Instructions for Toto’s demo

In this demo, we will use Toto to secure a supply chain with a very simple workflow. Alice will be the project owner and developer of the project and Bob will be a packager that turns Alice’s code into a tarball and sends it over to the client.

For the sake of demonstrating Toto, we will have you run all parts of the supply chain. This is, you will perform the commands on behalf of the Project Owner, the functionaries and the client.

Setting up the Virtual Machine

The virtual machine can be downloaded from here as an OVA appliance. To keep things simple, this machine only contains a very stripped-down linux box, with all the required libraries to run Toto.

Once having downloaded the virtual machine, use VirtualBox to import the appliance. This can be done by clicking “File -> Import Appliance” and selecting “toto-testV5.ova”.

Once VirtualBox is done importing, select the “Toto test” virtual machine on the sidebar and click on “Start”.

Once the machine starts, you will be dropped into a root shell automagically (no password required), and you are ready to test Toto.

Testing Toto

Inside the virtual machine, you will find four folders: “toto”, “project_owner”, “functionaries,” and “final product.” The “toto” folder is where the toto executables reside, whereas the rest of the folders will be where we perform the tasks of each role.

Supply chain layout

First, we will need to define the supply chain layout. To simplify this process, we provide a script that generates a simple layout for the purpose of the demo. In this supply chain, we have Alice, who is the project owner that creates the layout, Bob, who uses vi to create a

Python program foo.py, and Carl, who uses tar to package up foo.py into a tarball and includes Toto metadata. Finally, we have the end user, who installs and validates the files.

Creating the layout can be done by going into the project owner directory:

```
# cd project_owner/
# ls
create_layout.py
```

Once you are in the project_owner folder, you will find a create_layout.py script. This script will be used to generate the supply chain layout. We can execute it with:

```
# python create_layout.py
```

Once the script is done, two functionary keys will be generated, as well as the layout file (root.layout). You can check the contents of the layout file by doing

```
# less root.layout
```

There, you will find that (besides the signature and other information) there are two steps, write_code and package, that the functionaries will perform.

Now, we will provide the keys to the functionaries, so they can perform their operations:

```
# cp bob carl ../functionaries
```

Carrying out the steps

Now, we will take the role of the functionaries, to do this, we will change to the functionaries folder:

```
# cd ../functionaries
# ls
Bob carl
```

We will perform the steps on behalf of Alice, who is in charge of writing the code, first we do:

```
# toto-run.py --name write-code --products foo.py --key bob -- vi foo.py
```

A vi window will be open (and wrapped), so alice can write a script. You can write whatever you want here. After you save the file and close vi (this can be done with :x), you will find “write-code.link” inside the folder. This is the piece of link metadata that

Alice should send to the client for verification.

Here is what happens behind the scenes:

- Toto wraps the command “vi foo.py”,
- hashes the product “foo.py”,
- creates link metadata “write-code”,
- signs the metadata with alice’s private key, and
- stores everything to “write-code.link”

Now, we will perform Bob’s step. In the same folder, execute the following:

```
        # toto-run.py --name package --materials foo.py --products foo.tar.gz
--key carl -- tar zcvf foo.tar.gz foo.py
```

This will create a new link metadata file, called package.link. You can inspect the contents of the link metadata files by doing:

```
        # less package.link
```

With these two pieces, we can form the final product.

Forming the final product

Now, we will copy all the relevant files into the final product so we can run the client verification routine. First, we will copy the layout that we created.

```
        # cd ..
        # cp project_owner/root.layout project_owner/alice.pub final_product/
```

Then, we will copy the pieces of link metadata and the target files:

```
        # cp functionaries/foo.tar.gz functionaries/package.link
functionaries/write-code.link final_product/
```

Finally we will change into the final_product directory and run verification

```
        # cd final_product
        # toto-verify.py --layout root.layout --layout-key alice.pub
```

This instructs the script to:

- Verify the layout is signed with Alice’s key.[1]
- Run the verification routine using the root.layout file
    - Verify that each step was performed (signed) by the authorized functionary
    - Verify that the used commands (“vi”, “tar”) align with the expected commands
    - Run inspections (“untar”)
    - Verify the matchrules for products and materials

From it, you will see the meaningful input saying “PASSING” and a return value of 0, that indicates verification worked out well:

```
# echo $?
0
```

Now, let’s try to tamper with the supply chain.

Tampering with the supply chain

Imagine that someone got a hold of the foo.py file before it was passed over to Bob (e.g., someone hacked into the version control system). We will simulate this by changing foo.py before we generate Bob’s link:

```
# cd ../functionaries
# echo “something evil” >> foo.py
```

Now - foo.py was tampered before it was packaged - we will run Bob’s step, who unwittingly packages the tampered version of foo.py

```
        # toto-run.py --name package --materials foo.py --products foo.tar.gz
--key carl -- tar zcvf foo.tar.gz foo.py
```

This will regenerate bob’s link, and the foo.tar.gz file, let’s copy them over to the final product:

```
# cp foo.tar.gz package.link ../final_product
```

[1] For simplicity, we use alice’s key to both sign the layout and write the code.

And re-run the verification:

```
# cd ../final_product
# toto-verify.py --layout root.layout --layout-key alice.pub
```

This time, Toto will detect that the products from Alice's step do not match what Bob used as a material, and it will warn accordingly. It will also return a non-0 value to show this:

```
# echo $?
1
```

Wrapping up

Congratulations! You have completed the Toto demo! This exercise shows a very simple case in how Toto can protect the different steps within the supply chain. More complex supply chains that contain more steps can be created in a similar way. You can read more about what Toto protects against and how to use it here.