Device Tree Blob Verfication

Mariano Marciello
marciello.mariano@gmail.com

Università degli studi di Modena e Reggio Emilia
Kernel Hacking 2020/2021

Contents

1 Abstract

This report describe the verification procedure, as a proof of concept, for a subset of the Device Tree Blob from the Linux kernel. This is useful because we want to know, if some part of the Device Tree is modified during runtime and notify this to the user.

This verification, in a real scenario, could prevent malicious modificiation to the Device Tree and a possible **Side Channel Attack**.

This report is organized as follow. Introduction explain what is a Device Tree, the basic verification idea and some tools; Linux Kernel Crypto API explain the API used for the hash function; Device Tree navigation discuss how to navigate a Device Tree; Device Tree hash custom property define where we store the hash digest; Code flow summarize the code flow of the module; Testing show how to test and some example of output; conclusion will be in Conclusion and future implementation; Acknowledgements.

2 Introduction

In this section we will see what is a Device Tree, the main idea for the verification process, and the utilized tools.

2.1 What is a Device Tree

A Device Tree is a language and a data structure used for describing hardware components. Before the intruduction of the Device Tree, all the information described by this data structure were hardcoded in the kernel source. A Device Tree have nodes and property. Each node has name/value pairs for describe the hardware.

The following example describe a node with name *cpus* with a subnode *cpu*. The *cpu* subnode have 3 property: *reg, compatible, device_type.*

```
/dts-v1/;
{ /* incomplete example
cpus {
        #size-cells = <0x00>;
        #address-cells = <0x01>;
     cpu@0 {
```

```
                reg = <0x00>;
                compatible = "arm,cortex-a57";
                device_type = "cpu";
        };
};
```

2.2 Verification idea

The verification idea was base on the U-Boot Verified Boot described in this gitlab page. In this page is explained how the Flatten Image Tree is verified. It is a data structure with image and an associated hash value. The verification process use the **RSA** algorithm; it sign the Flatten Image Tree with the private key and store the public key in a **trusted place**. When U-boot try to verify the Flatten Image Tree, recover the public key from the **trusted place**. With this verification procedure every modification to the Flatten Image Tree is discovered and notified to the user.

The idea for the verification of the Device Tree is:

1. tag a subset of the Device Tree;
2. pull out the Device Tree nodes from the kernel;
3. verify each node;
4. indicate if the node is verified or not.

For this purpose we create a module that when inserted follows the previous steps. Instead of using **RSA** algorithm we use the **sha1** hash function.

2.3 Tools

For this task we have used the following tools:

1. **Linux Kernel**, version 5.9.0;
2. **QEMU**, a machine emulator virtualizer;
3. **U-boot**, as bootloader;

4. **buildroot**, to generate an environment for embedded system with cross compilation;

5. **tftp**, for transfer binary in the virtualized system.

All the previous tools are embedded in the setup provided by **Joakim Bech** at the following github repository.

3 Linux Kernel Crypto API

First of all we think that this verification, like the U-boot Flatten Image Tree verification, should be done with the **RSA** algorithm; but since this is only a proof of concept and not a real implementation, we decide to switch on the easiest **sha1** hash function. For this purpose we used the **Linux Kernel Crypto API**, some documentation about this api can be found at the following link `https://www.kernel.org/doc/html/v4.15/crypto/index.html`.

In this api there are two main entities:

1. **consumer**, that require cryptographic function;

2. **transformation**, that implement cryptographic function.

The transformation object is an instance of a transformation implementation.

For use the transformation object, reffered as ”cipher handle”, we need to follow this steps:

1. initializate a cipher handler;

2. execute all the cipher related operation, provided by the consumer (e.g. calculate the digest of a given array of data);

3. destroy the cipher handler.

We need to recover the data to hash; in order to do this we need to navigate the Device Tree and take all the property in a certain subnode.

4 Device Tree navigation

In the <linux/of.h> header file, there are a lot of helpful function that we used for take the node object from the Device Tree at run time.

In this header file is defined the **device_node** structure that describe a Device Tree's device node. There I propose a simplified version with commented element.

```
struct device_node {
        const char *name; /* node name */
        const char *type; /* node type */

        struct property *properties; /* node's properties */
        struct device_node *parent; /* node's parent */
        struct device_node *child; /* node's child */
        struct device_node *sibling; /*node's sibling */
};
```

We can navigate through the device node using **parent** and **child** pointers. Another importat element of this structure is the **property** pointers, that point to the **property** structure, that follows.

```
struct property {
        char *name; /* property's name */
        int length; /* value's length */
        void *value; /* property's value */
        struct property *next; /* next property in the node */
};
```

We can navigate the properties in a given node using the **next** pointer.

4.1 void *value

As we can see from the **property** structure, the **value** pointer is void. Therefore we need a function that can read every type of binary blob, including **0x00**, and copy this in an char array. If we try to copy this binary blob on an char array, without any special function, when we try to read from it, we have the special byte **0x00** that truncate the array.

We use the **hex_dump_to_buffer** function for convert a blob of data in an "hex ASCII" rappresentation.

5 Device Tree hash custom property

For accomplish the verification procedure, we need a place for store the digest. There are three main options:

1. create a new Device Tree property;
2. hardcode the digest in the module;
3. pass the digest as argument of the module.

Between this possibility we choose to create a new custom property, in the Device Tree because we think is the most appropriate and useful solution. So as an example a device node will be modified as follows:

```
/dts-v1/;
{ /* incomplete example
cpus {
        #size-cells = <0x00>;
        #address-cells = <0x01>;
        hash = "130e aea6 7c4c 2bb2 46fd a0de 65d0
782f c410 1c00";
     cpu@0 {
                reg = <0x00>;
                compatible = "arm,cortex-a57";
                device_type = "cpu";
        };
};
```

The **hash** property is the sha1 digest of the char array composed by namevalue element of the subnode, hex encoded. The entire char array is:

```
reg0000 0000compatible61 72 6d 2c 63 6f 72 74 65 78 2d
61 35 37 00device_type7063 0075name7063 0075
```

Therefore the parent node can verify the subnode.

6 Code flow

In this section we propose an overview of the code flow. The module accomplish the following job:

1. initailizate a cipher handler;
2. search for node whit name taken as input from command line, (if no name is provided the module will iterate over all the Device Tree nodes);
3. search if this node have the hash property;
4. calculate the total dimension of the chararray for sha1;
5. populate the chararray with namevalue element from the subnode;
6. calculate digest;
7. compare the stored digest with the calculated one;
8. print a message for notify the user (return to point 3 if need to iterate over all Device Tree nodes);
9. destroy the cipher handler.

7 Testing

In this section we will see how we test this module, and some example of output.

7.1 Creation of the custom Device Tree Blob

We build two Device Tree Source, called **correct.dts** and **incorrect.dts**.The first one is build with a valid digest, the second one instead have the same digest but we have modified some property. Since the sha1 function provide integrity guarantee, we will be notified about the modification.

Before start our testing section we need to compile the **.dts** file, using the **Device Tree Compiler**. The compilation will produce a **.dtb** file, that will be transfered in the virtualized environment with **tftp**.

7.2 Output

Verification of the **correct.dts**:

```
Error - node [flash] no property hash
Node [cpus] Hash value 130e aea6 7c4c 2bb2 46fd a0de 65d0
782f c410 1c00
Node [cpus] Hash len 49
Node [cpus] DTS hash len 49
Node [cpus] DTS hash value 130e aea6 7c4c 2bb2 46fd a0de 65d0
782f c410 1c00
!! Child node of [cpus] verified !!
Node [test] Hash value c7c9 ece2 204d 105d bc7c c0f2 7d8b
d3cd 38e7 b4e3
Node [test] Hash len 49
Node [test] DTS hash len 49
Node [test] DTS hash value c7c9 ece2 204d 105d bc7c c0f2 7d8b
d3cd 38e7 b4e3
!! Child node of [test] verified !!
Error - node [timer] no property hash
```

Verification of the **incorrect.dts**:

```
Error - node [flash] no property hash
Node [cpus] Hash value ff6a fb7b b7cd 4940 ab4f 7e74 0091
f4de 48d1 1895
Node [cpus] Hash len 49
Node [cpus] DTS hash len 49
Node [cpus] DTS hash value 130e aea6 7c4c 2bb2 46fd a0de 65d0
782f c410 1c00
ALERT!! child of [cpus] node not verified
Node [test] Hash value 28ff a6a3 3adc be89 caca 860c ab65
cdd1 6f50 e275
Node [test] Hash len 49
Node [test] DTS hash len 49
Node [test] DTS hash value c7c9 ece2 204d 105d bc7c c0f2 7d8b
d3cd 38e7 b4e3
ALERT!! child of [test] node not verified
Error - node [timer] no property hash
```

8 Conclusion and future implementation

With this proof of concept now we know that is possible to verify a subset of the Device Tree Blob from the Kernel side.

In a real scenario the verification must be done with a cryptographic system which uses pairs of keys: public keys and private keys; this will ensure not only integrity but authenticity too.

Moreover, there must be an autonomous way for create verification element in the Device Tree Blob, one way for do this is during the compilation of the Device Tree Source.

9 Acknowledgements

I'm greatful to Joakim Bech who have guided me during this project, and several times cleared my mind. I would also like to thank Prof. Paolo Valente for give me the opportunity to work on this interesting project.