Half-Blind Attacks: Mask ROM Bootloaders are Dangerous

Travis Goodspeed
travis@radiantmachines.com

Aurélien Francillon
INRIA Rhône-Alpes
aurelien.francillon@inrialpes.fr

Abstract

This paper presents a non-invasive, half-blind firmware extraction technique that subverts a mask ROM bootloader in order to recover the firmware of a microcontroller. A stack based buffer overflow is used to forcibly enter the bootloader. Further, a practical method of blind return-oriented programming is presented in which a gadget's entry point is brute forced, being unknown *a priori*. In this paper we show that when a software vulnerability has been found (e.g. by fuzzing), the attacker can locate by brute force the sequences of instructions, *gadgets*, required for bypassing the protections present in a bootloader. In a half-blind attack, the presence of a bootloader in mask ROM helps the attacker in that, while he must still discover blindly a vulnerability in unknown firmware and the appropriate gadgets, he knows the exact contents of the bootloader.

1 Introduction

Microcontrollers are used in a broad range of applications, from hobbyists projects and consumer devices to medical and industrial equipment. However, in contrast to a hobby project's open source code, the industrial world often relies on the secrecy of the firmware to prevent device cloning or re-purposing by competitors. Such confidentiality is often enforced by a bootstrap loader. For example, the Advanced Metering Infrastructure (AMI) integrates microcontrollers into each electric power meter of a city, and many smart cards rely upon a microcontroller to restrict access to cryptography. Most manufacturers rely on the possibility of updating the firmware on these devices as a way of fixing bugs without replacing physical components. On the other hand, some users would like to change the configuration of those devices, such as to commit theft of services by altering their own electricity billing data or extracting the keys to decrypt subscription television channels. Competitors with easy access to the firmware of a device would be able to clone it and provide counterfeit parts. It is therefore often essential that the firmware remain secret.

As a protection against reverse engineering and tampering, it is often desirable that a device accept firmware updates but not reveal firmware to an attacker. Bootloaders, which allow updates to be performed, include protective measures to prevent unauthorized modifications. While the simplest of these will be a password, our technique is sufficiently generic that it will also apply to a bootloader which uses symmetric or asymmetric cryptography.

In this paper we show how those bootloaders can be subverted by an attacker. Previous techniques for recovering firmware in microcontrollers were either reliant on a specific vulnerability of either the bootloader (password comparison in variable time, weak password [5, 2]) or the hardware (fault injection, micro-probing[13]). However, if software vulnerabilities are present in the application firmware, fuzzing can be used to find them. Then, by brute force, the locations of adequate sequence of instructions, *gadgets*, can be found that enable the attacker to bypass the protections present in the bootloader. In this case, the very existence of the bootloader contributes to the vulnerability, even when the bootloader itself has no exploitable bugs.

We implemented this attack on the MSP430 microcontroller, a widespread microcontroller having a bootloader in mask ROM. For this purpose we used a demo application in which we inserted a stack based buffer overflow vulnerability. We performed the fuzzing blindly, without using a debugger, disassembler, or similar tools.

As the bootloader code is shared among all chips of the same model and version, it is easy enough to purchase a chip and read its bootloader. We assume the attacker to have intimate knowledge of it, both as data and as annotated disassembly.

Once the attacker has discovered a vulnerability in

the application, he can brute-force the offset of a return pointer, having it point to known instructions in the bootloader. If a gadget outside of the bootloader is required, its location may be brute forced, as we demonstrate in Section 5.1. It is for this reason that we call our attack half-blind.

The result of this research is an attack which is extremely fast and low cost when compared to physical tampering. As the attack uses a vulnerability in application code to launch into the bootloader, the mere presence of a bootloader–even one which has no exploitable vulnerabilities of its own–is dangerous to the security of a device. Further, probabilistic defenses, such as a suicide triggered by failed exploit attempt, are of no use when the attacker has access to many potential victims.

2 Texas Instruments MSP430

The Texas Instruments MSP430 low-power microcontroller is used in many medical, industrial, and consumer devices. It may be programmed by JTAG or a serial bootstrap loader (BSL) which resides in mask ROM [10].

The JTAG test access is often deactivated before the device is shipped to customers, preventing direct recovery of the firmware. Recent versions of the BSL may be disabled by setting a value in flash memory. When enabled, the BSL is protected by a 32-byte password. If these access controls are circumvented, a device's firmware may be extracted or replaced.

2.1 Serial Bootstrap Loader (BSL)

The BSL of the MSP430 resides in mask ROM between addresses `0x0c00` and `0x1000`. As the BSL continues to function after the JTAG fuse has been blown, it is often used to allow for write-only updates without exposing internal memory to a casual attacker. For the same reason, it is a valuable attack vector. Each firmware image contains a password, and without that password little more is allowed by the BSL than erasing all of memory. While some versions of the BSL are vulnerable to a timing attack [7, 6], version 1.61 as found within the MSP430F1611 is invulnerable to timing attacks [1]. Further, Becher et al. demonstrated in [2] that brute forcing of the password–in the absence of a timing attack–is quite impractical. As our attack demonstration is against this version, all pointers and similar details refer to this version.

The BSL has two entry points. The first, a hard entry point, may be started by use of a test pin reset sequence. The second, a soft entry point, exists to allow an application to cede control to the BSL [10]. A disassembly of the BSL (Table 1) reveals few instructions between the two. The hard entry point executes only three instructions before continuing at the soft entry point.

Address	Data/Instruction	Comment
0x0c00	`0x0c04`	hard entry addr.
0x0c02	`0x0c0e`	soft entry addr.
hard entry point		
0x0c04	`mov 0x0220, r1`	set stack pointer
0x0c08	`clr r11`	clear r11
0x0c0a	`mov.b 0, 0xf60a`	?
soft entry point		
0x0c0e	...	...

Table 1: BSL Prologue

The first instruction sets the stack pointer, r_1, to address `0x0220`, a valid SRAM position commonly used for the stack. The second instruction clears r_{11}, ensuring that all bits of r_{11} are low. The password checking code sets bit 4 of r_{11} after proper password authentication. Clearing this bit during hard entry to the bootstrap loader ensures that a password must be presented before access.

When a user application calls the soft entry point, these three instructions are not run. The stack pointer is not reset, as it is assumed that the parent application already has a proper stack. Further, as r_{11} is not cleared, the BSL will grant the user administrative privileges if the relevant bit happens to be high.

In the rest of this paper we demonstrate that by using a vulnerability in the application the attacker can force the entry to the bootloader with administrative access. This is done by calling a gadget that pops a word from the stack into r_{11}.

2.2 Calling Convention

The MSP430 is a RISC processor with 16 general-purpose registers that are not memory-mapped. r_0 is reserved as the program counter. r_1 is the stack pointer. r_2 and r_3 are used as constant generators; further, r_2 is also used as a status register. r_{12}, r_{13}, r_{14}, and r_{15} are used for function parameters and a return value; they are also the only clobber registers.

As the lower registers must be preserved between function calls, it is common for a function to pop a saved value from the stack immediately before returning. GCC, for example, will often begin a function with "push r11; push r10; push r9; push r8; ..." and end the same function with "...; pop r8; pop r9; pop r10; pop r11;". This fact will become important in Section 5.

[1] All BSL code fragments quoted within this document are of that version.

3 Stack Based Buffer Overflow on Embedded Systems

A stack based buffer overflow involves the abuse of a string copy or a buffer copy to overwrite the return pointer of a function. Stack based buffer overflows were first demonstrated for the MSP430 in [4]. By overwriting the return pointer with an address within an incoming packet, an attacker can execute arbitrary code.

By overwriting the return pointer with the BSL's soft entry point, an attacker can enter the BSL without first clearing r_{11}. Further, as the BSL is in unchanging mask ROM, the attacker can be confident of its entry addresses.

Variants of stack based buffer overflows have been proposed that calls full function or call sequences of instructions [14, 9]. A more sophisticated technique is known as return-oriented programming. Here, the attacker constructs a stack which returns from one function suffix, or *gadget*, into another. This was first demonstrated for the X86 architecture in [11], then extended to Harvard-architecture embedded systems in [3].

4 Simple BSL Forced Entry

The simplest of our attacks involves the injection of the soft entry address, `0x0c0e`, into the program counter r_0 by overwriting the return pointer of the stack. This provides access to the BSL without first clearing r_{11}, such that if bit 4 of r_{11} is set, access is granted to the BSL as if a valid password had been entered. The TX Data Block command of the BSL may then be used to transmit the password to the attacker, providing for future access.

In the case that the relevant bit of r_{11} is clear, it is necessary to use return-oriented programming to set it. This technique will be described in Section 5.

4.1 Locating Return Pointer Offset

Further, it was found that the technique of fuzzing allowed for a proper injection even when the stack pointer depth was unknown. As shown in Figure 1, the return pointer was surrounded by the byte `0xff`. In the event that the offset is guessed correctly, `0x0c0e` falls into the program counter r_0 and BSL execution begins. When guessed incorrectly, r_0 is set to `0xffff`, `0x0eff`, or `0xff0c`. Of these, `0xffff` and `0xff0c` reset the device, which then waits to receive another packet. `0x0eff`, being odd, also causes the chip to reset. Therefore, an attacker who guesses at the packet offset through fuzz testing can be assured that the victim device will continue to receive packets until the exploit lands.

When the return pointer is correctly overwritten, the attacker will begin to see replies to BSL serial commands

Attempt	Payload							PC
1	0x0E	0x0C	0xFF	0xFF	0xFF	0xFF	0xFF	0xFFFF
2	0xFF	0x0E	0x0C	0xFF	0xFF	0xFF	0xFF	0xFF0C
3	0xFF	0xFF	0x0E	0x0C	0xFF	0xFF	0xFF	0x0C0E
4	0xFF	0xFF	0xFF	0x0E	0x0C	0xFF	0xFF	0x0EFF
5	0xFF	0xFF	0xFF	0xFF	0x0E	0x0C	0xFF	0xFFFF
6	0xFF	0xFF	0xFF	0xFF	0xFF	0x0E	0x0C	0xFFFF

Figure 1: Packet payloads when fuzzing.

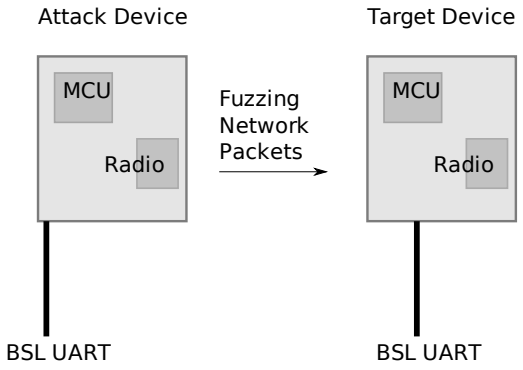

Figure 2: Experimental setup

that are unique to the BSL application. If bit 4 of r_{11} is set, he will also have direct access to protected commands through this port (which enables him to recover the firmware from the serial port using BSL features). If the bit is clear, he will only have access to unprotected commands, but he can elevate his privileges by the technique described in the next section.

5 Using a Gadget to Set R11

As the technique of the Section 4 is insufficient when bit 4 of r_{11} is clear, it will sometimes be necessary to set this register by use of a gadget. Lucky for the attacker, the relevant gadget is quite common. Because r_{11} is the highest register which must be restored before a return, it is quite common for GCC generated functions to end with "pop r11; ret;".

5.1 Blind Gadget Chain Injection

Searching our test application for this gadget yields 59 valid entry points between `0x4000` and `0x7000` in the victim application. Guessing an even address in this range at random yields success with a probability just beneath 1%, as

$$\frac{2 \times 59}{3 \times 16^3} = \frac{1}{104}$$

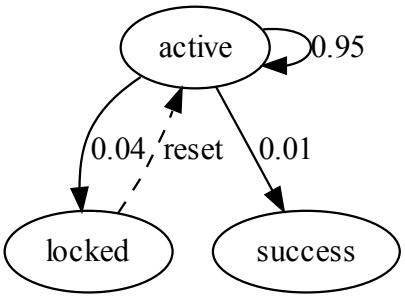

Figure 3: Observed Injection Markov Chain

Thus, even if an attacker does not have access to a copy of the victim firmware, he can guess the gadget entry point at random, succeeding in a reasonably small number of attempts. Similar frequencies are found in other TinyOS applications: $1/93$ in BaseStation, $1/86$ in MultihopOscilloscopeLqi, and $1/150$ in MViz.

This gadget is unique to applications compiled with GCC, and it is often not found within applications from other compilers. While this particular gadget is unavailable, others may be used instead. For example, the gadget "pop r11; pop r10; ret;" can commonly be found within firmware images produced by the popular IAR C Compiler.

Figure 3 demonstrates the probabilities of an active node's responses to an injection attempt, the coefficients having been rounded for convenient calculation. Each event is a blind injection attempt, with the new program counter value being the attacker's guess at a gadget. 95 percent of injections result in the node rebooting, resulting in another attempt for the attacker. In the 4 percent of attempts which result in an unrecoverable state, equivalent to `while(1);`, the victim device may be manually reset to return to the active state.

6 Related Microcontrollers

In reviewing some common microcontrollers, we found that many include a mask ROM bootloader, sometimes documented as a *utility ROM*. These include the Dallas Maxim MAXQ3210, Siemens C167, ST Microelectronics STM ST10, and STM8S. We expect that our half-blind attack will work on them.

We also note that the bootstrap loader we analyzed from the MSP430 is an almost completely inlined. It has only three child functions [2], none of which contain the desired code (i.e. "pop r11" in our attack) before their ending "ret" instructions. As a result, the desired gadgets do not exist within the mask ROM bootloader, requiring the use of a half-blind attack.

By contrast, if a mask ROM were to contain a function with the required gadget, the attacker might make use of that gadget for return-oriented programming without resorting to the half-blind technique. The MAXQ3210/12, for instance, has many functions within its utility ROM, fourteen of which are documented for use by the child application [8].

7 Countermeasures

7.1 Using a Flash Based Bootloader

Chips which lack bootloaders in mask ROM may have a bootloader in flash ROM. While the device will be slightly more expensive to manufacture (or have less flash memory available for the application), this gives more flexibility to the developer to include a bootloader with the functionality he needs or not to have a bootloader at all. Moreover, as our attack is outlined, if each application were to come with its own bootloader, an attacker cannot use the previous knowledge of the bootloader. In this case, an attack will become fully blind, reducing its chances of success.

7.2 Program Randomization

It has been shown in [12] that randomizing the address layout is often insufficient to defend against an attacker who has the luxury of being able to attempt multiple injections. This is particularly true in an embedded system, where the memory space is further constrained than that of an X86 host PC. For example, an X86 has 32 bit pointers, while the MSP430 has 15-bit instruction pointers [3]. In any case, as our example assumes no prior knowledge of the contents of memory, it should be clear that further randomization serves no purpose.

7.3 Self Destructing Firmware

As it has been shown that randomization is ineffective because of an attacker's privilege of multiple attempts, a logical defense might be to restrict the number of tries that an attacker has.

First, consider Figure 4 which extends Figure 3 to include a conservative defense. In this example, the device detects a crash and returns to an active state in most cases, with a random suicide that occurs with a

[2] `rxbyte()`, `txbyte()`, and `tarebitwidth()`

[3] Though addressed in 16-bit, all instructions are even aligned.

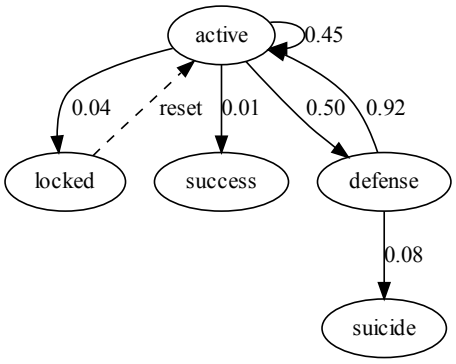

Figure 4: Conservative Defense

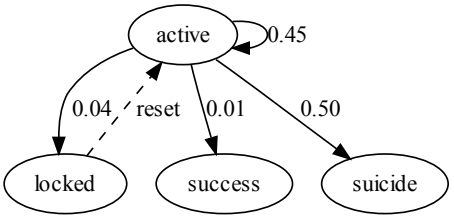

Figure 5: Aggressive Defense

slight probability. This is intended to model the case that an unreliable application wishes to erase its memory in the face of an attack, but not necessarily to kill itself in the case of an isolated, accidental crash. In this case, there is a probability $P[s] = 0.01$ that an attempt will be a success and $P[s'] = 0.04$ that an attempt will result in a suicide. Combining neutral resets to $P[r] = 0.04 + 0.50 * 0.92 + 0.45 = 0.95$, we find that $P[s|\overline{r}] = 0.20$ and $P[s'|\overline{r}] = 0.80$. Therefore, while a suicide s' is more likely than a success s, the difference is rather small, and an attacker need only purchase a few extra units to increase the likelihood of a success. As the probability of all n devices self-destructing in case of an assault would be $P[S'](n) = P[s'|\overline{r}]^n = 0.80^n$, an attacker with thirty units to attack will succeed with a probability of $P[S](30) = 1 - 0.80^{30} = 0.9988$.

Figure 5 presents a less conservative defense, one in which any fatal crash is assumed to be an attack, resulting in suicide. While such a behavior is likely to be unacceptable in many industries, more so with unreliable products, it is well worth considering, especially because

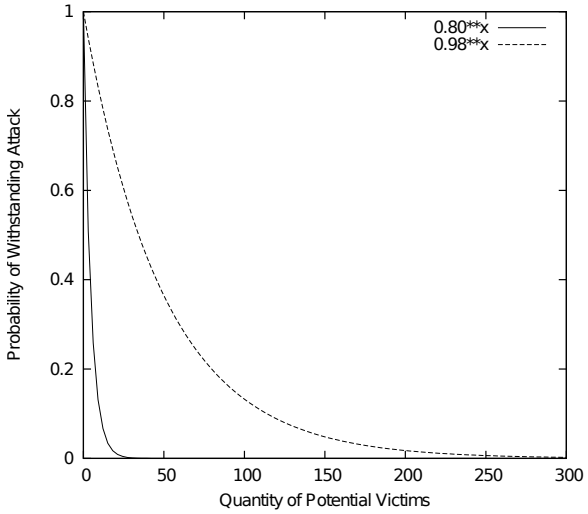

Figure 6: Optimistic Estimate of Defense Probabilities

it does not work. While $P[s]$ is still 0.01, $P[s'] = 0.50$. Thus, rounding a bit, $P[s|\overline{r}] = 0.02$ and $P[s'|\overline{r}] = 0.98$, which implies that $P[S'](n) = 0.98^n$. While the odds are better, an attacker might still feasibly buy enough victim devices as $P[S'](300) < 0.01$. Therefore, a suicide behavior is insufficient to defend a device which is available to an attacker in quantity, even when false positives are not a concern and half of all attempted exploits can be detected.

The calculations of this section are only to be considered a back-of-the-envelope attempt at a worst case for the attacker. Supposing that the probability of any attempt succeeding is independent of the past attempts neglects the likelihood that an attacker will search linearly or in some other manner that eliminate unnecessary guesses. As such, we have shown that probabilistic self defenses are not effective at protecting the contents of firmware from the half-blind attack, even supposing that the attacker guesses gadget addresses with no advanced strategy and that the defender does not ignore detectable overflows. An attacker may need fewer attempts than this, but he will not need more.

7.4 Limiting ROM Code Access

Making the mask ROM code available only during programming or debugging, e.g. when a specific fuse bit is set high, would effectively prevent it from being present in memory during normal operation. Such a mechanism might prevent an attacker from abusing it as we presented here.

An example of such a mechanism is found in some Atmel ARM7 based microcontrollers [1]. Those microcontrollers have a bootloader in mask ROM; however, this

ROM is not mapped to any address space during normal execution. Rather, the code is copied to an executable RAM only when a specific test sequence is applied to the chip.

On the MSP430 a similar mechanism might be implemented as a defense in which the bootloader only becomes executable–or perhaps even mapped–when the BSL entry sequence is performed. Care should be taken to ensure that such a mechanism cannot be called by return oriented programming.

8 Conclusion

We have demonstrated a technique for leveraging the existence of a bootloader ROM to break the firmware confidentiality of a microcontroller by a return-oriented programming attack against a vulnerable, but unknown application. Further, we have demonstrated that a probabilistic defense is not effective when the attacker may purchase multiple victim devices. As our exploit only uses pre-existing code, it is suitable for use on those Harvard-architecture chips which cannot execute RAM.

There are many exciting possibilities for future work. As a gadget can be found in isolation, an attacker can identify the gadgets individually. While our attack only required a single gadget, it should be feasible to construct blindly a library of gadget entry points, then to combine those for more sophisticated return-oriented programming.

Acknowledgments

The authors would like to thank Daniele Perito for his helpful comments and feedback. Lane Westlund is to be thanked for his rewrites of both the MSP430 bootloader and its documentation. We are also grateful for the comments from the anonymous reviewers.

The work presented in this paper was supported in part by the European Commission within the STREP WSN4CIP project. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsement of the WSN4CIP project or the European Commission.

References

[1] ATMEL INC. AT91 ARM Thumb-based microcontrollers. Atmel documentation DOC6175, December 2008.

[2] BECHER, A., BENENSON, Z., AND DORNSEIF, M. Tampering with motes: Real-world physical attacks on wireless sensor networks. In *SPC* (2006), pp. 104–118.

[3] FRANCILLON, A., AND CASTELLUCIA, C. Code injection attacks on Harvard-architecture devices. In *CCS* (2008).

[4] GOODSPEED, T. Stack overflow exploits of 802.15.4 wireless sensors. Toorcon, September 2007.

[5] GOODSPEED, T. MSP430 BSL passwords: Brute force estimates and defenses, June 2008.

[6] GOODSPEED, T. Practical attacks against the MSP430 BSL. 25C3, December 2008.

[7] GOODSPEED, T. A side-channel timing attack of the MSP430 BSL. Black Hat USA, August 2008.

[8] MAXIM. MAXQ family user's guide: MAXQ3210/MAXQ3212 supplement, 2006.

[9] NERGAL. The advanced return-into-lib(c) exploits (pax case study). *Phrack Magazine 58*, 4 (2001).

[10] SCHAUER, S. Features of the MSP430 bootstrap loader. TI Application Report SLAA089D, August 2006.

[11] SHACHAM, H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS* (2007).

[12] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. On the effectiveness of address-space randomization. In *CCS* (2004).

[13] SKOROBOGATOV, S. P. *Semi-invasive attacks – A new approach to hardware security analysis*. Doctor of Philosophy, University of Cambridge, 2005. ISBN 9729961506.

[14] SOLAR DESIGNER. *return-to-libc attack*. Bugtraq mailing list, August 1997.