

CPU to TPM Bus Protection Guidance – Passive Attack Mitigations

Version 1.0

Revision 12

May 4, 2023

Contact: admin@trustedcomputinggroup.org

This document is an intermediate draft for comment only and is subject to change without notice. Readers should not design products based on this document.

PUBLIC REVIEW

DISCLAIMERS, NOTICES, AND LICENSE TERMS

THIS DOCUMENT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, DOCUMENT OR SAMPLE.

Without limitation, TCG disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this document and to the implementation of this document, and TCG disclaims all liability for cost of procurement of substitute goods or services, lost profits, loss of use, loss of data or any incidental, consequential, direct, indirect, or special damages, whether under contract, tort, warranty or otherwise, arising in any way out of use or reliance upon this document or any information herein.

This document is copyrighted by Trusted Computing Group (TCG), and no license, express or implied, is granted herein other than as follows: You may not copy or reproduce the document or distribute it to others without written permission from TCG, except that you may freely do so for the purposes of (a) examining or implementing TCG documents or (b) developing, testing, or promoting information technology standards and best practices, so long as you distribute the document with these disclaimers, notices, and license terms.

Contact the Trusted Computing Group at www.trustedcomputinggroup.org for information on document licensing through membership agreements.

Any marks and brands contained herein are the property of their respective owners.

DRAFT

CHANGE HISTORY

REVISION	DATE	DESCRIPTION
1.0/12.00	May 4, 2023	First version for public review

DRAFT

CONTENTS

DISCLAIMERS, NOTICES, AND LICENSE TERMS 1

CHANGE HISTORY 2

Contributors..... 4

1 SCOPE 5

 1.1 Purpose..... 5

 1.2 Introduction 5

 1.3 Guidance summary 6

2 References 7

3 Problem statement..... 8

4 Using parameter encryption to protect bus communication 9

 4.1 Parameter encryption background 9

 4.1.1 Session key creation 9

 4.1.2 Choosing a session type 10

 4.1.3 Enabling encryption and decryption on commands..... 11

 4.2 High level workflow description 11

5 Example workflow – sample scripts 13

 5.1 Read/write NV index using an encrypted session (EK salted session) 13

 5.1.1 tpm2-software utilities example..... 13

 5.1.2 ibmtpm20tss utilities example 14

 5.2 Create password protected key using an encrypted session (bound session) 14

 5.2.1 tpm2-software utilities example..... 15

 5.2.2 ibmtpm20tss utilities example 16

Contributors

TCG would like to thank the following contributors to this document.

Amy Nelson, Dell
Boris Balacheff, HP Inc.
Federico Morini, HP Inc.
Joshua Schiffman, HP Inc.
Adrian Shaw, HP Inc.
Ken Goldman, IBM
Andreas Fuchs, Infineon
Jiewen Yao, Intel Corporation
Dan Morav, Nuvoton

DRAFT

1 SCOPE

This document provides just essential guidance for developers of TPM-based solutions to defend against passive attacks on CPU-TPM communications using parameter encryption capabilities defined in the TCG TPM 2.0 Library Specification Part 1 [2]. It explains and outlines how software developers should enable and use parameter encryption capabilities to provide a general defense against passive sniffing attacks that could take place on the communication path, including the hardware bus between a chipset and a discrete TPM. Example code is presented to illustrate how the solution could be implemented by a calling application.

1.1 Purpose

This document is a practical guide for developers and system architects to understand how their calling code or entity can protect against passive attacks and choose reasonable solutions proportionate to the threat. It does not aim to provide guidance on addressing active attacks, such as hardware man-in-the-middle (MITM) or more elaborate threats on TPM interactions, which would require a broader architectural approach.

1.2 Introduction

Developers frequently use a TPM to store, generate, and retrieve sensitive information such as keys and passwords. A TPM provides a session-based parameter encryption mechanism to ensure the confidentiality of values in transit. This is defined in TPM 2.0 Library specification Part 1 under the section, “Session-based encryption” [2]. Parameter encryption provides a good baseline defense against attackers that can passively intercept traffic between the calling code and the TPM, such as probing attacks on the CPU-TPM bus. However, parameter encryption is often under-utilized, perhaps due to a lack of explicit guidance. This guide provides a general overview of the feature, how to configure it, and examples of its use in different scenarios.

Parameter encryption is not available for all commands and their contents. Only the first parameter of a command or response can be encrypted, and even then, only if that parameter has an explicit size parameter. The TPM architecture orders parameters in commands and responses so that the first parameters are always the parameters that might benefit from encryption. While session encryption is configured at session establishment, it must be enabled by setting an attribute on each eligible command. Callers using higher level abstractions such as certain TPM Software Stack (TSS) APIs may not need to do this explicitly, as it is handled for them [3]. For lower-level interfaces, callers should be careful to set these attributes themselves.

Parameter encryption is based on using a shared secret with either AES-based encryption or KDF-based XOR obfuscation. The TPM architecture offers XOR obfuscation because some callers might be so resource-restricted that they do not have encryption algorithms and obfuscation may be acceptable. Both AES and XOR modes are described in the TPM 2.0 Library Specification Part 1 in the sections “XOR Parameter Obfuscation” and “CFB Mode Parameter Encryption” [2]. If the caller is capable of encryption, then the caller should always use AES where possible.

Performance is the same regardless of how the shared secret is established. There are three ways to establish the shared secret based on the type of session. The following is a short description of each session type and their tradeoffs.

1. **Unbound Session:** The shared secret is derived from the authorization value of the TPM entity being authorized. The key creation process has good performance since it is based on hashing. It requires no other entity to establish the secret, but the secret’s strength is based on the entropy of the authorization value. Additionally, the caller must supply the authorization each time the session is used.
2. **Bound Session:** The secret is created from the authorization value of a bind entity that is “bound” to the session. Unlike the unbound session, the TPM keeps track of the bind entity’s authorization value, so the caller

only needs to provide it once. If the bind entity is different from the authorized entity, then the authorization value for the authorized entity can be weak provided the bind entity's value is strong. A bound session also has good performance due to hashing based key generation. The caller is required to have previously established a shared secret to use the bind entity.

3. **Salted Session:** The secret is a salt encrypted to an asymmetric key loaded in the TPM. Since bound sessions require a pre-shared secret between the caller and the TPM, salted sessions are often used to establish such a shared secret because they do not require a shared secret. However, the performance is slower than bound sessions as it requires asymmetric key operations. Callers that do not have asymmetric cryptography capabilities cannot use this type of session. Moreover, a trusted asymmetric key must be loaded before use.

1.3 Guidance summary

In addition to technical tutorial and examples, this document provides guidance on why, how, and when to use parameter encryption. This subsection summarizes that guidance as follows:

- 1) Enable parameter encryption whenever sensitive data is passed to or from the TPM. This will provide confidentiality, integrity, and authentication for the data in transit.
- 2) Explicitly enable encryption on a per command and response basis, or use a TSS API that manages encryption on behalf of the caller, such as a FAPI compliant TSS [3].
- 3) Choose session parameters that create a strong *sessionKey*.
 - a. If there already exists a session with a strong *sessionKey*, then reuse that session. Even a policy session can be reused, and the session key persists through TPM2_PolicyRestart.
 - b. Else if the authorized entity has a strong *authValue*, then use that entity in an authorization session.
 - c. Else if the authorized entity has a weak *authValue*, use a bound session with a bind entity that has a strong authorization value.
 - d. Else, choose a Salted session with a trusted asymmetric key loaded in the TPM (e.g., an Endorsement Key with a verified Endorsement Credential). Salted sessions can be used to provision shared secrets so the caller can use bound sessions in the future, which will lower overhead.

2 References

- [1] Trusted Computing Group, "Trusted Platform Module Library Family 2.0 Level 00 Revision 01.59," [Online]. Available: <https://trustedcomputinggroup.org/resource/tpm-library-specification/>.
- [2] Trusted Computing Group, ""Trusted Platform Module Library Part 1: Architecture"," [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part1_Architecture_pub.pdf.
- [3] Trusted Computing Group, "TCG Feature API (FAPI) Specfication," [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TSS_FAPI_v0p94_r09_pub.pdf.
- [4] Trusted Computing Group, "TCG TSS 2.0 Enhanced System API (ESAPI) Specification," [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TSS_ESAPI_v1p0_r14_pub10012021.pdf.
- [5] Trusted Computing Group, "TCG TSS 2.0 System Level API (SAPI)," [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TSS_SAPI_v1p1_r36_pub10012021.pdf.
- [6] Trusted Computing Group, "TCG TPM v2.0 Provisioning Guidance," 17 March 2017. [Online]. Available: <https://trustedcomputinggroup.org/wp-content/uploads/TCG-TPM-v2.0-Provisioning-Guidance-Published-v1r1.pdf>.
- [7] Trusted Computing Group, "Registry of Reserved TPM 2.0 Version 1.1 Revision 1.00," 6 February 2019. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/RegistryOfReservedTPM2HandlesAndLocalities_v1p1_pub.pdf.

3 Problem statement

Software often interacts with a TPM to send and receive sensitive data such as cryptographic keys or authorization values. While the host OS and low-level firmware logically protect this communication from malicious software, physical attackers with access to the TPM bus may be able to capture these secrets by *passively* listening to the messages.

- 1) **Example 1:** An attacker attaches a physical probe to the TPM's bus to passively observe communication. This is not necessarily a destructive attack, as some buses are easily accessed.
- 2) **Example 2:** CPU-based confidential computing technologies can be used to execute software in an environment where code and data are encrypted to prevent malicious guest or supervisor code from introspecting code execution. However, host software still controls interactions with a TPM. In this scenario, malicious software could perform passive snooping attacks to capture plaintext secrets sent to and from the TPM.

Because passive attacks typically require less effort to mount than active attacks, they can be performed quickly or setup ahead of time to listen serendipitously for sensitive data such as application secrets, network access credentials, authentication keys, or content held in a TPM's NV locations. Such attacks are particularly serious when the attacker is able to obtain disk encryption secrets exchanged on early boot prior to user login, and can be performed while the user is absent.

This document focuses specifically on protecting against passive sniffing attacks on the communication path between the caller and the TPM. Implementing this guidance provides a strong baseline defense against attacks which could otherwise be stealthy and undetectable. However, addressing bus snooping alone does not prevent more sophisticated attacks such as destructive, active MITM, or TPM desolder-and-move attacks. More resources are required to defend against more sophisticated attacks.

4 Using parameter encryption to protect bus communication

By default, communication between the caller and the TPM is not protected from snooping, as data are presented in plaintext. TPM sessions support parameter encryption for some commands and their responses to protect sensitive data. However, parameter encryption is not enabled by default and such settings are often ignored or not known by developers. This section clarifies how parameter encryption works and provides guidance for software developers to enable it in TPM communications where possible.

A TPM can be used by different parts of the host software, each of which provides different abstractions to the caller. The higher up the software stack, the less effort and knowledge of TPM intricacies the caller needs to use the TPM. This can be summarized into a few distinct caller profiles:

- **Application developers** typically use the high-level TCG Feature API (FAPI) [3]. A compliant TSS implementation of FAPI automatically encrypts commands and responses, and no work is required by application developers.
- **OS service developers** typically use a lower-level API with more features, such as the TSS Enhanced System API (ESAPI) [4]. Developers using this API need to manually set the session properties to support parameter encryption. Some TSS implementations provide additional flags or features to increase convenience and reduce code repetition. Other implementations require explicit enablement of parameter encryption for each supported command.
- **Firmware developers** may have access to a rich API, such as the TSS System API (SAPI) [5]. However, if this is not available then firmware developers may need to program the TPM command buffers directly or use an alternative low-level API. In this scenario, the developer may need to deal with every detail related to session setup, session key establishment, ciphering, and understand which commands are capable of parameter encryption. This document provides guidance on the general steps.

4.1 Parameter encryption background

TPM 2.0 Library Specification Part 1 Sections 19 and 21 describe the TPM's session-based parameter encryption feature [2]. Parameter encryption protects sensitive data such as authorization values and key material. The first parameter in the parameter area of the command or response buffer is encrypted as long as the parameter has an explicit size field, i.e. the parameter is a TPM2B structure.

TPM implementations support XOR obfuscation at a minimum, but AES-CFB encryption may be supported as well. Neither method requires data padding, so the plaintext and ciphertext are the same length.

4.1.1 Session key creation

Session-based parameter encryption is enabled by the creation of a *sessionKey* during session setup. This *sessionKey* is derived in a way that is dependent on the choice of encryption parameters and session type.

When starting a session using `TPM2_StartAuthSession()`, two parameters, *tpmKey* and *bind*, determine the session type and derivation of the *sessionKey*. The *tpmKey* parameter takes a handle to a loaded TPM key used to encrypt a salt. The *bind* parameter takes a handle to a TPM entity with an *authValue* that is added to the *sessionKey* derivation process. Setting just the *tpmKey* parameter creates what is called a **salted session**. Setting just the *bind* parameter creates what is called a **bound session**. Setting both *tpmKey* and *bind* to `TPM_RH_NULL` will result in an **unbound session** with no *sessionKey*. The TPM will only use the *authValue* of the object being accessed in the key generation.

The caller should ensure a strong secret is used for encryption by using an authorized entity with a high entropy *authValue*, a bind entity with a high entropy *authValue*, or a salted session.

Table 1 Comparison of session types

tpmKey	Bind	Session type	Comments
TPM_RH_NULL	TPM_RH_NULL	Unbound session	The strength of protection is proportional to the entropy in the <i>authValue</i> of the object
TPM_RH_NULL	TPM entity (same as authorized entity)	Bound session	Advantages <ul style="list-style-type: none"> • Good performance, based on hashing. • Bind entity <i>authValue</i> need only be supplied once.
TPM_RH_NULL	TPM entity (different from authorized entity)	Bound session	Advantages <ul style="list-style-type: none"> • Good performance, based on hashing. • Bind entity <i>authValue</i> need only be supplied once. • Authorized entity can have weak <i>authValue</i> if bind entity's <i>authValue</i> is strong. Disadvantages <ul style="list-style-type: none"> • Requires caller to have shared secret to use bind entity. • Another entity with a strong authorization value must be used.
TPM key	TPM_RH_NULL	Salted session	Advantages <ul style="list-style-type: none"> • Caller does not need shared secret for TPM key. • Can be used to establish first shared secret of a bind entity. Disadvantages <ul style="list-style-type: none"> • Slower setup due to the asymmetric key operation. • A trusted asymmetric key must be loaded.
TPM key	TPM entity	Salted and bound session	Not recommended because it has the advantages of neither and the disadvantages of both.

4.1.2 Choosing a session type

The choice of session type is a trade-off between resources and performance, but depends upon the level of confidence that is required:

- In a **salted session**, the *sessionKey* is derived by encrypting a salt with an asymmetric key, preferably with the fixedTPM flag set, such as a key derived from one of the Primary Seeds (Endorsement, Null, etc). This session type is particularly useful for provisioning secrets into the TPM when there are no prior shared secrets established between the caller and TPM. The caller can simply use one of the trusted keys loaded and available to them, such as the Endorsement Key. However, salted session creation is expensive relative to bound sessions due to the use of asymmetric cryptography on the salt. Thus, it is more suitable for infrequent interactions with the TPM such as during provisioning or when unsealing a disk encryption secret during boot. The caller does not need secure access to a host secret to create a salted session.
- A **bound session** by contrast derives the *sessionKey* from the *authValue* of the bind entity. Bound sessions are preferable to salted sessions if performance is critical. For example, long running applications that

frequently use the TPM to sign, encrypt or decrypt data will have lower setup overhead with a bound session than a salted session. However, a bound session relies on having already provisioned an object with an *authValue* within the TPM. The caller also needs to have secure storage or access to the *authValue*. Bound sessions also eliminate the need to provide the bind entity's *authValue* multiple times. If the bind entity is not the same as the authorized entity, the bind entity will increase the *sessionKey*'s strength if the authorized object has a weaker *authValue* than the bind entity.

While outside the scope of this document, a caller concerned with which TPM is responding to the session should use a salted session with a *tpmKey* parameter that comes from a trusted source or that has been authenticated, such as a verified Endorsement Key (EK) or attestation key. For more guidance on validating an EK, see TCG TPM v2.0 Provisioning Guidance [6].

4.1.3 Enabling encryption and decryption on commands

After session setup, the caller enables parameter encryption on eligible commands. The caller does this by setting the *sessionAttributes.decrypt* and / or *sessionAttributes.encrypt* attributes to encrypt the first parameter of the command and response buffers respectively. To be eligible, commands must have an explicit size field for the first parameter and then only the data portion will be encrypted. In other words, the parameter must have a TPM2B structure.

When using low level APIs and commands, parameter encryption is not enabled by default. Care should be taken to enable it for each command. When using FAPI [3], all communication with the TPM is automatically performed in salted HMAC sessions and parameter encryption is enabled wherever applicable. When using ESAPI, the caller can choose to always set the flags since they are ignored by the TSS if the command does not support encryption.

4.2 High level workflow description

This section describes high-level flows for enabling parameter encryption on salted sessions and bound sessions.

In the example of a salted session, illustrated in Figure 1, the TPM Endorsement Key is used as the *tpmKey* and created as a precondition of this flow.

1. Retrieve the TPM's Endorsement Key (EK) handle [7].
2. Start a session with the TPM using the TPM2_StartAuthSession command.
 - 2.1 Pass in the EK handle as the *tpmKey* and specify that the session should be a HMAC session.
 - 2.2 Specify the algorithm of parameter encryption/decryption symmetric-key to AES-CFB. On success, the command returns a handle for the session.
3. Use the session handle with compatible commands, such as TPM2_Create, TPM2_NV_Read, TPM2_NV_Write, TPM2_NV_Extend, TPM2_Unseal and more.
 - 3.1 For each command, specify the use of parameter encryption.

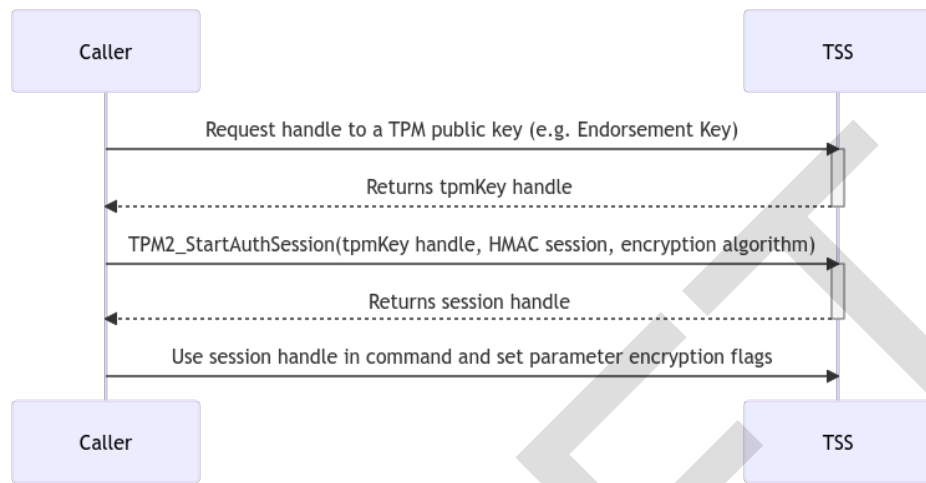


Figure 1, High level example of a salted session.

In the example of a bound session illustrated in Figure 2, an existing TPM key is used for *bind* and created as a precondition of this flow.

1. Start a session with the TPM using the TPM2_StartAuthSession command.
2. Pass in the handle of an existing TPM object (*bind* handle) and the *authValue* for that object (*bind* auth).
 - a. Specify that the session should be an HMAC session.
 - b. Specify the algorithm of parameter encryption/decryption symmetric-key to AES-CFB.
3. Use the session handle with compatible commands, such as TPM2_Create, TPM2_NV_Read, TPM2_NV_Write, TPM2_NV_Extend, TPM2_Unseal and more.
 - a. For each command, specify the use of parameter encryption.

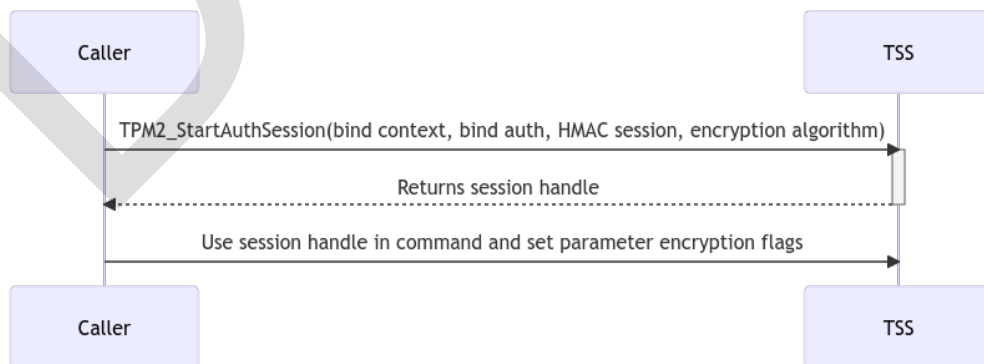


Figure 2, High level example of a bound session.

5 Example workflow – sample scripts

This section describes examples that use parameter encryption, including nvwrite, nvread and key creation. Each example is shown using utilities from two different TSS stacks:

- For the tpm2-software utilities, also known as tpm2-tools, parameter encryption is specified after a session is established using tpm2_sessionconfig with the **–enable-encrypt** and **–enable-decrypt** flags. Examples are shown in sections 5.1.1 and 5.2.1.
- For the ibmtpm20tss utilities, parameter encryption must be specified in each command along with the handle to the authenticated session. The parameter **20** specifies command parameter decryption and the parameter **40** specifies response parameter encryption. Examples are shown in sections 5.1.2 and 5.2.2.

5.1 Read/write NV index using an encrypted session (EK salted session)

Figure 3 illustrates how to safely create, read and write to an NV index by establishing a salted session with the EK and using the shared session secret to encrypt the commands and responses.

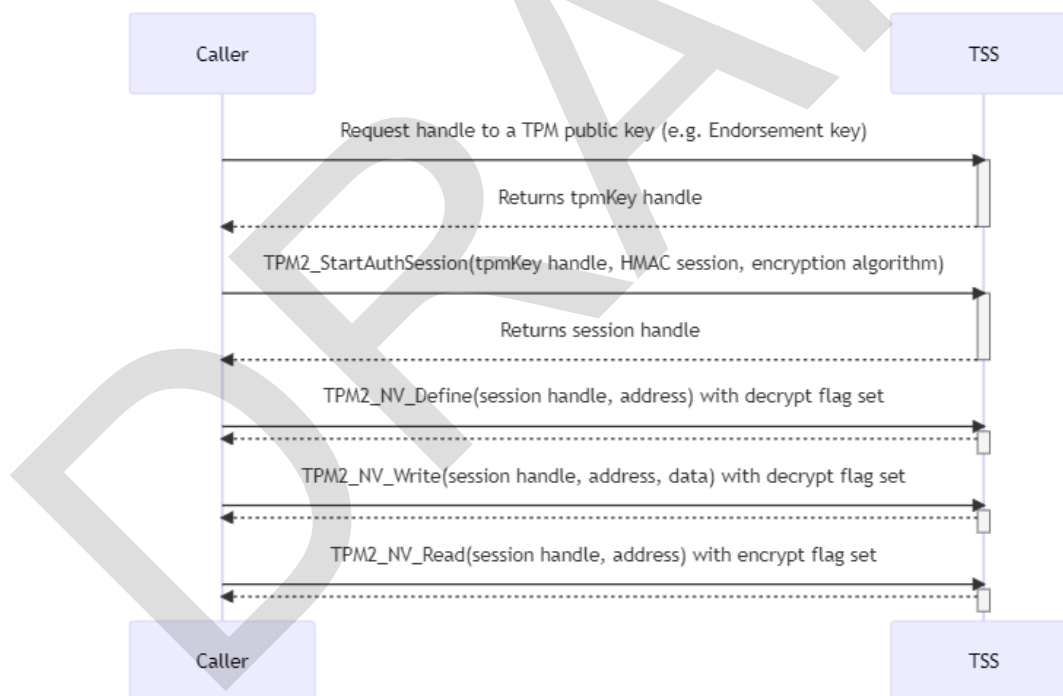


Figure 3, Salted session sequence diagram for example code.

5.1.1 tpm2-software utilities example

```
#!/bin/bash
set -xe # this prints each command to the console and terminates on error
export TSS2_LOG=all+NONE # configurable log level

# Retrieve public Endorsement Key and its handle/context
tpm2_createek --quiet --key-algorithm rsa --ek-context ek.ctx

# Create salted session using the Endorsement Key and enable parameter encryption
tpm2_startauthsession --session salted_session.ctx --hmac-session --tpmkey-context ek.ctx
tpm2_sessionconfig salted_session.ctx --enable-decrypt --enable-encrypt

# Example: Store a secret in a TPM NV index using an encrypted session. 'o' is short for 'owner'
echo mysecret > mysecret
tpm2_nvundefine --quiet 0x1500016 || true # Make sure this NVRAM slot is not used

tpm2_nvdefine 0x1500016 --session=salted_session.ctx --hierarchy=o --size=32 --index-auth=pwd1
tpm2_nvwrite 0x1500016 --input=mysecret --auth=session:salted_session.ctx+pwd1

# Example: Read a secret from a TPM NV index using an encrypted session
SECRET=$(tpm2_nvread 0x1500016 --auth=session:salted_session.ctx+pwd1)
echo "Reading secret: $SECRET"
```

5.1.2 ibmtpm20tss utilities example

```
#!/bin/bash
set -xe # this prints each command to the console and terminates on error
export TPM_ENCRYPT_SESSIONS=0 # Session state is locally saved in plaintext

# Retrieve public Endorsement Key and its handle/context
tsscreateek -cp -noflush -rsa 2048 # Retrieve public Endorsement Key and get handle 80000000

# Create salted session using the Endorsement Key
tssstartauthsession -se h -hs 80000000 # this returns handle to salted HMAC session at 02000000

# Example: Store a secret in a TPM NV index using the encrypted session.
echo mysecret > mysecret
tssnvundefinespace -hi o -ha 0x1500016 || true # Remove existing NV index
tssnvdefinespace -hi o -ha 0x1500016 -sz 128 # Create NV index. 'o' is short for 'owner'
tssnvwrite -ic mysecret -ha 0x1500016 -se0 02000000 20 # Write index using command parameter encryption (20)

# Example: Read a secret from a TPM NV index using the encrypted session
tssstartauthsession -se h -hs 80000000 # this returns handle to salted session at 02000000
tssnvread -ha 0x1500016 -sz 8 -se0 02000000 40 # Read index using response parameter encryption (40)
```

5.2 Create password protected key using an encrypted session (bound session)

The example illustrated in Figure 4 starts with a provisioning phase where the caller creates a primary object in the TPM (with a password) using an EK salted session. Then, the example illustrates a bound session using the object whereby the session secret used for parameter encryption is derived from the password.

The salted session is used to encrypt pwd1, the primary key's password. Afterwards, the bind session is used to encrypt pwd2.

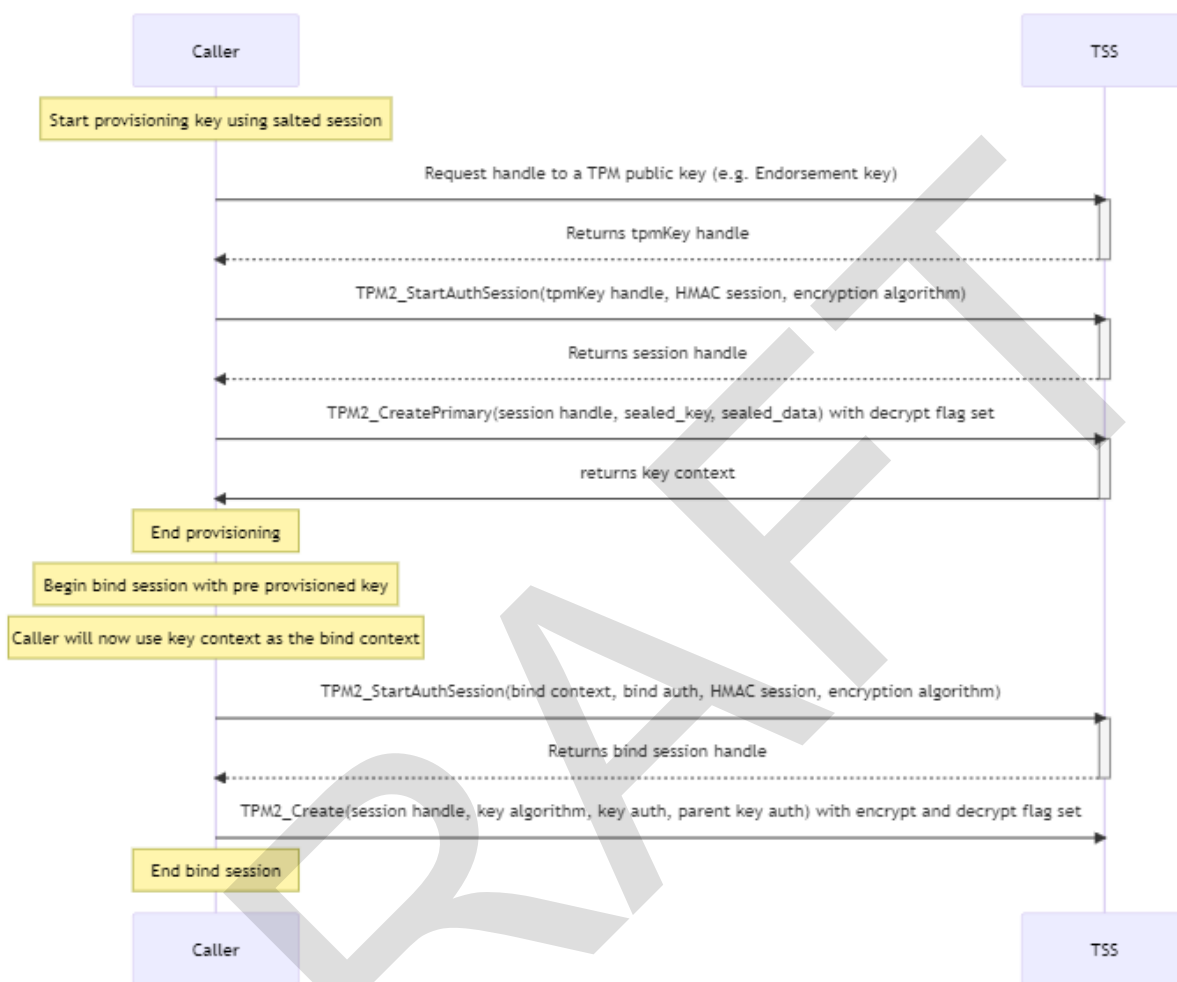


Figure 4, Bound session sequence diagram for example code.

5.2.1 tpm2-software utilities example

It should be noted that, as of writing, this tpm2_createprimary utility does not yet support sessions or parameter encryption.

```

#!/bin/bash
set -xe # this prints each command to the console and terminates on error
export TSS2_LOG=all+NONE # configurable log level

# Beforehand: Provision a primary key into the TPM using an EK salted HMAC session
tpm2_createek --quiet --key-algorithm rsa --ek-context ek.ctx
tpm2_startauthsession --session salted_session.ctx --hmac-session --tpmkey-context ek.ctx
tpm2_sessionconfig salted_session.ctx --enable-decrypt --enable-encrypt
tpm2_createprimary --key-algorithm=ecc --key-context=primary.ctx --key-auth=pwd1
tpm2_flushcontext salted_session.ctx

# At runtime: Start a bind auth session against the provisioned primary key handle
  
```



```
tpm2_startauthsession --session bind_session.ctx --bind-context=primary.ctx --bind-auth=pwd1 --hmac-session
tpm2_sessionconfig bind_session.ctx --enable-decrypt --enable-encrypt # enable parameter encryption

# Example: Create a key using the bind session 02000000
# and encrypt passwords using parameter encryption
tpm2_create --session=bind_session.ctx --key-algorithm=ecc256 --key-auth=pwd2 -C primary.ctx --parent-auth=pwd1
```

5.2.2 ibmtpm20tss utilities example

```
#!/bin/bash
set -xe # this prints each command to the console and terminates on error
export TPM_ENCRYPT_SESSIONS=0 # Session state is locally saved in plaintext

# Beforehand: Provision a primary key into the TPM using an EK salted session
tsscreateek -cp -noflush -rsa 2048 > /dev/null 2>&1
tssstartauthsession -se h -hs 80000000 # this returns a handle to the salted session at 02000000
tsscreateprimary -st -ecc nistp256 -pwdk pwd1 -se0 02000000 20 # create key and use parameter encryption

# At runtime: Start a bind auth session against primary key handle 80000001 with password "pwd1"
tssstartauthsession -se h -bi 80000001 -pwdb pwd1

# Example: Create a new key using the bind session 02000000
# and encrypt its password "pwd2" using parameter encryption (20)
tsscreateprimary -si -ecc nistp256 -pwdk pwd2 -se0 02000000 20
```