Attacking a Trusted Computing Platform
Improving the Security of the TCG Specification

Technical Report RT 05-05

D. Bruschi, L. Cavallaro, A. Lanzi, M. Monga
Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano
Via Comelico 39/41, I-20135, Milano MI, Italy
{*bruschi, sullivan, andrew, monga*}*@security.dico.unimi.it*

Abstract

We describe a flaw which we individuated in the Object-Independent Authorization Protocol (OIAP), an authorization protocol which represents one of the building blocks of the Trusted Platform Module (TPM), the core of the Trusted Computing Platform (TP) as devised by the Trusted Computing Group (TCG) standards. In particular we show that the protocol fails to protect messages exchanges against straight replay attacks. Using such a flaw an attacker could compromise the correct behavior of a TP, thus undermining its main property namely trust. A proposed solution, which requires the modification of the OIAP authorization protocol in order to provide "real" protection against replay attack, has been devised and described as well.

1 Introduction

Security is probably one of the Computer Science fields where the research community has spent most of its efforts in these last years. Since the beginning, it was clear that the construction of a secure system, i.e., a system which satisfies the confidentiality, integrity and availability properties, under any circumstances, is not feasible. The main difficulties in reaching such an objective are represented by the fact that computer security is based on the chain paradigm, i.e., a computer system is only as "strong" as its weakest link. Thus, the security property has to be verified by any secure component of the software stack, the firmware, the hardware and, of course, the environment in which the system operates and, in general it is impossible to prove if a complex system computes exactly what its designer want.

On the other hand, security is a mandatory requirement for many computer applications. Thus, researchers in this last years proposed various security solutions aimed at improving the security of a platform by reducing its exposure to attacks (see for example perimetrical defenses, strong authentications, antivirus, OS memory protections and so on).

From the architectural point of view the main characteristic of these solutions was based on the idea of providing an external layer of protection to the various objects (OS, application) "without interfering" with the object itself. Even if such an approach has been proven quite effective in protecting computer system, it also has shown some drawbacks. The most important one is the limited capacity of blocking unknown forms of attacks.

Many proposals appeared in literature in order to overcome such a limit: one of the most relevant contribution in such a direction is the research on Trusted Computing Platforms (TP) inspired by a work of Arbaugh et al. [8]. Roughly speaking a

Trusted Computing Platform is characterized by the property of either computing according to its "initial specification" or promptly detecting any unauthorized modification of its components.

Various proposals of TP appeared recently in literature (see [12, 21]); in this paper we will investigate the solution proposed by the the Trusted Computing Group (TCG) [4], a multi-vendor consortium formerly known as the Trusted Computing Platform Alliance (TCPA). The TCG has proposed several specifications for systems that, by using a modified BIOS and a supplementary chip hardwired on the motherboard, can systematically verify the integrity of each software component. The basic building block of a trusted platform is the Trusted Platform Module (TPM), a hardware component which is supposed to verify the integrity of the system, and grant the access to protected resources only to trusted components.

In order to grant this access, the TPM refers to a couple of authorization protocols, by which it verifies the credentials of principals interested in having access to critical protected resources. These protocols are the *Object-Independent Authorization Protocol (OIAP)* and the *Object-Specific Authorization Protocol (OSAP)* and they are defined in the TCG specification[1] [7].

These protocols were specifically designed to resist to *replay* and *Man-in-The-Middle (MiTM)* attacks, with the introduction, respectively, of the so-called *"rolling nonces"* paradigm and *Keyed-Hashing for Message Authentication (HMAC)* [2].

In this paper we will show that the OIAP specification issued by TCG, is not strong enough to protect a system from every replay attack. In fact, we discovered that when multiple sessions are on going, a replay attack is possible, and an intruder can subvert the correct behavior of a TP, without no one noting it. In particular, using such an attack, an intruder can modify memory regions protected by the TPM, compromising the correct behavior of the platform and, more importantly, without being detected. This last issue is very important, as it undermines one of the most fundamental properties of a TP and the one on which the *trust* towards the platform is built, namely promptly recognize an unauthorized modification of data.

[1]Version 1.2 of the TCG specification introduces another "authorization" protocol, DSAP, but its treatment is beyond the scope of this paper.

However, we have been able to devise a solution which will make the OIAP immune to replay attacks as the one described in this paper.

The paper is organized as follows: in Section 2 we describe the notation used throughout the paper, in Section 3 we recall the main concepts of the TCG specification with a focus on the authorization protocols (Section 4), in Section 5 we describe the attack we found and a possible exploitation (Section 7, in Section 7 we propose a modification of the protocols to protect a system against our attack, and finally, in Section 8, we draw some conclusions.

2 Notation

In this section we describe the notation which will be used throughout the paper which is required for a better understanding of the manuscript.

In the following we will use capital characters for denoting strings over the binary alphabet, while capital **bold** letters will denote generic entities, such as hosts, hardware/software components, generic users and so on.

- $X.Y$ denotes X concatenated to Y;
- $\mathbf{A} \rightarrow \mathbf{B}$: M denotes that A sends the message M to B;
- S_{i} denotes the identifier of the *i-th* authorization session;
- $A', A'', \cdots$ indicates further "instances" of an object A; for example, if A is a pseudo random number, A' would indicate another pseudo random number, different from A, generated from the same pseudo-random number generator, so that A and A' have the same features.

3 Trusted Computing Platforms

Security mechanisms may themselves be modified by malicious code [1, 10] so that it may be hard to detect such a violation. This is the key consideration that led the Trusted Computing Group (TCG) [4], to propose a new architecture, the so-called Trusted Computing Platform, which proposes to protect critical parts of a system by using cryptographic protocols and other features described ahead, implemented at the hardware level.

The specification defined by the TCG [7] states that Trusted Platforms (TPs) are computing platforms that add to themselves the property of *trust*. In other words, they provide proper mechanisms to verify, in a secure way, that the data yielded by them were not tampered with. When a manipulation is performed, a "violation" is detected and reported to the user who will decide whether to trust or not the data provided by the TPs.

The TPs trust mechanism is based on two hardware components, namely the *Core Root of Trust for Measurement*, CRTM, and the *Trusted Platform Module*, TPM. Both build up the so-called *root of trust*, they are hardwired on the TP motherboard and have to be "trusted"[2] in order to consider the whole computing platform a TP.

Generally speaking, TP can provide three main functionalities:

1. *identity*: a TP can be identified in a unique and secure way;

2. *measurement*: a "complete" integrity snapshot of software and hardware components of the TP may be computed;

3. *protected storage*: the TPM acts as a cryptographic portal, providing protection to sensible data (passwords, cryptographic keys, passphrases, data and so on).

[2]Trust here means that the behavior of these components have to be certified by trusted third parties.

A TP starts its execution by running the CRTM code[3] [5], whose main task is to begin the integrity measurement process, which consists of computing a cryptographic hash of every hardware and software component (both code and data), in order to get a "unique" fingerprint of the system. These measurements are then stored into a trusted environment provided by the TPM, through the use of a well defined API [7, 6].

The TPM is both the root of trust for *storing* and for *reporting*. The former goal is obtained by storing the measurement values into several tamper-resistant areas, called *shielded locations*. These are protected memory regions which can be accessed only by using special *protected capabilities*, which require the use of some form of authorization, defined more extensively in Section 4. The most important shielded locations are the *Platform Configuration Registers*, PCR, and *Data Integrity Registers*, DIR[4], which are also used to effectively store integrity measurements and to provide a *secure* and *authenticated* boot sequence [9]. Reporting, instead, is performed by the TPM by communicating the computed integrity measurements to a challenger, through a challenge/response protocol, to permit a trusted verification of the TP software integrity, as defined in [9, 19].

4 Authorization Protocols

A prominent role inside a TP is played by the authorization protocols. These protocols are used anytime a subject has to issue some command (for a list of such commands, see [7]) for accessing protected TP resources. The authorization protocols main scope is to provide a secure execution of the command by guaranteeing that it is executed by a subject who is entitled to do so, and in compliance with any confidentiality and integrity policy regarding the involved resource.

The two main authorization protocols are the

[3]Usually the whole BIOS or the BIOS Boot Block for compound BIOSes.

[4]This is true in the TCG specification version 1.1, since the version 1.2 generalized the concept by introducing Non-Volatile memory area and related capabilities.

Object-Independent Authorization Protocol (OIAP) and the *Object-Specific Authorization protocol (OSAP)*. The former enables a user to open an authorization session which can be used to issue the same authorized command several times during the same session, potentially acting on different protected TPM resources. The latter works in a similar way, but allows to issue different authorized commands acting on the same TPM protected resource, using the same authorization session. In this paper we concentrate our attention on the OIAP.

4.1 OIAP

The OIAP is used whenever a subject needs to send the same command GC to many different protected resources, during the same authorization session. The protocol works as follows and it is more formally depicted in Figure 1. Let **U** be a generic subject that wish to use a resource R protected by the TPM root of trust **T**, and A_{R} a secret shared between **U** and **T**.

Initially **U** request to open an authorization session with **T** (step 1, Figure 1) and **T** send back to **U** the session information needed to correctly handle the authorization session itself (step 2, Figure 1). If such a step is correctly performed, **U** will send to **T** the command GC to execute, which embeds the prove that she knows A_{R} (step 3, Figure 1). Afterwards, **T** will verify the message authenticity and integrity and, if they are satisfied, **T** will execute GC on behalf of **U**, sending back to **U** the result obtained (step 4, Figure 1). Otherwise the connection will be closed by **T**. Figure 1 depicts the OIAP above described, where $resAuth = \{R_{\mathrm{x}}.GC.S_1.N'_{\mathrm{e}}, N_{\mathrm{o}}\}$, R_{x} is either R_{c} or R_{e} accordingly to the substep 4 chosen, and let *CMD_OIAP* be the command issued by **U** for initiating an authorization session; this command does not necessitate to be authorized, i.e., every entity can issue it, let GC be the authorized command which **U** wants to execute on R, let R_{c} be a success return code, let R_{e} be an invalid authorization code, let D be the data related to GC (it may be empty), let RES be the result of the execution of GC on R (it may be empty), let N_{e} be a 160-bit non-predictable even random number used to provide the *freshness* property, let N_{o} be a 160-bit non-predictable odd random number, and finally let $Auth = \{GC.S_1.N_{\mathrm{e}}.N_{\mathrm{o}}\}$ be the concatenation of data on which the key-hashed cryptographic function has to be computed.

1. **U** → **T**: *CMD_OIAP*
2. **T** → **U**: S_1, N_{e}
3. **U** → **T**: $GC.R.D.S_1.N_{\mathrm{e}}.N_{\mathrm{o}}.\mathrm{H}(A_{\mathrm{R}}, Auth)$

if the verification of $\mathrm{H}(A_{\mathrm{R}}, Auth)$ is successful, then **T** execute GC on R and

4a. **T** → **U**: $R_{\mathrm{c}}.GC.RES.N'_{\mathrm{e}}.N_{\mathrm{o}}.$
$\mathrm{H}(A_{\mathrm{R}}, resAuth)$

else

4b. **T** → **U**: $R_{\mathrm{e}}.GC.RES.N'_{\mathrm{e}}.N_{\mathrm{o}}.$
$\mathrm{H}(A_{\mathrm{R}}, resAuth)$

Figure 1: Description of the OIAP

4.2 OIAP Threats

As any communication protocol, OIAP is subjected to replay, Man-in-The-Middle (MiTM) and Denial of Service (DoS) attacks. By the TP specification it turns out that, in the design of the TPM, only the first two types of attacks have been addressed while DoS has been voluntarily neglected [9]. Indeed, the specification does not avoid someone to be in the middle of a communication[5], but it tries to provide a protection against a Dolev-Yao MiTM, which acts like "an active saboteur, [one] who may impersonate another user and may alter or replay the message" [11]. In order to deal with such attacks, namely replay and packet mangling, the OIAP adopts, respectively, the rolling nonces paradigm and HMAC.

Nonces are pseudo-random unique non-predictable numbers which are used just once[6]. Rolling nonces

[5]Note that OIAP has been designed in order to also work in a network environment [7].

[6]Nonce can be interpreted as "Number [to be used] once.

are exchanged back and forth between the involved parties, in order to permit them to check the *freshness* of messages [17]. In fact, **T** can verify (see Figure 1, step 3), that the received nonce is equal to N_e, as chosen and sent in step 2 by **U**. More generally, any involved party is able to verify the freshness property for each exchanged message and, as long as the parties verify this property, no replay attacks would succeed.

On the other side, TCG-based TPs are able to detect packets alteration by deploying HMAC, but they are not able to distinguish between common network errors and real packet mangling performed by a MiTM. This will play a fundamental role in the OIAP attack we devised as explained in Section 5.

5 The Attack

Our attack leverages on two OIAP features:

1. the authorization session created by a genuine *CMD_OIAP* command is kept opened indefinitely by **T**, unless:
   - the TPM chooses to close it explicitly;
   - an erroneous[7] message is received by the TPM.
2. the authorization session is closed by the user's application if an erroneous message is received.

Feature two is not explicitly stated in the TCG specification, however, it is a common practice, for a client application, to close a connection if something goes wrong (network errors, malformed packets, and so on) [3].

Roughly speaking, when several command sessions are involved an attacker may be able to subvert the TP trusting mechanism by performing what P. Syverson in his taxonomy ([20]) calls *straight replays*. Straight replay attacks occur when *"a message is sent straight from the sender to the intended recipient in different protocol rounds or in different protocol runs though it may be just delayed or have other text appended to it for altering generally the significance of the message"* [13].

[7]i.e., a message with wrong parameters or an invalid HMAC.

Precisely, the attack we propose can be divided into three phases, namely *message storing phase* (Figure 2), *message re-sending phase* (Figure 3) and *replay attack phase* (Figure 4), that have to be completed in order to perform the whole *replay attack* successfully (Figure 5).

During the explanation of the three above mentioned phases, we use the notation $\mathbf{X}^*$ for denoting an intruder which impersonates the entity **X**. We also assume that $\mathbf{X}^*$ follows the Dolev-Yao model [11] so that it is able to intercept all the messages, store, drop or forward them. Note that, as already said, packets mangling is permitted but is detected by the involved parties which cannot, however, distinguish between either network errors or MiTM presence.

For the sake of simplicity, it's worth noting that what follows it is based on an intruder which plays only on a single protocol run at any given time (from his perspective), so he is enforced to execute the attack's phases in a sequential order. It is also worth noting that the proposed replay attack can be more complex and somewhat tricky to understand, when the intruder wants to monitor different protocol runs (between the involved parties) in parallel since the attack's phases may be not sequentially executed.

5.1 Attack Schema

The main message storing phase goals may be summarized in the following list:

- to permit the intruder to capture command sent by the user **U**, in order to be able to replay it later;
- to keep the authorization session on the **T** side opened.

During this phase, in fact, the attacker retrieves successfully a message originating from the user (step 3a, Figure 2), and stores it in order to be able to inject the message into another run of the protocol. In the meantime the authorized session remains opened,

Message storing phase

1a. $\mathbf{U} \rightarrow \mathbf{T}^*$: $\mathit{CMD\_OIAP}$
1b. $\mathbf{U}^* \rightarrow \mathbf{T}$: $\mathit{CMD\_OIAP}$
2a. $\mathbf{T} \rightarrow \mathbf{U}^*$: $S_1.N_\mathrm{e}$
2b. $\mathbf{T}^* \rightarrow \mathbf{U}$: $S_1.N_\mathrm{e}$
3a. $\mathbf{U} \rightarrow \mathbf{T}^*$: $GC.R.D.S_1.N_\mathrm{e}.N_\mathrm{o}.$ $\mathrm{H}(A_\mathrm{R}, \mathit{Auth})$
3b. $\mathbf{T}^* \rightarrow \mathbf{U}$: *reset*

Figure 2: The OIAP message storing attack phase.

Message re-sending phase

4a. $\mathbf{U} \rightarrow \mathbf{T}^*$: $\mathit{CMD\_OIAP}$
4b. $\mathbf{U}^* \rightarrow \mathbf{T}$: $\mathit{CMD\_OIAP}$
5a. $\mathbf{T} \rightarrow \mathbf{U}^*$: $S_2.N'_\mathrm{e}$
5b. $\mathbf{T}^* \rightarrow \mathbf{U}$: $S_2.N'_\mathrm{e}$
6a. $\mathbf{U} \rightarrow \mathbf{T}^*$: $GC.R.D'.S_2.N'_\mathrm{e}.N'_\mathrm{o}.$ $\mathrm{H}(A_\mathrm{R}, \mathit{Auth})$
6b. $\mathbf{U}^* \rightarrow \mathbf{T}$: $GC.R.D'.S_2.N'_\mathrm{e}.N'_\mathrm{o}.$ $\mathrm{H}(A_\mathrm{R}, \mathit{Auth})$
7a. $\mathbf{T} \rightarrow \mathbf{U}^*$: $R_\mathrm{c}.GC.RES'.N''_\mathrm{e}.N'_\mathrm{o}.$ $\mathrm{H}(A_\mathrm{R}, \mathit{resAuth})$
7b. $\mathbf{T}^* \rightarrow \mathbf{U}$: $R_\mathrm{c}.GC.RES'.N''_\mathrm{e}.N'_\mathrm{o}.$ $\mathrm{H}(A_\mathrm{R}, \mathit{resAuth})$

Figure 3: The OIAP message re-sending attack phase.

Replay attack phase

8a. $\mathbf{U}^* \rightarrow \mathbf{T}$: $GC.R.D.S_1.N_\mathrm{e}.N_\mathrm{o}.$ $\mathrm{H}(A_\mathrm{R}, \mathit{Auth})$
8b. $\mathbf{T} \rightarrow \mathbf{U}^*$: $R_\mathrm{c}.GC.RES.N'_\mathrm{e}.N_\mathrm{o}.$ $\mathrm{H}(A_\mathrm{R}, \mathit{resAuth})$

Figure 4: The OIAP replay attack phase.

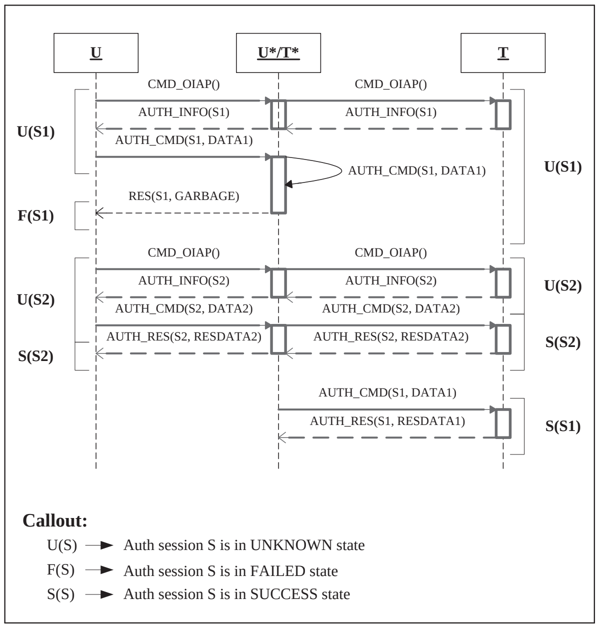

Figure 5: Overall OIAP replay attack.

while the user may be fooled by a *reset*[8] message as seen in step 3b, Figure 2.

At this point, the message re-sending phase may take place. This phase plays a fundamental role and has to be successfully completed in order to permit the intruder to perform a meaningful[9] replay attack. Its main objective is to let the intruder wait for an user action and, therefore, take the appropriate decision about what to do next (see the next phase). The possible user's actions are:

(a) open a new authorization session and re-send the faulty command acting on different data;

[8]This message resembles a "legal" reply message, such as the one sent at step 4a, shown in Figure 1, but with some erroneous bits in it, giving the "illusion" of a temporary network error.

[9]Meaningful here means that the intruder has the ability to replay every message captured by the message storing phase (obviously), but replaying a just stored message makes no sense because that would exactly execute what the user **U** wanted to.

(b) re-send the faulty command on the same data;

(c) execute another kind of authorized command.

Note that in this phase the intruder simply works as a forwarder between the user and the TP. Moreover, we will consider only the case (a), as only in this case the attack we envisaged can produce significant consequences during a single protocol run (intruder's view-point).

In fact, in the last replay attack phase, the intruder may replay the captured message (we recall that the "first" authorization session is still pending, due to feature 1, Section 5) and overwrite a TPM protected data resource, without the user knowledge, compromising the TP integrity and its trust property. For the sake of clarity and to better understand the relationship between the attack phases, a simple NFA which model the intruder's behavior during a single protocol run, is depicted in Figure 6, where *q0* represents the automaton initial state and edges' labels refer to the actions the user may perform.

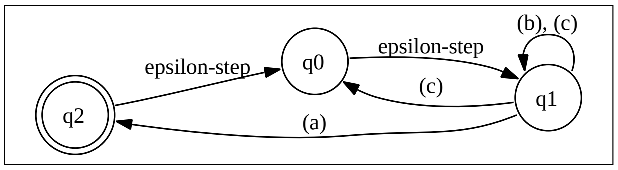

Figure 6: NFA modeling intruder's behavior.

6 Attack Scenario

In order to better understand the consequences of the replay attack described above, we propose an attack scenario along with a "rough" measurement model (a simple *Proof of Concept*), which show how it is possible to subvert the integrity property of the TP *post-boot* environment[10] components, such as software packages, eluding in such a way the sense of trust provided by the TP.

It's worth recalling (see Section 3) that one of the TP main functionality is to provide *software measurements*, that is, a "complete" integrity snapshot of software and hardware components of the TP itself. So, the main goal of this process is to perform such measurements and store the result into TPM shielded locations, in order to be able to detect any unauthorized modification of the measured components.

[10]It is the environment in which the TP switch after the OS kernel is loaded and executed.

The TCG specification only defines measurement model for the *pre-boot* environment[11] [5] while it is not stated anything about the post-boot environment.

We think that it may be possible to classify post-boot measurement models that can be adopted on TCG-based TPs as:

- *detection measurement models*, where the TP provides the mechanisms that will enable an entity to check for the TP components integrity. It's worth nothing that, given a certain environment measurement, it's up to this entity to trust the TP or not;

- *prevention measurement models*, where the TP provides the mechanisms that are able to detect and potentially stop any unauthorized modification of the TP components.

While it has been done some research about detection measurement models, such as the one proposed by [18, 15], to our knowledge, nothing has been proposed as prevention measurement model for TCG-based TPs post-boot environment yet.

6.1 Attack Environment

In the attack scenario described here below, we concentrate our attention on the deployment of a simple and bare prevention measurement model we are working on (a full and precise description of this model is out of the scope of this paper), we give a description of the players involved in the attack along with their roles and, finally, we conclude the attack scenario describing the attack itself.

[11]This is the environment in which the platform starts when it's turned on. The TP remains in this environment until the OS kernel is loaded and executed.

Prevention Measurement Model

This simple measurement model base its functionality on the following data structures:

- an *untrusted* measurement list, ML, which is made of ordered *(program, fingerprint)* pairs;

- a Non-Volatile (NV) Storage Area, namely DIR[12], provided and protected by the TPM in a way that only protected capabilities can modify its value.

  We use the DIR register to hold the hash of the whole ML in order to be able to detect unauthorized ML modification, as explained below (it's not worth speaking of the way this hash is computed here).

The measurement model is comprised of two different phases, namely a *setup* and a *running* phase. The setup phase objectives are to build up the ML of the interested software and to store the ML hash into the DIR by means of TPM protected capabilities (i.e. TPM authorized commands), while the running phase goal is either to grant or not the execution of a particular software belonging to the measurement list ML computed in the setup phase. More formally, these phases can be described here below.

Setup phase

Let SW be the set of the software that has to be measured in order to provide for its integrity and let ML be the measurement list initially empty, then:

$$\forall p \in \mathrm{SW} \qquad \mathrm{ML} = \mathrm{ML} \cup \{(p, F_p)\}$$

where F_p represents the fingerprint bound to the program p. Now it is possible to compute the hash over the measurement list ML ($H_{DIR} = \mathrm{H(ML)}$), and store this value into the DIR NV storage area by means of the `TPM_NV_WriteValueAuth` authorized command, which use OIAP as authorization protocol. After this initial phase, the running phase takes place whenever the TP is running[13].

It's worth pointing out that this NV (DIR) usage differs from the PCR usage. In fact, the former makes use of TP authorized commands, such as the aforementioned `TPM_NV_WriteValueAuth`, that writes a value *directly* into the pointed NV area, thus *overwriting* everything stored there [9, 7], while the latter case makes use of other *non-authorized* TP commands that trigger the TPM to perform a chained hash internally, taking care of what is stored in the referenced PCR [9, 7].

As already noted, a NV shielded location must be used in order to keep integrity values through TP reboot. Such a feature, in fact, would not be possible to achieve using PCRs shielded location.

Running phase

Upon setup phase accomplishment, whenever a TP user/process wants to execute a "protected" software, A, which belongs to the ML, the loader performs the following steps, granting or not A execution:

1. $h_A = \mathrm{H(A)}$

2. $\mathrm{ML}' = \mathrm{ML}$

3. $\mathrm{ML}' = \mathrm{ML}' \setminus \{(\mathrm{A}, f_A)\}$

4. $\mathrm{ML}' = \mathrm{ML}' \cup \{(\mathrm{A}, h_A)\}$

5. $H'_{DIR} = \mathrm{H}(\mathrm{ML}')$

6. if ($H'_{DIR} == H_{DIR}$) then

   "grant A execution"

   else

   "deny A execution"

[12]The TCG specification version 1.2 speak of protected general purpose NV Storage Area, which replace the limited DIR register defined in the TCG specification version 1.1.

[13]Note that is possible to perform again the setup phase whenever a software update is needed or whenever a "new" software has to be added to the measurement list.

Players & Roles

The attack scenario here proposed encompasses the presence of three main players that are described in the following list along with a brief description of their roles.

- *TP administrator* plays a central role in a TCG-based TP. She is the only person able to issue authorized commands in order to exploit the TP features, such as identity, measurement and protected storage.
- *Host administrator* is in charge of administering the TP performing actions which are common to normal system administrators, such as software updates, users and resources managements, and so on;
- the *attacker* is the active saboteur who wants to subvert the *trust* property introduced by the use of the Trusted Platform.

It's worth nothing that the role played by TP administrator and Host administrator, is different – although it may be related – and so, it may be covered by different people.

Attack Description

The possible attack description which follows is based on the deployment of the *prevention measurement* model aforementioned.

Let **T** be a remote TP, which performs the *secure boot* procedure [9], aimed at proving that the whole TP firmware and software components, have not been compromised. During the boot process, the CRTM perform integrity measures of all the TP components and such values are compared with the ones recorded into the TPM DIR. If everything matches, OS kernel is loaded into memory and executed.

Let **U** be a remote TP administrator who needs to upgrade some TP software components. **U** has to perform the following actions:

1. download a new software from an official site;
2. verify some sort of digital signature, if possible, by trusting the new package if the signature matches;
3. update the remote DIR integrity value, via the OIAP, by issuing the `TPM_NV_WriteValueAuth` authorized command [7].

Unfortunately, before the OIAP takes place, the attacker **A**, who acts as $\mathbf{U}^*$ and $\mathbf{T}^*$, starts the message storing phase, capturing the `TPM_NV_WriteValueAuth` authorized message which is used by the administrator to overwrite the remote DIR value. We also assume that **A** has *Host administrator* privileges on the TP either because he is an insider or he has gained unauthorized access.

On the other hand, the *TP administrator* will perform the message re-sending phase, by issuing the same command acting on the same parameters (point (b) of Section 5) since she may think that the received faulty message was due to some communication error. As a result, she may now authorize the installation of a new "trusted" software package.

Sooner or later the *Host administrator* become aware about the presence of a bug in the just installed and measured software package, so she promptly downloads a new patched version of it, verifies its signature and let the *TP administrator* to perform again the OIAP in order to update the "trust" state of the remote TP, so that a new patched version can be installed in the system.

At this point, **A** can perform the replay attack phase, by replaying the message captured in the message storing phase. By doing so, he is able to overwrite the new "trusted" DIR integrity value with the old one referring to an incorrect measure of the installed software, fundamentally flawing the integrity property and the sense of trust the user places in the TP mechanisms. In fact, **A** re-install the bugged software, replay the old measured value which has the effect to be stored into the DIR, overwriting any register content.

7 A proposed solution

Even if the rolling nonces paradigm is a good candidate to protect against replay attacks, as already noted in [16, 14], it may be a weak countermeasure under some circumstances.

In our case the main problem is due to the fact that a coherent and synchronized “session knowledge”, between the involved parties, is missing and, at the end of our replay attack, **T** and **U**, have a different knowledge about the session state. Thus, in order to avoid the replay attack just described, we have to provide the protocol with the mechanisms which enable the parties to share a common knowledge on the sessions state.

The solution we devised requires a modification to the TPM and it is based on the introduction of a new field computed by the user, which is added to any authorized exchanged message between the parties. Such a field contains the knowledge which any user has about the state of all previously opened authorization sessions. Moreover, in transmission such a field is protected from tampering attempts by HMAC.

This session-state field is a *bitmask* which is filled in the following way:

- set the *i-th* bit to 0 if the *i-th* authorization session is considered either *open* or in an *unknown* state;

- set the *i-th* bit to 1 if the *i-th* authorization session is considered in a *failed* state, i.e., a *reset* message is received by the user as a response to an issue of an authorized command (see step 3b, Figure 2).

Every time that the TPM finds an incoherent state between user sessions and the TPM ones (e.g. user’s S_1 *failed* and TPM S_1 *unknown*), it closes the “wrong” session. Doing so, every pending sessions held by an attacker cannot be used anymore, denying any further attempt to execute a previously stored authorized message.

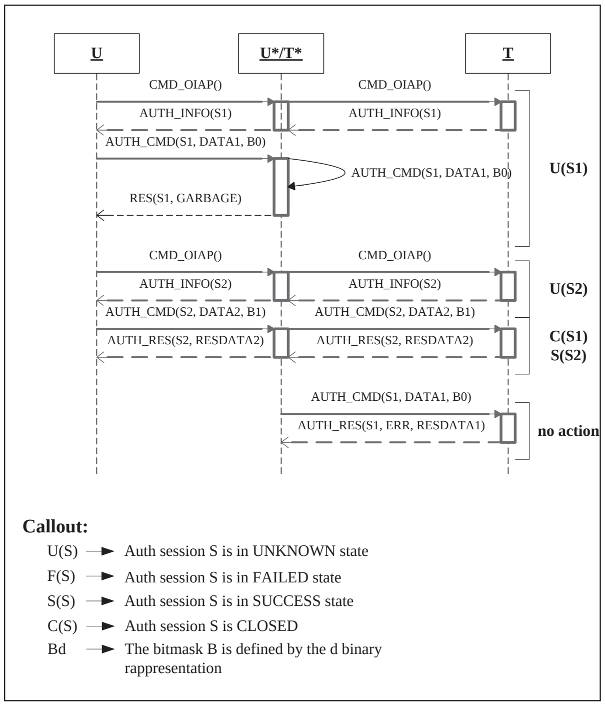

Figure 7: Overall OIAP replay attack’s solution.

8 Conclusions

In this paper, we analyzed one of the core components of the Trusted Computing Platform proposed by the Trusted Computing Group. In particular, we focused our attention on the Object-Independent Authorization Protocol, which is involved whenever a TPM protected resource has to be used.

Although the TCG specification tried to protect this sensible protocol from both *replay* and *MiTM* attacks (more precisely *packet mangling actions*), our analysis showed that that the protocol is flawed by design and a replay attack is indeed possible.

We proposed a solution to solve the problem, based on the idea of recording and sharing the session state between the communicating parties. It is our opinion that the proposed solution can be improved in order to allow also for further misuse detection allowing a party to detect MiTM presence. Work is in progress for investigating such issues.

References

[1] *Announcing adore-ng 0.31.* http://www.securityfocus.com/archive/1/348843/2003-12-30/2004-01-05/0.

[2] *HMAC: Keyed-Hashing for Message Authentication.* ftp://ftp.rfc-editor.org/in-notes/rfc2104.txt.

[3] *IBM Watson Research Center, Global Security Analysis Lab: TCPA Resources.* http://www.research.ibm.com/gsal/tcpa/TPM-2.0.tar.gz.

[4] *Trusted Computing Group.* http://www.trustedcomputinggroup.org.

[5] *TCG PC Specific Implementation Specification.* http://www.trustedcomputinggroup.org, August 2003.

[6] *TCG Software Stack (TSS) Specification.* http://www.trustedcomputinggroup.org, August 2003.

[7] *Trusted Platform Module Main Specification, Part 1: Design Principles, Part 2: TPM Structures, Part 3: TPM Commands.* http://www.trustedcomputinggroup.org, October 2003.

[8] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 65–71. IEEE Computer Society, 1997.

[9] B. Balacheff, L. Chen, S. Pearson, D. Plaquin, and G. Proudler. *Trusted Computing Platforms: tcpa technology in context.* Prentice Hall PTR, 2003.

[10] D. Bruschi, D. Fabris, V. Glave, and E. Rosti. How to unwittingly sign non-repudiable documents with Java applications. In *ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference*, page 192. IEEE Computer Society, 2003.

[11] D. Dolev and A. C. Yao. On the security of public-key protocols. In *IEEE Transactions on Information Theory*, 1983.

[12] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A Trusted Open Platform. *Computer*, 36(7):55–62, 2003.

[13] T. Kwon and J. Song. Clarifying straight replays and forced delays. *SIGOPS Oper. Syst. Rev.*, 33(1):47–52, 1999.

[14] G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In *TACAs '96: Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 147–166. Springer-Verlag, 1996.

[15] J. Marchesini, S. Smith, O. Wild, J. Stabiner, and A. Barsamian. Open-source applications of tcpa hardware. In *In 20th Annual Computer Security Applications Conference. IEEE Computer Society*, 2004.

[16] C. Meadows. Analyzing the Needham-Schroeder Public-Key Protocol: A Comparison of Two Approaches. In *ESORICS '96: Proceedings of the 4th European Symposium on Research in Computer Security*, pages 351–364. Springer-Verlag, 1996.

[17] Peter Ryan and Steve Schneider and Michael Goldsmith and Gavin Lowe and Bill Roscoe. *Modelling & Analysis of Security Protocols.* Addison-Wesley, 2000.

[18] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *13th USENIX Security Symposium*, 2004.

[19] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In

Proceedings of the 13th USENIX Security Symposium, pages 223–238, 2004.

[20] P. Syverson. A taxonomy of replay attacks. In *Proceedings of the 7th IEEE Computer Security Foundations Workshop*, 1994.

[21] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 193–206. ACM Press, 2003.