Travis Goodspeed

Nifty Tricks and Sage Advice for Shellcode on Embedded Systems
MAKE YOUR AVATAR HERE!!!

TRAVIS'S AVATAR
Button Matrix Encoding KH-930 Knitting Machine

On the board rows are in 4 pin ribbon cable, on mainboard denoted on P8 as 1, 2, 3, 4.

On the board columns are diodes D2 - D11 on mainboard, and pins 7-16 on large ribbon cable between boards.

<table>
<thead>
<tr>
<th>rows</th>
<th>columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>D2</td>
</tr>
<tr>
<td>1</td>
<td>D3</td>
</tr>
<tr>
<td>2</td>
<td>D4</td>
</tr>
<tr>
<td>3</td>
<td>D5</td>
</tr>
<tr>
<td></td>
<td>D6</td>
</tr>
<tr>
<td></td>
<td>D7</td>
</tr>
<tr>
<td></td>
<td>D8</td>
</tr>
<tr>
<td></td>
<td>D9</td>
</tr>
<tr>
<td></td>
<td>D10</td>
</tr>
<tr>
<td></td>
<td>D11</td>
</tr>
</tbody>
</table>

Unused: R3D7
Column pulling resistors under ribbon cable.
Pull up resistors...
Travis Goodspeed

Nifty Tricks and Sage Advice for Shellcode on Embedded Systems
Thank you Kindly

- Aurelien Francillon
  - ``Half-Blind Attacks: Mask ROM Bootloaders are Dangerous''
- Sergey Bratus
Let's Exploit Something Small

- 8, 16, and (low end) 32-bit microcontrollers
- No operating system, maybe a libc.
- Defensive features are an accident,
  - No ASLR, but still unknown code.
  - No NX-bit, but often Harvard architectures.
  - Lots of weird registers, custom code.
Rogue's Gallery

- 8051
  - More popular than X86, AMD64 and ARM.
  - Harvard Architecture
  - Instructions are byte-aligned.
  - Rarely able to execute RAM.
  - Thousands of different clones.
Rogue's Gallery

- MSP430
  - 16-bit Von Neumann
  - Most, but not all, versions can execute RAM.
  - 1kB Mask ROM Bootloader (BSL)
  - 16-bit aligned instructions, almost PDP11.
  - Used in the GoodFET, Facedancer, SPOT Connect, Metawatch, and other devices.
Rogue's Gallery

• AVR – 8-bit Harvard
• PIC – 8-bit Harvard
  – Some have hardware call stack.
• HCS08, 6502, 6805, etc.
  – Every old architecture is still around someplace.
Goals

• On a PC, we want code execution.
  – Load malware, drop a shell.
  – Hack the Gibson!

• On an MCU, we want code!
  – Exploits often used to dump firmware.
  – A PEEK primitive is as good as code execution.

• Strange exploit uses:
  – Stack smashing for temporary patches.
  – Upgrades of unpatchable firmware.
Exploiting the 8051

- 8-bit CPU, Harvard Architecture
- RAM is rarely executable.
- Dozens of clones, none of them the same.
8051 Memory Spaces

- No such thing as “just a pointer.”
- Call stack is hardware limited, sometimes two stacks.
- Different opcodes access different memories.
  - CODE – 64 kB, Mostly Flash, with a bit of ROM.
  - DATA – 256 bytes for variables and stack.
  - IO – Overlaps DATA, for Special Function Registers.
  - XDATA – 64kB of extended RAM.
- This architecture is everywhere.
8051 Exploitation Headaches: Executing RAM

- Class 8051 doesn't allow execution of RAM.
  - CODE and XDATA don't overlap.
- Modern chips have exceptions, but they're complicated.
  - Chips with little memory just unify the address space. \&CODE==\&XDATA
  - Chips with lots of memory map to different locations, small region of overlap.
8051 Exploitation Headaches: Writing to Flash

- Writing to Flash is tricky.
  - There is no standard instruction for writing Flash.
  - You could use multiple calls to a POKE primitive, and a good knowledge of the clocks, and you need to do this reliably in a loop, and you need to do it without native shellcode.

- There are options.
  - Varies by architecture.
  - Generally, you abuse the self-reprogramming feature.
8051 Exploitation Headaches: Writing to Flash

- 8051 was Harvard until self-reprogramming was a needed feature. Things change.
- The issue is that you can't read or execute from Flash while writing to Flash.
- Three solutions:
  - Map RAM into both XDATA and CODE memories.
  - Flash reads a JMP $-1 when busy.
  - Mask ROM contains code to copy XDATA to CODE. (RAM to Flash)
8051 Exploitation Headaches: Writing to Flash

- Map RAM into both XDATA and CODE memories.
  - Just force a return into it. 1996-style exploits work!
- Flash reads a JMP $-1 when busy.
  - Much harder, especially if there's no gadget to write to flash.
  - Sometimes you can use a POKE primitive.

- Mask ROM contains code to copy XDATA to CODE.
  - Nice and easy to exploit.
  - Calling convention is often documented!
Example: GPIO Blinking

- Vuln was in a USB bootloader.
- Exploit was supposed to dump Flash and RAM.
- USB buffer is preciously small
  - Our first-stage shellcode needs to be tiny.
  - We could call the USB stack, but it's complicated.
  - We only need to exfiltrate data.
  - Let's use the LEDs!
Example: GPIO Blinking

- A tiny standalone application:
  - 1. Setup the GPIO pin directions to output.
  - 2. Blink half of them with a clock.
  - 3. Blink the other half with data bits.
  - 4. Sniff pins with a logic analyzer to get the bits.

- As shellcode,
  - 1. The GPIO pins for LEDs are already directed out.
  - 2. while(1) and let God sort it out.
Example: GPIO Blinking

- Clock LEDs look solid.
- Data LEDs blink irregularly.
- Tap one of each into a logic analyzer.
Return to Libc

• Complicated by a lack of Libc
  – It's there, but statically linked and pruned.
  – Nothing like system() or exec().

• If our goal is to get the Flash, we can't know what's where in Flash.

• Two tricks:
  – Return to the bootloader with privilege escalation.
  – Privilege escalation gadget can be found blind!
Example: Returning to a Bootloader

- Many chips have a bootloader in Mask ROM.
  - This is permanently a part of the chip.
  - This cannot be patched or removed affordably.
- This ROM is an excellent return-to-libc target.
  - Always at a fixed position.
  - Very few revisions to reverse engineer.
  - Rather small.
  - Includes at least one command shell.
**Example: Returning to a Bootloader**

- MSP430 Bootloader
  - 0x0C00 to 0x0FFF, just 1 kB
  - Requires the Interrupt Table as a password.
  - R11 is a global containing the password status.
- Return-to-BSL Shellcode in Six Bytes
  - MOV 0xFFFF, R11; Pretend we gave a good pass.
  - CALL 0x0C0A; Enter a bit late to not clear R11.
Example: Blind Return-Oriented Programming

- What if we couldn't execute shellcode from RAM?
  - Some security-enhanced variants disallow RAM exec.
  - Competing processors (AVR, 8051) are Harvard.
- We could build a ROP chain
  - ROM doesn't contain enough gadgets.
  - We don't know where anything is in Flash.
  - Let's build it blind!
Example: Blind Return-Oriented Programming

• Suppose the following
  – We have a stack-buffer overflow bug.
  – We have a copy of ROM, but not of Flash.
  – We cannot execute RAM.

• Plan of attack,
  – Use ROM entry point to find return address offset.
  – Scan for RET statements in Flash by crashes.
  – Try each gadget in turn.
How the hell does this work!? 

- The gadget we need is rather common, rather small.
- We have a very small address space.
- We're not trying to be Turing Complete.
- We have a feedback mechanism,
  - Crash indicates the stack is mis-constructed.
  - No crash indicates we're getting some gadget.
  - Side effects tell us which gadget.
Example: Blind Return-Oriented Programming

- 1. Fuzzing gives us a stack buffer overflow.
- 2. Varying our offset verifies our control of the Program Counter by a successful jump into ROM.
Example: Blind Return-Oriented Programming

- Now we control the PC, but we don't know the password. We need a ROP gadget like `"POP R11"`, which is common in function epilogues.

- 3. Move the BSL entry one word up in memory, with a random address in its place.
  - If this enters the bootloader, we might have found a "RET" instruction.
  - If it doesn't, we've found a gadget of some sort.
Example: Blind Return-Oriented Programming

- Now we have some gadgets, but we don't know what they do.
  - 59 valid gadget entry points in my target.
  - 1/50 to 1/150 gadgets/addresses in other samples.
  - Varies drastically by architecture and compiler.

- 4. Try all gadget addresses with the appropriate stack layout. Bootloader pops open!
Example: Blind Return-Oriented Programming

- Final call stack, higher addresses at the top.
  - 0x0C0E – Bootloader entry, called last.
  - 0xFFFF – Value to pop into R11 by our gadget.
  - 0x???? – Address of a ``POP R11" gadget.
- Unknown address doesn't have many candidates,
  - Must be at an even address before a RET.
  - ~8,000 possibilities in address space, easy to search.
  - ~59 possibilities before RET, easier to search.
  - Two gadgets, 59**2 or ~4,000 tries.
  - Three gadgets, 59**3 or ~200,000 tries.
RAM Patching

• On higher-end chips, you patch RAM.
  – Many faster chips can't execute Flash directly.
  – RAM patches are less likely to brick the target.
  – Very useful for backdoor development.

• But RAM gets overwritten.
  – You'll need to hook functions that overwrite the IVT.
  – It works pretty much like a DOS TSR.
Flash Patching

• Suppose you can overwrite Flash, but you can't erase it.
  – Common when patching the IVT directly.

• NOR Flash isn't like RAM.
  – You can clear bits individually, but only set them as a page.
  – Overwrites are a bitwise AND.
Flash Overwrites

- 0xFFFF at erasure
- 0xDEAD written.
- ~0xDEAD cleared.
- 0xDEAD remains.

- 0xDEAD at start.
- 0xFF00 written.
- ~0xFF00 (0x00FF) cleared.
- 0xDE00 remains.
Flash Patching

- Given only a POKE primitive, you can more easily clear bits than set them.
  - Page writes are complicated.
  - Might break code that's needed to boot or to POKE.
- What tricks can help us choose the right bits to clear?
**Flash Patching**

- On the MSP430, RAM is beneath Flash.
  - By clearing significant bits, you can redirect a CALL to a target in RAM.
  - CALL 0xBEEF; Call to function in Flash.
  - CALL 0x02EF; Call to function in RAM.

- On 8051, 0x00 is a NOP.
  - By clearing bytes, you can NOP-out code.
  - Opcode table is conveniently arranged by bytes.
Parting Thoughts