# Heap corruption in foreign call results with nested tuple arrays

Critical **Savio-Sou** published **GHSA-jj7c-x25r-r8r3** last week

**Package**

No package listed

**Affected versions**

<=1.0.0-beta.18

**Patched versions**

1.0.0-beta.19

**Description**

## Description

Noir programs can invoke external functions through foreign calls. When compiling to Brillig bytecode, the SSA instructions are processed block-by-block in `BrilligBlock::compile_block()`. When the compiler encounters an `Instruction::Call` with a `Value::ForeignFunction` target, it invokes `codegen_call()` in `brillig_call/code_gen_call.rs`, which dispatches to `convert_ssa_foreign_call()`.

Before emitting the foreign call opcode, the compiler must pre-allocate memory for any array results the call will return. This happens through `allocate_external_call_results()`, which iterates over the result types. For `Type::Array` results, it delegates to `allocate_foreign_call_result_array()` to recursively allocate memory on the heap for nested arrays.

The `BrilligArray` struct is the internal representation of a Noir array in Brillig IR. Its `size` field represents the **semi-flattened size**, the total number of memory slots the array occupies, accounting for the fact that composite types like tuples consume multiple slots per element. This size is computed by `compute_array_length()` in `brillig_block_variables.rs`:

```
pub(crate) fn compute_array_length(item_typ: &CompositeType, elem_count: usize) -> e
    item_typ.len() * elem_count
}
```

For the **outer** array, `allocate_external_call_results()` correctly uses `define_variable()`, which internally calls `allocate_value_with_type()`. This function applies the formula above, producing the correct semi-flattened size.

However, for **nested** arrays, `allocate_foreign_call_result_array()` contains a bug. When it encounters a nested `Type::Array(types, nested_size)`, it calls:

```
Type::Array(_, nested_size) => {
    let inner_array = self.brillig_context.allocate_brillig_array(*nested_size as usize)
    // ....
}
```

The pattern `Type::Array(_, nested_size)` discards the inner types with `_` and uses only `nested_size`, the **semantic length** of the nested array (the number of logical elements), not the semi-flattened size. For simple element types this works correctly, but for composite element types it under-allocates. Consider a nested array of type `[(u32, u32); 3]`:

- Semantic length: 3 (three tuples)
- Element size: 2 (each tuple has two fields)
- Required semi-flattened size: 6 memory slots

The current code passes `3` to `allocate_brillig_array()`, which then calls `codegen_initialize_array()`. This function allocates `array.size + ARRAY_META_COUNT` slots, only 4 slots instead of the required 7 (6 data + 1 metadata). When the VM executes the foreign call and writes 6 values plus metadata, it overwrites adjacent heap memory.

## Impact

Foreign calls returning nested arrays of tuples or other composite types corrupt the Brillig VM heap.

## Recommendation

Multiply the semantic length by the number of element types when allocating nested arrays. Extract the inner types from the pattern and replace the `nested_size` argument to `allocate_brillig_array()` with `types.len() * nested_size` to compute the semi-flattened size. Alternatively, reuse the existing `compute_array_length()` helper function to maintain consistency with outer array allocation.

## AuditHub Issue Link

https://app.audithub.dev/app/organizations/161/projects/521/project-viewer?version=1201&_t=1768255975985&issueId=751

**Severity**

Critical

**CVE ID**

CVE-2026-41197

**Weaknesses**

No CWEs