Trusted Tamperproof Time on Mobile Devices

Using TPM 2.0 Services with Remote Attestation

August 2, 2013

Introduction

Personal computer (PC) tablets, laptops, and servers have a hardware real-time clock (RTC) at their disposal. This RTC is supported by a battery on most systems that keeps the clock running when the computer is off or disconnected. The system administrator or any physically present person may set this clock at any given point in time to any value he desires.

While the RTC is useful for informative purposes, it is useless when it comes to security tasks, since it is not guaranteed that time progression for a given client is unidirectional and uniform. A local GPS clock may address this problem; however, it does not protect from a misconfigured system by the administrator or from a virtualized GPS clock introduced by a malicious attacker. In summary, not even a physically present administrator can ensure that a machine under his control has or remains on the correct time.

When it is assumed that a local system ***does*** have a reliable accurate time source that is administered correctly and protected from attacks, a local user or a remote service does not have any strong proof or guarantee that this situation will persist. A remote service may compare the client time to local time continuously, and since time progresses at equal speed for all observers (let's assume this while we have not yet mastered traveling at light speed), this will allow it to ensure that the client time is accurate. But usually in this situation the client system time is ignored and replaced by the time on the remote service. However, the remote service itself may be under attack.

The service has no reason to assume that the time on the remote system will remain to be accurate, uniform and unidirectional once it disconnects. It has to be assumed that delegated time-based policies that are to be enforced on the remote system may not be honored reliably.

Enterprise managed machines, rights management systems, and time based triggers are therefore without guarantees. This whitepaper discusses how these applications benefit from secure time in a Trusted Platform Module (TPM) 2.0 capable device.

System Requirements

Initialization of time → T_0

Initialization of time is a well understood problem that is solved differently on individual platforms. This goal may be reached by a server on the network with a protocol like NTP or Windows Time or a locally attached hardware clock for example.

For the purpose of this document it is irrelevant *how* the correct time arrives on the local system. It does not even have to be the *correct* time for all observers. While there is a well-defined time normal, there may be a perceived delta between that time normal and the time of a given group of individuals, like computers in a datacenter for example. Within this group it is more important to be on the same shared time even if this time is differing from the absolute time normal. A good example here is an administrative domain of computers: an important task to be executed by the domain controllers is to synchronize the time on the clients to match with the time on the domain controllers.

Keeping time

As discussed above, PC and server platforms have an RTC clock. While modern computer systems use this clock to keep time from boot to boot, during the runtime of the operating system they rely on a different time source. Usually the system is using a timer that ticks with higher granularity than the seconds of the RTC. This is demanded by the fact that, within a second that is tracked by the RTC, millions of instructions are executed on the processors of the system. For example, on a Windows system a time source called file time is employed that is operated at 100 ns.

Traditional 32-bit POSIX time has a granularity of seconds; however, on UNIX systems this time is augmented with a second 32-bit counter that rolls over every second. It depends on the system at what frequency the main timer is operated; however, since the second register rolls over every second, the consumer of this time does not need to know in what intervals this register progresses. Since POSIX time is scheduled to roll over in 2035, modern UNIX systems track it and the fractions of seconds in 64-bit registers. This mitigates the bad side effects of a rollover and provides sufficient time granularity.

At system start time, the RTC is read and converted to a continuous progressing tick counter. This design is not prone to the same errors that the RTC itself is subject to. An example is the switch between summer and winter time, leap days and leap seconds. The RTC clock has to be adjusted for all these occurrences. In contrast, since file time is a simple count of ticks at a given rate from a well-defined epoch (the start time), it does not need to keep track of these events.

Calculating the RTC or human readable local time from the tick counter time is an error prone task that has lead in the past to security attacks on computer systems, to system crashes, and to service outages. To minimize these issues it is recommended to always use well defined library functions for time conversions.

Once the RTC time has seeded the tick counter, the RTC time is largely ignored in the system. Since the RTC keeps time at coarse granularity, from boot to boot the system may lose or gain fractions of seconds. However, since the boot of a system is usually taking more than a second, it is unlikely that a system may experience backward time travel, that is, experience the same second twice. With more systems using the “sleep” processor state, rather than reboot or hibernation, this leads to faster wakeup times, and it is conceivable that a system may accidentally relive the same second. While this may seem like an insignificant issue, the difference is that while the RTC may not be in the control of the regular user, processor sleep

transitions are. So if a user would like to trigger a certain policy twice, and hence undermine the intent of that policy, he could use such an attack to achieve this.

In summary, the challenge is that the coarse and complex RTC is retained, while the desirable high resolution tick counter is lost, on power events.

Remote Trust

While system software may operate in a way that it can protect access to the RTC clock and timer register, this does not provide sufficient assurance to remote services. The good news is that technologies such as measured boot allow a cloud service to measure the integrity of the remote system. That does not preclude all administrator-launched or undetected driver-level attacks, but it's a step in the right direction.

In order to address these issues and allow remote entities to form trust in the local system time, the same procedure must be employed that is used for measured boot: attestation. Using an established attestation key allows a remote party to trust TPM-provided and signed data structures. For an introduction to measured boot and remote attestation, see Hacking Measured Boot and UEFI.

Beyond the requirement of remote attestation, the actual tick counter must operate in a tamper proof environment. There is no additional gain of security if the administrator is free to change the clock immediately before or after an attestation statement is signed. This integrity-protected timer register further has requirements to never roll back and to progress in a monotonic fashion. Surprisingly, the TPM spec does ***not require*** the tick counter to progress while the machine is in a low power state. This is a fundamental difference between the system tick counter and the TPM's. However, this guarantees that every time interval will only be lived through once in sequential order.

The TPM is designed as a cheap low power device with a slow interface. Asking the TPM to operate at a nanosecond resolution or mandating the platform manufacturer to provide offline power would violate its specification. The TPM 2.0 Timer tick operates at 1 ms resolution. This allows the system to provide the finer granularity at a lower trust level but with possible attacks getting progressively harder below a millisecond resolution.

Enforcement needs a hostage

All of the time sources discussed above – the RTC, the tick counter, and even the TPM – are available to an operating system and its applications. An outside authority wants to ensure that these services have access to the correct time. However, to use only platform attestation for that purpose would be difficult, since the enforcement chain from the root of trust to a given application is long and traverses several logical layers where attacks may be launched.

Thus, a policy issued to the local system needs a strong immediate enforcement method. The TPM timer tick register cannot be set back and therefore excludes all attacks where an administrator would attempt to roll the clock back. However, since the TPM tick counter may not increment while in a low power state, a policy specifying an absolute time could still allow the consumer on the TPM to satisfy the policy for the additional time that a machine was in the low power state. This may not be desirable.

Instead, the issuer of a locally binding policy has to formulate policies for the TPM in a time delta format that always specifies a start and end time. This will at least ensure that the policy is not valid for longer than the defined time period; however, it may be interrupted and stalled. Since the TPM is unavailable during low power states, the user will not be able to take advantage of those gaps directly. For example, if the policy is bound to the usage of a key in the TPM, the user will only be able to use that key for the allowed duration.

This excludes low power states, but the user won't be able to use the key during those times anyway, since the TPM is not operational.

Thus, binding time based policy to a TPM object provides the remote party with an enforceable policy.

Application Example

Having discussed general aspects of time based policies, a detailed discussion of how trusted time can be made usable on a system with a TPM 2.0 follows. The discussion includes integration at all logical layers of the platform. It is up to the integrator to select and provide an implementation based on specific requirements.

Platform provided services

The computing platform must provide an execution-protected TPM 2.0 instance. This instance may be physical, firmware, virtualized, or even physically remote in order to guarantee the confidential execution of the device. The selection of the device representation is determined by implementation requirements.

The operational firmware to bootstrap the system must provide extensions described by the Trusted Computing Group. These specifications define a communication channel and a driver model that allow the firmware, and later the operating system (OS) loader, to communicate with the TPM. Further, these specifications define measurements that the firmware and pre-OS must execute to ensure the extension of the root of trust up into the OS.

However, what the client specifications leave out of scope is the task to maintain a reliable time on the device. This is important since the Dictionary Attack (DA) protection uses time to determine if a lockout may be lifted. Thus, a user that triggers the global DA logic may turn the platform off and put it aside to wait out the "punishment," only realizing that after turning on the device after 24 hours that the DA enforcement is still being enforced, since no time has actually passed for the TPM while the platform was turned off. This forces the user to let the platform ***run*** to wait out the DA lockdown. That's unrealistic for mobile, battery operated platforms.

A better implementation would employ a TPM embodiment that may be powered even if the device is in a low power state. A physical or remote TPM will be able to satisfy such a requirement. On the other hand, again, a TPM that is powered has the undesirable potential side effect of draining the battery.

An alternative implementation could use the RTC on the platform to measure the duration that the platform has spent in a low power state and then use the firmware to update the TPM's timer counter while still in the Core Root of Trust for Measurement (CRTM). The measurement would have to be taken after an administrator has the last chance to interact with the RTC and before he can interact with it again when the machine resumes from a low power state. This assumes that the RTC time matched the TPM time on shut down, and that RTC time at resume is sufficient to keep the time of the TPM roughly in the same ballpark, always only allowing an error for the clock to be slow, since the TPM's tick counter cannot be set back.

The procedure in the CRTM portion of the device firmware on resuming from a low power state would be to look for a persisted marker that the platform transitioned stably into a low power state and did not crash or have a forced reboot. The firmware would calculate the milliseconds from the selected epoch based on the RTC time and set the TPM accordingly using physical presence authorization. If no marker was persisted or the integrity of this marker was questionable the code would skip setting the TPM clock to not overshoot the current time.

Alternatively, in firmware that has network access, like PXE or UEFI, the firmware could contact a trusted time server to retrieve a signed time and set it in the TPM. NTP offers such functionality but it is currently not included in PXE or UEFI and would require new functionality and protocol support in the firmware for trusted NTP.

Booting the local client

Once the pre-OS loader has taken over on a typical TCG compliant system, the TPM is fully running and has transitioned out of its physical presence state. This means that the TPM owner has to now authorize setting the clock, which complicates this matter slightly, since the TPM owner credentials are not necessarily available.

While TPM 1.2 offered a credential delegation system, TPM 2.0 does not. The reason for this change is that managing delegations in 1.2 rather complex and consumed expensive internal non-volatile (NV) storage. TPM 2.0 uses policy authorization that may be used to authorize individual commands based on PCR measurements, for example. This mechanism supports the definition of an owner policy that endorses the OS boot manager to set the clock based on its CRTM measurements. In the next extended stage in boot, the boot loader would not have this privilege anymore.

However, the boot manager cannot take a dependency on the RTC. This is because an adversary may have booted the system from a different medium and sidestepped any notion to keep the integrity of the RTC in check, at least directly. The TPM offers a second counter that advances every time the platform emerges from a low power state. The boot manager could store the value of this counter in a secure NV location and compare it the next time it resumes from low power. If the count is contiguous, the boot loader may assume that no other OS has run on the system that may have tampered with the RTC.

This scenario bears the requirement that the legitimate OS on the platform enforces how and when the clock is set and synchronized. Compared to the previous section, this implementation is weaker, but allows the utilization of an off-the-shelf system.

Synchronizing the domain time

Network time synchronization is well established, but relies on the administrator and the system to be trustworthy. These entities may interrupt, alter, and divert time updates at any point. This assumption is fine, as long as only local policies are to be enforced. Once remote policies from a domain are to be enforced, the local administrator and system do not enjoy the highest privilege anymore. To still enforce these policies with the trust of the domain administrator, an isolated enforcement authority has to be present that is outside the control of the local administrator and the system.

The TPM is such an enforcement authority when the policies are attached to TPM objects, like principals, keys, sealed blobs and NV storage. However, TPM operations do not have to originate from the local system. In fact, the TPM does not see a difference between a command that was sent by local administrator and a remote domain administrator.

For example, the local administrator might not be the TPM owner. Instead, the TPM is set up with an owner policy that delegates the privilege to set the clock to a remote time server or an attestation authority. In this scenario, the remote entity starts a sequence of TPM commands, cryptographically protected to prevent playback, to set the clock on the TPM. The session is also going to ensure that the same communication may not be played back to a second TPM for which the time authority may be using the same credential. The OS in this case is acting as a man in the middle that forwards packets from the network to the TPM and back. However, the endpoints can detect if the communication was tampered with or had been delayed.

This approach offers high integrity but comes with the requirement that the platform has to be online to synchronize the time. It offers immediate feedback to the authority that the time on a platform has been successfully set and serves therefore as an attestation mechanism.

Supporting multiple Epochs on one TPM and providing a trusted mechanism to roll the clock back

Thus far, the discussion has assumed that the TPM provides the policy time based on a single epoch, that is, that the TPM tick counter is applied as a delta to a global number. The side effect is that times based on more than one epoch cannot be handled directly. An example is POSIX time and Windows file time. Since both times are based on different origination points the administrator would have to choose the time that he would want to track on the TPM while the other one would be converted in software. That procedure could be subject to attacks.

Further, an administrative error that set all TPM clocks in the domain to a distant time in the future would render all devices unusable, since the time may not be set back. The result would be that these devices would have to be retired or kept off until this point in time had passed.

Instead, we can treat the tick counter as a relative delta to a configured value in the TPM NV storage. Setting the clock consists of a controlled write to a well know NV storage index. The use of multiple offset registers supports the tracking several epochs or differing domain times.

However, policy definitions for TPM objects become more complex since it is no longer sufficient to just look at the tick counter. Now, a comparison on the desired epoch offset in NV storage must also be made. The policy comparison commands for NV storage are the same as they are for the tick counter, so a concatenation of the two must produce the same policy.

This flexibility is bought with a significant increase of complexity for the policy definition. Since complexity is the enemy of security this may allow room for badly defined TPM objects that will not adhere to the intended policy.

Attesting the local time to an authority

As discussed above, there are several scenarios in which it is desirable for a mobile client to be able to assert to a cloud service that the client has a trustworthy, secure time source. This trust can be earned the same way as attestation, based on platform configuration and key properties. The TPM offers an attestation statement signed with a restricted signing key that shows the remote party the current timer tick register. The handshake is performed in a similar manner to other attestations, starting with the authority sending a nonce that is included in the attestation message. In fact, the nonce may be used for multiple attestation messages at once, for example to pool platform and tick counter attestation together.

While platform attestation is only tangible within boundaries, since the platform may be rebooted immediately, time attestation proves the smallest value that this register may ever have which is not subject to reboot attacks. But beyond that the timer information also provides attestation over the reboot and power-up counter. This allows an authority to bind a policy to a crypto object that is usable for the next *n* reboots. This is a powerful tool that provides smart card-like lock-out but without the use of the global DA logic.

The authority can compile all attestation messages from this particular client and convert this into a statement of integrity, or compliance, that the device can present to a remote service. However, the statement of integrity is not yet challengeable by the remote service. This is the subject of the next section.

Policy Enforcement by means of a Hostage

It is difficult for a remote service to ensure that a client device is still in the same state as it was when an attestation took place. The service could issue its own challenges, but that undermines the goal that the service should not require an understanding about how the integrity of the client is guaranteed.

The introduction of a hostage into the protocol is an excellent way to keep the device honest. The hostage may be introduced by the client itself in the form of a key that is generated and bound to the current state of the system. Instead of sending signed attestation statements to the authority, the system provides a key attestation that shows the policies it is bound to. These policies may not be true at the creation time; however, they show that the client configuration and time is within the policy of the domain.

The authority then issues a regular certificate over that key after validating the policy conformance, turning it into the hostage. This model requires no trust of the authority in the client itself, but only in the policy enforcement of the TPM. Hence, there are no attestation messages necessary. If the client is dishonest and creates a key that is not in line with current policy, the client will never be able to use this key on the TPM. Since the authority issues a certificate of compliance over the key, the remote service may now challenge this key without having to understand what the certificate of compliance entails. A signed challenge is sufficient proof by the system that the client is still within the authority-specified policy.

As an optimization to avoid costly key generation on the TPM every time attestation has to take place, the authority can use a client-specific key, bind it for the required policies on every request, and send this key to the client to use. The result is a static compliance certificate that will never change while the client can only use the associated key if he is within policy.

What a time based policy could look like

Assuming that the TPM time is set to a global valid time, the policy associated with a TPM object should specify a start time and expiration time. The enclosed duration determines the lifetime of a policy-bound crypto object. If time synchronization on the client cannot be guaranteed, it is warranted to bind the object also to the current boot counter in order to invalidate access as soon as the device goes into a low power state and loses absolute time accuracy.

The more straightforward usage of a time policy is in a token form that is valid for a specified delta-T. In that case, the TPM will enforce this time, although it can be prolonged relatively by keeping the machine in a low power state in between operations. However, since the machine is unusable in a low power state, the enforcement works exactly the way it is supposed to. An example is a symmetric encryption key protected by the TPM and a data stream protected by the key. Since the data stream can only be consumed when the machine is not in a low power state, it is ensured that the relative availability of the TPM object will limit the consumption of the data stream to the specified policy.

Sample Scenario

To summarize the discussed technologies, a sample scenario follows. The requirements are:

1. All clients have a TPM 2.0
2. The administrative domain has established an Attestation Identity Key (AIK, a restricted signing key that links back to the TPM manufacturer's root of trust) with each client
3. All clients loosely synchronize the TPM time from a timeserver within the administrative domain
4. The administrative domain consists of a Certificate Authority (CA) that can issue compound user/machine certificates. The certificates are used to authenticate to remote services.

5. The administrative domain consists of an Attestation Server with a malware detection agent that inspects machine configurations. Clients on the network have to pass platform attestation for a successful login.
6. Login sessions are valid for a maximum of 8 hours. After that time, clients must re-authenticate with the authenticated domain
7. The authentication remains valid even if the client disconnects from the network or connects to a private network.
8. Clients that resume from a low power state have to redo attestation and acquire a new authentication certificate in order to mitigate the risk of offline attacks.

Next is a description of the authentication procedure that every client executes in order to acquire a valid authentication certificate from the administrative domain:

1. Client boots the OS. TCG compliant measurements are taken. Measurements are extended to one or more TPM Platform Configuration Registers (PCRs).
2. The OS contacts trusted time server to synchronize file time and the TPM tick counter
3. The client prepares for authentication and calculates a TPM policy session using the following concatenated policies:
    a. Current measurements of all relevant PCR registers in the TPM
    b. The current value of TPMS_CLOCK_INFO.restartCount equals TPMS_CLOCK_INFO.restartCount read at policy creation time
    c. The current value of TPMS_CLOCK_INFO.clock is greater or equal than TPMS_CLOCK_INFO.clock read at policy creation time
    d. The current value of TPMS_CLOCK_INFO.clock is less or equal than TPMS_CLOCK_INFO.clock read at policy creation time + 8 * 60 * 60 * 1000
    e. Key authentication value has to be provided at use time
4. Client creates an asymmetric signing key (RSA or ECC) bound to the calculated policy value. If the user desires two-factor authentication, he may associate a password with the key.
5. Client collects the TCG log of all relevant PCRs and reads TPMS_CLOCK_INFO structure from the TPM
6. Client creates a key attestation over the new key, signed with the trusted AIK
7. Client creates a certificate request and signs it with the new key as proof of possession. The X.509 request contains the following additional data:
    a. TCG Log
    b. TPMS_CLOCK_INFO structure
    c. Public Key data structure
    d. Policy sequence information
    e. Key attestation with signature
8. Certificate request is sent to administrative CA

The CA receives the request and forwards the TPM specific parts to the attestation server for validation. The attestation server processes the following:

1. Inspect the TCG log to see if platform boot was executed in a policy compliant manner. If malware or outdated boot drivers are detected, the attestation server may withhold access and force the client to clean or upgrade.
2. Verify that the TPMS_CLOCK_INFO.restartCount matches with the provided TPMS_CLOCK_INFO structure. Once the platform enters a low power state this counter will increment and the key will become inaccessible.

3. Verify that policy {c.} and {d.}, above, is eight hours apart. Use the provided TPMS_CLOCK_INFO.clock to calculate policies {b., c., d.}. The fact that the certificate request has to be self-signed to be accepted by the CA proves that the key is already usable, so the client TPM tick counter must be within the eight hour usage period.
4. Ensure that if an authorization value is set on the key, it will be enforced. This will ensure that the system will not fake two-factor authentication and then use the key behind the users back on the system
5. Verify the key attestation with the trusted AIK, tying everything together

The attestation server reports the public key from the key attestation back to the CA in order to verify that it matches the certificate request. There may be several different strength of attestation or key validity periods configured, allowing the CA to include different assurances in the certificate that will be issued. For example, if the attestation server detects that the client's software is out of date, it can issue a certificate that will allow the client to get updates, but not access confidential services. The CA sends the issued certificate back to the client.

The client uses the private key to sign challenges and provide the certificate as endorsement. In order to sign with the key in the TPM, the client has to play back the same policy that was executed at key creation time. The TPM will use the current values instead of the provided ones. If the client has lied about any of the policy values, the TPM will prevent the use of the private key, rendering the certificate useless.

The key becomes unavailable when the machine reboots, the PCRs change, the machine goes to sleep or hibernation, or when the validity period expires. These policies are enforced by the TPM. Remote services can share sensitive data with certified clients with high assurance.

Conclusion

The time features of the TPM are currently not widely in use, since their usage is not straightforward, and the design of policies is more complex than platform measurements alone. However, the use of time based policies opens new opportunities that allow the implementation of self-expiring tokens and usage tracking that cannot be enforced any other way. This includes mitigating the risk of a local administrator attack even if the machine goes offline after the policy has been issued to the system.

Issuing a TPM object that only stays available for the next *n* reboots, and that gets reissued on successful authentication, is useful in network logon scenarios to enforce lockout. While a physical attacker may revert all disk-based state of the OS, he may not circumvent the revocation of a TPM-bound key.

Compliance certificates can be given a client hardware-enforced time horizon after which the need to be renewed.

Regarding server-side scenarios, PCR measurements on a system with months of uptime eventually become of little value, since the system may have launched numerous software components, changed configuration, and in general expanded its Trusted Computing Base (TCB). For such a system, it is important to monitor runtime antivirus protection. However, as a notable improvement, the runtime protection can utilize time bound keys to enforce regular database updates and ensure that it remains connected with the outside world.

Even for local purposes, TPM time is a powerful enforcement tool that may be used to provide temporal and integrity order in an event log, for example. This helps protect intrusion records on a server, where the last step of every attacker is to hide the intrusion.

While trusted tamperproof time is powerful, integration is a highly custom process that is driven by the requirements and the abilities of the platform and the services. With the right experience and understanding of the problem set, solutions on regular devices can be created that are currently unrivaled.

To request a demonstration of the use of trusted tamperproof time in protecting sensitive data, or to inquire about the integration of TPM bound keys with line of business services and software, please contact JW Secure.

About JW Secure

Founded in 2006, JW Secure specializes in custom security software development. Our customers include Alstom Grid, DARPA, Lockheed Martin, and Microsoft. To learn more, please visit our website.

We know security technology, so you know who, when and why.