Designing a digital security envelope using the Trusted Platform Module

King Ables
School of Computer Science
University of Birmingham
Edgbaston, Birmingham, UK B15 2TT
rka891@cs.bham.ac.uk
kingables@yahoo.com

Abstract

In the physical world, one can write down a secret and seal it in a security envelope so that it may be given to someone else and either opened and read or returned unopened with the guarantee that the secret was not revealed. This property does not hold when the secret is represented in digital form. Once a copy of digital data is given to another user, it cannot simply be returned because multiple copies exist.

Inspired by an earlier paper, this report proposes and evaluates a design for a digital security envelope using the Trusted Platform Module (TPM). The proposed design allows data to be distributed such that it may later be opened or "returned unopened" (i. e., refused), but never both.

First, the required TPM primitives are summarized and a previously proposed solution is evaluated. Then, a concise set of requirements and a preliminary design are described. The design is evaluated and the complicating factors required to assure confidence in the solution are discussed.

Rather than suggest this complex design be implemented, enhancements to the Trusted Platform Module are proposed which would allow a much more straightforward solution to be implemented and that improved solution is described.

Index Terms

Attestation, monotonic counter, sealed data, security envelope, trusted platform.

1. Introduction

Alice is off on a well-deserved holiday. Her friend Bob has agreed to keep an eye on her home. As a precaution, should something unthinkable occur, Alice writes the combination to her home safe on a piece of paper and seals it in a tamper-evident security envelope and gives it to Bob along with the keys. Should she not return, Bob will be able to access important papers in her safe. Presuming she does return safely, Bob will return the unopened envelope to her and she can be confident that, not only did he not open the safe, but also that he no longer possesses the combination, and therefore, the ability to do so.

Digital data does not adhere to these physical restrictions. If Alice gives Bob digital data, he can later return it, but what he returns is merely a copy, not a unique instance. Bob is free to keep a copy of the data and Alice has no guarantee that he cannot or will not use it.

The Trusted Platform Module (TPM), now contained in most personal computers, provides a point of physical trust that can be used to build a digital equivalent to the security envelope. This report proposes a design for an *Envelope Service* which allows Alice to send Bob digital data such that Bob may do *one and only one* of the following:

- read the information contained in the envelope without further involvement from Alice.
- return the information to Alice unread.

Such an application could be useful for distributing an encryption key and allowing one recipient

to later “return” the key, both proving it was not used and eliminating the need for the owner to change and redistribute the key.

In Section 2, the areas of the TPM used to implement a solution are reviewed along with the original paper proposing a solution. Section 3 lists the specific requirements for the solution and Section 4 describes a preliminary design. In Section 5, the issues involved in expanding the design are discussed, and in Section 6, enhancements to the Trusted Platform Module that would eliminate much of the required complexity in the design and facilitate a much more straightforward and elegant solution are proposed.

2. Background

This proposed solution depends on several functions of the Trusted Platform Module and is an extension of a previous solution proposal.

2.1. Trusted Platform Module

The Trusted Platform Module (TPM) is a chip on the motherboard of most personal computers whose behavior is specified by the Trusted Computing Group[1] (TCG). As defined by version 1.2, the current specification published in [26], [24], and [25], the TPM:

- is tamper-resistant and tamper-evident.
- can generate high-quality random numbers.
- contains volatile and non-volatile storage.
- provides cryptography functions such as key generation, encryption, and hashing.
- provides monotonic counters (new in v1.2).
- maintains platform configuration data.
- protects encryption and signing keys.

The Trusted Software Stack (TSS) includes the driver and the API for the TPM.

Each TPM chip contains one Endorsement Key (EK), an RSA key pair, and a certificate of authenticity installed by the manufacturer.

The TPM protects the private part of an RSA key pair in a shielded location when loaded and by encryption when stored outside the TPM. In either case, the private portion of a key pair is never revealed.

1. http://trustedcomputinggroup.org

```
Getting number of PCRS:
numPcrs = 24
Reading PCRs
PCR 0: c1b06bf7337dd42cc3e61304f2e8d088221a7745
PCR 1: 96af25c5457f308e9b02a459e7d34d00898e6b12
PCR 2: 53de584dcef03f6a7dac1a240a835893896f218d
PCR 3: 3a3f780f11a4b49969fcaa80cd6e3957c33b2275
PCR 4: dd69f9acdbdbfffd92a81aaee08c3cb2119978a5
PCR 5: 3a3f780f11a4b49969fcaa80cd6e3957c33b2275
PCR 6: 585e579e48997fee8efd20830c6a841eb353c628
PCR 7: 3a3f780f11a4b49969fcaa80cd6e3957c33b2275
PCR 8: 0000000000000000000000000000000000000000
PCR 9: 0000000000000000000000000000000000000000
PCR 10: 0000000000000000000000000000000000000000
PCR 11: 0000000000000000000000000000000000000000
PCR 12: 0000000000000000000000000000000000000000
PCR 13: 0000000000000000000000000000000000000000
PCR 14: 0000000000000000000000000000000000000000
PCR 15: 0000000000000000000000000000000000000000
PCR 16: 0000000000000000000000000000000000000000
PCR 17: ffffffffffffffffffffffffffffffffffffffff
PCR 18: ffffffffffffffffffffffffffffffffffffffff
PCR 19: ffffffffffffffffffffffffffffffffffffffff
PCR 20: ffffffffffffffffffffffffffffffffffffffff
PCR 21: ffffffffffffffffffffffffffffffffffffffff
PCR 22: ffffffffffffffffffffffffffffffffffffffff
PCR 23: 0000000000000000000000000000000000000000
```

Figure 1. Example PCR values

TPM keys and some commands require *authorization data* be specified to gain access. Usually referred to as AuthData, these are 160-bit values that act as passwords to protect TPM data.

2.1.1. Platform Configuration

Platform Configuration Registers (PCRs) contain the hashed representation of various components of the system[4][9][6]:

- 0 – BIOS
- 1 – BIOS (hardware) configuration
- 2 – ROM BIOS
- 3 – ROM configuration
- 4 – boot loader / Master Boot Record (MBR)
- 5 – boot loader configuration

A PCR is *extended* by a new value with a hash of the old and new data. The value assigned to the PCR is output from the SHA-1 hash function, so it is computationally infeasible to assign a specific value to a PCR and the process is irreversible. Figure 1 is an example of PCR values on a system running Windows XP.

Data can be *sealed* (encrypted) against specific PCR values using the TPM command **TPM_Seal** and only *unsealed* (with **TPM_Unseal**) when those PCRs are set to the same values[25][14].

2.1.2. Platform Attestation

Attesting to the state of a platform proves to another party that:

- the configuration is in a known state.
- all software that was measured is correct.
- the attesting device is genuine.

Attestation requires measurement of hardware and software components and a statement of identity of the platform.

Attestation does not guarantee untrusted code cannot run at boot time, but that it will be detected if it does run[16]. Attestation also does not protect against flaws in any software that was measured[14].

2.1.2.1. Measurement

A *chain of trust* is created from the Core Root of Trust Measurement (CRTM), the immutable BIOS code executed at power on, through all running software to be measured.

On system reset, this chain of trust is created by each component measuring the next one before transferring control to it:

1) The CRTM measures the BIOS, extends PCR 0 with the value, then executes the BIOS code.
2) The TCG-compliant BIOS measures devices on the PCI and PCI-X buses and extends PCR 1.
3) The BIOS measures the MBR to PCR 4 and runs the boot loader.
4) A TCG-compliant boot loader also measures the program it will run.

For example, the boot sequence for Windows Vista is as follows[13]:

1) CTRM
2) mutable BIOS
3) bootmgr (Windows boot loader residing in the MBR)
4) `WINLOAD.EXE` (or `WINRESUME.EXE` if resuming from hibernation)
5) `CI.DLL` (the integrity checker which checks Vista and BitLocker modules)

Each step measures the code that it will run and extends the appropriate PCR values before transferring control to the next step.

Any device that can perform Direct Memory Access (DMA) is a risk, so device and device configurations must also be measured[9].

Measurement does not guarantee integrity but does provide evidence of tampering[10].

The TPM command **TPM_Quote** is used to obtain signed measurement results from the PCRs. These results may then be compared to expected values depending on the intended use of the platform[15]. These results must be digitally signed by the TPM to prove their authenticity to another party, so the TPM command **TPM_PCRRead**, which also returns PCR values, is insufficient.

The measurements themselves do not provide any judgment about the trustworthiness of the platform[28], it is incumbent upon the verifying party to make this judgment based on the measurement values.

2.1.2.2. Identity

For attestation to be credible, it must come from an authentic source, a real TPM. The TPM identifies itself with an Attestation Identity Key (AIK), an RSA key pair. A TPM may create an unlimited number of AIKs, or identities, each being identified by a unique label to which it is bound[15].

The AIK is certified as belonging to a TCG-compliant TPM by a Privacy Certificate Authority (PCA). The PCA verifies the relationship between a valid Endorsement Key (EK), which the PCA can verify, and the AIK. While the party to whom the AIK is identified cannot specifically identify the root EK, the PCA could, so in theory, represents a weak point in complete privacy[3].

The AIK can also be proven legitimate by a more anonymous process called Direct Anonymous Attestation (DAA) where the AIK is validated by way of a zero-knowledge proof[11] rather than via disclosure of the public EK, even to a PCA. DAA may be required where no risk of

exposure of the public EK, and hence a recognizable identity of the platform, can be tolerated[3]. The current solution does not require this level of anonymity, but this could change depending on future use.

The steps required to create and certify an AIK with a PCA are[15]:

1) Call **TPM_MakeIdentity** to create an AIK key pair.
2) Call **TSS_CollateIdentityRequest** to create the request for the PCA.
3) Send the request, including the public AIK, the public EK, and the EK certificate, to a Privacy Certificate Authority (PCA).
4) The PCA verifies the information, creates a certificate, encrypts it with the public EK, and sends it in response.
5) Call **TPM_ActivateIdentity** to decrypt the response and to verify and activate the AIK.
6) Call **TSS_recover_TPM_identity** to get the AIK certificate[11].

The TPM uses the private part of the Endorsement Key (EK) to decrypt data sent to it, but not to sign data. Identity keys are used to sign data and not to decrypt. It is considered bad practice to sign and decrypt with the same RSA key because of the mathematical vulnerability to attack this introduces due to the relationship between the two keys and the fact that one of the keys is public[15].

The AIK cannot be used to sign data generated outside the TPM to prevent rogue software creating data that looks like what might be generated by the TPM and using the AIK to sign the bogus information. Therefore, to sign arbitrary data, one must create a signing key that is, in turn, signed with an AIK[11][15]. However, PCR data returned by **TPM_Quote** can be signed directly with an AIK because it was generated by the TPM[15][23].

2.1.3. Monotonic counter

A *monotonic counter* is a new primitive added to the TPM specification in version 1.2. The TPM provides up to 4 monotonic counters, based on a single base counter which is a 32-bit value that can only be incremented. A trusted monotonic counter mechanism can be used to guard against replay attacks[22].

Only one named counter may be incremented per boot cycle[4][26]. The original intent of providing 4 counters was to allow different operating systems exclusive access to a unique value.

TPM commands providing access to counters include[25]:

- **TPM_CreateCounter** creates a new counter (must be done in a TPM transport session).
- **TPM_IncrementCounter** increments an existing counter (all subsequent uses must name the same counter until the TPM is reset when the system boots).
- **TPM_ReadCounter** returns the current value of the named counter.
- **TPM_ReleaseCounter** releases the named counter.

The base counter is incremented when any counter is incremented. When a new counter is created, it is assigned a value one higher than any previous value by incrementing the base counter and assigning that value to the new counter.

The **TPM_CreateCounter** command requires owner AuthData so only the owner of the TPM may create a new counter[26]. Each time a new counter is created, the base counter is incremented.

The TPM specification requires the counter to function properly for 7 years of increments at the rate of once every 5 seconds without a hardware failure. Manufacturers are free to throttle the speed at some point faster than once per 5 seconds, but the specification does not define a threshold. In [27], 2-5 seconds is cited as the counter update interval used by most TPM manufacturers. While the TPM counter can wrap[24], if manufacturers do indeed limit the rate of increments to once every two seconds, the 32-bit value would not wrap for 136 years, making it an impractical attack vector.

2.1.4. Transport session

Many TPM operations are performed in a *transport session* with the TPM. A transport

session represents an encrypted communications path that allows multiple TPM commands to be executed. Two of the selectable traits of transport sessions are of interest:

- exclusive, where no other transport session may have access to the TPM at the same time so no TPM state may change during the session other than what is changed by the commands in the session.
- logged and signed, where the result is a log of the entire operation of the transport session as well as a hash of the log that is signed by the TPM to authenticate it.

An exclusive logged and signed transport session may be used to prove to another party that a TPM operation was performed properly[23]. To obtain a signed transport log, use the TPM command **TPM_ReleaseTransportSigned**[25].

2.2. A PCR-based electronic envelope

In [17], Dr. Mark Ryan of the University of Birmingham proposed an electronic envelope using PCRs to restrict access to the message contained in the envelope.

The major disadvantage in using PCRs is that they maintain a volatile state which is lost when the TPM is reset. When the system is rebooted or crashes, Bob loses his ability either to read the information from Alice or to prove to Alice that he did not read it.

While this initial approach does work and has the advantage of being straightforward because the TPM controls the access to the data, it does not lend itself well to a production solution because of the fragility of the state information.

3. Requirements

The Envelope Service application should satisfy the following requirements:

1) No trusted third party is required, security is based solely on a platform containing a TPM v1.2.
2) The act of returning the message unread must negate Bob's ability to ever read it in the future, even if he keeps a copy of the data.
3) The service must be robust to both a planned and unplanned shutdown (i.e., system crash).
4) Bob must control the data and his ability to act must not depend on any resource belonging to Alice being available when he chooses to read or refuse the message.

While the system should be robust across system resets, future ability to read or refuse a message cannot be guaranteed in the face of a denial of service attack or a disk failure.

4. Design

To use the Envelope Service, Alice will create a blob[2] containing the message that can only be opened by the verified Envelope Service. The procedure she will follow is:

- Request an envelope which includes a TPM-protected key tied to a monotonic counter value.
- Verify the envelope has been created by an authentic TPM running a properly installed and configured Envelope Service application.
- Tie the message to this key (i. e., insert the message into the envelope).
- Send the envelope to Bob.

Then, Bob can, at a time of his choosing, use the Envelope Service to open the envelope or obtain proof that he did not open it and forfeit his ability to ever open it. The act of opening or refusing the message increments the counter so neither operation can be repeated nor can the other operation be performed later.

4.1. Assumptions

In validating the concept of this design, the environment is restricted to reduce the complexity of the problem. Steps required to broaden the scope, and therefore the usability, of the solution will be considered later in this report.

For the moment, the Envelope Service will be restricted to an environment where:

2. In TPM parlance, a *binary large object*, or blob, is a grouping of data containing multiple parts necessary to perform a task or represent an encrypted or otherwise protected data item.

- it runs on a dedicated system.
- the system has a TPM and TCG-enabled BIOS and boot loader.
- the application runs native on the hardware with no operating system or virtual machine support.
- the Envelope Service is capable of processing only one envelope at a time.

In addition to these specific assumptions about the platform, some way of obtaining PCR values for various makes and models of hardware platforms will also be required.

4.2. Data

The Envelope Service will maintain and use the following data:

- Envelope Service Sealing Key (ESSK) – the encryption key used to keep its own data secure.
- Envelope Service (Attestation) Identity Key (ESAIK) – the identity key (AIK) pair for the Envelope Service.
- Envelope Service (Attestation) Identity Certificate (ESAIK certificate) – the certificate from the PCA certifying the AIK.
- Envelope Encryption Key (EEK) – each envelope uses a unique RSA key pair.
- Envelope Counter Name (ECN) – the name of the TPM monotonic counter used to restrict access to the envelope.
- EEK AuthData (AuthData) – random authorization data that will be used to protect EEKs.

4.3. Proof of authenticity

How does the instance of the Envelope Service prove it is legitimate and not rogue software pretending to be an Envelope Service? Computer systems provide no mechanism to verify the application with which one communicates[19], but the TPM can be used to provide such assurances.

If the Envelope Service is included in the chain of trust and measured during the boot process, users can verify that it is running the legitimate Envelope Service code. To accomplish this, it must show proof of proper state signed by a proper TPM. This requires:

- a list of PCR values from **`TPM_Quote`** signed with the ESAIK.
- the PCA certificate for the ESAIK.

The certificate from the PCA attests to the validity of the AIK and guarantees that an authentic TPM has signed the data[11].

Only a legitimate Envelope Service can provide these particular PCR values and only an authentic TPM can obtain an AIK certificate. When she verifies these, Alice knows that by encrypting her data with the given EEK, it can only be decrypted by this legitimate instance of the Envelope Service.

This requires a boot loader that can be configured to measure the Envelope Service application such as TrustedGRUB described in [10]. Based on GRUB, the GRand Unified Boot loader that is part of most Linux distributions, TrustedGRUB is divided into two stages, where:

- stage 1 is loaded from MBR and measures and executes stage 2.
- stage 2 measures an operating system or whatever program is to be run next (such as the Envelope Service).

4.4. Architecture

The Envelope Service runs in two states: initialization and service. The initialization state, State 0, starts the service, creates or unwraps keys and data, and prepares to begin servicing envelope requests. The service state, State 1, is the “normal” operational state of the application.

4.4.1. State 0: initialization

State 0 initializes the environment in which the Envelope Service will run with the following steps:

- Unseal or create the initialization blob containing the ESAIK.
- If just created, seal the initialization blob against current PCR values set by State 0.
- Load the ESAIK into the TPM.
- Advance to State 1, the service state, by extending a particular PCR.

The sealing key (ESSK) can only be loaded during State 0 since it is sealed against the PCR values at the time the application is first executed. The sealing key requires no TPM authorization data (AuthData) because it is stored in a blob which was sealed against PCR values and is only accessible to a measurement-verified Envelope Service in State 0.

4.4.2. State 1: service

In State 1, the Envelope Service waits to provide one of these services upon request:

- Create a new (empty) envelope.
- Open an existing envelope and return the data.
- Return proof of refusal to open an existing envelope.

When creating a new empty envelope, the Envelope Service returns the new public EEK, counter information, public ESAIK, ESAIK certificate, and signed counter information to the envelope requestor.

Because the initialization information is sealed against the PCRs at the time the system boots, once the Envelope Service has loaded its data, it extends a particular PCR to enter State 1 to guarantee that no other process may then access the initialization information.

4.5. Operation

When the system boots, the chain of trust is followed all the way to the Envelope Service:

- Run the immutable BIOS which measures the mutable BIOS.
- Run the mutable BIOS which measures the MBR.
- Run the boot loader in the MBR which must measure the Envelope Service software.
- Run the Envelope Service.

The first time the Envelope Service executes, the following tasks are performed:

- Create the ESSK using the TPM command **`TPM_CreateWrapKey`** (with null AuthData) which will be used to seal state information blob to State 0 (current state).
- Create the ESAIK.
- Create the random AuthData for all EEKs using the TPM command **`TPM_GetRandom`**.
- The TPM signs a digest of PCA information to bind it to the public ESAIK.
- Submit the public ESAIK and public EK to the PCA.
- Receive the ESAIK certificate from the PCA.
- Set the name of the TPM monotonic counter (ECN).
- Seal a blob containing the ESAIK, its certificate, and the ECN to State 0.
- Generate a **`TPM_Quote`** of PCR state signed with the ESAIK.
- Extend a particular PCR by 1 to advance to State 1.

Subsequent runs of the Envelope Service need simply restore the state (which will only succeed in State 0):

- Load the ESSK.
- Unseal the ESAIK, ESAIK certificate, ECN, and AuthData.
- Load the ESAIK into the TPM.
- Generate a signed **`TPM_Quote`** of the PCR state.
- Extend a particular PCR to advance to State 1.

Note that the PCR values are generated by **`TPM_Quote`** during State 0, so cannot be generated with a nonce from Alice when she requests an envelope. Normally "real-time" attestation is validated with such a nonce, but Alice can accept the state data as being correct because it is signed with an authenticated TPM identity key (ESAIK). In any case, a "live" **`TPM_Quote`** from State 1 would not provide her the PCR values she expects since the Envelope Service will have extended a PCR to get to State 1.

An EEK is created in State 1 so is not protected by sealing against PCRs reflecting State 0. The EEK is protected by a random AuthData value created by the Envelope Service during initialization and stored in the initialization state blob. Because the initialization state is protected by sealing against PCRs reflecting State 0, the AuthData is inaccessible to any other application

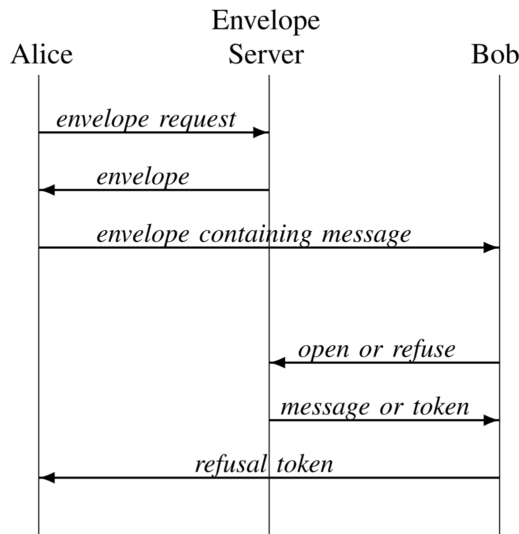

Figure 2. General Envelope Service protocol

at any other time. The AuthData for all EEKs is known only to the Envelope Service.

Figure 2 shows the generic protocol for the Envelope Service. The specific steps required for Alice to send Bob data in a digital security envelope are:

- Generate a random 160-bit nonce using the TPM command **`TPM_GetRandom`**.
- Send the Envelope Service a request for a new envelope (including the nonce).

The Envelope Service returns the following data to Alice as the envelope:

- ESAIK and certificate
- **`TPM_Quote`** of PCR values signed with the ESAIK
- name and new value of the incremented TPM monotonic counter
- public part of a new EEK for the envelope with AuthData known only to the Envelope Service
- her original nonce

When Alice has verified the envelope and is ready to send her data to Bob, she will:

- Generate a symmetric key with which she will encrypt her message.
- Encrypt the symmetric key with the public EEK.
- Encrypt the counter name and value with the public EEK.
- Encrypt the message with the symmetric key.

and send it to Bob. Only the Envelope Service can decrypt the symmetric key and it will only do so if the named counter still has the specified value.

5. Design enhancements

Some admittedly restrictive assumptions were made about the environment that would support the Envelope Service in order to simplify the process as much as possible to concentrate on the technical approach. Now that it has been shown that the TPM can play a role in the solution, the issues involved in expanding the original assumptions and deploying a more realistic solution will be examined.

5.1. Support for multiple envelopes

The most egregious limiting assumption in the preliminary design is that of supporting only one envelope at a time. Since each envelope requires its own counter and the TPM only allows the use of a single monotonic counter at any one time, *virtual monotonic counters* must be implemented to support multiple envelopes.

Virtual monotonic counters based on a TPM counter, first described in [22], can implement an infinite number of counters based on a single TPM counter. Without exclusive access to the hardware counter upon which it is based, however, a virtual monotonic counter still suffers from several problems:

1) It is tamper evident, but not tamper proof.
2) It could suffer from a denial of service attack.
3) Deterministic behavior of the counter is not guaranteed.

If software other than the Envelope Service were to increment the TPM counter (bypassing a virtual counter manager), the next increment performed by the virtual counter manager would be more than one greater than any current counter, thus the next value is unpredictable. This is not an issue for the Envelope Service since it uses the counter like a Lamport Clock timestamp[12] and non-deterministic increments are sufficient for a

timestamp[27] because it is only necessary to know the counter has changed.

Each increment of a virtual counter causes an increment of the real TPM counter via an exclusive logged and signed transport session. This creates a log of all increments performed for each virtual counter, the sum total of which can be used to verify the TPM counter.

Virtual counters can also be implemented in unmeasured software by using an exclusive logged and signed transport session to read or increment the counter. While more complicated to parse and verify, this TPM-signed log guarantees the proper operation was performed with the proper counter[22]. Using signed transport logs, it is possible to prove the trail of changes to each counter without requiring the operations be performed in measured software[27].

With the support of a virtual counter manager, it is also possible to implement an even higher-level abstraction of a one-time-use certificate or ticket, as in [23] or [11]. A ticket might be issued with the envelope that is good for either a read or refuse operation. Since the ticket can only be used once, only one of the two operations could ever be performed. Any design of a consumable credential must enforce the consumption of the credential when properly used as well as guard against invalid consumption (loss) if used improperly[2].

5.2. Run on a small kernel

For the sake of simplicity of measurement, the Envelope Service is designed to run as native code, with no operating system upon which to rely. In reality, having some kind of operating system support under the application would make it much easier to develop since all the required functions of an operating system would not have to be implemented in the application. Building the service on a small kernel that provides functions like a file system and a networking stack would simplify the implementation greatly. A customized operating system intended to support a specific application can be much smaller than a normal operating system[7]. Any such kernel would also need to be modified to extend the CRTM to the Envelope Service.

Some alternatives for a lightweight mechanism upon which to build the application include:

- the L4 microkernel[3]
- a stripped-down Linux distribution like Gentoo or xPUD
- the pending Google Chrome OS

Gentoo Linux is known for the relative ease with which parts can be included or excluded and modifications made and the kernel recompiled.

Whatever underlying mechanism is chosen, it effectively will become part of the application from an installation, distribution, and support point of view. Any code used will be customized especially for use with the Envelope Service.

5.3. Run on a shared system

Ideally, the Envelope Service should run on a generic system that is also running an operating system like Windows or Linux. However, this creates significant attestation complexities.

5.3.1. Dual boot

A shared system that *dual boots* two different programs, where the user selects which program or operating system is run at boot time, can be configured to run a normal operating system and the Envelope Service.

The main disadvantage of such a configuration is that the Envelope Service is not available whenever the operating system is running. The Envelope Service could be made available on demand by rebooting and selecting it, but then the user's work is interrupted.

5.3.2. Concurrent with an operating system

For both the user's operating system and the Envelope Service to run concurrently, they must both be run in a virtual machine (VM) environment under a *hypervisor* or virtual machine monitor (VMM). Because the Envelope Service keeps encryption keys in memory, any implementation must guarantee that no other process can access the memory it uses. A VMM separates

3. http://os.inf.tu-dresden.de/L4/

and protects the programs from each other while allowing them to run as if they have an entire machine to themselves.

Virtualization adds another level of complexity to the measurement and attestation of the Envelope Service code. The virtual machine code must not only be measured, but it also must be modified to measure the code that it runs to extend the chain of trust.

The most common VMMs today are Xen and VMware. Xen is open-source, so could more easily be modified as required to extend the chain of trust to the Envelope Service.

In addition to well-known solutions, some other specialized alternatives might be considered.

Perseus[19] is also a VMM, but is implemented on the L4 microkernel which is small (about 7000 lines of code). The small security kernel manages access to the hardware resources of the system and support for the TPM was added based on code from IBM. The chain of trust is extended by modifying the GRUB boot loader to support attestation using the TPM. Since the code is small and stable, it would likely not require frequent modification.

Terra[8][7] is a trusted virtual machine monitor (TVMM) providing two VM abstractions:

- open box – runs as a normal system
- closed box – cannot be inspected by the platform owner, secure, not modifiable

The TVMM simplifies attestation slightly because it only requires measuring the TVMM. The “closed box” VM provides its own method of proving what executables it runs. However, this TVMM mechanism to prove the authenticity of the Envelope Service must then be included in the verification process used by Alice. Since the TVMM itself provides assurance about the application, updating the Envelope Service application would no longer require that Alice obtain new PCR information, although updating the TVMM itself would.

In the case of both Perseus and Terra, the TPM is still shared amongst all processes, virtual machines, and users, so problems related to non-exclusive access to the TPM can still occur. To address this, virtualizing the TPM itself might be considered.

vTPM[1] solves the exclusive counter problem by providing a virtual TPM to each virtual machine. vTPM runs on Xen and provides a full TPM implementation in software. Each VM can create and destroy its own virtual TPM instance, so running the Envelope Service in a VM with its own vTPM guarantees it exclusive access to that TPM instance. However, because each vTPM instance has its own EK that was generated and not created by a TPM manufacturer, vTPM AIKs may not be verifiable by a PCA.

5.4. Measurement and attestation issues

Virtualization increases the complexity of measurement and attestation. Adding an operating system or a VMM to the equation creates more variability in PCR values and more complexity for Alice when verifying the configuration.

An operating system is a complex entity to measure and assuring trust is difficult[22]. Code that changes regularly changes the measurement result. An operating system may have a seemingly infinite number of “valid” configurations and it is impractical to verify the trustworthiness of every possible platform configuration[18].

To run on a shared system, the Envelope Service must be booted prior to the operating system to guarantee consistent measurements. Additionally, all components ahead of the Envelope Service in the boot chain must be modified to continue the chain of trust.

Such a configuration is also fragile. Data sealed to a specific set of PCRs could be rendered inaccessible by a minor change to the system[18][6].

Even a trusted operating system is not impervious to physical attack. If an attacker can read or modify memory while bypassing the CPU (e. g., via a device capable of DMA), attacks are possible even though the running code has been deemed trustworthy[9].

Measurement and attestation cannot guarantee reliable results in the face of some future hardware failure[7]. Nor can it protect a system from bugs[27]. It is estimated in [20] that one security-critical bug exists per 1000 lines of code. A typical Linux distribution containing about 5 million lines of code could contain 5000 potential

security-critical bugs. Windows Vista, estimated to have 50 million lines of code, could have 50,000 such bugs. A Trusted Computing Base (TCB) must be small to be trustworthy[19].

Successful measurement is extremely difficult, but not impossible. Sailer, et al., created a system for performing TCG-based integrity measurement[21] in a modified Linux kernel that measures executables before they are executed. This may be well-suited for an end-user operating system, but users of the Envelope Service must trust the complete instance of the software from the moment it begins execution.

If binary attestation is impractical, perhaps attesting to the security properties of the platform is more suitable[16]. Chen, et al., proposed using property certificates[5] and a trusted third party to make such determinations. However, the requirement of the trusted third party conflicts with one of the basic requirements for the Envelope Service.

6. Implementation in unmeasured code

The majority of the design of the Envelope Service involves assuring Alice that it is an authentic version and will perform as expected. The simplest design is not very practical as a real solution, yet as functionality is added to make it more usable, any practical ability to prove legitimacy is made extremely difficult.

The problem is the necessity of running as measured software. If that need could be eliminated, the Envelope Service could run as a service or daemon under a normal operating system. But the steps required to verify the counter and release the sealed information must be atomic:

- check the counter
- read or refuse the message
- increment the counter
- return the result

Alice must be able to trust that these steps are performed properly and completely. If the software performing these operations is not trusted, it could skip critical steps (e. g., incrementing or even checking the counter).

The ideal solution would have the TPM allow or deny the operation as it does when sealing data against PCRs. Any decision performed outside the TPM must be performed in measured software to be trusted. But no TPM command takes a counter name or value as an input argument. If only the TPM could seal and unseal data against a monotonic counter the way it can against PCRs.

6.1. New TPM commands

The entire Envelope Service could be implemented in untrusted code if the critical data was protected by the TPM and the TPM could make the decision whether to allow or deny access to the message. The author suggests the TCG add the following commands to the TPM specification:

- **`TPM_SealByCounter`** (*key*, *authdata*, *data-to-be-sealed*, *counter-name*, *counter-value*, *increment-on-unseal*)
  where *counter-name*, *counter-value* and *increment-on-unseal* may be in a `TPM_COUNTER_INFO` structure, analogous to the `TPM_PCR_INFO` structure used in the existing seal and unseal operations. This command seals arbitrary data with the specified key against a counter name and value just as **`TPM_Seal`** seals against one or more PCR values. The *increment-on-unseal* is a boolean value which specifies whether or not the specified counter should be incremented when the data is unsealed.
- **`TPM_UnsealByCounter`** (*key*, *authdata*, *data-to-be-unsealed*)
  This command obtains the counter name and value from the blob and compares them to the current value of the named TPM counter. If they match, the TPM unseals the data. Upon successful unsealing of the data, but before it is returned to the caller, the named counter is incremented if *increment-on-unseal* was set to `TRUE` when the data was sealed.

6.2. The new solution

Using these proposed new TPM commands, the Envelope Service could be designed in a much more straightforward manner and could run as normal, unmeasured software under any operating

system. In this case, the Envelope Service operation would be:

Alice requests the envelope and receives:

- the ESAIK and certificate as before
- a signed log of a transport session showing the current counter name and newly incremented value
- two public keys
- a signed log of a transport session where:
  - an OpenEnvelopeKey pair is created
  - a RefuseEnvelopeKey pair is created
  - the public portions of both key pairs are displayed
  - both keys are sealed against the current counter value (separately and with *increment-on-unseal* = TRUE)
  - a TPM-signed copy of her nonce (this requires a new signing key be generated and signed with the ESAIK since the ESAIK cannot sign Alice's data directly) to prove the envelope is in response to her request

Now Alice can:

- Verify that the TPM is genuine.
- Verify the counter was named and incremented properly.
- Verify the public parts of the envelope keys she received are the same as in the transport session log.
- Verify the proper sealing of the envelope keys.
- Verify her nonce.
- Encrypt her symmetric key with the OpenEnvelopeKey.
- Encrypt a refusal token with the RefuseEnvelopeKey.
- Encrypt her message with the symmetric key.

and send it all to Bob.

Bob can unseal either one of the two envelope keys, but not both. He can decrypt either the message or the refusal token.

7. Conclusion

This report has shown that the Trusted Platform Module can be used to implement a digital security envelope service. A design was proposed that satisfies a restrictive set of requirements to illustrate the methodology of the solution and the impediments to an actual implementation were examined. Due to the rigors of platform attestation, even the simplest solution is complex. As capabilities are added to the design, this complexity quickly increases to the point of impracticality.

It has also been shown that a much more straightforward solution could be achieved if the TPM provided a sealing operation using a monotonic counter analogous to the sealing operation it currently provides using Platform Configuration Registers. Therefore, two new TPM commands were proposed for addition to the TPM specification.

If the TPM provided these commands, the current requirement for measurement and attestation to prove trustworthiness of the software would be eliminated, and the Envelope Service could be easily implemented in unmeasured code running on any operating system.

Acknowledgements

I am grateful to my supervisor, Dr. Mark Ryan, not only for his guidance and original idea upon which this project is based, but also for his willingness to accommodate me in his already full schedule. Our brainstorming discussions were invaluable and enjoyable.

References

[1] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vTPM: Virtualizing the Trusted Platform Module," in *USENIX-SS'06: Proceedings of the 15th Conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2006, pp. 305–320.

[2] K. D. Bowers, L. Bauer, D. Garg, F. Pfenning, and M. K. Reiter, "Consumable credentials in logic-based access-control systems," in *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS'07)*, February 2007.

[3] E. Brickell, J. Camenisch, and L. Chen, "Direct anonymous attestation," in *CCS '04: Proceedings*

of the 11th ACM Conference on Computer and Communications Security. New York, NY, USA: ACM, 2004, pp. 132–145.

[4] D. Challener, K. Yoder, R. Catherman, D. Safford, and L. Van Doorn, *A Practical Guide to Trusted Computing*. Upper Saddle River, NJ: IBM Press, 2008.

[5] L. Chen, R. Landfermann, H. Löhr, M. Rohe, A.-R. Sadeghi, and C. Stüble, "A protocol for property-based attestation," in *STC '06: Proceedings of the First ACM Workshop on Scalable Trusted Computing*. New York, NY, USA: ACM, 2006, pp. 7–16.

[6] N. Ferguson, "AES-CBS + Elephant diffuser – A disk encryption algorithm for Windows Vista," August 2006. [Online]. Available: http://download.microsoft.com/download/0/2/3/0238acaf-d3bf-4a6d-b3d6-0a0be4bbb36e/BitLockerCipher200608.pdf.

[7] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," *SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 193–206, 2003.

[8] T. Garfinkel, M. Rosenblum, and D. Boneh, "Flexible OS support and applications for trusted computing," in *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*. Berkeley, CA, USA: USENIX Association, 2003, pp. 25–25.

[9] J. Hendricks and L. van Doorn, "Secure bootstrap is not enough: Shoring up the trusted computing base," in *Proceedings of the Eleventh ACM SIGOPS European Workshop*, September 2004.

[10] U. Kühn, M. Selhorst, and C. Stüble, "Realizing property-based attestation and sealing with commonly available hard- and software," in *STC '07: Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*. New York, NY, USA: ACM, 2007, pp. 50–57.

[11] N. Kuntze and A. U. Schmidt, *Trusted Ticket Systems and Applications*, ser. IFIP International Federation for Information Processing. Springer Boston, 2007, vol. 232/2007, pp. 49–60.

[12] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[13] Microsoft Corporation, "BitLocker drive encryption security policy," July 2008, for FIPS 140-2 validation. [Online]. Available: http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp947.pdf.

[14] B. Parno, "The Trusted Platform Module (TPM) and sealed storage," June 2007, technical note. [Online]. Available: http://www.rsa.com/rsalabs/technotes/tpm/sealedstorage.pdf.

[15] S. Pearson, Ed., *Trusted Computing Platforms: TCPA Technology in Context*. Upper Saddle River, NJ: Hewlett-Packard Professional Books / Prentice Hall PTR, 2003.

[16] J. Poritz, M. Schunter, E. V. Herreweghen, and M. Waidner, "Property attestation – scalable and privacy-friendly security assessment of peer computers," IBM, Tech. Rep. RZ3548, 2004.

[17] M. Ryan, "The electronic envelope," 2009, unpublished, provided by author.

[18] A.-R. Sadeghi and C. Stüble, "Property-based attestation for computing platforms: caring about properties, not mechanisms," in *NSPW '04: Proceedings of the 2004 Workshop on New Security Paradigms*. New York, NY, USA: ACM, 2004, pp. 67–77.

[19] ——, "Towards multilaterally secure computing platforms – with open source and trusted computing," *Information Security Technical Report*, vol. 10, no. 2, pp. 83–95, 2005.

[20] D. Safford, "The need for TCPA," October 2002. [Online]. Available: http://www.research.ibm.com/gsal/tcpa/why_tcpa.pdf.

[21] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *13th USENIX Security Symposium*, 2004, pp. 223–238.

[22] L. F. G. Sarmenta, M. van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas, "Virtual monotonic counters and count-limited objects using a TPM without a trusted OS," in *STC '06: Proceedings of the First ACM Workshop on Scalable Trusted Computing*. New York, NY, USA: ACM, 2006, pp. 27–42.

[23] L. F. G. Sarmenta, M. van Dijk, J. Rhodes, and S. Devadas, "Offline count-limited certificates," in *SAC '08: Proceedings of the 2008 ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2008, pp. 2145–2152.

[24] Trusted Computing Group, Inc., *TPM Main Part 2 TPM Structures*, Trusted Computing Group, October 2006. [Online]. Available: https://www.trustedcomputinggroup.org/specs/TPM/.

[25] ——, *TPM Main Part 3 Commands*, Trusted Computing Group, October 2006. [Online]. Available: https://www.trustedcomputinggroup.org/specs/TPM/.

[26] ——, *TPM Main Part 1 Design Principles, Revision 103*, Trusted Computing Group, July 2007. [Online]. Available: https://www.trustedcomputinggroup.org/specs/TPM/.

[27] M. van Dijk, L. F. G. Sarmenta, J. Rhodes, and S. Devadas, "Securing shared untrusted storage by using TPM 1.2 without requiring a trusted OS," MIT Computer Science and Artificial Intelligence Laboratory (CSAIL), Tech. Rep. Memo-498, May 2007. [Online]. Available: http://csg.csail.mit.edu/pubs/publications.html.

[28] B. Wiese, "Preliminary analysis of a Trusted Platform Modul (TPM) initialization process," Master's thesis, Naval Postgraduate School, Monterey, California, June 2007.