hexens × polygon zkEVM

SECURITY REVIEW REPORT FOR POLYGON ZKEVM

CONTENTS

CONTENTS

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

AUDIT LED BY

VAHE KARAPETYAN

Co-founder / CTO | Hexens

Audit Starting Date	Audit Completion Date
17.12.2022	27.02.2023

hexens × polygon zkEVM

METHODOLOGY

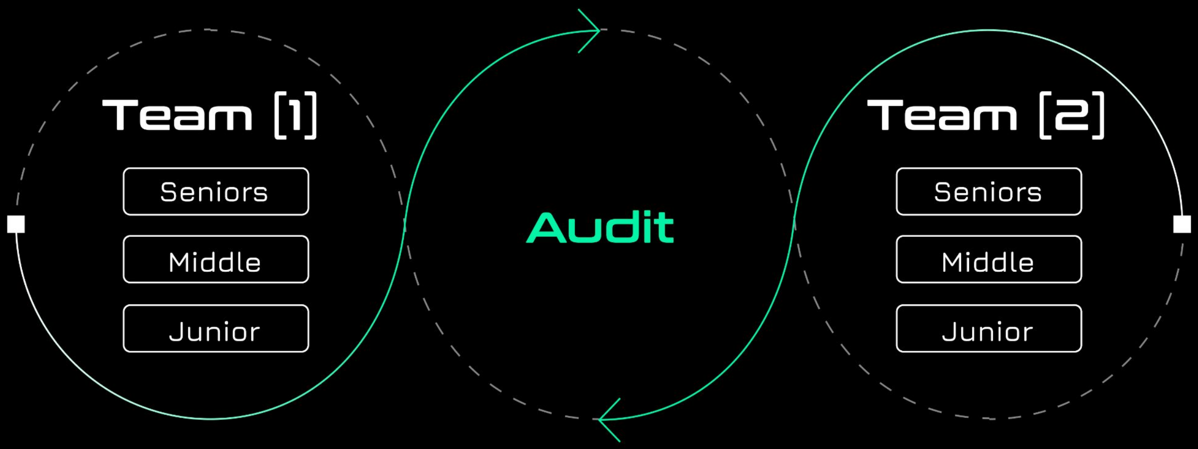
COMMON AUDIT PROCESS

Companies often assign just one engineer to one security assessment with no specified level. Despite the possible impeccable skills of the assigned engineer, it carries risks of the human factor that can affect the product's lifecycle.

Auditor*
Audit

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

IMPACT	PROBABILITY			
	Rare	Unlikely	Likely	Very Likely
Low / Info	Low / Info	Low / Info	Medium	Medium
Medium	Low / Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of vulnerabilities:

CRITICAL

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

HIGH

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

MEDIUM

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

LOW

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

INFORMATIONAL

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

It's important to consider all types of vulnerabilities, including informational ones, when assessing the security of the project. A comprehensive security audit should consider all types of vulnerabilities to ensure the highest level of security and reliability.

EXECUTIVE SUMMARY

OVERVIEW

Polygon zkEVM is a novel technology and is the first EVM-equivalent zero-knowledge scaling solution, where existing smart contracts, developer toolings and wallets can work seamlessly. Polygon zkEVM harnesses zero-knowledge proofs in order to reduce transaction costs and increase throughput, while inheriting the security of Ethereum.

The transactions in zkEVM network (L2) are being compiled into batches, these batches are then sequenced in an Ethereum smart contract and after that their state transitions are being proved and verified on the Ethereum, achieving a trusted state.

The zkEVM has multiple operational layers:

- Network layer: where the Sequencer and Aggregators operate.
- ROM layer: zkEVM utilizes a new language called zkASM to implement EVM, thus making EVM state transactions to be provable.
- Hardware layer: zkEVM utilizes a new language called PIL to create polynomial identities, constraints that guarantee the complete and sound execution of the zkASM ROMs.
- L1 Ethereum smart contracts: bridging assets between the networks and implementing the PoE (Proof of Efficiency) consensus which ensures the correct state transition for the batches.

During the security review Hexens team has covered the most crucial attack surfaces including smart contracts, PIL hardware and zkASM ROMs, unintended discrepancies between EVM and zkEVM and etc.

Due to its complexity and composability the review required wide range of expertise and mindset in different fields of cybersecurity.

We have managed to find critical vulnerabilities in some of the main components, as well as one high severity and mostly minor issues, recommendations.

Also during the audit the team has developed a static analysis tool for PIL language that has helped us throughout the review.

Finally, all of our reported issues are fixed by the development team and were validated by us.

We conclude that the overall security and code quality has increased after completion of the review.

SCOPE

The analyzed resources are located on:
https://github.com/0xPolygonHermez/zkevm-contracts
https://github.com/0xPolygonHermez/zkevm-proverjs
https://github.com/0xPolygonHermez/zkevm-rom
https://github.com/0xPolygonHermez/zkevm-storage-rom

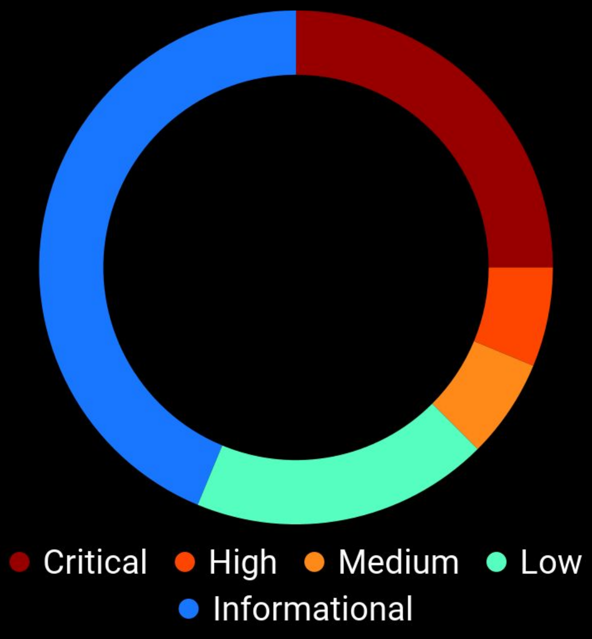
The issues described in the report were fixed in the following commit:
https://github.com/0xPolygonHermez/zkevm-contracts/commit/ec421a4499f07b65d2242f39bb039476ec1cf5e1
https://github.com/0xPolygonHermez/zkevm-proverjs/commit/ab3dbf24172b828e3ff4bbb0238f866199f0c834
https://github.com/0xPolygonHermez/zkevm-rom/commit/2ddeffbed7c022e04032e6d56ed6c6fb14cc38dc
https://github.com/0xPolygonHermez/zkevm-storage-rom/commit/97af71cd372ae6715e818795266d02a5a854cfa6

SUMMARY

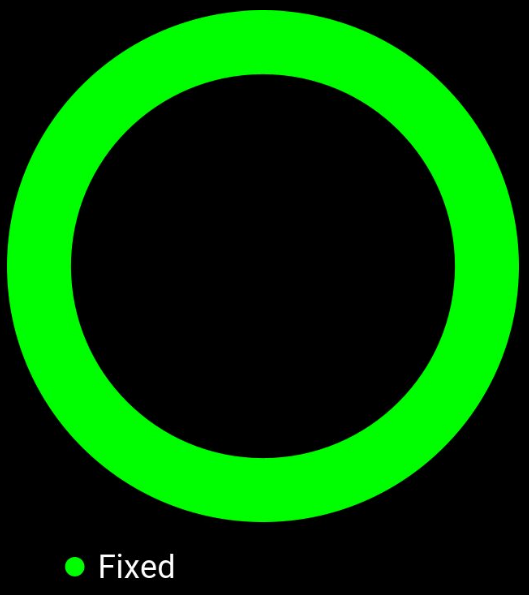
SEVERITY	NUMBER OF FINDINGS
CRITICAL	4
HIGH	1
MEDIUM	1
LOW	3
INFORMATIONAL	7

TOTAL: 16

SEVERITY



STATUS



WEAKNESSES

This section contains the list of discovered weaknesses.

1. ERC777 RE-ENTRANCY ATTACK

SEVERITY: Critical

PATH: PolygonZkEvmBridge.sol

REMEDIATION: implement reentrancy locks for the bridge/claim functions in the bridge contract

STATUS: fixed

DESCRIPTION:

In the function bridgeAsset() any user can bridge his ERC20 tokens specifying the destination network and the destination address, in case it is not a wrapped token it will call transferFrom (SafeERC20.safeTransferFrom) and update the balance of the contract and eventually create a deposit leaf in the Merkle tree. The bridge gives natural opportunity to bridge any kind of ERC20 tokens, also including extended version of ERC20, such as ERC777.

The ERC777 tokens (some examples are Verasity, imBTC, AMP, p.Network tokens and etc, for example Polygon Bridge has these type of tokens even in their controlled list) are type of ERC20 tokens that extend its functionality with a couple of features such as call hooks.

Given this and the fact of how balance calculation is being done in the contract there is a possibility for anyone to drain all of these tokens out of bridge via a subtle reentrancy issue.

Attack breakdown:

The vulnerability arises since the balance update is done the following way:

```
// In order to support fee tokens check the amount received, not the transferred
uint256 balanceBefore = IERC20Upgradeable(token).balanceOf(address(this));
IERC20Upgradeable(token).safeTransferFrom(msg.sender, address(this), amount);
uint256 balanceAfter = IERC20Upgradeable(token).balanceOf(address(this));

// Override leafAmount with the received amount
leafAmount = balanceAfter - balanceBefore;
```

So the leafAmount is calculating the balance difference before and after the call, to be compliant with fee-on-transfer tokens. Although at the first sight it may seem impossible to re-enter in the transferFrom call, since the receiver is the bridge contract and is not controlled by an attacker, there is a lesser known call hook called "ERC777TokensSender" which will be called for the "from" address before the balance has been actually transferred to the destination. In order to register the hook, attacker needs to call setInterfaceImplementer on _ERC1820_REGISTRY (which is a known address in the Ethereum) and register itself as its own "ERC777TokensSender" interface implementer.

At this moment attacking contract is able to receive tokensToSend() callback call, whenever the transferFrom (safeTransferFrom) is being called with "from" set to attackers address.

In order to utilize the reentrancy and generate unfair deposits the attacker needs to re-enter the bridgeAsset() function in the following way: all of the re-entered calls except the last one are called with amount = 0 (the attack is also fully possibly with amount=1 as well, so the zero check would not mitigate it), and only in the last re-entered call it should send some amount of tokens that he wants to amplify; for the sake of simplicity we will take 10**18 (one token).
Such designed reentrancy will work as the **uint256 balanceBefore = IERC20Upgradeable(token).balanceOf(address(this));** will already get set and will not change even if we re-enter in the bridgeAsset call. And the deposit will be amplified by the level of reentrancy.

Reentrancy breakdown:

- bridgeAsset(amount = 0):
- LEVEL = 1, balanceBefore=0:
  - token.safeTransferFrom() -> bridgeAsset(amount = 0):
  - LEVEL = 2, balanceBefore=0
    - token.safeTransferFrom() -> bridgeAsset(amount = 0):
    - LEVEL = 3, balanceBefore=0
      - token.safeTransferFrom() -> bridgeAsset(amount = 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}

It can be seen that all of the reentrancy levels will eventually calculate leafAmount = 10*18 and make a correct deposit and emit the corresponding event (with corresponding depositCount).
Thus the amount deposited in this case will be 3 * (10**18), this can be done for any number of token amount and any number of reentrancy level, in general having deposits for **LEVEL * amount**
Also the attack only needs one transactions to be executed, and it can be done prior to active bridge usage, as the attacker can create the deposit leaves and wait for the right moment to claim them.

2. MISSING CONSTRAINT IN PIL LEADING TO PROVING FAKE INCLUSION IN THE SMT

SEVERITY: Critical

PATH: storage.pil

REMEDIATION: add the missing binary constraint

STATUS: fixed

DESCRIPTION:

The Storage state machine uses SMT (Sparse Merkle Tree) and implements provable CRUD operations in conjunction with Storage ROM.

In order to prove the (Key, Value) inclusion in the SMT, the state machine represents the key as a bit string. It traverses the tree from the root down to the leaf using LSBs (least significant bits) of the key: 0/1 values of the bit correspond to the left/right edge traversal.

As the tree is sparse, the leaf level is not necessarily equal to the key bits length, that means that as soon as the leaf is inserted into the tree, the remaining part of the key (rkey) is being encoded into the leaf value.

The inclusion check algorithm consists of two parts (reference link: *https://wiki.polygon.technology/docs/zkEVM/zkProver/basic-smt-ops*):

1. Checking The Root - is done as a generic Merkel Tree root check by climbing from the leaf to the root using sibling hashes.

2. Checking the Key:

I. In order to check the key, the state machine prepends the next path-bits to the remaining key (rkey), e.g. **rkey||b1||b0** (for the leaf level = 2).

II. By the end of the operation, which can be distinguished in Storage ROM by the LATCH_GET command, the iLatchGet selector is being set to 1.

III. A Permutation constraint is used in the Main SM to ensure that the key passed from the zkEVM ROM matches the one constructed in the step (I):

```
sRD {
    SR0 + 2^32*SR1, SR2 + 2^32*SR3, SR4 + 2^32*SR5, SR6 + 2^32*SR7,
    sKey[0], sKey[1], sKey[2], sKey[3],
    op0, op1, op2, op3,
    op4, op5, op6, op7,
    incCounter
} is
Storage.iLatchGet {
    Storage.oldRoot0, Storage.oldRoot1, Storage.oldRoot2, Storage.oldRoot3,
    Storage.rkey0, Storage.rkey1, Storage.rkey2, Storage.rkey3,
    Storage.valueLow0, Storage.valueLow1, Storage.valueLow2, Storage.valueLow3,
    Storage.valueHigh0, Storage.valueHigh1, Storage.valueHigh2, Storage.valueHigh3,
    Storage.incCounter + 2
};
```

Hence, the part (1) is used to prove that the Value exists in the SMT, and the part (2) is used to prove that the Value is actually binded with the correct Key.

The issue arise as the next-bit polynomial rkeyBit is missing a binary constraint in the Storage SM, as well as the fact that the Storage ROM, pretty naturally, does not assert that the next bit, that comes from a free input, cannot be of any other value than 0 or 1 and uses JMPZ.

storage_sm_get.zkasm:

```
; If next key bit is zero, then the sibling hash must be at the right (sibling's key bit is 1)
${GetNextKeyBit()} => RKEY_BIT
RKEY_BIT                    :JMPZ(Get_SiblingIsRight)
```

Nonetheless, it can not be abused straightforwardly; a number of limitations must be overcome. Due to the specifics of the key reconstruction algorithm and the fact that the value inclusion check in the part (1) needs to hold true simultaneously.

Limitations overview:

In the Storage SM, The key is broken down into four registers: rkey0,..,rkey3 and the path is constructed by cycling the consecutive bits of that registers:

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

(reference link: *https://wiki.polygon.technology/docs/zkEVM/zkProver/construct-key-path*)

Thus, in order to reconstruct the key from the bits, the corresponding rkey polynomial needs to be prepended with that bit:

rkey[level % 4] ||= rkeyBit

It is important to mention that the key is actually the POSEIDON hash of the account address, storage slot and the query key.

In order to avoid **modulo 4** operation, the Storage SM introduces LEVEL register, which also consists of 4 parts: level0,..,level3 and the ROTATE_LEVEL opcode in the Storage ROM.

The LEVEL is firstly set to **leaf_level % 4**, and then ROTATE_LEVEL is used every time the prover needs to climb the tree:

storage-sm-get.zkasm:

```
; Update remaining key
                              :ROTATE_LEVEL
                              :CLIMB_RKEY


                              :JMP(Get_ClimbTree)
```

Level rotation(storage.pil):

```
pol rotatedLevel0 = iRotateLevel*(level1-level0) + level0;
pol rotatedLevel1 = iRotateLevel*(level2-level1) + level1;
pol rotatedLevel2 = iRotateLevel*(level3-level2) + level2;
pol rotatedLevel3 = iRotateLevel*(level0-level3) + level3;
```

Finally, the rkey will be modified when using the CLIMB_RKEY opcode

storage.pil:

```
pol climbedKey0 = (level0*(rkey0*2 + rkeyBit - rkey0) + rkey0);
pol climbedKey1 = (level1*(rkey1*2 + rkeyBit - rkey1) + rkey1);
pol climbedKey2 = (level2*(rkey2*2 + rkeyBit - rkey2) + rkey2);
pol climbedKey3 = (level3*(rkey3*2 + rkeyBit - rkey3) + rkey3);
```

In order to hold the above-mentioned permutation constraint all of the rkey0-3 registers must be modified by the end of the operation (when the LATCH_GET opcode is used). As well as the Storage ROM will be rearranging the left/right node hashes by matching the next-bit. Given the fact that one can only abuse the non-zero values of the next-bit, the limitations can be overcome by inserting arbitrary Value into the SMT with a Key that has **1111** as its LSB:.

Key = ***1111, this is needed to have the opportunity to change all 4 rkey registers.

This means that the POSEIDON hash that is derived from the account address and the storage slot number (in addition to the storage query key), needs to have the least significant bits of its result registers hash0,..,hash3 set as 1:

```
hash0 = ***1
hash1 =  ***1
hash2 = ***1
hash3 = ***1
```

As only 1 bit of every POSEIDON hash register is fixed, it is a trivial task to overcome the 4-bit entropy and find a storage slot (for any given account address) to meet the attack prerequisites.

Another limitation is that the leaf being inserted must have a level greater than 4, in the real-world scenario this is guaranteed to be the case (with a negligible opposite probability) as there will be millions of leafs inserted into the tree. Even if it's not the case the attacker will only need to precompute two storage slots, following the same rule, and insert them both to guarantee the minimal level.

After inserting (**KeyPrecomputed, ValueArbitrary**) into the SMT using the opSSTORE procedure, and thus fulfilling the prerequisites the attacker can fake the binding of any key **KeyToFake** with the value **ValueArbitrary**, by setting the last 4 next-bit values from the free input to:

```
rkeyBit[0] =  rkeyToFake[0] - rkey0*2
rkeyBit[1] =  rkeyToFake[1] - rkey1*2
rkeyBit[2] =  rkeyToFake[2] - rkey2*2
rkeyBit[3] =  rkeyToFake[3] - rkey3*2
```

As the Storage ROM uses JMPZ to distinguish the climb path, despite of being greater than 1 the rkeyBit will be treated the same way as if it was set to 1, and the root check (Value inclusion) will successfully be bypassed.

The main impact that will favour the attacker will be to fake the inclusion of (**KeyAttackerBalance, ArbitraryAmount**) in the SMT.

3. INCORRECT CTX ASSIGNATION LEADING TO ADDITION OF RANDOM AMOUNT OF ETHER TO THE SEQUENCER BALANCE

SEVERITY: Critical

PATH: process-tx.zkasm, precompiled/identity.zkasm

REMEDIATION: change the identity.zkasm code to save the originCTX in a register before jumping to the handleGas label

STATUS: fixed

DESCRIPTION:

The zkEVM ROM architecture uses Contexts (CTX) to divide and emulate virtual address to physical address translation between call contexts inside one transaction. One CTX address space is used to determine the dynamic memory space that changes between call contexts, as well as the stack and CTX variables (such as msg.sender, msg.value, active storage account and etc.). The context switch is done using auxiliary variables such as originCTX, which refers to the origin CTX that created the current context as well as currentCTX. There is a special CTX(0) that is used for storing the GLOBAL variables such as tx.origin or old state root, the first context a batch transaction starts with is CTX(1), and it increments as new calls, context switches, or transactions are being processed.

The vulnerability lies in the "identity" (0x4) precompiled contract implementation. In case there is no originCTX set, as that effectively means that the EOA is directly calling the precompiled contract and not within an inner contract call, the precompiled contracts should consume intrinsic gas and end transaction execution. Although the context switching is done correctly in the ecrecover (0x1) precompile, the identity precompile is erroneous in its context switching. To check that the transaction is calling the contract directly, is utilizes originCTX variable and checks whether it is equal to 0:
https://github.com/0xPolygonHermez/zkevm-rom/blob/develop/main/precompiled/identity.zkasm#L21

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

Although it immediately loads the originCTX into the CTX register, all of the memory operations will be done for the CTX(0).

As the context switch between GLOBAL and CTX contexts is done via useCTX:
https://github.com/0xPolygonHermez/zkevm-proverjs/blob/develop/pil/main.pil#LL203-L204C85

```
pol addrRel = ind*E0 + indRR*RR + offset;
pol addr = useCTX*CTX*2^18 + isStack*2^16 + isStack*SP + isMem*2^17+ addrRel;
```

GLOBAL -> useCTX = 0, CTX -> useCTX = 1

Effectively the final address will be the same if the CTX register is set to 0. Given that the variables are addressed by their offset, the ROM's global variables will be double-referenced by their appropriate CTX variables with the same offset.

Example:

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

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

...

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

Thus colliding the GLOBAL and CTX variable offsets.

The attack breakdown:

- User (EOA) creates a transaction with a destination address set to identity precompiled contract (0x4)
- When the execution reaches

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

  the CTX will be set to 0, and a jump will be done to the handleGas label.

The handleGas will check the refund (an important detail is that in the current VAR configuration, the gasRefund variable collides with nextHashPId, which will be 0 in this case, although if it were to collide with another VAR that has bigger absolute values, then the caller will have opportunity to "print money out of thin air" for himself), after refunding the sender it continues to a point where it needs to account the gas consumed to the sequencer address.

```
;; Send gas spent to sequencer
sendGasSeq:
        $ => A          :MLOAD(txGasLimit)
        A - GAS => A

        $ => B          :MLOAD(txGasPrice)
        ; Mul operation with Arith
        A               :MSTORE(arithA)
        B               :MSTORE(arithB), CALL(mulARITH)
        $ => D          :MLOAD(arithRes1) ; value to pay the sequencer in D
```

As the txGasLimit references the oldStateRoot, which is the hash of the state tree and has a very big absolute value, the **MLOAD(txGasLimit)** will return the oldStateRoot value instead. By setting a gasPrice to 1 (or an arbitrarily small value not to overflow the multiplication), the sequencer will be credited with an enormously big balance.

The attack requirements and probability:

For any user to be able to credit himself a very big ether balance, he needs to be the one sequencing it. The most convenient way to do so is to force a batch in L1 PolygonZkEVM contract.

As in the current configuration, the trusted sequencer ignores the forced batches; it stores them in separate state.forced_batch table in the DB: *https://github.com/0xPolygonHermez/zkevm-node/blob/develop/state/pgstatestorage.go#L316-L32*

And when the sequencer will query for the pending batches to be sequenced in the getSequencesToSend() function:
https://github.com/0xPolygonHermez/zkevm-node/blob/develop/sequencer/sequencesender.go#L114

It will only query for the batches from state.batch table:
https://github.com/0xPolygonHermez/zkevm-node/blob/develop/state/pgstatestorage.go#L535-L539

Thus the attacker will need to force a batch and then to wait for the timeout period to pass and sequence it, setting the sequencer to arbitrary address. In the current configuration the attack gives opportunity to anyone to force such batch and after the timeout period to be credited with an enormous ether balance, if combined with obfuscating the transaction with other "dummy" transactions and adding a bridgeAsset() call somewhere in the same batch, the attacker will gain a deposit leaf of arbitrary ether amount as soon as the batch is verified and can drain all of the ether held by the bridge.

4. MISSING CONSTRAINT IN PIL LEADING TO EXECUTION FLOW HIJACK

SEVERITY: Critical

PATH: utils.zkasm, main.pil

REMEDIATION: add a constraint for the inNeg polynomial to ensure that it evaluates only to 0 or 1. e.g. isNeg * (1-isNeg) = 0

STATUS: fixed

DESCRIPTION:

The combination of the free input checking in zkEVM ROM and a missing constraint in main.pil leads to execution hijack with a possibility to jump to an arbitrary address in ROM.
One of the impacts is the arbitrary increase of balance for any caller.

In the file utils.zkasm some of the procedures use free input calls to make small calculations, for example, computeSendGasCall:

```
; C = [c7, c6, ..., c0]
; JMPN instruction assures c0 is within the range [0, 2^32 - 1]
${GAS >> 6} => C        :JMPN(failAssert)
${GAS & 0x3f} => D

; since D is assured to be less than 0x40
; it is enforced that [c7, c6, ..., c1] are 0 since there is no value multiplied by 64
; that equals the field
; Since e0 is assured to be less than 32 bits, c0 * 64 + d0 could not overflow the field
C * 64 + D              :ASSERT
```

In such cases, to ensure the validness of the free input JMPN is used. JMPN will effectively check whether the free input set in register C is in the range [0,2^{32}-1]. This is a security assumption that ensures that the register is not overflowing in the assertion stage:

```
C * 64 + D              :ASSERT
```

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

```
pol jmpnCondValue = JMPN*(isNeg*2^32 + op0);
```

By checking that the jmpnCondValue is a 32-bit number we assure that the op0 is in the [-2^{32},2^{32}) range, thus preventing the overflow. The jump destination, as well as the zkPC constraints, are consequently based on the inNeg:
https://github.com/0xPolygonHermez/zkevm-proverjs/blob/develop/pil/main.pil#L322-L336

```
zkPC' = doJMP * (finalJmpAddr - nextNoJmpZkPC) + elseJMP * (finalElseAddr -
nextNoJmpZkPC) + nextNoJmpZkPC;
```

Nonetheless, a constraint is missing to ensure that isNeg evaluates only to 1 or 0. In the case of utils.zkasm procedures, there is no elseAddr specified, and having:

```
finalElseAddr = nextNoJmpZkPC
doJMP = isNeg
elseJMP = (1-isNeg)
```

The zkPC constraint can be reduced to:

```
zkPC' = isNeg * (finalJmpAddr - nextNoJmpZkPC) + nextNoJmpZkPC
```

Where both finalJmpAddr and nextNoJmpZkPC are known values in the ROM program compilation phase.

In order to be able to jump to arbitrary zkPC, the attacker needs to calculate corresponding values for isNeg and op0; this can be done using derived formulas:

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

op0 = - isNeg * 2^{32} mod P

The attack breakdown:

At this point attacker has a primitive to jump to an arbitrary address; the next step will be to find a suitable gadget to jump to, the main requirements for the target are to:

- Not to corrupt/revert zkEVM execution
- Impact favourably for the attacker

One of the jump chains found is to use one of the *CALL opcodes as the start of the attack chain to call the computeSendGasCall and subsequently craft a jump into the refundGas label's code:
https://github.com/0xPolygonHermez/zkevm-rom/blob/develop/main/process-tx.zkasm#L496-L498

```
$ => A                    :MLOAD(txSrcOriginAddr)
0 => B,C                   ; balance key smt
$ => SR                    :SSTORE
```

This will set txSrcOriginAddr balance to the value contained in register D and finish the transaction execution. To abuse the value set by the SSTORE instruction, attacker needs to set huge value in the register D, for this the DELEGATECALL opcode can be used, as in the implementation it sets the register D just before the computeSendGasCall call:

```
$ => D        :MLOAD(storageAddr)
...
E            :MSTORE(retCallLength), CALL(computeGasSendCall); in: [gasCall: gas sent to call] out: [A: min(
requested_gas , all_but_one_64th(63/64))]
```

So the D register will be set with storageAddr which has very big absolute value.

Additional setup for the attack:

- A contract should be deployed with a function (or fallback) that initiates a delegatecall() call to any address.
- The transaction should be initiated with gasPrice set to 0 not to overflow in gas when sending it to the sequencer, as well as this will favor the fact of attacker getting to prove the batch initiated.
- The gasLimit should be precalculated to end up with 0 gas at the end of the execution, this is done for the same reason mentioned above

5. BUG IN MAXMEM HANDLING CAN HALT THE BATCH VERIFICATION

SEVERITY: High

PATH: main.zkasm, create-terminate-context.zkasm

REMEDIATION: one of the approaches to remediate the issue will be to change the diffMem plookup with BITS17, or to shrink the MAX_MEM_EXPANSION_BYTES to 2^{21}-32, so that the biggest value for MAXMEM could be 2^{16}-1

STATUS: fixed

DESCRIPTION:

In the zkEVM ROM the MAXMEM register that is used to set the biggest offset of memory used is being set only once to zero - for the first step of the execution trace.

The contracts that will call MLOAD,MSTORE EVM-opcodes will alter the MAXMEM if the memory referenced has higher relative address than the one currently set:

```
pol addrRel = ind*E0 + indRR*RR + offset;

pol maxMemRel = isMem * addrRel;

pol maxMemCalculated = isMaxMem*(addrRel - MAXMEM) + MAXMEM;

MAXMEM' = setMAXMEM * (op0 - maxMemCalculated) + maxMemCalculated;
```

There is a plookup check for the "difference" of the current MAXMEM and the relative address:

```
pol diffMem = isMaxMem* ( (maxMemRel - MAXMEM) -
                (MAXMEM - maxMemRel) ) +
        (MAXMEM - maxMemRel);
isMaxMem * (1 - isMaxMem) = 0;
...
diffMem in Global.BYTE2;
```

This effectively checks that the difference is not a negative number, as well as that the isMaxMem is correctly set, and also it restrains the the difference to be: |maxMemRel - MAXMEM| < 2^{16}

On the other hand zkEVM ROM has a limit check for the relative memory offset done in the **utils.zkasm:saveMem** procedure:

```
$ => B                  :MLOAD(lastMemOffset)
; If the binary has a carry, means the mem expansion is very big. We can jump to oog directly
; offset + length in B
$ => B                  :ADD, JMPC(outOfGas)
; check new memory length is lower than 2**22 - 31 - 1 (max supported memory expansion for
%TX_GAS_LIMIT of gas)
%MAX_MEM_EXPANSION_BYTES => A
$                       :LT,JMPC(outOfGas)
```

Although the difference is that the **MAX_MEM_EXPANSION_BYTES** is equal to 2^{22} - 32. As the offset that the ROM effectively uses as the relative address will be the offset/32, this means that the contract can push the memory offset to the maximum of (2^{22}-32) / 32 = 2^{17} - 1.

Since the diffMem should be in BYTE2 range, attacker will need two MLOAD or MSTORE operations to push the MAXMEM to the value bigger than BYTE2 range, e.g. **opMSTORE(1000) + opMSTORE(2^{16} + 999)**.

As the MAXMEM is never reset and the **diffMem in Global.BYTE2;** does not have a selector - for the memory operations with GLOBAL/CTX or stack variables the maxMemRel will be equal to 0 (as the isMem will be 0):

```
pol maxMemRel = isMem * addrRel;
```

This will mean that the diffMem:

```
pol diffMem = isMaxMem* ( (maxMemRel - MAXMEM) -
                        (MAXMEM - maxMemRel) ) +
              (MAXMEM - maxMemRel);
```

will either be equal to **maxMemRel - MAXMEM** -- a negative value or **MAXMEM - maxMemRel** which will be > 2^{16} -1, so the consecutive plookup with BYTE2 will not be satisfied.

This gives ability to any illicit actor to send a transaction to the trusted sequencer or force it, and as soon as the batch will be sequenced it will be impossible to prove the next state transition.

6. INCORRECT LIMIT CHECK IN ZKASM ECRECOVER IMPLEMENTATION

SEVERITY: Medium

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

REMEDIATION: the FNEC_DIV_TWO should be equal to 57896044618658097711785492504343953926418782139537452191302581570759080747168 (Fp/2)

STATUS: fixed

DESCRIPTION:

In the zkASM implementation of the ecrecover function, there must be a check against ECDSA signature malleability. The check should be done when the transaction's signature is verified and is omitted when the precompiled ecrecover is called. For ECDSA signature not to be malleable, the S value should not be greater than Fp/2 (S <= Fp/2); Fp/2 = **57896044618658097711785492504343953926418782139537452191302581570759080747168** This check is implemented in EVM (Go-ethereum) *https://github.com/ethereum/go-ethereum/blob/f53ff0ff4a68ffc56004ab1d5cc244bcb64d3277/crypto/crypto.go#L268-L270*

Although in case of zkASM the ecrecover checks againt Fp/2 + 1 (**57896044618658097711785492504343953926418782139537452191302581570759080747169**), instead of Fp/2, thus the allowed range of S values also includes Fp/2 + 1. This discrepancy can be abused to generate proof for transactions which do not comply with EVM.

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

    ; s in [1, ecrecover_s_upperlimit]
    $ => A      :MLOAD(ecrecover_s_upperlimit)
    $ => B      :MLOAD(ecrecover_s)
    $           :LT,JMPC(ecrecover_s_is_too_big)
    0n => A
    $           :EQ,JMPC(ecrecover_s_is_zero)
```

7. DISCREPANCY IN TRANSACTION RLP DECODING BETWEEN ZKEVM AND EVM

SEVERITY: Low

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

REMEDIATION: add checks for string and list decoding to ensure that long-format encoding is done for sizes more than 55 bytes

STATUS: fixed

DESCRIPTION:

In the load-tx-rlp.zkasm implementation of transaction RLP decoding the code label dataREAD stands for the part of decoding the DATA field of the transaction. The data field is encoded as an RLP string and can be of both short and long size, as the RLP format states (*https://ethereum.org/en/developers/docs/data-structures-and-encoding/rlp#definition*) the string can be represented in two ways based on the length (0-55 bytes and 55+ bytes):

- ...
- *Otherwise, if a string is 0-55 bytes long, the RLP encoding consists of a single byte with value 0x80 (dec. 128) plus the length of the string followed by the string. The range of the first byte is thus [0x80, 0xb7] (dec. [128, 183]).*
- *If a string is more than 55 bytes long, the RLP encoding consists of a single byte with value 0xb7 (dec. 183) plus the length in bytes of the length of the string in binary form, followed by the length of the string, followed by the string. For example, a 1024 byte long string would be encoded as \xb9\x04\x00 (dec. 185, 4, 0) followed by the string. Here, 0xb9 (183 + 2 = 185) as the first byte, followed by the 2 bytes 0x0400 (dec. 1024) that denote the length of the actual string. The range of the first byte is thus [0xb8, 0xbf] (dec. [184, 191]).*
- ...

The problem arises when one wants to construct an RLP transaction with a short data field (less than 55 bytes) but encode it in long string format (0xb801AA = ["AA"], example of 1 byte string but in long string format). The EVM has these checks implemented to avoid such situations:
https://github.com/ethereum/go-ethereum/blob/master/rlp/decode.go#L1008-L1011 and consequently, as the zkEVM-node inherits the lib, it has the checks as well.
However, the zkEVM ROM's RLP decoding mechanism fails to check these cases (both for short/long strings and short/long lists).

A guaranteed impact is that the batch sequencing mechanism will be blocked and broken because of this “poison” transaction. Possible steps will be:

1. The attacker forces a batch with wrongful RLP encoding
2. The trusted sequencer either sequences it or ignores and the timeout period passes
3. The trusted sequencer or the attacker sequences the forced batch
4. The attacker verifies the batch as it is fully provable by the zkEVM ROM and claims from the bridge or uses the assets which were transferred by the poison transaction
5. The zkEVM network halts as synchronizers cannot sync with the forced batch, and the network stays desynced
6. The possible response steps to the situation:
    1. Fix the RLP decoding in the zkEVM ROM and rollback the state in L1 by redeploying the Proof-of-Efficiency contract with older state roots
    2. Change the RLP decoding to a wrongful one in the nodes and push an update for the network with incorrect RLP

As the most probable scenario is to go with 6.a solution, rather than not actually fixing the bug and introducing a new one in the nodes (6.b), the situation will favour the attacker as after the rollback, there will be possible to basically double spend the assets.

```
dataREAD:
        $ => D                      :MLOAD(batchHashPos)
        D                          :MSTORE(dataStarts)
        1 => D
        %CALLDATA_OFFSET => SP        :CALL(addHashTx)
                                  :CALL(addBatchHashData)
        A - 0x80                    :JMPN(veryShortData)
        A - 0x81                    :JMPN(endData)
        A - 0xb8                    :JMPN(shortData)
        A - 0xc0                    :JMPN(longData, invalidTxRLP)

veryShortData:
        1                          :MSTORE(txCalldataLen)
        31 => D                     :CALL(SHLarith)
        A                          :MSTORE(SP++), JMP(endData)

shortData:
        $ => D                      :MLOAD(batchHashPos)
        D                          :MSTORE(dataStarts)
        A - 0x80 => B                :MSTORE(txCalldataLen), JMP(readData)

longData:
        A - 0xb7 => D                :CALL(addHashTx)
                                  :CALL(addBatchHashData)
        $ => D                      :MLOAD(batchHashPos)
        D                          :MSTORE(dataStarts)
        A => B                      :MSTORE(txCalldataLen)

readData:
        ; check binaries
        32 => D
        B - D                       :JMPN(readDataFinal)
        B - D                       :MSTORE(txDataRead), CALL(addHashTx)
        A                          :MSTORE(SP++), CALL(addBatchHashByteByByte)
        $ => B                      :MLOAD(txDataRead), JMP(readData)

readDataFinal:
        B - 1                       :JMPN(endData)
        B => D                      :CALL(addHashTx)
        32 - D => D                  :CALL(SHLarith)
        A                          :MSTORE(SP)
        32 - D => D                  :CALL(addBatchHashByteByByte)
```

Evm:

```
case b < 0xC0:
  // If a string is more than 55 bytes long, the RLP encoding consists of a
  // single byte with value 0xB7 plus the length of the length of the
  // string in binary form, followed by the length of the string, followed
  // by the string. For example, a length-1024 string would be encoded as
  // 0xB90400 followed by the string. The range of the first byte is thus
  // [0xB8, 0xBF].
  size, err = s.readUint(b - 0xB7)
  if err == nil && size < 56 {
     err = ErrCanonSize
  }
  return String, size, err
```

8. GASLIMIT AND CHAINID MAX SIZE DIFFERENCE BETWEEN ZKEVM AND EVM

SEVERITY: Low

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

REMEDIATION: change Gas Limit and Chain ID RLP decoding according to the EVM's Gas Limit and Chain ID max sizes

STATUS: fixed

DESCRIPTION:

There is a difference between the transaction's gas limit and chain id max sizes between zkEVM and EVM implementations.

In
https://github.com/0xPolygonHermez/zkevm-rom/blob/develop/main/load-tx-rlp.zkasm GasLimit's maximal size is defined as 256 bit and ChainID's 64 bit.
The EVM sizes are:
Gas limit's maximal size is 64 bit
https://github.com/ethereum/go-ethereum/blob/79a478bb6176425c2400e949890e668a3d9a3d05/core/types/tx_legacy.go#L29 and ChainID's maximal size is 256 bit

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

A possible impact is we can create a transaction with the correct chainID but encoded as a larger uint which is acceptable by zkEVM nod as well as EVM (RLP lib), but it will be impossible to prove a batch including that transaction. Given the fact that the zkASM EVM will be failing in the load-rlp stage, the batch will be discarded, and also the fact that there is a possibility to freely spam these kinds of transactions to the sequencer, it will be possible to stop the network availability.

zkASM ROM:

```
;; Read RLP 'gas limit'
    ; 256 bits max
gasLimitREAD:
    1 => D                  :CALL(addHashTx)
                          :CALL(addBatchHashData)
    A - 0x80                :JMPN(endGasLimit)
    A - 0x81                :JMPN(gasLimit0)
    A - 0xa1                :JMPN(shortGasLimit, invalidTxRLP)
```

```
;; Read RLP 'chainId'
    ; 64 bits max
chainREAD:
    1 => D                  :CALL(addHashTx)
                          :CALL(addBatchHashData)
    A - 0x80                :JMPN(endChainId)
    A - 0x81                :JMPN(chainId0)
    A - 0x89                :JMPN(shortChainId, invalidTxRLP)
```

EVM Transaction:

```
// LegacyTx is the transaction data of regular Ethereum transactions.
type LegacyTx struct {
  Nonce    uint64          // nonce of sender account
  GasPrice *big.Int        // wei per gas
  Gas      uint64          // gas limit
  To       *common.Address `rlp:"nil"` // nil means contract creation
  Value    *big.Int        // wei amount
  Data     []byte          // contract invocation input data
  V, R, S  *big.Int        // signature values
}
```

```
// TxData is the underlying data of a transaction.
//
// This is implemented by DynamicFeeTx, LegacyTx and AccessListTx.
type TxData interface {
  txType() byte // returns the type ID
  copy() TxData // creates a deep copy and initializes all fields

  chainID() *big.Int
  accessList() AccessList
  data() []byte
  gas() uint64
  gasPrice() *big.Int
  gasTipCap() *big.Int
  gasFeeCap() *big.Int
  value() *big.Int
  nonce() uint64
  to() *common.Address

  rawSignatureValues() (v, r, s *big.Int)
  setSignatureValues(chainID, v, r, s *big.Int)
}
```

9. RECOMMENDATION TO CHANGE A CONDITIONAL JUMP

SEVERITY: Low

PATH: /main/opcodes/block.zkasm

REMEDIATION: consider using LT opcode and JMPC conditional jump instead of JMPN; in that case, the txCount should be incremented with ADD opcode

STATUS: fixed

DESCRIPTION:

While analysing the zkEVM ROM opcodes and the appropriate PIL representation in the state machines it has been noticed that some of the conditional jumps differ in their operational register size:

- *JMPN (jump negative)*

```
pol jmpnCondValue = JMPN*(isNeg*2**32 + op0);
```

- *JMPZ (jump zero) and JMPNZ (jump not zero)*

```
/// op0 check zero
pol commit op0Inv;
pol op0IsZero = 1 - op0*op0Inv;
op0IsZero*op0 = 0;
...
pol doJMP = JMPN*isNeg + JMP + JMPC*carry + JMPZ*op0IsZero + return + call;
```

It can be seen that the JMPN/Z/NZ opcodes consider the op0 register only, unlike other conditional jumps such as JMPC,JMPNC, which use binary state machine carry latch, which operates on 256-bit values.

As a result, these type of jumps will consider only the lower part of the registers that have 8-slot (A,B,C,...):

```
A   :JMPZ(someLabel)
```

The jump will happen as long as A0 = 0, even for A = [A0 = 0, A1 != 0 ,...,A7 != 0], for example if A = 2^{32}. This holds true as op0=A0,op1=A1,...op7=A7 and only the op0 will be used in the jump condition.

We have iteratively looked through all of the JMPN,JMPZ,JMPNZ calls done throughout all of the zkEVM ROM in order to find the ones that use 8-slot registers and don't have other limitations of the register's value (e.g., to be smaller than 2^{32}).

After multiple iterations, the only case found where the register can hold values bigger than 2^{32} is in the opBLOCKHASH:

```
opBLOCKHASH:
...
  ; Get last tx count
  $ => B         :MLOAD(txCount)
  $ => A         :MLOAD(SP) ; [blockNumber => A]
  ; Check batch block is lt current block number, else return 0
  B - A - 1      :JMPN(opBLOCKHASHzero)
```

Here the JMPN can be incorrectly triggered in the scenario where the txCount is greater than 2^{32}; this is probable to happen, although not in the near future, as the blocks in zkEVM network actually represent transactions (hence the reason why txCount is being used), and with an average expectation of 75 TPS it will overlap 32-bit values after 1.5-2 years. As this is a very prolonged scenario, the severity of the issue is lowered.

Nonetheless, if this happens, the BLOCKHASH instruction for older blockNumbers will start returning 0 instead of their actual values.

10. LOOP OPTIMISATION

SEVERITY: Informational

PATH: PolygonZkEvm.sol

REMEDIATION: the totalBatchesAboveTarget can be incremented by currentBatch - currentLastVerifiedBatch and after that the loop should break

STATUS: fixed

DESCRIPTION:

In the function updateBatchFee the function calculates number of blocks that were verified above and below the verification time target. The fee is updated based corresponding to their ratio.

The loop that checks for targets that are above the target does a backward loop from the newly verified batch up to the last verified batch, it does it until it reaches the last verified batch traversing all of the sequences in between. The loop can be optimised as the sequencedTimestamp's are strictly growing and from the first time that a sequence is found that is above the target time, the rest of the batches up until the last verified batch can be considered to be verified above the target (as every next sequencedTimestamp will be smaller than the current).

An optimisation will be to account all of the batches up until the last verified batch and break out of the loop as soon as such a sequence is met.

PolygonZkEvm.sol

```
function _updateBatchFee(uint64 newLastVerifiedBatch) internal {
   uint64 currentLastVerifiedBatch = getLastVerifiedBatch();
   uint64 currentBatch = newLastVerifiedBatch;uint256 totalBatchesAboveTarget;
uint256 newBatchesVerified = newLastVerifiedBatch -
   currentLastVerifiedBatch;

while (currentBatch != currentLastVerifiedBatch) {
   // Load sequenced batchdata
   SequencedBatchData
       storage currentSequencedBatchData = sequencedBatches[
           currentBatch
       ];

   // Check if timestamp is above or below the VERIFY_BATCH_TIME_TARGET
   if (
       block.timestamp - currentSequencedBatchData.sequencedTimestamp >
       veryBatchTimeTarget
   ) {
       totalBatchesAboveTarget +=
           currentBatch -
           currentSequencedBatchData.previousLastBatchSequenced;
   }

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

11. INCORRECT INDEX SIZE IN VERIFYMERKLEPROOF

SEVERITY: Informational

PATH: DepositContract.sol:L90-L112

REMEDIATION: the index should be checked to be lower than 2**_DEPOSIT_CONTRACT_TREE_DEPTH (since the leaf traversing will be done for _DEPOSIT_CONTRACT_TREE_DEPTH levels)

STATUS: fixed

DESCRIPTION:

In the contract DepositContract.sol

The index is of size uint64, and as the merkle tree levels are 32, the most significant bits of the index can be manipulated to double spend with the same index.

As the functions calling this context are using correct size of index (uint32) this issue is with lowered severity, although potentially this can bring to critical impact attack vectors if the code will be refactored and mislead by the parameter's uint64 size.

```
function verifyMerkleProof(
      bytes32 leafHash,
      bytes32[] memory smtProof,
      uint64 index,
      bytes32 root
  ) public pure returns (bool) {
      bytes32 node = leafHash;

      // Check merkle proof
      uint256 currrentIndex = index;
      for (
         uint256 height = 0;
         height < _DEPOSIT_CONTRACT_TREE_DEPTH;
         height++
      ) {
         if ((currrentIndex & 1) == 1)
            node = keccak256(abi.encodePacked(smtProof[height], node));
         else node = keccak256(abi.encodePacked(node, smtProof[height]));
         currrentIndex /= 2;
      }

      return node == root;
  }
```

12. REDUNDANT IMPORTS

SEVERITY: Informational

PATH: PolygonZkEVM.sol

REMEDIATION: remove the redundant imports

STATUS: fixed

DESCRIPTION:

The imports for **ERC20BurnableUpgradeable.sol** and **Initializable.sol** are redundant as they are either not used or already imported recursively.

```
import "@openzeppelin/contracts-upgradeable/token/ERC20/utils/SafeERC20Upgradeable.sol";
//import
"@openzeppelin/contracts-upgradeable/token/ERC20/extensions/ERC20BurnableUpgradeable.
sol";   // redundant import
import "./interfaces/IVerifierRollup.sol";
import "./interfaces/IPolygonZkEVMGlobalExitRoot.sol";
//import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol"; // redundant
import
import "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";
import "./interfaces/IPolygonZkEVMBridge.sol";
import "./lib/EmergencyManager.sol";
```

13. PLOOKUP AND PERMUTATION SELECTOR POLYNOMIAL DISCREPANCY BETWEEN VERIFYPIL TOOLING AND STARK GENERATION

SEVERITY: Informational

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

REMEDIATION: verifyPil method must check whether two selectors are equal

STATUS: fixed

DESCRIPTION:

In PIL language there is a selector polynomial for choosing specific lines to meet the constraints.

But there is differences how verifyPIL method and STARK verifier uses that selector. For verifyPIL polynomial trace's lines are being chosen in case of the selectors are not 0 and can be not equal.

For example we have a PIL:

```
constant %N = 4;

namespace Global(%N);
  pol constant L1;

namespace PermutationExample(%N);

  pol constant b1, b2; // selector polynomials
  pol commit a1, a2;
  b1{a1} is b2{a2};
```

and corresponding polynomials:

```
const b1 = [0,5,0,0];
const b2 = [6,0,0,0];

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

verifyPil method will work in this case, but STARK generator throws an exception as the z polynomial: the grand product will not be equal to 1.

The difference arises as the verifyPil method doesn't check whether two selectors are, only checks not zero case (*https://github.com/0xPolygonHermez/pilcom/blob/107765f18d78a865c12517c9c84cbdfbad1d99d0/src/pil_verifier.js#L194-L222*) , but in the STARK generation selector polynomials are being interpolated and used in grand product's calculations, so our example will fail on STARK generation phase.

The same issues arises also for plookup identities.

PermutationIdentities code:

```
for (let j=0; j<N; j++) {
  if ((pi.selT==null) || (!F.isZero(pols.exps[pi.selT].v_n[j]))) {
     const vals = []
     for (let k=0; k<pi.t.length; k++) {
        vals.push(F.toString(pols.exps[pi.t[k]].v_n[j]));
     }
     const v = vals.join(",");
     t[v] = (t[v] || 0) + 1;
  }
}

for (let j=0; j<N; j++) {
  if ((pi.selF==null) || (!F.isZero(pols.exps[pi.selF].v_n[j]))) {
     const vals = []
     for (let k=0; k<pi.f.length; k++) {
        vals.push(F.toString(pols.exps[pi.f[k]].v_n[j]));
     }
     const v = vals.join(",");
     const found = t[v] ?? false;
     if (!t[v]) {
        res.push(`${pi.fileName}:${pi.line}:  permutation not `+(found === 0 ? 'enought ':'')+`found w=${j} values: ${v}`);
        console.log(res[res.length-1]);
        if (!config.continueOnError) j=N;  // Do not continue checking
     }
     else {
        t[v] -= 1;
     }
  }
}
```

PlookupIdentities code:

```
let t = {};
      for (let j=0; j<N; j++) {
         if ((pi.selT==null) || (!F.isZero(pols.exps[pi.selT].v_n[j]))) {
            const vals = []
            for (let k=0; k<pi.t.length; k++) {
               vals.push(F.toString(pols.exps[pi.t[k]].v_n[j]));
            }
            t[vals.join(",")] = true;
         }
      }


      for (let j=0; j<N; j++) {
         if ((pi.selF==null) || (!F.isZero(pols.exps[pi.selF].v_n[j]))) {
            const vals = []
            for (let k=0; k<pi.f.length; k++) {
               vals.push(F.toString(pols.exps[pi.f[k]].v_n[j]));
            }
            const v = vals.join(",");
            if (!t[v]) {
               res.push(`${pil.plookupIdentities[i].fileName}:${pil.plookupIdentities[i].line}:  plookup not found w=${j} values:
${v}`);
               console.log(res[res.length-1]);
               if (!config.continueOnError) j=N;  // Do not continue checking
            }
         }
      }
```

14. PERMUTATION CONSTRAINT DISCREPANCY BETWEEN VERIFYPIL TOOLING AND STARK GENERATION

SEVERITY: Informational

PATH: pilcom/src/pil_verifier.js

REMEDIATION: make sure that 2 sequences have the same length when there is used a selector in permutation constraint

STATUS: fixed

DESCRIPTION:

In PIL language there is a selector polynomial for choosing specific lines to meet the constraints.

But there is differences how verifyPIL method and STARK generator uses that selector.

In case of permutation constraint, 2 sequences must have the same length.

We can make 2 different length sequences to be checked via selectors.

For example we have a PIL:

```
constant %N = 4;

namespace Global(%N);
  pol constant L1;

namespace PermutationExample(%N);

  pol constant b1, b2; // selector polynomials
  pol commit a1, a2;
  b1{a1} is b2{a2};
```

and corresponding polynomials:

```
const b1 = [1,1,1,0];
const b2 = [1,1,1,1];

const a1 = [1,2,3,4];
const a2 = [4,3,2,1];
```

In that case verifyPIL check will pass but the STARK generator will throw an exception while counting the **z** polynomial.

```
for (let j=0; j<N; j++) {
  if ((pi.selT==null) || (!F.isZero(pols.exps[pi.selT].v_n[j]))) {
     const vals = []
     for (let k=0; k<pi.t.length; k++) {
        vals.push(F.toString(pols.exps[pi.t[k]].v_n[j]));
     }
     const v = vals.join(",");
     t[v] = (t[v] || 0) + 1;
  }
}

for (let j=0; j<N; j++) {
  if ((pi.selF==null) || (!F.isZero(pols.exps[pi.selF].v_n[j]))) {
     const vals = []
     for (let k=0; k<pi.f.length; k++) {
        vals.push(F.toString(pols.exps[pi.f[k]].v_n[j]));
     }
     const v = vals.join(",");
     const found = t[v] ?? false;
     if (!t[v]) {
        res.push(`${pi.fileName}:${pi.line}:  permutation not `+(found === 0 ? 'enought ':'')+`found w=${j} values: ${v}`);
        console.log(res[res.length-1]);
        if (!config.continueOnError) j=N;  // Do not continue checking
     }
     else {
        t[v] -= 1;
     }
  }
}
```

15. CALL DEPTH CHECK MISSING

SEVERITY: Informational

PATH: opcodes/create-terminate-context.zkasm

REMEDIATION: add the limit check in context creating opcodes

STATUS: fixed

DESCRIPTION:

The opcodes in the create-terminate-context.zkasm corresponds to the instructions that create new call contexts, such as opCALL, opDELEGATECALL, opSTATICCALL and etc. By the EVM specification, the call depth needs to have a limit of 1024.

Refer to page 37 of the Yellow Paper:

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

Additional comment:

Although is does not have the check right now, there is no visible security impact at the moment of writing as the fact that the call forwards at max 63/64th of the remaining Gas, and this makes it practically impossible to reach the 1024th depth level. Hence, the status “acknowledged” is treated as “fixed” in this issue.

16. REDUNDANT JUMPS

SEVERITY: Informational

PATH: utils.zkasm

REMEDIATION: consider removing the redundant jumps

STATUS: fixed

DESCRIPTION:

In the utils.zkasm:readPush procedure some of the unconditional jumps are redundant as they jump to the very next operation and can be removed for clean code considerations

utils.zkasm:readPush

```
...
   0 => B                      :JMP(readPushBlock)

readPushBlock:
...

   A*16777216 + C => C         :JMP(doRotate)

doRotate:
...
  B - 1 => A                   :JMP(doRotateLoop)

doRotateLoop:
...
```

hexens