Embedded devices as an attack vector

Stephen Lewis
Stephen.Lewis@cl.cam.ac.uk

Computer Laboratory
University of Cambridge

21C3

Outline

Embedded devices

- Switches, printers, routers etc.
- Proprietary OS
- Small amount of RAM, NVRAM, FLASH
- Management interface accessible using SNMP/telnet
- No documentation: ‘closed’ devices
- Not interested in larger devices (with ‘real’ OS)

Threat model

- Attacker has some access to the management interface of the device over the network
- This is realistic for a large number of devices
- For example, consider a switch at the edge of a network (with the management interface on the same VLAN as the users)
- (But no exploits in the case study)

Aims

- Run own code on embedded devices without replacing original functionality
- The device should not become unstable (or crash!)
- Understand enough of the networking code to send/receive packets on management interface

Traceability

- Embedded devices are unlikely to keep useful logs
- Nobody expects spoofed packets to originate from an embedded device (e.g. an Ethernet switch)
- At least with some devices, it is easy to make your code lie dormant until you are long gone (hook into some kind of interrupt/event handler)

Detectability

- With a lot of hard work, you can make yourself undetectable, even on the device itself
- Write your own custom firmware
  - Contains your attack code
  - Hides its own presence
  - Makes it impossible to re-flash with 'good' firmware
- Some devices have a catch-all recovery mode in ROM, but this might not be documented

Starting off

- In most cases, no technical documentation
- Need to get (some) code off the device
  - Or maybe not: firmware image could be available
  - Firmware images can be difficult to decode: you don't know where various segments get mapped
  - Use (undocumented) debugging modes to dump memory
- Architecture may be unfamiliar (case study 68k-based)
- Very little (or no) prior knowledge about the memory map

Your enemies

- Dynamic memory allocation
- Complex structures with lots of function pointers
- Time-sensitive event handling code
- Custom scripting languages
- Proprietary filesystems
- I/O buffering code

Taking the device apart

- This isn't a hardware talk, but...
- ...you can gather a lot of useful information by just looking!
- Get some idea of which memory devices/microprocessors are present
- Architectural hints (what's done in ASICs/FPGAs/what's left to general purpose microprocessors)

Documentation

- Read whatever you can get your hands on
  - Make sure you know the features of the device well; this will provide clues for how to proceed
  - Get hold of documentation for microprocessor/architecture of the device
- Write your own documentation too
- It's important to keep track of what you know, and refer back to it when you get stuck

'Local' reverse engineering

- Never underestimate the benefits of running code on the device itself
- Often much quicker than doing static analysis of memory dumps
- I/O functions on the device itself can be expensive
- Build up an armoury of tools
  - Print out data structures you understand
  - Walk more complex structures (e.g. lists, tables)
  - Test out hypotheses about what various functions do

Errors are good!

- Spend some time working out what information you can gather from getting into an error state
- Some devices will have hardware/software watchdog timers that will cause a reset
- Try to find something giving you a post-mortem of the crash (cf. `hist` command in case study)
  - Stack dumps
  - System error codes

Health warning

- It's quite hard to cause permanent damage to these devices, but...
- Be careful what you do with the NVRAM
- Especially if you don't know where it is!
- Careful preparation is necessary if you're planning on rolling your own firmware

3Com Switch 3300

- All the usual features of an Ethernet switch
- 12/24 autosensing 10BASE-T/100BASE-TX ports
- Services running
  - SNMP agent
  - Command line accessible via `telnet` (or serial port)
  - Web server
  - (also responds to ICMP ping)
- Software upgrade via TFTP

The gory details

- Motorola 68EC020 microprocessor (like 68020 but with 24-bit address bus)
- 32 MiB RAM (I think!)
- NVRAM, FLASH, small ROM
- Approximately 5 MiB code (not including scripts)
- Custom hardware for switching (e.g. fast look up table, Ethernet PHYs)

Getting a firmware image

- Download from manufacturer
  - Not an ideal solution: probably no memory map, but...
  - ...easy to do
- Download from device
  - Need (undocumented) commands (run `strings` on downloaded image, 'ask' manufacturer...)
  - Reveals memory map
  - Reading sensitive regions may cause problems

Getting a firmware image

```
Select menu option: ?bug
Bug (q to return)> pre mem
Bug (q to return)> dump 0

00000000: 00 40 19 fc 00 00  [...]  .@......  ...1.00.
00000010: ff ff ff ff 48 e7  [...]  ....H...  2..Y.A..
```

- All that's left to do is script this...
- ...and wait, and wait, and wait
- Make sure you handle errors gracefully

Exploring undocumented commands

- This device has lots of them
- Two modes accessible from normal CLI
  - `?bug`
  - `?debug`
- `?bug` mode gives access directly to hardware
- `?debug` mode accesses a wider variety of commands, written in scripting language

Crash post mortem (with hist)

- Gives reboot history, with reasons
  - Unknown, Power, Watchdog, Manual Reset, Defaults, Stack Conflict, System Error, Over Temp., POST Reset, S/W Upgrade, S/W Watchdog, Mini Upgrade, Assertion, Serial Upgrade, Exception
- Current value of stack pointer
- Dump of stack frame
- Other register values
- Stack traceback (very useful)
- Demo...

Beginning disassembly

- Stack trace can give a good starting point
- Initially, we're looking for some high-level I/O code
- This isn't too hard

```
...
cmpi.b  #$25,var_D(a6) ; '%'
...
```

- Or just look for places where it's called

```
...
move.l  #$A0654,-(sp)    ; " Empty\n" at 0xA0654
jsr     sub_36794
...
```

Running our own code

- We know where the stack is
- Manipulate return addresses to hook into control flow
- Tools
  - Toolchain to target m68k (`m68k-elf-as`, `m68k-elf-ld`)
  - Appropriate linker script
  - Perl script to load code into device
- Simple stuff, but it convinces us that we can execute code from RAM, and that we've understood the stack layout

Linker script

```
OUTPUT_FORMAT("srec")
OUTPUT_ARCH("m68k")
OUTPUT("3300.srec")

SECTIONS
{
        . = 0x445000;
        .text : { *(.text) }
        .data : { *(.data) }
}
```

hello.s

```
.data
hello:
.ascii "Hello world!\n\0"
.set   printf,  0x36794

.text
   movem.l  %a0-%a5/%d0-%d5,-(%sp)

   move.l   #hello,-(%sp) | Print test string
   jsr      printf
   addq.l   #0x4,%sp

   movem.l  (%sp)+,%a0-%a5/%d0-%d5
   jmp      0x00036944
```

hello.srec

- Run `m68k-elf-as` and `m68k-elf-ld` to get:

```
S00D000068656C6C6F2E7372656303
S21444500048E7FCFC2F3C0044501C4EB90003679410
S210445010588F4CDF3F3F4EF900036944C4
S21244501C48656C6C6F20776F726C64210A00D6
S80444500067
```

- Does this really work?

Other techniques

- 'Local' reverse engineering
  - Useful for, for example, searching the RAM
  - Or copying data structures and producing diffs
  - Make sure you code bounds checks on any pointers you follow!
- Comparison across reboots
- Finding references to string constants
- Looking at statically allocated data structures
- Code referenced in failed assertion handlers

Problems with this device

- Scripting language
  - Quite a lot of the functionality is implemented in the scripting language
  - Appears to be some kind of runtime-interpreted bytecode
  - Interpreter is hard to understand
  - (But there is a 'compiler' on the device itself)
- Function pointers fairly heavily used
- Most memory is dynamically allocated: this means you have to traverse data structures in order to reliably find things

Plenty of starting points

- Seach for own IP/MAC?
- ARP tables?
- Follow I/O code from high-level functions?
- TCP/UDP checksum code?
- Search for `move.w #$17,d??`
- TCP connection control blocks?
- TFTP code?
- ICMP echo response?
- Memory mapped I/O into the correct regions?

So it's easy?

- Yes, in that it's easy to find the code and data structures, but...
- ...no, because it's very hard to understand enough code to actually call any of these functions
- Low down in the network stack, the structures used are complex (lots of interdependencies between functions)
- Higher in the stack, there's not enough flexibility to send what you want to send!

Results

- A large number of long-lived network data structures found
- Short-lived ones are more problematic
- There is an event handler in RAM(!), which seems to be hooked into the control flow for incoming network packets
- Most promisingly, can now fairly reliably create already-open TCP connections
- Basis of a simple port scanner here?
- Can change MAC address of originated packets
- Buffered incoming network packets seem to hang around for a while: easy to capture if you know what you're looking for

Conclusions

- Use of embedded devices as an attack vector is
  - difficult to detect
  - difficult to trace
- On-device reverse-engineering is not a black art
- Using existing functionality from your own code is hard, at least in moderately complicated devices
- Errors (with crash dumps) are good
- Code will be at `http://www.cl.cam.ac.uk/šrl32/21c3/` (but not yet!)