5 Some Shellcode Tips for MSP430 and Related MCUs

by Travis Goodspeed

Howdy y’all,

I’m writing this to introduce you as an exploiter of desktops and servers to some of the tricks that I’ve used in writing shellcode for microcontrollers, with examples from the MSP430 in particular. You can try most of these examples on a GoodFET or Facedancer board, and many of them are portable to other embedded targets, such as AVR or the lower-end ARM devices.

5.1 Flash Patching is Weird

In Unix and Windows, you are used to processes operating within virtual memory. On a microcontroller, they often run directly in physical memory, so the rules are rather different. It helps to take the German approach, learning all of the rules to get away with things that ought to be illegal.

The first difference you’ll run into on the MSP430 is that code runs in-place from Flash memory. Flash has some very different rules from RAM, because it’s a different technology and a proper programmer knows better than to rely on layers of abstraction.

- Flash is erased to ones as segments or globally, never as bytes or words.
- Flash writes clear bits at word granularity, but can’t set them.
- Flash writes require a safety password to be written into a register.

Thus, to do a normal write to Flash, an MCU programmer is taught to first disable the Flash write protection and configure the right special-function registers, then erase the entire page, then rewrite the entire page. Many programmers never bother, opting for an external memory chip or relying on battery-backed RAM.

To make smaller changes, there’s another option. After disabling Flash, a neighbor could clear individual bits rather than rewriting the entire page. This is handy for regular developers to do what’s called EEPROM Emulation, which emulates memory that can be written bytewise, but it’s also damned useful when patching code in-place.

For example, Figures 1 and 2 show that 0x3Cxx is an unconditional Jump while 0x38xx is a conditional Jump if Less Than instruction. If we overwrite a JMP instruction with 0x3BFF, it will have the effect of bitwise ANDing that instruction with 0x3BFF, changing the 3C opcode to a 38 while retaining the jump offset.

Figure 1: MSP430 Instruction Set, from the MSP430X2xx Family User’s Guide
Since MSP430 instructions are 16-bit word aligned, the 10-bit PC offset is multiplied by two and then added to the program counter. 0x3FFF is an unconditional jump backward by one word, or an unconditional infinite while loop. If you zero-out the offset by overwriting the instruction with 0x3C00, you can turn any jump instruction into a NOP.

When attacking a poorly protected bootloader, you might find yourself with the ability to write and to checksum, but not to read. If you can write without erasing, then writing all 1’s with a single 0 will change the checksum if and only if that bit previously was a 1. Repeating for each bit of Flash is slow, but it might get you a firmware dump.

### 5.2 Efficient Shellcode

Quite often, the first thing you’ll do with shellcode is to dump out the state of the microcontroller being attacked. It’s worth studying ways to make that code in as few bytes as possible, as a microcontroller generally processes very small packets and you won’t have room for anything fancy.

To quickly dump memory on an architecture that you don’t know very well, it helps to have simple code that already has its environment configured. The code should be completely oblivious to timing, and it should access as few structures as possible. It should also be portable, requiring neither knowledge of its position in memory nor knowledge of the specifics of the rest of the device motherboard at compile time.

My solution is to blink the LEDs, half with a clock and half with data, to dump all of the memory to an SPI sniffer. The LEDs that light up with consistent brightness are the clock, while those that sporadically become very bright or very dim are the data. Tapping one of each with my handy Saleae Logic analyzer gives me a firmware dump.

### 5.3 Mask ROMs have Useful Gadgets

In my WOOT ’09 paper with Aurélien Francillon, we toyed around with using the MSP430’s BSL (BootStrap Loader) ROM to aid in exploiting an unknown executable. That paper concerns exploiting firmware without having a copy, but I’ll recount one of its tricks here.

The MSP430 BSL has two entry points. The first is the Hard Entry Point, whose address is always stored at 0x0C00. By twiddling the reset and test pins with proper timing, the chip will boot from this address instead of from the RESET handler in the interrupt table.

The second entry point is called the Soft Entry Point, and it is rather poorly documented. The original idea was that a program could return into the bootloader ROM by branching to the address stored at 0x0C02, with some of the initialization routines skipped. One of these routines is the instruction that initializes the register holding password protection, so by setting or clearing a bit in that register, the calling application can enable or disable password checking.

While the soft entry point is sometimes useful to an MSP430 developer, it’s damned useful for an attacker. On an MSP430F1612, my favorite shellcode for dumping firmware is a bit like the following, which assembles to just six bytes of memory.

```asm
mov #0xFFFF, r11 ;; Disable BSL password protection.
br $0x0C02       ;; Branch to the BSL Soft Entry Point
```

---

6 Half-Blind Attacks: Mask ROM Bootloaders are Dangerous, WOOT 2011, Goodspeed and Francillon
5.4 Unused RAM is Not Erased at Reboot

In larger machines, memory which is not used by a process is not mapped into that process’s virtual memory. In microcontrollers, it is still accessible, since the code is running with physical rather than virtual memory. Rather than reset every RAM word during a reboot, most microcontrollers simply leave it alone and let the program take care of clearing its values.

Now an MSP430 application is compiled with a view of memory that it sparingly uses. GCC, for example, will allocate code (.text) into Flash from the lowest Flash address in its linker script.

RAM is only used by the compiler for data, never for code, unless the linker script is carefully and intentionally hand-crafted. It is divided into two segments by the linker, .data and .bss. The .data region is initialized by copying the data over from Flash, while the .bss region is initialized to zero through a simple while() loop. This provides us with two nifty tricks.

The first trick is that, given a poor POKE gadget, we can slowly place a large chunk of shellcode into upper regions of RAM. For example, an MSP430F2618 has enough RAM to fit the GoodFET firmware, so a device using that chip could have the GoodFET firmware itself act as second-stage shellcode! Smaller chips, such as the MSP430F2274, could have a Flash driver loaded into unused RAM, with third-stage shellcode written into unused Flash.

5.5 Where Flash is Protected, RAM is Not

Recalling that unused RAM is never cleared by an application, let’s abuse that behavior in a second way.

Back in 2010, Texas Instruments released their ZStack implementation of Zigbee for use with the Smart Energy Profile. I found that the random number generator was crap, and they patched that bug. So how was little ol’ me supposed to get more Zigbee Smart Energy Profile keys without a Certicom license?

The remaining vulnerability was a combination of the BSL ROM with the ZStack firmware. ZStack relied upon the BSL ROM and the JTAG fuses to prevent keys and firmware from being read out of the device, but the BSL ROM was only intended to keep code from being read out of the device. A second bug in that Zigbee stack was that keys were stored in the .data segment instead of the .text segment, so the firmware would copy the key from Flash into RAM during startup.

As a quick recap, the bootloader requires a password to run most commands, but some are unprotected. Among them are the ones to supply a password and the Mass Erase command, which wipes all of Flash and resets the password, which is stored in Flash, to 32 bytes of 0xFF.

So to get keys out of locked ZStack devices, I just needed to use the serial bootloader, first sending the command to Mass Erase and then—without losing power—to supply a password of all 0xFF and then to dump all of RAM to disk. A little bit of RAM is overwritten by the BSL’s call stack, but only the lowest 32 bytes. Everything else is saved.

I hope you find these tricks to be handy. If you’d like to hear more, buy me a nice India Pale Ale.

— Travis
Who would remember Noah if he had just bought a boat from the store?
Build your own fucking birdfeeder.