Introduction

Overview

• Embedded Systems Basics
• Real Time OS
• The ARM Architecture
• The JTAG Interface
• The UART Interface
• Introduction To the ICE (In-Circuit Emulator)
• Interfacing With The Embedded System
• Reverse Engineering And Debugging
• Defeating The Watchdog
• Exploiting The Vulnerability
• Shell-code Example
Embedded Systems Are Everywhere

Automobiles, Cell-phones, Routers, Microwaves – Embedded devices are an integral part of our daily lives.

The popularity of internet-connected and wireless devices is rapidly increasing

Where there is code – There are flaws!
The Target

**DI-604 Broadband Router**

A popular home router – ARM9E 150Mhz processor, 1MB FLASH, 8MB SDRAM, 6 ETH PORTS, ThreadX RTOS
Embedded System OS

**Thread-Based:**
VxWorks, ThreadX, Integrity, etc

**Process-Based:**
RTLinux, OS-9

Thread-based RTOS’ are widely used due to fast performance and low overhead.
The ARM Architecture

**The ARM9E-S Processor**

The ARM9E processor in the DLINK is a synthesizable version of the ARM9TDMI core.

Implements an extended version of the ARM instruction set

 Supports full ARM architecture v5TE.
ARM Assembly

The ARM is a typical RISC architecture with some additions:

- Control of the ALU and shifter in every data processing instruction
- Auto-increment and auto-decrement addressing modes
- Load and Store multiple instructions
- Conditional execution on all instructions

ARM has 31 general purpose 32 bit registers – 16 registers are visible at any one time.

R13 and R14 have special roles – the link register and program counter.
The ARM Architecture

**ARM exception vectors**

<table>
<thead>
<tr>
<th>Exception type</th>
<th>Mode</th>
<th>Normal address</th>
<th>High vector address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Supervisor</td>
<td>0x00000000</td>
<td>0xFFFFF0000</td>
</tr>
<tr>
<td>Undefined instructions</td>
<td>Undefined</td>
<td>0x00000004</td>
<td>0xFFFFF0004</td>
</tr>
<tr>
<td>Software interrupt (SWI)</td>
<td>Supervisor</td>
<td>0x00000008</td>
<td>0xFFFFF0008</td>
</tr>
<tr>
<td>Prefetch Abort (instruction fetch memory abort)</td>
<td>Abort</td>
<td>0x0000000C</td>
<td>0xFFFFF000C</td>
</tr>
<tr>
<td>Data Abort (data access memory abort)</td>
<td>Abort</td>
<td>0x00000010</td>
<td>0xFFFFF0010</td>
</tr>
<tr>
<td>IRQ (interrupt)</td>
<td>IRQ</td>
<td>0x00000018</td>
<td>0xFFFFF0018</td>
</tr>
<tr>
<td>FIQ (fast interrupt)</td>
<td>FIQ</td>
<td>0x0000001C</td>
<td>0xFFFFF001C</td>
</tr>
</tbody>
</table>
The Marvell 88E6218

The 88E6218 chipset is specifically designed for Router applications. Processor is ARM9E based, running at 150MHZ, 7 Switch ports, 1MB embedded memory, 16 GPIO ports, 1 UART

Supports both the ARM and THUMB instruction set.
The JTAG Interface

Introduction to JTAG

The JTAG interface is supported on-chip.

Five dedicated signals must be provided on each chip that supports the JTAG standard.

- **TRST** – Test-Reset: initializes and disables the test interface
- **TCK** – Test Clock: independent timing control
- **TMS** – Test Mode Select: controls state transitions
- **TDI** – Test Data Input: supplies data to the JTAG registers
- **TDO** – Test Data Output: outputs data from the JTAG registers.
The JTAG Interface

Locating the JTAG test points

Acquire vendor chip pin-out.
Use voltmeter to verify connections.
Possibilities:
• Full JTAG connector on PCB
• Individual JTAG points
• No JTAG points (solder directly to chip)
The JTAG interface

Building a JTAG Connector

If a pre-installed JTAG connection is not available, a JTAG connector must be built.

This schematic is for the ARM Multi-ICE.
The ICE (In-Circuit Emulator)

The ARM Multi-ICE

The Multi-ICE supports all current ARM processors:
ARM7, ARM9, XSCALE, etc
Fast and reliable, supported by most debuggers via RDI interface.
The UART Interface

The Serial Interface

Most chipsets will support a UART interface.

The serial interface can be used to view program output over a serial connection.

A level-shifter must be incorporated into the adapter. The USBMOD3 level shifts and outputs over USB.
Simulating the UART with macros

Serial output can be viewed by setting breakpoint macros.
Set macro to execute on dbgprint, and use macro to print serial data to console

```
// UART simulator
__var sbuf;

uartsim()
{
    sbuf =__readMemory32(0x00,"Register");
    __message "UART output:", (char *)sbuf, "\n";

    return 0;
}
```
Connected JTAG
Unfortunately, no excellent embedded debuggers – each have their shortcomings.

We use IAR Embedded Workbench

Unfortunately problems exist: breakpoints sometimes flaky, occasional incorrect values.
DEBUGGING DEMO
Retrieving the firmware image

Retrieve the firmware image to disassemble within IDA. Useful for locating vulnerabilities, mapping API calls, etc.

3 options:

- Rip image from flash chip via JTAG
- Dump memory to file from within debugger
- Download firmware image from Vendor (usually encoded)
Retrieving The Firmware Image

There are many standalone products that offer flash reading/writing. Debuggers often include a flash read/write option. Most common flash memory chips are supported.
Firmware Reversing

Most firmware updates are encoded/check-summed.

Decoder routine can be found by reversing the memory dump or live debugging.

The DI-604 firmware is compressed and check-summed to verify original firmware.

Checksum routine is simple to find by reversing “upload firmware” code snippet.

A small tool was written to patch firmware after modification – any hacked firmware may be uploaded.
Analyzing Firmware

Checksum routine

```assembly
; CODE XREF: firmware_checksum+34j
firmware_checksum
    STRFD SP!, {R4}
    MOV R2, #0
    MOV R3, R0
    MOV R12, #0
    B loc_8a98c

; CODE XREF: firmware_checksum+10fj
loc_8a97c
    LDR R0, [R3]
    EOR R2, R2, R0
    ADD R3, R3, #4
    ADD R12, R12, #1

; CODE XREF: firmware_checksum+40fj
loc_8a98c
    MOV R0, R12
    MOV R4, R1
    MOV R4, R4, LSR #2
    CMP R0, R4
    BCC loc_8a97c
    LDR R0, =x6AABB88AA
    CMP R2, R0
    BNE loc_8a90b4
    MOV R0, #0
    B loc_8a99d8

; CODE XREF: firmware_checksum+48fj
loc_8a984
    MOV R0, #0

; CODE XREF: firmware_checksum+48fj
loc_8a998
    LDMFD SP!, {R4}
    BX LR

; End of function firmware_checksum
```

C:\DLINK-4>recheck_ip_patch.bin
DLINK604 Firmware Re-Checksum
520165 Bytes Read.
New checksum: D63C2F6B
917504 Bytes Written.
Many embedded systems implement a watchdog timer. The watchdog can be a dedicated hardware register, or may be implemented with external circuitry. Watchdog will reset embedded system when a debugger is attached.
Defeating the Watchdog

- Write to watchdog register
- Locate watchdog reference and patch
- Trap exception vector 0 (reset)
Exploiting The Vulnerability

Stack Overflow
(upnp)
Exploiting The Vulnerability

Shellcode Creation

ARM9E processor supports switching to thumb state. Thumb state is very helpful for shellcode creation.

ARM:
- 32 bit instructions, word aligned

THUMB:
- 16 bit instructions, half-word aligned. (smaller code size, easier to avoid NULL bytes)

Achieve thumb state by executing BX with state bit cleared in register.
Exploiting The Vulnerability

Basic Instructions

<table>
<thead>
<tr>
<th>ADC</th>
<th>Add with carry</th>
<th>LDSH</th>
<th>Load sign-extended half-word</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Add</td>
<td>LSR</td>
<td>Logical shift right</td>
</tr>
<tr>
<td>AND</td>
<td>AND</td>
<td>MOV</td>
<td>Move register</td>
</tr>
<tr>
<td>ASR</td>
<td>Arithmetic shift right</td>
<td>MUL</td>
<td>Multiply</td>
</tr>
<tr>
<td>B</td>
<td>Unconditional branch</td>
<td>MVN</td>
<td>Move negative register</td>
</tr>
<tr>
<td>Bxx</td>
<td>Conditional branch</td>
<td>NEG</td>
<td>Negate</td>
</tr>
<tr>
<td>BIC</td>
<td>Bit clear</td>
<td>ORR</td>
<td>OR</td>
</tr>
<tr>
<td>BL</td>
<td>Branch and link</td>
<td>POP</td>
<td>Pop registers</td>
</tr>
<tr>
<td>BX</td>
<td>Branch and exchange</td>
<td>PUSH</td>
<td>Push registers</td>
</tr>
<tr>
<td>CMN</td>
<td>Compare negative</td>
<td>ROR</td>
<td>Rotate right</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
<td>SBC</td>
<td>Subtract with carry</td>
</tr>
<tr>
<td>EOR</td>
<td>EOR</td>
<td>STMIA</td>
<td>Store multiple</td>
</tr>
<tr>
<td>LDMIA</td>
<td>Load multiple</td>
<td>STR</td>
<td>Store word</td>
</tr>
<tr>
<td>LDR</td>
<td>Load word</td>
<td>STRB</td>
<td>Store byte</td>
</tr>
<tr>
<td>LDRB</td>
<td>Load byte</td>
<td>STRH</td>
<td>Store half-word</td>
</tr>
<tr>
<td>LDRH</td>
<td>Load half-word</td>
<td>SWI</td>
<td>Software interrupt</td>
</tr>
<tr>
<td>LSL</td>
<td>Logical shift left</td>
<td>SUB</td>
<td>Subtract</td>
</tr>
<tr>
<td>LDSB</td>
<td>Load sign-extended byte</td>
<td>TST</td>
<td>Test bits</td>
</tr>
</tbody>
</table>
Exploiting The Vulnerability

Shellcode Possibilities

• Program the hardware directly (ref: Yuji Ukai, PacSec 2005)
• Reverse Firmware and map out API calls (dump password over port, embedded sniffer.)
• Memory patch to allow debug access to router

But we’ll take a different route.
Exploiting The Vulnerability

How to Own Everyone

Two phase exploitation.

1. Send initial exploit
   • Disable admin password
   • Enable remote configuration

2. Upload hacked firmware
   • Modify firmware, re-checksum
   • Modified code monitors data and injects hostile code
Exploiting The Vulnerability

Initial exploit

Debug/reverse code that opens remote configuration.

```
loc_208F4 ; CODE XREF: ROM:000200D8↑j
E5 91 04 48  40 LDR  R9, =dword_293154
E5 9D 82 13  LDRB R1, [R9,#0x3B2] ; 0x293506 (remote config flag)
E5 31 80 00  CMP  R1, #0
E8 A6 00 91  BEQ  loc_20C20
EB C4 01 4C  BL   set_remote_cfg
E1 80 10 00  MOV  R1, R0
E5 91 04 40  LDR  R8, =dword_293154
E5 9D 82 13  LDRB R1, [R8,#0x3A4]
E5 31 80 00  CMP  R1, #0
E8 A6 00 91  BEQ  loc_20C20
EA FF FF FE  MOV  R0, #0
EB C4 01 4C  BL   save_entry
EA 00 00 02  B    loc_20C20
```

Remote cfg flag: 0x293506

0 = DISABLED 1 = ENABLED
Exploiting The Vulnerability

Initial exploit

Clear Administrator password:

```
34 01 9F E5 93 FD 01 EB 08 10 80 E2 2C 21 9F E5 24 20 02 E2 10 00 02 E2 8F FD 01 EB FF 10 05 E2 D4 80 9F E5
```

LDR R0, =dword_290190 ; stored password
BL sub_A0010
ADD R1, SP, #8
LDR R2, =dword_290180
ADD R2, R2, #0x24
ADD R0, R2, #0x10
BL sub_A0010
AND R1, R5, #0xFF
LDR R0, =dword_293154

Password compared at location 0x290190. Overwrite data with NULL bytes.
Exploiting The Vulnerability

**Initial exploit**

Set up password and remote configuration flags, then directly call the save_settings routine.

Mem-patch end of subroutine to perform a soft reset. *(mov pc, #0)*

Save_settings is called when settings are changed via the dlink web interface.

We also can take advantage of this function 😊
**Final exploit.**

Exploit stack overflow in upnp processing, execute shellcode:

- Uses thumb state to overwrite needed memory locations (password, remote cfg)
- Overwrites the end of the save_settings routine with a branch to the reset vector
- Leaves thumb state, and branches to the save_settings routine
- Router reboots – admin password cleared, remote configuration enabled.
The Injector

- Monitors all web traffic
- Waits for an executable download
- Injects custom executable code into download
- Everyone downloads EXE files 😊
The Injector

Considerations:

• Will need a patch location with a pointer to the IP packet
• Will need to modify each packet that meets criteria
• Will need to re-checksum packet and update fields.
• Only process HTTP traffic, and only executable downloads
Exploiting The Vulnerability

The Injector

Patch Location:

Router checks for malformed IP packets – lets patch!

Patch location has a pointer to the IP packet, replace branch with a branch to our ROM code.

```
30 80 9D E5   LDR   R8, [SP,#0x244+var_214]

loc_6E6C

00 00 58 E3   CMP   R8, #0   ; check ip length
10 00 00 1A   BNE   loc_6EB8   ; jump if not 0
04 30 A0 E1   MOV   R3, R4
18 29 9F E5   LDR   R2, =aPacketDropped
30 19 9F E5   LDR   R1, =alpWithZeroLeng
10 00 A0 E3   MOV   R0, #0x10
3E 8F 01 EB   BL    dbgprint2
FC 28 9F E5   LDR   R2, =word_2900E0
0C 10 92 E5   LDR   R1, [R2,#0x0C]
01 10 81 E2   ADD   R1, R1, #1
0C 10 82 E5   STR   R1, [R2,#0x0C]
```

PATCH
Exploiting The Vulnerability

The Injector – Our ROM Code

- Check for port 80 (HTTP)
- Check headers for download
- Check content-length size
- Check for executable headers
- Inject executable code
- Create pseudo-header
- Re-checksum TCP packet
- Return to original code
Exploiting The Vulnerability

INJECTOR DEMO
Conclusion

Embedded Systems open a whole new world of possibilities
They still have the “classic” bugs which are near non-existant in the software realm
I hope I’ve shown that exploiting hardware isn’t just a “gimmick” and that the threat is real.

Thanks!
NO TOASTERS ARE SAFE!!!