CSC574 – Fall 2011 – Experiment Proposal

Ashwin Shashidharan and Jitesh Shah
{ashashi3,jhshah}@ncsu.edu

October 24, 2011

1 Overview

1.1 Problem Overview

In recent times there has been growing popularity for cloud providers as infrastructure suppliers. Cheap and easily scalable web-services at IaaS providers like Amazon stand testimony to this trend. But despite it's wide spread acceptance, the security of user data on the cloud remains a primary concern. A significant chunk of applications haven't yet found a place in the cloud due to lack of trustworthiness and data security in the cloud.

In general, we anticipate two broad types of attacks in the cloud: 1) Attacks of dynamic nature where the attack vector is aimed at the integrity of a running machine, for e.g., buffer overflows, CSRF, etc 2) Static attacks that compromise elements of our trusted computing base making the basis of further deployments vulnerable, e.g., malware infecting executables.

We address the latter by implementing a security model that protects us from attacks of static nature.

1.2 Solution Overview

We begin this section by drafting a few requirements that the cloud provider must guarantee for our security scheme. First and foremost, we expect the cloud provider to use a Trusted Platform Module(TPM) as its basis for a root of trust. The provider must then use the TPM to setup a trusted computing base that insures against malicious software components. Specifically, we extend our chain of trust beginning from the BIOS of the machine, through the Virtual Machine Monitor (VMM) up to the VM launcher. This extended chain of trust is used each time to verify integrity of a component before a launch.

The launcher uses a CA-signed certificate to prove authenticity to the client. The hash of the extended Platform Configuration Register (PCR) in the TPM is used to protect the private key. Such a use of the TPM prevents malicious modifications of the launcher so that the launcher cannot be hijacked to steal client data.

To facilitate this process, we modify the launcher's functionality to use the host TCP/IP stack. Once the client trusts the launcher, it sends the kernel decryption key and data decryption key using a secure socket connection to the launcher. This kernel decryption key is used to decrypt the kernel in memory and boot the virtual image. At this point, the launcher hands over the data decryption keys to the image of the client's choice running as a trusted VM in the cloud. Data encryption and decryption is then handled by the client's running OS. To enable encryption/decryption we use symmetric key encryption for our data. The key exchange process happens at the beginning and only once over the lifetime of the VM.

To establish a proof of concept we use the following software components in our project:

1. TPM emulator by BerliOS (`http://tpm-emulator.berlios.de/`)
2. KVM (`http://www.linux-kvm.org/page/Main_Page`)

2 Methodology

1. The first step is to generate the SSL certificate for use in authentication of the cloud provider (Client generates one on its side too). In practice, this certificate can be obtained from a trusted third party like Verisign. In our case, we generate a self-signed certificate using *openssl*. Thus we have an RSA private key of 2048-bits and a public self-signed certificate.

2. The next step is to configure the TPM to load only the trusted copy of the launcher (qemu-kvm binary, in our case) and protect the private part of the SSL key. This involves the following steps:

   - Calculate the hash of the launcher binary and store the result in a file.
   - Say PCR#16 of the TPM has the extended hash for (ROM, BIOS, VMM). Extend PCR#16 using the launcher binary. PCR#16 now contains extended hash for (ROM, BIOS, VMM, Launcher).
   - The next step is to protect the private key generated in Step (1). TPM_SEAL operation using PCR#16 does the job. It will encrypt the private key using the value in PCR#16. This private key can be decrypted only if the exact same sequence of steps is followed to boot the VMM, thus establishing trust.
   - Delete the original copy of the private key and keep the encrypted copy. Only the trusted launcher binary can now unseal the private key and hence, establish communication with the client.
   - Protect the encrypted copy of the certificate using appropriate access controls (preferably SeLinux).

3. When the launcher starts executing, it will decrypt the private key using the TPM_UNSEAL operation.

4. It then initiates an SSL connection with the client site and does a *mutual* authentication over SSL.

5. The client sends over the decryption key for the kernel and the decryption key for the data over the SSL connection. The kernel decryption key is used to decrypt the kernel in-memory.

6. Once the kernel is read into memory the launcher can resume original operation. When the kernel boot is complete, the launcher can pass over the data-decryption key to the kernel using host-guest communication mechanisms (typically, a shared page). To restrict the scope of our project, we plan to compile the data decryption key in the kernel itself. This is safe since the kernel image is stored encrypted.

7. The key is only cached in-memory at the cloud-site, never persistently stored. Thus, avoiding leakage of data in-case the data on the disk is stolen.

3 Experiment Hypotheses

3.1 Hypothesis 1

Our scheme is safe against the threat model described earlier

- With TPM as the root of trust, the trust is extended to the VM launcher. If a malicious attacker plants a malware either in the BIOS, VMM or the launcher, the private key for SSL would not be decrypted (since the hash in PCR#16 will be different). Hence, the launcher won't be able to access the kernel/data decryption keys.
- If an attacker manages to steal a copy of the client data, it is encrypted and hence, protected.
- If an attacker manages to break into the cloud provider and possibly has shell access to the VMM, there is still a missing part to the puzzle: kernel/data decryption keys. Without these, the attacker cannot access client data. The client divulges those keys only to authenticated copies of the launcher, so stealing the keys is hard as well.
- The attacker cannot possibly replace the copy of the guest kernel with a malicious once, since it is stored encrypted and the key is on the client's servers.
- Security can be enhanced using SeLinux so that users logged in via remote shell on the VMM cannot access the TPM device (i.e., TPM device can only be programmed with physical access to the server) or the encrypted copy of the private SSL key.
- The only reasonable attacks are online attacks on the launcher. The launcher can be hardened to make these attacks infeasible since it has access to the network for just one pre-defined job: establish an SSL connection and get the decryption keys. Root-kits on the VMM can be avoided by building the VMM with kernel modules disabled.
- A possible attack would be to attack the guest VM directly. These can be mounted whether or not we offload the encryption/decryption of data to the cloud, so we don't consider these attacks in our discussion.

3.2 Hypothesis 2

The performance degradation is negligible

Assuming the client originally used to encrypt data at its site, offloading the encryption to the cloud causes no further performance degradation. Otherwise, the performance degradation is equal to the performance degradation by the encryption software used on the guest VM. Our project doesn't impose the use of a particular scheme.

There is a slight performance degradation while booting a guest VM. The performance degradation comes from three sources: decryption of the private key by TPM, getting the kernel decryption key from the client over SSL and actually decrypting the kernel image. We argue that booting a guest OS is a sufficiently rare phenomena so that such an overhead is acceptable.

3.3 Hypothesis 3

The core idea is platform agnostic

The only change needed at the cloud-side is a modified launcher binary and support for an encrypted kernel. The protocols used in the implementation - SSL, TPM 1.2, symmetric key encryption/decryption - are all

platform agnostic and widely implemented. We impose no restriction on the type of encryption used on the guest OS. So, this scheme can be adopted on a wide range of platforms.