Tutorial

Session 2a: Stack-based Buffer Overflow Vulnerabilities in Embedded Systems

Kristopher Watts, MS and Paul Oman, PhD
University of Idaho

August 11, 2009
Idaho Falls, Idaho
Presentation Overview

1. Architecture Discovery
   • Identify Components
   • Identify Processor in Black Box Device

2. Buffer Overflows in Embedded Devices
   • Von Neumann or Harvard architectures
   • Harvard with Pre-computed Stack

3. Payloads in Embedded Devices
   • Execute Functions/Inject Code

4. Defenses
Goals

• Understand Exploit Development Process
  • How to approach black box
  • What is required to develop and deliver exploit
• Understand Overflows
  • Keil C51
• Understand Possible Payloads and Impact
  • Code Injection
• Better Understand Defense
  • How to approach an embedded device
1. Architecture Discovery

- Approaching New Device

- Identify components that affect software
  - Memory scheme (RAM)
  - Memory scheme (ROM)
  - Boot system
  - Communications

- Identify processor
  - To disassemble binary image
1. Architecture Discovery

- Three main sections for observation
  - Visual Inspection
  - Runtime probing
  - Software inspection
1. Architecture Discovery

- Visual Inspection
1. Architecture Discovery

- **Runtime probing**
  - Pull diagnostic information
  - Attempt to induce errors
  - Fuzz variables
  - TCP/IP finger printing
1. Architecture Discovery

- **Software Inspection**
  - Segments
  - Word size
  - Program addressable
  - Strings embedded through out
    - As opposed to some constants area
  - Error handling
    - Failure messages
1. Architecture Discovery

- Software
- HTML
- Javascript
- GIF
- GIF
- GIF
- HTML
- HTML
- GIF

Diagnostic Strings

ID String: $123456789ABCDEFGH

Evenly Spaced Instructions
Appears to be an interrupt vector table

Size correlates well with 16 bit address space
1. Architecture Discovery

- Proper Disassembly
  - Loop structures
  - If statements
  - Proper Function calls
    - Prologue
    - Epilogue
  - I/O routines
  - Interrupt routines
    - Proper function exit

```assembly
mov DPL, R1 ; Data Pointer, Low Byte
mov DPH, R2 ; Data Pointer, High Byte
mov RESERVED0093, R3 ; RESERVED
dec RESERVED0093 ; RESERVED
movx @DPTR, A

code_1BE:
cjne R3, #0x80, code_1C1 ; '¢'
code_1C1:
jnc code_1CC

mov DPL, R1 ; Data Pointer, Low Byte
mov DPH, R2 ; Data Pointer, High Byte
mov RESERVED0093, R3 ; RESERVED
dec RESERVED0093 ; RESERVED
movx @DPTR, A

code_1CC:
ret ; End of function code_1BE
```
1. Architecture Discovery

- **Traits of a processor**
  - Pin count
    - Arrangement – ground, UART
  - Memory segmentation
  - Interrupt vector arrangement
  - Peripherals
    - Battery
    - Clock generator
    - UART
    - etc
1. Architecture Discovery

- **Identifying Processor**
  - Cross reference known data points
  - Create subset of candidate processors
  - Find data that excludes
    - Illegal instructions
    - Interrupt vector table in correct
  - Run through disassemble
    - Check for valid code flow

- Identified IPPower as Intel 8051-C509
2. Buffer Overflows in Embedded Devices

Overflow Mechanics

• Von Neumann Machine
  • Stack grows “down”
  • Writes go “up”
  • Unified memory
2. Buffer Overflows in Embedded Devices

Overflow Mechanics

- Frames
  - Each function instance has frame
  - Push
    - Parameters
    - Return pointer
  - Allocate local variable space

[Diagram showing stack frames and memory allocation]

Previous Frames
Parameter 2
Parameter 1
Return Pointer
Frame Pointer
Local Variable
Local Variable
Local Variable

High Memory
Bottom of Stack

Low Memory
Top of Stack
2. Buffer Overflows in Embedded Devices

Overflow Mechanics

func() calls strcpy()

strcpy(dst, src);
2. Buffer Overflows in Embedded Devices

Harvard Overflow Mechanics

- Program Memory
- Stack Memory
- Unused Space
- Local Variable
- Return Pointer
- Parameter 1
- Parameter 2
- Previous Frames
- High Memory
- Low Memory
2. Buffer Overflows in Embedded Devices

Harvard Overflow Mechanics

func() calls strcpy()

strcpy(dst, src);
2. Buffer Overflows in Embedded Devices

Keil C51 Stack

• Stack is non-reentrant
  • No recursion

• Local variable spaces hard coded
  • May not be contiguous

• Global space “spread around”

• All return pointers at top of stack

• No traditional frame structure
2. Buffer Overflows in Embedded Devices

- **Sample application**
  - Level 0 software manages UART I/O
  - Level 1 software reads/writes from UART
  - Several global buffers for UART
    - Common in embedded systems

- **Overflow**
  - `strcpy()`
    - Copy large local buffer into small global buffer
    - Overflow occurs on a global variable
2. Buffer Overflows in Embedded Devices
2. Buffer Overflows in Embedded Devices

- Targeting `strcpy()` ptr
- Writes to `canary_check()` ptr
  Must be valid
- `strcpy()` will exit to our location
- Other return pointers
- Bad data and device crashes
- Good data and device MAY continue
2. Buffer Overflows in Embedded Devices

**Compile Time Defenses**

- **Buffer Canaries**
  - Null byte at end of every local buffer
  - \( \text{str}[i] = 9 \)
    - Translated into a complete function call
    - Performs write if NULL byte intact, returns if not

- **Canary in global buffers**
  - Error in compiler generated memory address
    - Checks local source instead of global destination
2. Buffer Overflows in Embedded Devices

Compile Time Defenses

Translates to:

LOAD addresses into R1, R2, R3
CALL SOME_ADDR

...  

CJNE R3, #01H, 06H
MOV 82H, R1
MOV 83H, R2
MOVX @DPTR, A
RET
JNC 02H
MOV @R1, A
RET

• Will write to NULL byte
• Check is made at NULL location
• Addresses are static in prologue
2. Buffer Overflows in Embedded Devices

**Buffer Overflow Exploit**

- We gained control of execution
- Device most likely crashes after our code
- No way to “push stack up” with overflow
- If previous return pointers aren’t lost
  - We still lost first return point
  - Can return to next good point and it might survive
2. Buffer Overflows in Embedded Devices

Post Overflow Crash

- All state information is most likely lost
- Variables changed via overflow
  - Most likely reset to good values
- Poor programming
  - May have assumed startup == power failure
  - May not initialize data and assume to be zero
    - Additional attack vectors here
- Keil C51 wipes RAM space
  - IPPower 9258 wipes ROM space as well
2. Buffer Overflows in Embedded Devices

Post Overflow

- Redirecting execution can be easy

- Target functions
  - Single functions that complete desired tasks
  - VERY RARE
  - Functions exit and control is lost

- Real goal is to exploit device with a payload
3. Payloads in Embedded Systems

- **Payloads**
  - **Von Neumann**
    - Shellcode in buffer
    - Old rules still apply
  - **Harvard**
    - No code in buffer
    - Only access functionality already present
  - **Harvard with non-reentrant stack**
    - Complicate calls to functionality
3. Payloads in Embedded Systems

- **Payload in Von Neumann**
  - Overflow return pointer on stack
    - Return to location in buffer
  - Execute on the stack
  - Allows injection of any functionality
  - Few embedded architectures support
    - NX bit
    - Memory segmentation controls
    - Memory permissions
    - Permissions rings
  - If supported, not used
3. Payloads in Embedded Systems

- Harvard Architecture
  - No execution of RAM area
  - Strictly limited to functionality already present

- Crash problem is exacerbated
  - RAM is probably initialized to zero
  - CPU ROM “can not be written to”
    - Not true
  - One-shot functions appear to be only option
3. Payloads in Embedded Systems

Harvard

• One-shot functions
  • After function finishes device crashes/resets

• Some useful one shot functions
  • Function that tells HAL9000 to die
  • Password updates
  • Time updates
  • IP update
  • Self update / boot loader

• Coordinated crash could be devastating
  • Crash may not reset the CPU
3. Payloads in Embedded Systems

- **Self update**
  - Most devices have the ability to self update
  - Patches, fixes, new OS’s
  - These routines are in the code
  - Harvard class
    - CPU instructions to modify data space
    - Somewhat violates the Harvard architecture
  - Boot loaders bring in new code
    - Von Neumann machines shift execution
      - Faster DRAM
3. Payloads in Embedded Systems

- payloads in harvard

- return-to-libc style attack
  - chain functions together
  - pass parameters

- allows construction of “functionality”

- previous research

  - write to flash space
    1. prep registers
    2. turn on writes
    3. perform write
    4. reset device
3. Payloads in Embedded Systems

- Payloads in Keil C51 Harvard
  - No frame control
    - Pre computed stack disallows cleanly chained functions
    - Parameters are hard coded addresses
    - Only return addresses can be popped in succession
  - Have to chain legitimate calls
    - Get data into static locations
    - Relying on stack space reuse
  - May be able to exploit nature of pre-computed stack
3. Payloads in Embedded Systems

- Sample application
  - Same as before

- **Overflow**
  - recv()-like function
    - Streams data from input source to buffer
    - Overflow occurs on a local variable
      - Heavily obstructed access to return pointers
      - Dealing with canary
3. Payloads in Embedded Systems

- **Goal of Exploit**
  1. Gain access to return pointers
  2. Redirect to useful functionality
     - The boot loader / update routine
  3. Control parameters
  4. Write byte to ROM memory
  5. Repeat
     - Create a loop that allows continuous writes to ROM
3. Payloads in Embedded Systems

**Targeting Update**

- Locate routine to write to ROM
- Registers R6, R7 control address
- Register R5 controls content
- Find functionality that sets R5 – R6
  - Then returns
  - Or makes call to that location
3. Payloads in Embedded Systems

- **Targeting Update**
  
  - Launch vulnerable function after each write
  
  - Loop on vulnerable function
  
  - Stream bytes to any location
    - Cannot modify core functionality
    - Interrupt table must stay intact
      - Mostly
3. Payloads in Embedded Systems

- **Modify Variables**
  - char buff[32];
  - Int i = 0;
  - while(get_char(&buff[i])) {
    - i++;
  }
  - Manipulate pointer and incremented integer
    - Gain access to other memory (RAM space)
    - Possibly all memory!
  - Allows attacker to place data in static locations
    - Which he/she knows because they are hard coded
  - Exploit integer overflow to reach “lower” memory
3. Payloads in Embedded Systems

**Results of Exploit**
- Write bytes to any ROM location of our choosing
- Can continuously stream bytes
- Can “infect” a device, or destroy it
  - May not be able to repair device
  - Destroy interrupt vector table and device will not boot
- Added routine in free space
  - Change interrupt vector to new routine (possibly reset vector)

**Challenges**
- Cannot change any code that is required to perform writes
- Most Interrupts must continue to function
- Watchdog will reset the device if we write out of bounds
  - Attacker may be able to disable watchdog (there is code to do this all over)
3. Payloads in Embedded Systems

- Impacts of Exploit
  - Self propagation of malicious code
  - Modify constants in a running program
  - Modify running program of an embedded device
    - Inject wholly new functionality
  - Can “infect” a device, or destroy it
    - Add routines that corrupt data at a certain time
    - Destroy interrupt vector table and device will not boot
  - Change interrupt vector table
    - Divide by zero causes device to unload a payload rather than reset
  - All of the dangers posed by root kits in I.T. systems transfer
4. Defenses

- **Common Techniques**
  - **Canaries**
    - Static canaries don’t work
    - Randomized canaries
  - **Stack randomization**
    - Difficult in embedded systems
    - Impossible for pre-computed stack
  - **Static checks**
    - Compiler knows the bounds at compile time
    - Check with static addresses rather than canary
    - Can make writes slow (even slower than Keil C51 method)
Conclusions and Future Work

• Obfuscating hardware does not work
  • Given enough time and energy attackers will figure it out
  • Xbox, Ipod, Iphone, PSP, Satellite receivers, etc…

• The buffer overflow is still a threat
  • Impact is high
  • Exploitability is high
  • Target rich environment

• Mitigate code injection threat
  • Require physical interaction to modify ROM space (switch, button, etc…)

• Do we defend the processor or the code?
  • Static code analysis is cheap and deployable on current hardware
  • Run time defense (canary, coprocessor, anomaly detection, etc) can be slow
  • Permission systems may not be supported by processor
<table>
<thead>
<tr>
<th>Name</th>
<th>Email</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kristopher Watts</td>
<td><a href="mailto:Kristopher.watts@gmail.com">Kristopher.watts@gmail.com</a></td>
<td>(208) 557-8332</td>
</tr>
<tr>
<td>Dr. Paul Oman</td>
<td><a href="mailto:Oman@uidaho.edu">Oman@uidaho.edu</a></td>
<td>(208) 885-6899</td>
</tr>
</tbody>
</table>