# Bushwhacking your way around a bootloader

Rebecca ".bx" Shapiro

2018.04.05

How I managed to develop understanding of U-Boot

KernelCon

# > whoami

## Dr. .bx

- Senior security researcher @ Narf Industries
- Studied w/ Sergey Bratus & the Dartmouth Trust Lab
- Commander of ELF metadata-based weird machines
- ELF, bootloaders, dynamic and static analysis
- Defensive research (more or less) with a dash of reverse engineering
- @ bxsays on Twitter

Photo circa March 2019

# Meet Das U-Boot bootloader

```
U-Boot 2010.12-xes_r3 (Aug 25 2011 - 11:04:04)

CPU0:  P2020E, Version: 1.0, (0x80ea0010)
Core:  E500, Version: 4.0, (0x80211040)
Clock Configuration:
       CPU0:1066.667 MHz, CPU1:1066.667 MHz,
       CCB:533.333 MHz,
       DDR:400  MHz (800 MT/s data rate) (Asynchronous), LBC:133.333 MHz
L1:    D-cache 32 kB enabled
       I-cache 32 kB enabled
Board: X-ES XPedite5501 PMC/XMC SBC
       Rev SA, Serial# 36093001, Cfg 90015130-1
I2C:   ready
DRAM:  2 GiB (DDR3, 64-bit, CL=6, ECC on)
FLASH: Executed from FLASH1
POST memory PASSED
FLASH: 256 MiB
L2:    512 KB enabled
NAND:  4096 MiB
PCIE1: connected as Root Complex (no link)
PCIE1: Bus 00 - 00
PCIE2: disabled
PCIE3: disabled
In:    serial
Out:   serial
Err:   serial
DTT:   37 C local / 59 C remote (adt7461@4c)
DTT:   37 C local (lm75@48)
Net:   eTSEC1 connected to Broadcom BCM5482S
eTSEC2 connected to Broadcom BCM5482S
eTSEC1, eTSEC2
POST i2c PASSED
Hit any key to stop autoboot:  0
=>
```

# Meet Das U-Boot bootloader

```
[user@boot-dev ~]$ cloc u-boot/
   13518 text files.
   12700 unique files.
    4701 files ignored.
github.com/AlDanial/cloc v 1.76  T=4.02 s (2196.7 files/s, 504571.1 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                             3958         177722         230606         911861
C/C++ Header                  3540          64684         108111         429854
Assembly                       236           5927          10632          24037
Python                         119           4380           9180          12486
Perl                             6           1660           1346           9850
make                           911           2263           4664           8500
Bourne Shell                    32            427            626           2164
C++                              1            233             58           1588
yacc                             2            169             75           1076
Glade                            1             58              0            603
lex                              2             98             41            539
NAnt script                      1             91              0            367
YAML                             1             13             25            347
Bourne Again Shell               3             75             66            316
Markdown                         1             80              0            283
DOS Batch                        3             20              0            176
CSS                              2             24             10             90
Kermit                           3              4             20             83
Tcl/Tk                           1              5              5             28
sed                              2              1             27             24
INI                              2              3              0             14
XSLT                             1              0              1              9
-------------------------------------------------------------------------------
SUM:                          8828         257937         365493        1404295
```

# Meet Das U-Boot bootloader

```
[user@boot-dev ~]$ cloc u-boot/
   13518 text files.
   12700 unique files.
    4701 files ignored.
github.com/AlDanial/cloc v 1.76  T=4.02 s (2196.7 files/s, 504571.1 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                             3958         177722         230606         911861
C/C++ Header                  3540          64684         108111         429854
Assembly                       236           5927          10632          24037
Python                         119           4380           9180          12486
Perl                             6           1660           1346           9850
make                                                                       8500
Markdown                         1             80              0            283
DOS Batch                        3             20              0            176
CSS                              2             24             10             90
Kermit                           3              4             20             83
Tcl/Tk                           1              5              5             28
sed                              2              1             27             24
INI                              2              3              0             14
XSLT                             1              0              1              9
-------------------------------------------------------------------------------
SUM:                          8828         257937         365493        1404295
```

**"Only" 111 MB of source code**

```
[user@boot-dev ~]$ make -C u-boot distclean
make: Entering directory '/home/user/u-boot'
make: Leaving directory '/home/user/u-boot'
[user@boot-dev ~]$ du -sh u-boot/
111M  u-boot/
```

# Meet Das U-Boot bootloader

```
[user@boot-dev ~]$ cloc u-boot/
   13518 text files.
   12700 unique files.
    4701 files ignored.
github.com/AlDanial/cloc v 1.76  T=4.02 s (2196.7 files/s, 504571.1 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                             3958         177722         230606         911861
C/C++ Header                  3540          64684         108111         429854
Assembly                       236           5927          10632          24037
Python                         119           4380           9180          12486
Perl                             6           1660           1346           9850
make                                                                      8500
Markdown                         1             80              0            283
DOS Batch                        3             20              0            176
CSS                              2             24             10             90
Kermit                           3              4             20             83
Tcl/Tk                           1              5              5             28
sed                              2              1             27             24
INI                              2              3              0             14
XSLT                             1              0              1              9
-------------------------------------------------------------------------------
SUM:                          8828         257937         365493        1404295
```

**"Only" 111 MB of source code**

```
[user@boot-dev ~]$ make -C u-boot distclean
make: Entering directory '/home/user/u-boot'
make: Leaving directory '/home/user/u-boot'
[user@boot-dev ~]$ du -sh u-boot/
111M  u-boot/
```

# Quick aside: **What is a loader**?

## Magic that transforms a binary image into a running application

- **Loader:** Software that transduces binary images into memory for execution
- **Binary image:** Static representation/encapsulation of binary (machine) code
  - e.g. An ELF or PE file

### Some useful terminology

- **Address space:** general term referring to addressable memory
- **Memory map:** address space model that semantically labels memory regions

# Who loads the loader?

A loader, of course

# Who loads the loader?

## A loader, of course

(It's turtles all the way down)

# Who loads the loader?

## A loader, of course

(It's turtles all the way down)

**Bootloader**: a loader

that executes before the OS (or primary application) is executed

# Who loads the loader?

## A loader, of course

(It's turtles all the way down)

**Bootloader**: a loader

that executes before the OS (or primary application) is executed

# The existential question.

## Overall research goals

1. **Identify weaknesses** underlying (boot)loader security
2. **Develop (boot)loader hardening techniques** that:
   - are **realistic**
   - lend themselves to **formal reasoning**
   - can be **retroactively applied** to **existing** loaders
3. Demonstrate technique **feasibility**

# The existential question.

## Overall research goals

1. **Identify weaknesses** underlying (boot)loader security
2. **Develop (boot)loader hardening techniques** that:
   - are **realistic**
   - lend themselves to **formal reasoning**
   - can be **retroactively applied** to **existing** loaders
3. Demonstrate technique **feasibility**

Hence the bootloader reversing

# The existential question.

## * *Flashback to 2012* *

### The Turing-complete ELF-metadata weird machine

a brainfuck to ELF-metadata compiler @ https://github.com/bx/elf-bf-tools

(ELF is *NIX's file format for executables, libraries, etc.)

AS SEEN ON YouTube DEF CON, 29C3, USENIX WOOT, PoC || GTFO

([1], [2], [3], [4])

Hence the bootloader reversing

# The existential question.

## Overall research goals

1. **Identify weaknesses** underlying (boot)loader security
2. **Develop (boot)loader hardening techniques** that:
   - are **realistic**
   - lend themselves to **formal reasoning**
   - can be **retroactively applied** to **existing** loaders
3. Demonstrate technique **feasibility**

Hence the bootloader reversing

# The existential question.

## Overall research goals

1. **Identify weaknesses** underlying (boot)loader security
2. **Develop (boot)loader hardening techniques** that:
   - are **realistic**
   - lend themselves to **formal reasoning**
   - can be **retroactively applied** to **existing** loaders
3. Demonstrate technique **feasibility**

Hence the bootloader reversing

# This talk for those in a hurry

- ~~Introduce goals and case study~~
- Pontificate about bootloaders in general
- Debugging U-Boot (as according to U-Boot)
- My instrumentation toolsuite
- Identifying code/data relationships
- Build a flexible memory access policy based on these relationships
- A test drive of a simple example (if time allows)

# What to expect from a bootloader

- Discovery and initialization of resources/hardware
- Address space preparation
- Binary image loading and preparation
- Self-relocation (occasionally)

## In general, (boot)loaders:

- Allocate **non-overlapping** addresses
- Manage address **alignment requirements**
- **Prepare memory map** for target
  - **Load** target image into memory
  - **Patch** (link) loaded images
- **Extract** and **enforce** requirements and restrictions imposed by both resources and target

# Resource discovery for humans

# Resource discovery for machines

# How bootloaders discover resources

## Hard-coded "discovery"

```
/*************************************************
 * get_sysboot_value(void) - return SYS_BOOT[4:0]
 *************************************************/
u32 get_sysboot_value(void)
{
        int mode;
        mode = __raw_readl(CONTROL_STATUS) & (SYSBOOT_MASK);
        return mode;
```

## Semi-hard-coded/table-based discovery

```
/*************************************************************
 * Routine: get_mem_type(void) - returns the kind of memory connected
 * to GPMC that we are trying to boot form. Uses SYS BOOT settings.
 *************************************************************/
u32 get_mem_type(void)
{

        if (beagle_revision() == REVISION_XM)
                return GPMC_NONE;

        u32   mem_type = get_sysboot_value();
        switch (mem_type) {
        case 0:
```

## Protocol/standards-based discovery

```
argument = 0x0000 << 16;
err = mmc_send_cmd(MMC_CMD55, argument, resp);
if (err == 1) {
        mmc_card_cur->card_type = SD_CARD;
        ocr_value |= hcs_val;
        ret_cmd41 = MMC_ACMD41;
} else {
        mmc_card_cur->card_type = MMC_CARD;
```

```
#ifdef CONFIG_MMC
        /* first try mmc */
        if (mmc_init(1)) {
                size = file_fat_read("u-boot.bin", buf, 0);
                if (size > 0) {
#ifdef CFG_PRINTF
                        printf("Loading u-boot.bin from mmc\n");
#endif
                        buf += size;
                }
```

# How a bootloader uses resources

## Initialization

```
/* Turn on all 3 I2C clocks */
sr32(CM_FCLKEN1_CORE, 15, 3, 0x7);
sr32(CM_ICLKEN1_CORE, 15, 3, 0x7);      /* I2C1,2,3 = on */
```

```
switch(beagle_revision()) {
case REVISION_C4:
        if (identify_xm_ddr() == NUMONYX_MCP) {
                __raw_writel(0x4, SDRC_CS_CFG); /* 512MB/bank */
                __raw_writel(SDP_SDRC_MDCFG_0_DDR_NUMONYX_XM, SDRC_MCFG_0);
                __raw_writel(SDP_SDRC_MDCFG_0_DDR_NUMONYX_XM, SDRC_MCFG_1);
                __raw_writel(NUMONYX_V_OCTIMO_165, SDRC_OCTIM_CTRLO_0);
```

## Use

```
/* set slave address */
outw (devaddr, I2C_SA);
/* stop bit needed here */
outw (I2C_CON_EN | I2C_CON_MST | I2C_CON_STT | I2C_CON_TRX |
      I2C_CON_STP, I2C_CON);

/* wait until state change */
status = wait_for_pin ();

if (status & I2C_STAT_XRDY) {
ined(CONFIG_OMAP243X) || defined(CONFIG_OMAP34XX)
        /* send out 1 byte */
        outb(regoffset, I2C_DATA);
        outw(I2C_STAT_XRDY, I2C_STAT);
        status = wait_for_pin();
```

# How a bootloader loads binary images

## Locate image

```
#ifdef CONFIG_MMC
        /* first try mmc */
        if (mmc_init(1)) {
                size = file_fat_read("u-boot.bin", buf, 0);
                if (size > 0) {
#ifdef CFG_PRINTF
                        printf("Loading u-boot.bin from mmc\n");
#endif
                        buf += size;
                }
```

## Load and parse header

# A bit about self-relocation

Bootloaders self-relocate, this may sound crazy
but when RAM is small, you can't be lazy

```
ENTRY(relocate_code)
  ldr r1, =__image_copy_start /* r1 <- &__image_copy_start */
  subs    r4, r0, r1      /* r4 <- relocation offset */
  beq relocate_done       /* skip relocation */
  ldr r2, =__image_copy_end /* r2 <- &__image_copy_end */
copy_loop:
  ldmia   r1!, {r10-r11}  /* copy from source orgaddress [r1]    */
  stmia   r0!, {r10-r11}  /* copy to   target address [r0]    */
  cmp r1, r2      /* until source end address [r2]    */
  blo copy_loop
   /* fix .rel.dyn relocations */
  ldr r2, =__rel_dyn_start /* r2 <- SRC &__rel_dyn_start */
  ldr r3, =__rel_dyn_end /* r3 <- SRC &__rel_dyn_end */
fixloop:
  ldmia   r2!, {r0-r1} /* (r0,r1) <- (SRC location,fixup) */
  and r1, r1, #0xff
  cmp r1, #23  /* relative fixup? */
  bne fixnext
  /* relative fix: increase location by offset */
  add r0, r0, r4
  ldr r1, [r0]
  add r1, r1, r4
  str r1, [r0]
fixnext:
  cmp r2, r3
  blo fixloop
relocate_done:
  bx  lr
ENDPROC(relocate_code)
```

**Code from u-boot arch/arm/lib/relocate.S**

# Questions to ask a bootloader

- On what **architecture(s)** does it run?
  - ARM? x86?
- **Where** can it be **stored**?
  - First disk sector? Flash memory?
- **Where** in memory can it be **loaded**?
- **How** is its image **packaged**?
  - MBR? ELF?
- **What** runtime **arguments** are expected. **How** are they located?
- From **what devices** can it load its **next stage**?
  - Disk? Network?
- **What formats** can it handle as it locates the next stage?
  - FAT? GPT?
- How can the **next stage** be **packaged**?
  - ELF? Compressed? Signed?
- **What arguments** does it pass to the next stage and **how**?

# Bootloader observations & hardening (1/2)

## During bootloading

- Memory map/address space may change multiple times
  - Memory initialized, stack moved, subsequent stage loaded
- Execution can be divided into **substages** by memory usage

## During a substage, the bootloader

- **Prepares/loads/patches/links** a region of memory used by a future substage

- or -

- Performs internal **bookkeeping**

# Bootloader observations & hardening (2/2)

Substages can be classified by intended use

i.e., Loading, patching, bookkeeping

Memory can be classified by intended use

i.e., Code (text), registers, bookkeeping, future substage image

**Now let's retrofit U-Boot with hardening measures!**

# Introducing RBWAC

## RBWAC

address Region-Based Write Access Control

**(Pronounced: ARRRRR….whack!)**

## Components of an RBWAC policy

- **Substages**: distinct phases of bootloader process
- **Memory regions**: regions of memory each with a particular use during a particular substage

# Introducing RBWAC

RBWAC

address ess Control

Compone VAC policy

- **Substages**: distinct
- **Memory regions**: r rticular use during a particular substage

to next stage (◎)

_st

bookke

# Phase one

## Let's begin exploring U-Boot

# Bootloader instrumentation: the tricky bits

1. Self-relocation, encryption, decompression
2. Magic/hard-coded numbers
3. Inaccurate documentation
4. Operations/peripherals may silently fail
    - e.g., nothing happens after writing a read-only register
5. Address space may change
6. Operating mode of CPU may change (e.g., privileged, unprivileged)

# Bootloader instrumentation in action

## A demo (?)

# How to debug a self-relocating bootloader

## As according to U-Boot's doc/README.arm-relocation

### start debugger

```
[hs@pollux u-boot]$ arm-linux-gdb u-boot
```

# How to debug a self-relocating bootloader

## As according to U-Boot's doc/README.arm-relocation

### start debugger

```
[hs@pollux u-boot]$ arm-linux-gdb u-boot
```

### connect to target

```
(gdb) target remote localhost:4444
```

# How to debug a self-relocating bootloader

## As according to U-Boot's doc/README.arm-relocation

### start debugger

```
[hs@pollux u-boot]$ arm-linux-gdb u-boot
```

### connect to target

```
(gdb) target remote localhost:4444
```

### execute until relocation complete

```
(gdb) break _relocation_done
(gdb) c
```

# How to debug a self-relocating bootloader

## As according to U-Boot's doc/README.arm-relocation

### start debugger

```
[hs@pollux u-boot]$ arm-linux-gdb u-boot
```

### connect to target

```
(gdb) target remote localhost:4444
```

### execute until relocation complete

```
(gdb) break _relocation_done
(gdb) c
```

### discard symbols

```
(gdb) symbol-file
Discard symbol table from `/home/hs/celf/u-boot/u-boot'? (y or n) y
No symbol file now.
```

# How to debug a self-relocating bootloader

## As according to U-Boot's doc/README.arm-relocation

### start debugger

```
[hs@pollux u-boot]$ arm-linux-gdb u-boot
```

### connect to target

```
(gdb) target remote localhost:4444
```

### execute until relocation complete

```
(gdb) break _relocation_done
(gdb) c
```

### discard symbols

```
(gdb) symbol-file
Discard symbol table from `/home/hs/celf/u-boot/u-boot'? (y or n) y
No symbol file now.
```

### load new symbol table at relocated address

```
gdb) add-symbol-file u-boot 0x8ff08000
add symbol table from file "u-boot" at
  .text_addr = 0x8ff08000
(y or n) y
Reading symbols from /home/hs/celf/u-boot/u-boot...done.
```

# How to debug a self-relocating bootloader

## As according to U-Boot's doc/README.arm-relocation

### start debugger

```
[hs@pollux u-boot]$ arm-linux-gdb u-boot
```

### connect to target

```
(gdb) target remote loc
```

### execu ete

```
(gdb) break _relocatior
(gdb) c
```

```
(gdb) symbol-file
Discard symbol table fr r n) y
No symbol file now.
```

### load new s address

```
gdb) add-symbol-file u-
add symbol table from file "u-boot" at
  .text_addr = 0x8ff08000
(y or n) y
Reading symbols from /home/hs/celf/u-boot/u-boot...done.
```

# "Demystifying" magic numbers

## As suggested by the U-Boot developers

```
Program received signal SIGSTOP, Stopped (signal).
0x8ff17f18 in serial_getc () at serial_mxc.c:192
192     while (__REG(UART_PHYS + UTS) & UTS_RXEMPTY);
(gdb) add-symbol-file u-boot 0x8ff08000
```

# "Demystifying" magic numbers

## As suggested by the U-Boot developers

```
Program received signal SIGSTOP, Stopped (signal).
0x8ff17f18 in serial_getc () at serial_mxc.c:192
192     while (__REG(UART_PHYS + UTS) & UTS_RXEMPTY);
(gdb) add-symbol-file u-boot 0x8ff08000
```

# "Demystifying" magic numbers

## As suggested by the U-Boot developers

```
Program received signal SIGSTOP, Stopped (signal).
0x8ff17f18 in serial_getc () at serial_mxc.c:192
192     while (__REG(UART_PHYS + UTS) & UTS_RXEMPTY);
(gdb) add-symbol-file u-boot 0x8ff08000
```

"Get this address from u-boot bdinfo command or get it from gd->relocaddr in gdb. Interrupt execution by any means and re-load the symbols at the location specified by gd->relocaddr -- this is only valid after board_init_f."

# "Demystifying" magic numbers

## As suggested by the U-Boot developers

```
Program received signal SIGSTOP, Stopped (signal).
0x8ff17f18 in serial_getc () at serial_mxc.c:192
192     while (__REG(UART_PHYS + UTS) & UTS_RXEMPTY);
(gdb) add-symbol-file u-boot 0x8ff08000
```

"Get this address from u-boot bdinfo command or get it from gd->relocaddr in gdb. Interrupt execution by any means and re-load the symbols at the location specified by gd->relocaddr -- this is only valid after board_init_f."

```
=> bdinfo
arch_number = XXXXXXXXXX
boot_params = XXXXXXXXXX
DRAM bank   = XXXXXXXXXX
-> start    = XXXXXXXXXX
-> size     = XXXXXXXXXX
ethaddr     = XXXXXXXXXX
ip_addr     = XXXXXXXXXX
baudrate    = XXXXXXXXXX
TLB addr    = XXXXXXXXXX
relocaddr   = 0x8ff08000
reloc off   = XXXXXXXXXX
irq_sp     = XXXXXXXXXX
sp start    = XXXXXXXXXX
FB base     = XXXXXXXXXX
```

# "Demystifying" magic numbers

## As suggested by the U-Boot developers

```
Program received signal SIGSTOP, Stopped (signal).
0x8ff17f18 in serial_getc () at serial_mxc.c:192
192     while (__REG(UART_PHYS + UTS) & UTS_RXEMPTY);
(gdb) add-symbol-file u-boot 0x8ff08000
```

"Get this address from u-boot bdinfo command or get it from gd->relocaddr in gdb. Interrupt execution by any means and re-load the symbols at the location specified by gd->relocaddr -- this is only valid after board_init_f."

```
=> bdinfo
arch_number = XXXXXXXXXX
boot_params = XXXXXXXXXX
DRAM bank   = XXXXXXXXXX
-> start    = XXXXXXXXXX
-> size     = XXXXXXXXXX
ethaddr     = XXXXXXXXXX
ip_addr     = XXXXXXXXXX
baudrate    = XXXXXXXXXX
TLB addr    = XXXXXXXXXX
relocaddr   = 0x8ff08000
reloc_off   = XXXXXXXXXX
irq_sp     = XXXXXXXXXX
sp start    = XXXXXXXXXX
FB base     = XXXXXXXXXX
```

# Hardware-related magic numbers

## Remember

```
/*********************************************
 * get_sysboot_value(void) - return SYS_BOOT[4:0]
 *********************************************/
u32 get_sysboot_value(void)
{
        int mode;
        mode = __raw_readl(CONTROL_STATUS) & (SYSBOOT_MASK);
        return mode;
```

?

## Go RTFM

*Public Version*

**AM/DM37x Multimedia Device**
**Silicon Revision 1.x**

**Version R**

**Technical Reference Manual**

Texas Instruments

*PRCM Register Manual* www.ti.com

**Table 3-225. CAM_CM Register Summary**

| Register Name | Type | Register Width (Bits) | Address Offset | Physical Address | Reset Type |
|---|---|---|---|---|---|
| CM_FCLKEN_CAM | RW | 32 | 0x0000 0000 | 0x4800 4F00 | W |
| CM_ICLKEN_CAM | RW | 32 | 0x0000 0010 | 0x4800 4F10 | W |
| CM_IDLEST_CAM | R | 32 | 0x0000 0020 | 0x4800 4F20 | C |
| CM_AUTOIDLE_CAM | RW | 32 | 0x0000 0030 | 0x4800 4F30 | W |
| CM_CLKSEL_CAM | RW | 32 | 0x0000 0040 | 0x4800 4F40 | W |
| CM_SLEEPDEP_CAM | RW | 32 | 0x0000 0044 | 0x4800 4F44 | W |
| CM_CLKSTCTRL_CAM | RW | 32 | 0x0000 0048 | 0x4800 4F48 | W |
| CM_CLKSTST_CAM | R | 32 | 0x0000 004C | 0x4800 4F4C | C |

CC BY-NC 2.0 alexwhite // flickr

# Can we do better?

# The Fiddle toolsuite

Featuring:

- Static and dynamic analysis
- Instrumentation (via GDB)
  - Mediates **all** memory writes
- Language to express (and enforce) memory write polices
- Supports 32-bit ARM (partly arm64 & x86 (32/64))
- Can handle self-relocation
- Not just for bootloaders

Source code at https://typedregions.com

# Phase two

# Relocation reconnaissance

Featuring the U-Boot SPL for BeagleBoard-xM

Tools at https://typedregions.com

# Identifying relocation phases

## Hypothesis:

The relocation process makes use of memcpy-like operations

(Copying a sequence of adjacent bytes in a tight loop)

I call these **block write operations**

**Block write operation**

($ip, offset in image, destination, size, call stack)

# Identifying relocation phases

## Hypothesis:

The relocation process makes use of memcpy-like operations

(Copying a sequence of adjacent bytes in a tight loop)

I call these **block write operations**

**Block write operation**

($ip, offset in image, destination, size, call stack)

## Corollary:

We should be able to identify relocation phases from large block write operations

# Block writes in U-Boot

```
[262144x] memset        wrote 1048576 B @ 0x80208000 str
[49233x]  clbss_l       wrote 196932 B  @ 0x80000000 strcc     <-- Zero BSS
[128x]    mmc_read_data wrote 512 B     @ 0x4020f2c0 str.w
[128x]    mmc_read_data wrote 512 B     @ 0x4020f2c0 str.w
[128x]    mmc_read_data wrote 512 B     @ 0x80104bc0 str.w     <-- Read target image to memory
[128x]    mmc_read_data wrote 512 B     @ 0x80104dc0 str.w
[83x]     memset        wrote 332 B     @ 0x80208038 str
[9x]      next2         wrote 288 B     @ 0x4020f840 stmia     <-- Relocate "go_to_speed" function
[54x]     memset        wrote 216 B     @ 0x4020fe10 str
[30x]     memcpy        wrote 120 B     @ 0x800200c0 str
[16x]     memcpy        wrote 64 B      @ 0x4020f118 str       <-- Relocate bookkeeping data
[1x]      omap_smc1     wrote 40 B      @ 0x40200234 push      <-- Function call/stack push
```

| | |
|---|---|
| # write operations | ~400,000 |
| # block writes | 10,000 |

# U-Boot's static call graph (for reference)

Generated using IDA Pro (and then simplified by hand)

# Calltrace of successful U-Boot execution

## Example output from U-Boot SPL execution

```
> save_boot_params {arch/ahrm/cpu/armv7/omap-common/lowlevel_init.S::22}
```

**This is a javascript-enabled demo**

First 4800* SPL repeated block writes & call graph
*of about 9000 total block writes
__start
save_boot_params
lowlevel_init
(relocate)
cpy_clk_code
per_clocks_enable
s_init
__main
set_muxconf_regs
board_init_f_mem
arch_setup_gd
(clear global data)
clear_bss
(printf)
spl_init
board_init_r
spl_mmc_load_image
mmc_init
initialize mmc driver
get_partition_info
read filesystem partition table
spl_load_image_fat
file_fat_read
read target image header
spl_parse_image_header
file_fat_read
read full target image
jump_to_image_no_args
block write index
0
200
400
600
800
1,000
1,200
1,400
1,600
1,800
2,000
2,200
2,400
2,600
2,800
3,000
3,200
3,400
3,600
3,800
4,000
4,200
4,400
4,600
4,800
1
8
64
512
4K
32K
256K
2M
consecutive bytes written

# Understanding U-Boot SPL execution
## as a sequence of phases

# Devising code and data relationships

# Other magic numbers

```
switch(beagle_revision()) {
 case REVISION_C4:
  if (identify_xm_ddr() == NUMONYX_MCP) {
    /* 512MB/bank */
   __raw_writel(0x4, SDRC_CS_CFG);
   __raw_writel(SDP_SDRC_MDCFG_0_DDR_NUMONYX_XM,
      SDRC_MCFG_0);
```

```
argument = 0x0000 << 16;
err = mmc_send_cmd(MMC_CMD55, argument, resp);
if (err == 1) {
 mmc_card_cur->card_type = SD_CARD;
} else {
 mmc_card_cur->card_type = MMC_CARD;
```

# Other magic numbers

```
switch(beagle_revision()) {
 case REVISION_C4:
  if (identify_xm_ddr() == NUMONYX_MCP) {
    /* 512MB/bank */
   __raw_writel(0x4, SDRC_CS_CFG);
   __raw_writel(SDP_SDRC_MDCFG_0_DDR_NUMONYX_XM,
      SDRC_MCFG_0);
```

```
argument = 0x0000 << 16;
err = mmc_send_cmd(MMC_CMD55, argument, resp);
if (err == 1) {
 mmc_card_cur->card_type = SD_CARD;
} else {
 mmc_card_cur->card_type = MMC_CARD;
```

## So I built a PDF scraper

```
for o in page:
    if cls.is_not_in_table_bounds(tablestart, tableend, o):
        continue
    if isinstance(o, layout.LTRect) and (abs(o.bbox[1] - o.bbox[3]) < 1.5):  # is a line
        # if left edge of line is not near left edge of name col_info
        if abs(namecol.bbox[0] - (o.bbox[0] + offset)) > 5:
            continue
        odiff = cls.vertical_offset(namecol, o)
        if (odiff > 0) and (closest_rect is None):
            closest_rect = o
```

# Other magic numbers

```
switch(beagle_revision()) {
 case REVISION_C4:
  if (identify_xm_ddr() == NUMONYX_MCP) {
    /* 512MB/bank */
   __raw_writel(0x4, SDRC_CS_CFG);
   __raw_writel(SDP_SDRC_MDCFG_0_DDR_NUMONYX_XM,
      SDRC_MCFG_0);
```

```
argument = 0x0000 << 16;
err = mmc_send_cmd(MMC_CMD55, argument, resp);
if (err == 1) {
 mmc_card_cur->card_type = SD_CARD;
} else {
 mmc_card_cur->card_type = MMC_CARD;
```

## So I built a PDF scraper

```
for o in page:
    if cls.is_not_in_table_bounds(tablestart, tableend, o):
        continue
    if isinstance(o, layout.LTRect) and (abs(o.bbox[1] - o.bbox[3]) < 1.5):  # is a line
        # if left edge of line is not near left edge of name col_info
        if abs(namecol.bbox[0] - (o.bbox[0] + offset)) > 5:
            continue
        odiff = cls.vertical_offset(namecol, o)
        if (odiff > 0) and (closest_rect is None):
            closest_rect = o
```

## that transforms

**Table 10-28. GPMC Registers Mapping Summary**

| Register Name | Type | Register Width (Bits) | Address Offset | Physical Address |
|---|---|---|---|---|
| GPMC_REVISION | R | 32 | 0x0000 0000 | 0x6E00 0000 |
| GPMC_SYSCONFIG | RW | 32 | 0x0000 0010 | 0x6E00 0010 |
| GPMC_SYSSTATUS | R | 32 | 0x0000 0014 | 0x6E00 0014 |
| GPMC_IRQSTATUS | RW | 32 | 0x0000 0018 | 0x6E00 0018 |

# Other magic numbers

```c
switch(beagle_revision()) {
 case REVISION_C4:
  if (identify_xm_ddr() == NUMONYX_MCP) {
    /* 512MB/bank */
   __raw_writel(0x4, SDRC_CS_CFG);
   __raw_writel(SDP_SDRC_MDCFG_0_DDR_NUMONYX_XM,
      SDRC_MCFG_0);
```

```c
argument = 0x0000 << 16;
err = mmc_send_cmd(MMC_CMD55, argument, resp);
if (err == 1) {
 mmc_card_cur->card_type = SD_CARD;
} else {
 mmc_card_cur->card_type = MMC_CARD;
```

## So I built a PDF scraper

```python
for o in page:
    if cls.is_not_in_table_bounds(tablestart, tableend, o):
        continue
    if isinstance(o, layout.LTRect) and (abs(o.bbox[1] - o.bbox[3]) < 1.5):  # is a line
        # if left edge of line is not near left edge of name col_info
        if abs(namecol.bbox[0] - (o.bbox[0] + offset)) > 5:
            continue
        odiff = cls.vertical_offset(namecol, o)
        if (odiff > 0) and (closest_rect is None):
            closest_rect = o
```

## that transforms

**Table 10-28. GPMC Registers Mapping Summary**

| Register Name | Type | Register Width (Bits) | Address Offset | Physical Address |
|---|---|---|---|---|
| GPMC_REVISION | R | 32 | 0x0000 0000 | 0x6E00 0000 |
| GPMC_SYSCONFIG | RW | 32 | 0x0000 0010 | 0x6E00 0010 |
| GPMC_SYSSTATUS | R | 32 | 0x0000 0014 | 0x6E00 0014 |
| GPMC_IRQSTATUS | RW | 32 | 0x0000 0018 | 0x6E00 0018 |

## into

```
CM_FCLKEN_IVA2,RW,W,32,0x00000000,0x48004000,Table 3-93. IVA2_CM Register Summary
CM_CLKEN_PLL_IVA2,RW,W,32,0x00000004,0x48004004,Table 3-93. IVA2_CM Register Summary
CM_IDLEST_IVA2,R,C,32,0x00000020,0x48004020,Table 3-93. IVA2_CM Register Summary
CM_IDLEST_PLL_IVA2,R,C,32,0x00000024,0x48004024,Table 3-93. IVA2_CM Register Summary
CM_AUTOIDLE_PLL_IVA2,RW,W,32,0x00000034,0x48004034,Table 3-93. IVA2_CM Register Summary
```

# A simpler example

# Hello, world!

```c
#include <stdio.h>
#include <string.h>

#define SIZE 512
char memory[SIZE];

void do_nothing() {}

void say_hello() {
  printf("Hello, world\n");
}

void modify_memory() {
  for (int i = 0; i <= SIZE; i++) {
    memory[i] = 'A';
  }
}

int main(int argc, char *argv[]) {
  say_hello();
  modify_memory();
  do_nothing();
  return 0;
}
```

(Built with -static,-no-pie)

# Calltrace of Hello, world!

```
> _libcstartmain {arm-linux-gnueabihf-glibc/src/glibc-2.27/csu/libc-start.c::137}
```

**This is a javascript-enabled demo**

# Checking for unexpected writes

1. Configure instrumentation suite to work with sample
2. Run sample through instrumentation suite's **static analysis**
3. Construct policy to target "substages" and memory regions of interest
4. Import policy
5. Execute **dynamic analysis**
6. Use **post-analysis** to highlight policy violations

# Checking for unexpected writes

1. Configure instrumentation suite to work with sample
2. Run sample through instrumentation suite's **static analysis**
3. Construct policy to target "substages" and memory regions of interest
4. Import policy
5. Execute **dynamic analysis**
6. Use **post-analysis** to highlight policy violations

## Region definitions

```
regions:
 ALL:
  type: "global"
  addresses: [0, 0xFFFFFFFF]
  subregions:
   buffer:
     type: "global"
     addresses: [0x8bb98, 0x8bd98]
   stack:
     type: "stack"
     addresses: [0xfffe0000, 0xffff0000]
   ro:
     type: "global"
     addresses: [[0x0, 0x8bb898],
       [0x8bd98,0xfffe0000],
       [0xffff0000,0xffffffff]]

stagename: "_single"
```

## Substage/policy definitions

```
_start:
  substage_type: "bookkeeping"
  new_regions: ["ALL.buffer", "ALL.stack",
"ALL.ro"]
  reclassified_regions:
    ALL.ro: "global"
    ALL.stack: "stack"
    ALL.buffer: "global"

main:
  substage_type: "bookkeeping"

modify_memory:
  substage_type: "bookkeeping"
  reclassified_regions:
    ALL.ro: "readonly"

do_nothing:
  substage_type: "bookkeeping"
  reclassified_regions:
    ALL.ro: "global"
```

# Phase three

## An attempt at a demo

# Final thoughts

## How can we trust...

- a system without trusting its bootloader?
- a bootloader when it's so hard to understand?

Fiddle: instrumentation for bootloaders and beyond

RBWAC: a region-based write access control

& we can incrementally retrofit U-Boot with RBWAC

# Thank you

More details and tools at: https://typedregions.com
Blog post on bootloaders at: https://bit.ly/2zbentY
Many thanks to Sergey Bratus, my PhD advisor
**bx@narfindustries.com** || **bxsays** on twitter

# Thank you

More details and tools at: https://typedregions.com
Blog post on bootloaders at: https://bit.ly/2zbentY
Many thanks to Sergey Bratus, my PhD advisor
**bx@narfindustries.com** || **bxsays** on twitter

# References

See also typedregions.com


[1]. R. .. Shapiro, “The care and feeding of weird machines found in executable metadata,” in *Chaos communication congress*, 2012.

[2]. R. Shapiro, S. Bratus, and S. W. Smith, “Weird machines in ELF: A spotlight on the underappreciated metadata,” in *Workshop on offensive technologies*, Washington, D.C., 2013.

[3]. R. .. Shapiro, “Returning from ELF to libc,” in *POC or GTFO*, vol. 0x0, 2013.

[4]. R. .. Shapiro, “Calling putchar() from an ELF weird machine,” in *POC or GTFO*, vol. 0x02, 2013.

[5]. J. Bangert, S. Bratus, R. Shapiro, and S. W. Smith, “The page-fault weird machine: Lessons in instruction-less computation,” in *Workshop on offensive technologies*, Washington, D.C., 2013.

[6]. J. Bangert, S. Bratus, R. Shapiro, M. E. Locasto, J. Reeves, S. W. Smith, and A. Shubina, “ELFbac: Using the loader format for intent-level semantics and fine-grained protection,” Dartmouth College, Computer Science, Hanover, NH, TR2013-727, May 2013.

[7]. S. Bratus, M. E. Locasto, and M. L. Patterson, “Exploit programming: From buffer overflows to "weird machines” and theory of computation,” in *USENIX; login*, 2011, pp. 13–21.

[8]. S. Bratus and J. Bangert, “ELFs are dorky, elves are cool,” in *POC or GTFO*, vol. 0x0, 2013.

[9]. J. Oakley and S. Bratus, “Exploiting the hard-working DWARF: Trojan and exploit techniques with no native executable code,” in *Workshop on offensive technologies*, 2011, p. 11.