hexens × RISC ZERO

AUG.23

SECURITY REVIEW REPORT FOR RISC ZERO

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.

SCOPE

The analyzed resources are located on:

https://github.com/risc0/risc0/commit/323b5f58e45a7b0de02df6227e1e9de475f70176

The issues described in this report were fixed in the following commit:

https://github.com/risc0/risc0/commit/83cf6867025846cd8ec91d9776ef8fb6c74b3b88

AUDITING DETAILS

STARTED
2023.08.22

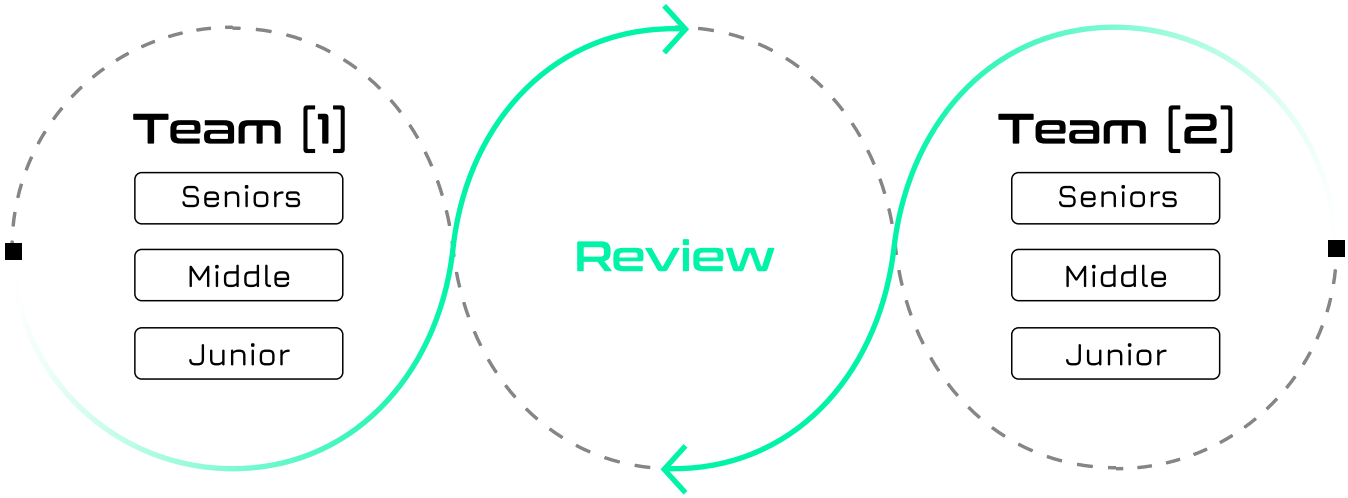
DELIVERED
2023.10.31

Review
Led by

VAHE KARAPETYAN
Co-founder / CTO | Hexens

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

Smart contract 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 smart contract 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.

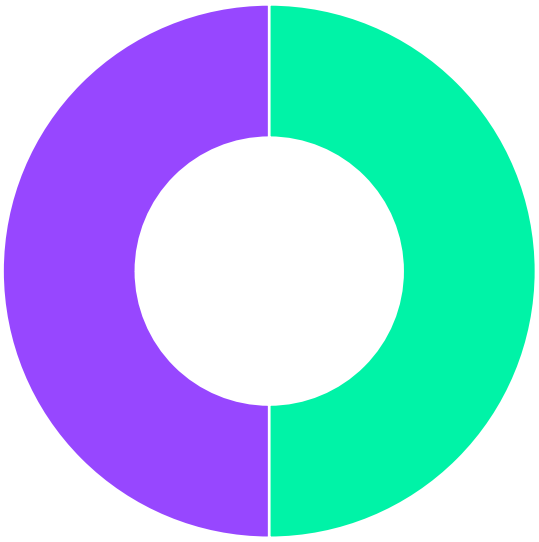
ISSUE SYMBOLIC CODES

Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	1
Medium	2
Low	4
Informational	3

Total: 10



WEAKNESSES

This section contains the list of discovered weaknesses.

RSC0-3

DOS WITH SHA ECALL USING BIG BLOCK COMPRESSION COUNT

SEVERITY: High

PATH:

zkvm/src/exec/executor.rs

REMEDIATION:

add a check in executor's ecall_sha() function to panic or bail in case the the count is bigger than MAX_SHA_COMPRESS_BLOCKS

STATUS: Fixed

DESCRIPTION:

In the executor.rs the function ecall_sha() implements the SYSCALL(3), which is meant to be the SHA256 hash function's implementation. It takes as arguments the out-state, in-state, blocks' pointers, and the count of the block compressions.

In case the syscall is being called indirectly through library functions provided by Risc0, there is a check to ensure that the count is no bigger than MAX_SHA_COMPRESS_BLOCKS (1000)

```
// Limit syscall buffers so that the Executor doesn't get into an infinite
// split situation.
pub const MAX_BUF_BYTES: usize = 4 * 1024;
pub const MAX_BUF_WORDS: usize = MAX_BUF_BYTES / WORD_SIZE;
pub const MAX_SHA_COMPRESS_BLOCKS: usize = 1000;

...

pub unsafe extern "C" fn sys_sha_buffer(
    out_state: *mut [u32; DIGEST_WORDS],
    in_state: *const [u32; DIGEST_WORDS],
    buf: *const u8,
    count: u32,
) {
...
    while count_remain > 0 {
        let count = min(count_remain, MAX_SHA_COMPRESS_BLOCKS as u32);
        ecall_4(
            ecall::SHA,
            out_state as u32,
            in_state as u32,
            ptr as u32,
            ptr.add(DIGEST_BYTES) as u32,
            count,
        );
...
    }
}
```

The SYSCALL(3) can also be called directly through the ELF's assembly, passing the count parameter (A4 register) as a bigger number, and the ecall_sha() function will fail to do the sanity check on the count's value.

This leads to a Denial of Service of the executor system, with the CPU and Memory resources being exhausted. We have created a Proof of Concept, which uses a count value as low as 10000 and overflows the Memory with more than 130GB as well as overloading the CPU, keeping in mind that the count can be set up to a maximum value of 2^32 -1.

It is important to note that this attack vector is crucial for proving ecosystems such as Bonsai or any other systems that users might want to create in future. E.g. Bonsai's single server can be DoS'ed with only one proving request transaction.

```
fn ecall_sha(&mut self) -> Result<OpCodeResult> {
    let out_state_ptr = self.monitor.load_register(REG_A0);
    let in_state_ptr = self.monitor.load_register(REG_A1);
    let mut block1_ptr = self.monitor.load_register(REG_A2);
    let mut block2_ptr = self.monitor.load_register(REG_A3);
    let count = self.monitor.load_register(REG_A4);

                        let      in_state:      [u8;      DIGEST_BYTES]      =
self.monitor.load_array(in_state_ptr)?;
                    let     mut     state:     [u32;     DIGEST_WORDS]     =
bytemuck::cast_slice(&in_state).try_into().unwrap();
    for word in &mut state {
        *word = word.to_be();
    }

    log::debug!("Initial sha state: {state:08x?}");
    for _ in 0..count {
      ...
    }
```

Proof of Concept:

```
fn ecall_sha(&mut self) -> Result<OpCodeResult> {
    let out_state_ptr = self.monitor.load_register(REG_A0);
    let in_state_ptr = self.monitor.load_register(REG_A1);
    let mut block1_ptr = self.monitor.load_register(REG_A2);
    let mut block2_ptr = self.monitor.load_register(REG_A3);
    let count = self.monitor.load_register(REG_A4);

                    let     in_state:     [u8;     DIGEST_BYTES]     =
self.monitor.load_array(in_state_ptr)?;
                  let     mut     state:     [u32;     DIGEST_WORDS]     =
bytemuck::cast_slice(&in_state).try_into().unwrap();
    for word in &mut state {
        *word = word.to_be();
    }

    log::debug!("Initial sha state: {state:08x?}");
    for _ in 0..count {
      ...
    }
```

RSC0-6

ELF SEGMENT VIRTUAL ADDRESS CAN BE SET TO AN ARBITRARY ADDRESS

SEVERITY: Medium

PATH:

binfmt/src/elf.rs

REMEDIATION:

add a security check to ensure the ELF sections' virtual addresses can only be loaded in the user memory region

STATUS: Fixed

DESCRIPTION:

The function load_elf() loads and checks the ELF sections and checks that the file size and each segment's size are not bigger than MAX_MEM (2^28 by default). However, it fails to check that the virtual address (vaddr), which eventually is the address of the section in the guest environment, does not overlap with the system memory region.

The virtual memory layout in Risc0 has hardcoded address regions for user and system spaces:

```
/// Program (text followed by data and then bss) gets loaded in
/// starting at this location.  HEAP begins right afterwards.
pub const TEXT_START: u32 = 0x0000_0400;
/// Top of stack; stack grows down from this location.
pub const STACK_TOP: u32 = 0x0BF_FFC00;
/// Minimum mount of room to leave for the stack when allocating from the
heap.
pub const RESERVED_STACK: u32 = mb(1) as u32;
pub const SYSTEM: Region = Region::new(0x0C00_0000, mb(16));
pub const PAGE_TABLE: Region = Region::new(0x0D00_0000, mb(16));
pub const PRE_LOAD: Region = Region::new(0x0D70_0000, mb(9));
```

This situation can lead to system memory overwriting, where a malicious ELF can subtly change the guest program's initial state, which otherwise would have been impossible due to memory read/write security checks. This means the code disassembly or decompilation will not show malicious instructions. However, the other segments' layouts may collide with the system memory region and undermine the expected program execution flow.

```
let vaddr: u32 = segment.p_vaddr.try_into()?;
let offset: u32 = segment.p_offset.try_into()?;
for i in (0..mem_size).step_by(4) {
    let addr = vaddr.checked_add(i).context("Invalid segment vaddr")?;
    if i >= file_size {
        // Past the file size, all zeros.
        image.insert(addr, 0);
    } else {
        let mut word = 0;
        // Don't read past the end of the file.
        let len = std::cmp::min(file_size - i, 4);
        for j in 0..len {
            let offset = (offset + i + j) as usize;
            let byte = input.get(offset).context("Invalid segment offset")?;
            word |= (*byte as u32) << (j * 8);
        }
        image.insert(addr, word);
    }
}
```

ECALL PARAMETERS MISSING ADDRESS RANGE CHECK

SEVERITY: Medium

PATH:

zkvm/src/exec/executor.rs

REMEDIATION:

add checks to ecall implementations to ensure that the memory addressed by the parameters does not overlap with system space

STATUS: Fixed

DESCRIPTION:

The executor's ecall functions that correspond to their syscalls (ecall_sha, ecall_halt, ecall_input, ecall_software, ecall_bigint) are missing the security check to ensure that the memory pointers provided by the guest programs are not out of user space memory. As the ecall functions have input and output parameters using which the syscall reads and writes the data, a malicious guest program can abuse this to bypass the MemoryMonitor's security check done in read_mem and write_mem functions:

```
fn read_mem(&mut self, addr: u32, size: MemAccessSize) -> Option<u32> {
    if addr < TEXT_START || addr as usize >= SYSTEM.start() {
        return None;
    }
    ...
}

fn write_mem(&mut self, addr: u32, size: MemAccessSize, store_data: u32) ->
bool {
    if addr < TEXT_START || addr as usize >= SYSTEM.start() {
        return false;
```

```
    }
    ...
}
```

Thus giving the ability to read and write to arbitrary addresses, including SYSTEM and PAGE_TABLE regions.

```
fn ecall_input(&mut self) -> Result<OpCodeResult> {
    log::debug!("ecall(input)");
    let in_addr = self.monitor.load_register(REG_A0);
    self.monitor
        .load_array::<{ DIGEST_WORDS * WORD_SIZE }>(in_addr)?;
    Ok(OpCodeResult::new(self.pc + WORD_SIZE as u32, None, 0))
}
```

```
fn ecall_sha(&mut self) -> Result<OpCodeResult> {
    let out_state_ptr = self.monitor.load_register(REG_A0);
    let in_state_ptr = self.monitor.load_register(REG_A1);
    let mut block1_ptr = self.monitor.load_register(REG_A2);
    let mut block2_ptr = self.monitor.load_register(REG_A3);
    let count = self.monitor.load_register(REG_A4);

    let in_state: [u8; DIGEST_BYTES] = self.monitor.load_array(in_state_ptr)?;

    ...
    self.monitor
      .store_region(out_state_ptr, bytemuck::cast_slice(&state))?;
    ...
}
```

RSCO-11

CUDA MEMORY CAN LEAK PRIVATE INPUTS FROM DATA TRACE

SEVERITY: Low

PATH:

risc0/zkvm/src/host/server/prove/prover_impl.rs

REMEDIATION:

it is recommended to shorten the lifetime of the GPU buffer as much as possible and explicitly zero out the memory before freeing it using DeviceBuffer.set_8 function or calling cuMemset function from CUDA API

STATUS: Acknowledged

DESCRIPTION:

Risc0 implements GPU acceleration for proving calculations, such as field operations, NTT/INTT, and Poseidon/SHA hashes functions. The prover supports CUDA and Metal architectures and has built kernels for parallelized calculations. In order to gain performance and for kernels to access the trace columns, the prover allocates GPU device memory to store the trace and to do calculations on the field elements.

In the case of CUDA, the CudaBuffer is used, which eventually calls cust's DeviceBuffer.uninitialized() function and allocates GPU device memory via cuda_malloc().

```
pub unsafe fn uninitialized(size: usize) -> CudaResult<Self> {
    let ptr = if size > 0 && size_of::<T>() > 0 {
        cuda_malloc(size)?
    }
    ...
}
```

Another observation is that the prover allocates the HAL buffer (CUDA buffer in this particular case) in the ProverServer.prove_segment() function using copy_from_elem and immediately passes it as an argument to the prover.commit_group:

```
prover.commit_group(
    REGISTER_GROUP_DATA,
    hal.copy_from_elem("data", &adapter.get_data().as_slice()),
);
```

It may seem that the CUDA buffer's lifetime will be limited to the commit_group function's call as it transfers the ownership, although the buffer is being inserted into the prover's groups vector:

```
        // buf ownership is transferred to make_coeffs and returned back to
coeffs
        let coeffs = make_coeffs(self.hal, buf, group_size);
        // PolyGroup takes the ownership and gets inserted into self.groups
        let group_ref = self.groups[tap_group_index].insert(PolyGroup::new(
            self.hal,
            coeffs,
            group_size,
            self.cycles,
            "data",
        ));
```

This makes the CUDA buffer outlive the commit_group's scope and be dropped at the end of the prove_segment function; this is crucial as the buffer is not being deallocated explicitly in the code, but rather when the object is being dropped (end of prove_segment) the cuda_free will be called:

```
pub fn drop(mut dev_buf: DeviceBuffer<T>) -> DropResult<DeviceBuffer<T>> {
    ...
          unsafe {
              match cuda_free(ptr) {
                  Ok(()) => {
                      mem::forget(dev_buf);
                      Ok(())
                  }
    ...
  }
```

Eventually, this means that the data trace will not be deallocated and rewritten in the GPU device's memory during proof generation; our empirical observations validate this as well.

The weakness arises due to a number of facts: the trace can contain private inputs (secret values), the buffer lifetime is prolonged during the proof generation (a computationally expensive process) and the fact that the buffer is not explicitly zeroed out on the deallocation. The security rule of thumb is to store secret values (no matter what type of memory) for the shortest possible period of time, as well as to guarantee that it will be zeroed out upon deallocation.

For the attack, one can leverage how the GPU handles the device memory and the allocations/deallocations. The function cuda_malloc() allocates a buffer in the device memory; this memory is also referred to as "global memory" - as technically, this memory can be further reallocated by other GPU contexts (processes) upon its deallocation.

There are security measures implemented in the modern GPU drivers to implicitly address this kind of memory leakage, hence the reason why the local testing setup will most likely not be vulnerable. The main mitigation is to zero out the memory upon freeing; this mechanism is also referred to as memory scrubbing.

It is crucial to mention that Memory scrubbing is an undefined behaviour in CUDA/Nvidia drivers and is not a guaranteed feature. There are setups where the automatic memory scrubbing can be disabled, e.g:

- Some chips may not have it enabled
- Most of the SLI configurations will have it disabled
- Manual configurations can disable the feature
- Older drivers don't have the feature implemented
- Building sophisticated kernels to trick the driver into failing to allocate the scrubber's internal structures.
- etc.

These situations can be observed in Nvidia's open-source kernel modules source code; nevertheless, as most of the drivers' code is closed-source and undocumented, it is hard to tell which are the exact cases when the memory scrubbing is enabled or disabled.

Given all this information, a low-privileged attacker (non-root) or even a remote attacker (in case the GPU resource is being shared and in an insecure manner) can create a malicious CUDA kernel to leak the secret data from the trace. The attacker can leverage the GPU allocator mechanisms to allocate the recently freed memory. Much like the libc's malloc() function, the GPU memory allocator also tends to allocate the same-sized chunks from the recently freed list.

This, and the fact that the memory allocation size for the trace is publicly known for every guest program, gives the attacker a big edge over the chances of a successful attack.

```
#include <stdio.h>

#define TRACE_SIZE 29229056 * 4 //this size is taken for JSON example

__global__
void leak(unsigned int* ptr1, unsigned int* ptr2)
{
  //we have two allocations of same size
  //there is some possiblity that ptr2 allocated the trace instead of ptr1
  //so we check it
  if(ptr1[TRACE_SIZE-1] != 0){ //we check the last element at it is a random
element generated by executor in the compute_verify and most likely will not
be 0
    for(int i=0; i<TRACE_SIZE; i++){
      ptr2[i] = ptr1[i];
    }
  }
  if(ptr2[TRACE_SIZE-1] !=0){
    return; //do nothing, we will always expect that ptr2 containt the
leaked data
  }
}

int main(void)
{
  unsigned int* ptr1;
  unsigned int* ptr2;
  unsigned int* result = (unsigned int*)malloc(TRACE_SIZE);

  //starting race
  while(1){

    //hopefully one of this allocations will end up on trace
    cudaMalloc(&ptr1, TRACE_SIZE);
    cudaMalloc(&ptr2, TRACE_SIZE);

    //try leak
    leak<<<1,1>>>(ptr1,ptr2);

    //get the result and check it
    cudaMemcpy(&result, ptr2, TRACE_SIZE, cudaMemcpyDeviceToHost);
```

```
    if(result[TRACE_SIZE-1] !=0){
      break;  //we won!
    }

  }
  printf("Race won!");
  for(int i=0; i<TRACE_SIZE; i++){
      printf("DATA TRACE[%d]: %d\n", i, result[i]);
  }

  cudaFree(ptr1);
  cudaFree(ptr2);
  free(result);
}
```

The attacker will need to start the race condition attack; although the chances to win are heavily inflated as the trace size is usually big and the lifetime of buffer outlives the allocation-extensive proof generation procedure.

It is worth mentioning that the older versions of GPU drivers had almost no security measures implemented, such as scrubbing, virtual address randomization and segregation, making the attack straightforward.

RSCO-8

EXECUTOR ELF LOADER VIRTUAL ADDRESS WORD ALIGNMENT CHECK MISSING

SEVERITY: Low

PATH:

binfmt/src/elf.rs

REMEDIATION:

add a check to ensure that the ELF sections' virtual addresses are all aligned by word size

STATUS: Fixed

DESCRIPTION:

The function load_elf() parses the ELF binary, loads the memory segments, and does sanity checks on the ELF structure, such as checking that the binary is 32-bit, RISC-V machine type, max memory checks, etc. Nonetheless, it fails to check that the section's virtual address (vaddr) is aligned to word size (4 bytes by default).

This leads to situations where the page size can become bigger than 1024 bytes (1025-1027 bytes). Although the executor will fail to execute such a program and panic out the process, we are unaware of the circuit behaviour.

```
let vaddr: u32 = segment.p_vaddr.try_into()?;
let offset: u32 = segment.p_offset.try_into()?;
for i in (0..mem_size).step_by(4) {
    let addr = vaddr.checked_add(i).context("Invalid segment vaddr")?;
    if i >= file_size {
        // Past the file size, all zeros.
        image.insert(addr, 0);
    } else {
        let mut word = 0;
        // Don't read past the end of the file.
        let len = std::cmp::min(file_size - i, 4);
        for j in 0..len {
            let offset = (offset + i + j) as usize;
            let byte = input.get(offset).context("Invalid segment offset")?;
            word |= (*byte as u32) << (j * 8);
        }
        image.insert(addr, word);
    }
}
```

RSCO-13

NON CONSTANT-TIME IMPLEMENTATION IN FINITE FIELD OPERATIONS

SEVERITY: Low

PATH:

risc0/core/src/field/baby_bear.rs, risc0/core/src/field/goldilocks.rs

REMEDIATION:

mitigation from timing attacks is using constant-time operations. This involves various techniques such as replacing branches with bitwise operations, avoiding array accesses with secret indices, or making sure that loops have a fixed number of iterations (https://eprint.iacr.org/2021/1121.pdf). Use GitHub - dalek-cryptography/subtle: Pure-Rust traits and utilities for constant-time cryptographic implementations. library for implementing constant-time operations

STATUS: Acknowledged

DESCRIPTION:

Timing attack is a side channel attack which allows an attacker to retrieve potentially sensitive information from the application by observing the normal behavior of the response times.

There are several implementation cases that can cause timing attacks:

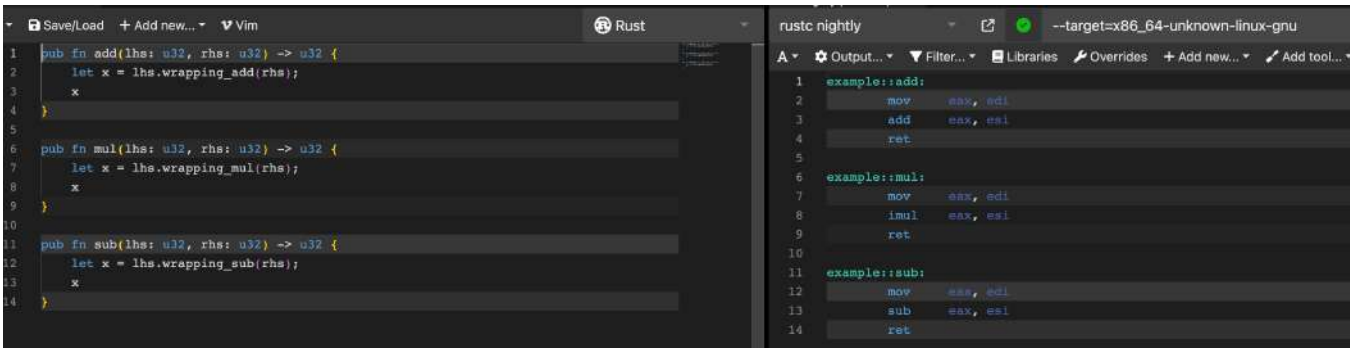
1. Any loop leaks the number of iterations taken.
2. Any memory access leaks the address (or index) accessed.
3. Any conditional statement leaks which branch was taken.

Finite field operations in the application are using branchings (add, sub, mul).

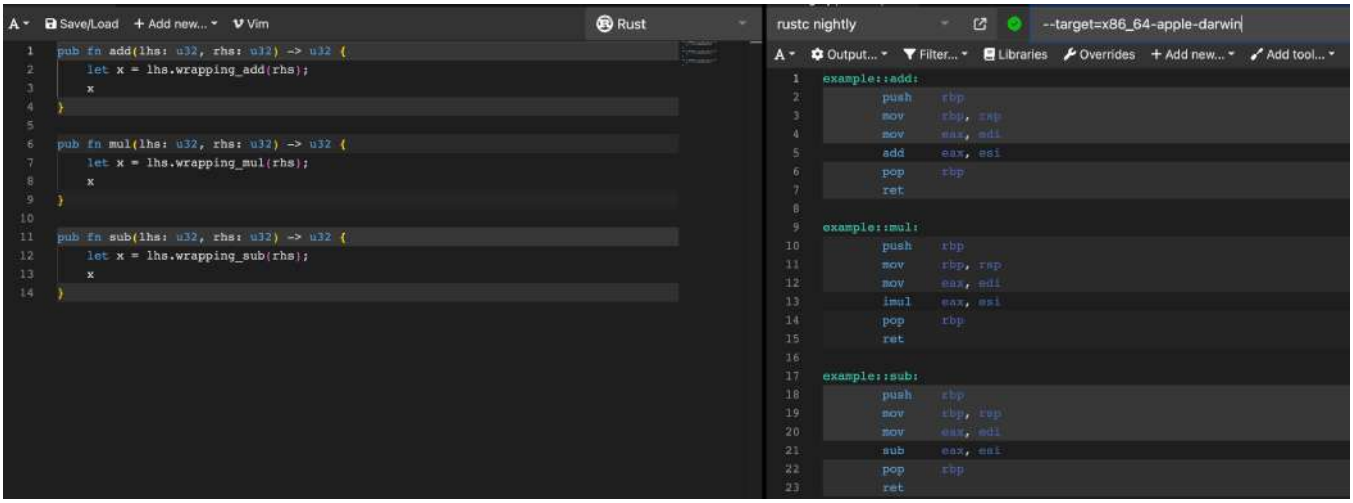
```rust
/// Wrapping addition of [Elem] using Baby Bear field modulus
fn add(lhs: u32, rhs: u32) -> u32 {
    let x = lhs.wrapping_add(rhs);
    if x >= P {
        x - P
    } else {
        x
    }
}

/// Wrapping subtraction of [Elem] using Baby Bear field modulus
fn sub(lhs: u32, rhs: u32) -> u32 {
    let x = lhs.wrapping_sub(rhs);
    if x > P {
        x.wrapping_add(P)
    } else {
        x
    }
}

/// Wrapping multiplication of [Elem]  using Baby Bear field modulus
// Copied from the C++ implementation (fp.h)
const fn mul(lhs: u32, rhs: u32) -> u32 {
    // uint64_t o64 = uint64_t(a) * uint64_t(b);
    let mut o64: u64 = (lhs as u64).wrapping_mul(rhs as u64);
    // uint32_t low = -uint32_t(o64);
    let low: u32 = 0u32.wrapping_sub(o64 as u32);
    // uint32_t red = M * low;
    let red = M.wrapping_mul(low);
    // o64 += uint64_t(red) * uint64_t(P);
    o64 += (red as u64).wrapping_mul(P_U64);
    // uint32_t ret = o64 >> 32;
    let ret = (o64 >> 32) as u32;
    // return (ret >= P ? ret - P : ret);
    if ret >= P {
        ret - P
    } else {
        ret
    }
}
```

Rust's wrapping_add, wrapping_sub, wrapping_mul methods on different CPU architectures are not creating branches.

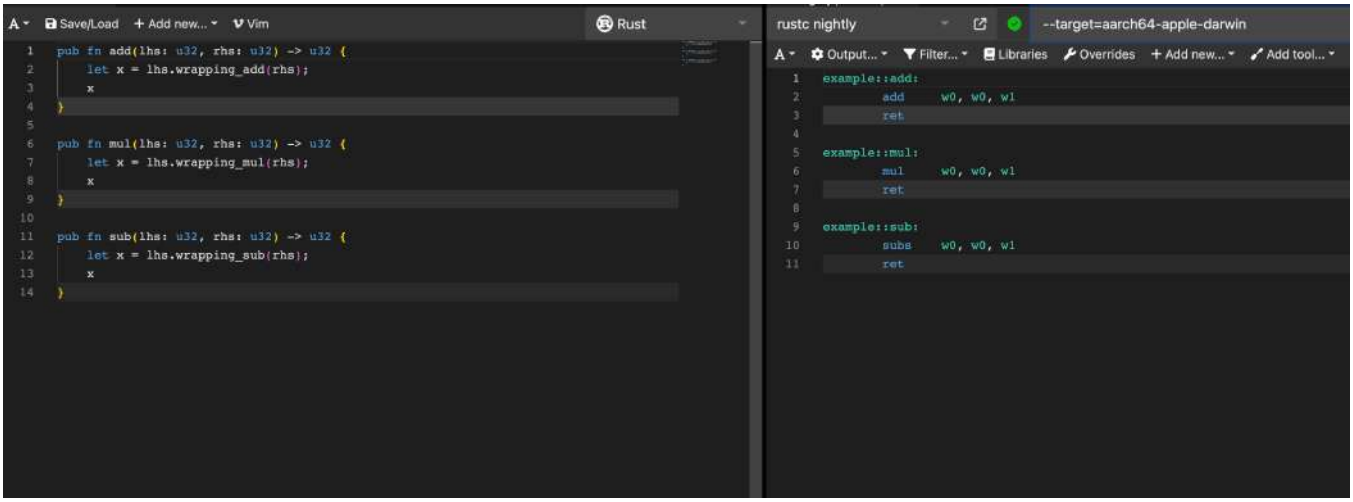
- x86_64-unknown-linux-gnu



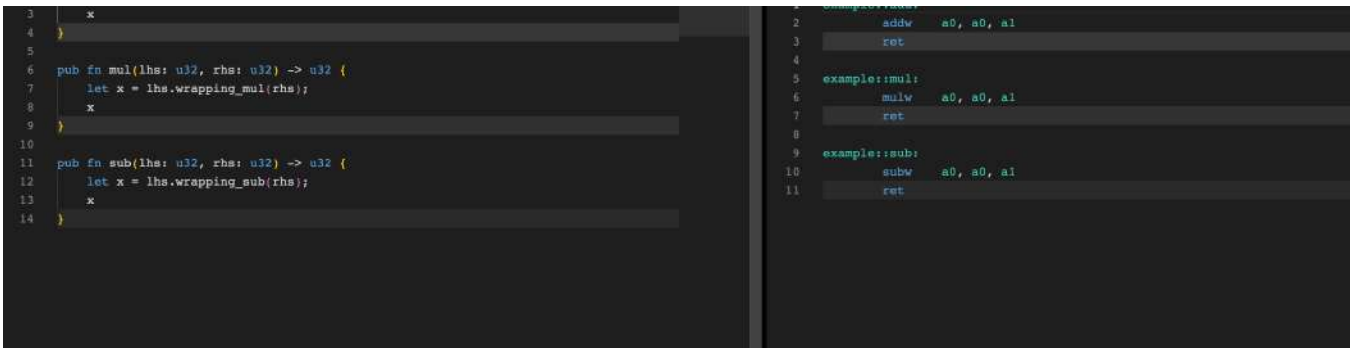
- x86_64-apple-darwin



- aarch64-apple-darwin



- riscv64gc-unknown-linux-gnu



```
    x
}

pub fn mul(lhs: u32, rhs: u32) -> u32 {
    let x = lhs.wrapping_mul(rhs);
    x
}

pub fn sub(lhs: u32, rhs: u32) -> u32 {
    let x = lhs.wrapping_sub(rhs);
    x
}
```

```
        addw    a0, a0, a1
        ret

example::mul:
        mulw    a0, a0, a1
        ret

example::sub:
        subw    a0, a0, a1
        ret
```

RSC0-10

SYSCALL POINTER TYPE PARAMETERS MISSING ADDRESS RANGE CHECK

SEVERITY: Low

PATH:

zkvm/src/host/server/exec/syscall.rs

REMEDIATION:

add checks to syscall implementations to ensure that the buf_ptr and buf_ptr + buf_len addresses does not overlap with system space

STATUS: Fixed

DESCRIPTION:

Risc0's syscalls (SysGetenv, SysLog, SysPanic, SysSliceIo, PosixIo's sys_write) read a buffer from the memory via buf_ptr and buf_len parameters. The buf_ptr parameter is missing the security check to ensure that the memory pointers provided by the guest programs are not out of user space memory also for the buf_ptr + buf_len pointer. Thus giving the ability to read from arbitrary addresses, including SYSTEM and PAGE_TABLE regions.

```rust
impl Syscall for SysGetenv {
    fn syscall(
        &mut self,
        _syscall: &str,
        ctx: &mut dyn SyscallContext,
        to_guest: &mut [u32],
    ) -> Result<(u32, u32)> {
        let buf_ptr = ctx.load_register(REG_A3);
        let buf_len = ctx.load_register(REG_A4);
        let from_guest = ctx.load_region(buf_ptr, buf_len)?;
        ...
}
```

```rust
impl Syscall for SysLog {
    fn syscall(
        &mut self,
        _syscall: &str,
        ctx: &mut dyn SyscallContext,
        _to_guest: &mut [u32],
    ) -> Result<(u32, u32)> {
        let buf_ptr = ctx.load_register(REG_A3);
        let buf_len = ctx.load_register(REG_A4);
        let from_guest = ctx.load_region(buf_ptr, buf_len)?;
        let msg = from_utf8(&from_guest)?;
        println!("R0VM[{}] {}", ctx.get_cycle(), msg);
        Ok((0, 0))
    }
}
```

RSCO-9

ZK SHIFT IS HARDCODED AND IS NOT A GENERATOR OF MULTIPLICATIVE SUBGROUP

SEVERITY: Informational

PATH:

risc0/zkp/src/hal/cpu.rs, metal.rs, cuda.rs, dual.rs

REMEDIATION:

consider adding generator parameter to Field and using it for the zk_shift

STATUS: Acknowledged

DESCRIPTION:

The function zk_shift() in HAL (Hardware Abstraction Layer) is used as part of generating the Low Degree Extension (LDE) evaluation domain of the "original" trace domain. It multiplies the coefficients of the polynomials by powers of 3 (the zk shift value), such that the Ci is multiplied by 3i. This way, the polynomial P(x) is shifted to P(3x). As can be seen in the ethSTARK Documentation [Section 3.4], the LDE needs to be a coset of the multiplicative subgroup of the field; the way to generate this coset is to use the generator of the F_p^x (which coincidently was chosen to be 3 as well).

The reason why the generator is being used is that the "shift" element needs to satisfy the following properties in order to comply with the security properties of STARK:

- The element should not be part of the coset
- The multiplicative order of the element needs to be bigger than the order of the coset

The best way to ensure these properties is to use the subgroup's generator. In the case of the ethSTARK, the finite field used has $p = 2^{61} + 20 * 2^{32} + 1$ and the element 3 is a generator for its multiplicative subgroup.

```
sage: F = GF(2**61 + 20*(2**32) + 1)
sage: F.unit_group().order()
2305843095113039872
sage: F(3).multiplicative_order()
2305843095113039872
```

Although the Risc0 prover uses other finite fields, the Baby Bear (~31-bit prime) and Goldilocks (~63-bit prime).

After checking the BB and GL fields, we found out that element 3 is not a generator for their multiplicative subgroups:

```
sage: F = GF(2**61 + 20*(2**32) + 1)
sage: F.unit_group().order()
2305843095113039872
sage: F(3).multiplicative_order()
2305843095113039872
```

Baby Bear:

```
sage: BB = GF(2013265921)
sage: BB.unit_group().order()
2013265920
sage: BB(3).multiplicative_order()
503316480
```

We also checked that element 3 is not part of the coset as well, using a small sage script:

```
BB = GF(2013265921)

rous  =  [1,  2013265920,  284861408,  1801542727,  567209306,  740045640,
918899846,  1881002012,  1453957774,  65325759,  1538055801,  515192888,
483885487,  157393079,  1695124103,  2005211659,  1540072241,  88064245,
1542985445,  1269900459,  1461624142,  825701067,  682402162,  1311873874,
1164520853, 352275361, 18769, 137]

print(BB.order())

for i in range(len(rous)):
print("testing out" + str(i))
elem = BB(rous[i])
for j in range(2**i):
if elem^j == 3:
print(i)
print(j)
print(elem)
break
```

Given this and the fact that “luckily” multiplicative orders were bigger than Root of Unities orders, the configuration chosen does not pose any security issues at the moment.

Nevertheless, we recommend not to hardcode the shift and better use the generator for every field in use. In the future, Risc0 may change the fields used or add new ones, as these properties are subtle and easy to overlook for hardcoded 3 shift; thus, they may not hold and pose serious security issues.

```
fn zk_shift(&self, io: &Self::Buffer<Self::Elem>, poly_count: usize) {
    let bits = log2_ceil(io.size() / poly_count);
    let count = io.size();
    assert_eq!(io.size(), poly_count * (1 << bits));
    let mut io = io.as_slice_mut();
    (&mut io[..], 0..count)
        .into_par_iter()
        .for_each(|(io, idx)| {
            let pos = idx & ((1 << bits) - 1);
            let rev = bit_rev_32(pos as u32) >> (32 - bits);
            let pow3 = Self::Elem::from_u64(3).pow(rev as usize);
            *io *= pow3;
        });
}
```

RSCO-7

REDUNDANT IMAGE MERKLE ROOT CALCULATION

SEVERITY: Informational

PATH:

risc0/zkvm/src/host/server/exec/monitor.rs

REMEDIATION:

remove the redundant hashing

STATUS: Acknowledged

DESCRIPTION:

The function mark_page() will add the root page to faults.writes vector, and thus when the build_image() is called, the hash of the root page will be calculated and written to the 704th-byte position in the root page. Furthermore, the build_image will calculate the root hash once more via compute_id() call.

```
fn mark_page(&mut self, addr: u32) {
    let info = &self.image.info;
    let page_idx = info.get_page_index(addr);
    if self.dirty[page_idx as usize] {
        return;
    }

    log::debug!("mark_page: 0x{page_idx:08x}");
    let page_cycles = if page_idx == info.root_idx {
        let num_root_entries = info.num_root_entries as usize;
        cycles_per_page(num_root_entries / 2)
    } else {
        let entry_addr = info.get_page_entry_addr(page_idx);
        self.mark_page(entry_addr);
        cycles_per_page(BLOCKS_PER_PAGE)
    };

    self.dirty[page_idx as usize] = true;
    self.pending_actions
        .push(Action::PageWrite(page_idx, page_cycles));
    self.page_write_cycles += page_cycles;
    self.faults.writes.insert(page_idx);
}
```

CLEAN CODE RECOMMENDATIONS

SEVERITY: Informational

PATH:

zkvm/src/host/recursion/prove/preflight.rs

REMEDIATION:

set these numbers as constants inside set_top function and document their meanings

reuse result variable inset_micro in self.externs.wom_write call

STATUS: Acknowledged

DESCRIPTION:

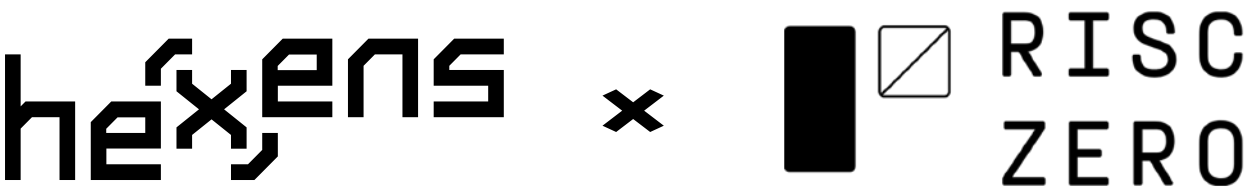
The set_top function in Preflight implementation converts from/to Montgomery form via magic numbers (presumably Montgomery Rs).

```
if do_mont != 0 {
    // Convert from montgomery form
    load = load * Fp::from(943718400u32);
  }
```

```
if do_mont != 0 {
    // Convert to montgomery form
    store = store * Fp::from(268435454u32);
}
```

Inside set_micro function micro_op::INV matching uses field's element inverse to store WoM. Modular inverse is counted two times.

```
let result = a.inv();
trace!("inv({a:?}) -> {result:?}");
self.externs.wom_write(write_addr, a.inv());
```


hexens
RISC
ZERO