CS259 Project Report

Security Analysis of Remote Attestation

Lavina Jain (jainl@stanford.edu)
Jayesh Vyas (jayesh@stanford.edu)

Contents:

1. Abstract

Remote attestation is a method by which a host (client) authenticates it's hardware and software configuration to a remote host (server). The goal of remote attestation is to enable a remote system (challenger) to determine the level of trust in the integrity of platform of another system (attestator). The architecture for remote attestation consists of two major components: Integrity measurement architecture and remote attestation protocol.
The remote attestation protocol proposed by IBM [3] is vulnerable to Man-In-The-Middle attack. We propose a modified version of the protocol that is secure. We have implemented a model of our proposed protocol on Murphi [7] to analyse the security properties of the protocol. This report describes the design of our protocol and it's analysis using Murphi. We conclude that our proposed protocol is secure under certain assumptions of trust and can be used for remote attestation.

2. Remote Attestation Protocol

The remote attestation protocol that we propose is illustrated in Figure 1. We have integrated the process of remote attestation in the SSL/TLS handshake protocol. We suggest a modification to the client key exchange message in SSL/TLS handshake protocol using keys and signatures generated by TPM so as to provide the required level of security and trust. The protocol steps are:

1. Client/attestator C creates a non-predictable nonce and sends it along with it's identity to the server/challenger S. This is same as client hello message in SSL/TLS handshake.
2. Server responds with a server hello message. This message contains a non predictable nonce generated by the server and the server's certificate signed by a trusted CA. This is the same as server hello message in SSL/TLS.
3. The message that the client sends back to server is similar to client key exchange message in SSL/TLS with the following modifications:
    a. The client sends measurement list (ML) along with the session secret encrypted with server's public key. Refer to Appendix II to read how ML is generated.
    b. The client owns a pair of public/private RSA keys called AIK keys generated by TPM. It also obtains an AIK certificate which contains the AIK public key signed by a trusted CA. The client sends this AIK certificate to the server to authenticate itself.

c. Client sends a TPM Quote response to the server. In order to obtain a TPM Quote from TPM, client sends a hash of the two nonces and the session secret to the TPM and requests a quote signed by AIK. TPM returns signature over PCR values and given hash by AIK private key.
d. The server validates whether the client's AIK certificate was signed by a trusted CA and belongs to a genuine TPM. Then it verifies the freshness of Quote response by comparing hash of nonces and secret with the signed hash. Then it validates the integrity of ML by verifying the hash of ML against the PCR value in signature. Lastly, server validates individual entries in the ML by comparing the hashes against acceptable values.

4. If the integrity of the client platform is trusted by server in the above step, then the server and client continue to exchange messages as in SSL/TLS handshake protocol to establish a secure session.

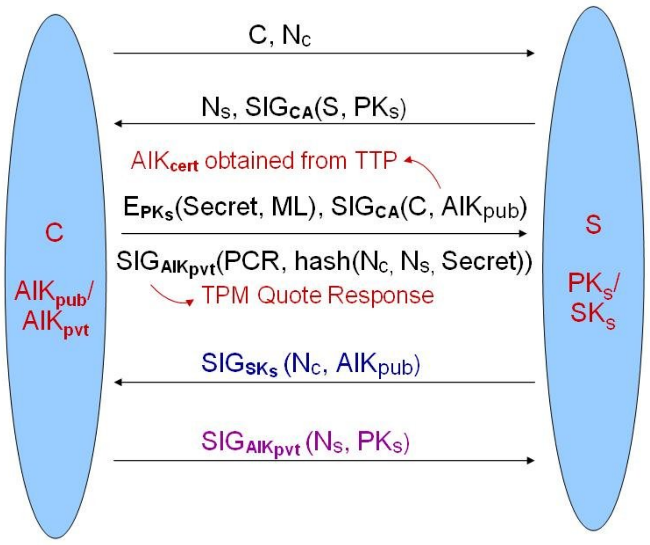

Figure 1: Remote Attestation Procotol

3. Threat Model

We have analyzed the following threat models:

1. Replay Attack: A malicious attesting system can replay old values of measurements and TPM Quote that correspond to a valid platform (before the system was corrupted).
2. Masquerading: An attacker can send measurement list and TPM Quote of another valid system.
3. Tampering: An attacker may tamper with measurement list or TPM Quote.
4. Malicious measurement agent: It may report incorrect integrity values.
5. Hardware attacks: An attacker may reset PCRs, and store in them new values corresponding to a trusted measurement list. It may break open the TPM and retrieve private signing key (AIK private).

4. Murφ Model

To analyze the security properties of remote attestation protocol we used a finite state model checker Mur*φ* [7].

The modeled network has the following entities:

1. Client – Attestator, who wants to attest it's platform to remote server.
2. Server – Challenger, who wants to determine it's trust in client.

3. Intruder – Malicious host on the network.
4. Attacker – Malicious entity on the client.

Behavior of Entities

The server strictly follows the protocol steps. The client also does the same, unless a malicious software (called the Attacker) is ACTIVE on the client.

To keep things simple PCR and ML values 0 are considered trustworthy by the server, and values 1 are considered untrustworthy. When there is no attacker running on client, its PCR and ML are both 0. But when the attacker becomes ACTIVE, the PCR is set to 1. A client with an ACTIVE attacker may not follow the protocol steps exactly.

Intruder is another host in the network, capable of eavesdropping on all packets on the networks storing their contents and replaying them. It can act as a client or server of the remote attestation protocol. Since it has malicious software running on it, so its fingerprint in its PCR is untrustworthy (that is 1).

Invariants

For the system to be secure, the following invariants should always hold:

Server Authentication:
INV1: If Client is talking to Server, then Server is talking to Client.

Client Authentication:
INV2: If Server is talking to Client, then Client is talking to Server.

Secrecy:
INV3: If server attests a client, then they share the same secret.
INV4: If client attests itself to a server, then it shares the same secret with the server.
INV5: Intruder should not learn server's secret if the server is talking to a client.
INV6: Intruder should not learn client's secret if the client is talking to a server.

Remote Attestation:
INV7: If server attests a client, then attacker should not be active on that client
INV8: Server must not attest an intruder

Flags to Control Model Behavior

The following flags can be used to control the model checker's behavior:

- FOUR_STEP: Run four-step protocol.
- FIVE_STEP: Run five step protocol.
- CERT_FORGED: Allow Intruder to forge certificates.
- SUPER_ATTACKER: Enable hardware attacks (Resetting PCRs, compromised TPM etc) on client.
- TPM_COMPROMISED: TPM on intruder is compromised (it knows AIK_{pvt}).
- SERVER_NONCE_PREDICTABLE: Random nonce generated by the server is known to client/intruder.

Note: Please ensure that you use `make run` to run the Murphi checker, or use the appropriate flags to run it.

5. Analysis and Attacks

We followed a rational construction approach using Murphi to design a secure protocol. We started with a 3-step protocol, and added fourth step and fifth step because of attacks found on the 3-step and 4-step protocols. Then we analyzed the most secure (5-step) version of the protocol with different flags that are described in Section 4.

Three-Step Protocol

The authentication invariants as well as secrecy invariants in 3-step protocol fail because of the following reasons:

- There is no server authentication.
- An intruder can impersonate as a server and fool the client to talk to him. If the client is happy to talk to the intruder, the intruder can establish a session with client. At the same time, the intruder can initiate a session with server, and replay all messages that he receives from client except the encrypted message in the third step. Intruder encrypts the secret obtained from client with server's public key in this message. So, server thinks that that he is talking to the client but the client is talking to the intruder (MITM attack).

Four-Step Protocol

Adding fourth step provides server authentication. So, the server authentication invariant does not fail. However, the client authentication invariant and the secrecy invariant still fail because of the attack described above.

Five-Step Protocol

Adding fifth step prevents the intruder from fooling the server. The intruder cannot generate this message because it is signed by client's AIK private key. If the intruder replays this message to the server, it is revealed to the server that client is talking to somebody else and not the server himself. Hence, the server aborts the handshake.

Intruder Capable of Forging Certificates

The flag CERT_FORGED can be used to observe the system behavior when intruder can forge certificates (both server and client). It turns out that this is the most dangerous attack and all the invariants fail!
To understand this, see the figure below:

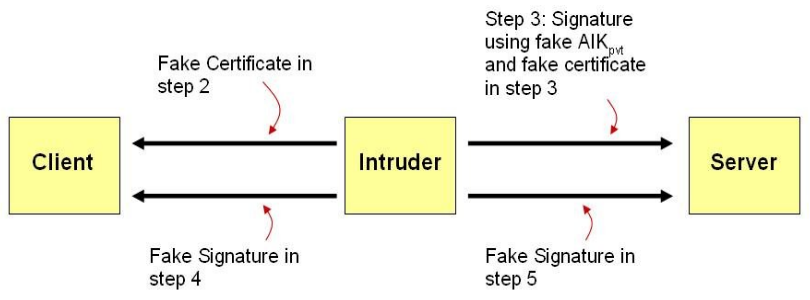
Step 3: Signature using fake AIK$_{pvt}$ and fake certificate in step 3
Fake Certificate in step 2
Client
Intruder
Server
Fake Signature in step 4
Fake Signature in step 5

Figure 2: Man In The Middle attack if intruder can forge certificates

The intruder can mount a **Man In The Middle (MITM)** attack on the system. By forging server's certificate, and sending its own public key in the certificate, it impersonates as a server to the client and attests the client. To attest itself to the server, it sends a forged certificate having a fake AIK_{public} key, and the signature using a fake $AIK_{private}$ over the nonces, secret and PCR. Failure of invariants 1-6 and 8 follows directly from MITM attack. Invariant 7 fails because of intruder's behavior of generating messages from known nonces, secrets and certificates. To attest a client having a malicious attacker running on it, intruder generates messages 2 and 4 of the protocol (from client to the server).

Hardware Attacks and Attacks on IMA

The attacker can hide its presence on client by one of the following ways:

- Exploiting some bug in Integrity Measurement Architecture (IMA): It could not measure the malicious software and put it in PCR.
- Resetting PCRs without rebooting the system, and pushing known good values into the PCRs.
- Changing a running process's image (through DMA).
- Breaking the TPM and retrieving secret key used for signing PCR.

The final impact of any of these attacks would be a situation in which there is a malicious application running on client, but its fingerprint is not recorded in PCRs. An attacker capable of mounting such an attack can be enabled by setting the flag SUPER_ATTACKER to true in the Murphi code.
It is obvious that invariant 7, "if server attests a client, then attacker should not be active on that client" fails in this situation.

Compromised TPM

Suppose an intruder is able to break the TPM and is able to retrieve the private signing key ($AIK_{private}$). It will use this key to sign the nonces, secrets and PCRs (messages 3 and 5 of the protocol). This way it will be able to attest itself to the server.
The flag TPM_COMPROMISED can be used to enable such an attack in the Murphi model. Invariant 8, "Server must not attest an intruder" fails in this case. But no other invariant fails. This shows that even if one of the TPMs is compromised, it doesn't jeopardize security for other systems (although there could be attacks not caught by us or Murphi!).

Predictable Server Nonce

Consider the case when the Server uses a bad source of randomness, and so its nonce is predictable by the intruder and client. This situation can be modeled by the flag SERVER_NONCE_PREDICTABLE.

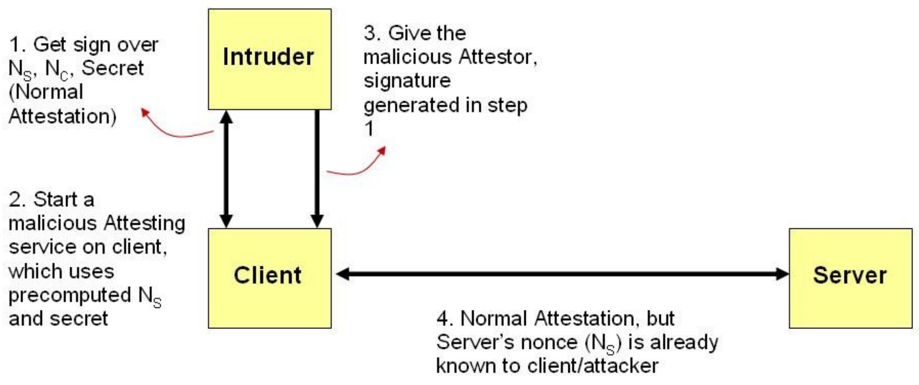
1. Get sign over N_S, N_C, Secret (Normal Attestation)
Intruder
3. Give the malicious Attestor, signature generated in step 1
2. Start a malicious Attesting service on client, which uses precomputed N_S and secret
Client
Server
4. Normal Attestation, but Server's nonce (N_S) is already known to client/attacker

Figure 3: Attack when server's nonce is predictable

In this situation the client can pre-compute signature over that nonce and PCRs before a malicious application starts, that is when the PCRs have trustworthy value. But a client that does so is itself malicious!

To mount this kind of attack, two hosts need to collude. One of the hosts (intruder) acts as a server, and takes signature from the other host (client) over the predicted nonce, secret and PCRs (used in step 3 of protocol). After this it starts a malicious attestation service on the client, and passes on this signature and predictable nonce to it. Then this client having a malicious attestation service can attest itself to a server whose nonce was predicted.

6. Conclusion

The analysis of protocol using Murphi verifies that the proposed protocol for remote attestation is secure and can be used to attest a client to a remote server. A malicious system cannot attest itself unless one or more of the following conditions hold true:

- Certificates can be forged.
- Hardware attacks such as DMA or resetting PCR are possible.
- TPM is broken and private key is retrieved.
- Server's source of entropy is weak and hence it's nonce is predictable.

It is verified by our model on Murphi that the protocol breaks under the above conditions.
Thus, we conclude that the protocol is secure under the following assumptions:

- CRTM (BIOS) and CPU are trusted.
- TPM is compliant with TCG specifications.
- IMA (Integrity Measurement Architecture) is bug free.
- A running process behavior or image cannot be changed (e.g. via DMA).

Appendix I: Acronyms

TCPA: Trusted Computing Platform Architecture
TCG: Trusted Computing Group
CA: Certification Authority
TPM: Trusted Platform Module
AIK: Attestation Identity Key
IMA: Integrity Measurement Architecture
CRTM: Core Root of Trust of Measurement
PK: Public Key
SK: Secret Key
SIG: Signature
ML: Measurement List

Appendix II: TPM and Integrity Measurement Architecture

Trusted Platform Module (TPM) is a tamper proof piece of hardware (usually a chip) which provides secure cryptographic services, such as encryption/decryption and hashing, and secure storage. It must follow the standards specified by Trusted Computing Group (TCG) [1].

Services of TPM relevant to this protocol are secure key and data storage. Each TPM is assigned a certificate from a trusted third party, which can be used to authenticate the TPM. The private key, known as Attestation Identity Key (AIK), corresponding to that certificate is securely stored inside the TPM and cannot be retrieved from it. But the TPM provides an interface to get signature over some data using AIK. TPM also provides secure storage through a set of registers called Platform Configuration Registers (PCRs). The only way to change the value of this register is to use the command PCR_Extend (new_hash), which concatenates the old value in PCR with the new_hash value, calculates the hash over the concatenated value, and overwrites the old PCR value with this hash. The only other way to change PCR values is to reset it, which can be done only at system boot up, and not at any other time.

IBM proposed an integrity measurement architecture in [3] based on the above mentioned properties of the TPM. We use this architecture to build our remote attestation protocol. The objective is to have an accurate fingerprint of every piece of software which can affect the runtime environment of the system. The architecture assumes the following things:

1. CRTM that initiates chain of measurements is trusted and cannot be tampered.
2. TPM is compliant with the standards of TCG.
3. CPU is trusted.
4. Hardware attacks are not possible on the system, like changing a running process's image after it has started using DMA

To achieve this, there has to be a core root of trust, which is usually the first few bytes of BIOS. This is known has Core Root of Trust of Measurement (CRTM). When the system boots up, PCRs in the TPM are reset. The control then passes to CRTM, which calculates the hash of the BIOS and *extends* this hash to PCRs. The control is then passes to BIOS, which calculates the hash of the operating system (that is the kernel image), and extends this hash to PCRs. Then the system boots up.

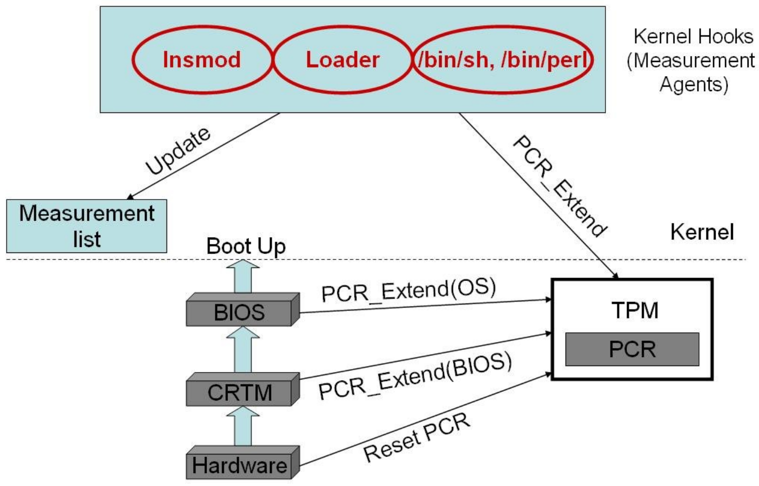

Figure 4: Integrity Measurement Architecture

To measure everything else that might affect the runtime behavior of the system, kernel sets up certain hooks at appropriate places in the system. These hooks are: insmod utility for loadable modules, loader for user level programs and shared libraries, and script interpreters for scripts. Before any software is started, the hash over its binary is calculated by the appropriate hook, and is added to the measurement list (ML) maintained by the kernel. This hash is also extended to the PCRs. Thus before any software runs, its fingerprint is recorded into the PCRs in an incremental fashion, and thus cannot be reverted back. So a system's measurement list has a list of hashes (one for every piece of executable), and the PCR has the incremental hash of all these hashes. Note that at system boot up, the kernel calculates the hash of BIOS and its own hash, and puts them into the measurement list. Thus we can say that measurement list together with PCR completely give the measurement of the runtime behavior of the system.

Appendix III:References

1. Trusted Computing Group, http://www.trustedcomputing.org.
2. Reiner Sailer, Trent Jaeger, Xiaolan Zhang, Leendert van Doorn, “Attestation-based Policy Enforcement for Remote Access”, In Proceedings of the 11th ACM conference on Computer and Communications Security, October 2004.
3. Reiner Sailer, Xiaolan Zhang, Trent Jaeger and Leendert van Doorn, “Design and Implementation of a TCG-Based Integrity Measurement Architecture”, In Thirteenth Usenix Security Symposium, pages 223-238, August 2004.
4. H. Maruyama, T. Nakamura, S. Munetoh, Y. Funaki, Y. Yamashita, “Linux with TCPA Integrity Measurement”, IBM Research Report, Tokyo Research Laboratory, Japan, Jan. 2003.
5. D. Safford, J. Kravitz, and L. van Doorn, “Take Control of TCPA”, Linux Journal, pages 50-55, August 2003.
6. Karim Faez, Ashkan H. Karimabad, “Open-Source Applications of TCPA Hardware”, In International Journal of Computer Science and Network Security, Vol. 7, No. 3, March 2007.
7. Murphi description language and verifier, http://verify.stanford.edu/dill/murphi.html.