Provisioning secrets from a deployment machine to deployed VM's using the Certifier Framework

Overview

This document describes the steps needed to create a collection of cooperating programs and procedures within a *security domain,* as that term is used in the *Certifier Framework,* to create, and configure *deployable VM*'s using the *Certifier Framework*, so that the *deployable VM*'s can be securely provisioned with application secrets, at run time, ensuring that *security domain policy* is enforced wherever such application secrets are visible. These secrets can be used by cooperating applications, for example, to encrypt data that can be accessed by, and only by, VM instances in the *security domain* that enforce security domain *security policy* everywhere.

We illustrate this in a simple scenario ("scenario 1"), using tools and examples available in the *Certifier Framework*, as distributed. In scenario 1, there are two *Certifier Framework* environments (one for a *deployment machine* and one for the *deployed VM's*). The *security policy* specifies the measured programs which are fully trusted to comply with *security domain* rules and procedures, on both the *deployment machine* and pre-distributed *deployed VM* instances, when they run on specified platforms (like SEV-SNP).

The *Certifier Framework* employs platform elements that can isolate programs that are unforgeably identified by their measurement (usually a cryptographic hash), store secrets which are only revealed to isolated instances of that program, and attest to the isolation and identity of those programs. A "program" can consist of a single application (protected by, say, SGX, or operating within an otherwise fully protected environment) or an entire VM configured to run only identified and fully trusted programs under SEV-SNP. In the case of a VM, the "measurement" includes the entire OS image for the VM image, together with boot time configuration information, and all the programs can run in the VM in accordance with *security domain policy*; in this case, on Linux, the "measurement" of the VM consists of a cryptographic of the VM kernel and the concatenated initramfs, used at boot; this is essentially the measurement used by SEV-SNP[1].

[1] In practice, depending on the platform, other boot time software or arguments may also be included in the measurement like arguments provided to the kernel on boot or the trusted firmware or grub environment used to boot the kernel.

This description assumes the reader is generally familiar with *Certifier Framework* concepts. This includes the notion of isolation, measurement, certification, sealed storage, security policy, policy-key and attestation, as those terms are used in the *Certifier Framework*. The reader is also assumed to be familiar with the role of the Policy Server (*simpleserver*, below) in the *Certifier Framework*.

In scenario 1, the *security policy* for the *security domain* is specified using the *Certifier Framework*, by a *security domain* administrator, named Paul, here. Paul constructs this policy as part of the scenario example. In the *Certifier Framework*, each *security domain* is associated with a *policy-key*. This *policy-key* is an asymmetric cryptographic key pair consisting of a secret key (often denoted pk_{policy}) and a public key (denoted PK_{policy}). Abusing the definition, the term *policy-key* often refers to just PK_{policy}. In our example scenario, PK_{policy} is embedded in each Certifier protected program and is part of its measurement. Only the *security domain* administrator (Paul) has access to pk_{policy} and that key is only used as part of certification (i.e. – it is available to *simpleserver* and the utilities used to construct signed policy for this *security domain*).

For the remainder of this document, we use the terms *policy-key*, *security domain,* and *security policy* to refer to Paul's *policy-key*, Paul's *security domain* and Paul's se*curity policy* as constructed below.

Once Paul constructs the *security policy* and provisions *simpleserver* on the *deployment machine,* Paul runs *simpleserver* on the *deployment machine*.

Next, `cf_utility.exe` (provided with the *Certifier Framework*) is used on the *deployment machine* to get the deployment machine programs "certified" (in this case, the certification communications is conducted entirely within the *deployment machine*). Similarly, an instance of the *deployed VM* uses `cf_utility.exe` to get a new instance of a *deployed VM* certified (in this case, the certification communications is conducted over the internet).

During certification, each subject program on each platform generates a public/private key pair ($pk_{program,platform-instance}$ and $PK_{program,platform-instance}$). As a result of a successful certification, the policy server (*simpleserver*), relying on signed evidence including an attestation naming $PK_{program,platform-instance}$, produces a certificate signed by pk_{policy}, for the now certified program and transmits that certificate to the protected program. These certificates are called *Admission Certificates*.

Now, as described in the Certifier documentation, each protected environment can use its $pk_{program,platform-instance}$ to authenticate itself and sign statements. Only an isolated

program, in a protected environment, can access $pk_{program,platform-instance}$ (it is protected using sealed storage).

The recipient of such a certificate can rely that statements signed by $pk_{program,platform-instance}$ originated in a protected environment running the identified program which complies with *security domain policy*[2].

In scenario 1, there are two Certifier protected environments:

- The first environment is the *deployment environment* on the *deployment machine* which, as described, consists of a fully trusted Linux environment and associated storage. For simplicity, in this example, this environment is protected by the *simulated-environment* platform provided by the Certifier Framework (It could also be an SGX application or a fully protected VM under SGX but that would complicate the description and add little for a reader familiar with the Certifier.)
- The second environment is an SEV-SNP protected environment on a *deployed VM*. Such an environment can be instantiated on any SEV-SNP capable machine (say in "the cloud"), simply by booting the VM image provided by Paul on such a machine. This second environment is fully and unforgeably described by the SEV-SNP measurement made and signed on the SEV-SNP as part of *attestation*.

In scenario 1, the *deployment machine* is used to create the *security policy*, build the *deployable VM* image, and operate certification infrastructure which enforces the protections afforded by the *Certifier Framework*.

In our scenario 1, the *deployment machine* generates, stores and securely transmits application secrets (typically secret keys used to encrypt data that should only be used in environments that enforce security domain policy) that Paul intends to securely furnish to *deployed VM* instances in his *security domain*; the *deployed VM's*, as distributed, may know the names of these secrets, but the *deployed VM's*, as distributed, have no knowledge of the secrets themselves. We use the *Certifier Framework* on the now certified *deployment machine* (which does know the secrets) to securely provision the secrets to a (certified) *deployed VM* and protect those secrets as they are used and stored in *deployed VM instances*.

To do this, we use two programs, furnished with the *Certifier Framework*, *keyserver*[3] (on the deployment machine), and *keyclient* (on the *deployed VM*).

---

[2] You should understand this if you understand the *Certifier Framework,* but you'd be forgiven if you failed to fully understand this otherwise.

[3] *Keyserver* is actually `cf_key_server.exe` and *keyclient* is actually `cf_key_server.exe`. But we will refer to them as keyserver and keyclient usually.

As a result of certification, *keyserver* has an Admissions Certificate (for the deployment environment), signed by the *policy-key*, naming $PK_{deployment-machine,instance}$, as well as access to the corresponding $pk_{deployment-machine,instance}$. Similarly, *keyclient* has an Admissions Certificate (for the VM), signed by the *policy-key*, naming $PK_{deployed-machine,instance}$, as well as access to the corresponding $pk_{deployed-machine,instance}$. Both *keyclient* and *keyserver* have access to the verified PK_{policy}, because its self-signed certificate is embedded in their respective programs and is part of their measurements. Consequently, each program in the *security domain* can validate another program in the *security domain's* Admissions Certificate. Using TLS with client auth, the self-signed policy Cert, the Admissions Certificates and each program's generated private key, *keyserver* and *keyclient* can establish an encrypted, integrity protected TLS channel. Only *keyclient* and *keyserver,* in the certified environments, can read subsequent communications over that channel[4]. Having done this elaborate dance, application secrets (or any sensitive data) can be provisioned over the secure channel[5].

Phew!

This background enables us to describe the entire scenario 1 procedure in detail below.

Scenario 1

To recap, Paul has a *deployment machine,* at home, and wants to build one or more *deployed VM*'s that can run securely on any machine anywhere under SEV. Paul wants to provide some "secret" material (e.g., keys) on the *deployment machine* to certified instances running on *deployed VM's,* in a confidential, integrity protected, authenticated manner. In our scenario, once Paul builds the *deployed VM* image and the image runs on an SEV-SNP protected machine, he need not specify or provide any further protected information; further, he does not know where a *deployed VM* instance may run in the future. Paul and the programs in his *security domain* must also be able to provision additional new, previously unidentified, secrets, or rotate existing secrets, any time in the lifetime of a *deployed VM* without changing the VM. Finally, Paul wants to develop (or let others develop) new *deployed VM* images (that comply with Paul's *security policy*) and allow both previously existing images and future images to get secrets without rebuilding or redeploying existing VM images or existing data.

Paul can do this secret provisioning (as well as protect distributed programs) using the *Certifier Framework*. We describe the steps below. However, we should note that in a real application,

[4] And that the communications have not been modified in transmission. Again, TLS with client auth.
[5] In practice, once the secure channel is established and channel keys negotiated, a client will usually interact with a server using an API exposed by server on the channel. As a result, this mechanism provides a rather general method for secure interaction which is compatible with most internet-enabled service API's.

Paul may want to do a bit more. He should make sure he has an accessible backup of any secret material (the stuff in `client.in` below), in case his deployment machine breaks or is corrupted. In fact, the procedures we described here is illustrative and does not demonstrate all the operational procedures (like the one mentioned in the previous sentence) that Paul will implement in a real application. *However,* the mechanism we describe should be compatible with any steps Paul might want to take to build a resilient application without making any changes to the core of this procedure. Finally, for simplicity, in this procedure, we assume the *deployment machine* is absolutely trustworthy.

The steps described here are mostly implemented in shell scripts provided with the *Certifier Framework*, in `$(CERTIFIER_ROOT/vm_model_tools/examples/scenario1`. The instructions for using these scripts are in the file `instructions.md` in that directory.

Procedure

0. If you have not already built the certifier or its utilities, build it with the script build-certifier.sh. This is done in the `build-certifier.sh`.
1. Next, we generate the public-private key pair, PK_{policy}, pk_{policy}, described in the overview, and the corresponding self-signed certificate, $Cert_{policy}$, (generated by `cert-utility.exe`). The files generated by this step are:
   a. `policy_key_file` (see the warning below)
   b. `policy_cert_file.domainname` which contains $Cert_{policy}$
   c. Files used to support the simulated enclave (not needed in the VM)

   Warning: Never, ever, copy the `policy_key_file` into the VM (or anywhere else). This is the private policy key and should only be accessible to the administrator.
2. Next, we need to prepare the keys and programs needed to simulate sev in the test environment and generate and copy key material needed by the simulator. This is done in the `build-sev-sim.sh`. These scripts also generate certificates for corresponding ark, ask and vcek keys that normally come from the SEV platform.
3. Next (for real SEV), we build the *deployable VM* image. This produces a loadable kernel and initramfs (constituting the final deployable VM). Paul can upload this image to any machine he wishes, at this point or any point later in the procedure. This is done in `build-vm.sh`.
4. Because both the *deployment machine* and *deployable VM* must be certified, they must be "measured" using whatever "measurement" is required at runtime. In our example, since the simulated enclave on the *deployment machine*, this is simple; it measures `cf_utility.exe`. For the test environment, the same measurement (a fixed value in

the SEV simulator) is used on the *deployed machines*. The script `measure-programs.sh` does this. For a real VM, we must measure the constructed *deployable VM* along with some of its boot time parameters. The script `measure-vm-programs.sh` does this. The measurement for `cf_utility.exe` goes in `cf_utility.measurement` and the measurement for this program for the simulated environment goes in `sev_cf_utility.measurement`. The "real" sev measurement is typically in `pauls_vm.measurement`.

5. Next, we build the *security policy* using the measurements he obtained in the last step; we also use either the real or fabricated ark cert for the *deployed machines*. This is done inn `build-policy.sh`.
6. Next, we need to copy files, programs and keys into the locations required for execution. This is done in `copy-files.sh`. For the VM, additional files are copied in `copy-vm-files.sh`. Among the files copied are $Cert_{policy}$ includes `cf-utility.exe`, `keyserver.exe`, and `keyclient.exe`, as well as shell scripts. In the VM, Paul must also copy all the trusted programs he plans to run in the *deployable VM* instances; this includes `cf-utility.exe, keyserver.exe, and keyclient.exe`, as well as shell scripts, data files and libraries that the VM instance. Usually these are copied into initramfs[6]. Among these shell scripts are the are the ones that initiate certification (by calling `cert-utility.exe`) and the ones that start *keyclient* to get the application secrets, these scripts will include the IP address of Paul's *deployment machine* and the two port addresses on the deployment machine for *simpleserver* and *keyserver* so that the *deployed machines* can get certified and communicate with *the deployment machine's keyserver* during secret provisioning [7].
7. We must now run *simpleserver* in the *development environment*. *Simpleserver* will wait to be contacted by machines requiring certification. This is done in `run-policy-server.sh`.
8. Next, we need to get the *deployment machine* certified, by contacting the policy server. This is done in `certify-deployment-machine.sh`.
9. Next, we need to generate a sample application secret on the *deployment machine* and putting it in *cryptstore* on the *deployment machine*. The secret is originally in a file called "client.in" and this secret is put into *cryptstore* using the *keyclient* utility on the deployment machine. This is done in `generate-and-store-secret-for-deployment.sh`. The *cryptstore* and *policystore* files are encrypted on both the *development* and *deployed machines* and can be stored on any filesystem whether it is otherwise protected or not.

[6] Alternatively, Paul can statically link the application with the libraries before copying them.
[7] There are other ways to provision this information to the deployed VM's but we will stick to this for simplicity.

10. Next, we need to run *keyserver* on the *development machine* so it can respond to requests for application secrets. *keyserver* incorporates the *Certifier Framework* and uses secrets corresponding to "resource names"; both the secrets and corresponding names are in *cryptstore*. This is done in `run-deployment-keyserver.sh`. *Keyserver* retrieves the application secrets from *cryptstore* on the *deployment machine* and transmits them over a secure channel to the *deployed machines*.

Note: The key names being requested are arguments to *keyclient* in the deployed machines, initially these names will be in the scripts Paul provisions.

11. When a *deployed machine* starts, one of the shell scripts provisioned by Paul, calls `cf-utility.exe` to get certified on the *deployed VM* instance. This produces an Admissions Certificate for the *deployed VM* instance naming its measurement and $PK_{program,platform-instance}$. This certificate is obtained from *simpleserver* and is securely stored in the *policystore* and the *cryptstore*. For the test environment, this is done in `certify-deployed-machine.sh`. However, the VM builder must do this in the *deployed VM* instance at startup; he can copy the relevant commands from this script.
12. Now we need to run *keyclient* on the *deployed machine* to obtain the secret we want. This is done in the new script `obtain-application-secrets.sh`. Again, Paul will have provided shell scripts in the *deployed VM* to run *keyclient* on the *deployed VM* instance after certification and can use `obtain-application-secrets.sh` as a template for writing that script. *Keyclient,* invoked by this script, will store the application secrets in its *cryptstore*. Note: Both *cryptstore* nor *policystore* are encrypted and integrity protected so they can be on saved on any storage; ***initramfs should not be modified in a real sev deployment***. *keyserver* on the deployment machine transmits the application secret on the *deployment machine* in the protocol between *keyclient* and *keyserver* over a negotiated, authenticated, encrypted channel between *keyclient* and *keyserver*. Thereafter, the application secret (or any other secret in *cryptstore*) can be retrieved from *cryptstore* on the *deployed machine*. The *keyclient* utility, using the right arguments[8], can do this as can `cf_utility.exe`, using the *get-item* option or by using keyclient with the associated *cryptstore* or programmatically with the certifier routines used by *keyclient*. The application secrets are thus available on the *deployed VM*[9] and Paul has accomplished his goal.

[8] An example doing this appears at the end of the `test-script.sh` file in …/`scenarios/examples`.
[9] Of course, care must be taken to make sure the decrypted application secret (which is written to a file) is not visible outside the *deployed VM.* One can also a programmatic interface in *Certifier Framework* supplied routines retrieving the keys directly from *cryptstore*.

This procedure has been largely automated using shell scripts and the complete instructions for this are now in `instructions.md` in `…/scenarios/examples`:

Running the tests, are considerably simplified by a new consolidated script, `run-test-scenario1.sh`, which carries out all these steps; a flag, `-tt`, distinguishes between the test environment and the real SEV in the consolidated test. See `instructions.md` for more details about `run-test-scenario1.sh`.

Variations on Scenario 1

Here are some variations on Scenario 1.

Suppose Paul doesn't want to use the deployment machine to provide the services named above. Paul simply uses the same mechanism to provision one or more SEV protected cloud VMs with application secrets and policy allowing it to provide the same functions provided in Paul's *deployment machine* in Scenario 1 using the very same software.

More sophisticated versions of *keyclient/keyserver* can impose additional authentication (say by using *acl-lib*) to provide more granular key distribution.