Singularity Design Note

38

TPM Support in Singularity

This note describes the support for secure boot and software authentication to Singularity

1. Introduction

The Trusted Platform Module (TPM) is a motherboard-attached crypto-processor that is becoming common in PC platforms.[*] TPMs create or are furnished with a public/private key pair during manufacture, and this key can be used to authenticate the hardware via public-key cryptography. More interestingly, this key can also be used to authenticate the *software* running on the platform, and once so bootstrapped, the software can be trusted to make more sophisticated security claims.

The TPM model of software authentication is to provide a secure log of interesting actions that can be used to build up a description of the running software. On compliant platforms (which should be all platforms shipping with a TPM) the BIOS performs the first few steps. In particular, the BIOS boot sector hashes the BIOS and records the hash value in a TPM log. The BIOS then records other aspects of platform configuration into other logs. Finally, the BIOS hashes, records, and then executes the boot sector (disk or PXE).

The operating system takes over at this point. In order to continue the "chain of trust" the OS loader and kernel can record the hashes of the modules and data that it loads and so on up to a functioning operating system. The platform firmware performs simple program hashes, but once OS code is running, more sophisticated boot policies may be enforced, although they still have to be encoded *usually by hashing) into 20 bytes to be consumed by current versions of the TPM.

Once boot is complete, the TPM has recorded within internal log-registers a more or less sophisticated description of the software and software configuration of the platform.

The TPM can use the contents of these registers in local access control decisions. One of the more well known example of this is "sealing." The TPM "seal" primitive lets OS or application software access-protect data so that only certain configurations (as reflected in the TPM logs) can "unseal" it.

The TPM can also sign statements avowing to current log register settings (and hence currently running software). This functionality is commonly known as attestation, and is supported by several TPM primitives, the most famous of which is called "Quote."

In addition to the secure-boot procedure we have just described, in order for any form of software authentication to be believable the software must be able to protect itself against unauthorized modification. Since the primary design goal of Singularity is dependability, we believe that Singularity is an ideal platform for exploring this security technology.

Our initial in goal is to provide TPM-based services that let a SIP authenticate itself and other aspects of system state to third parties. This supports the following scenarios:

- Authentication of the Singularity kernel to support trustworthy kiosks
- Authentication of a Singularity-hosted transaction-authorization program to a web service
- Digital rights management

However, the details of the scenarios are less important than the fact that they force us to design end-to-end TPM support without being distracted by the dozens of additional features offered by the TPM.

To achieve these goals the following features must be added to the Singularity system:

1) An MBR that authenticates the singularity loader
2) Modifications to the singularity loader to integrity check the code and data that it loads
3) A device driver for the TPM
4) A service that multiplexes and virtualizes the TPM for application access
5) Libraries/interfaces to provide application-level access to the TPM
6) Integration with the existing Singularity security system to allow the TPM to be used to report on application and OS state

These feature areas are described in more detail later in this design note, but we start with an overview of the TPM.

2. Introduction to the TPM

The TPM is a low-cost and low-performance motherboard-attached crypto-processor that is becoming common on PC clients and servers. The TPM is a device designed by a committee, and it's features and structures reflect that fact rather too plainly. However, Microsoft has had a fairly definite strategy in the features that we have designed and submitted. The main design principal that we have used is that the TPM should not be a separate programmable device, but instead should provide services to help the platform do things that software does poorly or not at all. This features that we have designed and advocated have included:

- Hardware authentication
- Software authentication
- Protection of stored state against offline attacks (reading, modification, rollback)
- Better protection for cryptographic keys
- RNG, access-protected NV storage, monotonic counters, privacy and other controls

For the purposes of a brief introduction to motivate the rest of this design guide we will sketch the capabilities that support the first two features.

TPMs are supplied with an RSA key pair called the *Endorsement Key* (EK) that uniquely identifies the TPM. Some vendors may also furnish certificates for the endorsement key indicating what sort of TPM and what sort of platform the key is associated with. The private key is never revealed, but can be used to decrypt as part of a cryptographic protocol to authenticate the platform. Most commonly this key is used to obtain a second key (an *Attestation Identity Key*, or AIK) , that is used for day-to-day authentication (signing and "quoting.").

The secure boot mechanism supported by the TPM was sketched in the first section. In this section we describe how the TPM is used to log software, policies, and other data so that they can be used for local access control decisions and remote authentication.

Software loaded into the TCB during boot results in irrevocable changes to the state of the platform: the changes cannot be reliably undone until the platform is rebooted. To capture this we must provide a logging mechanism that has the characteristic that logged items can only be removed on reboot. Flexibility implies that the logs should be large, but since we want the TPM to be cheap we chose: a) have the TPM integrity protect log entries kept externally by only logging the hash of the entry in the TPM, and b) have the TPM maintain the cumulative log by means of a hash-chain. The operation that supports this is called *TPM_Extend()* and the

registers that support the logs are called *Platform Configuration Registers*, or PCRs . Some additional flexibility is afforded by providing 24 PCRs, and compliant platforms use some of these for defined purposes (appendix A).

As previously indicated, we designed most of these registers and the logging mechanism to reflect the security transitions that occur when a platform boots. However, once enough software is running to establish its own protection domains, it is no longer necessarily the case that loading and running a program results in an irreversible change to the TCB. To capture this (but still within the semantics of PCRs), we provide a few resettable PCRs whose contents can be controlled by the TCB. We describe how these can be used to represent the identity of a calling program in section *.

3. Secure Boot of Singularity

The table in Appendix A summarizes PCR usage in compliant PCAT (non-EFI) platforms. The first 6 or 7 PCRs are used to record BIOS-boot of the platform. PCR18 is used in LaGrand systems (Intel) to record the hash of the hypervisor loader (but we do not plan to use this facility in the initial prototype).

Operating systems and applications are free to use the other PCRs (TPMs are required to provide 24 PCRs numbered 0 through 23). We will use PCR8 and PCR23, and we need to be aware of the platform use of PCR4. These PCRs are used as follows:

PCR4
The BIOS records the hash of the IPL (initial program load code) in PCR4. The IPL is (most of the) MBR, ISO boot image, or PXE bootstrap program.

Implementation note: We have not been able to obtain a TCG compliant PXE implementation from Intel. We expect there will be one in the near future.

PCR8
We will use PCR8 to record the singularity kernel boot policy.

Implementation note: We have agreed that there is only one bit of security related configuration state: whether or not a kernel debugger is attached. We haven't implemented PCR8 usage for now.

PCR23
We will allow application access to the TPM, but Singularity will control the contents of PCR23. Applications can request Singularity load representations of system and application state into this register.

For the initial implementation the kernel boot policy loaded into PCR8 will be the hash of the concatenation of the files loaded by the Singularity boot loader (singldr) as listed in SingBoot.ini. To do this we will use the BIOS-provide (INT 1A) HashLogExtendEvent(PCR8, HashValue) function (perhaps with a supplemental SHA implementation in the loader, if this proves to be significantly faster).

Implementation note:

Accessing TPM BIOS functions requires a TCG compliant BIOS. All TCG compliant BIOSes we have evaluated do not have legacy PNP support to enumerate devices and cannot boot Singularity. Once a full ACPI parser is implemented into Singularity a secure boot can be achieved in a matter of a few days.

As Singularity evolves we expect more sophisticated boot-policy descriptions will be required. One ongoing challenge will be management of operating system state: in the current implementation of Singularity all security-relevant state is generated at compile time, but this unlikely to remain true for ever.

If Singularity moves towards a model where a system administrator can make security-relevant configuration changes to the trusted computing base (at run time, or to be enforced on the next reboot) we will either need to capture these changes in the OS load policy (feasible if the changes are simple and few), or move towards a

model where the recorded boot policy names the identity of the administrator (for example, her public key) rather than the actual OS boot policy. In the latter case we need to protect persistent storage against offline modification, but are less concerned with matching bits read at boot time with a static policy.

4. The TPM Software Support Stack

The software and interfaces necessary to support the TPM are shown in figure 1 and are described in more detail in the sections that follow. The channels and contracts between the driver and the two services are private, but the application interface presented by the TPM service is designed to be safely used by any application.

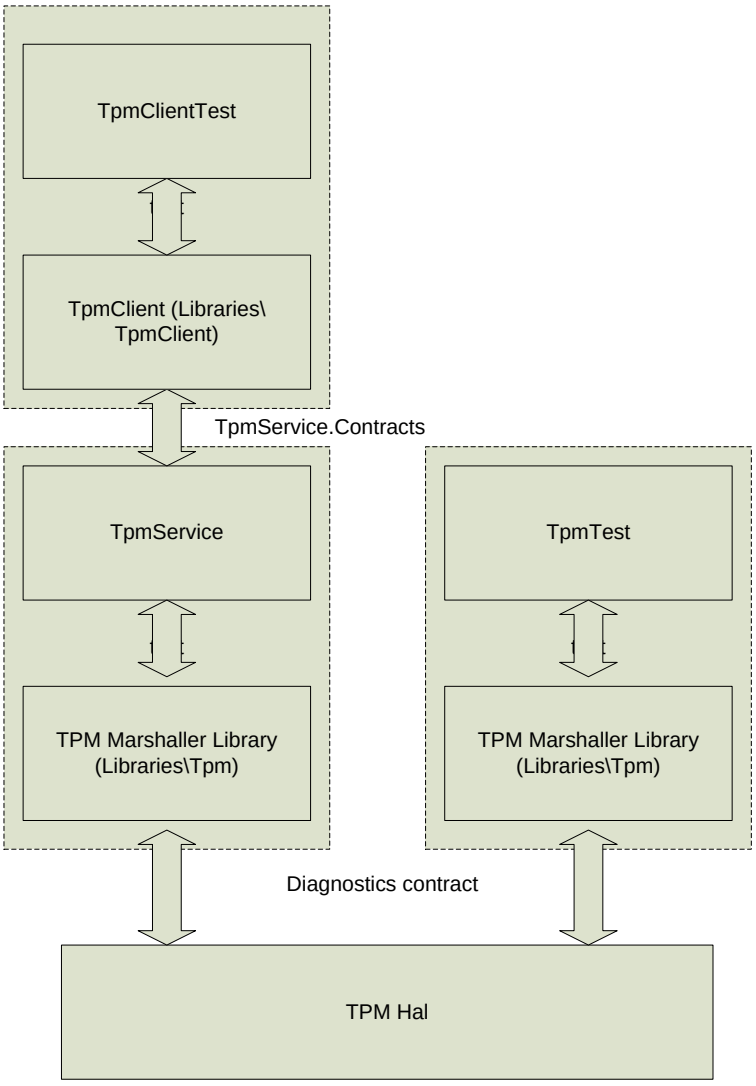

Figure 1: TPM software stack

The TPM service performs low-level TPM resource management. The primary resources that must be managed are key and authorization session "slots." The TPM has a few of each, and simultaneous access by

more than one program (or even a complicated program on its own) can exhaust the supply. The TPM provides commands to export key- and session-data in encrypted form, and this is designed to be used by a resource manager to ensure each command issued by each application has the resources needed to execute. This is discussed more in section 6.

5. The TPM Device Driver

The security model demands that the TPM device driver be part of the TCB, and that TPM hardware resource access must only be granted to the authorized device driver. This will naturally be the case in the boot model described above.

The TPM is a relatively simple device to control at the hardware level, but rather more complex at the command level (section 6). At the hardware level, the TPM is located at an architecturally defined physical memory location (0xFED4_0xxxh, and nearby addresses). The address range decodes to TPM control registers and to a FIFO used for sending and receiving command and response byte streams.

TPM command completion can be determined by interrupt or polling. The TPM is a low-performance and likely infrequently used device, so polling will suffice.

We have integrated the TPM driver into the Singularity HAL as a HAL device (implemented in Singularity.Hal.Tpm). All other HALs (APIC, Legacy, Enlightened) instantiate the device in their respective HalDevices.cs.

A sketch of the interface is shown below. Only one instance of the TPM device is constructed, and only the TPM service should be able to bind to it. In addition, only one command can be outstanding, and the caller is responsible for creating a byte-stream that the TPM will recognize, and parsing the TPM's response.

```
public class HalDevices
 {
    public static bool TpmSendReceive(byte[] command, out byte[] response);
 }
```

The Tpm device is exposed through the kernel diagnostics contract (base\Contracts\Diagnostics.Contracts). Currently only synchronous SendReceive operation is implemented.


```
public contract TpmContract : ServiceContract
{
    // The name you should use to look up this module using the NameServer.
    public const string ModuleName = "/tpm";

    // START messages
    out message Ready();

    // READY messages
    in  message GetDriverName();
    out message DriverName(char []! in ExHeap data);

    in  message GetDriverVersion();
    out message DriverVersion(char[]! in ExHeap data);

    in  message Send(byte[]! in ExHeap data);
    out message AckStartSend();

    in  message Receive();
    out message ReceivedData(byte[]! in ExHeap data);

    in  message SendReceive(byte[]! in ExHeap data);
    out message Response(byte[]! in ExHeap data, bool status);

    // IO_RUNNING messages

    in  message GetTpmStatus();
    out message TpmStatus(byte[]! in ExHeap status);


    in  message Cancel();
    out message AckCancel();

    out message SendComplete();

    override state Start: one
    {
        Ready! -> ReadyState;
    }

    state ReadyState: one
    {
        GetDriverName?              -> DriverName!          -> ReadyState;
        GetDriverVersion?           -> DriverVersion!       -> ReadyState;
        Receive?                    -> ReceivedData!        -> ReadyState;
        GetTpmStatus?               -> TpmStatus!           -> ReadyState;
        SendReceive?                -> Response!            -> ReadyState;
        Send?                       -> AckStartSend!        -> IO_RUNNING;
    }


    //
    // Running state
```



```
        //
        state IO_RUNNING: one
        {
            GetTpmStatus? -> TpmStatus!  -> IO_RUNNING;
            Cancel?       -> AckCancel! -> ReadyState;
            SendComplete! -> ReadyState;
        }

    }
```

6. The TPM Device Service and client library

Most useful TPM functions need to be constructed from a sequence of TPM opcode steps. For example, to perform a quote operation using an existing key would involve the following TPM operations:

```
TpmSession session = new TpmSession();
AuthorizationData auth = new AuthorizationData();
AuthSession1 = session.Oiap(auth, false)       -- authorization session to load a key
QuoteKeyHandle = session.LoadKey(AuthSession,
       KeyBlobToLoad)                          -- load the “quoting” key
AuthorizationData auth2 = new AuthorizationData();
AuthSession2 = session.Oiap(aut2h, false)      -- auth session for using key
quoteSession.Quote(QuoteKeyHandle, new TpmHash(), PcrsToQuote, AuthSession2,
       DataToQuote, out QuotedPcrs, out signature)    -- do the “quote”
session.FlushSpecific(AuthSession1)                   -- clean up
session.FlushSpecific(QuoteKeyHandle)
```

This illustrates the use of authorization sessions (OIAP is a specific class of authorization session) and key slots. The TPM contains other resources that must be managed, but managing key- and OIAP-slots will satisfy our preliminary goals.

The TPM service exposes TPM functionality to callers and manages the resources they use in such a way that

- An application cannot use resources belonging to another
- Resources needed for execution of a command are loaded when the command is invoked
- Authorization sessions and keys are context-swapped when space is needed
- Resources are cleaned up when an application exits

We accomplish this by:

a) Demanding all TPM-operations be executed in the context of a *TPM session* (see the interface below)
b) Provide a reasonably high semantic level interface to the TPM: for example a class interface library that exposes the underlying TPM operations in the context of the client (rather than demanding that the caller create byte-streams that the TPM will recognize).
c) Tracking key and authorization slot creation and deletion, and *virtualizing* the resulting TPM handles
d) Ensuring that the necessary resources are loaded into the TPM when the commands are issued to the TPM
e) Command queuing and scheduling

One of the challenges of using the TPM is that it is designed to be used securely over a network. In order to protect against adversarial modification and observation of the commands are responses, certain parameters must be encrypted and most parameters must be integrity protected with hashes and HMACs. Another challenge is that the data structures that must be marshaled are complex and the marshalling rules are rococo.

To support the commands necessary for the scenario goals, we have added SHA1, HMAC-over-SHA1, raw RSA (for taking ownership), and some padding schemes. Marshalling is be performed by the TCB in the context of the TPM Service SIP (implemented in Libraries\Tpm).

Implementation note:

For now the initial operation of taking ownership by executing TestTakeOwnership under Vista. Once the TPM is properly owned, it can be used by Singularity. Porting the TestTakeOwnership does not pose significant technical challenges, but we postponed its implementation till the next sprint.

There are various semantic levels at which a public interface can be defined. We have implemented one in which a subset of the TPM commands are exposed directly, except for internal translation of TPM-handles to virtual-handles. This allows for increased programmers control and flexibility.


```
public class TpmSession
{

    public TpmSession();


    //
    // authorization sessions
    //

    public OiapSession Oiap(
                        AuthorizationData EntityAuthorization,
                        bool ContinueAuthSession
                             );


    public OsapSession Osap(
                        TpmEntityType EntityType,
                        int EntityValue,
                        AuthorizationData EntityAuthorization,
                        bool ContinueAuthSession
                        );


    //
    // sealed storage
    //

    public byte[] Unseal(
                           TpmKeyHandle ParentHandle,
                           TpmStoredData12 SealedBlob,
                           AuthorizationSession ParentAuthSession,
                           AuthorizationSession UnsealAuthSession
                       );


    public TpmStoredData12 Seal(
                            TpmKeyHandle ParentHandle,
                            PcrSelection SourceSelection,
                            TpmLocality ReleaseLocality,
                            PcrSetting[] ReleasePcrValues,
                            AuthorizationData UnsealAuthorization,
                            OsapSession ParentHandleAuthSession,
                            byte[] DataToSeal
                            );



    //
    // keys
    //
```



```
        public TpmPubKey OwnerReadInternalPub(
                                              TpmKeyHandle EkOrSrkHandle,
                                              AuthorizationSession OwnerAuthSession
                                             );


        public TpmKeyHandle
        LoadKey(
                 TpmKeyHandle ParentTpmKeyHandle,
                 TpmKey KeyBlobToLoad,
                 AuthorizationSession ParentKeyUseAuthorizationSession
                 );


        public TpmKey CreateWrappedKey(
                                    TpmKeyHandle ParentTpmKeyHandle,
                                    AuthorizationData UseAuthorization,
                                    AuthorizationData MigrationAuthorization,
                                   TpmKey KeyPrototype,
                                   OsapSession ParentHandleAuthSession
                                   );


       public byte[] Sign(
                           TpmKeyHandle SigningTpmKeyHandle, byte[] DataToSign,
                           AuthorizationSession SignKeyUseAuthorizationSession
                           );

        //
        // random number generator
        //

       public byte[] GetRandom(int numBytes);


        public void StirRandom(byte[] RandomDataToStir);


        //
        // PCR related
        //

        public PcrValue PcrExtend(uint PcrIndex, PcrValue ValueToExtend);


        public PcrValue PcrRead(int PcrIndex);


        public void PcrReset(PcrSelection PcrsToReset);


        //
        // identity keys
```



```
        //

        public void MakeIdentity(
                                AuthorizationData NewIdKeyUseAuthorization,
                                TpmHash ChosenIdHash,
                                TpmKey IDKeyPrototype,
                                AuthorizationSession SrkAuthSession,
                                OsapSession OwnerAuthSession,
                                out TpmKey NewIdentityKey,
                                out byte[] SignatureOveridentityBinding
                                );



        public TpmSymmetricKey ActivateIdentityNotWorking(
                        TpmKeyHandle KeyToActivate,
                        TpmEkBlobActivateHolder ActivationBlob,
                        AuthorizationSession KeyToActivateAuthSession,
                        AuthorizationSession OwnerAuthSession);


        public void Quote(
                           TpmKeyHandle IdentityTpmKeyHandle,
                           TpmHash Nonce,
                           PcrSelection PcrsToQuote,
                           AuthorizationSession IdKeyAuthSession,
                           out PcrComposite CurrentPcrs,
                           out byte[] SignatureOverCurrentPcrs
                           );


        //
        // timers and counters
        //

        public TpmCurrentTicks GetTicks();

        public void CreateCounter(
                                   AuthorizationData NewCounterAuthData,
                                   byte[] CounterLabel,
                                   OsapSession OwnerAuthSession,
                                   out TpmCounterId NewCounterId,
                                   out TpmCounterValue NewCounterValue
                                   );


        public TpmCounterValue IncrementCounter(
                                   TpmCounterId CounterId,
                                   AuthorizationSession CounterAuthorization
                                   );


        public TpmCounterValue ReadCounter(TpmCounterId CounterId)
```



```
    public void ReleaseCounter(
                                TpmCounterId CounterId,
                                AuthorizationSession CounterAuthorization
                                );

    public void ReleaseCounterOwner(
                                     TpmCounterId CounterId,
                                     AuthorizationSession OwnerAuthorization
                                     );

}
```

7. Integration with the security subsystem

We have extended the principal implementation to include the cryptographic hash of a user SIP during creation. In our prototype, a quote operation uses all information singularity keeps about the process (e.g. principal, module hashes). We will get more sophisticated as needs arise.

Appendix A

Table A1 shows relevant PCR usage in compliant PCAT (non-EFI) platforms. The initial prototype will use the PCR registers in bold type.

PCR Index	Description of Use (summary only: refer to TCG documentation for details)
0	Version number, hash of BIOS
1	Motherboard configuration, microcode patches
2	Option ROM code
3	Option ROM configuration and data
4	Initial Program Load (IPL) code and data (typically MBR, ISO boot image, PXE network bootstrap program)
5	IPL configuration and data
8	Singularity kernel load-policy
18	Hash of hypervisor loader (Intel LaGrand)
23	**Resettable PCR (by application code)**

