Garmin Forerunner 235: Integer overflow in NEWA TVM instruction

Vendors

- Garmin

Affected Products

Forerunner 235 firmware version 7.90

Summary

The ConnectIQ program interpreter fails to check for overflow when allocating the array for the `NEWA` instruction. This a constrained read/write primitive across the entire MAX32630 address space. A successful exploit would allow a ConnectIQ app store application to escape and perform activities outside the restricted application execution environment.

Mitigation

This issue was fixed by Forerunner 235 software version 8.20.

Credit

This issue was found by Dion Blazakis of Atredis Partners.

References

- https://github.com/atredispartners/advisories/ATREDIS-2020-0004.txt
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-27484

Report Timeline

- 2020-04-17: Atredis Partners sent an initial notification to Garmin, including a draft advisory.
- 2020-08-17: Atredis Partners shared a copy of the draft advisory with CERT/CC.
- 2020-10-05: Atredis Partners published this advisory.

Technical Details

The TVM interpreter is responsible for running the application (or `.PRG`) downloaded from the Garmin ConnectIQ app store. The `.PRG` file packages both resources (e.g., images and text) and TVM bytecode needed to run the program. Applications are programmed in the proprietary MonkeyC language and are built into `.PRG` programs via the free Garmin ConnectIQ SDK. Once on the device, the virtual machine ensures the applications are strictly constrained to prevent excess use of memory or computation time. Additionally, the runtime API exposed to each program is constrained based on the type of application installed (e.g., a watchface, a widget, a full application).

The TVM is a stack-based virtual machine. One instruction, `NEWA` , is used to create an runtime array of TVM values of a fixed size initialized with the null value. `NEWA` expects a number-like value on the top of the stack indicating the size of the array. Decompilation of the `NEWA` opcode implementation shows the one check on the length value (ensuring it is not negative) before passing it to `tvm_value_array_allocate` for the array size calculation.

```
int __fastcall tvm_op_newa(struct tvm *ctx)
{
  struct stack_value *sp;
  int rv;
  unsigned int length;
  struct tvm_value value;

  sp = ctx->stack_ptr;
  length = 0;
  value = *sp;
  rv = tvm_value_to_int(ctx, &value, &length);
  if ( rv ) {
    if ( length < 0 )
    {
      rv = 10;
      tvm_value_decref(ctx, &value);
      return rv;
    }

    rv = tvm_value_array_allocate(ctx, length, ctx->stack_ptr);
    if ( rv )
    {
      rv = tvm_value_decref(ctx, &value);
      if ( !rv )
        return tvm_value_incref(ctx, ctx->stack_ptr);
    }
  }
  tvm_value_decref(ctx, &value);
  return rv;
}
```

The `tvm_value_array_allocate` function will perform the unchecked array size calculation as shown below.

```
int __fastcall tvm_value_array_allocate(struct tvm *ctx, int length, struct tvm_value *array_value)
{
  unsigned int allocation_size; // r6
  int rv; // r0 MAPDST
  struct tvm_value_array_data *array_data; // r9
  void *array_data_handle; // [sp+4h] [bp-24h] MAPDST

  array_data_handle = 0;
  allocation_size = 5 * length + 15;
  rv = tvm_alloc_for_app(ctx, allocation_size, &array_data_handle);
  if ( !array_data_handle )
    return 7;
  array_data = (struct tvm_value_array_data *)mem_pointer_borrow(array_data_handle);
  memset((int *)array_data, 0, allocation_size);
  array_data->m_0x01 = 1;
  array_data->type = ARRAY;
  array_data->length = length;
  mem_pointer_release(array_data_handle);
  array_value->type = ARRAY;
  array_value->value = (unsigned int)array_data_handle;
  return rv;
}
```

This can be triggered by creating an array of size `0x33333333`. The value is still positive for a 32-bit integer (passing the check in the `tvm_op_newa` function). When the `allocation_size` is calculated, the result will overflow the 32-bit unsigned int:

```
>>> length = 0x33333333
>>> allocation_size = 5 * length + 15
>>> hex(allocation_size)
'0x10000000e'
>>> hex(allocation_size & 0xffffffff)
'0xe'
```

The original `length` value is stored in the resulting `tvm_value_array_data` and this is the value used to check bounds during the array read and write operations (performed by the `AGETV` and `APUTV` instructions).

This can be directly triggered through MonkeyC and requires direct bytecode manipulation to create a proof-of-concept. There are a number of additional constraints to turn this into a reliable read/write anything anywhere primitive but they do not seem insurmountable.