

# The Trusted Platform Module (TPM) and Sealed Storage

Bryan Parno

June 21, 2007

The Trusted Computing Group (TCG) is an organization that promotes open standards to help strengthen computing platforms against software-based attacks [1]. The TCG issued a specification for a Trusted Platform Module (TPM) [4], which is a dedicated security chip designed to enhance software security. The background on the TPM necessary to understand sealed storage is presented below, along with an example of how sealed storage might be used by an application.

## 1 TPM Background

**Platform Configuration Registers.** The TPM contains a number (at least 16) of Platform Configuration Registers (PCRs), essentially internal memory slots. At boot time, all PCRs are initialized to a known value (0 for PCRs 1–16 and -1 for PCRs 17–22). The only way for software to change the value of a PCR is by invoking the TPM operation:

$$\text{PCRExtend}(index, data)$$

When this operation is invoked on the TPM, it updates the value of the PCR indicated by *index* with a SHA-1 hash (H) of the previous value of that PCR concatenated with the data provided. In other words, the TPM performs the following update:

$$PCR_{index} \leftarrow H(PCR_{index} || data)$$

In practice, *data* must be 20 bytes, so larger *DATA* values must be hashed  $H(DATA) \rightarrow data$  before invoking `PCRExtend`.

**Sealed Storage.** At an abstract level, the TPM presents a simple interface for binding data to the current platform configuration (as defined by the PCRs) via the TPM operations:

$$\text{Seal}(indices, data) \rightarrow (C, MAC_{K_{root}}((index_0, PCR_{index_0}), (index_1, PCR_{index_1}), \dots))$$

$$\text{Unseal}(C, MAC_{K_{root}}((index_0, PCR_{index_0}), (index_1, PCR_{index_1}), \dots)) \rightarrow data$$

The `Seal` command takes a set of PCR indices as input, and encrypts the data provided using its Storage Root Key ( $K_{root}$ ), a key that never leaves the TPM. It outputs the resulting ciphertext *C*, along with an integrity-protected list of the indices provided and the values of the corresponding PCRs at the time `Seal` was invoked. For example, to seal a secret key  $K_{secret}$  under the values stored in PCRs 1, 3, and 17, you would invoke:

$$\text{Seal}((1, 3, 17), K_{secret}) \rightarrow (C, MAC_{K_{root}}((1, PCR_1), (3, PCR_3), (17, PCR_{17})))$$

Note that it is also possible to provide the `Seal` command not only with the PCR indices of interest, but also with the values those PCRs should have before `Unseal` will decrypt the data.

The `Unseal` command takes in a ciphertext and PCR list created by the `Seal` command. The TPM verifies the integrity of the list of PCR values, and then compares them against the current values of those PCRs. If they match, the TPM decrypts  $C$  and outputs the resulting data. If any of the checks fail, the TPM simply returns an error. Continuing the example from above, invoking the following `Unseal` command

$$\text{Unseal}(C, \text{MAC}_{K_{root}}((1, \text{PCR}_1), (3, \text{PCR}_3), (17, \text{PCR}_{17})))$$

will produce *data* if

$$\begin{aligned} \text{PCR}'_1 &= \text{PCR}_1 \\ \text{PCR}'_3 &= \text{PCR}_3 \\ \text{PCR}'_{17} &= \text{PCR}_{17} \end{aligned}$$

where  $\text{PCR}'_1$ ,  $\text{PCR}'_3$ , and  $\text{PCR}'_{17}$  represent the current values of those PCRs.

## 2 Sample Implementation<sup>1</sup>

We will consider an application  $\mathcal{A}$  that wants to protect some secret data such that malicious software cannot access it. We assume that the BIOS ( $\mathcal{B}$ ), the bootloader ( $\mathcal{L}$ ) and the operating system ( $\mathcal{O}$ ) have all been modified to support sealed storage. When the computer first boots, ROM code measures (computes a hash) of the BIOS ( $\mathcal{B}$ ) and invokes `PCRExtend` with a canonical PCR index, e.g., 5:

$$\text{PCRExtend}(5, \mathcal{B})$$

As a result, the TPM computes:

$$\text{PCR}_5 \leftarrow H(0||\mathcal{B})$$

The ROM code then starts executing the BIOS. The BIOS performs its usual initialization routines and extends a measurement of the bootloader ( $\mathcal{L}$ ) into the TPM. It could choose a different PCR, but we will assume it continues to use  $\text{PCR}_5$ , so we have:

$$\text{PCR}_5 \leftarrow H(\underline{H(0||\mathcal{B})}||\mathcal{L})$$

Note that the underlined value simply represents the previous value of  $\text{PCR}_5$ . After the `PCRExtend` operation, the BIOS can launch the bootloader. Similarly, the bootloader will extend a measurement of the OS ( $\mathcal{O}$ ) into the TPM before starting to execute it. Finally, the OS will extend a measurement of the application ( $\mathcal{A}$ ) into the TPM and launch the application. At this point, the value of  $\text{PCR}_5$  is :

$$h = \underline{\underline{H(H(H(\underline{H(0||\mathcal{B})}||\mathcal{L})||\mathcal{O})||\mathcal{A})}}$$

The application can generate secret data  $D_{secret}$  and seal it under the current value of  $\text{PCR}_5$  by invoking:

$$\text{Seal}((5), D_{secret}) \rightarrow (C, \text{MAC}_{K_{root}}(5, h))$$

---

<sup>1</sup>Much of this example is similar to a secure or trusted boot process. With a secure boot architecture (such as AEGIS [2]), the boot process can be terminated if the software to be loaded at each stage fails to match known-good values stored in the TPM. Alternatively, the platform can perform a trusted boot [3] by measuring each piece of software loaded, and storing these measurements in the TPM for later attestation to a third party.

What benefit does this provide? If the same boot sequence is repeated (in other words, if the exact same BIOS, bootloader, OS and application are loaded in the same order) then clearly  $PCR_5$  will take on the same value it had before. Thus a call to `Unseal` will produce  $D_{secret}$ . However, if any of these pieces of software changes, then the `Unseal` will fail. For example, suppose an attacker replaces the OS with a malicious OS ( $\hat{O}$ ). When the application is executed, the value of  $PCR_5$  will be:

$$\hat{h} = \underline{\underline{H(H(H(H(0||\mathcal{B})||\mathcal{L})||\hat{O})||\mathcal{A}))}}$$

The properties of the hash function  $H$  guarantee that with extremely high probability  $\hat{h} \neq h$ , and thus if an attacker invokes `Unseal`, the TPM will refuse to decrypt  $C$ . Note that this example is highly simplified; the TCG specifications [1] precisely define both what and when code should be measured.

### 3 Limitations

An important limitation of TPM-based protections, including sealed storage, is that the TPM only provides a guarantee at software load time. Thus, if a legitimate application is loaded and then modified in memory, the TPM will not be aware of the change. In other words, the TPM does not protect against an attacker who exploits a flaw in currently executing software. Also, as noted above, the legitimate software must be modified to extend measurements into the correct PCRs, and secrets must be sealed under the correct PCRs.

### References

- [1] Trusted Computing Group. <http://www.trustedcomputinggroup.org/>, March 2005.
- [2] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, 1997.
- [3] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.
- [4] Trusted Computing Group. Trusted platform module main specification, Part 1: Design principles, Part 2: TPM structures, Part 3: Commands. <http://www.trustedcomputinggroup.org/>, March 2006. Version 1.2, Revision 94.