无法检查这些情况（短/长字符串和短/长列表）。

一个有保证的影响是批量排序机制将是
由于这种“毒”交易而被阻止和破坏。可能的步骤是：

1. 攻击者强制使用错误的 RLP 编码进行批处理
2. 受信任的定序器对其进行排序或忽略以及超时
   经期过去
3. 受信任的排序者或攻击者对强制批次进行排序
4. 攻击者验证批次，因为它完全可以被 zkEVM ROM 证明
   并从桥梁中索取或使用由毒物交易转移的资产

5. zkEVM 网络停止，因为同步器无法与强制批处理同步，并且网络保持不同步

6. 针对这种情况可能采取的应对措施：
   1. 修复 zkEVM ROM 中的 RLP 解码并通过重新部署具有较旧状态根的效率证
      明合约来回滚 L1 中的状态

   2. 将 RLP 解码更改为节点中的错误解码，并
      使用不正确的 RLP 为网络推送更新

由于最可能的情况是采用 6.a 解决方案，而不是不实际修复错误并在节点 (6.b)
中引入新错误，因此
情况将有利于攻击者，因为在回滚之后，将有可能
基本上双花资产。

```
数据读取:
    $=>丁                  :负载(batchHashPos）:
    丁                   商城(数据开始)
    1个=>丁
    %CALLDATA_OFFSET=>SP          :称呼(添加HashTx)
                        :称呼(addBatchHashData)
    A-0x80                 :JMPN(非常短数据)
    A-0x81                 :JMPN(结束数据)
    A-0xb8                 :JMPN(短数据)
    A-0xc0                 :JMPN(长数据,无效的TxRLP)

非常短的数据:
    1个                  :商城(txCalldataLen)
    31=>丁                 :称呼(SH拉瑞斯)
    A                    :商城(SP++),JMP(结束数据)

短数据:
    $=>丁                  :负载(batchHashPos)
    丁                   :商城(数据开始)
    A-0x80=>乙                :商城(txCalldataLen),JMP(读取数据)

长数据:
    A-0xb7=>丁                :称呼(添加HashTx)
                        :称呼(addBatchHashData)
    $=>丁                  :负载(batchHashPos）:
    丁                   商城(数据开始)
    A=>乙                  :商城(txCalldataLen)

读取数据:
    ;检查二进制文件
    32=>丁
    乙-丁                  :JMPN(读取数据最终)
    乙-丁                  :商城(发送数据读取),称呼(添加HashTx)
    A                    :商城(SP++),称呼(添加 BatchHashByteByte)
    $=>乙                  :负载(发送数据读取),JMP(读取数据)

读取数据最终:
    乙-1个                 :JMPN(结束数据)
    乙=>丁                 :称呼(添加HashTx)
    32-丁=>丁                :称呼(SH拉瑞斯)
    A                    :商城(SP)
    32-丁=>丁                :称呼(添加 BatchHashByteByte)
```

评估值：

```
案件b <0xC0:
  // 如果字符串长度超过 55 个字节，则 RLP 编码包含一个
  // 值为 0xB7 的单字节加上长度的长度
  // 二进制形式的字符串，后面是字符串的长度，后面是
  // 通过字符串。例如，长度为 1024 的字符串将被编码为
  // 0xB90400 后跟字符串。因此第一个字节的范围是
  // [0xB8, 0xBF]。
  尺寸,呃= 秒。读取单元(b -0xB7)
  如果错误==零&& 大小 <56{
    呃= ErrCanonSize
  }
  返回字符串，大小，错误
```

8. ZKEVM 和 EVM 之间的 GASLIMIT 和 CHAINID MAX 大小差异

严重性：低的

小路：
https://github.com/0xPolygonHermez/zkevm-rom/blob/develop/main/load-tx-rlp.zkasm

整治：根据 EVM 的气体限制和链 ID 最大大小更改气体限制和链 ID RLP 解码

地位：菲固定的

描述：

zkEVM 和 EVM 实现之间的交易 gas 限制和链 ID 最大大小之间存在差异。

在 *https://github.com/0xPolygonHermez/zkevm-rom/blob/develop/main/load-tx-rlp.zkasm*GasLimit 的最大大小定义为 256 位，而 ChainID 的最大大小为 64 位。
EVM 大小为：
Gas limit 的最大大小为 64 位
*https://github.com/ethereum/go-ethereum/blob/79a478bb6176425c2400e949890e668a3d9a3d05/核心/类型/tx_legacy.go#L29*ChainID 的最大大小为 256 位

https://github.com/ethereum/go-ethereum/blob/01808421e20ba9d19c029b64fcda841df77c9aff/core/types/transaction.go#L75

一个可能的影响是我们可以用正确的 chainID 创建一个交易
但被编码为一个更大的 uint，zkEVM nod 以及
EVM（RLP lib），但不可能证明包含它的批次
交易。鉴于 zkASM EVM 将在 load-rlp 中失败
阶段，批次将被丢弃，而且事实上有可能将这些类型的交易自由地发送到排序器，它
将有可能停止网络可用性。

zkASM ROM:

```
;; 阅读 RLP 'gas limit'
    ; 最大 256 位
gasLimit阅读:
    1个=>丁                  :称呼(添加HashTx)
                            :称呼(addBatchHashData)
    A-0x80                   :JMPN(结束气体限制)
    A-0x81                   :JMPN(气体限制0)
    A-0xa1                   :JMPN(shortGasLimit,无效的TxRLP)
```

```
;; 读取 RLP 'chainId'
    ; 最多 64 位
链读:
    1个=>丁                  :称呼(添加HashTx)
                            :称呼(addBatchHashData)
    A-0x80                   :JMPN(端链ID)
    A-0x81                   :JMPN(chainId0)
    A-0x89                   :JMPN(短链ID,无效的TxRLP)
```

EVM 交易：

```
// LegacyTx 是常规以太坊交易的交易数据。
类型遗留TX结构{
    随机数uint64               // 发送者账户的随机数
    GasPrice *big.Int        // 每个气体的 wei
    气体     uint64          // 气体限制
    到       * common.Address`rlp：“无”`// nil 表示合约创建
    价值      * big.Int       // 微量
    数据     []字节          // 合约调用输入数据
    V, R, S *big.Int         // 签名值
}
```

```
// TxData 是一笔交易的底层数据。//

// 这是由 DynamicFeeTx、LegacyTx 和 AccessListTx 实现的。 类型发送数据
界面{
    发送类型()字节// 返回类型 ID
    复制() 发送数据// 创建深拷贝并初始化所有字段

    链号() *big.Int
    访问列表() 访问列表
    数据() []字节
    气体()uint64
    汽油价格() *big.Int
    气嘴帽() *big.Int
    gasFeeCap() *big.Int
    价值() *big.Int
    随机数()uint64
    到() *common.Address

    原始签名值() (v, r, s *big.Int) 设置签名值
    (chainID, v, r, s *big.Int)
}
```

9. 改变条件跳跃的建议

严重性：低的

小路：/主/操作码/block.zkasm

整治：考虑使用 LT 操作码和 JMPC 条件跳转代替 JMPN；在这种情况下，txCount 应该用 ADD 操作码递增

地位：菲固定的

描述：

在分析 zkEVM ROM 操作码和状态机中适当的 PIL 表示时，已经注意到一些

条件跳转的操作寄存器大小不同：

- *JMPN（跳负）*

```
波尔jmpn条件值=JMPN*(是负数*2个**32+op0);
```

- *JMPZ（跳零）和 JMPNZ（不跳零）*

```
/// op0 检查零
pol提交op0Inv;
波尔op0IsZero=1个-op0*op0Inv;
op0IsZero*op0=0; . . .

波尔多JMP=JMPN*是负数+JMP+JMPC*携带+JMPZ*op0IsZero+返回+称呼;
```

可以看出 JMPN/Z/NZ 操作码只考虑 op0 寄存器，
与其他条件跳转不同，例如 JMPC、JMPNC，它们使用二进制状态机进位锁存器，它对 256 位值进行操作。

因此，这些类型的跳跃将只考虑下半部分
具有 8 槽（A、B、C、...）的寄存器：

```
A:JMPZ(一些标签)
```

只要 A0 = 0，跳跃就会发生，即使 A = [A0 = 0, A1 != 0 ,...,A7 != 0]，
例如，如果 A = 2^{32}. 这适用于 op0=A0,op1=A1,...op7=A7 并且只有 op0 将用于跳转条件。

我们迭代地查看了在所有 zkEVM ROM 中完成的所有 JMPN、JMPZ、JMPNZ 调用，以便找到使用 8 槽寄存器并且没有其他寄存器值限制（例如，更小大于 2^{32}).

经过多次迭代，找到了唯一一个寄存器可以容纳大于 2 的值的情况^{32}在 opBLOCKHASH 中：

```
opBLOCKHASH:
...
  ; 获取最后的 tx 计数
  $=>乙          :负载(发送计数)
  $=>A           :负载(SP); [区块编号 => A]
  ; 检查batch block是lt当前块号，否则返回0
  乙-A-1个       :JMPN(opBLOCKHASHzero)
```

这里的 JMPN 可以在以下场景中被错误触发
txCount 大于 2_{32}; 这很可能会发生，尽管不会在不久的将来发生，因为 zkEVM 网络中的区块实际上代表交易
（因此使用 txCount 的原因），并且平均
75 TPS 的预期它将在 1.5-2 年后与 32 位值重叠。因为这是
如果情况持续很长时间，问题的严重性就会降低。

尽管如此，如果发生这种情况，旧块编号的 BLOCKHASH 指令将开始返回
0 而不是它们的实际值。

10.循环优化

严重性：信息性的

小路：多边形ZkEvm.sol

整治：totalBatchesAboveTarget 可以通过 currentBatch - currentLastVerifiedBatch 递增，之后循环应该中断

地位：菲固定的

描述：

在函数 updateBatchFee 中，该函数计算在验证时间目标之上和之下验证的块数。费用根据他们的比率更新。

检查高于目标的目标的循环从新验证的批次到最后验证的批次进行反向循环，它做到了
直到它到达遍历其间所有序列的最后一个验证批次。可以优化循环，因为 sequencedTimestamp 是
严格增长并且从第一次发现一个序列是
在目标时间以上，直到最后一个验证批次之前的其余批次可以被认为是在目标时间以上进行验证（因为每个下一个 sequencedTimestamp 都将小于当前时间戳）。

优化将考虑所有批次，直到最后一次验证
批处理并在满足这样的序列后立即跳出循环。

多边形ZkEvm.sol

```
功能_updateBatchFee(uint64新上次验证批次)内部的{
  uint64currentLastVerifiedBatch =获取最后验证批次();
  uint64currentBatch = newLastVerifiedBatch;uint256totalBatchesAboveTarget;
uint256newBatchesVerified = newLastVerifiedBatch -
  当前最后一个验证批次;

尽管(currentBatch != currentLastVerifiedBatch) {
  // 加载排序的批处理数据
  序列化批数据
    贮存currentSequencedBatchData = sequencedBatches[
      当前批次
    ];

  // 检查时间戳是高于还是低于 VERIFY_BATCH_TIME_TARGET
  如果(
    堵塞.timestamp - currentSequencedBatchData.sequencedTimestamp >
    veryBatchTimeTarget
  ) {
    totalBatchesAboveTarget +=
      当前批次 -
      currentSequencedBatchData.previousLastBatchSequenced;
  }

  // 更新 currentLastVerifiedBatch
  currentBatch = currentSequencedBatchData.previousLastBatchSequenced;
}
```

11. VERIFYMERKLEPROOF 中的索引大小不正确

严重性：信息性的

小路：DepositContract.sol：L90-L112

整治：应检查索引是否低于 2**_DEPOSIT_CONTRACT_TREE_DEPTH（因为叶遍历将针对 _DEPOSIT_CONTRACT_TREE_DEPTH 级别完成）

地位：菲固定的

描述：

合约中DepositContract.sol

该索引的大小为 uint64，并且由于 merkle 树级别为 32，因此最
可以操纵索引的重要位以对同一索引进行双花。

由于调用此上下文的函数使用正确大小的索引 (uint32)，此问题的严重性较低，尽管这可能会导致
如果代码将被重构并被参数的 uint64 大小误导，则关键影响攻击向量。

```
功能验证默克尔证明(
    字节 32叶哈希,
    字节 32[]记忆smt打样,
    uint64指数,
    字节 32根
) 公纯回报(布尔值) { 字节 32节
    点=叶哈希;

    // 检查 merkle 证明
    uint256当前索引 = 索引;  为了(

        uint256身高=0;
        高度 < _DEPOSIT_CONTRACT_TREE_DEPTH;
        身高++
    ) {
        如果((currrentIndex &1个) ==1个)
            节点 =keccak256(阿比.编码打包(smtProof[高度], 节点));
        别的节点 =keccak256(阿比.编码打包(节点, smtProof[高度]));
        当前索引/=2个;
    }

    返回节点==根;
}
```

12. 冗余进口

严重性：信息性的

小路：PolygonZkEVM.sol

整治：删除多余的导入

地位：菲固定的

描述：

的进口**ERC20BurnableUpgradeable.sol**和**可初始化.sol**是多余的，因为它们要么未被使用，要么已经被递归导入。

```
进口“@openzeppelin/contracts-upgradeable/token/ERC20/utils/SafeERC20Upgradeable.sol”；
//进口
“@openzeppelin/contracts-upgradeable/token/ERC20/extensions/ERC20BurnableUpgradeable。
sol"; // 冗余导入
进口“./interfaces/IVerifierRollup.sol”；
进口“./interfaces/IPolygonZkEVMGlobalExitRoot.sol”；
//导入“@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol”； // 多余的
进口
进口“@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol”；
进口“./interfaces/IPolygonZkEVMBridge.sol”；
进口“./lib/EmergencyManager.sol”；
```

13. PLOOKUP 和置换选择器多项式

VERIFYPIL 工具与 STARK 生成之间的差异

严重性：信息性的

小路：
https://github.com/0xPolygonHermez/pilcom/blob/107765f18d78a865c12517c9c84cbdfbad1d99d0/src/pil_verifier.js#L194-L222

整治：verifyPil 方法必须检查两个选择器是否相等

地位：菲固定的

描述：

在 PIL 语言中，有一个选择器多项式用于选择特定的行以满足约束。

但是 verifyPIL 方法和 STARK 验证器使用该选择器的方式存在差异。对于 verifyPIL，如果选择器不为 0 并且可以不相等，则选择多项式跟踪的线。

例如我们有一个 PIL：

```
持续的%N=4个;

命名空间全球的(%N);
  极性常数L1;

命名空间排列示例(%N);

  极性常数b1,b2;// 选择器多项式 pol提交a1,a2;

  b1{a1}是b2{a2};
```

和相应的多项式：

```
常数b1= [0,5个,0,0];
常数b2= [6个,0,0,0];

常数a1= [4个,2个,3个,21];
常数a2= [2个,6个,19,7];
```

验证码方法在这种情况下会起作用，但 STARK 生成器会抛出一个异常作为 z 多项式：总积将不等于**1个**.

区别在于 verifyPil 方法不检查两个
选择器是，只检查不为零的情况
(*https://github.com/0xPolygonHermez/pilcom/blob/107765f18d78a865c12517c9c84cbdfbad1d99d0/src/pil_verifier.js#L194-L222*)，但在 STARK 生成中，选择器多项式被内插并用于大
product 的计算，因此我们的示例将在 STARK 生成阶段失败。

同样的问题也出现在 plookup identities 中。

PermutationIdentities 代码：

```
为了(让j=0;j<否;j++) {
  如果(((π.选择==无效的) || （！ F.为零(政客.经验值[π.选择].v_n[j]））) {
    常数值= []
    为了(让k=0;k<π.吨.长度;k++) {
      值.推(F.(政客.经验值[π.吨[k]].v_n[j]));
    }
    常数v=值.加入(","); 吨[v]
    = (吨[v] ||0) +1个;
  }
}

为了(让j=0;j<否;j++) {
  如果(((π.自己==无效的) || （！ F.为零(政客.经验值[π.自己].v_n[j]））) {
    常数值= []
    为了(让k=0;k<π.F.长度;k++) {
      值.推(F.(政客.经验值[π.F[k]].v_n[j]));
    }
    常数v=值.加入(","); 常数成立=吨[
    v】？？ 错误的; 如果 （！ 吨[v]) {

      资源.推(`${π.文件名}:${π.线}: 排列不是 `+(成立===0? '够了':'')+`发现w=${j}价值观： ${v}`); 安慰.日志(资源[资源.长度-
      1个]);
      如果 （！ 配置.继续出错)j=否;// 不继续检查
    }
    别的{
      吨[v] -=1个;
    }
  }
}
```

PlookupIdentities 代码：

```
让吨= {};
    为了(让j=0;j<否;j++) {
        如果(((π.选择==无效的) || （！ F.为零(政客.经验值[π.选择].v_n[j]））) {
            常数值= []
            为了(让k=0;k<π.吨.长度;k++) {
                值.推(F.(政客.经验值[π.吨[k]].v_n[j]));
            }
            吨[值.加入(",")] =真的;
        }
    }


    为了(让j=0;j<否;j++) {
        如果(((π.自己==无效的) || （！ F.为零(政客.经验值[π.自己].v_n[j]））) {
            常数值= []
            为了(让k=0;k<π.F.长度;k++) {
                值.推(F.(政客.经验值[π.F[k]].v_n[j]));
            }
            常数v=值.加入(","); 如果（！
            吨[v]) {
                资源.推(` ${皮尔.查询身份[我].文件名}:${皮尔.查询身份[我].线}: 找不到搜索 w=${j}价值观：
${v}`);
                安慰.日志(资源[资源.长度-1个]);
                如果（！ 配置.继续出错)j=否;// 不继续检查
            }
        }
    }
```

14. VERIFYPIL 工具和 STARK 生成之间的排列约束差异

严重性：信息性的

小路：pilcom/src/pil_verifier.js

整治：当在排列约束中使用选择器时，确保 2 个序列具有相同的长度

地位：菲固定的

描述：

在 PIL 语言中，有一个选择器多项式用于选择特定的行以满足约束。

但是 verifyPIL 方法和 STARK 生成器使用该选择器的方式有所不同。

在排列约束的情况下，2 个序列必须具有相同的长度。我们可以制作 2 个不同长度的序列，通过选择器进行检查。

例如我们有一个 PIL：

```
持续的%N=4个;

命名空间全球的(%N);
  极性常数L1;

命名空间排列示例(%N);

  极性常数b1,b2;// 选择器多项式
  pol提交a1,a2;
  b1{a1}是b2{a2};
```

和相应的多项式：

```
常数b1= [1个,1个,1个,0];

常数b2= [1个,1个,1个,1个];


常数a1= [1个,2个,3个,4个];

常数a2= [4个,3个,2个,1个];
```

在这种情况下，verifyPIL 检查将通过，但 STARK 生成器将抛出一个计数时出现异常**z**多项式。

```
为了(让j=0;j<否;j++) {
  如果(((π.选择==无效的) || （！ F.为零(政客.经验值[π.选择].v_n[j]））) {
    常数值= []
    为了(让k=0;k<π.吨.长度;k++) {
      值.推(F.(政客.经验值[π.吨[k]].v_n[j]));
    }
    常数v=值.加入(",");
    吨[v] = (吨[v] ||0) +1个;
  }
}

为了(让j=0;j<否;j++) {
  如果(((π.自己==无效的) || （！ F.为零(政客.经验值[π.自己].v_n[j]））) {
    常数值= []
    为了(让k=0;k<π.F.长度;k++) {
      值.推(F.(政客.经验值[π.F[k]].v_n[j]));
    }
    常数v=值.加入(",");
    常数成立=吨[v】 ？？ 错误的; 如果
     （！ 吨[v]) {
      资源.推(`${π.文件名}:${π.线}: 排列不是 `+(成立===0？ '够了':'')+`发现w=${j}价值观： ${v}`);
      安慰.日志(资源[资源.长度-1个]);
      如果 （！ 配置.继续出错)j=否;// 不继续检查
    }
    别的{
      吨[v] -=1个;
    }
  }
}
```

15. 调用深度检查丢失

严重性：信息性的

小路：操作码/创建终止上下文.zkasm

整治：在创建操作码的上下文中添加限制检查

地位：菲固定的

描述：

create-terminate-context.zkasm中的操作码对应创建新调用上下文的指令，比如opCALL，
opDELEGATECALL、opSTATICCALL等。根据EVM规范，调用深度需要有1024的限制。

参考黄皮书第37页：

https://ethereum.github.io/yellowpaper/paper.pdf

附加评论：

虽然现在没有检查，但没有可见的安全性
在撰写本文时的影响是调用最多剩余 Gas 的 63/64 的事实，这使得它几乎不可能

达到第 1024 层深度。因此，在这个问题中，状态“已确认”被视为“已修复”。

16. 冗余跳跃

严重性：信息性的

小路：工具.zkasm

整治：考虑删除冗余跳转

地位：菲固定的

描述：

在 utils.zkasm:readPush 过程中，一些无条件跳转是多余的，因为它们跳转到下一个操作，可以删除

干净代码注意事项

utils.zkasm:readPush

```
...
  0=>乙                  :JMP(读推块)

读推块:
...

  A*16777216+C=>C        :JMP(做旋转)

做旋转:
...
  乙-1个=>A              :JMP(旋转循环)

旋转循环:
...
```