**Provisioning secrets with cf\_utility from a deployment machine to deployed VM’s**

**Scenario 1**

Paul has a deployment machine at home and one or more deployed VM’s running in the cloud under SEV. He wants to provide some material to the deployed machines in a confidential, integrity protected, authenticated manner. The deployed VM’s were constructed by him on a deployment machine.

**Procedure**

1. Paul uses CF utilities (cert-utility) to generate a public-private key pair, . He places the self-signed certificate, , (also generated by *cert-utility*) naming in the VM image he constructs (so it is part of the SEV measurement) along with the *cf-utility* program and a new program described below call *key-client*. should be accessible to programs on the VM. He also places the IP address of his deployment machine and two port addresses in the image. The first port will be the “*simpleserver*” port second port will be the “*key-server*” port on his deployment machine. He arranges for the deployed VM, when it first starts, to use *cf-utility* to contact his deployment machine on the *simpleserver* port to get certified and then have *key-client* on the VM contact his deployment machine on the *key-server* port to get the confidential material (probably a key). He instructs *key-client* to store this key securely on the VM, for example, by putting it in *cryptstore* (see below). In practice, the url of the deployment machine and the two ports will be specified in command line parameters in a script the VM runs when it first starts on an SEV machine in the cloud.
2. Paul “measures,” using something like virtee, his constructed VM as well as a measurement on his deployment machine[[1]](#footnote-1) (possibly using the simulated-enclave or SEV). and goes through the usual procedure, using *prepare-test.sh* in *vm\_model\_tools/examples/scenario1*, to construct a policy, for *simpleserver*, specifying these trusted measurements along with the other customary policy stuff. The complete policy will be in *vm\_model\_tools/examples/scenario1/server* on his machine and will include in that material. All this is scripted now.
3. Paul runs *simpleserver* on his deployment machine pointing to the policy. *key-server* incorporates the Certifier framework. Paul runs *key-server,* which contacts *simpleserver* on the same deployment machine to get certified using a private key, *key-server,* generates. He stores the private key, and the Admissions certificate, in a safe location (e.g.-*cryptstore*) accessible to programs on his deployment machine. Incidentally, names and the “measurement” of *key-server* and is signed by .
4. When a deployed machine starts, it uses *cf-utility* to generate the deployed VM’s generated public/private authentication key, , , as well as the Admissions certificate, , obtained from *simpleserver*, and stores them securely on the deployed VM (say in *cryptstore*).
5. Next, *key-client*, which has access to the private key, and corresponding certificate from step 4, opens a secure channel (using *secure-authenticated-channel* in the certifier) between *key-client* and *key-server* requesting the desired secret material, supplying the key name (see the *key-server* interface below) and stores the key securely, again, say in *cryptstore*. The secure channel is established using , and on the *key-server* end and , and all provisioned above. The material is sent to the secure channel in plaintext but is encrypted and integrity protected as it’s sent over the channel. *Key-server* has all the information it needs to grant the access request based on the exchanged certificates although it may do something fancier later (see below).

Paul has accomplished his goal.

**Scenario 2 (and variations)**

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 keys and policy allowing it to provide the same services Paul’s deployment machine in Scenario 1 using the very same software.

Paul need not use *simpleserver* to provide a signed certificate to *key-server*, he can simply sign certificates directly using the policy key (since he has the private policy key), but this involves writing a little more software. The above mechanism does NOT require additional software.

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

**Work to do**

Paul writes no new software and provisions no data (except the policy cert, url address and port numbers mentioned above) in his deployed VMs.

John will write *key-server*, *key-client* and also make sure *cryptstore* is suitable for any of the secure storage described above on the deployed machine. John thinks it will take about a week and that key-servers are probably generally useful.

**Key-server Interface**

The arguments to *key-server* are provided in a protobuf. The client supplies:

1. A domain relative resource name (e.g., datica-test/storage-disk key).
2. An action: “store” or “retrieve”.
3. Data (For example, a serialized key, if the action is “store”).
4. Credentials (a serialized certificate chain) [optional]

Return values are:

1. Return status (“success”, “fail”, “auth-challenge”)
2. Data (For example, the serialized key return if the action is “retrieve” or a nonce if authentication is required)

In the first implementation, all domain members can add or retrieve any key. However, future implementation will add granular access control using *acl-lib*.

1. See below for a slightly different way to do this. [↑](#footnote-ref-1)