Introduction

Abstract

Broadcom wireless cards for mobiles devices, specifically the Broadcom line BCM4325/29/30/34 are the most common wireless cards found in most popular smartphones & tables (iPhone/iPad, Samsung, Nokia, Motorola and HTC among others). Even with such an installed base and being a key client component in any wireless network -at least any Wi-Fi network where mobile devices participate- not much has been said about such cards. Previous research, in this area includes approaches to modify the firmware to enable monitor mode and raw 802.11 traffic injection in popular smartphones [1, 2]. In those occasions most of the work was performed by static firmware reverse engineering. In this paper, we will describe how to get a more dynamic approach to analyze the behavior of the firmware execution on the network card CPU.

Background

Network card firmware analysis and attacks are not new. Several previous works were published in this area: Debugger and Rootkits were developed [3] cards were modified to pivot attacks to other peripherals through DMA or even abuse the PCI BUS to create a P2P hardware attack [4] (i.e. attack a video card by communicating directly from the network card), specific vulnerabilities were discovered that provided remote attack vectors [5] among others. In this paper, however, we will focus on 802.11 Wi-Fi network cards for mobile devices and present a tool that will allow us to perform dynamic analysis of the card firmware. Specifically, a limited tracer will be created. It is our hope that, by making such tools available, further research will be conducted.

Objectives

Our main objective is to provide a mechanism to inspect execution states of the network card at different code points in a way that is, as much as possible, independent of a specific model, version and mobile device operating system. It is important to achieve this portability, because the daunting speed of device development makes devices obsolete rather quickly. Additionally, aggressive competition among mobile device vendors creates the need to constantly introduce the latest 802.11 features thus new network card models are constantly being introduced. Furthermore market share of mobile devices can shift in short time frames, tying our approach to a specific vendor might render it irrelevant in the near future.

Architecture

These types of network cards communicate with the host (mobile device) over a SDIO bus. The card itself consist of a main CPU, generally ARM Cortext M3 or R4, a limited amount of volatile RAM memory to support the firmware execution and persistent ROM storage. Additionally, the low level functionality is grouped into several functional modules called *cores.* A number of cores provide the low level functionality such as: PHY/MAC layers, chip specific setup, *d11 core* that implements certain aspects of the 802.11 protocol. Cores communicate over memory mapped registers and DMA with the network card firmware. It is important to note that this DMA is internal to the network card itself and as such the NiC device is not provided access to the host device memory. This layout is illustrated by the following figure:

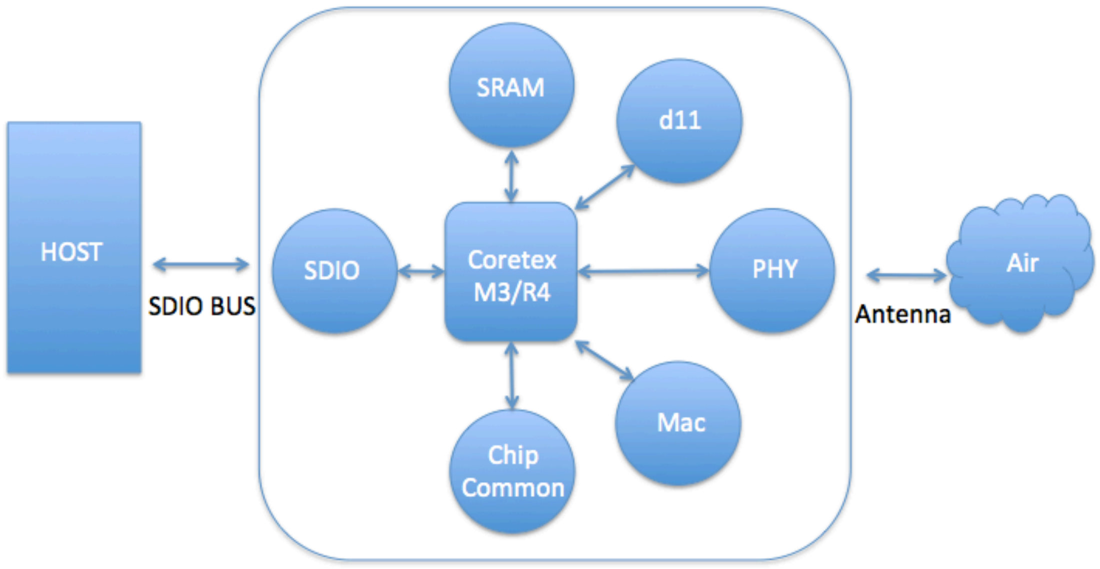

Figure 1: Logic diagram of the cad structure

The firmware

The firmware for the card consists of two sections or regions containing both data and code. A first region is loaded by the host (mobile) device into the card volatile RAM, by a process called *firmware upload*; we will call this "Region 1". The second region is non-writable and is part of an EPROM type memory. The firmware code for Region 1 is protected by a simple CRC checksum for integrity, modification of this region is simple and has been previously demonstrated [1].

Communication

As mentioned before, the mobile device OS (iOS/Android/Windows) wireless driver communicate with the Wi-Fi card over an SDIO bus. This bus has different capabilities like SPI-mode, but in this case there are no DMA capabilities involved. However, a higher-level protocol is layered on top of this bus, which enables communications by means of a set of IOCTLs commands. It is important to differentiate these IOCTLs from those exposed on the user-mode to kernel-mode boundary, to differentiate we will code the first *"Firmware defined IOCTL"* and the later *"user-to-kernel IOCTL".* The following figure illustrates the communication.

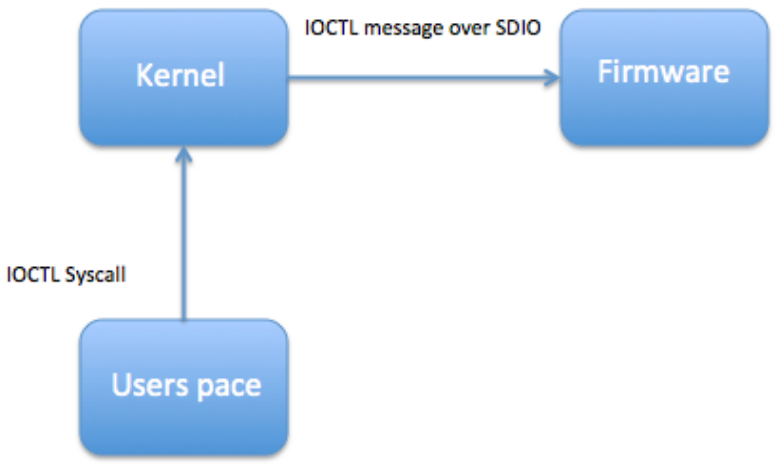

Figure 2: IOCTL messages send at different layers

Proposed Solution

In order to accomplish our objective of remaining as card neutral as possible, we will modify the firmware implementation of the Firmware defined IOCTLs. Our modification is going to introduce two new IOCTL commands that will allow us to read and write memory respectively. Even though we will need to perform this step individually and rather manually on each specific network card version and model, once this read-and-write interface is provided we will be able to work uniformly with different card models and versions across different operating systems. In other words, this would be the only version specific modification we propose; the rest of out implementation will be portable.

Our first step will consist of identification of the firmware defined IOCTL handler code on the firmware program, this is, find the code that processes the received IOCTL command at the firmware. In order to accomplish this task, we will first disassemble the firmware (see [1] for details of how to proceed). We will rely on IDA Pro to accomplish this, once we have the disassembled version Region 1 of the firmware program, the simplest way to identify this portion of code is simply to search for all switch idioms within the disassembly. The IOCTL handler typically resides in Region 1, and consists of a switch with about 300 “cases”.

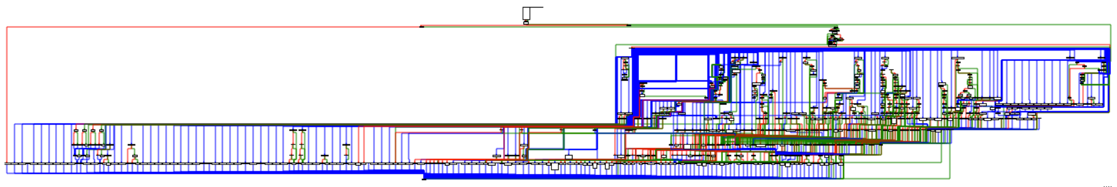

Figure 3: Ida pro graph of the large number of cases implemented by IOCTL handler switch

Once we have identified such code, we will then proceed to modify it in order to add two new IOCTL functions: memory read and write operations. This can be accomplished, for example, by placing our own code over one of the firmware strings, then modifying the firmware code that handles the IOCTL to implement a hook, this means to jump to our code that was placed over a string. For example, for iOS 8.1.2 at iPhone 5s, we would modify the string “smdebug=%08x,phydebug=%08x,psm_brc=%08x\nwepctl=%” (debug string that is not normally being used by the firmware) so that it is overwritten with the following code:

```
0004B6C4    read_write_impl
0004B6C4      CMP.W   R1, #0xFA00 ;Read cmd code
0004B6C8      BEQ     read_cmd
0004B6CA      CMP.W   R1, #0xFB00 ;Write cmd code
0004B6CE      BEQ     write_cmd
0004B6D0      MOV     R7, R0
0004B6D2      MOV     R6, R1
0004B6D4      BX      LR
0004B6D6    ; ----------------------------------
0004B6D6
0004B6D6    read_cmd
0004B6D6      MOV     R0, R2
0004B6D8      LDR     R1, [R2]
0004B6DA      LDR     R2, [R2,#4]
0004B6DC      B       done
0004B6DE    ; ----------------------------------
0004B6DE
0004B6DE    write_cmd
0004B6DE      LDR     R0, [R2]
0004B6E0      ADDS.W  R1,  R2,#8
0004B6E4      LDR     R2, [R2,#4]
0004B6E6
0004B6E6    done
0004B6E6      LDR     R3, =(memcpy+1)
0004B6E8      BLX     R3 ; memcpy
0004B6EA      MOVS    R0, #0
0004B6EC      POP.W   {R2-R8,PC}
0004B6EC    ; END OF FUNCTION read_write_impl
0004B6EC    ; ----------------------------------
0004B6F0  DCD memcpy+1
```

At this point the firmware now supports two new messages that read and write to arbitrary memory locations. One way to get these new messages to be executed would be to modify the host (mobile device) driver so that it sends these new messages, however this would result in dependency on the operating system, in other words, we would need to create patches for the drivers of each of the mobile devices we want to support. In order to avoid this dependency, we will rely on an existing user-to-kernel IOCTL that is already implemented by Broadcom’s drivers within the Operating System kernel. This particular IOCTL, will allow us to send custom messages from user-space, these messages will be encapsulated over the SDIO bus protocol to finally be handled by the firmware. Minor differences exits for each operating system, but the differences are not complex to manage.

For the Apple case we will be using SIOCSA80211, and within the payload of this IOCTL we will send the message APPLE80211_IOC_CARD_SPECIFIC. For the Android case we will be using SIOCDEVPRIVATE. These combinations will allows us to send "firmware defined IOCTL's" over user-to-kernel IOCTL's. To clarify: we will be sending an IOCTL inside an IOCTL. These messages will let us communicate with our firmware primitives read and write. Once we have done this, we will continue by developing user space code, we have chosen python as our language of choice, that will rely on these read and write primitives in an OS independent and firmware version independent manner.

We will then build read and write primitive wrappers in python. These wrappers simply send the relevant user-to-kernel IOCTLs. At this point we can move forward with the development of our tracing tool. The tracing tool will provide hooking functionality in a similar manner to the implementation of the read and write function. Whenever inspection of the value of a register or memory location at a specific code address is desired, our tracer will hook this address so that a jump to the handler code is injected. The handler will then copy the value contained by the register or address of interest to the storage area. The relevant portions of the tracer code look like this:

```
from rawio import read, write
import bcalc
import struct

class Tracer:
    HookAddr   = 0x4B6F4 # (4334- iphone 5s- 8.1)
    DataAddr   = 0x4B72C

    def __init__(self, point, register):
        self.point = point
        self.register = register
        self.sizeOfData = 0x10 # Size of our storage.

    def createHook(self, pointCode):
        code = (
            "00BF" #   NOP ; placeholder for the
            "00BF" #   NOP ; instructions smashed by jmp.
            "07B4" #   PUSH    {R0-R2}
            "00BF" #   NOP ; placeholder for a mov.
            "0449" #   LDR     R1, =DataAddr
            "0A68" #   LDR     R2, [R1]
            "102A" #   CMP     R2, #0x10
            "02D0" #   BEQ     done
            "0432" #   ADDS    R2, #4
            "0A60" #   STR     R2, [R1]
            "8850" #   STR     R0, [R1,R2]
                   #   done
            "07BC" #   POP     {R0-R2}
            "7047" #   BX      LR
            "0000" #   align
                   # "A02C0200" ; DataAddr goes here.
            ).decode('hex')
        code += struct.pack("<L", self.DataAddr)

        code = code.replace('\x00\xbf\x00\xbf', pointCode)
        code = code.replace('\x00\xbf', self.assembleMov())
        return code

    def hook(self):
        # Setup data region
        self.dataBackup = read(self.DataAddr, self.sizeOfData)
        write(self.DataAddr, '\x00' * self.sizeOfData)

        # Setup code region
        self.pointBackup = read(self.point, 4)
        hookCode = self.createHook(self.pointBackup)
        self.hookBackup = read(self.HookAddr, len(hookCode))
        write(self.HookAddr, hookCode)

        # Setup hook call
        write(self.point, bcalc.bl(self.point, self.HookAddr))
```

Tracer constructor simply stores the address of the code on which we are interested to inspect the state (point address) together with the register of interest. Function *CreateHook*, will create the code that we will call from the point of interest. This code gets rendered by: assembling a mov instruction is to copy the contents of the register of interest to "R0", replacing the first to NOP instruction with the code that was originally at the point of interest appending the storage area address were we will be storing the values. Function *hook* creates a backup of the data stored at the addresses that we will be using as storage and initializes the area with zeros, reads the instructions that will be replaced by the call to our hook (*pointBackup*). The rendered code is placed at the defined address and finally the code at the point of interest is replaced by a branch-with-link (call) instruction.

This basic tracer is not without its limitations: 1. Since code at the trace point will be relocated, instructions that are not position independent can't be traced. 2. Instructions that depend on the previous state of the CPU status flags will not have the expected behavior. 3. Since branch-with-link instruction was used (call) the link will be overwritten and wont have a meaningful value. 4. Since the instructions at the tracepoint are replaced, the tracer will only work for addresses that we are able to write at.

We can address these limitations in different ways, we consider the most relevant to be 4. As was mentioned before the entire code on Region 2 is not writable this limits our tracer to Region 1. In order to provide a more dynamic analysis of the code on Region 2, we need a way to address such limitation.

To accomplish our objective, we will rely on a feature present on the Cortex-

M3 (the micro processor used on most of the mentioned cards) called FlashPatch breakpoint [6]. The FPB unit allows, by means of a remap table, interception of the opcodes when the fetch operation of the fetch-decode-execute cycle is being carried out. As described in the documentation, by setting up a remap table, the comparator registers and enabling the FPB unit by means of the FP_CTRL register we can create the intended effect of a "flash patch", this is modification of non-writable code.

Furthermore we consider the FPB feature of these CPUs to be of interest in relation to the analysis of code integrity from a security perspective. The configuration of this unit allow us to "modify" code that was considered read-only, additionally since the FPB unit works by altering the fetch process and since these CPUs Cortext M3 processors use different internal buses for data and code reads (See Figure 4) a situation in which the executed code is different from the code that can be read as data by the very CPU is achieved. This situation allows for the possibility of a non-persistent rootkit: This rootkit would work by modifying the execution of the firmware at certain addresses and yet reading the firmware memory at those addresses will not reveal the modification. A protection against such condition might rely on detecting the remap table in memory, however it was discovered by experimenting with the FPB that the remap table itself can be remapped. This results in correct operation of the FPB feature while hiding the remap table. Ultimately, the detection of this hiding technique can be accomplished by inspection of the FP_CTRL register. The address of this control register cannot be remapped since the comparator register does not allow for addresses that go as high as the necessary addresses.

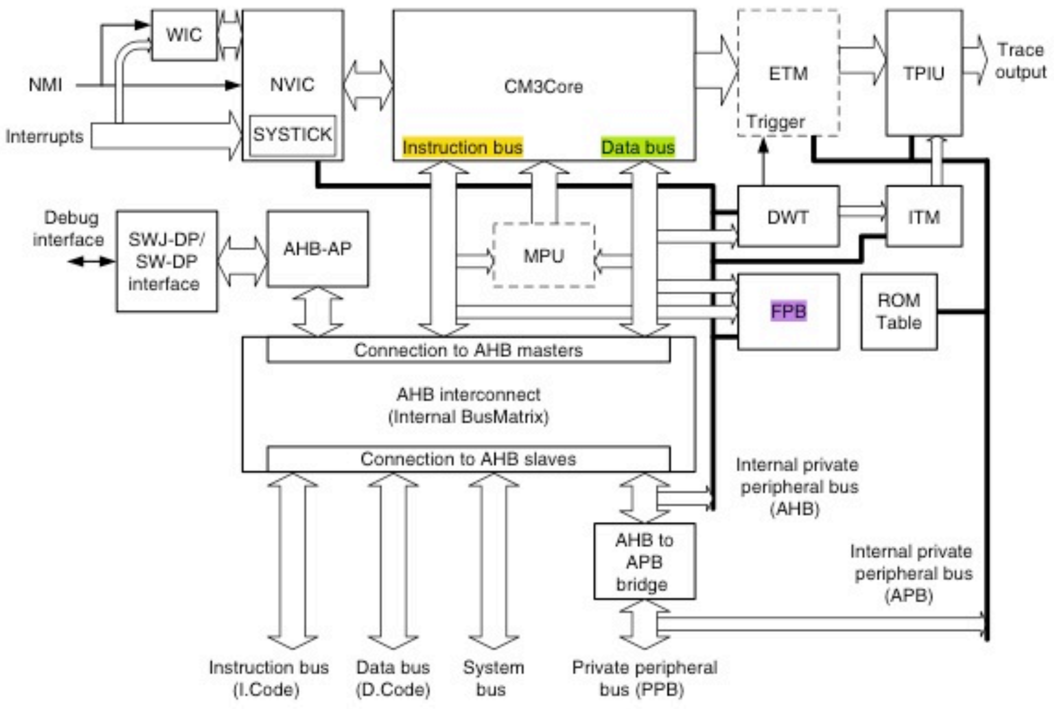

Figure 4: Logical diagram of the Cortex M3 and its components [7]

Continuing with the tracer, we have now achieved tracing capabilities of the code in both memory regions. Since the configuration of the FPB unit only consist of memory mapped registers and we already had write primitives that could be used for our purposes, we can implement this functionality by simply writing to the adequate memory addresses that are required. We will use this new development to study 802.11 handling code.

Finally, in order to inspect code of interest we need to point our tracer python program to the address(es) that we are interested in evaluating, together with the register we are interested to obtain. A full example of this can be found under the appendix of this document.

References

[1] http://archive.hack.lu/2012/Hacklu-2012-one-firmware-Andres-Blanco-Matias-Eissler.pdf
[2] http://bcmon.blogspot.nl/
[3] Guillaume Delugré. Closer to metal: reverse-engineering the broadcom netextreme's firmware. Hack.lu, 2010.
[4] Arrigo Triulzi. Project maux mk. ii, i own the nic, now i want a shell. The 8th annual PacSec conference, 2008.
[5] Lo¨ıc Duflot, Yves-Alexis Perez, Guillaume Valadon, and Olivier Levillain. Can you still trust your network card? CanSecWest Applied Security Conference, 2010.
[6] http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337e/DDI0337E_cortex_m3_r1p1_trm.pdf
[7] The definite guide to the ARM Cortex-M3, Joseph Yiu.

IOCTL.PY

```
#!/usr/bin/env python

import ctypes
import socket
import struct
import subprocess

# Apple ioctl codes # 32bit
SIOCGA80211 = 0xC02069C9
SIOCSA80211 = 0x802069C8


# broadcom wl_ioctl codes
WLC_MAGIC       = 0
WLC_GET_VERSION = 1
WLC_GET_CHANNEL = 29
WLC_SET_CHANNEL = 30
WLC_GET_RADIO   = 37
WLC_SET_RADIO   = 38
WLC_GET_VAR     = 262
WLC_SET_VAR     = 263

class apple80211req(ctypes.Structure):
    _fields_ = [("ifname",   ctypes.c_char * 16),
                ("req_type", ctypes.c_int),
                ("req_val",  ctypes.c_int),
                ("req_len",  ctypes.c_uint),
                ("req_data", ctypes.c_void_p)]

def wl_ioctl(cmd, buff=''):
    req = apple80211req()
    req.ifname   = "en0\0"
    req.req_type = APPLE80211_IOC_CARD_SPECIFIC
    req.req_val  = cmd

    if len(buff) != 0:
        buff         = ctypes.create_string_buffer(buff)
        req.req_data = ctypes.cast(buff, ctypes.c_void_p)
        req.req_len  = len(buff) - 1
    else:
        buff         = ctypes.create_string_buffer(4)
        req.req_data = ctypes.cast(buff, ctypes.c_void_p)
        req.req_len  = 4

    libSystem = 
ctypes.cdll.LoadLibrary("/usr/lib/libSystem.B.dylib")
    s = socket.socket()
    if libSystem.ioctl(s.fileno(), SIOCSA80211, ctypes.byref(req)) 
!= 0:
        libSystem.__error.restype  = ctypes.POINTER(ctypes.c_int)
        libSystem.strerror.restype = ctypes.c_char_p
        errno = libSystem.__error().contents.value
        raise Exception("ioctl error: %s" % 
libSystem.strerror(errno))

    s.close()
    return ''.join(x for x in buff)
```

```python
def test_ioctl():
    magic = wl_ioctl(WLC_MAGIC)
    return (struct.unpack("<L", magic)[0] == 0x14e46c77 and
            struct.unpack("<L", wl_ioctl(WLC_GET_VERSION))[0] == 1)


def get_channel():
    """ returns (current, target, scan) channels """
    chan = wl_ioctl(WLC_GET_CHANNEL, '\x00' * 12)
    return struct.unpack("<LLL",chan[:-1])

def set_channel(number):
    wl_ioctl(WLC_SET_CHANNEL, struct.pack("<L", number))

def get_radio():
    return struct.unpack("<L", wl_ioctl(WLC_GET_RADIO))[0]

def set_radio(status):
    mask = 7
    status = struct.pack("<l", (mask << 16) | status)
    wl_ioctl(WLC_SET_RADIO, status)

def get_intvar(var):
    return struct.unpack("<L", wl_ioctl(WLC_GET_VAR, var +
'\0')[:4])[0]

def set_intvar(var, val):
    wl_ioctl(WLC_SET_VAR, var + '\0' + struct.pack("<L", val))


if __name__ == "__main__":
    if not test_ioctl():
        raise Exception("test failed")
    else:
        print 'test ok'
```

RAWIO.PY

```python
import ioctl
import struct

def read(addr, length):
    buf = struct.pack("<LL", addr, length)
    buf += "\x00" * (length - len(buf))
    return ioctl.wl_ioctl(0xfa00, buf)[:length]

def write(addr, data):
    buf = struct.pack("<LL", addr, len(data))
    buf += data
    ioctl.wl_ioctl(0xfb00, buf)

if __name__ == "__main__":
    print read(0x4B6AB, 8)
```

FPBTRACE.PY

```python
from rawio import read, write
```

```
import bcalc
import struct
import time

class Hook:
    def __init__(self, point, hookAddr, dataAddr):
        self.point = point
        self.pointCode = read(point, 4)
        self.hookAddr = hookAddr
        self.dataAddr = dataAddr

    def registerNumber(self, register):
        regNum = register[1:]

        try:
            ret = int(regNum)
        except ValueError, e:
            if register.upper() == 'SP':
                ret = 13
            elif register.upper() == 'LR':
                ret = 14
            else:
                raise ValueError('Unknown Register:' + register)

        return ret

    def assembleMov(self, dst, src):
        mov = '\x00\xbf' # nop
        srcReg = self.registerNumber(src)
        dstReg = self.registerNumber(dst)

        if srcReg >= 0 and srcReg <= 7:
            mov = chr((srcReg << 3) | dstReg) + '\x00' # 08 00 ->
movs r0, r1; 10 00 -> movs r0, r2 ... etc
        elif srcReg > 7 and srcReg <= 14:
            srcReg -= 8
            mov = chr(0x40 | (srcReg << 3) | dstReg) + '\x46' # 40
46 -> mov r0, r8; 48 46 -> mov r0, r9 ... etc
        else:
            raise ValueError('Invalid register number:' +
self.register)

        return mov

    def render(self):
        raise NotImplementedError()

class RegisterHook(Hook):
    def __init__(self, point, hookAddr, dataAddr, register):
        Hook.__init__(self, point, hookAddr, dataAddr)
        self.register = register

class RegisterMov(RegisterHook):
    def render(self):
        code = (
            "00BF"      #   NOP
            "00BF"      #   NOP
            "07B4"      #   PUSH    {R0-R2}
            "00BF"      #   NOP
            "0449"     #   LDR     R1, =sub_22CA0
            "0A68"     #   LDR     R2, [R1]
            "102A"     #   CMP     R2, #0x10
            "02D0"     #   BEQ     done
            "0432"     #   ADDS    R2, #4
            "0A60"     #   STR     R2, [R1]
            "8850"     #   STR     R0, [R1,R2]
                       #   done
            "07BC"     #   POP     {R0-R2}
            # "A02C0200"
        ).decode('hex')
        code += bcalc.bw(self.hookAddr + len(code), self.point + 4)
        code += struct.pack("<L", self.dataAddr)

        code = code.replace('\x00\xbf\x00\xbf', self.pointCode)
        code = code.replace('\x00\xbf', self.assembleMov('R0',
self.register))
        return code

class RegisterPtr(RegisterHook):
    Memcpy = 0x0080C41C + 1

    def __init__(self, point, hookAddr, dataAddr, register, size):
        RegisterHook.__init__(self, point, hookAddr, dataAddr,
register)
        self.size = size

    def render(self):
        code = (
            "00BF"       # NOP     ; smashed instruction 1
            "00BF"       # NOP     ; smashed instruction 2
            "0FB5"       # PUSH    {R0-R3,LR}
            "4DF804CD"   # STR.W   R12, [SP,#-4]! ; a.k.a PUSH R12
            "00BF"       # NOP     ; mov r1, srcreg
            "0748"       # LDR     R0, dataAddress
            "0268"       # LDR     R2, [R0]
            "002A"       # CMP     R2, #0
            "04D1"       # BNE     done
            "064A"       # LDR     R2, dataSize
            "0260"       # STR     R2, [R0]
            "0430"       # ADDS    R0, #4
            "064B"       # LDR     R3, =(memcpy+1)
            "9847"       # BLX     R3
                         # done
            "5DF804CB"   # LDR.W   R12, [SP],#4 ; a.k.a POP R12
            "BDE80F40"   # POP.W   {R0-R3,LR}
            #"00BF00BF"  # NOP NOP ; jump back1
            #"0000"      # NOP  ; align
            #"41414141"  # dataAddress DCD 0x41414141
            #"41414141"  # dataSize    DCD 0x41414141
            #"1DC48000"  # memcpyaddr  DCD memcpy+1
            ).decode('hex')

        code += bcalc.bw(self.hookAddr + len(code), self.point + 4)
        code += "0000".decode('hex') # ALIGN
        code += struct.pack("<L", self.dataAddr)
        code += struct.pack("<L", self.size)
        code += struct.pack("<L", self.Memcpy)
        code = code.replace('\x00\xbf\x00\xbf', self.pointCode)
        code = code.replace('\x00\xbf', self.assembleMov('R1',
self.register))

        print 'len(code)', len(code), code.encode('hex')
        return code


class Tracer:
    HookAddr   = 0x4B6F4 # (4334-iphone5s-8.1)
    DataAddr   = 0x4B72C
    RemapTable = 0x4B600
    FP_CTRL    = 0xE0002000
    FP_REMAP   = 0xE0002004
    FP_COMP0   = 0xE0002008

    def __init__(self, point, register, size=None):
        self.point = point
        self.register = register
        self.sizeOfData = 0x20
        self.size = size

    def fpbHook(self):
        branchCode = bcalc.bw(self.point, self.HookAddr)
        remaps = self.remaps = []

        if self.point % 4 == 0:
            remaps.append( (self.point, branchCode) )
        elif self.point % 2 == 0:
            addr1 = self.point - 2
            addr2 = self.point + 2

            inst1 = read(addr1, 4)[:2] + branchCode[:2]
            inst2 = branchCode[2:] + read(addr2, 4)[2:]

            remaps.append( (addr1, inst1) )
            remaps.append( (addr2, inst2) )
        else:
            raise ValueError('Trace address must be half-word
aligned')

        # point remap register to the remap table
        write(self.FP_REMAP, struct.pack("<L", self.RemapTable))

        # setup remap table with replacement instructions to be
executed at point
        self.remapTableBK = read(self.RemapTable, 4*len(remaps))
        for i, (addr, code) in enumerate(remaps):
            print 'remapping', hex(addr), '->', code.encode('hex')
            write(self.RemapTable + i*4, code)

            # Setup FP_COMP Register pointing to the instruction to
patch
            compval = (addr & 0x1FFFFFFC) | 1# | 3 << 30
            write(self.FP_COMP0 + i*4, struct.pack("<L", compval))

        # turn on fpb
        write(self.FP_CTRL, '63020000'.decode('hex'))
    def hook(self):
        # Setup data region
        self.dataBackup = read(self.DataAddr, self.sizeOfData)
        write(self.DataAddr, '\x00' * self.sizeOfData)

        # Setup code region
        if self.register.startswith('['):
            register = self.register.replace('[', '').replace(']',
'')
            h = RegisterPtr(self.point, self.HookAddr,
self.DataAddr, register, self.size)
        else:
            h = RegisterMov(self.point, self.HookAddr,
self.DataAddr, self.register)

        hookCode = h.render()
        self.hookBackup = read(self.HookAddr, len(hookCode))
        write(self.HookAddr, hookCode)

        # Setup hook call
        self.fpbHook()


    def fpbUnhook(self):
        write(self.FP_CTRL, '62020000'.decode('hex'))
        for i, (_, _) in enumerate(self.remaps):
            write(self.FP_COMP0 + i*4, '\x00' * 4)
        write(self.FP_REMAP, '\x00' * 4)
        write(self.RemapTable, self.remapTableBK)

    def unhook(self):
        # Restore hook call
        # write(self.point, self.pointBackup)
        self.fpbUnhook()
        write(self.HookAddr, self.hookBackup)
        write(self.DataAddr, self.dataBackup)

    def traces(self):
        size = read(self.DataAddr, 4)
        size = struct.unpack("<L", size)[0]

        if self.size:
            ret = ''
            if size > 0:
                ret = read(self.DataAddr+4, size)
                write(self.DataAddr, '\x00' * self.sizeOfData)
      print " returning", ret

            return ret

        values = read(self.DataAddr+4, size)
        values = struct.unpack("<" + "L" * (size/4), values)

        write(self.DataAddr, '\x00' * self.sizeOfData)
        return values

class Printer:
    def __init__(self, addr, register, size=None):
        t = Tracer(addr, register, size)
        t.hook()
        sizeOfData = 32
        DataAddr =  0x4B728
        try:
            while True:
                  print ' '.join(hex(x) for x in t.traces())
                time.sleep(1)
        finally:
            t.unhook()

if __name__ == '__main__':
   Printer(0x26CBA, 'R2')
```