Trellix

Technique: Glitching U-Boot (or Other Bootloaders) by Shorting the NAND Flash

The McAfee Advanced Threat Research team conducts security research with the aim of staying ahead of the evolving threat landscape to expose and reduce attack surfaces. This series of white papers discusses laboratory security research techniques that are generally known among the professional community of security researchers. The white papers are provided to elevate collaboration and security within the industry and are not to be used for unlawful purposes. Security researchers are responsible for lawfully obtaining equipment and for complying with contracts and licenses for their research.

Goal

U-Boot (and any other bootloader) is a piece of code that runs before the main operating system on embedded devices. It is the component responsible for loading up the next stage of the boot process, usually the operating system (the Linux image for instance). In most cases, bootloaders have a command line interface that lets you interrupt the boot process and interact with the bootloader itself. Potentially, you can change how the system is loaded or get privileged access to the system memory before the OS takes over. Readers familiar with dual booting their main computer can draw a parallel to the GRUB bootloader on Linux. In many embedded systems, the bootloader's command line interface is going to be locked down to prevent unauthorized manipulation of the system. This is (un)fortunate as, from a vulnerability research perspective, having this type of access would give a researcher more leverage on the system to access its internals. For instance, it is sometimes possible to access the content of the flash memory holding the filesystem or change the boot arguments to get a Linux shell at startup.

Prerequisite

As this technique relies on accessing the bootloader console, it is necessary to first identify if a serial console is available and potentially solder a UART connection to the embedded system. See this link to learn how to do so.

This approach relies on glitching a NAND Flash and, as such, a certain amount of physical tampering will be required. Also, the next stage of the loading process needs to be stored on external storage. Consequently, a System on Chip (SoC) with built-in flash memory will likely not be susceptible to this attack (although other glitching might be viable).
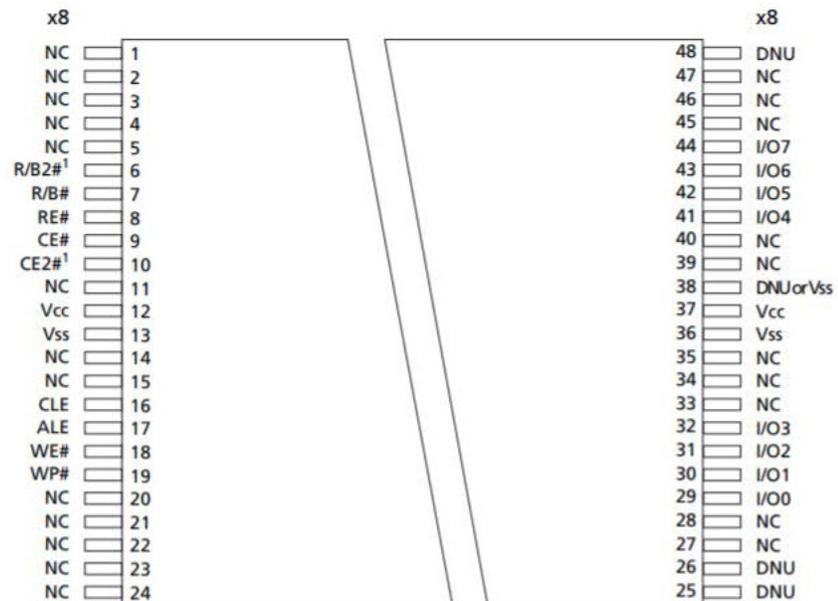
Approach

The idea is simple; the bootloader will print on the console that it is in the process of loading the image it wants to boot. If you make it read this data incorrectly, it will retry a few more times but will eventually give up. In most cases, as a last resort, the bootloader will default to bringing back its command line prompt. The window of action to glitch the loading process depends on the speed of the memory and the size of the image. In many cases you have at least 2-3 seconds to perform the attack. Be aware that if you try to glitch the system too early on, you may potentially prevent the bootloader from loading properly (in some case it is also stored on the NAND flash) and the system will simply hang there. Finally, as is the case for any kind of glitching attack, there is a risk of frying the chip or the whole device when performing the attack. It is usually good to keep that in mind and be ready to lose a device or two. That being said, we were able to perform this dozens of times without unfortunate accidents.

Glitching the Loading Process

A NAND flash is typically a rectangular component with lots of pins on two sides (usually 48, but the number may vary slightly). For instance, this chip was in a VoIP phone:

A description of the pins can be found in the relevant datasheet of the chip (Google the label written on the chip to find it). The pinouts are usually fairly standard; a typical one would look something like the one below. It is also good to know that pin number 1 is identified by a marking on the chip (a recessed circle, highlighted in orange in the top left corner in the picture on page 2).

x8						x8
NC	1			48		DNU
NC	2			47		NC
NC	3			46		NC
NC	4			45		NC
NC	5			44		I/O7
R/B2#[1]	6			43		I/O6
R/B#	7			42		I/O5
RE#	8			41		I/O4
CE#	9			40		NC
CE2#[1]	10			39		NC
NC	11			38		DNU or Vss
Vcc	12			37		Vcc
Vss	13			36		Vss
NC	14			35		NC
NC	15			34		NC
CLE	16			33		NC
ALE	17			32		I/O3
WE#	18			31		I/O2
WP#	19			30		I/O1
NC	20			29		I/O0
NC	21			28		NC
NC	22			27		NC
NC	23			26		DNU
NC	24			25		DNU

Notes: 1. CE2# and R/RB2# are available on 8Gb 2-CE# devices and 16Gb devices only. These pins are NC for other configurations.

Figure 2: 48-Pin TSOP Type 1 Pin Assignment (Top View)

The glitching method consists of connecting a jumper wire to ground on one side and poking at one of the I/O pins with the other end of the jumper cable, while the filesystem loads. Grounding the IO pin will cause misreads and panic the bootloader. In the case described above, pin 44 would be the best candidate as the pins above are not connected (NC) and the pins below are also IOs, so being off by a pin or two would not cause much trouble.

If you are lucky, you should then be able to enter `command` in the boot loader prompt.

Example

Here is an example setup, where we applied this method on a cable modem:

The green-black jumper cable is soldered to Ground on one side, and the other end is left floating. It will be used to poke at the memory chip.

Applications

There are multiple valuable applications to this technique. A straightforward one is to use the u-boot bootloader to load the content of the NAND flash into memory and from there use the "md" command to display the content of the memory loaded with the flash content. When this approach works, it makes it one of the easiest ways to dump the contents of flash memory, without requiring anything more than a serial cable for the boot console. There are two shortcomings to this method though. One is that the dumping is pretty slow (roughly an hour for 10 mb), and the other one is that in certain situations, the memory controller needs to be restarted (`mmc rescan` in u-boot) which might not be possible depending on the bootloader used by the system.

Another interesting application is to change the boot arguments passed to the Linux kernel. For instance, adding a "`init=/bin/bash`" might replace the normal init script with a bash shell. This is a limited approach as the normal initialization of the system is not taking place, but it is an interesting entry point for more research on the embedded device. See this Wink Hub hack for an example of this method in action.

Finally, another potential avenue in more hardened systems is to use this glitch to get a boot prompt, and then leverage separate vulnerabilities in the bootloader code itself to gain a foothold in the system. A more advanced method based on similar principles was used to defeat secure boot in Cisco Phones (link).

Conclusion

With this technique, we can see how some unexpected physical disturbance can open up a new attack surface on embedded systems and having a minimal understanding on how these systems function can be all that is needed to get a root shell and perform security research on a previously closed down device.