RC23363 (W0410-029) October 6, 2004
Computer Science

IBM Research Report

The Role of TPM in Enterprise Security

Reiner Sailer, Leendert Van Doorn, James P. Ward
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

The Role of TPM in Enterprise Security

Reiner Sailer, Leendert van Doorn, James P. Ward

Establishing trust in a remote computer system is an essential building block for distributed systems. Unfortunately trust is a hard property to achieve without appropriate hardware support. In this article we describe our trusted computing platform where we extend the hardware rooted trust guarantees of TCG technology to the (Linux) operating system and all its applications and allow remote parties to verify these trust guarantees.

Reiner Sailer

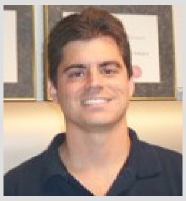
Research Staff Member
IBM Research
Secure Systems

E-Mail: [sailer@watson.ibm.com]

Leendert van Doorn

Senior Manager
IBM Research
Secure Systems

E-Mail: [leendert@watson.ibm.com]



James P. Ward

President Trusted Computing Group

Senior Technical Staff Member, IBM Software Group

E-Mail: [jpward@us.ibm.com]

Introduction

Establishing trust between entities is becoming an increasingly important requirement in today's highly connected and distributed Enterprise business environment. Business critical capabilities such as remote access, distributed workforce, dynamic outsourcing, and business portals all implicitly depend on mechanisms for verifiably establishing the authenticity and integrity of the connected devices, processes, and services. Emerging business models and architectures such as GRID, On Demand, and Utility Computing will further emphasize the need for determining these trust attributes in a standardized and interoperable manner.

The Trusted Computing Group (TCG, [1]) specifications are intended to provide an open set of security related building blocks for enhancing the trust associated with a computing platform. These common building blocks are developed to be platform and vendor agnostic such that they can be applied into any device type (i.e. PCs, servers, mobile phones, embedded devices), operating system (e.g., Linux, Windows, UNIX, Solaris), or solution framework (Web Services, etc). The TCG trust model is based on establishing a common assurance root and function definition for these trust characteristics. The TCG Trusted Platform Module, or TPM, serves as the starting point, or root, for this transitive trust model. The TPM as core root of Trust for measurement, or CRTM, can measure additional system attributes and then later verifiably report them as a basis for determining the overall trustworthiness of a platform. The TPM must be and is designed to be initially trusted because it represents the start of the trust chain. The measure, record, and report process is cumulatively referred to as attestation. The TCG function does not qualitatively assess what the information means in terms of trustworthiness. Rather, information reported via the TCG building blocks can be considered trustworthy. The manner this information is used to determine trust is a function of policy outside of the existing TCG standards.

The TCG standardizes the measurement and reporting of attributes covering trust establishment into the boot process of a system. However, this does not yield information about the trustworthiness of the runtime environment built on top of it. We solve this problem by extending the measurement from the static boot sequence into the dynamic runtime of the operating system (OS) and enabling the attestation of properties of a system’s runtime.

The remaining of the paper is organized as follows: in Section 1, we introduce attestation based on TPM hardware; in Section 2, we present our architecture in detail and explain how we extend attestation into the system runtime. Section 3 describes the major results and shows at an example how we can detect a compromised system runtime using attestation. We conclude in Section 4 describing future work in this area.

1 TPM-based Attestation

The TPM represents a separate trusted coprocessor, whose state cannot be compromised by potentially malicious host system software. TPM-based attestation represents a powerful tool for establishing the trust attributes of a system. Attestation based information about the device hardware, firmware, operating system, and applications can all be dynamically assessed to determine if the system should be trusted prior to granting a privilege (network / resource access, service, etc).

Unlike secure boot, which loads only signed and verified software, the TCG trusted boot process only takes measurements up to the bootstrap loader and leaves it up to the remote party to determine the system's trustworthiness. Thus, when the system is powered on it transfers control to an immutable base. This base will measure the next part of BIOS by computing a SHA1 secure hash over its contents and protect the result by using the TPM. This procedure is then applied recursively to the next portion of code until the OS has been bootstrapped.

We adjust the role of the TPM by using it to protect the integrity of the in-kernel measurement list rather than holding measurements directly. To prove to a remote party what software stack is loaded, the system needs to present the TPM state using the TCG attestation mechanisms and this ordered list. The remote party can then determine whether the ordered list has been manipulated and, once the list is validated, what kind of trust it associates with the measurements. We have modified the Linux kernel and the runtime system to take integrity measurements as soon as executable content is loaded into the system, before it is executed [2]. We maintain the ordered list of measurements inside the Linux kernel.

Unlike existing approaches, such as secure boot or authenticated boot [3], where a system is instrumented to boot only signed and verified software, or secure coprocessors [4], which offer a closed environment to run certified and signed software in a protected environment, our approach is non-intrusive and does not change the behavior of the system that is being attested. It can be used in open environments where a large spectrum of software runs and changes regularly.

2 Mutual Attestation

In this Section, we demonstrate the power of TPM-based remote attestation in the process of making informed decisions about trust for Web services, and help to clarify how these concepts could be used in an open environment. Mutual platform attestation is the process by which peers in a transaction, with potentially no previous knowledge of each other, can establish trust based on the integrity of each other's computing environment – e.g., that a peer is truly running the service being offered in an environment that is acceptable.

Prior to the actual transaction, peers exchange integrity measurements of each other's environment – e.g., fingerprints of all the software running on each system. As trust decisions are then based on these measurements, authenticity is a critical factor. Our architecture uses the Trusted Platform Module (TPM) to protect and assure the validity of the integrity measurement for each executable that is loaded into the OS runtime. Thus, coupled with the integrity measurements of the boot process from system firmware through OS program load (which also uses TPM), remote parties are assured the authenticity of the integrity measurements since system boot and can make appropriate trust decisions based on service requirements. This approach is applicable in a host of other scenarios including remote Systems Management and Service Level Agreement or Quality of Service verification on-demand.

Goal: Our goal is to measure what is useful and necessary for a challenging party to regenerate the software stack of an attested system securely and to determine trusted properties of the attested system. We instrumented the Linux kernel to create and store such measurements as well as protect them against compromised systems by using the TPM hardware. Our approach is non-intrusive in the sense that it will prevent a system neither from becoming compromised nor from manipulating the kernel-held measurements. However, we do prevent such a system from posing as a non-compromised system by allowing challenging parties to independently validate the attested party's integrity by means of the received measurement list.

Assumptions: We assume that the TPM works correctly and its operation is not manipulated using physical attacks (which it is not designed to withstand at this time). We also assume that the challenging party possesses a valid certificate to the signature key used by the TPM of the attested system.

Limitations: As we measure data when it is loaded, vulnerabilities propagated by running software – e.g. through buffer-overflow exploits – will not be represented in the measurements. However, the known potential of executables to become compromised during run-time is well represented in the measurements. Challenging parties can derive the identity of the program and its version from the measurement and relate it to known vulnerabilities, e.g., using CERT [10] data bases when deciding whether to trust this part of the software stack.

2.1 Experiment Setup

Figure 1 gives an overview of the process followed in our experiment. We distinguish two independent systems: the *attested system* – the system whose software-stack is to be validated – and the *challenging party system* that is going to validate the software-stack of the attested system.

The attested system is instrumented to produce evidence (called measurements, see Fig. 1, step 1) that allow challenging parties to re-create its run-time safely. The attested system contains a TPM security chip that protects the integrity of the created evidence even if the attested system should become compromised later on. In response to the demand of an authorized challenging party, the attested system returns its evidence and also provides related contents of the security chip that allows the challenging party to validate the integrity of the provided evidence (i.e., the measurements).

The challenging system requests from the attested system the actual measurement list as well as the integrity value over the list, which is stored inside the TPM, validates the integrity of the list (step 2), evaluates the individual measurement, and finally reconstructs an image of the attested system's software stack (step 3). Based on this image, the challenging system concludes about properties of the attested system's runtime (step 4). Exchanging the roles of attested and challenging system then implements mutual attestation.

The following subsections describe the instrumentation of the attested system to produce and protect evidence about the loaded software stack since reboot (2.2), the establishment of trust into the evidence of a system (2.3), and the interpretation of evidence to conclude properties of the software stack of the attested system (2.4).

2.2 System Instrumentation

We have instrumented the Linux kernel [2] of the attested system to produce measurements of post-boot events that affect the run-time of the system. Our measurements are taken in a way that allows (remote) parties to securely reconstruct what was actually loaded into the software stack of a system and to determine if this system can be trusted according to the local security policy. To establish trust into the instrumented Linux kernel, preceding boot stages produce measurements through the TPM hardware, the BIOS, and the Grub boot loader stages, each stage in turn gathering and storing information about the attested system's next boot stage up to the running kernel.

As opposed to other approaches, e.g., terra [8] measuring whole partition contents of virtual machines, we aim at representative measurements of software stack components that are rich in semantic value and allow challenging parties to reconstruct functional properties of the actual software stack. Therefore, we instrumented the Linux kernel running on the attested system to create such evidence based on which the software stack can be reconstructed.

What we measure: The goal is to create *representative evidence* that can be interpreted by a challenging party in order to decide whether loading the represented data maintains or breaks the trust into the overall software stack of a system.

We consider the following information about loading data into the run-time as being representative evidence:

- Kernel modules – they potentially affect the measurement architecture in the kernel
- Executables and shared libraries – they don't change often and can be related to functionality as well as known vulnerabilities
- Configuration files – they don't change often once the system is correctly configured. Additionally, configurations can be decisive for the trustworthiness of the program consuming and interpreting them.
- Other important input files that affect trust into run-time software stack, e.g., Bash command files, Java Servlets, and java libraries.

We don't consider measuring dynamic input data such as user input, web requests, and remote commands because they would not lead to representative semantic value. Vulnerabilities based on such data are better addressed by operating system access control mechanisms (e.g., SELinux [11]), which are represented in the kernel measurement and available to attesting parties for trust establishment.

How we measure: A measurement is implemented as the computation of the 160bit result of a SHA1 hash function (fingerprint) applied to the file that contains data or executables loaded into the run-time. The slightest difference in a data file will generate a distinguished fingerprint and, hence, variations in programs (e.g., due to viruses or Trojan horses) are easily detected by differing measurement values.

In order to prevent attested systems from unnoticed cheating, we have integrated functions of the TPM into the measurement architecture:

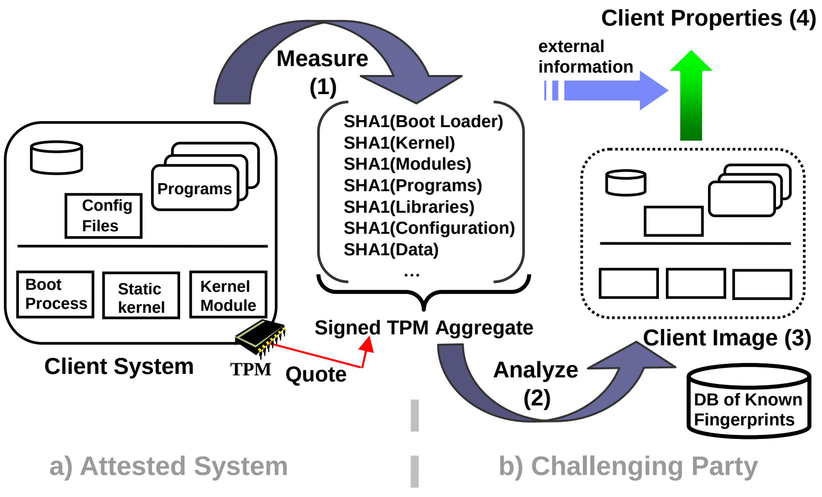


Figure 1: Attestation Architecture Overview

- We use the TPM to maintain an integrity value (stored in its protected hardware) over the current measurement list that is kept in the kernel. This protects the measurement list from being manipulated unnoticedly even if the system and kernel become corrupted.
- We create measurements of files before they are loaded and potentially affect the system. Thus, once loaded data can corrupt the measurement list, it is too late to cover its own traces from the TPM.

Our TPM (Version 1.1) offers 16 platform configuration registers (PCR) that allow extending 160 bit numbers (length of a SHA1 value) into them. These PCR are reset to 0 whenever the system is reset (e.g., reboot). The first 8 PCRs (PCR0 – PCR7) are used for attesting the booting steps, the remaining 8 PCRs (PCR8 – PCR15) are allocated for use by the booted system [9]. We use a configurable PCR number greater than 7 (e.g., 10) to maintain an integrity value over the current measurement list after system boot. If a new measurement is added to the measurement list, we also write its 160bit measurement value into TPM PCR 10. The TPM computes the new register content by building a SHA1 over the current content concatenated with the new 160bit number written into the PCR. The cryptographic properties of SHA1 (being collision-free) guarantee protection against the adaptation of a TPM PCR to match a manipulated measurement list by compromised systems later on.

We refer the interested reader to [2] for further details of the integrity measurement architecture and its implementation. Figure 2 shows a partial snapshot of a measurement list for a Redhat Linux system including executables, shared libraries, kernel modules, bash command files (e.g., server initialization scripts) and bash source files (e.g., bash configuration files). We include some additional information in our kernel-held measurement list, such as the file name of the measured file. Our Web-based project description [5] includes a complete measurement list including measurements collected during system boot.

```
 # SHA1(160bit)  File          Type

000:D6DC…D3DB n/a       [boot aggregate]
001:84AB…DA4F init           [exec]
002:9ECF…BE3D ld-2.3.2.so    [library]
003:3365…2342 libc-2.3.2.so  [library]
004:A4DC…C12B bash           [exec]
…
027:2AC8…980D clock          [bash-src]
028:C0F7…9A3D hwclock        [exec]
…
070:01B3…9A1E rc             [bash-cmd]
071:CEBA…1AA4 runlevel       [exec]
072:2998…8ED4 egrep          [bash-cmd]
073:6846…B72D kudzu          [bash-cmd]
…
080:147D…8168 parport        [module]
081:F940…0115 parport_pc      [module]
…
244:D312…DA7C rc.local       [bash-cmd]
245:BB2C…AAB3 mingetty       [exec]
```

Figure 2: Measurement List Example

The measurement list is always initialized with the boot aggregate representing the measurements of the boot stages up to and including the running kernel. The actual measurements – aggregated into the boot aggregate – are stored in the BIOS as the kernel is not yet running. They are protected by specific PCRs throughout boot-time [9], aggregated and included into the measurement list once the kernel is running, and can thus be verified later throughout the attestation. Subsequently, the init program is shown, which controls the rest of the system boot. Every program or data file is measured and its evidence is added to the measurement list if it wasn't recorded before.

The measurement list of a Redhat Linux system running an Apache web server and Jakarta Tomcat Servlet machine or X windows, and the Gnome desktop system collects about 400-600 measurements.

2.3 Measurement Integrity Validation

The initial step of the trust establishment process consists in the challenging party retrieving from the attested system the current measurement list and the signed PCRs necessary to validate the integrity of this list. For this purpose, the *challenging party* sends a random number RN to the attested system.

The *attested system* first validates the authorization of the challenger. Thus, the attesting system controls the release of its potentially sensitive state-information. If the challenging system is authorized, then the attested system returns its current list of measurements (in the order they where collected) and a quote from its TPM including the random number RN. The TPM will quote its PCR registers by signing them with a 2048bit RSA signature key that was created inside the TPM and to which the public key was securely certified as belonging to this TPM [1]. This signature also includes RN and is done inside the TPM hardware.

The validation of the measurement list by the challenging party consists of the following steps:

- Verify the signature of the TPM quote. This determines i) whether the quoted PCR values are tampered with or not, and ii) whether the quoting TPM is actually the one on the attested system.
- Ensure that the signed random number equals RN. This ensures that the quote is not a replay attack by a compromised system, as long as the chosen RN is unpredictable by the attested system.
- Calculate the boot aggregate by computing SHA1(PCR0 || … || PCR7). Compare it to the first measurement of the measurement list, which is supposed to be exactly this boot aggregate. If they don't match, the attestation fails. This step links the boot measurements to the run-time measurements.
- Recalculate virtually the PCR value for the run-time measurements in the measurement list. To do so, start with virtPCR=0 and with the first (oldest) measurement M_0 of the list (here: boot-aggregate). Calculate virtPCR := SHA1(virtPCR || M_0); continue with the next measurement until the measurement list is consumed. The resulting value in virtPCR must now match the value of the signed TPM PCR that was used by the attested system to protect the integrity of the measurement list (in our case PCR10). If the values don't match, then the measurement list must be assumed tampered and the attestation fails. This can happen if the attested system is compromised and tries to cheat or if the measurement instrumentation of the attested system has recognized suspicious system behavior and invalidated the measurement PCR pessimistically (fail-safe measurement-bypass protection).

Now that trust into the correctness of the measurement list is established, every measurement list entry must be validated to build trust into the software-stack of the attested system.

2.4 Software Stack Measurement Analysis

In order to establish a trust chain from the TPM hardware root-of-trust into the current run-time of a running system, we distinguish two parts of the chain. The first chain extends from the TPM over the boot stages to the running Linux kernel. The second chain is maintained by the running kernel and extends over the uptime of the attested system starting with the first file loaded (here: init). The two chains are securely linked by the running kernel extending an aggregate over the first part of the chain as first measurement into the PCR that represents the aggregate of the second chain.

To establish the trust chain from the TPM hardware to the running Linux kernel, we need to keep track of all steps during the system boot in order to ensure that the next step will continue to measure the succeeding step correctly. To jumpstart this process, initial trust is necessary and placed into the correct implementation and embedding of the TPM hardware into the system platform. Then, trust into the measurement representing the boot BIOS is necessary, including its property to measure the succeeding boot steps correctly (Master boot record). This process continues to the boot loader (here: Grub) and finally the kernel being measured before it becomes active. All these measurements are protected by Platform Configuration Registers PCR0 – PCR7 as standardized for PC architectures in [9]. Validating the pre-kernel boot process means trusting the code that was executed throughout these stages based on the collected SHA1 fingerprints and resulting aggregates in these PCRs.

Using this trust model, an attesting party can establish trust into the kernel and its (measurement) properties through the completeness and integrity of the measurement chain from the root-of-trust (TPM) up to the kernel. If any of the intermediate fingerprints is not trusted, then the kernel cannot be trusted because any measurements following the distrusted fingerprint cannot be guaranteed to represent the following loaded stage correctly. A measurement can therefore only be trusted if its represented code is known to correctly measure the actually loaded code taking over the next boot stage and to protect the measurement in the TPM PCR as specified in [9]. Unknown fingerprints or fingerprints of known malicious code break this trust chain. Only configuration changes of the boot sequence and rebooting the system (resetting the TPM PCRs) can re-establish trust into a distrusted measurement chain.

The aforementioned boot measurements are pretty static (regarding order and fingerprint value). Thus, we can simply check the boot PCRs 0–7 against a set of permitted values. If they match, the first part of the chain is trusted. If not, the boot sequence is not trusted and the attested system fails the test based on the challenger's policy.

We focus in the following on attesting to the much more dynamic software stack established by the running system kernel to extend the established chain of trust into the run-time of the system. Here, the changing order of measurements and the dynamic program versions will lead to a very large range of possible PCR values even for similar systems. Therefore, rather than attesting to a predetermined aggregated PCR value, every single measurement is validated and evaluated as trusted or distrusted according to the policy of the challenging party. The overall attested client's run-time image is then build bottom up using these measurements and known properties of the represented part of the software stack (e.g., program or configuration file).

We trust a measurement if and only if the following conditions hold:

- We know what it represents, e.g., executable functionality or configuration file and how it affects the run-time of the attested system.
- The represented data loaded into the system run-time does not compromise our measurement instrumentation in a way that prevents future correct and complete measurements. If, for example, we expect all executed bash script files to be measured, then we won't trust a measurement representing a loaded bash shell that does not induce such measurements. Another example is a loaded kernel module that compromises the kernel instrumentation and prevents complete measurements in the future. Even such a kernel module – compromising and taking over the whole system software stack – cannot eliminate its own measurement without invalidating the integrity value kept in the TPM PCR.
- The measured data is assumed to work correctly after loading it into the run-time even in the assumed presence of attackers. This means that potentially known vulnerabilities in the represented data (e.g., local or remote exploits) are considered acceptable by the policy.

The above evaluation is done by the attesting party only once (unless policy changes) for each program or configuration file and stored together with its SHA1 value in a so-called known-fingerprints data base. Thus, when evaluating a measurement, the challenging party looks up the respective 160bit fingerprint in its policy data base and extracts directly the information about the trustworthiness of this fingerprint under the active policy. More complex analysis could also relate multiple measurements to each other, e.g. to ensure that interdependent programs are interoperable.

Any program or configuration, whose fingerprint is unknown (potentially malicious), could corrupt the system and prevent future correct measurements. Thus, evaluation can stop here because later measurements and the protection thereof cannot be trusted to represent the real software-stack of the attested system. An example would be a malicious kernel module corrupting the kernel by intercepting measurement requests and hiding malicious software being loaded into the system. Malicious components could then be loaded without measurements being taken and thus without evidence being produced.

In conclusion, the software stack validation is successful only if all individual measurements taken on the attested system are trusted by the challenging system.

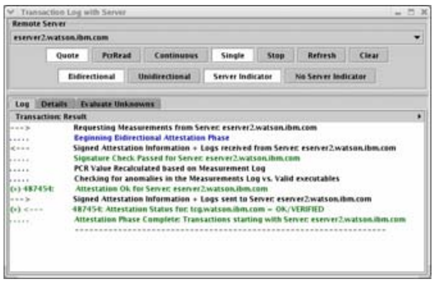
Future work includes partitioning of the measurement space and allowing for finer-grained evaluation of measurement lists. This could mean to allow unknown software to be loaded on the attested system (represented by measurements unknown to the challenging party) as long as its impact is controlled and does not affect the security of other parts of the run-time that might be of interest to the challenging party. Trust into such strong isolation between system parts can be justified by secure virtualization or by mandatory security enforcement in the kernel (e.g., SELinux [11]).

3. Results

We implemented mutual attestation on two Redhat Linux systems running our instrumented Linux 2.6.5-bk2-lsmtcg kernel [5] and open-source TPM drivers [6,7]. The attestation service is implemented as a Web service running in a Jakarta Tomcat Container. The database of known fingerprints is compiled for each system independently by measuring existing executables and libraries and attaching *trusted* or *distrusted* labels and comments. In our case, the data base had about 20 000 entries, 5 of which where fingerprints of known Linux Rootkit exploits [12]. We have supplied both machines with valid certificates of TPM keys that were created on the peer system for validating the signed TPM PCRs.

We use a Java GUI (c.f. Figure 3) to initiate and control the mutual attestation of the two systems named *Tcg* and *Eserver2*. *Tcg* initiates the mutual attestation by calling the attestation Web service on Eserver2, providing a random number RN1.

Eserver2 answers with the current measurement list and the signed TPM PCR values including RN1. *Tcg* validates the signature over the TPM PCRs, then validates the included random number (nonce) and recalculates the assumed PCR aggregate using the measurement list. If the computed aggregate matches the value of the signed TPM PCR10, then the measurement list is successfully validated. Afterwards, *Tcg* runs through the measurement list, looking up the known fingerprint database for each measurement value in turn.



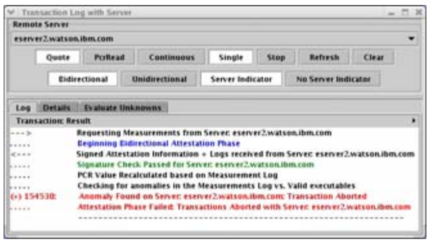
```
--->              Requesting Measurements from Server: eserver2.watson.ibm.com
.....             Beginning Bidirectional Attestation Phase
<---              Signed Attestation Information + Logs received from Server: eserver2.watson.ibm.com
.....             Signature Check Passed for Server: eserver2.watson.ibm.com
.....             PCR Value Recalculated based on Measurement Log
.....             Checking for anomalies in the Measurements Log vs. Valid executables
(+) 487454:        Attestation Ok for Server: eserver2.watson.ibm.com
--->              Signed Attestation Information + Logs sent to Server: eserver2.watson.ibm.com
(+) <---          487454: Attestation Status for: tcg.watson.ibm.com = OK/VERIFIED
.....             Attestation Phase Complete: Transactions starting with Server: eserver2.watson.ibm.com
                  ---------------------------------------------------------------------
```

Figure 3: Demo GUI showing successful mutual attestation

If at any time, it does not find the measurement value or if the data base tags the value as distrusted, the validation fails. The attestation succeeds if all measurements are found and trusted. Following this attestation is the reverse attestation of *Tcg* against *Eserver2*, which proceeds symmetrically. Adapting the database of known fingerprints, we can as well validate the measurement list against service level agreement policies.

3.1 Detecting Compromised Systems

We have successfully deployed our prototype for detecting Rootkit-exploits [12].

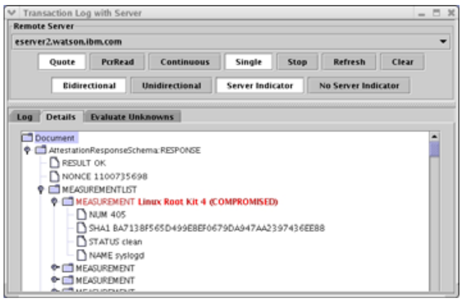


```
--->              Requesting Measurements from Server: eserver2.watson.ibm.com
.....             Beginning Bidirectional Attestation Phase
<---              Signed Attestation Information + Logs received from Server: eserver2.watson.ibm.com
.....             Signature Check Passed for Server: eserver2.watson.ibm.com
.....             PCR Value Recalculated based on Measurement Log
.....             Checking for anomalies in the Measurements Log vs. Valid executables
(+) 154530:       Anomaly Found on Server: eserver2.watson.ibm.com: Transaction Aborted
.....             Attestation Phase Failed: Transactions Aborted with Server: eserver2.watson.ibm.com
                  ---------------------------------------------------------------------
```

Figure 4: Demo GUI showing exploit

Figure 4 shows the main GUI window for the case that a distrusted measurement is found in the measurement list.

Figure 5 shows the details of the measurement that was responsible for failing the attestation: a Syslogd (audit program) that is part of a Rootkit exploit; it replaces the original Syslogd program and contains hidden code that covers traces of attackers. It is safely distinguished from the non-compromised Syslogd by its differing SHA1 hash value. Along with this compromised root kit program, there are usually other programs installed that include hidden functionality and allow attackers to bypass normal system access control.



```
Document
  AttestationResponseSchema:RESPONSE
    RESULT OK
    NONCE 1100735698
    MEASUREMENTLIST
      MEASUREMENT Linux Root Kit 4 (COMPROMISED)
        NUM 405
        SHA1 BA7138F565D499E8EF0679DA947AA2397436EE88
        STATUS clean
        NAME syslogd
      MEASUREMENT
      MEASUREMENT
```

Figure 5: Measurement indicating exploit

Although those specific programs are not shown in the figure, they are detected as well.

3.2 Overhead

The instrumentation and measurement overhead for creating and maintaining the measurement list on the attested system is negligible. A new measurement, which occurs mostly throughout the booting of the system, incurs overhead for computing the SHA1 hash value over the file to be loaded and additionally about 5ms overhead for extending the new measurement into the TPM PCR. We measured a throughput of about 80 Megabyte/second for computing the SHA1 hash value in the Linux kernel. Remeasuring files that were measured before is very efficient because we use dirty-flagging and caching mechanisms that skip computing the SHA1 value for files that cannot have changed since the last time they were measured. As a result, re-loading a file that was already measured before incurs less than 1 microsecond overhead.

We experienced a latency of about 1 second for a single unidirectional attestation and about 2-3 seconds for mutual attestation via the non-optimized Demonstration GUI. This includes the 2048bit TPM RSA signatures of the PCRs, the communication of the Web services exchanging the measurements, as well as validating the measurement list and comparing the measurement list entries against the known fingerprint database. In conclusion, the overhead introduced by our technique is negligible for most application scenarios.

4. Outlook

The guiding principles for the system described in this article were low-overhead and the separation of measurements and verification. The latter was important because we did not want to limit the remote party in what programs it can execute.

The system presented here is an essential first step in establishing a trusted platform. However, it dose not consider the following areas, which we are addressing in our future work:

- *Scalability*. The list of individual fingerprints of programs, libraries and scripts do not scale very well. In an enterprise setting this is less of an issue, typically only few software configurations exists and patches are applied centrally so it is easier to maintain an enterprise wide database of *trusted* programs.
- *Trust*. Trust is in the eye of the beholder. While each recipient of attestation statements could derive its own trust level, it would probably delegate this to trusted-third-parties and make the decision for it based on, for example, a company policy. On a world-wide scale this is analogous to a PKI infrastructure which has its own set of challenges.
- *Privacy*. The individual measurements give potential attackers a wealth of information about the system at hand. Clearly, this is undesirable.
- *Isolated execution*. The TPM, while an excellent trust anchor, is passive. For many applications, active trusted components such as compliance checkers or monitoring agents are needed. Trust into such components can only be established through strong isolation.

Conclusion

We have shown one example of how the TPM security chip can be used to establish trust into previously unknown systems. In addition to the here discussed role as Root of Trust for Reporting, the TPM also implements functions that make it suitable as a Root of Trust for Storage, supporting the local system to protect storage and to implement its own security mechanisms based on trusted hardware functions.

TPM hardware –implementing open interface specifications– has the potential to become the foundation for many trust-establishment processes needed in vital emerging and established areas, such as On-Demand environments, Autonomic Computing, and Web Services. A crucial role herein plays the TPM's protection against the system software, which makes it suitable as a root-of-trust.

Acknowledgments

The authors wish to thank Emily Ratliff and Kylene Hall and their colleagues from the IBM Linux Technology Center for their support, Seiji Munetoh and his colleagues from the IBM Tokyo Research Lab for their boot-loader enhancements, and our colleagues Trent Jaeger, Xiaolan Zhang, Ronald Perez, Anca Sailer, and Steve Bade for their useful comments.

Literature

[1] Trusted Computing Group: *Trusted Platform Module Main Specification Part 1: Design Principles, Part 2: TPM Structures, Part 3: Commands*. October 2003, Version 1.2, Revision 62, https://www.trustedcomputinggroup.org.

[2] R. Sailer, X. Zhang, T. Jaeger, L. Van Doorn: *Design and Implementation of a TCG-based Integrity Measurement Architecture*, 13th Usenix Security Symposium, California, August 2004.

[3] W. A. Arbaugh, D. J. Farber, J. M. Smith: *A Secure and Reliable Bootstrap Architecture*, in IEEE Computer Society Conference on Security and Privacy, 1997.

[4] J. Dyer., M. Lindemann., R. Perez, R. Sailer, S. W. Smith, L. van Doorn, S. Weingart: *The IBM Secure Coprocessor: Overview and Retrospective*, IEEE Computer, October 2001.

[5] IBM Watson Research – Secure Systems Department: *tcgLinux – TPM-based Linux Run-time Attestation*, http://www.research.ibm.com/ secure_systems _department/projects/tcglinux.

[6] David Safford, Jeff Kravitz and Leendert van Doorn: *Take Control of TCPA*, Linux Journal No. 112, August 2003.

[7] IBM Watson Research – Global Security Analysis Lab: *TCPA Resources*, http://www.research.ibm.com/gsal/tcpa.

[8] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, D. Boneh: *Terra: A Virtual Machine-Based Platform for Trusted Computing*, Proc. 9th ACM Symposium on Operating System Principles, 2003.

[9] TCG PC Specific Implementation Specification, Version 1.1, August 2003.

[10] CERT Coordination Center, http://www.cert.org/.

[11] National Security Agency. Security-Enhanced Linux (SELinux). http://www.nsa.gov/selinux, 2001.

[12] John Levine, Brian Culver, Henry Owen: A Methodology of Detecting New Binary Rootkit Exploits. Proceedings IEEE SouthEastCon 2003, April 2003.