U-BOOTING SECURELY

Dmitry Janushkevich

F-Secure Hardware Security Team

Version 1.0, May 2020

F-Secure.

1 ABOUT THIS PAPER

This paper aims to provide an independent analysis of known pitfalls and production misconfigurations related to using U-Boot (officially: Das U-Boot) in secure embedded systems as well as provide developers with guidance towards securing their products. It is aimed at teams and organisations planning to use or already using U-Boot as part of their existing products. Most of the examples have been encountered by the F-Secure Consulting Hardware Security team when researching secure boot implementations and resulted in either a partial or complete compromise of device security.

A small introduction of the relevant security features of U-Boot is included, though some degree of familiarity with the U-Boot ecosystem is helpful.

The U-Boot project is a great asset for system developers providing flexibility and tooling to build and debug embedded systems. As a primary security goal is to ensure an established and maintained chain of trust, there is a clear need to ensure U-Boot is appropriately hardened so it does not pose a threat to system security. Succeeding in this strongly depends on the many configuration choices made during the development.

Given the dynamic nature of the U-Boot project and the many ways it can be configured in the system, this paper should not be taken as a complete or exhaustive guide for your solution; some attack scenarios may not be applicable to all systems or attack scenarios may exist which are not documented here. It may also be the case that the SoC or platform vendor extends U-Boot functionality beyond what is available in the upstream project; such extensions are not covered by this paper.

CONTENTS

2 INTRODUCTION TO U-BOOT

The U-Boot project provides a highly configurable boot loader program supporting multiple processor architectures such as ARM, MIPS, PowerPC, x86 and a multitude of boards. While technically any software may be booted as the application stage, its primary focus is on preparing a suitable boot environment for starting Linux based systems.

To make the system development process easier and more accessible, a command console is provided which allows execution of simple but powerful commands to, for example, interact with storage devices, read files from various media and file systems into memory, read and write arbitrary memory locations, and finally booting an image loaded into memory. Furthermore, custom commands may also be developed and added to U-Boot. Having access to this console, or otherwise being able to execute arbitrary commands, will prove instrumental to any attacker aiming to compromise the target system, akin to executing arbitrary OS commands in conventional environments.

U-Boot also features a kind of in-memory key-value storage presented as environment variables. These associate arbitrary textual values with string keys. A set of console commands is provided to manipulate these variables. In addition, another console command implements scripting by allowing the contents of an environment variable to be interpreted as a series of console commands. This makes it possible to develop a variety of boot scenarios, from simple read-then-execute to complex scripts scanning and choosing boot media to boot from as well as providing debugging mechanisms. Therefore, inspecting environment variables typically provides a good insight into how the boot process is implemented for the specific device.

In addition to supporting legacy image formats such as zImage and uImage, U-Boot integrates support for a new image format based on device tree, called Flattened Image Tree (FIT). This format can include multiple sub-images such as kernels, RAM disks, or device tree blobs. Multiple configurations can be defined, defining specific sub-images to be used during boot should that configuration be chosen. In addition, cryptographic signing of both individual sub-images as well as complete configurations is supported with the verified boot feature, making it possible to check the authenticity and integrity of images being loaded at boot time.

U-Boot is a highly privileged system component, tasked with making important security decisions such as allowing the device to boot a kernel image. This leads to the need for U-Boot to be configured in a way that conforms to the security requirements stated in a product's design document.

The following sections will provide an insight into common misconfigurations and issues, as well as areas of interest to consider when hardening your own implementation.

3 SECURITY CHALLENGES

3.1 OUTDATED U-BOOT VERSION

Developers often prefer to use tried-and-proven versions of software products in their work. This is a valid option to minimise risks related to tooling incompatibilities and related downtime. This is why we see U-Boot versions dating back to 2013 or 2014 being used in products released in 2020.

However, this is a significant issue when it comes to security. Old versions are rarely maintained by developer teams through applying upstream patches, leading to newly released products containing a whole host of already known security problems which were identified by maintainers or the community and patched in later versions.

At least 5 CVEs (of which two were reported[1] by F-Secure) were assigned[2] in 2018 to various versions of U-Boot, while in 2019 this number went up to 20. An important issue was reported by F-Secure[3] in 2020 affecting the verified boot function. As a result, compromising a device which uses an outdated version of U-Boot is as easy as developing an exploit for a known and patched security issue. Typically, if exploitable at all, this does not pose a significant challenge to a person moderately skilled in exploit development.

Further challenges exist when the field update of boot loader components is considered from a reliability perspective. For example, in the case of a failed U-Boot update, the device will stop booting properly unless some additional system is implemented on top to revert to a previous version. Such challenges, however, might lead to integrators choosing to maintain insecure fallback mechanisms.

3.2 EXTRA BOOT OPTIONS PRESENT

It is not a rare event during development that a newly deployed software version breaks the device, making it unbootable. It is thus desirable to have a way to switch the booting process to some sort of fallback mode which, for example, could be provided on removable media such as a microSD card or a USB mass storage drive. Such a feature may also be useful during the initial provisioning process or when deploying software updates in the field.

This feature, however, can be exploited by an attacker to bring forward more attack surface which otherwise would be inaccessible. A good example would be commands related to TFTP or NFS protocols, known to be vulnerable to a variety of issues[4]. In other cases, being able to choose the boot medium might save an attacker the effort of opening the device's enclosure and thus cut down on the time required to conduct an attack.

As an implementation example, boot configurations may typically be chosen via e.g. on-board jumpers, DIP switches, or voltages on exposed connector pins, or otherwise probed automatically by e.g. polling for the presence of USB mass storage device or Ethernet link. By manipulating the jumpers, an attacker may choose

[1] https://github.com/f-secure-foundry/advisories

[2] https://www.cvedetails.com/vendor/18843/Denx.html

[3] https://labs.f-secure.com/advisories/das-u-boot-verified-boot-bypass/

[4] For example: https://github.com/f-secure-foundry/advisories/blob/master/Security_Advisory-Ref_IPVR2018-0001-U-Boot_verified_boot_bypass.txt

other boot modes than the default one and, for example, be able to exploit known issues in the TFTP implementation of U-Boot.

3.3 AUTOBOOT CAN BE INTERRUPTED

Autoboot is a process that allows the system to proceed with the default boot configuration. Crucially, this process can be interrupted through user interaction in the default configuration; such a condition can be easily identified by the characteristic prompt being displayed on the console device. To demonstrate with an example, the following capture from the serial console presents such a situation:

```
U-Boot 2017.03-linux4sam_5.8 (Jan 26 2020 - 14:16:26 +0000)

[ REMOVED FOR BREVITY ]

Hit any key to stop autoboot:  0
=>
```

By following the displayed guidance, it is possible for an attacker to gain access to the U-Boot command console, thereby taking full control over the execution of this and any of the following boot stages. In practice, as the prompt may be not shown on the console (e.g. set to an empty string) an attacker may attempt to interrupt the boot process by issuing key presses during the boot process, for example, pressing Ctrl-C multiple times from the moment the device is powered up and until the control is handed over to the kernel.

3.4 ENVIRONMENT VARIABLES LOADED FROM UNTRUSTED SOURCE

Typically, environment variables are pre-configured and embedded into the final U-Boot image during the build process. But this approach may be suboptimal, for example, where the U-Boot environment may need to be customised after production or per device. It is possible to accommodate this requirement by allowing U-Boot to load the environment variables from an external source such as on-board storage – a file such as `uEnv.txt` on a partition within the eMMC chip, or a location on an SPI Flash.

An example output on the serial console would look like this:

```
...
SD/MMC found on device 1
reading uEnv.txt
58 bytes read in 4 ms (13.7 KiB/s)
Loaded environment from uEnv.txt
...
```

It should be noted, though, that the environment being loaded is not authenticated in any way. This allows an attacker with appropriate access to the board to manipulate the environment at will and in that way execute arbitrary console commands, resulting in taking full control over the boot process.

As a concrete example, consider a system where the SoC is configured to boot securely (the U-Boot image is encrypted) from a serial SPI Flash chip; the default behaviour of loading the environment variables from the save area was left compiled in. An attacker would proceed to desolder the Flash chip and reprogram the save area to contain a crafted `bootcmd` variable; it is easily doable because serial Flash chips can be desoldered with minimal effort and the save area is only CRC protected. While the choice is wide for commands to be used, typically the most useful approach is to reprogram peripheral registers or patch the loaded software to gain full code execution or unrestricted U-Boot console access. The attacker would then place the Flash chip back and observe the programmed commands being executed.

3.5 BOOT PROCESS FAILS OPEN

It is common to implement the customised boot process as a script; indeed, provisions are made in the default U-Boot configuration to use the contents of the `bootcmd` environment variable for such purposes. The typical boot flow includes loading an image into memory, optionally validating it, and passing control to the loaded image. In more complicated scenarios, this may be repeated for multiple boot options until one is found that works.

Similarly, it is common to find implementations where the case of no valid image being found is handled improperly or not handled at all. The default behaviour of U-Boot in this case is to enter the command console mode. Example output:

```
...
## Booting kernel from FIT Image at 22000000 ...
ERROR: can't get kernel image!
=>
```

Leveraging this, an attacker is able to temporarily deny access to the boot media can gain access to an otherwise inaccessible command console, subsequently taking full control over the boot process.

Practical exploitation of such issues is trivial – anything goes, from manipulating the storage contents through the desoldering and reprogramming cycle to simply shorting chip pins which carry image data at the right moment to corrupt the transferred data.

3.6 IMPROPER SIGNING OF FIT IMAGES

When using the verified boot feature, the build process requires embedding public keys used for component verification into U-Boot's device tree so that U-Boot can make use of them for authentication. This is implemented by adding the `-K` command line parameter to the invocation of the `mkimage` tool used to build the FIT image. The end effect is that a new device tree node is added, which contains the public key data together with the configuration setting enabling signature verification for the image to be accepted as valid.

An example from the U-Boot documentation is reproduced below:

```
tools/mkimage -f fit.its -K control.dtb -k keys -r image.fit
```

This parameter is not required to build valid, signed kernel images. Moreover, if it is omitted, built (and signed) FIT images will be accepted and booted anyway as U-Boot's configuration is not updated to actually require signatures. This results in a situation where it is too easy to assume everything is working as expected while the actual security checks are omitted.

In practice, an attacker can leverage this condition by simply writing (or supplying via any other available method, see 3.2) a crafted image, which will be booted without any issue.

3.7 INSECURE BOOT COMMAND IN USE

In situations where more than one boot source or flow is supported, it may happen that there is a need to use multiple boot commands, for example, to use `bootz` in addition to `bootm` in some cases.

However, care should be taken to ensure that in production builds, only the `bootm` command is used or reachable via available boot options. This is due to the fact that only the `bootm` command is able to perform proper image authentication when verified boot is used. The `bootz` command only accepts a raw kernel image and does not perform any kind of image authentication.

This means that if an attacker is able to reach a flow where e.g. the `bootz` command is used, they will be able to boot arbitrary code.

To illustrate this with an example, consider a situation where a script was developed to attempt booting the system security using verified boot from an on-board eMMC chip using the `bootm` command (the normal boot flow) and, in case that fails, try booting from USB using the `bootz` command (the factory provisioning flow). If an attacker is able to force the device to fail booting from eMMC via any available means (corrupting the image, shorting pins, etc), they gain an easy way to bypass verified boot restrictions and boot arbitrary code from a crafted USB mass storage device.

3.8 ACCEPTING UNSIGNED IMAGES

A powerful tool, the `bootm` command is able to handle multiple image types, including the new FIT image format as well as the legacy uImage format, which does not support any kind of image authentication. Support for each image format is enabled during U-Boot configuration. Naturally, handling of these image formats requires the command is able to tell them apart, which it does by validating the header for each supported image format. A typical configuration would generate a signed FIT image, load it into memory and then use the `bootm` command to both authenticate and boot the loaded image.

It follows that if support for the legacy image format is compiled in[5], an opportunity is opened to abuse the automatic guessing of image formats and replace the expected FIT image with a legacy image specially crafted by an attacker. The `bootm` command will validate and accept the loaded legacy image, proceeding to boot unauthenticated code.

Practical exploitation of this issue involves nothing more than preparing a legacy image and replacing the signed FIT image with the crafted one via any means available in a concrete system.

3.9 MISSING AUTHENTICATION IN SPL / TPL BUILDS

Due to restrictions placed on the primary boot loader it may happen that U-Boot needs to be split into two stages, a secondary program loader (SPL) and U-Boot proper. As an example, this is the case when ROM code can only load a small piece of code e.g. only 64K to fit in the available on-chip static memory. In a more complex situation where limits are even stricter, it is also possible to split U-Boot into three parts:

[5] Note that in versions 2013.07-rc1 through 2014.07-rc2 it was impossible to disable support for legacy image format via configuration. This is the subject of the following advisory from Cisco TALOS: https://talosintelligence.com/vulnerability_reports/TALOS-2018-0633

SPL, tertiary program loader (TPL), and U-Boot proper[6]. SPL then loads TPL, and TPL loads the full U-Boot image.

In the secure boot situation, to maintain the chain of trust, the primary boot loader authenticated the next stage. However, there appears to be no universal solution provided by U-Boot for SPL and TPL code to authenticate the next stage they load and pass control to; only some platform-specific implementations seem to exist[7] in the main branch. This means, such functionality may need to be implemented by custom code, otherwise the chain of trust is broken.

When no authentication of code loaded by SPL and TPL is implemented, an attacker is able to leverage this by writing arbitrary code for the stage that is missing authentication and subsequently gain arbitrary code execution on the target device.

[6] https://www.denx.de/wiki/pub/U-Boot/MiniSummitELCE2013/tpl-presentation.pdf

[7] https://github.com/u-boot/u-boot/blob/master/doc/SPL/README.spl-secure-boot

4 PRODUCING A HARDENED U-BOOT SYSTEM

The following steps are suggested to aid in hardening a U-Boot configuration, so the deployment is no longer subject to the security challenges described in the previous chapter:

- Implement a patch and vulnerability management process to ensure the latest available version is deployed in new software builds or all upstream security patches are applied and are effective against the version in use.

- Review the available boot options, if any, and ensure only the required boot flows are included in production software builds in order to limit the attack surface available to an attacker.

- If the ability to interrupt the boot process is not required, disable this feature by setting the `bootdelay` environment variable to a negative value. Disable the `CONFIG_USE_AUTOBOOT_MENUKEY` feature. If interrupting the boot process remains necessary, enable the password requirement for autoboot by choosing the `CONFIG_AUTOBOOT_KEYED` and `CONFIG_AUTOBOOT_ENCRYPTION` options and set the `CONFIG_AUTOBOOT_STOP_STR_SHA256` value to the SHA-256 hash of the chosen password.

- Avoid loading environment variables from any external storage by verifying any use of the `env import` command in boot scripts[8]. Enable the `ENV_IS_NOWHERE` option and disable other environment storage options[9] to avoid automatic environment loading by U-Boot and only use the default environment compiled into the U-Boot image.

- Implement a dead loop or use the `reset` command[10] at the end of any boot flow to ensure the boot process does not fail open in case no valid image is found.

- Verify that the U-Boot device tree is updated with the signature node during the build process by converting the resulting `.dtb` file back to its textual form using the `dtc` program and verifying the presence of the signature node. Implement a test case verifying that unsigned images are rejected during boot.

- Review the boot flow and ensure, if verified boot is used, that no boot commands other than `bootm` are in use or reachable via any implemented boot options – whether explicitly selectable by the end user or implicitly chosen via e.g. bootstrapping resistors.

- Disable the legacy image format if it is not required. This can be achieved via the `LEGACY_IMAGE_FORMAT` configuration option.

- Verify the presence and implement, if required, authentication of all stages when U-Boot is compiled with SPL or TPL. Note that such code may already be supplied by the platform manufacturer.

[8] An example: https://github.com/u-boot/u-boot/blob/master/include/configs/baltos.h#L105

[9] https://github.com/f-secure-foundry/usbarmory/blob/master/software/u-boot/0001-ARM-mx6-add-support-for-USB-armory-Mk-II-board.patch#L342

[10] https://github.com/f-secure-foundry/usbarmory/blob/master/software/u-boot/0001-ARM-mx6-add-support-for-USB-armory-Mk-II-board.patch#L1154

5 CONCLUSIONS

U-Boot is an extremely feature rich and flexible software. As such, significant effort is required to ensure only features that are required and secure are selected during the development and in use during the boot process. A significant amount of cases and situations exist where U-Boot features are not set up to maximum security by default as U-Boot is a tool designed with the primary goal of helping with and speeding up product development.

Producing a highly secure system using U-Boot as the boot loader may not be very straightforward, but it is certainly within reach. As an example, the USB Armory project provides[11] a hardened configuration which could be used as a reference.

6 ABOUT THE AUTHOR

Dmitry Janushkevich began his career as a testing and later embedded software engineer working on the development of leading-edge solid-state drive technologies. Together with a bachelor's degree in computer systems design, this provided a strong background in embedded systems design and development for future explorations in their security.

After joining F-Secure and gaining experience in customer-facing consultancy, embedded systems security became his primary focus. Currently a senior consultant, he has a strong track record in providing security-related consulting for automotive, aerospace, and consumer electronics industries.

7 ABOUT F-SECURE HARDWARE SECURITY TEAM

F-Secure’s Hardware Security team, founded as Inverse Path in 2005, provides information security consulting to the most unique, challenging and critical industries in the world. We provide industry-leading services to secure hardware, safety-critical embedded systems, software applications and IT infrastructure.

We deliver detailed and comprehensible security analysis of software and hardware systems, along with practical and effective mitigation and protection strategies.

With a vast breadth of experience in hardware and software design and engineering, we are trusted by companies across the globe to assess and test their products and processes. Our work safeguards products from malicious compromise, and in doing so protects the safety of passengers, ensures the resilience of critical infrastructure, and secures company trade secrets and intellectual property.

[11] https://github.com/f-secure-foundry/usbarmory/blob/master/software/u-boot/0001-ARM-mx6-add-support-for-USB-armory-Mk-II-board.patch