TRUST IN SOFT

Proof-of-concept for Goodix Driver GT9xx Analysis

January 23, 2017

Contents

1 Introduction

This document details the formal analysis performed using TrustInSoft Analyzer on the following Linux kernel driver: https://source.codeaurora.org/quic/la/kernel/msm-3.18/tree/drivers/input/touchscreen/gt9xx?h=LA.HB.1.1.1.c2

A configuration of the Linux Kernel and the GT9xx driver is described. A plan for the formal verification of the GT9xx driver thus configured is formulated and implemented. This formal verification has been carried out with TrustInSoft Analyzer, a source code analyzer that relies on state-of-the-art formal verification techniques. All the elements that, taken together, allow to conclude that given the usage shown, the driver is safe from a large number of defaults which could compromise the complete operating system.

Section §2 of this document provides an overview of the guarantees provided. Section §4 describes the perimeter of the analysis performed on the GT9xx driver. Sections §5, §6 and §7 provide descriptions of the internal specifications and models used for the analysis. Section §8 describes the model to exhaustively checks the GT9xx driver.

2 Analysis results

This report states the total immunity of the GT9xx driver to the set of security weaknesses enumerated below when used according to the model and perimeter of this analysis.

- CWE-119 Improper Restriction of Operations within the Bounds of a Memory Buffer
- CWE-120 Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
- CWE-121 Stack-based Buffer Overflow
- CWE-122 Heap-based Buffer Overflow
- CWE-123 Write-what-where Condition
- CWE-124 Buffer Underwrite ('Buffer Underflow')
- CWE-125 Out-of-bounds Read
- CWE-127 Buffer Under-read
- CWE-369 Divide By Zero
- CWE-415 Double Free
- CWE-416 Use After Free
- CWE-457 Use of Uninitialized Variable
- CWE-476 NULL Pointer Dereference
- CWE-562 Return of Stack Variable Address
- CWE-690 Unchecked Return Value to NULL Pointer Dereference

3 Technical Overview

The GT9xx driver is a driver for the Linux kernel, for Goodix Touchscreen drivers.

The source code used for the analysis is taken from this repository: https://source.codeaurora.org/quic/la/kernel/msm-3.18/tree/drivers/input/touchscreen/gt9xx?h=LA.HB.1.1.1.c2.

The commit used for the analysis is `f7d281d16eff5031b39c41e6af6c527ecec31385`.

The data-sheet used to model a GT915 driver is: http://www.datasheetspdf.com/PDF/GT915/945606/1

4 Perimeter of the Analysis

The analysis scope is a proof-of-concept verification for a specific GT915 configuration.

To perform an analysis we need to define a `main` function and stubs. There are two kind of stubs:

1. The stubs needed to simulate the Linux Kernel APIs
2. The stubs needed to simulate the behavior of the Hardware

At the Kernel level no parallelism is simulated, *i.e.* the Kernel is supposed not to be running in a SMP system.

The interrupt handlers are called sequentially after the initialization of the driver.

No verification is done to check if the Linux Kernel APIs are called properly by the driver.

The Kernel simulation is very coarse.

At the hardware level, the simulation is based on the GT915 data-sheet.

The testing driver is implemented in function `main`.

- It calls the initialization function of the driver
- It forces the probing of the hardware
- It calls infinitely the IRQ handler

The behavior of the driver is very dependent on the Linux Device Tree. A fragment of this Device Tree is defined for the analysis.

5 Compilation options

The GT9xx driver is analyzed with the following configuration.

```
CPP_EXTRA=(
    -D__CHECKER__
    -nostdinc
    -isystem /usr/lib/gcc/x86_64-linux-gnu/5/include
    -I./arch/x86/include
    -Iarch/x86/include/generated
    -Iinclude
    -I./arch/x86/include/uapi
    -Iarch/x86/include/generated/uapi
    -I./include/uapi
    -Iinclude/generated/uapi
    -include ./include/linux/kconfig.h
    -D__KERNEL__
    -m64
    -DCONFIG_AS_CFI=1
    -DCONFIG_AS_CFI_SIGNAL_FRAME=1
    -DCONFIG_AS_CFI_SECTIONS=1
    -DCONFIG_AS_FXSAVEQ=1
    -DCONFIG_AS_CRC32=1
    -DCONFIG_AS_AVX=1
    -DCONFIG_AS_AVX2=1
    "'-DKBUILD_STR(s)=#s'"
    "'-DKBUILD_BASENAME=KBUILD_STR(gt9xx)'"
    "'-DKBUILD_MODNAME=KBUILD_STR(gt9xx)'"
    -Dvolatile=
)
```

6 Kernel Modeling

This section describes the necessary stubs performed to simulate the Linux Kernel APIs.

6.1 Kernel Level Stubs

Basic Kernel work queues are modeled. Only one element in the queue needs to be handled.

```
bool
queue_work_on(int cpu, struct workqueue_struct *wq,
              struct work_struct *work)
{
    if(work)
        work->func(work);
    return 0;
}
```

Figure 1: Example of basic model for the `queue_work_on` function.

List of modeled functions:

- ```
  bool
  queue_work_on(int cpu, struct workqueue_struct *wq,
                struct work_struct *work)
  ```
- ```
  struct workqueue_struct *
  __alloc_workqueue_key(const char *fmt, unsigned int flags, int max_active,
                        struct lock_class_key *key, const char *lock_name, ...)
  ```
- `bool cancel_work_sync(struct work_struct *work)`
- `void flush_workqueue();`
- `void destroy_workqueue();`

6.2 Kernel mutex

Kernel mutex are ignored in this analysis.

```
void mutex_lock(struct mutex *lock)
{
    return;
}
```

Figure 2: Example of basic model for the `mutex_lock` function.

List of modeled functions:

- `void mutex_lock(struct mutex *lock)`
- `void mutex_unlock(struct mutex *lock)`
- ```
  void __mutex_init(struct mutex *lock, char const *name,
                    struct lock_class_key *key)
  ```
  ```

6.3 Linux Input Driver

Linux input driver functions are modeled in a way that their execution is always a success.

List of functions modeled:

- `void input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value)`
- `void input_mt_report_slot_state(struct input_dev *dev, unsigned int tool_type, bool active)`
- `void input_set_abs_params(struct input_dev *dev, unsigned int axis, int min, int max, int fuzz, int flat)`
- `int input_mt_init_slots(struct input_dev *dev, unsigned int num_slots, unsigned int flags)`
- `struct input_dev *input_allocate_device(void)`
- `int input_register_device(struct input_dev *)`

6.4 Linux Device

Linux device functions are modeled in a way that their execution is always a success.

List of modeled functions:

- `int dev_set_name(struct device *dev, const char *name, ...)`
- `void device_initialize(struct device *dev)`

6.5 Linux GPIO

Linux GPIO functions are modeled in a way that their execution is always a success.

Also the direct access to the Device Tree is modeled as follow.

```
/* Direct access to the Device Tree. */
int of_get_named_gpio_flags(struct device_node *np,
                            char const *list_name, int index,
                            enum of_gpio_flags *flags)
{
    /* Hard-coding some nodes of the device tree in order
       to short-cut the Kernel behavior. */
    if (strcmp(list_name, "reset-gpios") == 0) {
        return 16;
    } else if (strcmp(list_name, "interrupt-gpios") == 0) {
        return 2;
    }
    return -1;
}
```

Figure 3: Example of basic model for the `of_get_named_gpio_flags` function.

List of modeled functions:

- `int gpiod_direction_input(struct gpio_desc *p)`

- `int gpiod_direction_output_raw(struct gpio_desc *desc, int value)`
- `struct gpio_desc *gpio_to_desc(unsigned gpio)`
- `int gpio_request( unsigned x,char const *  y)`
- `void gpio_free(unsigned int gpio);`
- `struct gpio_desc *gpio_to_desc();`
- `int gpiod_direction_output_raw();`
- `int of_get_named_gpio_flags(struct device_node *np,`
  `char const *list_name, int index,`
  `enum of_gpio_flags *flags)`

6.6 IRQ Management

The IRQ array is modeled as an array of length 2, with only one handler.

```
int request_threaded_irq(unsigned int irq,
                         irqreturn_t (*handler)(int , void *),
                         irqreturn_t (*thread_fn)(int , void *),
                         unsigned long flags, char const *name,
                         void *dev)
{
    if (irq >= TIS_MAX_IRQ || tis_irqs_handlers[irq]) {
        printf("Cannot allocate irq %u", irq);
        return 1;
    }
    tis_irqs_handlers[irq] = thread_fn;
    tis_irq_args[irq] = (void *)dev;
    return 0;
}
```

Figure 4: Example of basic model for the `request_threaded_irq` function.

List of modeled functions:

- `int request_threaded_irq(unsigned int irq,`
  `irqreturn_t (*handler)(int , void *),`
  `irqreturn_t (*thread_fn)(int , void *),`
  `unsigned long flags, char const *name,`
  `void *dev)`
- `void enable_irq(unsigned int irq)`
- `void disable_irq_nosync(unsigned int irq)`

6.7 I2C Kernel Simulation

The I2C Kernel transfer function is modeled to call directly the GT9xx transfer function.

```
static struct i2c_driver TIS_I2C_DRIVER = {0};

int i2c_register_driver(struct module *module, struct i2c_driver *driver) {
    TIS_I2C_DRIVER = *driver;
    return 0;
}

int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs,
                 int num)
{
    return adap->algo->master_xfer(adap, msgs, num);
}
```

Figure 5: Example of basic model for the I2C Kernel simulation.

6.8 Kernel Memory Management

The Kernel memory management functions are modeled in a way that their execution is always a success and are mapped on the `calloc` function.

List of modeled functions:

- `void *calloc(size_t nmemb, size_t size);`
- `void *devm_kmalloc(struct device *dev, size_t size, gfp_t gfp)`
- `void *__kmalloc(size_t size, gfp_t gfp)`

6.9 Kernel Timer Management

Kernel timer management functions are modeled in a way that their execution is always a success.

List of modeled functions:

- `void init_timer_key(struct timer_list *timer, unsigned int flags,`
  `char const *name, struct lock_class_key *key)`
- `void msleep(unsigned int);`
- `int usleep(int);`

6.10 Device Tree

A Device Tree is a linked list of properties. The properties are supposed to be provided by the Device Tree at boot time. For this analysis, the properties are hard-coded.

The properties and their values are extracted from *Documentation/devicetree/bindings/input/touchscreen/gt9xx/gt9xx.txt*. Changing them will impact the coverage of the analysis.

```
uint32_t tis_XXX[] = { 6666 } ; // Dummy property for testing purpose.
struct property tis_prop_XXX =
{ .name = "XXXX",
  .length = sizeof(tis_XXX),
  .value = tis_XXX,
  .next = 0
};

char tis_product_id[] = "915" ;
struct property tis_prop_product_id =
{ .name = "goodix,product-id",
  .length = sizeof(tis_product_id),
  .value = tis_product_id,
  .next = &tis_prop_XXX
};

uint8_t tis_driver_send_cfg = 0 ;
struct property tis_prop_driver_send_cfg =
{ .name = "goodix,driver-send-cfg",
  .length = sizeof(char),
  .value = tis_driver_send_cfg,
  .next = &tis_prop_product_id
};

uint8_t tis_cfg_data0[] = {
    0x41, 0xD0, 0x02, 0x00, 0x05, 0x0A, 0x05, 0x01, 0x01, 0x08, 0x12,
    0x58, 0x50, 0x41, 0x03, 0x05, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x8C, 0x2E, 0x0E, 0x28, 0x24, 0x73,
    0x13, 0x00, 0x00, 0x00, 0x83, 0x03, 0x1D, 0x40, 0x02, 0x00, 0x00,
    0x00, 0x03, 0x64, 0x32, 0x00, 0x00, 0x00, 0x1A, 0x38, 0x94, 0xC0,
    0x02, 0x00, 0x00, 0x00, 0x04, 0x9E, 0x1C, 0x00, 0x8D, 0x20, 0x00,
    0x7A, 0x26, 0x00, 0x6D, 0x2C, 0x00, 0x60, 0x34, 0x00, 0x60, 0x10,
    0x38, 0x68, 0x00, 0xF0, 0x50, 0x35, 0xFF, 0xFF, 0x27, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x01, 0x1B, 0x14, 0x0C, 0x14, 0x00, 0x00, 0x01,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x02, 0x04, 0x06, 0x08, 0x0A, 0x0C, 0x0E, 0x10, 0x12,
    0x14, 0x16, 0x18, 0x1A, 0x1C, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x00,
    0x02, 0x04, 0x06, 0x08, 0x0A, 0x0C, 0x0F, 0x10, 0x12, 0x13, 0x14,
    0x16, 0x18, 0x1C, 0x1D, 0x1E, 0x1F, 0x20, 0x21, 0x22, 0x24, 0x26,
    0x28, 0x29, 0x2A, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0x22, 0x22, 0x22, 0x22, 0x22, 0x22, 0xFF, 0x07, 0x01
};
```

```
struct property tis_prop_cfg_data0 =
{ .name = "goodix,cfg-data0",
  .length = sizeof(tis_cfg_data0),
  .value = tis_cfg_data0,
  .next = &tis_prop_driver_send_cfg
};

uint32_t tis_buttons[3]  = { 55, 67, 77 };
struct property tis_prop_button =
{ .name = "goodix,button-map",
  .length = 3*4,
  .value = tis_buttons,
  .next = &tis_prop_cfg_data0
};

uint32_t reset_gpios[]  = { 16 };
struct property tis_prop_reset_gpios =
{ .name = "reset-gpios",
  .length = sizeof(reset_gpios),
  .value = reset_gpios,
  .next = &tis_prop_button
};

uint32_t display_coords[4]  = { 666 };
struct property tis_prop_display_coords =
{ .name = "goodix,display-coords",
  .length = 4*4,
  .value = display_coords,
  .next = &tis_prop_reset_gpios
};

uint32_t panel_coords[4]  = { 666 };
struct property tis_prop_panel_coords =
{ .name = "goodix,panel-coords",
  .length = 4*4,
  .value = panel_coords,
  .next = &tis_prop_display_coords
};
```

7 Hardware Modeling

This section describes the necessary stubs performed to simulate the behavior of the hardware.

Goodix hardware simulation is inspired from the data-sheet for GT915 (http://www.datasheetspdf.com/PDF/GT915/945606/1).

It simulates:

- A fixed configuration
- Any user events on the device

The `GUP_REG_FW_MSG` (0x41E4) register is ignored because no documentation about it was found.

Only one 16 bits register defines the state of the hardware in the analysis model.

It contains the address of the hardware register that needs to be read when an I2C read is requested.

It is written whenever the I2C bus is sending a write command containing at least 2 bytes.

```
uint16_t tis_goodix_register = 0;
int tis_master_xfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num)
{
    for (int message_nb=0 ; message_nb < num ; ++message_nb) {
        struct i2c_msg message = msgs[message_nb];
        printf("TIS_Flags %d: %x ", message_nb, (unsigned int)message.flags);
        if (message.flags & I2C_M_RD) {
            switch (tis_goodix_register) {
            case 0x8047: // Cf. GT915 Doc page 11 Section 6.2.b
                printf("TIS_Config_Version %x", (unsigned)tis_goodix_register);
                message.buf[0] = 42;
                break;
            case 0x8144: // Cf. GT915 Doc page 16 Section 6.2.c
                printf("TIS_Firmware_Version %x", (unsigned)tis_goodix_register);
                message.buf[0] = 0x1;
                message.buf[1] = 0x2;
                break;
            case 0x8140: // Cf. GT915 Doc page 16 Section 6.2.c
                printf("TIS_Firmware_Version %x", (unsigned)tis_goodix_register);
                message.buf[0] = '9';
                message.buf[1] = '1';
                message.buf[2] = '5';
                message.buf[3] = '\0';
                break;
            case 0x814e: // Cf. GT915 Doc page 17 Section 6.2.c
                printf("TIS_get_hw_point_1 %x", (unsigned)tis_goodix_register);
                tis_make_unknown(message.buf, message.len);
                break;
            case 0x8158: // Cf. GT915 Doc page 17 Section 6.2.c
                printf("TIS_get_hw_point_2 %x", (unsigned)tis_goodix_register);
                tis_make_unknown(message.buf, message.len);
                break;
            default:
                printf("TIS_Unknown_Register %x",
                       (unsigned)tis_goodix_register);
            }
        } else {
            if (message.len >= 2) {
                tis_goodix_register = (message.buf[0] << 8) + message.buf[1];
                printf("TIS_set_hw_register: %x ",
                       (unsigned int)tis_goodix_register);
                if (message.len > 2)
                    printf("TIS_writing %d other bytes", message.len - 2);
            } else {
                printf("TIS_Writing command too small %d", message.len);
            }
        }
    }
    return num;
}
u32 tis_functionality(struct i2c_adapter *adapt) {
    return 1;
}
```

8 Analysis Entry Point

Eventually, the testing driver is modeling an entry point for the analysis and:

1. A call to the initialization function of the GT9xx driver
2. A probe of the hardware
3. An infinite number of calls to the IRQ handler

```
/* Definition of the basic data structures used by the Kernel to
   support the hardware */
struct goodix_ts_data tis_goodix_ts_data = { .abs_x_max = 4096 };
struct device_node tis_dev_node = {
    .properties = &tis_prop_panel_coords,
    .data = &tis_goodix_ts_data
};
struct i2c_algorithm tis_algo = {
    .functionality = &tis_functionality,
    .master_xfer = &tis_master_xfer,
    .smbus_xfer = 0
};
struct i2c_adapter tis_adapter =
{ .owner = 0,
  .algo = &tis_algo,
  .name = "TIS Adapter"
};
struct i2c_client tis_client =
{ .flags = 0,
  .addr = 0,
  .name = "tis_client",
  .adapter = &tis_adapter,
  .dev = { .of_node = &tis_dev_node },
  .irq = 0,
  .detected = { 0 }
};

/* This is the entry point for the analysis */
int main() {
    /* Force this initialzation of the driver */
    int result = goodix_ts_init();
    /* Probe the hardware */
    TIS_I2C_DRIVER.probe(&tis_client, 0);

    /* Infinite loop calling the interrupt handlers. */
    while(1)
        for (unsigned int irq = 0 ; irq < TIS_MAX_IRQ; ++irq)
            if (tis_irqs_handlers[irq])
                (*tis_irqs_handlers[irq])(irq, tis_irq_args[irq]);

    return result;
}
```

9 Conclusion

TrustInSoft Analyzer guarantees the absence of undefined behavior for the GT9xx driver, according to the C89 standard, when used in an environment that is identical to the model.