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. `instructions.md` can read in conjunction with this document for clarification. In fact, `instructions.md` contains step by step instructions for all this.

Procedure

1. On his *deployment machine*, Paul uses *Certifier Framework* utilities (e.g., `cert-utility.exe`) to generate the public-private key pair, PK_{policy}, pk_{policy}, described in the overview, and the corresponding self-signed certificate, $Cert_{policy}$, (also 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. Paul copies $Cert_{policy}$ to initramfs, so it is part of the *deployed VM*'s SEV measurement when it runs. Paul copies 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 will use 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]. A list of service files that

[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.

should be copied (in the test environment) appears in step 9 in `instructions.md`. The files copied in this step differ in the test case and in the VM case. The new script described below, `copy-vm-files.sh`, in the next section should be authoritative[8]. Among the files copied are:

   a. The policy cert, typically in the file `policy_cert_file.`*`domainname`*
   b. For the simulated sev platform but NOT for real sev, simulated sev certs must be copied. For the simulated enclave (and no others), the enclave's platform cert, `platform_key_file.bin`, and attestation key, `attest_key_file.bin` are copied.
   c. For real SEV, the installed scripts must be able to access the platform's "ark," "ask" and "vcek" certs.
   d. For a VM, you must copy the certifier libraries and utilities including: `cf_utility.exe, cf_key_client.exe, cf_key_server.exe`; the list in `vm_copy_files.sh` is authoritative.

3. Paul now builds the final *deployable VM* image resulting in a loadable kernel and initramfs (these constitute the final deployable VM). Paul can upload this image to any machine he wishes, at this point or any point later in the procedure.
4. Next, Paul uses steps illustrated in the older test script `prepare-test.sh`. It builds a *security policy* using the measurements he obtained in the last step. The script uses *Certifier Framework* utilities like "`make-vse-unary-clause.exe`" to build the security policy for *simpleserver.* These utilities, which builds the policy files are not needed in the VM. **Important note**: again, `prepare-test.sh` only correctly does this for the test environment and the script need to be modified for real SEV. See the instructions at the end of this section and in `instructions.md` for a real sev environment. The new test script described below works in both environments. The "real" sev measurement is typically in `sev_cf_utility.measurement`, and is used to build policy.
5. 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. For the VM, Paul "measures," using something like *virtee*, his constructed *deployable VM*. These measurements are needed when constructing the policy *simpleserver* uses during certification below. The new test script described below works in both environments.

[8] Although the new script always works, there are alternatives. For example, some of these files could, instead, be passed in as measured parameters at boot. Here we stick with the initramfs solution since it is simplest to explain (although not always the most convenient. In fact some of the new scripts describe an alternative.

The file `sev_policy.bin`, generated in step 4 is needed by simpleserver; it contains the consolidated policy; it is not needed in the VM.

6. Paul next runs *simpleserver* on his *deployment machine* pointing to the policy he just built. For the legacy test scripts, this is illustrated in the `test-script.sh`. *Simpleserver* will wait to be contacted by machines requiring certification. The new test scripts described below works in both environments.
7. Paul will call the appropriate scripts on the *deployed VM*, when it first starts on a new SEV platform, to use `cf-utility.exe` to contact the *deployment machine* on the *simpleserver* port to get certified[9]. The new test script described below works in both environments.
8. Next, Paul will generate a sample application secret on the *deployment machine,* putting it in the *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. *Keyserver* retrieves the application secrets from *cryptstore* on the deployment machine. *Cryptstore* for distribution to certified requestors in the security domain; *cryptstore* can only be read using secrets available after certification using an intermediate symmetric key in the *policystore*. The *cryptstore* and *policystore* files are encrypted and can be stored on any filesystem whether it is otherwise protected or not. This is illustrated for the legacy scripts in step 11 in `instructions.md`. The new test script described below works in both environments.
9. Next, Paul runs *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*. The new test script described below works in both environments.

Note: In the test environment, prepare-test.sh builds "fake" SEV ark, ask and vcert certificates and the corresponding private keys for the simulated SEV test; in a real SEV environment, Paul's startup shell scripts will need to import those certificates from the VMM or wherever they are stored on the SEV machine as part of step 10 below. Stgeps 10 and 11 are carried out on *deployed VM* instances. They are initiated by a shell script provisioned by Paul in a previous step. The key names being requested by *keyclient* are arguments to *keyclient*.[10]

[9] This certification can only happen after Paul constructs the certification policy on the deployment machine and runs *simpleserver* on his deployment machine, as described below.

[10] Step 4 can be omitted if you use `cf-utility.exe`, on the deployment machine, to generate keys. Even if you don't use `cf-utility.exe` to generate the application secrets, you can use the `cf-utility.exe` (using the *put-item* function) to put keys in *cryptstore* without using *keyclient* provide *keyserver* on the same machine the generated secrets

10. When a *deployed machine* starts, one of the shell scripts provisioned above by Paul, calls `cf-utility.exe` to get certified on the *deployed VM* instance. This produces the Admissions Certificate for the *deployed VM* instanced, obtained from *simpleserver*, which is securely stored in the *policystore* and the *cryptstore*. This is illustrated for the legacy test in step 11 in `instructions.md`. The new test script described below works in both environments. Note however that the VM builder must do this in the deployed VM instance at startup. Consult the new test script description below.
11. In step 2, Paul will have provided shell scripts in the *deployed VM* to run *keyclient* on the *deployed VM* instance after certification (step 10) contacting the *keyserver* on the *deployment machine* using the identified *keyserver* port to get the application secrets it needs. *Keyclient,* invoked by this script in the *deployed VM* instance, will store these application secrets in *cryptstore*. Both *cryptstore* nor *policystore* are encrypted and integrity protected so they can be on saved on any storage; <u>*initramfs should not be modified in a real sev deployment*</u>. For the test environment, this is illustrated in `test-script.sh`. Note however that the VM builder must do this in the deployed VM instance at startup. Consult the new test script description below. *keyserver* on the deployment machine returns the application secrets on the *deployment machine* in the protocol between *keyclient* and *keyserver* over a negotiated, authenticated, encrypted channel between *keyclient* and *keyserver*. This works because the application secrets were stored in the *deployment machine*'s *cryptstore* in step 8. *keyclient* on the *deployed machine* now stores the application secrets in the *cryptstore* on the *deployed VM* instance[11]. Thereafter, these application secrets can be retrieved from *cryptstore* on the *deployed VM*. The *keyclient* utility (using the right arguments)[12] can do this as can `cf_utility.exe`, using the *get-item* option. The application secrets are thus available on the *deployed VM*[13] and Paul has accomplished his goal.

This procedure has been largely automated using shell scripts and the complete instructions for this are now in instructions.md along with the "legacy" test instructions.

1. Step 1 can be carried out by `provision-keys.sh` in …`/scenarios/examples.`
2. Step 2 can be carried out in the by `copy-files-test-simulated-sev.sh` in …`/scenarios/examples` for the test platforms and `copy-files-test-sev.sh` in …`/scenarios/examples` for the test platforms. The user will have to copy the

[11] Once the secret is stored in the deployed machine's cryptstore, the VM does not need to contact keyserver unless additional application secrets are needed. The keyserver protocol is used once when an instance is first started on a new platform.
[12] An example doing this appears at the end of the `test-script.sh` file in …`/scenarios/examples`.
[13] 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*.

applications and VM specific executable, scripts and files using a custom script; however, see file-notes.txt for instructions. For the test example, this is generally not required.

3. Step 3 is illustrated in `build-application.sh` in the test environment (but doesn't require a VM). For real sev, we must construct a custom script called `build-sev-vm.sh`.
4. Step 4 is carried out by the platform specific shell scripts `measure-simulated-enclave.sh`, `measure-simultated-sev.sh` and `measure-sev.sh`.
5. Step 5 can be carried out using `build-policy.sh`, which takes the enclave type and measurement as an argument.
6. Step 6 can be carried out by invoking `run-policy-server.sh`.
7. Step 7 can be carried out by invoking `certify-deployment-machine.sh`.
8. Step 8 can be carried out by invoking `generate-and-store-secret-for-deployment.sh`.
9. Step 9 can be carried out by `run-deployment-keyserver.sh`.
10. Step 10 can be carried out using the script `certify-deployed-machine.sh`. This shell script must be incorporated in the shell files Paul puts in the *deployed VM* and causes to be invoked on the *deployed VM* when a VM instance first starts on a new platform.
11. Step 11 can be carried out by `obtain-application-secrets.sh` which is invoked in `test-script.sh` for the test environment. For a real SEV VM, this accomplished in a script Paul puts in the *deployed VM* that is invoked after step 10 succeeds.

A shell script called `run-complete-test.sh` carries out these 11 steps in a test environment. For real SEV, a shell script called `run-complete-sev-test.sh` carries out these 11 steps when called with the right arguments.

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.

Appendix -Using *keyclient* and *keyserver* to provision keys in the test environment

keyclient and *keyserver,* in the `$CERTIFIER_ROOT/vm_model_tools/src` directory, are programs which use an existing *cryptstore* initialized by `cf-utility.exe`, for example, one that results from the following calls in `$CERTIFIER_ROOT/vm_model_tools/examples/scenario1`:

```
      ./prepare-test.sh fresh dom0
      ./prepare-test.sh all dom0
      ./run-test.sh fresh dom0
      ./run-test.sh run se dom0
```

This sequence of calls will certify the VM and result in the files `policy_store.dom0` and `cryptstore.dom0` as well as support files in the subdirectory `cf_data`. *keyclient* and *keyserver* will not run unless the domain has been initialized and certified, as above. *key-server* stores saved values in `cryptstore.dom0` on the machine it runs on. `cryptstore.dom0` is encrypted using a key sealed using *Seal* in the enclave. Items in *cryptstore* can be exportable or not. *key-server* will not return items which are not exportable (like the private authorization key for the domain).

keyclient makes a request over a secure channel to *key-server* on the same or different machine. It can request either that *keyserver* store a new secret in `cryptstore.dom0` which *key-client* transmits in a request or that *keyserver* retrieves a secret in `cryptstore.dom0` on the *keyserver* machine. Here is an annotated set of argument (there are more default arguments in each program.

```
      $CERTIFIER_ROOT/vm_model_tools/src/cf_key_server.exe
        --policy_domain_name=dom0
```
This specifies the domain name.
```
        --encrypted_cryptstore_filename=cryptstore.dom0
```
This is the location of the cryptstore relative to data_dir specified below.
```
        --enclave_type=simulated-enclave
```
This is the enclave type.
```
        --policy_store_filename=policy_store.dom0
```
This is the location of the policy store relative to data_dir specified below.
```
        --policy_key_cert_file=policy_cert_file.dom0
```
This is the location of the policy cert relative to data_dir/cf_data.
```
        --data_dir=./
```

Keyclient has some additional arguments annotated below.

```
$CERTIFIER_ROOT/vm_model_tools/src/cf_key_client.exe
  --policy_domain_name=dom0
  --encrypted_cryptstore_filename=cryptstore.dom0
  --enclave_type=simulated-enclave
  --policy_store_filename=policy_store.dom0
  --policy_key_cert_file=policy_cert_file.dom0
  --input_format=raw, --input_file=client.in
```

For a store request, the secret to be stored is in the file specified by the argument to --input_file. If the format is "raw" it is just a binary blob; if the format is "cryptstore-entry", must contain a serialized cryptstore-entry. The input file name is not relative to data_dir.

```
  --output_format=raw, --output_file=client.out
```

For a retrieve request, the retrieved secret is written to the file whose name is specified ---output_file. If the format is "raw" it is just a binary blob; if the format is "cryptstore-entry", it is a serialized cryptstore-entry. The output file name is not relative to data_dir.

```
  --action=store
```

This is the action to be performed; it is either "store" or "retrieve."

```
  --key_server_url=localhost
```

This is the URL of the *key-server key-client* wishes to access.

```
  --key_server_port=8120
```

This is the port of the *key-server key-client* wishes to access.

```
  --data_dir=./
  --resource_name=key-client-test-key
```

This is the "tag" in cryptstore associated with the item to be stored or received.

```
  --version=0
```

This is the key version of the secret, if 0, the version is the latest on retrieval or the latest version + 1 for storage.

Here is a sequence of three calls consisting of starting *keyserver*, having *keyclient* ask to store a new key and then using *keyclient* to retrieve the key. *keyserver* runs as a tls service.

```
$CERTIFIER_ROOT/vm_model_tools/src/cf_key_server.exe \
   --policy_domain_name=dom0 \
   --encrypted_cryptstore_filename=cryptstore.dom0 \
   --enclave_type=simulated-enclave \
   --policy_store_filename=policy_store.dom0 \
   --policy_key_cert_file=policy_cert_file.dom0 \
   --data_dir=./ &

$CERTIFIER_ROOT/vm_model_tools/src/cf_key_client.exe \
   --policy_domain_name=dom0 \
    --encrypted_cryptstore_filename=cryptstore.dom0 \
    --enclave_type=simulated-enclave \
    --policy_store_filename=policy_store.dom0 \
    --policy_key_cert_file=policy_cert_file.dom0 \
    --data_dir=./ \
    --resource_name=key-client-test-key \
    --version=0 \
    --input_format=raw \
   --output_format=raw \
   --input_file=client.in \
   --output_file=client.out \
   --action=store

$CERTIFIER_ROOT/vm_model_tools/src/cf_key_client.exe \
    --policy_domain_name=dom0 \
    --encrypted_cryptstore_filename=cryptstore.dom0 \
    --enclave_type=simulated-enclave \
    --policy_store_filename=policy_store.dom0 \
    --policy_key_cert_file=policy_cert_file.dom0 \
    --data_dir=./ \
    --resource_name=key-client-test-key \
    --version=0 \
    --input_format=raw \
    --output_format=raw \
    --input_file=client.in \
    --output_file=client.out \
    --action=retrieve
```

Testing

Once performing the initialization above with `cf-utility.exe`, described above, you can run a test script by typing

```
    ./test_script.sh
```

In the `$CERTIFIER_ROOT/vm_model_tools/examples/scenario1` directory.