Replay Attack in TCG Specification and Solution

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

Abstract

We prove the existence of a flaw which we individuated in the design of the Object-Independent Authorization Protocol (OIAP), which represents one of the building blocks of the Trusted Platform Module (TPM), the core of the Trusted Computing Platforms (TPs) as devised by the Trusted Computing Group (TCG) standards. In particular, we prove, also with the support of a model checker, that the protocol is exposed to replay attacks, which could be used for compromising the correct behavior of a TP. We also propose a countermeasure to undertake in order to avoid such an attack as well as any replay attacks to the aforementioned protocol.

1. Introduction

One of the most recent trends, undertaken by the Computer Security community for the construction of more secure Information and Communication Technology systems, is based on the use of Trusted Computing Platforms (TPs). Roughly speaking, TPs are computing platforms which contain built-in trusted components whose functionalities can be used for building secure services inside the platform such as secure boot, digital signatures, software integrity checking, and so on.

Initially inspired by the seminal work of Arbaugh et al. [6], TPs have been extensively studied in these last years and various computational model based on the TP concept appeared in literature (see for example [10, 20, 2]). In this paper we will concentrate our attention on the model proposed by the Trusted Computing Group (TCG) [2], a multi-vendor consortium formerly known as the Trusted Computing Platform Alliance (TCPA). The TCG has proposed a specification 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 the framework is known as the Trusted Platform Module (TPM), whose main task is to provide facilities that can be used to verify the integrity of the system as well as to grant the access to protected resources only to trusted components. For performing such a task the TPM executes a set of well defined protocols which have been designed for resisting to various types of attacks.

In this paper we are interested in the *Object-Independent Authorization Protocol (OIAP)*, used by the TPM for performing authorization tasks and specifically designed for resisting to replay attacks. In particular, we will show that during the execution of such a protocol, a replay attack is indeed possible, and there exists the opportunity for an intruder to capture a message M exchanged between the TPM and some authorized user, and to use M in a future session for compromising the correct behavior of the trusted platform. We will also be able to provide a formal proof of the existence of such a bug via a model checker. The countermeasures for solving the problem have been investigated and a possible solution will be described as well.

The rest of 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 in Section 6 we describe our use of a model checker to prove the existence of the design flaw. In Section 7 we propose a modification of the protocol 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.

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;
- **A** → **B**: M denotes that A sends the message M to B;
- S_i denotes the *i-th* authorization session;
- $A', A'', \cdots$ indicates further "instances" of a pseudo-random number A, e.g., 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, and so on.

3. Trusted Computing Platforms

Trusted Platforms (TPs) are computing platforms that include a set of built-in hardware components which are used as a basis for creating trust in software processes. Such components are the *Core Root of Trust for Measurement* (CRTM) and the *Trusted Platform Module* (TPM). They are hardwired on the TP motherboard and have to be "trusted"[1] and protected from tampering in order to consider the whole computing platform a TP.

Roughly speaking, TPs provide three additional functionalities with respect to standard computing platforms:

1. *identity*: a TP can be identified in a unique and secure way, i.e., thus avoiding impersonation;
2. *measurement*: a TP can compute a "complete" and reliable integrity check of its software and hardware components;
3. *protected storage*: the TP may provide protection to sensible data (passwords, cryptographic keys, passphrases, data and so on).

A TP starts its execution by running the CRTM code[2] [3]. Such code performs an integrity check of every hardware and software components (both code and data). These measurements are then stored into a protected storage provided by the TPM, through the use of a well defined API [5, 4]. In fact, such measurement values are stored 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* (PCRs) and *Data Integrity Registers* [3] (DIRs), which are also used to effectively store integrity measurements and to provide a *secure* and *authenticated* boot sequence facilities [7]. Moreover, in order to enable a trusted verification of the software environment, the TPM communicates the computed integrity measurements to a challenger by exploiting a challenge/response protocol, as defined in [7, 18].

4. Authorization protocols

TPs use authorization protocols every time a subject issues a command (for a list of such commands see [5]) which requires the access to some protected resource. The main scope of the authorization protocols is to release access capabilities to authorized subjects in compliance with the confidentiality and integrity policies regarding the involved resource.

The TCG specification describes two main authorization protocols: the *Object-Independent Authorization Protocol* (OIAP) and the *Object-Specific Authorization Protocol* (OSAP). OIAP opens an authorization session during which the same command can be issued several times, potentially acting on different protected resources. OSAP works in a similar way, but, in a single authorization session, every command has to refer to the same protected resource. In this paper we focus our attention on the OIAP.

4.1. OIAP

The OIAP works as follows. Let **U** be a generic subject that wishes to use a resource R protected by the TPM **T**, and let A_R be a secret shared between **U** and **T**. Moreover, let

- GC be the authorized command which **U** wants to execute on R, that is, GC is a command which requires authorization in order to be used. An authorization protocol is used to provide such a requirement.
- R_c be a success return code,
- R_e be an invalid authorization code,
- D be the data related to GC (it may be empty),
- RES be the result of the execution of GC on R (it may be empty),
- N_e be a 160-bit non-predictable even random number used to provide the *freshness* property (more on this, later),

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

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

[3] This holds in the TCG specification version 1.1. Version 1.2 of the specification generalizes the concept by introducing Non-Volatile memory area and related capabilities.

- N_o be a 160-bit non-predictable odd random number having the same N_e properties, and,
- $Auth = \{GC.S_1.N_\mathrm{e}.N_\mathrm{o}\}$ be the concatenation of data on which a key-hashed cryptographic function has to be computed in order to detect for corrupted data transmission.

Initially, **U** requests to open an authorization session by sending the command *CMD_OIAP* to **T** (step 1, Figure 1), and **T** sends back to **U** the session information needed to 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 (correct) or R_e (error) accordingly to the sub-step 4 chosen.

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. TCG specification only addresses the first two types of attacks, while explicitly does not take into account DoS [7]. Indeed, the specification does not avoid someone to be in the middle of a communication (note that OIAP has been designed in order to also work in a network environment [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" [9].

In order to deal with replay and packet mangling attacks, the OIAP adopts, respectively, a rolling nonces paradigm and Hashed Message Authentication Code (HMAC).

Nonces are unique non-predictable pseudo-random numbers which are used just once[4]. *Rolling* nonces are exchanged back and forth between the involved parties, in order to permit them to check for the *freshness* of messages (freshness property) [17]. In fact, referring to the OIAP protocol previously described, **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, they are able to detect if a packet coherently belongs to the ongoing session.

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

In this section we will outline the strategy adopted for attacking the OIAP, while a more formal description of the attack will be presented in Section 6. The attack exploits a particular feature of the OIAP, as defined in the TCG specification, which requires that the authorization session created by a genuine `TPM_OIAP` command is kept opened indefinitely by a TPM, unless either it chooses to close the session explicitly or an erroneous message (i.e., a message with wrong parameters or an invalid HMAC) is received on that authorization session.

Given such a specification an attacker may be able to perform a *straight replay attack* ([19, 14]) in the following way. After the user and the TPM have exchanged the first two messages of the protocol (Steps 1, 2 of Figure 2) the attacker intercepts the next message originating from a user (Step 3 of Figure 2), and stores it in order to be able to inject the message into another run of the protocol. Meantime, the attacker fools the user by sending her a *reset* message (remember that the attacker lies in between the communication channel between the user and the TP). 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. Moreover, we are assuming that, as it is common practice, the client application, closes a connection when something goes wrong [1], though this is not explicitly stated in the specification.

At the end of this phase there exists an open authorized session towards the TPM, and the user is not aware of it. At

[4] Nonces can be thought as a "*numbers* [to be used] *once*".

this point, the intruder will wait for the next user action and, subsequently, he will take the appropriate decision.

Having seen that his first connection attempt failed the user could:

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

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

(c). execute another kind of authorized command.

Obviously from the intruder point of view only the case (a) can produce significant consequences in order to perpetrate the attack in a meaningful way, considering the fact that the attacker has already been able to start the attack. In fact, opting for case (c) would bring us to another run of the protocol where the attack might be performed again from the beginning, while opting for case (b) would not produce significant effects beside a mere DoS attack due to request, reset messages sent back and forth respectively by the user and the attacker. Instead, if the user opted for case (a) the last phases of the attack might take place. In fact, using the aforementioned still opened authorization session and the intercepted message, the intruder will be able to overwrite a TPM protected resource, hopelessly compromising the correct behavior of the platform.

As a consequence of such an attack, the "active saboteur" is able to replay any (captured) authorized commands at will, potentially altering the sense of trust users place in the TP. For example, the TCG specification provides a shielded location, known as Non-Volatile (NV) storage area, that is protected by the TPM in a way that only protected capabilities can modify its value. If such an area is used to hold important integrity measurement values, playing the attack herein proposed, would permit an attacker to overwrite the stored "up-to-date" value, with an old "out-of-date" one, previously captured. Doing so, the attacker would be able to replace up-to-date measured software packages or data, with older ones that represent a security risk for the system, avoiding TP detection.

For ease of exposition, we denoted by $\mathbf{X}^*$ an intruder who impersonates the entity $\mathbf{X}$ and we divided the attack into three phases, namely the *message storing phase* (Figure 2) aimed at intercepting the message, the *message re-sending phase* (Figure 3) aimed at observing the user behavior and the *replay attack phase* (Figure 4) in which the attack is finally perpetrated.

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. $\mathbf{U} \rightarrow \mathbf{T}^*$:	$GC.R.D.S_1.N_\mathrm{e}.N_\mathrm{o}.$ $\mathrm{H}(A_\mathrm{R}, Auth)$
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. $\mathbf{U} \rightarrow \mathbf{T}^*$:	$GC.R.D'.S_2.N'_\mathrm{e}.N'_\mathrm{o}.$ $\mathrm{H}(A_\mathrm{R}, Auth)$
6b. $\mathbf{U}^* \rightarrow \mathbf{T}$:	$GC.R.D'.S_2.N'_\mathrm{e}.N'_\mathrm{o}.$ $\mathrm{H}(A_\mathrm{R}, Auth)$
7a. $\mathbf{T} \rightarrow \mathbf{U}^*$:	$R_\mathrm{c}.GC.RES'.N''_\mathrm{e}.N'_\mathrm{o}.$ $\mathrm{H}(A_\mathrm{R}, resAuth)$
7b. $\mathbf{T}^* \rightarrow \mathbf{U}$:	$R_\mathrm{c}.GC.RES'.N''_\mathrm{e}.N'_\mathrm{o}.$ $\mathrm{H}(A_\mathrm{R}, resAuth)$

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

Replay attack phase

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

Figure 4. The OIAP replay attack phase.

6. Model checking the protocol

We analyzed formally the OIAP in order to mathematically prove the existence of the design flaw described in the previous section. Our analysis was performed using SPIN, a model checker written by Gerald J. Holzmann [13] based on Büchi automata[5]. Given a system M (appropriately modeled) and a property P (often expressed as a proposition in a Linear Temporal Logic), the model checker is able to verify whether P is valid in any sequence of states which M may traverse. More precisely, SPIN represents M and the *negation* of P as a Büchi automaton, respectively automaton$_M$ and automaton$_{\neg P}$, and computes the intersection I of the languages accepted by the automaton$_M$ and the automaton$_{\neg P}$. If it is empty, then the property P is verified for every possible sequence of states of M. Conversely, every phrase of the non-empty language I is a stat of M in which P is not verified. In our context, M is a system executing the OIAP and P is a proposition stating that the protocol execution is not flawed.

In order to model a system, SPIN provides its own description language called PROMELA. PROMELA allows for expressing state-based computations non-deterministically. The computation flow is regulated by guarded commands à la Dijkstra and communication among different sequential processes is expressed by Hoare's CSP primitives [11]. Properties to be checked can be expressed as formulae in a linear time temporal logic (LTL), as system or process invariants, as formal Büchi automata, or as "never" claims.

From its beginning SPIN was used to find flaws in network protocols [12], however its use in a security context is quite new. PROMELA does not provide any security primitives, such as encryption or hashing and even if model checkers with such capabilities exist (see for example Brutus [8]), we decided to build a model of OIAP in which the use of cryptographic hash functions were abstracted away.

A challenging problem we faced was that, since we wanted to check OIAP strength against replay attacks, we had to model not only the protocol itself, but also the attacker. To this end, we decided to model the attacker as a process able to produce every valid OIAP related command. Moreover, we assumed the attacker was unable to forge hashes, since we think HMAC usage is correctly deployed by the TCG specification as protection against packets mangling attacks[6]. In the following we sketch the model we used and the results we found.

6.1. Modeling the property that should be preserved

PROMELA models consist of processes (which we call *actors*), message channels, and variables. Actors are global objects that represent the concurrent entities of the system under analysis. Messages are modeled without considering HMACs, since we were interested just in analyzing counter-replay attack strength. As actors we considered a *Caller*, a *MiTM* and a *TPM*. Following the Dolev-Yao [9] model, in order to represent an intruder (MiTM) who lies in the middle of the communication channel between the Caller and the TPM, we defined two half-duplex unbuffered "physical" channels, namely `caller_mitm_wire` and `mitm_tpm_wire`, that represent a unique logic channel used for Caller/TPM communication, as shown in Figure 5.

Subsequently using PROMELA we modeled the Caller, MiTM and TPM using the following criteria (the code we implemented is reported in the Appendix A).

The Caller strictly follows the OIAP rules. Precisely, the caller can:

1. open an authorization session with the TPM by which an authorized command is executed;

2. send an authorized request command to the TPM in order to use a TPM protected object; an authorization session must be previously opened, i.e., a `TPM_OIAP` must be issued before sending an authorized command;

3. receive the authorized command reply from the TPM.

The TPM is also acting strictly according to the TCG specification, so it can:

1. open authorization session as requested by a caller, by setting up all the relevant protocol's information, such as nonces, needed by the TPM itself or that have to be sent back to the caller;

2. execute an authorized command which acts on some TPM protected objects, after having verified command's consistency, i.e., its authentication and integrity;

3. send back to the caller a reply about the just issued authorized command, by stating whether everything went fine or not.

The MiTM lies in the middle of the communication channel between the caller and TPM actors and it is not constrained to necessarily follow the protocol rules. In fact, the MiTM "model" acts like "an active saboteur, [one] who may impersonate another user and may alter or replay the message" [9]. We model him as an actor able to:

1. open an authorization session with the TPM, like the caller. However no authorized command can be generated, since the MiTM does not know the authorization shared secret;

2. forward legal messages coming from the caller going to the TPM and viceversa.

[5] Büchi automata are finite state automata with infinite inputs able to recognize ω-regular languages.

[6] That is also the reason why we removed cryptographic has functions away from our model description.

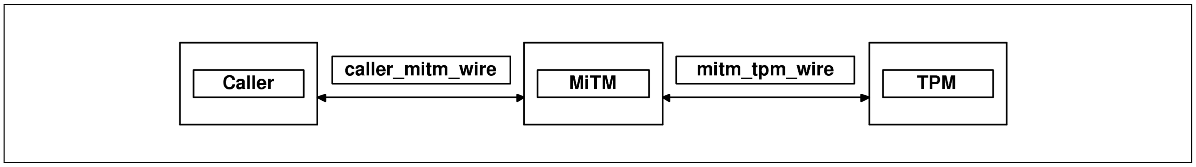

Figure 5. MiTM environment.

3. store in transit messages;
4. inject previously stored message, in attempt to perform replay attacks;
5. close opened authorization sessions by sending malformed packets.

Once defined the model of the system we have to formalize the property which the system should always satisfy. In order to define such a property, we associated three session states, namely *Failed*, *Success* and *Unknown*, to the TPM and the Caller.

We say that during a protocol session, the TPM enters the Failed state when it receives a request command with wrong parameters such as nonces, session parameters and so on. When the TPM receives such a wrong command, it closes the connection, if it has been previously opened. Instead, if the command is a valid one, the TPM executes it and after the execution, it enters into Success state, sending back to the Caller the command answer. In all the other cases the TPM session state will be Unknown.

On the Caller side, we used the Failed state when the Caller receives a command answer with wrong parameters. If so happen, the Caller closes the connection and enters into Failed state. Instead, if the Caller receives the right command answer, it enters into the Success state and closes the connection. In all the other cases the Caller session state will be set to Unknown. Using these three states, we are able to define an OIAP logical property as follows.

In order to perform a replay attack, a MiTM has to be able to inject to the TPM a captured authorized command and let this command be executed, without any caller knowledge. If such elusion occurs, the caller session may theoretically fall into two main states, namely either *Failed* or *Success* state. Indeed, the latter case cannot happen because the MiTM has no ability to reply with a legal authorized command answer, unless he knows the shared secret protecting the target resource. On the other hand, the TPM authorization session may fall into either an *Unknown* or a *Success*. Thus, if the caller authorization session falls into the *Failed* state and the TPM authorization session falls into the *Success* state, the MiTM has been able to perform a replay attack successfully, as described in Section 5.

Thus a system where a MiTM has no success has to be characterized by the following:

Correctness Property 1 (Session Understanding) *Let S be a set of sessions of the OIAP, and T_t and T_c the set of the states of respectively the TPM and the Caller during all the sessions contained in S. Given $t_{\mathrm{s}} \in T_t$ and $c_{\mathrm{s}} \in T_c$, both bound to the same authorization session $s \in S$, then*

$$\nexists s \in S \,:\, (t_{\mathrm{s}} = Success \wedge c_{\mathrm{s}} = Failed)$$

In other words, Property 1 means that both TPM and Caller session states should be synchronized[7].

6.2. Results of the analysis

Both the model M and the above property have been given as input to SPIN which exposed two major OIAP drawbacks, namely a DoS attack and a more dangerous trouble related to the counter-replay mechanism adopted by the TCG specification.

While the DoS attack is already considered by the TCG specification and it is not an issue at all in this context (although it exists), the counter-replay measures weaknesses are, indeed, a serious security issue, since, it is possible to perform a *straight replay* attack as shown in the trace depicted in Figure 6. More precisely, after the OIAP session is opened by the TPM on behalf of the Caller, the MiTM stores the authorized command issued by the Caller and resets (i.e., a forced "close") her just opened authorization session. At this point, the Caller enters into the state *Failed*, while nothing can be said about the TPM, whose corresponding session is still opened and thus set to an *Unknown* state. Sooner or later the MiTM will be able to replay the previously captured authorized message, without the Caller knowledge. At this point, no changes have been done on the Caller session state, while the TPM enters into a *Success* state. As we can see, this is in contrast with the logical property aforementioned, leading to an inconsistent session understanding between the involved parties.

7. A proposed solution

As just mentioned, rolling nonces were the mechanism adopted by OIAP for avoiding replay attacks. Our attack,

[7] Although we considered the case of $t_{\mathrm{s}} = Failed \wedge c_{\mathrm{s}} = Success$ in the OIAP PROMELA description we made, the logical property here exposed does not consider it since it should not happen in a real case scenario.

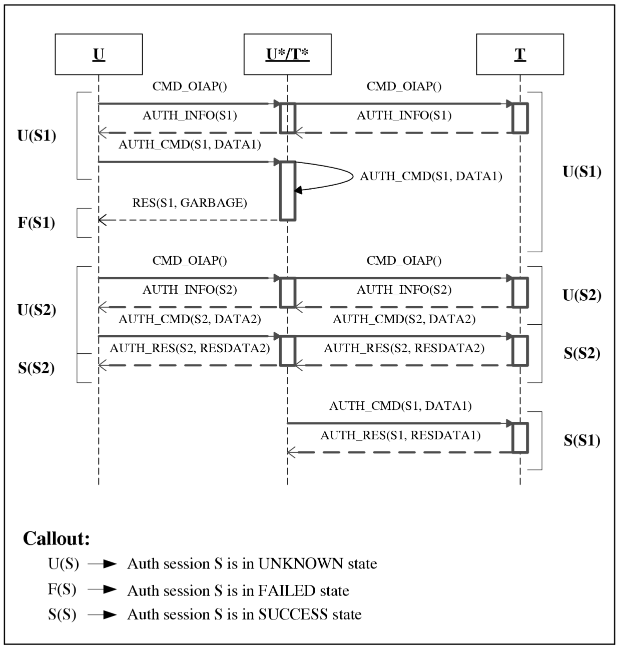

Figure 6. Overall OIAP replay attack.

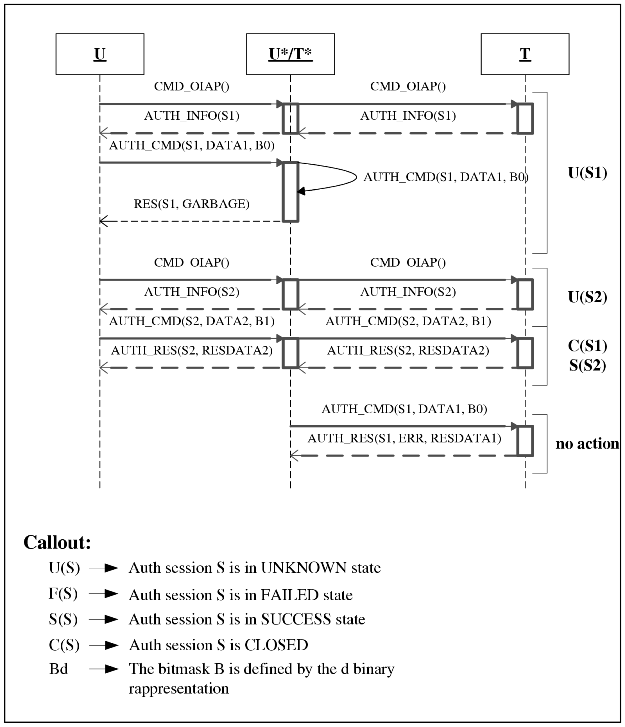

Figure 7. Overall OIAP replay attack's solution.

however, shows that the mechanism was not sufficient confirming that, as just noted by other authors ([16, 15]), rolling nonces are not a silver bullet against replay attacks.

In our specific case the attack was possible because of the possibility of having multiple open sessions and the lack of a coherent and synchronized "session knowledge" between the parties involved. In fact, at the end of our replay attack, **T** and **U**, have a different knowledge about the session state.

The countermeasure we devised in order to avoid this situation is based on the introduction in any authorized exchanged message between the parties, of a new field whose value is computed by the user. Such a field is a *bitmask* and represents the user knowledge about the state of all authorization sessions previously opened. Its value is computed by the user according to the following rules:

- the *i-th* bit is set to 0 if the *i-th* authorization session is considered either *open* or in an *unknown* state;
- the *i-th* bit is set to 1 if the *i-th* authorization session is considered *failed*, i.e., either a *reset* or an erroneous message[8] has been received as a response to an authorized command previously sent (see step 3b, Figure 2).

Using such an information, which will be protected from tampering by using HMAC, the TPM will become aware of the user knowledge about "established" connections and it will be able to detect any inconsistency. Furthermore, when it finds some incoherence between its own knowledge and the user one (e.g., user S_1 *Failed* and TPM S_1 *Unknown*), it closes the correspondent session. In such a way, it will be impossible for any attacker to exploit any pending session anymore in a significant manner as explained in Section 5.

Obviously the dimension of the bitmask will bound the number of open authorization sessions, thus the bitmask field should be big enough to represent a reasonable number of them.

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 sensibly tried to protect this protocol from both *replay* and *MiTM* attacks (more precisely *packet mangling actions*), our analysis showed 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 sessions state between

[8] So far, they are, however, indistinguishable.

the communicating parties. It is our opinion that the proposed solution can be improved in order to also allow for further misuse detection by allowing a party to detect MiTM presence. To this end, work is in progress for investigating such issues.

9. Acknowledgments

We would like to thank the anonymous referees for their useful suggestions and comments on this paper.

References

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

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

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

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

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

[6] 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.

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

[8] E. M. Clarke, S. Jha, and W. Marrero. Verifying Security Protocols with Brutus. *ACM Transactions on Software Engineering and Methodology*, 9(4):443–487, Oct. 2000.

[9] D. Dolev and A. C. Yao. On the Security of Public Key Protocols. *IEEE Transaction on Information Theory*, 29(2):198–208, 1983.

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

[11] C. Hoare. *Communicating Sequential Processes*. Elettronic version available at http://www.usingcsp.com, first published in 1985 by Prentice Hall International, 2004.

[12] G. J. Holzmann. A theory for protocol validation. *IEEE Transactions on Computers*, C-31(8):730–738, Aug. 1982.

[13] G. J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

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

[15] 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.

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

[18] 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.

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

[20] 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.

A. PROMELA source code

Global data, property and system initialization

```
mtype = {TPM_OIAP, ACK, CMD, ANS, FINISH}

typedef payload {
    bit session; /* only 2 sessions permitted */
    short ne;
    short no;
    bool cont;
    /* int hmac; */
}

chan caller_mitm_wire = [0] of { mtype,  payload };
chan mitm_tpm_wire = [0] of { mtype,  payload };
byte no[2];
byte ne[2];

#define CONSISTENT(s) (no[s] == r_no && ne[s] == r_ne)

#define FAIL 2
#define SUCCESS 1
#define UNKNOWN 0

byte tpm_session_understanding[2];
byte caller_session_understanding[2];

#define MAX_CALLER_ACTIVITY 100

#define PROPERTY ( (tpm_session_understanding[0] == FAIL \
&& caller_session_understanding[0] == SUCCESS) || \
(tpm_session_understanding[1] == FAIL \
&& caller_session_understanding[1] == SUCCESS) || \
(tpm_session_understanding[0] == SUCCESS \
&& caller_session_understanding[0] == FAIL) || \
(tpm_session_understanding[1] == SUCCESS \
&& caller_session_understanding[1] == FAIL) )

active proctype monitor(){

   atomic{
           PROPERTY -> assert(!PROPERTY);
   }

}

init{

   atomic {

      no[0] = 1; no[1] = 1; ne[0]=0; ne[1]=0;

      caller_session_understanding[0] = UNKNOWN;
      caller_session_understanding[1] = UNKNOWN;
      tpm_session_understanding[0] = UNKNOWN;
      tpm_session_understanding[1] = UNKNOWN;

      run tpm(0);    /* run tpm(1); */
      run mitm();
      run caller(0); /* run caller(0);        */

      timeout ->
         mitm_tpm_wire!FINISH(0,0,0,0);
         caller_mitm_wire!FINISH(0,0,0,0);
   }
}
```

Caller code

```
proctype caller(bool cont){

   byte r_no, r_ne;  /* received even nonce and odd nonce */
   bit r_session;    /* received session */
   do
   :: no[0] + no[1] > MAX_CALLER_ACTIVITY -> break
   :: else ->

         caller_mitm_wire!TPM_OIAP(0,0,0,0);
         caller_mitm_wire?ACK(r_session,r_ne,_,_);

              do
         :: no[0] + no[1] > MAX_CALLER_ACTIVITY -> goto END
         :: else ->

            atomic {
                         no[r_session] = no[r_session] + 2;
                     caller_session_understanding[r_session] = UNKNOWN;
                         caller_mitm_wire!CMD(r_session, r_ne, no[r_session], cont);
                           }
                           atomic {
                              caller_mitm_wire?ANS(eval(r_session), r_ne, r_no, _);

                              if
                              :: !CONSISTENT(r_session) ->

                  /* Exit without retrying */
                                  caller_session_understanding[r_session] = FAIL; break

                              :: !CONSISTENT(r_session) ->
                                     caller_session_understanding[r_session] = FAIL;

                              :: CONSISTENT(r_session) && cont ->
                                  caller_session_understanding[r_session] = SUCCESS;

                              :: CONSISTENT(r_session) && !cont ->
                                  caller_session_understanding[r_session] = SUCCESS;
                                  break /* Exit without retrying */
               fi;
            }
         od;
   od;

END:
  printf("End Caller")
}
```

TPM code

```
proctype tpm( bit session ){

   byte r_ne, r_no;  /* received even and odd nonce */
   bool r_cont;      /* session continued */
   bool consistent;

   do
   :: mitm_tpm_wire?FINISH(_,_,_,_) -> break
   :: mitm_tpm_wire?TPM_OIAP(_,_,_,_) ->

           atomic {
         ne[session] = ne[session] + 2;
         tpm_session_understanding[session] = UNKNOWN;
         mitm_tpm_wire!ACK(session, ne[session], 0, 0);
      }
      do
      :: mitm_tpm_wire?FINISH(_,_,_,_) -> goto END
      :: atomic {
         mitm_tpm_wire?CMD(eval(session), r_ne, r_no, r_cont);
         consistent = CONSISTENT(session) };

         if
         :: atomic { consistent ->

               tpm_session_understanding[session] = SUCCESS;
               ne[session] = ne[session] + 2;
               mitm_tpm_wire!ANS(session, ne[session], r_no, r_cont);

               if
               :: r_cont -> skip
               :: else -> break
               fi;
            }
         :: else -> atomic {
               /*
                * tpm_session_understanding[session] = FAIL;
                *
                * The command sent on this session was not successful due to
                * nonce mismatch but we're gonna to serve another session,
                * giving a wrong ans back to the caller.
                *
                * There's no need for a FAIL tpm session since we're gonna try
                * to serve another one (setting its state to UNKNOWN)
               */

               mitm_tpm_wire!ANS(session, 0,0,0);
               break;
            }
         fi;
      od;
   od;
END:
  printf("End TPM")
}
```

MiTM code

```
proctype mitm(){

   payload r_oiap, r_ack, r_cmd, r_ans;

   bit r_oiap_session, r_ack_session, r_cmd_session, r_ans_session;
   short r_oiap_ne, r_ack_ne, r_cmd_ne, r_ans_ne;
   short r_oiap_no, r_ack_no, r_cmd_no, r_ans_no;
   bool r_oiap_cont, r_ack_cont, r_cmd_cont, r_ans_cont;
   bool sent_oiap;

   do
   :: caller_mitm_wire?FINISH(_,_,_,_);
   :: caller_mitm_wire?TPM_OIAP(r_oiap_session,r_oiap_ne,r_oiap_no,r_oiap_cont);
   :: caller_mitm_wire?ACK(r_ack_session,r_ack_ne,r_ack_no,r_ack_cont);
   :: caller_mitm_wire?CMD(r_cmd_session,r_cmd_ne,r_cmd_no,r_cmd_cont);
   :: caller_mitm_wire?ANS(r_ans_session,r_ans_ne,r_ans_no,r_ans_cont);

   :: caller_mitm_wire!TPM_OIAP(r_oiap_session,r_oiap_ne,r_oiap_no,r_oiap_cont);
   :: caller_mitm_wire!ACK(r_ack_session,r_ack_ne,r_ack_no,r_ack_cont);
   :: caller_mitm_wire!CMD(r_cmd_session,r_cmd_ne,r_cmd_no,r_cmd_cont);
   :: caller_mitm_wire!ANS(r_ans_session,r_ans_ne,r_ans_no,r_ans_cont);
   :: caller_mitm_wire!ANS(0,0,0,0);

   :: mitm_tpm_wire?TPM_OIAP(r_oiap_session,r_oiap_ne,r_oiap_no,r_oiap_cont);
   :: mitm_tpm_wire?ACK(r_ack_session,r_ack_ne,r_ack_no,r_ack_cont);
   :: mitm_tpm_wire?CMD(r_cmd_session,r_cmd_ne,r_cmd_no,r_cmd_cont);
   :: mitm_tpm_wire?ANS(r_ans_session,r_ans_ne,r_ans_no,r_ans_cont);

   :: mitm_tpm_wire!TPM_OIAP(r_oiap_session,r_oiap_ne,r_oiap_no,r_oiap_cont);
   :: mitm_tpm_wire!ACK(r_ack_session,r_ack_ne,r_ack_no,r_ack_cont);
   :: mitm_tpm_wire!CMD(r_cmd_session,r_cmd_ne,r_cmd_no,r_cmd_cont);
   :: mitm_tpm_wire!ANS(r_ans_session,r_ans_ne,r_ans_no,r_ans_cont);
   :: mitm_tpm_wire!ANS(0,0,0,0);
   od
}
```