CENSUS
IT Security Works

Android EL0 and EL1 debugging nightmares

Chariton Karamitas
huku@census.gr

March 23, 2018

Table of contents

1. Whoami
2. Problem statement
3. Preparation
   - Patching GDB
     - Cross-compiling GDB
4. Android userland (EL0) debugging
   - Native code debugging
     - Initiating a debugging session
   - ART debugging
     - ro.debuggable, JDWP and JDB
     - Placing runtime Java hooks
   - Automation
   - Demo
5. Android kernel (EL1) debugging
   - Kernel debugging subsystem internals
   - Debugging over ttyUSB
   - Debugging over ttyGS
   - Automation
   - Demo
6. Conclusion
7. Future
8. Questions & Thanks

Whoami

- Electrical Engineer
- Working for CENSUS
- Interested in reverse engineering, maths and exploit development
- Proud Phrack author
- huku@census.gr and huku@grhack.net
- https://github.com/huku-/

Problem statement

What's the problem?

- The Android ecosystem is composed of several players (hardware vendors, OEMs, carriers, etc.)
  - Endless customizations on various Android components
- Admittedly, there's a lack of:
  - Standards and procedures (e.g. OEM unlock, firmware updates)
  - Public resources (tools, documentation etc.) but for high level stuff
  - Usable interfaces (only USB, headset jack on Nexus)

What do we need?

Ultimate goal

Make exploit development easier

- More specific goals:
  - Be able to debug a userland (EL0) application
  - Be able to debug a kernel (EL1) on a live production device
  - Automate debugging tasks on EL0 and EL1
- We need a well designed, device agnostic methodology for doing all the above
  - Start from a developer-friendly device (e.g. Samsung S7 Exynos)
  - Generalize and apply the results to devices with stricter security policies (e.g. Samsung S7 Qualcomm)
- Push vendors to enable debugging facilities on Android phones by default!
  - According to the Android CDD (Compatibility Definition Document), `systrace` (which uses ftrace) must be supported on all devices! This a good step!

How to get it?

Means

Turn a production device into a fully configurable debugging environment by gradually bypassing its protections one by one

- Steps (not necessarily in this order; actual order may vary depending on the device's security features):
  - Get root
  - Boot custom kernel
  - Setup kernel debugging
  - Enable userland debugging
  - Setup userland debugging
  - Automate using ADB, JDB, GDB, Python etc.
- We assume:
  - Device has been rooted
  - SELinux has been bypassed

So what?

- Not interested in showing off about how I exploited a specific vulnerability
  - If you can setup a functional debugging environment, then you can easily write your own exploits
  - ...but still, there's Google Project Zero if you like
- This presentation summarizes part of our 1-year research on Android debugging and exploitation
  - Many details have been left out for brevity
  - Many things might seem obvious at first sight, but keep in mind that is hard to debug a kernel when you don't have a debugger
  - Interested in discussing ideas
  - Don't hesitate to contact me if you would like to discuss privately

Preparation

Preparation

- Need to cross-compile GDB
  - Host Darwin/Linux, target ARM/Aarch64 (remote debugging)
  - Host ARM/Aarch64, target ARM/Aarch64 (local debugging)
- Required for both kernel and userland debugging
- Steps
  1. Patch GDB source
  2. Prepare GCC hook scripts to force PIE (where is `--enable-pie`?)
  3. Prepare environment (CC, CFLAGS, CXX, CXXFLAGS, etc.)
  4. Build and deploy

Patching GDB

Bug #1 - Breakpoints just don't work

- GDB developers still believe Rust support is of higher priority than ARM/Aarch64 breakpoint support
- Breakpoints hit on debuggee but never caught on debugger
  - `CORE_ADDR` for stored Thumb mode breakpoint holds address with low bit set, while `CORE_ADDR` of current PC does not
  - `ptrace(PTRACE_CONT, ...)` signal delivery not working
  - Quoting `ptrace(2)`: "If data is nonzero, it is interpreted as the number of a signal to be delivered to the tracee;"
  - Debuggee remains in a frozen state forever
    - `kill -CONT` also caught by GDB and never delivered
    - Debugger process acts like parent in signal delivery

Patching GDB

Bug #1 - Breakpoints just don't work

gdb/breakpoint.c

```
1  int
2  breakpoint_address_match (struct address_space *aspace1, CORE_ADDR addr1,
3                            struct address_space *aspace2, CORE_ADDR addr2)
4  {
5    /* If target is ARM and this is a Thumb breakpoint, adjust PC. */
6    if (gdbarch_bfd_arch_info (target_gdbarch ())->arch == bfd_arch_arm
7        && (addr1 & 1))
8      addr1 -= 1;
9
10   return ((gdbarch_has_global_breakpoints (target_gdbarch ())
11         || aspace1 == aspace2)
12        && addr1 == addr2);
13 }
```

Patching GDB

Bug #2 - CPSR bit width

- On Aarch64, representing PSTATE requires more than 32-bits (see the ARMv8 specification, chapter D1.7), but GDB's XML configuration specifies CPSR for Aarch64 as a 32-bit quantity
- Confuses GDB when connected to KGDB

gdb/features/aarch64-core.xml

```
-  <reg name="cpsr" bitsize="32"/>
+  <reg name="cpsr" bitsize="64"/>
```

Patching GDB
Other modifications

- Made other minor modifications so that GDB could be cross-compiled with a standard NDK standalone toolchain
  - Fixed use of undefined `ptrace()` constants
  - Replaced undefined `W_STOPCODE()` macro with `WSETSTOP()`
  - Made GDB treat `td_thrhandle_t` as opaque object
  - Removed unresolved calls to `__fsetlocking()` (why do we need this anyway?)
  - Removed unresolved calls to `setpwent()`
  - Tweaked a few Makefiles
- Hopefully all this will be published in the near future

Cross-compiling GDB
Forcing PIE

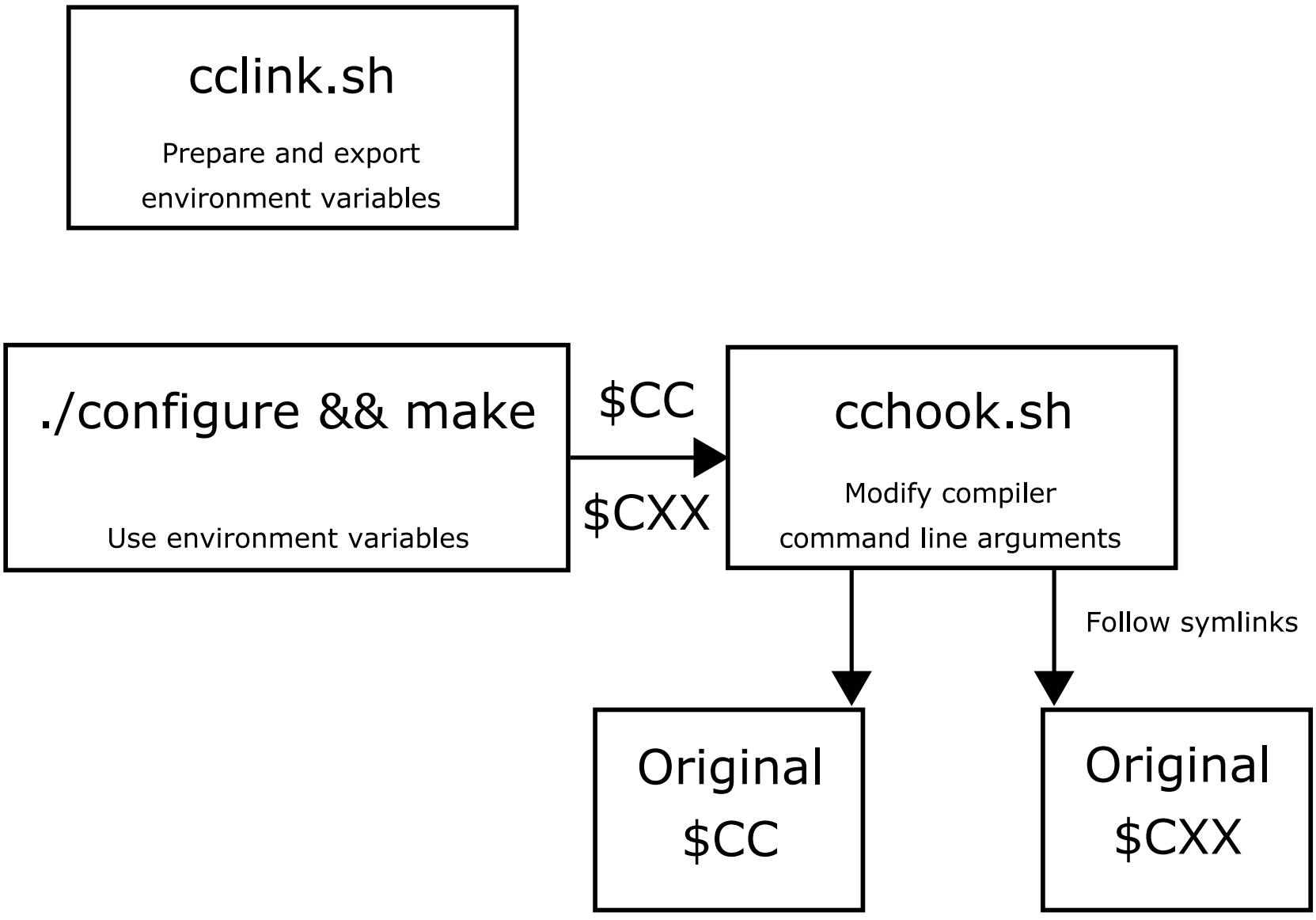
Compiler trickery used to actually compile GDB as PIE

Prepare GCC hook script

```
$ ls -l cchook-arm/
total 80
lrwxr-x--- ... cc -> cchook.sh
-rwxr-x--- ... cchook.sh
-rw-r----- ... env.sh
lrwxr-x--- ... g++ -> cchook.sh
lrwxr-x--- ... gcc -> cchook.sh
lrwxr-x--- ... ld -> cchook.sh
lrwxr-x--- ... real_cc -> real_gcc
lrwxr-x--- ... real_g++ -> /toolchain-arm/bin/arm-linux-androideabi-g++
lrwxr-x--- ... real_gcc -> /toolchain-arm/bin/arm-linux-androideabi-gcc
lrwxr-x--- ... real_ld -> /toolchain-arm/bin/arm-linux-androideabi-ld
```

Cross-compiling GDB

Forcing PIE



Cross-compiling GDB

Forcing PIE

cchook.sh

```
local extra_ops=
if [[ "$program" =~ -?g?cc$ || "$program" =~ -?g\+\+$ ||
        "$program" =~ -?ld$ ]]; then
    local link=1
    local shared=0
    for opt in "$@"; do
        if [[ "$opt" = "-shared" ]]; then
            shared=1
        elif [[ "$opt" = "-c" ]]; then
            link=0
        fi
    done

    if [[ $link -eq 1 && $shared -eq 0 ]]; then
        extra_ops="-pie"
    fi
fi

...

"$program" $@ $extra_ops
```

Cross-compiling GDB

Forcing PIE

Prepare environment

```
$ cd cchook-arm/
$ export CROSS_COMPILE=/toolchain-arm/bin/arm-linux-androideabi-
...

$ export d=$(pwd)
$ export CC="$d/gcc"
$ export CXX="$d/g++"
$ export LD="$d/ld"
$ export AS="${CROSS_COMPILE}as"
$ export AR="${CROSS_COMPILE}ar"
$ export RANLIB="${CROSS_COMPILE}ranlib"
$ export NM="${CROSS_COMPILE}nm"
$ export OBJDUMP="${CROSS_COMPILE}objdump"
```

Cross-compiling GDB
Building

- Always use well defined triplets/quadruplets in `configure` options
- Not doing so may break certain Makefiles

Prepare environment

```
$ ./configure --prefix=/usr --host=arm-none-linux-androideabi \
    --target=arm-none-linux-androideabi
$ ./configure --prefix=/usr --host=aarch64-none-linux-androideabi \
    --target=aarch64-none-linux-androideabi
$ make
```

Cross-compiling GDB
Building

Voila!

Compiled binaries

```
$ find bins/ -type f
bins/host-aarch64-target-aarch64/gdb-aarch64
bins/host-aarch64-target-aarch64/gdbserver-aarch64
bins/host-arm-target-arm/gdb-arm
bins/host-arm-target-arm/gdbserver-arm
bins/host-darwin-target-aarch64/gdb
bins/host-darwin-target-arm/gdb
```

Android userland (EL0) debugging

Initiating a debugging session
GDB 101

Once our modified GDB has been compiled, debugging a userland application is just a matter of firing `gdbserver`

Debuggee (Samsung S7)

```
# cd /data/local/tmp
# ./gdbserver-arm --attach *:31337 $(pgrep whatever)
```

Debugger (Ubuntu)

```
$ adb forward tcp:31337 tcp:31337
$ gdb
...

(gdb) target remote 127.0.0.1:31337
```

GDB 101

- Ctrl+C will not work (see how signal delivery works on `ptrace()`'d processes for more information)
- Use `kill -TRAP` instead

Debugger (Ubuntu)

```
(gdb) set pagination off
(gdb) handle SIG33 nostop
(gdb) continue
```

Debuggee (Samsung S7)

```
# kill -TRAP $(pgrep whatever)
```

ro.debuggable, JDWP and JDB

Enabling ro.debuggable

- Unfortunately `ro.debuggable` is not set on production devices
- Property service managed by `init`
- On newer kernels we can `ptrace()` PID 1, and modify the property store mapped from `/dev/__properties__`
  - Format varies depending on the Android version
  - Tree vs. array storage
- Of course, we need root and SELinux bypass first

Debuggee (Samsung S7)

```
$ ./setprop
[*] Getting root ;)
[*] I haz UID 0
[*] Attaching to PID 1
[*] Locating property tree node
[*] Currently @node "", left=+0, right=+0, children=+0x14
[*] Currently @node "ro", left=+0x5734, right=+0x1AA8, children=+0x2C
[*] Traversing /dev/__properties__ tree
[*] Currently @node "ro", left=+0x5734, right=+0x1AA8, children=+0x2C
...
[*] Traversing /dev/__properties__ tree
...
[*] Currently @node "debuggable", left=+0, right=+0x16ED0, children=+0
[*] Current value "0"
[*] Updating value
[*] Verifying
[*] Updated value "1"
[*] Success!
```

ro.debuggable, JDWP and JDB

Attaching

Last but not least, attach to the JVM using JDB

Debugger (Ubuntu)

```
$ adb jdwp
...
$ adb forward tcp:31337 jdwp:whatever
$ jdb -attach 127.0.0.1:31337
```

ro.debuggable, JDWP and JDB

Limitations

- JDB is practically unusable when you don't have the source
  - No way to insert breakpoints at arbitrary locations
  - No way to examine locals (only `this` is available)
- JDB is unusable anyway
  - Super difficult to handle overloaded methods
  - No way of automating debugging tasks (combining `monitor` and `read` commands randomly switches between threads)
  - Obviously, `redefine` not implemented in ART
  - I guess not even JDB developers use JDB
- How come noone solved this problem yet?

Placing runtime Java hooks

Public domain tools

- The limitations of JDB can only be avoided using a Java hooking framework
- Unfortunately Cydia substrate just doesn't work
  - ...and Saurik won't even reply to e-mails but who gives a fuck anyway?
- Xposed modifies the filesystem and triggers dm-verity failures
- ARTDroid kept crashing (found several problems in its source code in just a few seconds)
- Frida has limited ART support but its API looks really good
- Legend also seems like a good choice
- Didn't have enough time to evaluate all these frameworks (TODO)

Placing runtime Java hooks
Our technique

- Developed a simple DSO which can be injected in an Android process using injectoid to hook arbitrary Java methods
  - Technique is almost similar to Legend's
- Steps:
  1. Get Java VM pointer using JNI's `JNI_GetCreatedJavaVMs()`
  2. Hook `AttachCurrentThread()` to catch newly created threads
  3. Wait until a thread that uses a class loader other than Java's default is detected
  4. Unhook `AttachCurrentThread()`
  5. Load custom Java classes from custom JAR/APK using application's class loader
     - This is where our custom Java hooks should be defined
  6. Modify ART's `ArtMethod` instances to redirect application methods to our Java hooks

Placing runtime Java hooks
Automation

- Still haven't implemented any automation features
- We will have a look at Frida's ART hooking support first and act accordingly
  - Patch/extend Frida
  - Develop custom tool with Python API

DEMO

Android kernel (EL1) debugging

Kernel debugging subsystem internals
Kernel debugging core

- Kernel debugging subsystem at `kernel/debug/debug_core.c` (comes with GDB stub at `kernel/debug/gdbstub.c`)
- Kernel parameter parsing in `drivers/tty/serial/kgdboc.c` (calls `param_set_kgdboc_var()`)
- Calls `configure_kgdboc()` that locates the appropriate TTY polling driver (more on this later)
- Calls `kgdb_register_io_module()` that installs the `struct kgdb_io` callbacks for GDB protocol I/O
  - They just off-load the actual task to the TTY driver
- Calls `kgdb_register_callbacks()` which also registers the KGDB console if enabled
  - It's just a `printk()`-supported console driver
  - Use the kgdb-agent-proxy or a custom Python script to receive, parse and display messages as GDB just ignores them (this world is full of surprises)

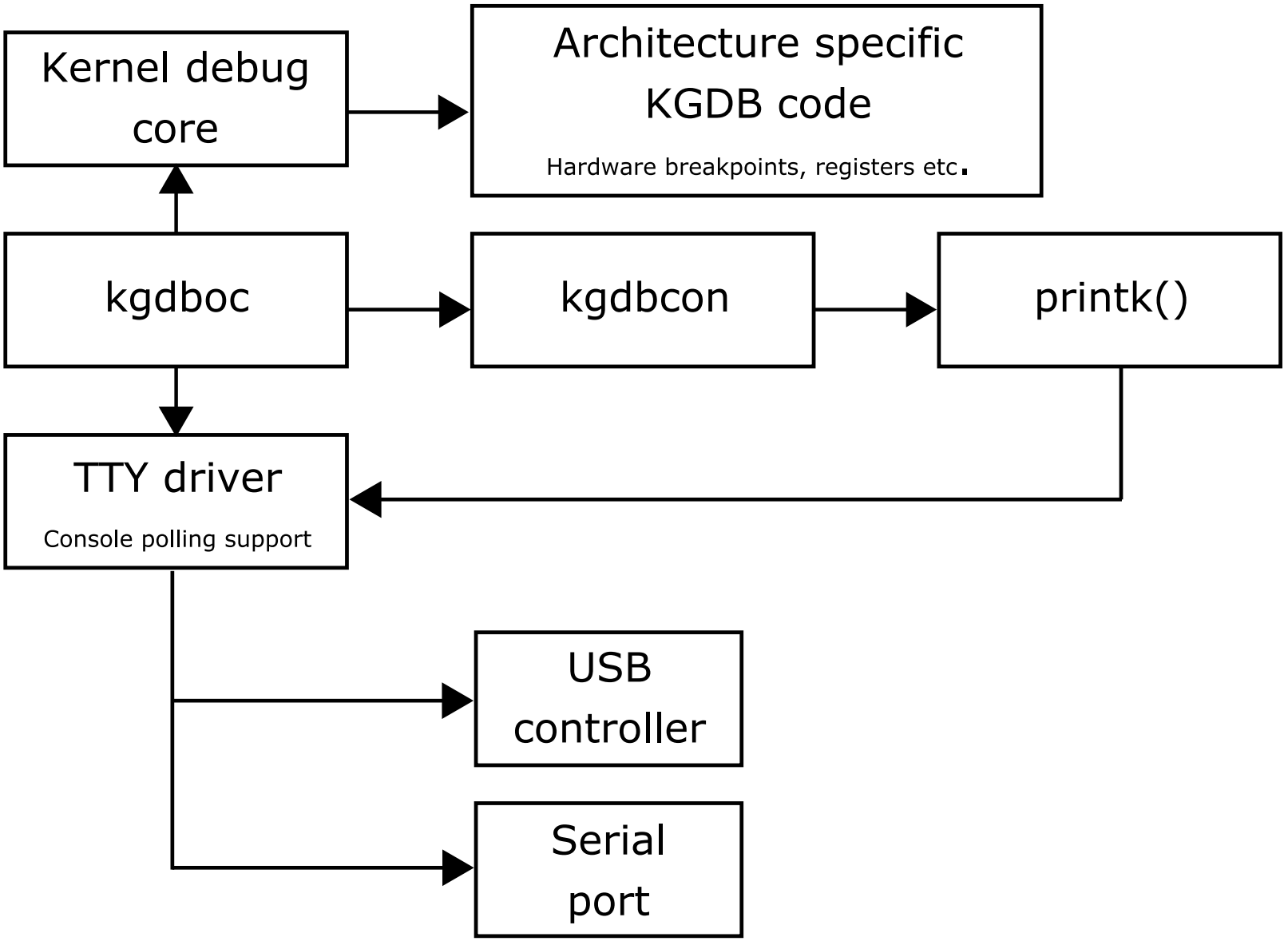
Kernel debugging subsystem internals

TTY operations & console polling support

- TTY drivers that support console polling should implement the following callbacks in their `struct tty_operations`
  - `poll_init()` - Receives the string passed to `kgdboc` kernel boot parameter
    - On the contrary to what kernel-parameters.txt mentions, this boot parameter can have any format
  - `poll_get_char()` - Synchronously read one character (should block until the character is received)
  - `poll_put_char()` - Synchronously write one character (should block until the character has been received from the remote end)
- More information can be found in Documentation/serial/driver
- Quoting Using kgdb and the kgdb internals by Jasson Wessel, Tom Rini & Amit S. Kale: "Keep in mind that polling hooks have to be implemented in such a way that they can be called from an atomic context and have to restore the state of the UART chip on return such that the system can return to normal when the debugger detaches."

Kernel debugging subsystem internals

Architecture



Kernel debugging subsystem internals
Sample

Sample driver at `drivers/tty/serial/serial_core.c`

TTY operations

```
1  static const struct tty_operations uart_ops = {
2      ...
3  #ifdef CONFIG_CONSOLE_POLL
4      .poll_init = uart_poll_init,
5      .poll_get_char = uart_poll_get_char,
6      .poll_put_char = uart_poll_put_char,
7  #endif
8  };
```

KGDB prerequisite

To enable KGDB, we first need to implement console polling support in the available serial driver(s)

- Steps
  1. Locate the serial port hardware
  2. Locate serial driver in the Linux kernel source tree
  3. Implement console polling support
  4. Recompile and boot custom kernel (boot parameters can be hardcoded into the kernel using `CONFIG_CMDLINE`)
- Let's have a look at `ttyUSB` and `ttyGS`

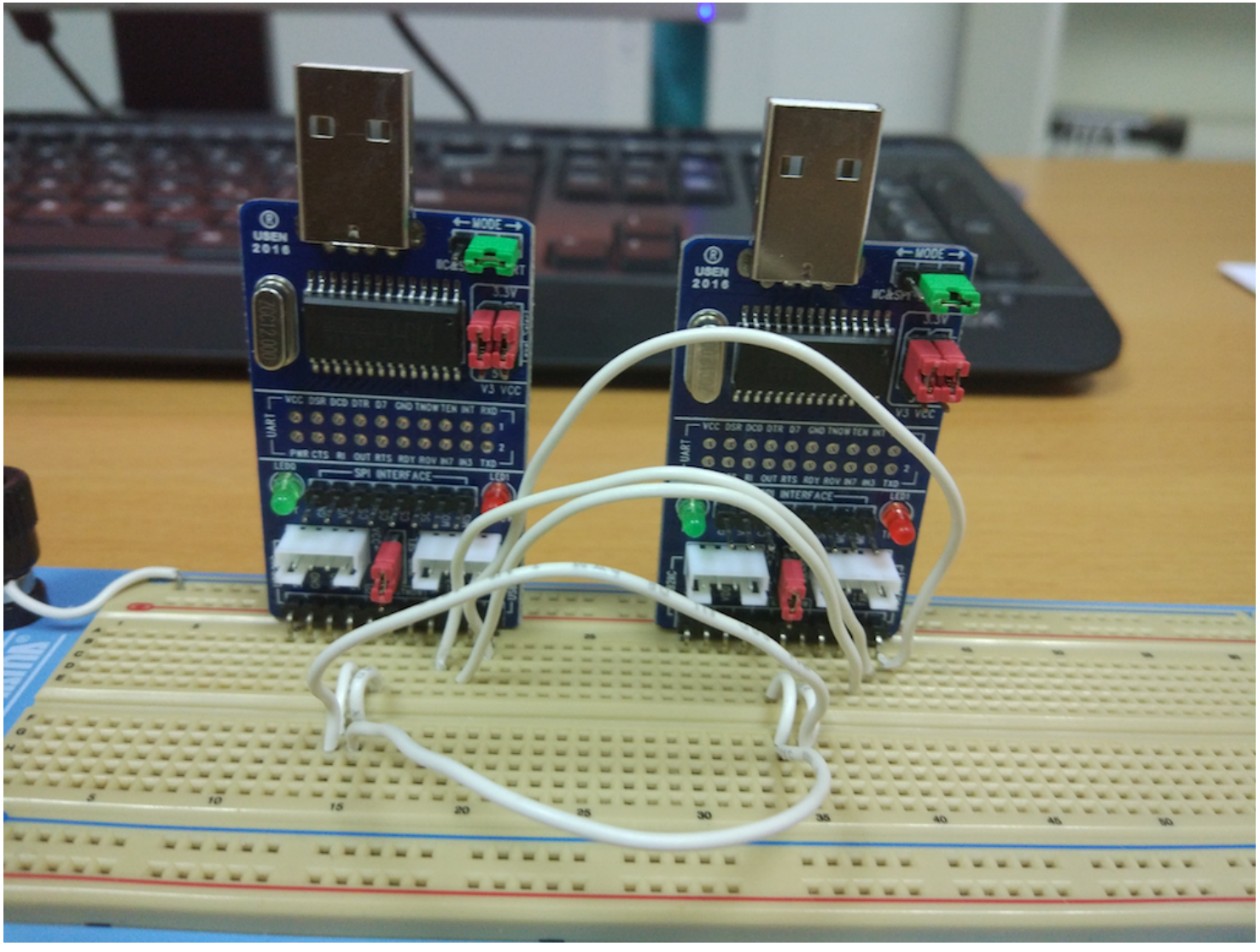
Debugging over ttyUSB

Idea #1

- Most (all?) Android devices come with support for USB OTG (On-The-Go)
- Why not plug an all-in-one USB to SPI/I2C/UART YSUMA01-341A?
  - Driver source already in vanilla at `drivers/usb/serial/ch341.c`
- Device may or may not function depending on device's kernel configuration
  - Requires `USB_SERIAL` & `USB_SERIAL_CH341`
- Bridged two USB-to-serial adapters using a raster board and cables (could have been better with dupont cables though)
  - Connected one end to Android device with custom kernel and the other end on an Ubuntu Desktop host
- Let's use `ttyUSB0` for KGDB I/O

Debugging over ttyUSB

Test setup



Debugging over ttyUSB

Test setup

Issued the following commands but no data arrived on the debugger host

Debugger (Ubuntu)

```
# stty -F /dev/ttyUSB0 115200 raw -echo crtscts
# cat /dev/ttyUSB0
```

Debuggee (Samsung S7)

```
# stty -F /dev/ttyUSB0 115200 raw -echo crtscts
# echo -n test > /dev/ttyUSB0
```

Debugging over ttyUSB

USB-to-serial subsystem modifications

- CH341 is part of the kernel's generic USB-to-serial support
  - Bulk transfer mode used by default by the generic driver (even though USB device descriptors exported by the device also define an interrupt endpoint)
  - CH341 bulk transfer size is 16 bytes
- Modified `usb/serial/generic.c` to pad messages to 16 bytes and prefix them with a size byte
  - Messages similarly unpacked on the other side
  - Modified `generic.c` should also be used on the host side
- Also made minor modifications to `ch341.c`
- Alternatively we could have used FTDI TTL-232RG

Debugging over ttyUSB

USB-to-serial subsystem modifications

- Written URBs (USB Request Blocks) should be padded and prefixed with the actual message length to avoid caching on the hardware side

usb_serial_generic_write_start()

```
1  urb = port->write_urbs[i];
2  memset(urb->transfer_buffer, 0, port->bulk_out_size);
3  count = port->serial->type->prepare_write_buffer(port, urb->transfer_buffer,
4      port->bulk_out_size - 1);
5  *((char *)urb->transfer_buffer + port->bulk_out_size - 1) =
6      (char)(count & 0xff);
7  urb->transfer_buffer_length = port->bulk_out_size;
```

- Read URBs should be unpacked on the receiving side

usb_serial_generic_process_read_urb()

```
1  count = ch[urb->actual_length - 1];
2  if(count < 0 || count >= port->bulk_in_size)
3  {
4      ...
5  }
6
7  urb->actual_length = count;
```

Debugging over ttyUSB

USB-to-serial subsystem modifications

Hook `process_read_urb()` to append data in a KFIFO

poll_init()

```
1  static int serial_poll_init(struct tty_driver *driver, int line, char *options)
2  {
3      ...
4
5      kgdb_port = usb_serial_port_get_by_minor(line);
6      ...
7
8      spin_lock_irqsave(&kgdb_port->lock, flags);
9
10     process_read_urb = kgdb_port->serial->type->process_read_urb;
11     kgdb_port->serial->type->process_read_urb = process_read_urb_hook;
12
13     write_bulk_callback = kgdb_port->write_urbs[0]->complete;
14     for(i = 0; i < ARRAY_SIZE(kgdb_port->write_urbs); i++)
15         kgdb_port->write_urbs[i]->complete = write_bulk_callback_hook;
16
17     spin_unlock_irqrestore(&kgdb_port->lock, flags);
18     ...
19
20     kgdb_port->serial->type->open(NULL, kgdb_port);
21     ...
22 }
```

Debugging over ttyUSB

USB-to-serial subsystem modifications

Use KFIFO for KGDB I/O

poll_get_char()

```
1  static int serial_poll_get_char(struct tty_driver *driver, int line)
2  {
3      int ch = 0;
4
5      kgdb_unlock();
6
7      if(kfifo_is_empty(&kgdb_fifo) ||
8              kfifo_out_locked(&kgdb_fifo, (char *)&ch, 1, &kgdb_fifo_lock) < 1)
9          ch = NO_POLL_CHAR;
10
11     kgdb_lock();
12     return ch;
13 }
```

poll_put_char()

```
1  static void serial_poll_put_char(struct tty_driver *driver, int line, char ch)
2  {
3      kgdb_unlock();
4
5      atomic_set(&sent, 0);
6      while(kfifo_is_full(&kgdb_port->write_fifo) ||
7          usb_serial_generic_write(NULL, kgdb_port, &ch, 1) < 1) {};
8
9      while(atomic_read(&sent) != 1) {};
10
11     kgdb_lock();
12 }
```

Debugging over ttyUSB

Pros & Cons

- Pros
  1. Reliable
- Cons
  1. Requires special hardware
  2. Requires kernel source code modifications on both communicating ends
  3. Slow even when operated on 115200 bauds
- Ideally we would like a solution which is faster and less complex
  - Certain debugging tasks require large volumes of data transfer (e.g. locating a pattern in memory)
  - Trade some reliability in favor of speed

Debugging over ttyGS

Idea #2

- Most (all?) Android devices come with USB gadget support

Serial gadget device

```
$ ls -la /dev/ttyGS0
crw-rw---- system   system   229,   0 2016-02-10 22:39 ttyGS0
```

- Source code for serial port USB gadget contained in two files:
  - `drivers/usb/gadget/function/u_serial.c`
  - `drivers/usb/gadget/function/f_serial.c`
- Primarily used for communication between a `gdbserver` running on the device and a GDB client on the host
- Let's use `ttyGS0` for KGDB I/O
- Work already published by Jack Tang of Trend Micro
  - His solution is slightly more elegant than ours

Debugging over ttyGS

Serial gadget modifications

On initialization, read the DWC3 IRQ number

Find DWC3 IRQ number

```
# grep dwc3 /proc/interrupts
245:    2204    0    0    0    0    0    GIC 245  dwc3
# cat /proc/irq/245/smp_affinity
0f
```

poll_init()

```
1  static int gs_poll_init(struct tty_driver *driver, int line, char *options)
2  {
3      ...
4
5      dwc3_irq = DWC3_IRQ_DEFAULT;
6      if(options != NULL)
7          dwc3_irq = (int)simple_strtol(options, NULL, 10);
8      ...
9  }
```

Debugging over ttyGS

Serial gadget modifications

Running in atomic context; schedule DWC3 I/O IRQ to fire on a different CPU

poll_get_char()

```
1  static int gs_poll_get_char(struct tty_driver *driver, int line)
2  {
3      ...
4
5      kgdb_unlock();
6      ...
7
8      while(kfifo_is_empty(&kgdb_fifo) ||
9          kfifo_out_locked(&kgdb_fifo, &ch, 1, &kgdb_fifo_lock) < 1)
10     {
11         cpu = task_cpu(current);
12         cpumask_setall(&mask);
13         cpumask_clear_cpu(cpu, &mask);
14         irq_set_affinity(dwc3_irq, &mask);
15         ...
16
17         atomic_set(&recvd, 0);
18         while(usb_ep_queue(ep, req, GFP_ATOMIC) < 0) {}
19         while(atomic_read(&recvd) != 1) {}
20         ...
21     }
22
23     kgdb_lock();
24     return ch;
25 }
```

Debugging over ttyGS

Serial gadget modifications

Same for writing characters

poll_put_char()

```
1  static void gs_poll_put_char(struct tty_driver *driver, int line, char ch)
2  {
3      kgdb_unlock();
4      ...
5
6      cpu = task_cpu(current);
7      cpumask_setall(&mask);
8      cpumask_clear_cpu(cpu, &mask);
9      irq_set_affinity(dwc3_irq, &mask);
10     ...
11
12     atomic_set(&sent, 0);
13     while(usb_ep_queue(ep, req, GFP_ATOMIC) < 0) {}
14     while(atomic_read(&sent) != 1) {}
15     ...
16
17     kgdb_lock();
18 }
```

Debugging over ttyGS

Pros & Cons

- Pros
  1. No special hardware required
  2. Less kernel source code modifications (only in the debuggee's kernel)
  3. Much faster
- Cons
  1. Less reliable (ADB seems to mess with USB transfers and slows down things some times)
- Works quite reliably in practice
- We have been using it to automate debugging tasks on production devices

Automation

- Developed set GDB Python extensions for examining kernel structures (IDN, `kmalloc()`, etc.); code named KBBD (Kernel Black-Box Debugger)
  - Simple API for defining new kernel structures
  - Automated detection of KASLR
  - Custom commands for easier exploit development
  - Simple heap spray detection logic
  - etc.
- Additional kernel debugging scripts at `scripts/gdb`
- Production was boosted!

DEMO

Conclusion

- We have presented ways of performing reliable debugging on Android
  - Userland (EL0) components, native executables and application bundles, using GDB, JDB and Java hooks
  - The Android kernel (EL1) using GDB/KGDB and Python scripts
- We have also discussed the potential of automating debugging tasks
- Unfortunately...
  - Android is on version 7.1 and still lacks proper debugging tools... but also still lacks a sane runtime
  - Vendors follow Apple's foot-steps by introducing various incompatibilities instead of unifying their custom bullshit
  - We still have to pay hundreds of USD for a smartphone that we don't actually own... just like iPhone

Future

- Finish all our automation methods
- Maybe publish all our kernel patches, toolchain patches and Python scripts?
- Compiler toolchain on the smartphone (90% done)
- EL2/EL3 debugging (20% done)

Questions & Thanks

Questions? Kthanxbai!