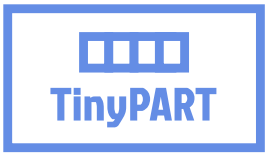

D2.2 : Secure Microcontainer Provisionning

Delivrable number: D2.2
Work package: WP2
Version: 1.0
Deliverable Lead Org: Inria
Deliverable type: R (Document, Report) | DEM (Demonstrator, pilot, prototype, plan designs)
Dissemination Level: PU (Public)
Date: November 24, 2022

Authors:
Emmanuel Baccelli (Inria), Chrystel Gaber (Orange)

Reviewers:
Samuel Legouix (Orange), Damien Amara (University of Lille)

Abstract:
This deliverable presents a prototype of embedded software update manager for ultra-low power script microcontainer, enabling updates over the network, with authenticity, and integrity guarantees on software updates. This deliverable also defines metadata that can be sent along with the container for access control purpose.

Keywords: Microcontainer, Secure Software Update, access control

Acknowledgment The research leading to these results partly received funding from the MESRI-BMBF German-French cybersecurity program under grant agreements no ANR-20-CYAL-0005 and 16KIS1395K. The paper reflects only the authors' views. MESRI and BMBF are not responsible for any use that may be made of the information it contains.

Document Revision History

Document revision history			
Version	Date	Description of change	List of contributor(s)
v0.1	22/06/2022	Creation	Emmanuel Baccelli
v0.2	26/10/2022	Review	Chrystel Gaber, Samuel Legouix
v0.3	28/10/2022	Review	Damien Amara
v0.4	07/11/2022	Take into account review comments	Chrystel Gaber
v1.0	24/11/2022	Take into account Steering Committee comments	Chrystel Gaber

Table 1: Document revision history

Contents

Introduction

Based on RIOT, SUIT end-to-end security, and standard low-power network protocols, the prototype update manager we designed can provide security-enhanced embedded system software and secure updates over the network for general-purpose IoT OS firmware update or for more lightweight and more specific IoT software modules hosted and isolated in femto-containers.

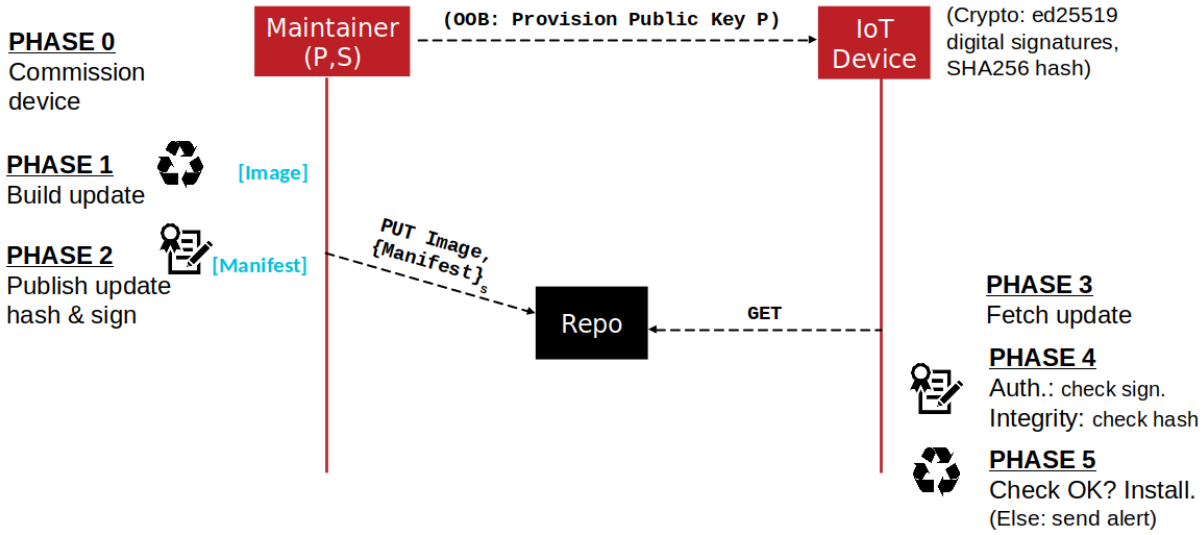

Figure 1: SUIT secure software update workflow.

1 SUIT

We contributed to the standardization of software update security mechanism at IETF: the Software Updates for Internet of Things (SUIT) specifications [1]. SUIT defines a security architecture, standard metadata and cryptographic schemes able to secure IoT software updates, applicable on microcontroller-based devices.

1.1 SUIT Workflow

Figure 1 shows the SUIT workflow. In the preliminary *Phase 0*, the authorized maintainer flashes the IoT device with commissioning material: the bootloader, initial image, and authorized crypto material. Once the IoT device is commissioned, up and running, we iterate a cycle of Phases 1-5, whereby the authorized maintainer can build a new image (*Phase 1*), hash and sign the corresponding standard metadata (the so-called SUIT manifest, *Phase 2*) and transfer to the device over the network via a repository (e.g. a CoAP resource directory). The IoT device fetches the update and SUIT manifest from the repository (*Phase 3*), and verifies the signature (*Phase 4*). Upon successful verification, the new software is installed and booted (*Phase 5*); otherwise, the update is dropped.
The cryptographic tools needed for software updates in general, and SUIT in particular, are a digital signature scheme and a hash function. The digital signature authenticates (a hash of) the software update binary. To make the signature verification less cumbersome, the signature is not performed on the software update binary itself, but on a hash of the software update binary. Thus, this hash function is also a crucial cryptographic primitive in the SUIT workflow. Fig. 1 depicts this workflow combining SHA-256 hashing and Ed25519 signatures. Other hash functions, and other signature schemes can be used with SUIT. For exmaple, in our recent paper [2] we prototyped and comparatively evaluated the use of SHA-3 hashes and quantum-resistant digital signature schemes instead of SHA-256 hashing and Ed25519 signatures (based on pre-quantum elliptic curve cryptography).

1.2 Security features of SUIT

The metadata and the cryptographic primitives specified by SUIT can mitigate attacks exploiting software updates, including:

- *Tampered/Unauthorized Software Update Attacks:* Adversaries may try to update the IoT device with a modified, intentionally flawed software. To counter this threat, SUIT specifies the use of digital signatures on a hash of the image binary and the metadata, to ensure the integrity of both.
- *Unauthorized Software Update Attacks:* An unauthorized party may attempt to update the IoT device with modified software. Using digital signatures and public key cryptography, our prototype based

on SUIT ensure that only the authorized maintainer (holding the authorized private key) will be able to update de device.

- *Software Update Replay Attacks:* Adversaries may replay a valid, but old (known-to-be-flawed) update. To mitigate this threat, SUIT metadata includes a sequence number that is increased with each new software update.
- *Software Update Mismatch Attacks:* Adversaries may send an authentic update to an incompatible device. To counter this, SUIT specifies the inclusion of device-specific conditions, to be verified before installing new software.

2 IoT Software Update Manager Prototype

We provide an open-source implementation of the SUIT workflow which we integrated in the main branch o RIOT [3], a common operating system for low-power IoT devices [4] which we used as base for our microcontainer prototypes.
Our update manager prototype is generic in the sense that it enables both to manage the secure update of:

- Software modules (i.e. OTA microcontainer update);
- Firmware (FOTA updates)

The microcontainer update demo shows the integration between the secure update manager and the microcontainer runtime environment. The secure update manager leverages SUIT as described above to deliver the application payload. The microcontainer runtime uses Femto-Containers to run the applications.
The demonstrator uses two in-memory slots that can be updated with new payloads. Both of these in-memory slots are hooked up to the Femto-Container CoAP endpoints. These act as a demonstrator for the SUIT capabilities together with Femto-Containers. After an application is loaded in either of the available slots, the CoAP endpoint can be used to trigger the Femto-Container application and the CoAP response will contain the resulting return value from the Femto-Container execution.
The in-memory slots are updated using the SUIT workflow in RIOT, as described in the `README.md` file present with the deliverable.

3 Microcontainer Update Manager Demo

When building the example application for the native target, the firmware update capability is removed. Instead two in-memory slots are created that can be updated with new payloads. Both of these in-memory slots are hooked up to the Femto-Container CoAP endpoints. These act as a demonstrator for the SUIT capabilities together with Femto-Containers.
The steps described here show how to use SUIT manifests to deliver content updates to a RIOT instance. The full workflow is described, including the setup of simple infrastructure.

3.1 Prerequisites

Install python dependencies (only Python3.6 and later is supported):

```
pip3 install --user cbor2 cryptography
```

Install aiocoap from the source

```
pip3 install --user aiocoap[linkheader]>=0.4.1
```

See the aiocoap installation instructions more details. add `/.local/bin` to PATH The aiocoap tools are installed to `/.local/bin`. Either add `export PATH=$PATH: /.local/bin` to your `/.profile` and re-login, or execute that command *in every shell you use for this tutorial*.

3.2 Key Management

SUIT keys consist of a private and a public key file, stored in `$(SUIT_KEY_DIR)`. Similar to how ssh names its keyfiles, the public key filename equals the private key file, but has an extra `.pub` appended. `SUIT_KEY_DIR` defaults to the `keys/` folder at the top of a RIOT checkout. If the chosen key doesn't exist, it will be generated automatically. That step can be done manually using the `suit/genkey` target.

3.3 Setting up networking

To deliver the payload to the native instance, a network connection between a coap server and the instance is required. First a bridge with two tap devices is created:

```
sudo RIOT/dist/tools/tapsetup/tapsetup -c
```

This creates a bridge called 'tapbr0' and a 'tap0' and 'tap1'. These last two tap devices are used by native instances to inject and receive network packets to and from.
On the bridge device 'tapbr0' an routable IP address is added such as `2001:db8::1/64`:

```
sudo ip address add 2001:db8::1/64 dev tapbr0
```

3.4 Starting the CoAP server

As mentioned above, a CoAP server is required to allow the native instance to retrieve the manifest and payload. The 'aiocoap-fileserver' is used for this, hosting files under the 'coaproot' directory:

```
aiocoap-fileserver coaproot
```

This should be left running in the background. A different directory can be used if preferred.

3.5 Building and starting the example

Before the natice instance can be started, it must be compiled first. Compilation can be started from the root of your RIOT directory with:

```
make -C suit_femtocontainer
```

Then start the example with:

```
make -C suit_femtocontainer term
```

This starts an instance of the `suit_update` example as a process on your computer. It can be stopped by pressing `ctrl+c` from within the application.
The instance must also be provided with a routable IP address in the same range configured on the 'tapbr0' interface on the host. In the RIOT shell, this can be done with:

```
> ifconfig 5 add 2001:db8::2/64
```

Where 5 is the interface number of the interface shown with the 'ifconfig' command.

3.6 Generating the femto-container application and manifest

To update the storage location we first need the Femto-Container payload application:

```
make -C suit_femtocontainer/bpf
cp suit_femtocontainer/bpf/temp_sens.bin
```

Make sure to store it in the directory selected for the CoAP file server.
Next, a manifest template is created. This manifest template is a JSON file that acts as a template for the real SUIT manifest. Within RIOT, the script `dist/tools/suit/gen_manifest.py` is used.

```
dist/tools/suit/gen_manifest.py --urlroot coap://[2001:db8::1]/
  --seqnr 1 -o suit.tmp coaproot/temp_sens.bin:0:ram:0
```

This generates a suit manifest template with the sequence number set to `1`, a payload that should be stored at slot offset zero in slot `.ram.0`. The url for the payload starts with `coap://[fe80::4049:bfff:fe60:db09]/`. Make sure to match these with the locations and IP addresses used on your own device.
SUIT supports a check for a slot offset. Within RIOT this is normally used to distinguish between the different firmware slots on a device. As this is not used on a native instance, it is set to zero here. The location within a SUIT manifest is an array of path components. Which character is used to separate these path components is out of the scope of the SUIT manifest. The `gen_manifest.py` command uses colons (`:`) to separate these components. Within the manifest this will show up as an array containing `[ "ram", "0" ]`.
The content of this template file should look like this:

```
{
    "manifest-version": 1,
    "manifest-sequence-number": 1,
    "components": [
        {
            "install-id": [
                "ram",
                "0"
            ],
            "vendor-id": "547d0d746d3a5a9296624881afd9407b",
            "class-id": "bcc90984fe7d562bb4c9a24f26a3a9cd",
            "file": "coaproot/temp_sens.bin",
            "uri": "coap://[fe80::4049:bfff:fe60:db09]/temp_sens.bin",
            "bootable": false
        }
    ]
}
```

The manifest version indicates the SUIT manifest specification version numbers, this will always be 1 for now. The sequence number is the monotonically increasing anti-rollback counter.
Each component, or payload, also has a number of parameters. The install-id indicates the unique path where this component must be installed. The vendor and class ID are used in manifest conditionals to ensure that the payload is valid for the device it is going to be installed in. It is generated based on the UUID(v5) of 'riot-os.org' and the board name ('native').
The file and uri are used to generated the URL parameter and the digest in the manifest. The bootable flag specifies if the manifest generator should instruct the node to reboot after applying the update.
Generating the actual SUIT manifest from this is done with:

```
dist/tools/suit/suit-manifest-generator/bin/suit-tool create -f suit
  -i suit.tmp -o coaproot/suit\_manifest
```

This generates the manifest in SUIT CBOR format. The content can be inspected by using the 'parse' subcommand:

```
dist/tools/suit/suit-manifest-generator/bin/suit-tool parse
  -m coaproot/suit\_manifest
```

The manifest generated doesn't have an authentication wrapper, it is unsigned and will not pass inspection on the device or RIOT instance. The manifest can be signed with the 'sign' subcommand together with the keys generated earlier.

```
dist/tools/suit/suit-manifest-generator/bin/suit-tool sign
  -k keys/default.pem -m coaproot/suit\_manifest
  -o coaproot/suit\_manifest.signed
```

This generates an authentication to the manifest. This is visible when inspecting with the 'parse' subcommand. The URL to this signed manifest will be submitted to the instance so it can retrieve it and in turn retrieve the component payload specified by the manifest.

3.7 Updating the storage location

The update process is a two stage process where first the instance pulls in the manifest via a supplied url. It will download the manifest and verify the content. After the manifest is verified, it will proceed with executing the command sequences in the manifest and download the payload when instructed to.
The URL for the manifest can be supplied to the instance via the command line.

```
> suit coap://[2001:db8::1]/suit_manifest.signed
```

The payload is the full URL to the signed manifest. The native instance should respond on this by downloading and executing the manifest. If all went well, the output of the native instance should look something like this:

```
suit coap://[2001:db8::1]/suit_manifest.signed
suit_coap: trigger received
suit_coap: downloading "coap://[2001:db8::1]/suit_manifest.signed"
suit_coap: got manifest with size 276
suit: verifying manifest signature
suit: validated manifest version
Retrieved sequence number: 0
Manifest seq_no: 1, highest available: 0
suit: validated sequence number
Formatted component name: .ram.0
validating vendor ID
Comparing 547d0d74-6d3a-5a92-9662-4881afd9407b
  to 547d0d74-6d3a-5a92-9662-4881afd9407b from manifest
validating vendor ID: OK
validating class id
Comparing bcc90984-fe7d-562b-b4c9-a24f26a3a9cd
  to bcc90984-fe7d-562b-b4c9-a24f26a3a9cd from manifest
validating class id: OK
SUIT policy check OK.
```

```
Formatted component name: .ram.0
Fetching firmware |XXXXXXXXXXXXXXXXXXXXXXXX| 100%
Finalizing payload store
Verifying image digest
Starting digest verification against image
Install correct payload
Verifying image digest
Starting digest verification against image
Install correct payload
```

The storage location can now be inspected using the built-in command. If the same payload as suggested above was used, it should look like this:

```
> storage_content .ram.0 0 64
72425046000000000000...
```

The process can be done multiple times with both slot '.ram.0' and '.ram.1' and different payloads. Keep in mind that the sequence number is a strict monotonically number and must be increased after every update.
subsectionFemto-Containers
The Femto-Containers executable is stored in slot `.ram.0` for simplicity in this example. It can be triggered for execution by sending a POST request over CoAP to the texttt/bpf/exec/0 endpoint on the instance running.

```
aiocoap-client -m POST coap://[2001:db8::2]/bpf/exec/0
```

4 Delivery source code location

The prototypes described in this document are listed in table 2.

Source Code delivered			
Prototype	License	Location	Entity
RIOT	OpenSource	TinyPART github D2.2.1 repository	INRIA
suit femtocontainer	OpenSource	TinyPART github D2.2.1 repository	INRIA

Table 2: Prototypes delivered

References

[1] B. Moran *et al.*, "A Concise Binary Object Representation (CBOR)-based Serialization Format for the Software Updates for Internet of Things (SUIT) Manifest," *IETF Internet Draft draft-ietf-suit-manifest-17*, 2022.

[2] G. Banegas *et al.*, "Quantum-Resistant Security for Software Updates on Low-power Networked Embedded Devices," *20th International Conference on Applied Cryptography and Network Security (ACNS)*, 2022.

[3] "RIOT Open Source Implementation of SUIT." `https://github.com/RIOT-OS/RIOT/tree/master/examples/suit_update`. Accessed: 2022-06-22.

[4] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Riot: An open source operating system for low-end embedded devices in the iot," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, 2018.

[5] C. Gaber, N. Dejon, S. Legouix, K. Zandberg, E. Baccelli, E. Gulati, and D. Amara, "D4.1.1 - Interfaces definition," tech. rep., TinyPART Consortium, 2022.