Improving Trusted Tickets with State-Bound Keys

Jan Nordholz[1], Ronald Aigner[2], and Paul England[2]

[1] TU Berlin, Germany
jnordholz@sec.t-labs.tu-berlin.de
[2] Extreme Computing Group, Microsoft Research
{ronald.aigner,paul.england}@microsoft.com

Abstract. Traditional network authentication systems like Windows' Active Directory or MIT's Kerberos only provide for mutual authentication of communicating entities, e.g. a user's email client interacting with an IMAP server, while the user's machine is inherently assumed to be trusted. While there have been first attempts to explicitly establish this trust relationship by leveraging the Trusted Platform Module, these provide no means to directly react to potentially relevant changes in the client's system state. We expand previous designs by binding keys to the current platform state and involving these in the network authentication process, thereby guaranteeing the continued validity of the attestee.

1 Introduction

As by now a TPM can be found in almost every contemporary desktop computer, so has its adoption in security software become widespread. Chipset firmware, OS loader programs and finally the OS kernel itself use it to log the respective subsequent components of the boot chain into a verifiable log buffer, harddisk encryption solutions like BitLocker[3] use it to seal their cryptographic material once the booting process has finished, and remote attestation protocols use the recorded system events and the platform state represented by the TPM's registers to judge the trust to place into that system according to a certain policy.

Network authentication services, on the other hand, usually do not bother with establishing a trust relationship with the system a client logs on from; in fact – although this is an orthogonal observation and neither of the two features strictly requires the other – most do not even identify the client host[1]. While identifying the host may be undesirable or just unnecessary, garnering trust can only be beneficial.

Especially in a corporate setting, where the IT department can exert tight control over the tolerated configurations, trust in a client can be defined as the combination of two assertions: first, that the client host is running an uncompromised operating system, and second, that the configuration of the system remains inside a predefined subset of the configuration state space theoretically

[1] Except for logging its purported identity, represented e. g. by its IP address.

M. Huth et al. (Eds.): TRUST 2013, LNCS 7904, pp. 37–46, 2013.

allowed by the OS. Our approach presents a TPM-assisted solution that allows a service instance to gain reliable insights to both of these questions, while at the same time integrating nicely into the well-known Kerberos network authentication framework. We also elaborate on viable techniques for withdrawing credentials given out based on these observations.

2 Background

The Trusted Platform Module was conceived as a vessel for placing trust into a running system. Part of that trust relationship is that private parts of asymmetric encryption keys never leave the TPM. A special asymmetric key is the so-called Endorsement Key (EK), because it is endorsed by the manufacturer of the TPM or the platform manufacturer. The endorsement comes in the form of a certificate containing the public portion of the EK signed by the manufacturer. For TPM 2.0 devices the EK is generated from a unique seed using a key template. This process will always generate the same EK on a TPM, if the seed stays the same. This EK itself, accompanied by its EK certificate, would suffice to sign data and prove its uniqueness. However, on TPM 1.2 devices an EK cannot be used to create signatures. Our solution should target the broadest set of TPMs, so we used this limitations as a prerequisite. The one-to-one mapping between EKs and hardware platforms has several benefits, e.g., it allows for the revocation of individual keys if their factorization becomes known. However, it comes with the drawback of making platforms exposing their EK easily identifiable and thus traceable.

To ameliorate this problem, the EK is used to establish trust in an additional "pseudonymous" key, which is also non-exportable and therefore tied to the platform it was generated on, but does not share the privacy concerns and usage limitations of EKs. These Attestation Identity Keys (AIKs) are certified by a certification authority (CA) based on the EK certificate presented together with the AIK. The CA can inspect the EK certificate and the AIK public portion and generate an activation challenge for the TPM. Only a TPM which possesses both keys can decrypt the challenge. Part of the challenge is a secret, which can be a symmetric key to decrypt the certificate for the AIK issued by the CA. The AIK certificate can then be used to identify the AIK as originating from a TPM[7]. The concept of AIKs allows to use different keys for different services, which further protects users from cross referencing keys.

With this technique the exposure of the EK can be limited to a single trusted service. As we are chiefly discussing corporate settings in our approach, this seems like a reasonable assumption. For scenarios where this might not be desirable, there is also the option of engaging in a zero-knowledge protocol to prove a platform's authenticity[2]. Whichever protocol is used, further communication partners of the platform can then trust a given AIK by virtue of a matching signature by a "privacy CA" instead of relying on the EK.

Another important property of a TPM are non-resettable Platform Configuration Registers (PCR). A PCR can be extended through a trapdoor function,

which takes as input the PCR's previous value and a new value to incorporate into the new PCR value. Because these non-resettable PCRs take a well-known value only when the TPM is powered on and can only be extended after that, the value of a PCR cannot be rolled back, only forward. The TCG defines 16 non-resettable PCRs for PC architectures. Most of these contain well defined values, as specified in [8]. The BIOS of a PC will extend hashes of boot events into the PCRs. The operating system loader and the operating system itself continue these measurements throughout the boot process. Such boot events include hashes of all the binaries loaded and executed. In parallel the system keeps a log of these events containing additional information, such as name of binary, size, etc. Because the PCRs are non-resettable, they can be used to verify the integrity of the TCG event log. Under the assumption that every stage of the boot chain checks the validity (cryptographic signatures, soundness of configuration, etc.) of the subsequent stage and writes the resulting measurement into the event log and a hash into the PCRs, there is reliable proof that a certain boot chain has been traversed. This ensures that even if a component of the chain is later found to be malicious (due to a zero-day vulnerability), it cannot erase its presence or alter its image to appear benign without creating a mismatch between log and PCR values. Using a TPM-held asymmetric key (e. g. an AIK), the TPM can then create and sign a "quote" of the PCR values. This way trust in the AIK certificate can be extended to the TCG event log.

A TPM may also create a key bound to the values of a selection of PCRs. That key can only be used as long as the selected PCRs have the same values used at the time of the creation of the key. Should one of the selected PCRs change its value, the key is rendered useless. We call these keys PCR bound keys or state bound keys.

3 Design

3.1 TPM-Based Protocol

Using the AIK signed PCR quote together with a TCG event log and AIK certificate allows an attestor to successfully verify the integrity of the TCG event log, tracing back the AIK to a cryptographic root of trust. However, the result of this attestation process only applies to the very moment in time when the quote was generated on the attestee – even while the examination of the attestor is running, new entries might be added to the attestee's event log. This would alter the system state and possibly also its conformance level to the attestor's standards. Therefore the basic AIK based variant of this protocol (as described in [11]) applies only to systems where there is little change in system state and/or this problem is negligible.

While the pragmatic approach to this problem might be to simply reduce the attack window by keeping the validity period of (possibly stale) attestation certificates at a minimum, there is also a systematic solution using state bound keys. At the cost of the generation of the state bound key, a platform can prove

at any time whether its state still matches the one which has been examined at the time of the creation of the PCR quote.

The common remote attestation protocol can be easily extended to include a new TPM-shielded state bound – ephemeral – key (EPH). Congruently to the EK → AIK trust transition above, the TPM can issue a creation certificate, which proves the basic characteristics of the EPH and is signed by the AIK. If this data is supplied to the attestation service, it can compare the quoted set of PCRs with the system state to which the EPH has been bound. If the two match, the EPH can then serve as an unforgeable proof of the client's system state.

This approach completely eliminates the need for an explicit validity period on the attestation certificate – it is valid as long as the EPH bound to it can be used and as the attestor trusts the components running on the system. However, the chosen method is of course unable to protect against unknown vulnerabilities and the ensuing possibility of memory-resident malware; however once the vulnerable component is identified, systems which have loaded (and therefore executed) a vulnerable version of this specific component can be denied attestation, effectively forcing them to upgrade to a safe version and to reboot.

All this requires that every significant change in system state actually renders the EPH unusable. Changes on the OS layer, such as subsequent loading of additional code modules is discussed in detail in related work[12,5]. Depending on the scenario there can be other forms of system state changes which should be considered significant, e. g., changes to the system code libraries. If an attestor trusts a system configuration it can also establish trust in the policy enforced on the attestee, which will change the PCRs if a known significant event occurs. This policy enforcement is technically trivial, as all necessary API elements[2] are available on today's major operating systems. The attestee can run a "watchdog" service, which enforces the requirements of the attestor.

Due to this scenario-dependent definition of the system state space, we have expanded the basic remote attestation protocol to allow the attestation service to push a list of state inspection requests to the client. These checks can be instantaneous, meaning that the results should be transmitted as part of the system quote, or continuous, i. e. changes to the requested property should be constantly monitored by the watchdog and EPH should be revoked (by extending a PCR and logging the event) if a property leaves the defined "safe state space".

These additional checks allow for easy determination of system configuration like the version of the operating system, the patchlevel of core libraries or the state of vital system services. As a side effect this also defeats the common "proxy" attack: the possibility for an attacker to have all TPM operations executed on a third, clean machine in order to make his own compromised machine appear innocuous can be thwarted by simply including a check that asks for the attestee's hostname or network interface configuration. If an attacker is acting

[2] This includes registering file system directories for kernel notifications, communicating with the TPM from userland – on Windows monitoring the System Registry is advisable, too.

as man-in-the-middle, the answer will not match the client address as seen by the attestation daemon.

3.2 Kerberos Integration

The Kerberos authentication framework and its surrounding ecosystem of glue libraries (libsasl, libgssapi etc.) have become the de-facto standard for large-scale authentication settings in Unix-based networks. While its definition, conventions and API designs span more than a dozen RFC documents, its basic messages are comparatively simple and easily extensible. At the heart of the protocol lie the message exchanges between client and Authentication Service (AS) and between client and Ticket Granting Service (TGS). The former establishes the client's identity by issuing a long-lived Ticket Granting Ticket (TGT) encrypted to the user's pre-shared key (more commonly the "password"), the latter can be requested to hand out tickets for individual services, provided that an appropriate permitting TGT can be presented.

The flow of information in both directions can be extended by supplying optional data in generic holes of the protocol which have been designed explicitly for this purpose. Requests allow for additional "preauthentication" data to be supplied, and returned tickets can be constrained to specific use cases or circumstances by filling in the so-called (and somewhat mis-named) "authorization" element.

The idea of integrating the additional trust gained by a remote attestation certificate into the Kerberos protocol leads to the fundamental question at which point the additional information should be fed into the protocol. Theoretically all three entry points (AS / TGS / kerberized service) would lead to a valid combination – we have consciously chosen the second approach in our design for the following reasons:

- As the attestation certificate includes EPH, including it into the request to the AS would mean that the TGT was bound to EPH, too. This breaks the Single Sign-On property of the Kerberos protocol, as the AS exchange would have to be repeated (and therefor the user's password reentered) whenever EPH expires.
- Establishing the TPM-based trust into the client includes many checks which do not actually involve the client interactively: verifying the RSA signature on the attestation certificate against the well-known attestation service key, parsing of contained policy compliance statements, unpacking of EPH modulus etc. These could be performed by each kerberized service individually – we deemed it more suitable to keep these operations inside the KDC instead of the server-side Kerberos libraries.

The only part of the proof which still has to be performed by the actual kerberized service is then a signature by the key EPH on a nonce chosen by the server. In order to keep the protocol backwards-compatible with standard Kerberos implementations, this requires an additional pair of messages to be exchanged after the service ticket itself has been presented and validated.

4 Implementation

The network service components of our scenario do not depend on a specific hardware configuration or feature set and they implement little new functionality beyond the well-established roles in the Remote Attestation protocol. We have therefore opted to build the Privacy CA and the Attestation Service as small standalone C♯ applications with only several hundred lines of code each.

The attestation client has been implemented on top of the TSS API as specified by the TCG[9]. Our Windows implementation uses the primitive operations provided by the recent `tpm.sys` kernel driver[4], whereas our Linux implementation uses the full TSS interface as provided by the TrouSerS `tcsd` daemon[1].

Finally, we have made small changes to the MIT Kerberos implementation and the GSS-API library to add our attestation components, thereby replacing about 200 lines and adding another 350, spread out over 12 files. The changes consist of the following logical items:

- inclusion of attestation certificate in TGS request
- inclusion of certificate in returned TGT
- introduction of a new GSSAPI context flag representing TPM-based trust
- protocol extension: extra message pair containing EPH signature
- creation and verification of extra message pair
- implementation of above concepts in GSS sample applications[3]

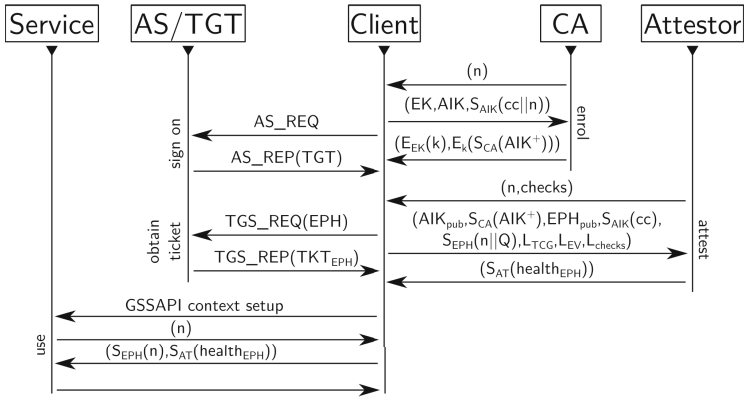

Fig. 1. Protocol visualization. Note that the AS exchange and the remote attestation can be completed in arbitrary order, as they do not depend on each other

We will now describe each part of the protocol (cf. also figure 1). TPM operations as defined in version 1.2 of the TPM Specification[10] are typeset in italics.

[3] This alone accounts for nearly 50% of the whole patch set.

1 **communication with Privacy CA**

1a The client connects to the Privacy CA, creates an AIK bound to a nonce chosen by the CA (*TPM_CreateIdentity*), and transmits the activation request.

1b The PCA validates the properties of the AIK, the soundness of the EK and the signature, and replies with an activation blob which is decrypted by the client using *TPM_ActivateIdentity* and stored for later use.

2 **remote attestation**

2a The client connects to the attestation service and receives a nonce and a list of system checks.

2b The client determines the current system state by repeatedly calling *TPM_PCRRead* and finally generates an ephemeral key EPH bound to the determined set of PCR values by executing *TPM_CreateWrapKey*. In the rare case of a change in PCR state during this operation, the step is repeated.

2c The client links EPH to AIK by executing *TPM_CertifyKey2*.

2d The client performs the requested checks on its state.

2e The client generates a quote on the current system state through *TPM_Quote2*, using EPH as the signing key and the combination of the TCG bootup event log, the log of accumulated system state change events and the results of the requested instantaneous system checks as "external quote data".

2f The client transmits the quote, the actual log data and check results, the required public key material to verify the signatures and the PCA certificate from step 1 back to the attestation daemon.

2g The attestation daemon verifies the trust chain, assesses the client's bootup chain, its history of monitored events and its responses to the requested checks and determines its compliance to a set of pre-defined policies (e. g. "Win8 patchlevel 2013/01/01" or "Ubuntu Lucid").

2h The daemon issues an attestation certificate that ties EPH to the determined list of fulfilled policies and transmits it to the client.

3 **Kerberos protocol**

3a During sign-on, the client obtains a regular TGT from the AS.

3b While trying to obtain a ticket for a particular kerberized service, the client checks for the presence of a valid attestation certificate. If one is found, the client includes it as preauthentication in its request to the TGS.

3c The TGS performs validity checks on the included certificate and stores a parsed version (which still includes the list of policies and the public part of EPH) as an authorization element in the resulting service ticket.

3d The client connects to the kerberized service, presents its ticket and indicates that additional operations to establish TPM-based trust are desired.

3e The service issues a nonce.

3f The client executes *TPM_Sign* on the nonce, thus producing an `SS_INFO` signature of it using EPH as the key, and transmits the result. Note that the attestation service has to verify that EPH's key properties specify the TPM_SS_RSASSAPKCS1v15_INFO signature scheme. Otherwise it would

be possible to create and sign a message with EPH which is indistinguishable from a system quote – a problem particular to this specific protocol, as AIKs are explicitly forbidden to engage in *TPM_Sign* operations.

3g The service verifies the signature using the public part of EPH and is then able to grant additional privileges according to the set of policies the client is now proven to fulfil.

Several parts of this protocol can be easily optimized. Step 1 does not have to be repeated at all if the AIK is persistently stored on the client – it may only be desirable in order to change the platform's pseudonym for the later stages, and doing so is possible at all times because a change in AIK does not invalidate already issued attestation certificates.

EPH on the other hand has to be generated at least once per bootup due to its dependency on the PCR values. As *TPM_Sign* is the only primitive cryptographic operation exposed by the TPM interface, it is impossible to tie the key closer into the establishment of the TLS context. If the system state changes and EPH becomes unavailable, existing connections which have been created through the above protocol will continue to work if they are not torn down explicitly. This is no technical challenge however, as the expiration of EPH has been caused by an extension of a PCR value, which can only have been issued by the OS or the userland component of our aforementioned policy watchdog: this daemon can also communicate this event to all processes which have running EPH-based network connections.

5 Evaluation

Our protocol incurs only modest overhead and has almost no surprising factors compared to regular incarnations of remote attestation. Measurements of boot components vary between 10 and 60 ms each (mainly dependent on the TPM model), so assuming a number of about 100 components – which represents a typical Windows7 installation; typical Linux installations tend to have less – yields a total slowdown for a cold boot of just a few seconds.

Due to the infrequent need for fresh RSA key pairs, these requests are usually satisfied by the key pregeneration cache of the TPM and thus do not incur an additional delay. The remote attestation protocol therefore completes in about 4 seconds (privacy CA) and 5 to 7 seconds (attestation service), depending on the length of the transmitted TCG log and the list of requested system checks. Our experiments indicate further that, once a system has performed its initial boot up and loading of additional driver modules has completed, the configuration of the system remains stable if a few basic optimizations are applied (e. g. the generation of additional PCR events for repeated loading and unloading of identical instances of the same driver module is suppressed). This holds true even if additional aspects of system configuration are being monitored, like watching core keys of the Windows Registry or the contents of `/lib` on a Linux installation. These only change during system upgrade procedures or deliberate system

reconfiguration by an administrator, in which cases the penalty of regenerating a TPM RSA keypair and repeating the attestation process seems admissible.

The only unique delay introduced by our protocol is the *TPM_Sign* operation on the nonce during the initialization of the GSSAPI session. Our measurements indicate an average duration of about 2.5 seconds per connection attempt – however this operation is only executed after the service ticket has been presented and both parties have agreed to engage in the "upgraded" handshake method, so this time is never wasted.

6 Conclusion, Future Work

The designed protocol extensions to the Kerberos framework allow for a convenient integration of hardware-based trust certificates, thus allowing network services to include this new trust dimension into the authentication process. Next steps may include extending the scenario to support mutual authentication, or to integrate TPM-shielded RSA keys into an adaptation of the Kerberos PKINIT extension (cf. [13]). Finally, the use of state-bound keys is not restricted to Kerberos, e. g. Goldman et al.[6] describe an approach to link TPM based attestation to SSL certificates. Their implementation differs in the reversed model (the service attempts to prove its validity to a client) and their reliance on short certificate expiration times and instantaneous certificate revocation by the watchdog, but is quite similar with respect to melding the current system properties into a TPM-bound health certificate. Combining and further exploring these techniques may provide valuable insights.

References

1. TrouSerS - the open-source software stack, `http://trousers.sourceforge.net`
2. Brickell, E., Camenisch, J., Chen, L.: Direct anonymous attestation. In: Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, pp. 132–145. ACM, New York (2004), `http://doi.acm.org/10.1145/1030083.1030103`
3. Corporation, M.: TPM and BitLocker drive encryption, `http://msdn.microsoft.com/en-us/library/windows/hardware/gg487306.aspx`
4. Corporation, M.: TPM platform crypto-provider toolkit, `http://research.microsoft.com/en-us/downloads/74c45746-24ad-4cb7-ba4b-0c6df2f92d5d/default.aspx`
5. Corporation, M.: Secured Boot and Measured Boot: Hardening Early Boot components against malware. Tech. rep., Microsoft Corporation (2012)
6. Goldman, K.A., Perez, R., Sailer, R.: Linking Remote Attestation to Secure Tunnel Endpoints. IBM Technical Paper (2006), `http://domino.research.ibm.com/library/cyberdig.nsf/1e4115aea78b6e7c.....85256b360066f0d4/fb0d5a04296a0bee852571ff0054f9fb`
7. Group, T.I.W.: A CMC profile for AIK certificate enrollment. Tech. rep., Trusted Computing Group (2011)
8. Group, T.P.C.W.: TCG PC client specific implementation specification for conventional BIOS. Tech. rep., Trusted Computing Group (2012)

9. Group, T.C.: TCG Software Stack (TSS) Specification Version 1.2 level 1. Tech. rep., Trusted Computing Group (2007),
http://www.trustedcomputinggroup.org/files/resource_files/6479CD77-1D09-3519-AD89EAD1BC8C97F0/TSS_1_2_Errata_A-final.pdf
10. Group, T.C.: TPM main specification level 2 version 1.2 revision 116. Tech. rep., Trusted Computing Group (2011),
http://www.trustedcomputinggroup.org/resources/tpm_main_specification
11. Leicher, A., Kuntze, N., Schmidt, A.U.: Implementation of a trusted ticket system. In: Gritzalis, D., Lopez, J. (eds.) SEC 2009. IFIP AICT, vol. 297, pp. 152–163. Springer, Heidelberg (2009),
http://dx.doi.org/10.1007/978-3-642-01244-0_14
12. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and implementation of a TCG-based integrity measurement architecture. In: Proceedings of the 13th USENIX Security Symposium, pp. 223–238. ACM (2004)
13. Zhu, L., Tung, B.: Rfc4556: Public key cryptography for initial authentication in kerberos (PKINIT) (2006),
http://tools.ietf.org/html/rfc4556