Quarkslab

Vulnerabilities in the TPM 2.0 Reference Implementation Code

Pass The Salt 2023

Francisco Falcón / @fdfalcon

Whoami

- I'm Francisco Falcón, from Argentina.
- Reverse engineer, security researcher at Quarkslab since 2016.
- Formerly: Exploit writer at Core Security.
- Interested in the usual low-level stuff: reverse engineering, vulnerability research, exploitation...
- **@fdfalcon** on

Motivation

Why doing security research on TPMs?

1. Virtualized TPMs offer a little explored path for VM escape on virtualization software.
    1. This is also true for cloud environments!

Motivation

Why doing security research on TPMs?

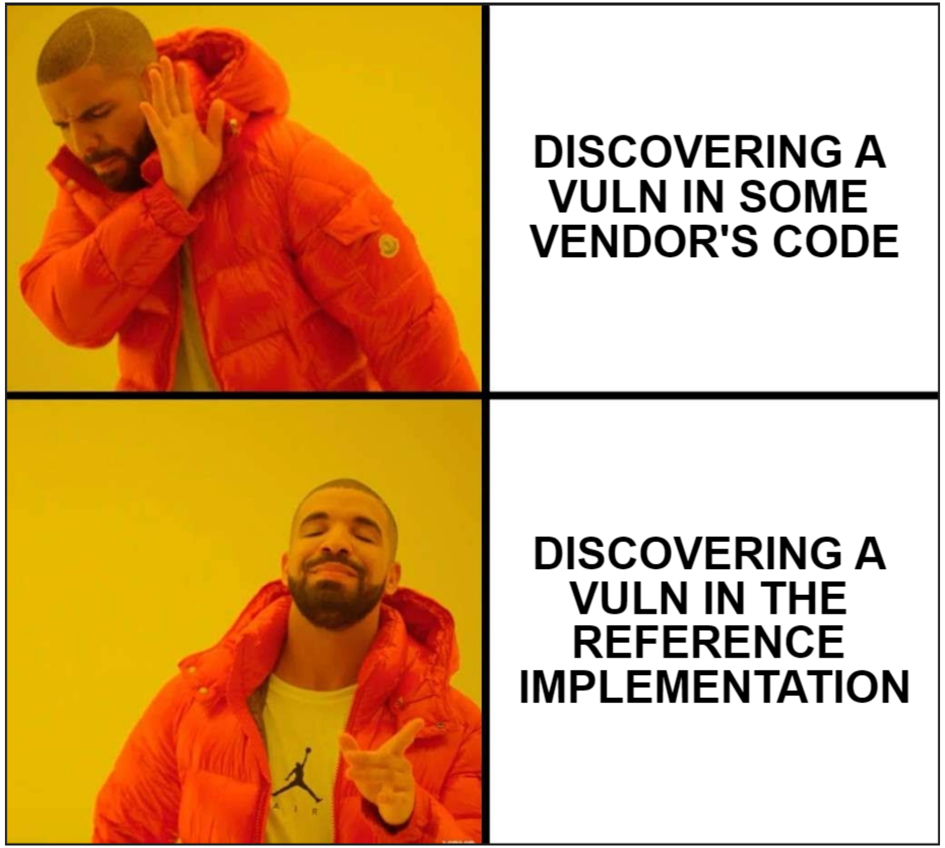
1. Virtualized TPMs offer a little explored path for VM escape on virtualization software.
    1. This is also true for cloud environments!
2. TPM firmware runs on a separate processor ⟶ whatever happens there, it's not observable from the main CPU. If you get RCE on it, it may be hard to detect.

Motivation

Why doing security research on TPMs?

1. Virtualized TPMs offer a little explored path for VM escape on virtualization software.
    1. This is also true for cloud environments!
2. TPM firmware runs on a separate processor ⟶ whatever happens there, it's not observable from the main CPU. If you get RCE on it, it may be hard to detect.
3. The underlying protocol is complex, and the code parsing it is written in C.

4 . Widely adopted reference implementation ⟶ a vuln in the reference implementation code ends up affecting everyone.


DISCOVERING A VULN IN SOME VENDOR'S CODE
DISCOVERING A VULN IN THE REFERENCE IMPLEMENTATION

Agenda

1. TPM basics
2. Virtual TPMs
3. TPM 2.0 protocol internals
4. Vulnerabilities: CVE-2023-1017 and CVE-2023-1018
5. Disclosure details
6. Conclusions

Part 1

TPM Basics

Trusted Platform Module (TPM)

A **standard** secure crypto-processor designed to perform cryptographic operations:

- Generation and storage of cryptographic keys
- Symmetric and asymmetric encryption/decryption
- Digital signatures generation/verification
- Random number generation

Trusted Platform Module (TPM)

Typical use cases:

- Attestation of the boot process integrity

> See Nicolas' talk up next for details!

Trusted Platform Module (TPM)

Typical use cases:

- Attestation of the boot process integrity

> See Nicolas' talk up next for details!

- Storage of disk encryption keys (e.g Bitlocker)

Trusted Platform Module (TPM)

Typical use cases:

- Attestation of the boot process integrity

> See Nicolas' talk up next for details!

- Storage of disk encryption keys (e.g Bitlocker)
- Digital rights management

TPM Flavors

- Integrated TPMs
  - Dedicated hardware integrated into one or more semiconductor packages alongside, but logically separate from, other components.

TPM Flavors

- Integrated TPMs
  - Dedicated hardware integrated into one or more semiconductor packages alongside, but logically separate from, other components.
- Discrete TPMs
  - Separate component in its own semiconductor package.

TPM Flavors

- Integrated TPMs
  - Dedicated hardware integrated into one or more semiconductor packages alongside, but logically separate from, other components.
- Discrete TPMs
  - Separate component in its own semiconductor package.
- Virtual TPMs
  - QEMU, VirtualBox, VMware, Hyper-V, Parallels Desktop...

TPM Flavors

- Integrated TPMs
  - Dedicated hardware integrated into one or more semiconductor packages alongside, but logically separate from, other components.
- Discrete TPMs
  - Separate component in its own semiconductor package.
- Virtual TPMs
  - QEMU, VirtualBox, VMware, Hyper-V, Parallels Desktop...
- Firmware-based TPMs
  - Run the TPM in firmware in a Trusted Execution mode of a general purpose computation unit.
  - Intel Platform Trust Technology (PTT)
    - Based on Intel Converged Security & Management Engine (CSME), runs in the Platform Controller Hub (PCH)
  - AMD fTPM

TPMs on the Cloud

All the major cloud computing providers offer instances with virtual TPMs:

- Amazon AWS has **NitroTPM**
- Microsoft Azure provides virtual TPMs as part of **Trusted Launch**
- Google Cloud offers virtual TPMs as part of **Shielded VMs**
- Oracle Cloud Infrastructure provides virtual TPMs as part of **Shielded Instances**

Part 1.2

The TPM 2.0 Reference Implementation

TPM 2.0 Reference Implementation

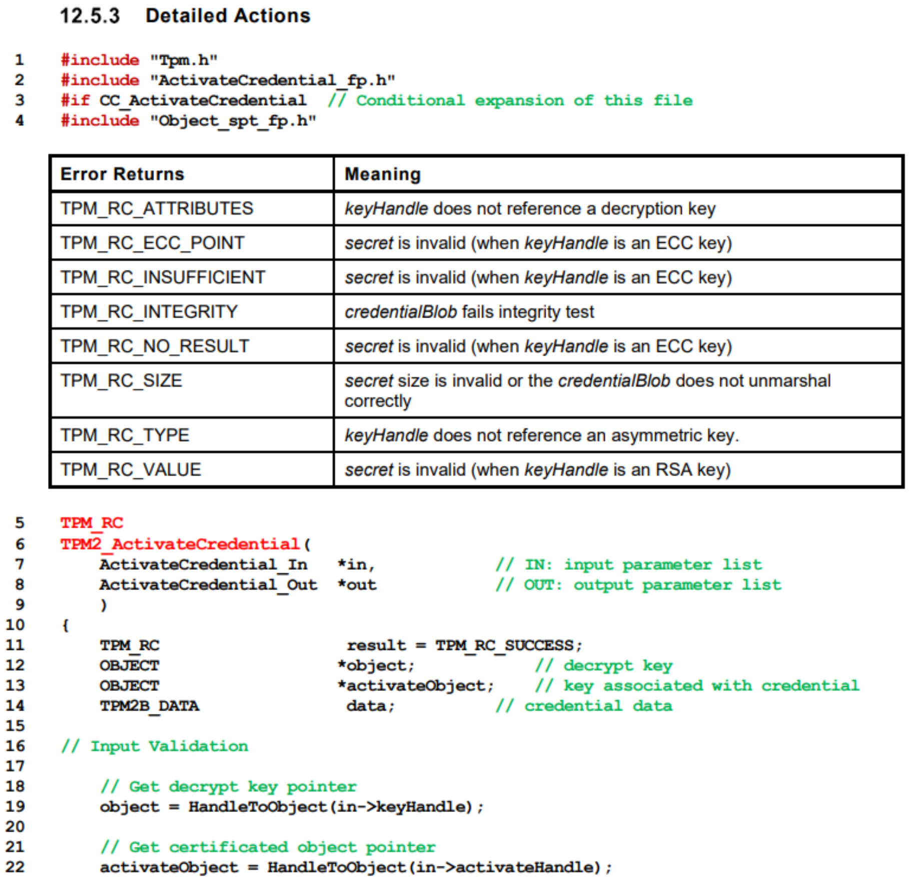
- The TPM standard is published and maintained by the Trusted Computing Group (TCG), a nonprofit organization.
  - They publish the reference implementation code for the firmware of TPMs
  - Adopted by (almost?) all vendors: hardware/firmware/virtual/cloud TPMs...
- Old standard: TPM 1.2
  - Only allows for the use of RSA for key generation
  - Only allows for the use of SHA1 as hashing function
  - Deprecated
- Current standard: TPM 2.0

TPM 2.0 Reference Implementation

- Latest version: Trusted Platform Module Library Specification, Family "2.0", Level 00, Revision 01.59 – November 2019
- 6 PDF documents, accounting for 2568 pages:
  - Part 1: Architecture (306 pages)
  - Part 2: Structures (177 pages)
  - Part 3: Commands (432 pages)
  - Part 3: Commands - Code (498 pages)
  - Part 4: Supporting Routines (146 pages)
  - Part 4: Supporting Routines - Code (1009 pages)

1.2 The TPM 2.0 Reference Implementation

- C code is embedded in the PDF documents (no TCG source code repository)
  - Intertwined with descriptions, section names, line numbers, tables...
  - Microsoft extracts the code from the PDF files and keeps a repository on Github
  - IBM keeps a repository on Sourceforge



12.5.3 Detailed Actions

```c
#include "Tpm.h"
#include "ActivateCredential_fp.h"
#if CC_ActivateCredential  // Conditional expansion of this file
#include "Object_spt_fp.h"
```

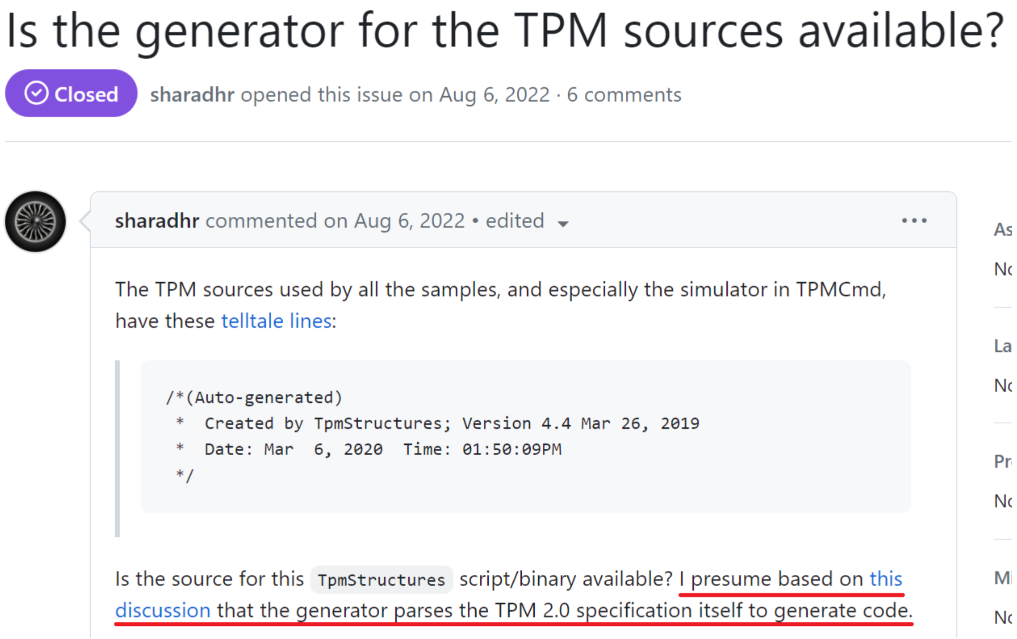
Error Returns	Meaning
TPM_RC_ATTRIBUTES	*keyHandle* does not reference a decryption key
TPM_RC_ECC_POINT	*secret* is invalid (when *keyHandle* is an ECC key)
TPM_RC_INSUFFICIENT	*secret* is invalid (when *keyHandle* is an ECC key)
TPM_RC_INTEGRITY	*credentialBlob* fails integrity test
TPM_RC_NO_RESULT	*secret* is invalid (when *keyHandle* is an ECC key)
TPM_RC_SIZE	*secret* size is invalid or the *credentialBlob* does not unmarshal correctly
TPM_RC_TYPE	*keyHandle* does not reference an asymmetric key.
TPM_RC_VALUE	*secret* is invalid (when *keyHandle* is an RSA key)

```c
TPM_RC
TPM2_ActivateCredential(
    ActivateCredential_In   *in,            // IN: input parameter list
    ActivateCredential_Out  *out            // OUT: output parameter list
    )
{
    TPM_RC                   result = TPM_RC_SUCCESS;
    OBJECT                  *object;            // decrypt key
    OBJECT                  *activateObject;    // key associated with credential
    TPM2B_DATA               data;          // credential data

// Input Validation

    // Get decrypt key pointer
    object = HandleToObject(in->keyHandle);

    // Get certificated object pointer
    activateObject = HandleToObject(in->activateHandle);
```



Is the generator for the TPM sources available?

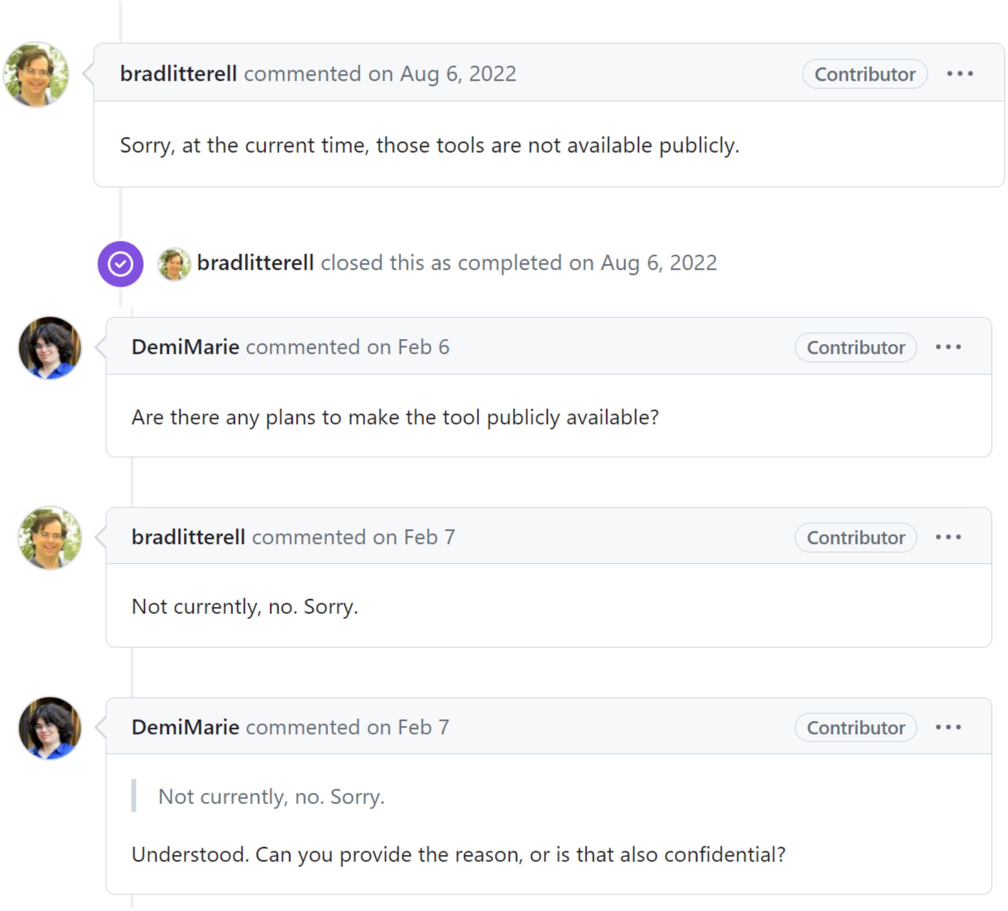
Closed sharadhr opened this issue on Aug 6, 2022 · 6 comments

sharadhr commented on Aug 6, 2022 • edited

The TPM sources used by all the samples, and especially the simulator in TPMCmd, have these telltale lines:

```
/*(Auto-generated)
 *  Created by TpmStructures; Version 4.4 Mar 26, 2019
 *  Date: Mar  6, 2020  Time: 01:50:09PM
 */
```

Is the source for this `TpmStructures` script/binary available? I presume based on this discussion that the generator parses the TPM 2.0 specification itself to generate code.



bradlitterell commented on Aug 6, 2022 Contributor

Sorry, at the current time, those tools are not available publicly.

bradlitterell closed this as completed on Aug 6, 2022

DemiMarie commented on Feb 6 Contributor

Are there any plans to make the tool publicly available?

bradlitterell commented on Feb 7 Contributor

Not currently, no. Sorry.

DemiMarie commented on Feb 7 Contributor

> Not currently, no. Sorry.

Understood. Can you provide the reason, or is that also confidential?

https://github.com/microsoft/ms-tpm-20-ref/issues/79

1.2 The TPM 2.0 Reference Implementation

- User space tools such as `tpm2-tools` abstract the underlying complexity.

1.2 The TPM 2.0 Reference Implementation

- User space tools such as `tpm2-tools` abstract the underlying complexity.
- Let's consider the `TPM2_StartAuthSession` command defined in the spec.
  - *This command is used to start an authorization session using alternative methods of establishing the session key (sessionKey). The session key is then used to derive values used for authorization and for encrypting parameters.*

1.2 The TPM 2.0 Reference Implementation

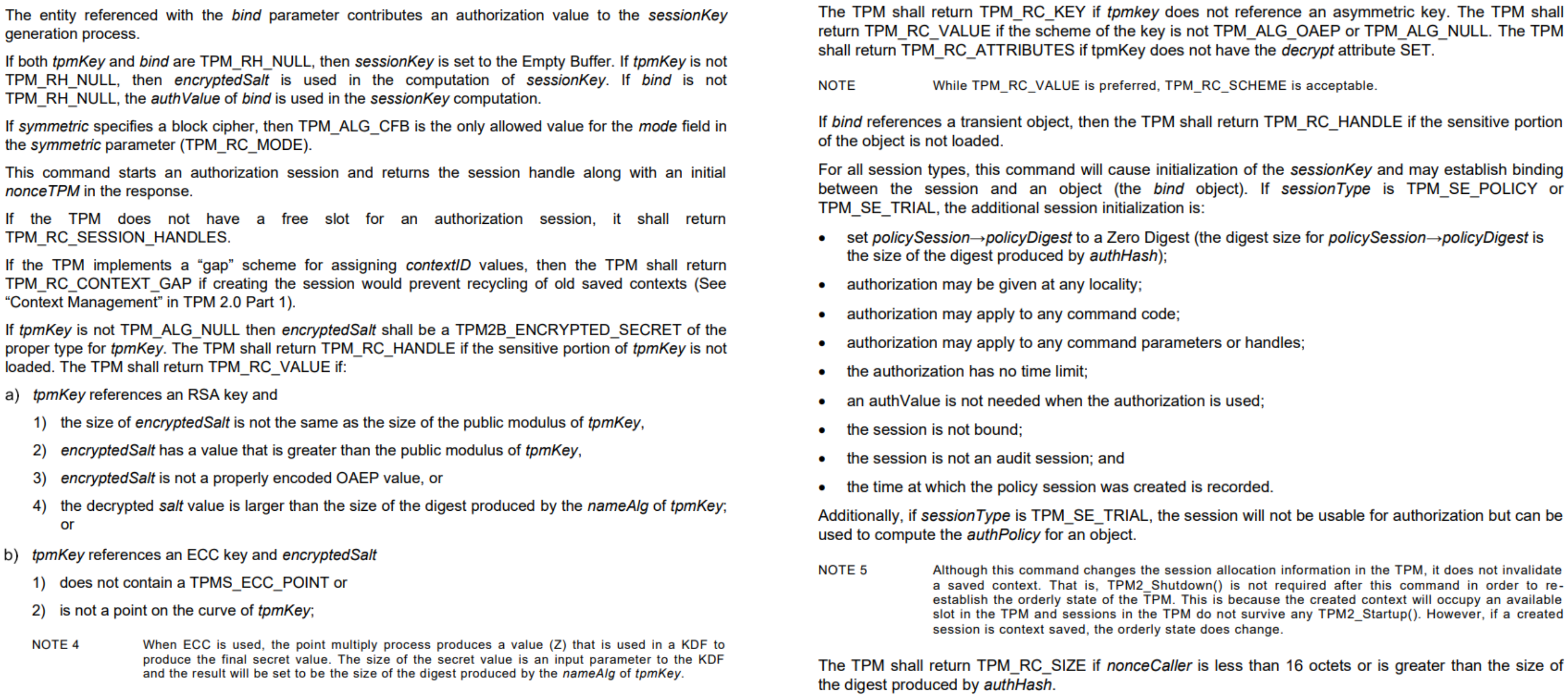
- User space tools such as `tpm2-tools` abstract the underlying complexity.
- Let's consider the `TPM2_StartAuthSession` command defined in the spec.
  - *This command is used to start an authorization session using alternative methods of establishing the session key (sessionKey). The session key is then used to derive values used for authorization and for encrypting parameters.*
- You can start an auth session using `tpm2-tools` like this:

```
# mknod "$HOME/backpipe" p
# while [ 1 ]; do tpm2_send 0<"$HOME/backpipe" | nc -lU "$HOME/sock" 1>"$HOME/backpipe"; done;

# tpm2_startauthsession --tcti="cmd:nc -q 0 -U $HOME/sock" <options>
```

1.2 The TPM 2.0 Reference Implementation

- But under the surface, the TPM 2.0 protocol is quite complex…



The entity referenced with the *bind* parameter contributes an authorization value to the *sessionKey* generation process.

If both *tpmKey* and *bind* are TPM_RH_NULL, then *sessionKey* is set to the Empty Buffer. If *tpmKey* is not TPM_RH_NULL, then *encryptedSalt* is used in the computation of *sessionKey*. If *bind* is not TPM_RH_NULL, the *authValue* of *bind* is used in the *sessionKey* computation.

If *symmetric* specifies a block cipher, then TPM_ALG_CFB is the only allowed value for the *mode* field in the *symmetric* parameter (TPM_RC_MODE).

This command starts an authorization session and returns the session handle along with an initial *nonceTPM* in the response.

If the TPM does not have a free slot for an authorization session, it shall return TPM_RC_SESSION_HANDLES.

If the TPM implements a "gap" scheme for assigning *contextID* values, then the TPM shall return TPM_RC_CONTEXT_GAP if creating the session would prevent recycling of old saved contexts (See "Context Management" in TPM 2.0 Part 1).

If *tpmKey* is not TPM_ALG_NULL then *encryptedSalt* shall be a TPM2B_ENCRYPTED_SECRET of the proper type for *tpmKey*. The TPM shall return TPM_RC_HANDLE if the sensitive portion of *tpmKey* is not loaded. The TPM shall return TPM_RC_VALUE if:

a) *tpmKey* references an RSA key and
   1) the size of *encryptedSalt* is not the same as the size of the public modulus of *tpmKey*,
   2) *encryptedSalt* has a value that is greater than the public modulus of *tpmKey*,
   3) *encryptedSalt* is not a properly encoded OAEP value, or
   4) the decrypted *salt* value is larger than the size of the digest produced by the *nameAlg* of *tpmKey*; or

b) *tpmKey* references an ECC key and *encryptedSalt*
   1) does not contain a TPMS_ECC_POINT or
   2) is not a point on the curve of *tpmKey*;

NOTE 4 When ECC is used, the point multiply process produces a value (Z) that is used in a KDF to produce the final secret value. The size of the secret value is an input parameter to the KDF and the result will be set to be the size of the digest produced by the *nameAlg* of *tpmKey*.

The TPM shall return TPM_RC_KEY if *tpmkey* does not reference an asymmetric key. The TPM shall return TPM_RC_VALUE if the scheme of the key is not TPM_ALG_OAEP or TPM_ALG_NULL. The TPM shall return TPM_RC_ATTRIBUTES if tpmKey does not have the *decrypt* attribute SET.

NOTE While TPM_RC_VALUE is preferred, TPM_RC_SCHEME is acceptable.

If *bind* references a transient object, then the TPM shall return TPM_RC_HANDLE if the sensitive portion of the object is not loaded.

For all session types, this command will cause initialization of the *sessionKey* and may establish binding between the session and an object (the *bind* object). If *sessionType* is TPM_SE_POLICY or TPM_SE_TRIAL, the additional session initialization is:

- set *policySession→policyDigest* to a Zero Digest (the digest size for *policySession→policyDigest* is the size of the digest produced by *authHash*);
- authorization may be given at any locality;
- authorization may apply to any command code;
- authorization may apply to any command parameters or handles;
- the authorization has no time limit;
- an authValue is not needed when the authorization is used;
- the session is not bound;
- the session is not an audit session; and
- the time at which the policy session was created is recorded.

Additionally, if *sessionType* is TPM_SE_TRIAL, the session will not be usable for authorization but can be used to compute the *authPolicy* for an object.

NOTE 5 Although this command changes the session allocation information in the TPM, it does not invalidate a saved context. That is, TPM2_Shutdown() is not required after this command in order to re-establish the orderly state of the TPM. This is because the created context will occupy an available slot in the TPM and sessions in the TPM do not survive any TPM2_Startup(). However, if a created session is context saved, the orderly state does change.

The TPM shall return TPM_RC_SIZE if *nonceCaller* is less than 16 octets or is greater than the size of the digest produced by *authHash*.

Part 2

Virtual TPMs

Virtual TPMs

- Nowadays, most virtualization solutions provide a virtual TPM.
  - AFAIK, Xen is the only one to provide pass-through access to the TPM, instead of using a virtual one.
- Implemented as an additional process running in the host system.
  - Both QEMU and VirtualBox use `libtpms`, an open source library based on the reference implementation.
- The way of sending TPM commands from the guest system to the TPM process on the host (and the other way around) is up to each implementation
  - SWTPM (embeds `libtpms`, used by QEMU) uses a TCP socket.

Virtual TPMs

- Virtual TPMs allow us to easily debug TPM firmware.
  - Great for doing research!
- On the other hand, they expose additional attack surface, that in a worst case scenario could allow to escape from the VM to the host side.

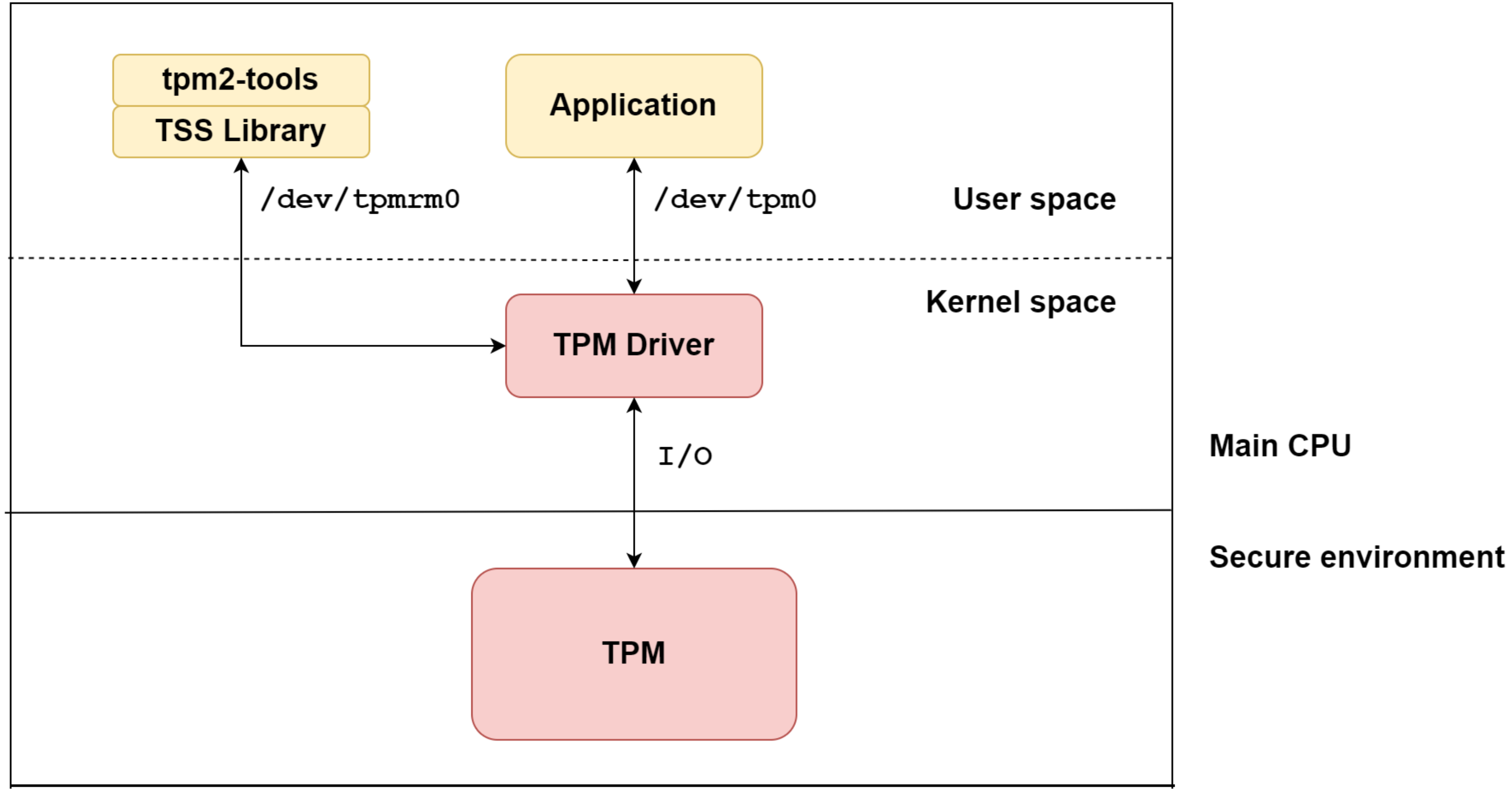
Part 3

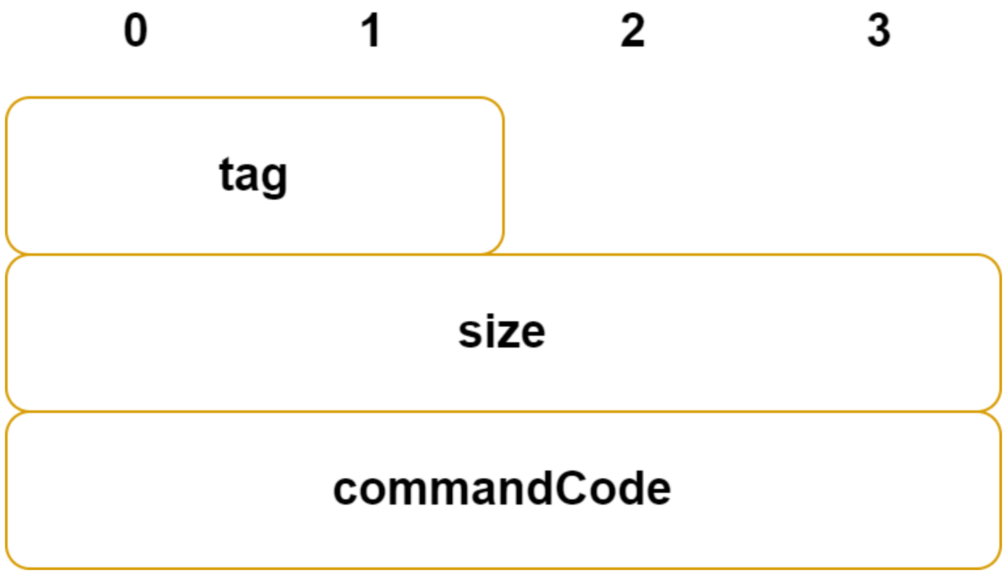
TPM 2.0 protocol internals

Part 3.1

Commands and Responses

Architecture





0 1 2 3

tag

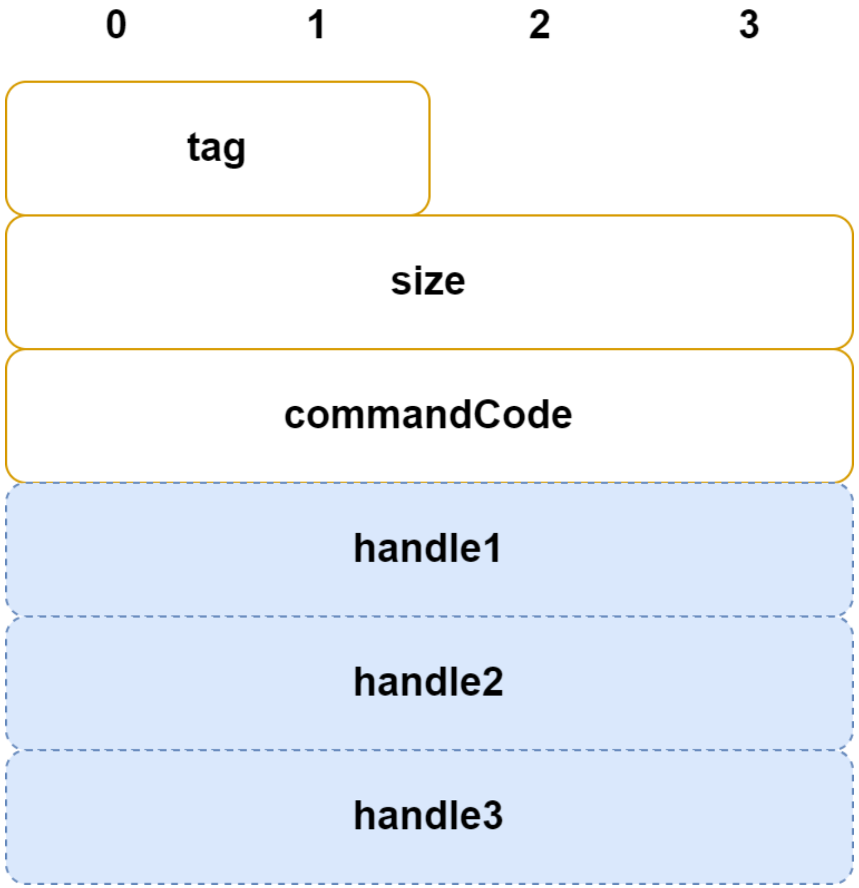
size

commandCode

TPM Base Command Header

```
/* Tpm2 command tags. */
#define TPM_ST_NO_SESSIONS 0x8001
#define TPM_ST_SESSIONS     0x8002
```

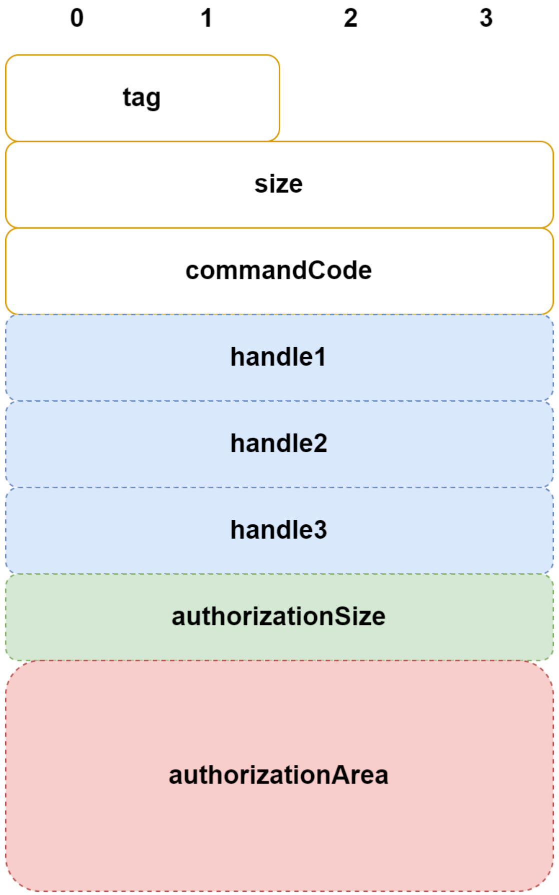
```
typedef UINT32                  TPM_CC;
[...]
#define TPM_CC_PolicySecret   (TPM_CC)(0x00000151)
#define TPM_CC_Rewrap         (TPM_CC)(0x00000152)
#define TPM_CC_Create         (TPM_CC)(0x00000153)
#define TPM_CC_ECDH_ZGen      (TPM_CC)(0x00000154)
#define TPM_CC_HMAC           (TPM_CC)(0x00000155)
#define TPM_CC_Import         (TPM_CC)(0x00000156)
#define TPM_CC_Load           (TPM_CC)(0x00000157)
#define TPM_CC_Quote          (TPM_CC)(0x00000158)
#define TPM_CC_RSA_Decrypt    (TPM_CC)(0x00000159)
[...]
```



TPM Command with Handles

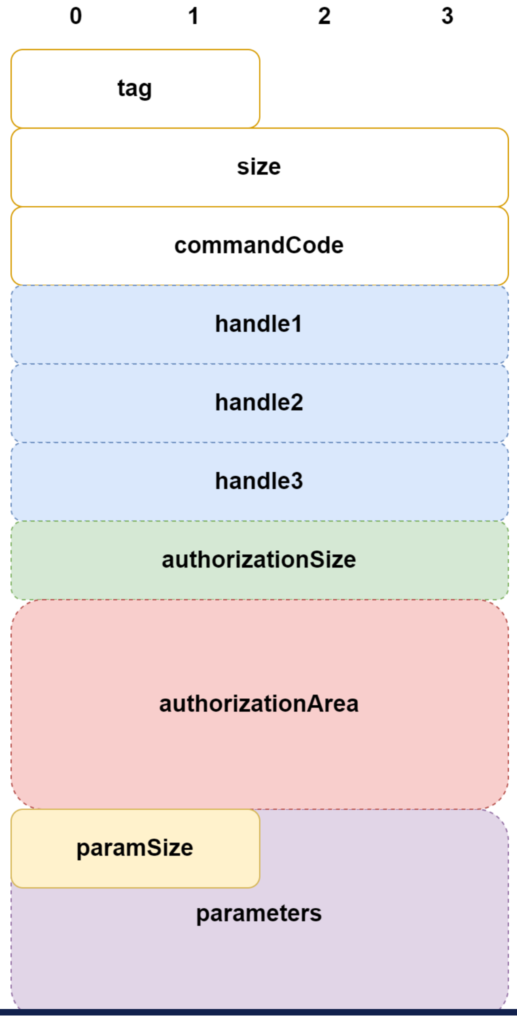
- Command-dependent
- 0 to 3 handles

```
typedef TPM_HANDLE TPM_RH;
#define TPM_RH_FIRST           0x40000000
#define TPM_RH_SRK             0x40000000
#define TPM_RH_OWNER           0x40000001
#define TPM_RH_REVOKE          0x40000002
#define TPM_RH_TRANSPORT       0x40000003
#define TPM_RH_OPERATOR        0x40000004
#define TPM_RH_ADMIN           0x40000005
[...]
```



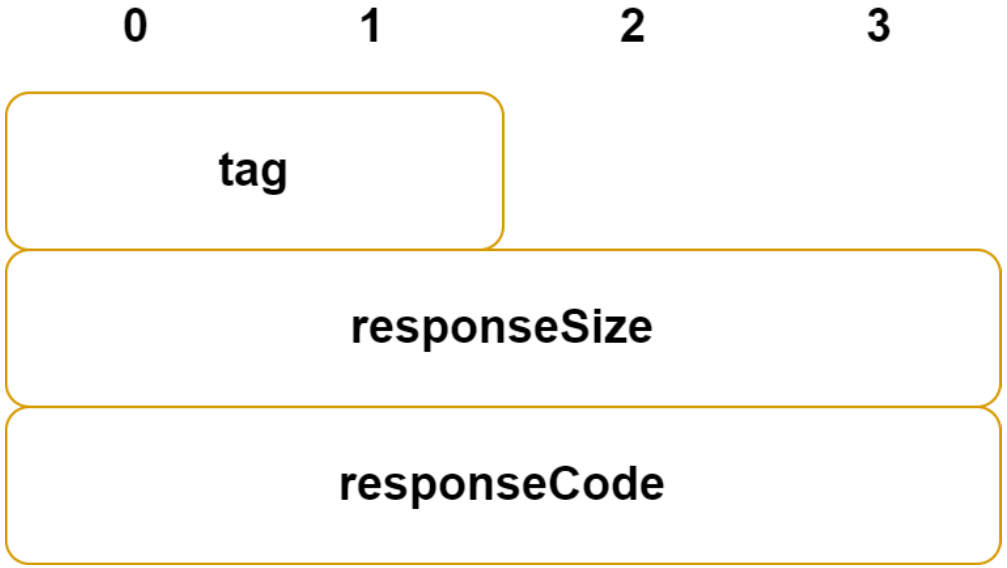
TPM Command with Authorization Area

- Authorization area contains 1 to 3 session structures.
    - Also called *Session Area* in the reference implementation code.
- Authorization area is only present if the `tag` of the command is `TPM_ST_SESSIONS`



TPM Command with Parameters

- Parameter contents are command-dependent.
- Parameters are only present if the `tag` of the command is `TPM_ST_SESSIONS`



0 1 2 3

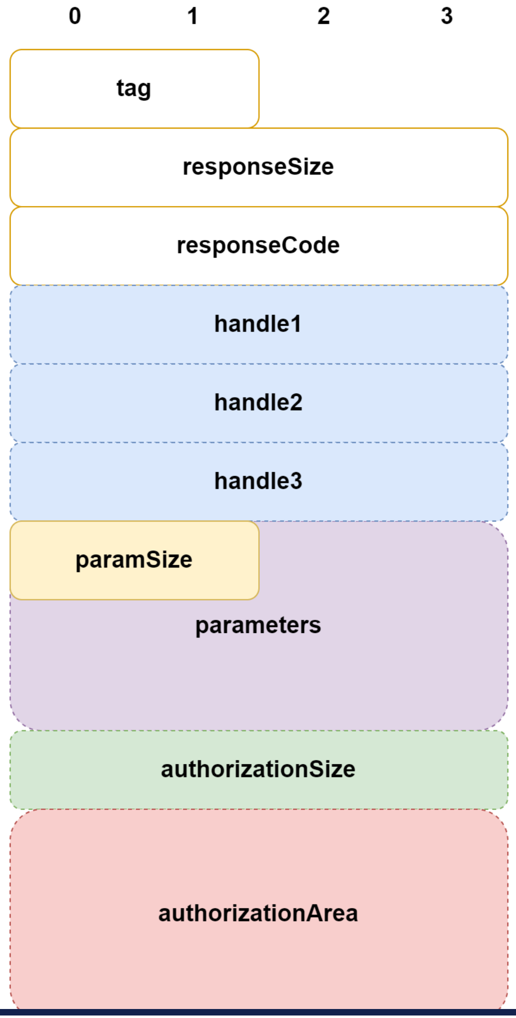
tag

responseSize

responseCode

TPM Basic Response

- `responseCode == 0` ⟶ indicates success
- `responseCode != 0` ⟶ indicates error condition



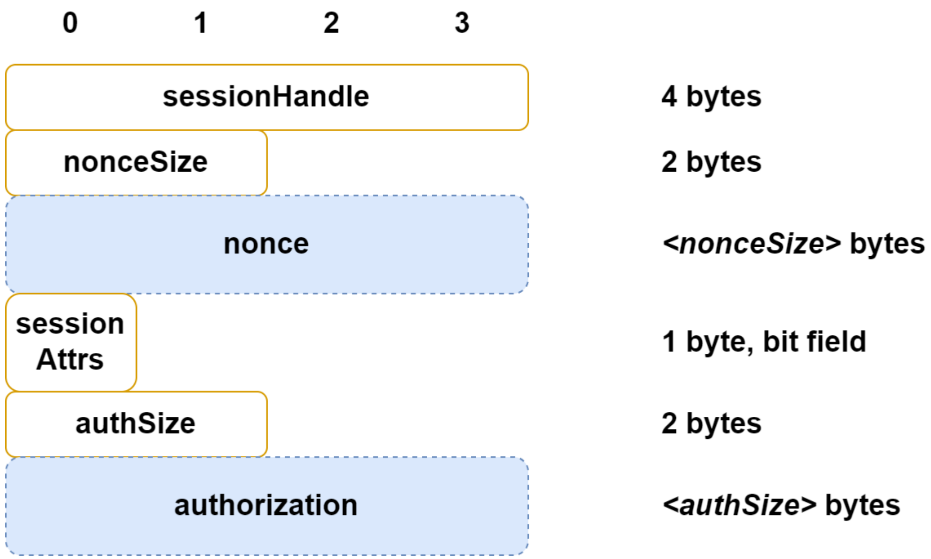
TPM Response with Fields

- Response may contain handles
- Response may contain parameters
- Response may contain authorization area
  - It's all command-dependent
- Notice the inverted order between authorization and parameters areas

Part 3.2

Authorization Area

Authorization Area



0 1 2 3	
sessionHandle	4 bytes
nonceSize	2 bytes
nonce	*<nonceSize>* bytes
session Attrs	1 byte, bit field
authSize	2 bytes
authorization	*<authSize>* bytes

- Session attributes:

```
typedef struct _TPMA_SESSION {
    UINT8 continueSession : 1;
    UINT8 auditExclusive : 1;
    UINT8 auditReset : 1;
    UINT8 reserved3_4 : 2;
    UINT8 decrypt : 1;
    UINT8 encrypt : 1;
    UINT8 audit : 1;
} TPMA_SESSION;
```

- Authorization: either HMAC or password

Part 4

Vulnerabilities: CVE-2023-1017 and CVE-2023-1018

Part 4.1

Bug #1 - OOB read in CryptParameterDecryption function (CVE-2023-1018)

4. CVE-2023-1017 and CVE-2023-1018

ExecCommand.c

```
LIB_EXPORT void
ExecuteCommand(
    unsigned    int        requestSize,       //   IN: command buffer size
    unsigned    char      *request,           //   IN: command buffer
    unsigned    int       *responseSize,      //   OUT: response buffer size
    unsigned    char      **response          //   OUT: response buffer
    )
        [...]
        // Find out session buffer size.
        result = UINT32_Unmarshal(&authorizationSize, &buffer, &size);
        if(result != TPM_RC_SUCCESS)
            goto Cleanup;
        // Perform sanity check on the unmarshaled    value. If it is smaller than
        // the smallest possible session or larger    than the remaining size of
        // the command, then it is an error. NOTE:    This check could pass but the
        // session size could still be wrong. That    will be determined after the
        // sessions are unmarshaled.
[1]     if(    authorizationSize < 9
            || authorizationSize > (UINT32) size)
        {
             result = TPM_RC_SIZE;
             goto Cleanup;
        }
[...]
```

4. CVE-2023-1017 and CVE-2023-1018

```
[...]
        // The sessions, if any, follows authorizationSize.
        sessionBufferStart = buffer;
        // The parameters follow the session area.
[2]     parmBufferStart = sessionBufferStart + authorizationSize;
        // Any data left over after removing the authorization sessions is
        // parameter data. If the command does not have parameters, then an
        // error will be returned if the remaining size is not zero. This is
        // checked later.
[3]     parmBufferSize = size - authorizationSize;
        // The actions of ParseSessionBuffer() are described in the introduction.
[4]     result = ParseSessionBuffer(commandCode,
                                    handleNum,
                                    handles,
                                    sessionBufferStart,
                                    authorizationSize,
[5]                                 parmBufferStart,
[6]                                 parmBufferSize);
        [...]
```

4. CVE-2023-1017 and CVE-2023-1018

SessionProcess.c

```
TPM_RC
ParseSessionBuffer(
    TPM_CC              commandCode,                    //   IN:   Command code
    UINT32              handleNum,                      //   IN:   number of element in handle array
    TPM_HANDLE          handles[],                      //   IN:   array of handle
    BYTE               *sessionBufferStart,             //   IN:   start of session buffer
    UINT32              sessionBufferSize,              //   IN:   size of session buffer
    BYTE               *parmBufferStart,                //   IN:   start of parameter buffer
    UINT32              parmBufferSize                  //   IN:   size of parameter buffer
    )
{
        [...]
        // Decrypt the first parameter if applicable. This should be the last operation
        // in session processing.
[1]     if(s_decryptSessionIndex != UNDEFINED_INDEX){
            [...]
            size = DecryptSize(commandCode);
[2]         result = CryptParameterDecryption(
                          s_sessionHandles[s_decryptSessionIndex],
                          &s_nonceCaller[s_decryptSessionIndex].b,
[3]                       parmBufferSize, (UINT16)size,
                          &extraKey,
[4]                       parmBufferStart);
```

4. CVE-2023-1017 and CVE-2023-1018

CryptUtil.c

```c
//        This function does in-place decryption of a command parameter.
TPM_RC
CryptParameterDecryption(
   TPM_HANDLE           handle,                 //   IN: encrypted session handle
   TPM2B               *nonceCaller,            //   IN: nonce caller
   UINT32               bufferSize,             //   IN: size of parameter buffer
   UINT16               leadingSizeInByte,      //   IN: the size of the leading size field in byte
   TPM2B_AUTH          *extraKey,               //   IN: the authValue
   BYTE                *buffer                  //   IN/OUT: parameter buffer to be decrypted
   )
{
       [...]
       // The first two bytes of the buffer are the size of the
       // data to be decrypted
[1]    cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
[2]    buffer = &buffer[2];    // advance the buffer
       [...]
```

swap.h

```c
#define BYTE_ARRAY_TO_UINT16(b)         (UINT16)( ((b)[0] << 8) \
                                                + (b)[1])
```

Bug #1 - OOB read in CryptParameterDecryption function (CVE-2023-1018)

- `CryptParameterDecryption` function in `CryptUtil.c` uses the `BYTE_ARRAY_TO_UINT16` macro to read a 16-bit field (`cipherSize`) from the buffer pointed by `parmBufferStart` without checking if there's any parameter data past the session area.

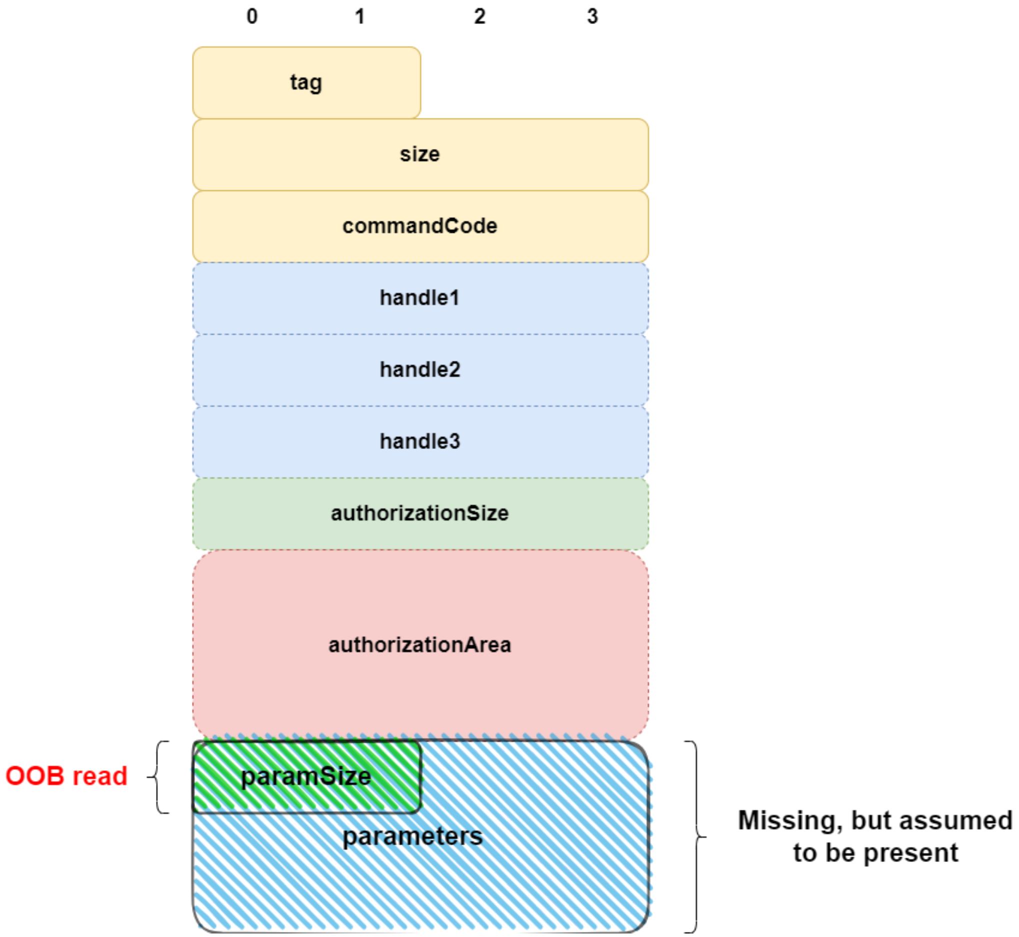
Bug #1 - OOB read in CryptParameterDecryption function (CVE-2023-1018)

- `CryptParameterDecryption` function in `CryptUtil.c` uses the `BYTE_ARRAY_TO_UINT16` macro to read a 16-bit field (`cipherSize`) from the buffer pointed by `parmBufferStart` without checking if there's any parameter data past the session area.
- If a malformed command doesn't contain a `parameterArea` past the `sessionArea`, it will trigger an out-of-bounds memory read, making the TPM access memory past the end of the command.

Bug #1 - OOB read in CryptParameterDecryption function (CVE-2023-1018)

- `CryptParameterDecryption` function in `CryptUtil.c` uses the `BYTE_ARRAY_TO_UINT16` macro to read a 16-bit field (`cipherSize`) from the buffer pointed by `parmBufferStart` without checking if there's any parameter data past the session area.
- If a malformed command doesn't contain a `parameterArea` past the `sessionArea`, it will trigger an out-of-bounds memory read, making the TPM access memory past the end of the command.
- The `UINT16_Unmarshal` function should have been used instead, which performs proper size checks before reading from a given buffer.

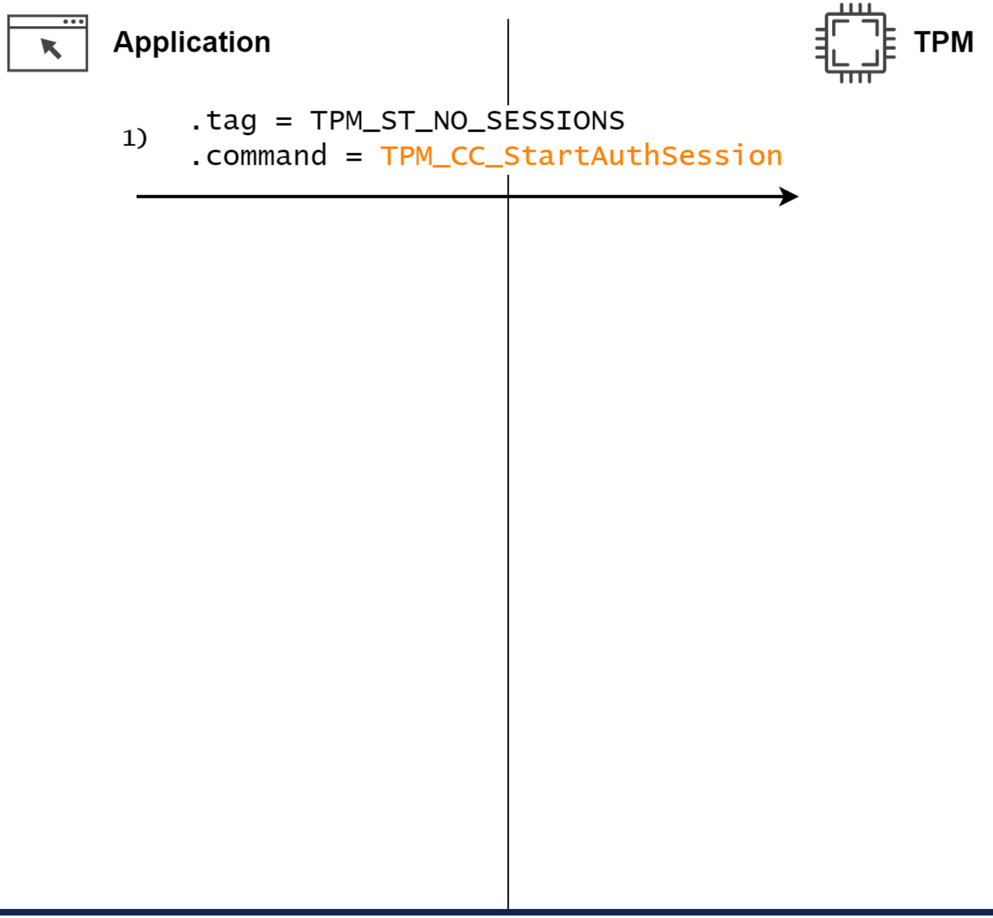
4. CVE-2023-1017 and CVE-2023-1018



Bug #1 - OOB read in CryptParameterDecryption function (CVE-2023-1018)

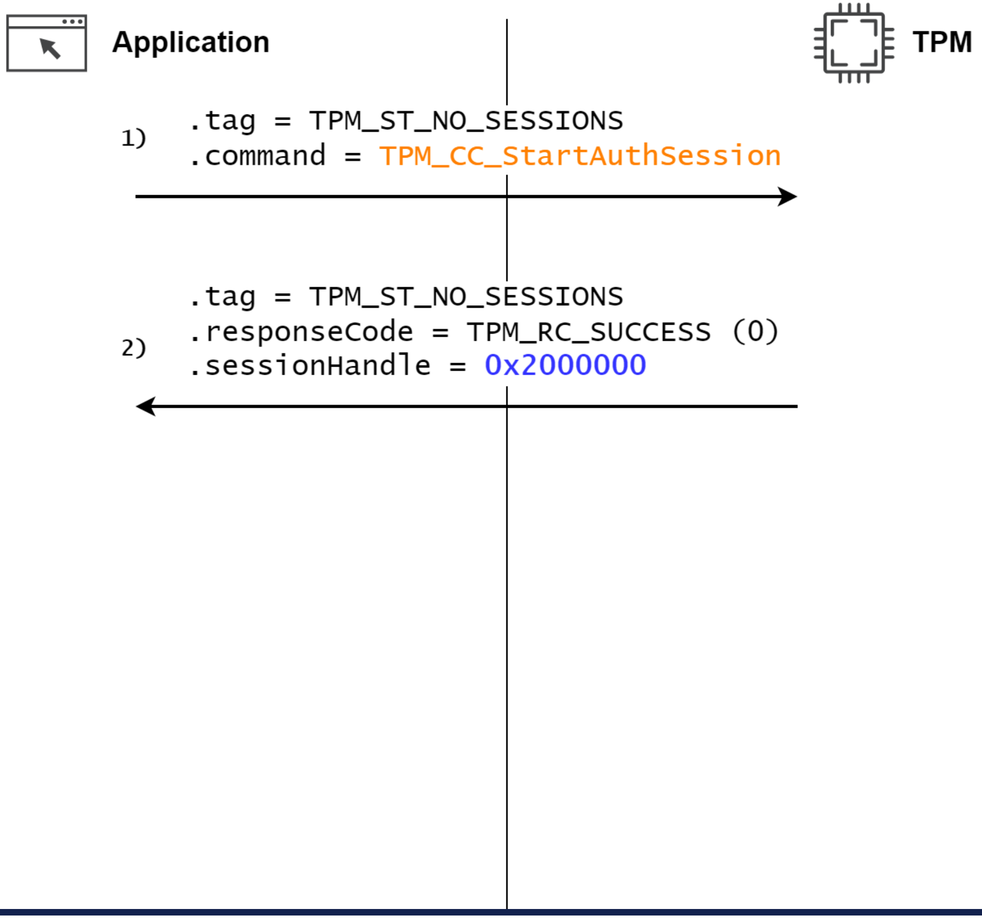
```
TPM_RC uint16_t_Unmarshal(uint16_t* target, BYTE** buffer, INT32* size) {
  uint16_t value_net = 0;
   if (!size || *size < sizeof(uint16_t)) {
     return TPM_RC_INSUFFICIENT;
  }
  memcpy(&value_net, *buffer, sizeof(uint16_t));
  switch (sizeof(uint16_t)) {
    case 2:
      *target = be16toh(value_net);
      break;
    case 4:
      *target = be32toh(value_net);
      break;
    case 8:
      *target = be64toh(value_net);
      break;
    default:
      *target = value_net;
  }
  *buffer += sizeof(uint16_t);
  *size -= sizeof(uint16_t);
  return TPM_RC_SUCCESS;
```

Step 1) - Start Auth Session



1)
```
.tag = TPM_ST_NO_SESSIONS
.command = TPM_CC_StartAuthSession
```

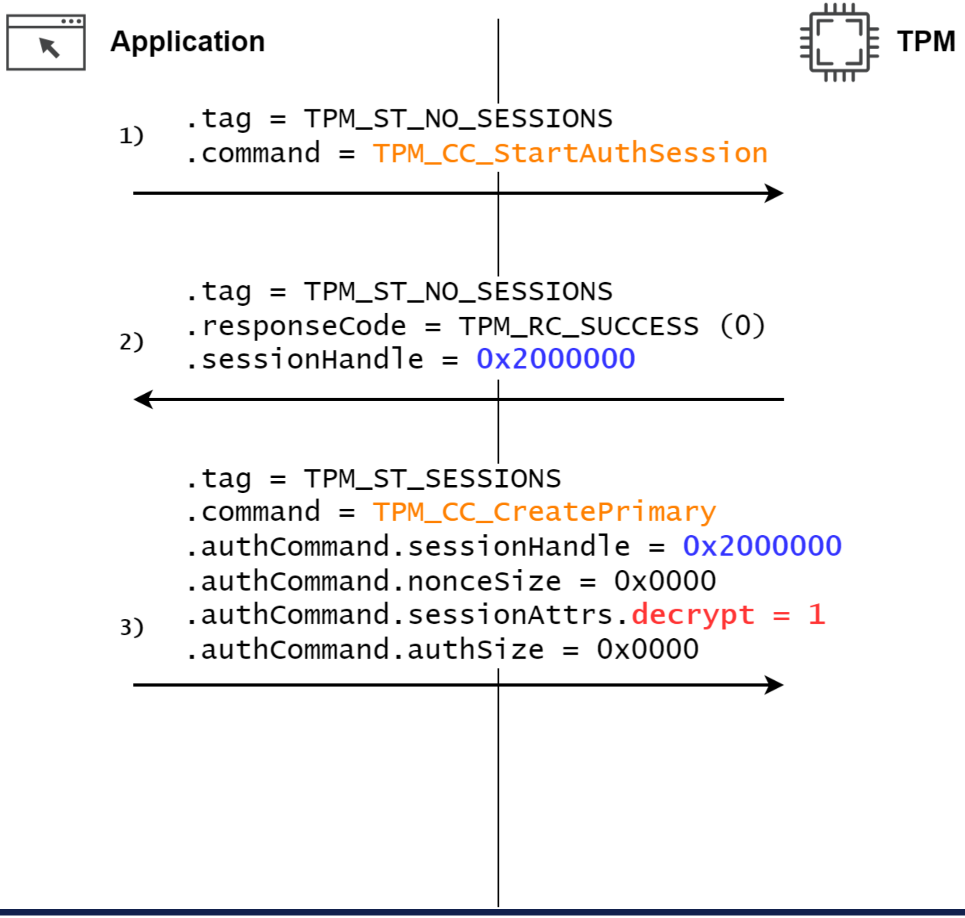
Step 2) - Auth Response



```
.tag = TPM_ST_NO_SESSIONS
.command = TPM_CC_StartAuthSession
```

```
.tag = TPM_ST_NO_SESSIONS
.responseCode = TPM_RC_SUCCESS (0)
.sessionHandle = 0x2000000
```

Step 3) - Create Primary with no Parameter Area



1)

```
.tag = TPM_ST_NO_SESSIONS
.command = TPM_CC_StartAuthSession
```

2)

```
.tag = TPM_ST_NO_SESSIONS
.responseCode = TPM_RC_SUCCESS (0)
.sessionHandle = 0x2000000
```

3)

```
.tag = TPM_ST_SESSIONS
.command = TPM_CC_CreatePrimary
.authCommand.sessionHandle = 0x2000000
.authCommand.nonceSize = 0x0000
.authCommand.sessionAttrs.decrypt = 1
.authCommand.authSize = 0x0000
```

Part 4.2

Bug #2 - OOB write in CryptParameterDecryption function (CVE-2023-1017)

4. CVE-2023-1017 and CVE-2023-1018

CryptUtil.c

```
//       This function does in-place decryption of a command parameter.
TPM_RC
CryptParameterDecryption(
   TPM_HANDLE          handle,                 //    IN: encrypted session handle
   TPM2B              *nonceCaller,            //    IN: nonce caller
   UINT32              bufferSize,             //    IN: size of parameter buffer
   UINT16              leadingSizeInByte,      //    IN: the size of the leading size field in byte
   TPM2B_AUTH         *extraKey,               //    IN: the authValue
   BYTE               *buffer                  //    IN/OUT: parameter buffer to be decrypted
   )
{
       [...]
       // The first two bytes of the buffer are the size of the
       // data to be decrypted
[1]    cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
[2]    buffer = &buffer[2];     // advance the buffer
       [...]
```

(continues next slide)

4. CVE-2023-1017 and CVE-2023-1018

(continued)

```
[...]
[3] if(cipherSize > bufferSize)
       return TPM_RC_SIZE;
   // Compute decryption key by concatenating sessionAuth with extra input key
   MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
   MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));
   if(session->symmetric.algorithm == TPM_ALG_XOR)
       // XOR parameter decryption formulation:
       //    XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
       // Call XOR obfuscation function
[4]    CryptXORObfuscation(session->authHashAlg, &key.b, nonceCaller,
                                  &(session->nonceTPM.b), cipherSize, buffer);
   else
       // Assume that it is one of the symmetric block ciphers.
[5]    ParmDecryptSym(session->symmetric.algorithm, session->authHashAlg,
                             session->symmetric.keyBits.sym,
                             &key.b, nonceCaller, &session->nonceTPM.b,
                             cipherSize, buffer);
   return TPM_RC_SUCCESS;
}
```

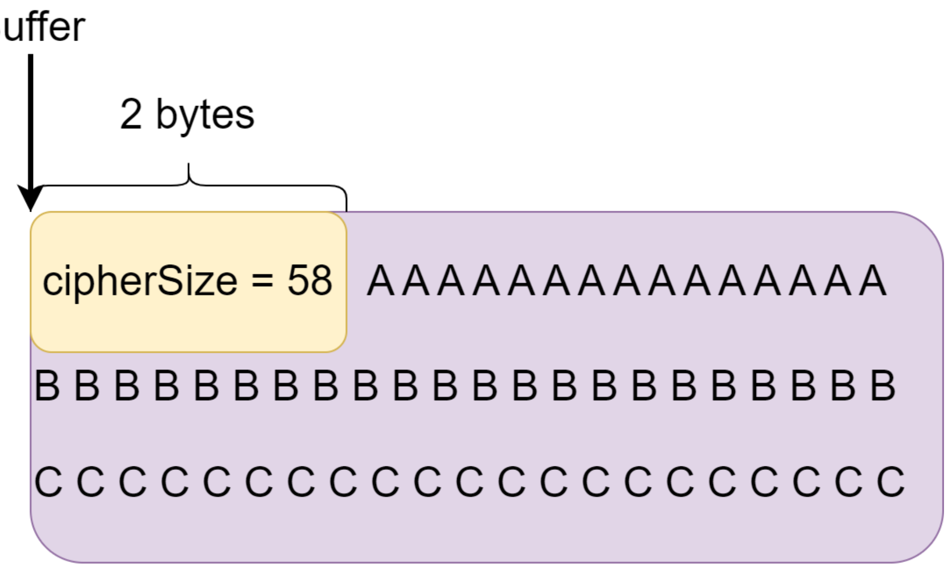
Bug #2 - OOB write in CryptParameterDecryption function (CVE-2023-1017)

- If a proper `parameterArea` is provided (avoiding bug #1), the first two bytes of it are interpreted as the size of the data to be decrypted (`cipherSize`), and the buffer pointer is advanced by 2.
- There's an attempt of a sanity check: if `cipherSize` value is greater than the actual size of `parameterArea`, then it bails out.
- But there's a problem here: after reading the `cipherSize` 16-bit field and advancing the buffer pointer by 2, the function forgets to subtract 2 from `bufferSize`, to account for the 2 bytes that were already processed.

Bug #2 - OOB write in CryptParameterDecryption function (CVE-2023-1017)

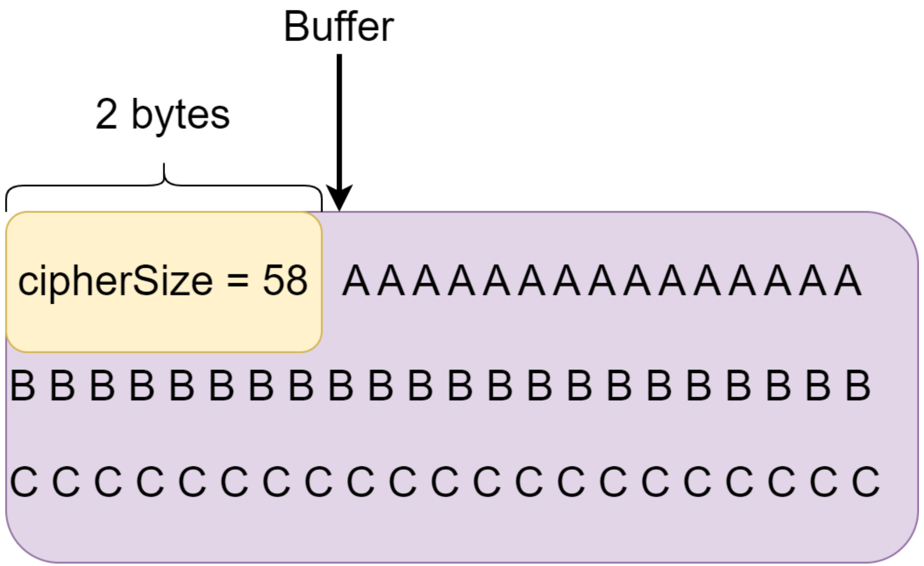
- It's possible to pass the sanity check with a `cipherSize` value that is **larger by 2** than the actual size of the remaining data.
- As a consequence, when either `CryptXORObfuscation()` or `ParmDecryptSym()` are called to decrypt the data in the `parameterArea` following the `cipherSize` field, the TPM ends up **writing 2 bytes past the end of the buffer**, resulting in an out-of-bounds write.

State before parsing Parameter Area


Buffer
2 bytes
cipherSize = 58
A A A A A A A A A A A A A A A
B B B B B B B B B B B B B B B B B B B B B B
C C C C C C C C C C C C C C C C C C C C C

BufferSize (length of remaining data) = **60**

Expected state after parsing cipherSize



Buffer

2 bytes

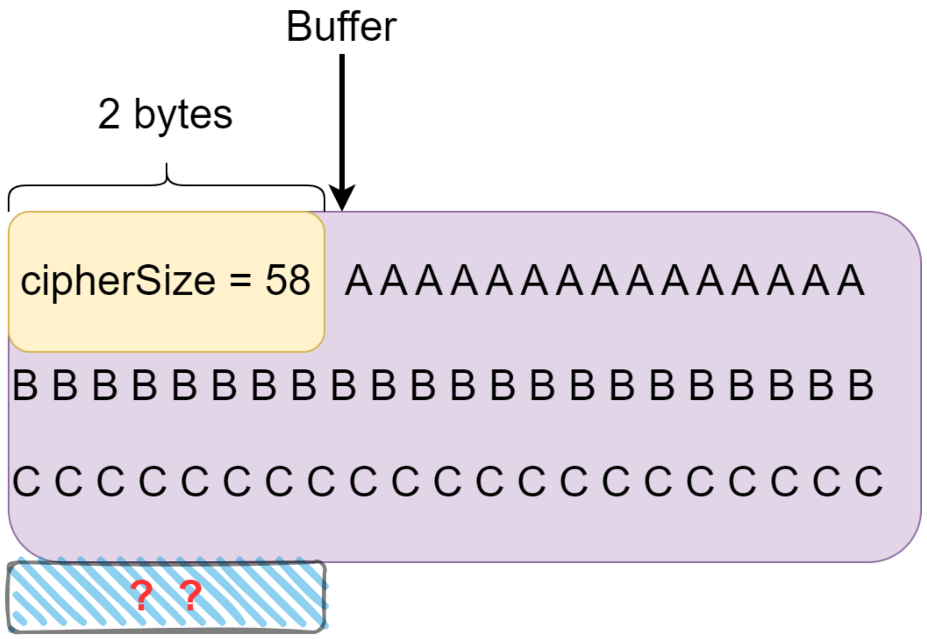
cipherSize = 58 A A A A A A A A A A A A A A A

B B B B B B B B B B B B B B B B B B B B B B

C C C C C C C C C C C C C C C C C C C C C

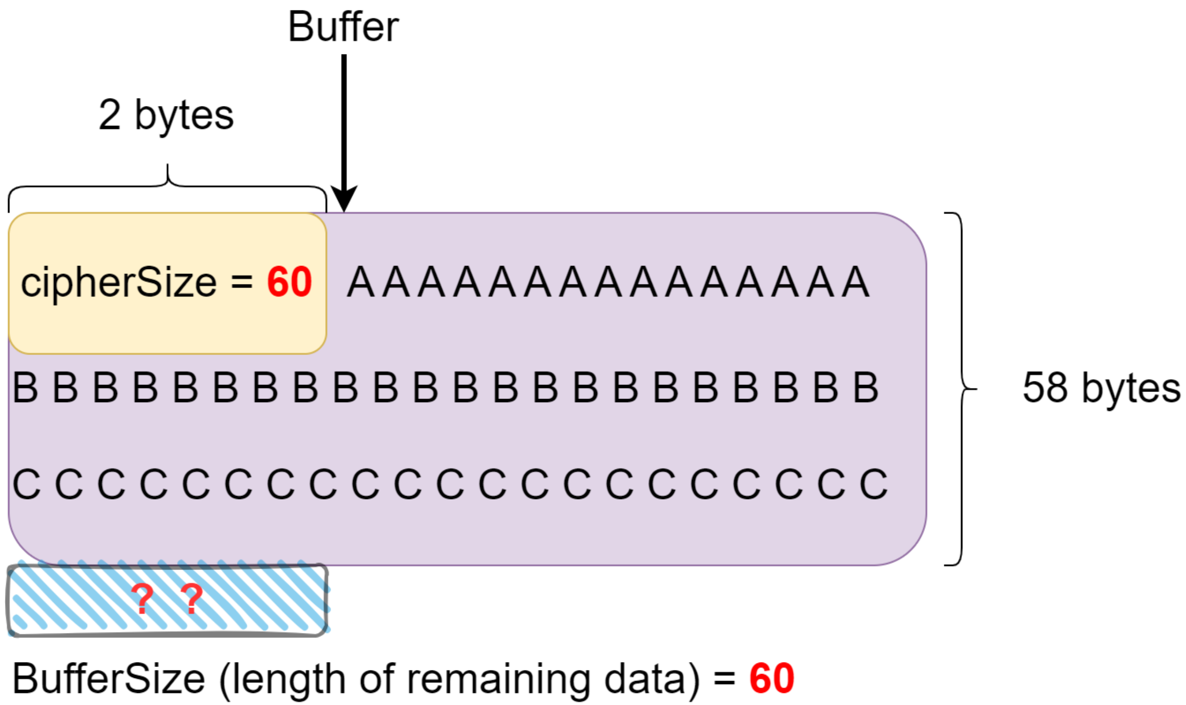
BufferSize (length of remaining data) = **58**

Actual state after parsing cipherSize

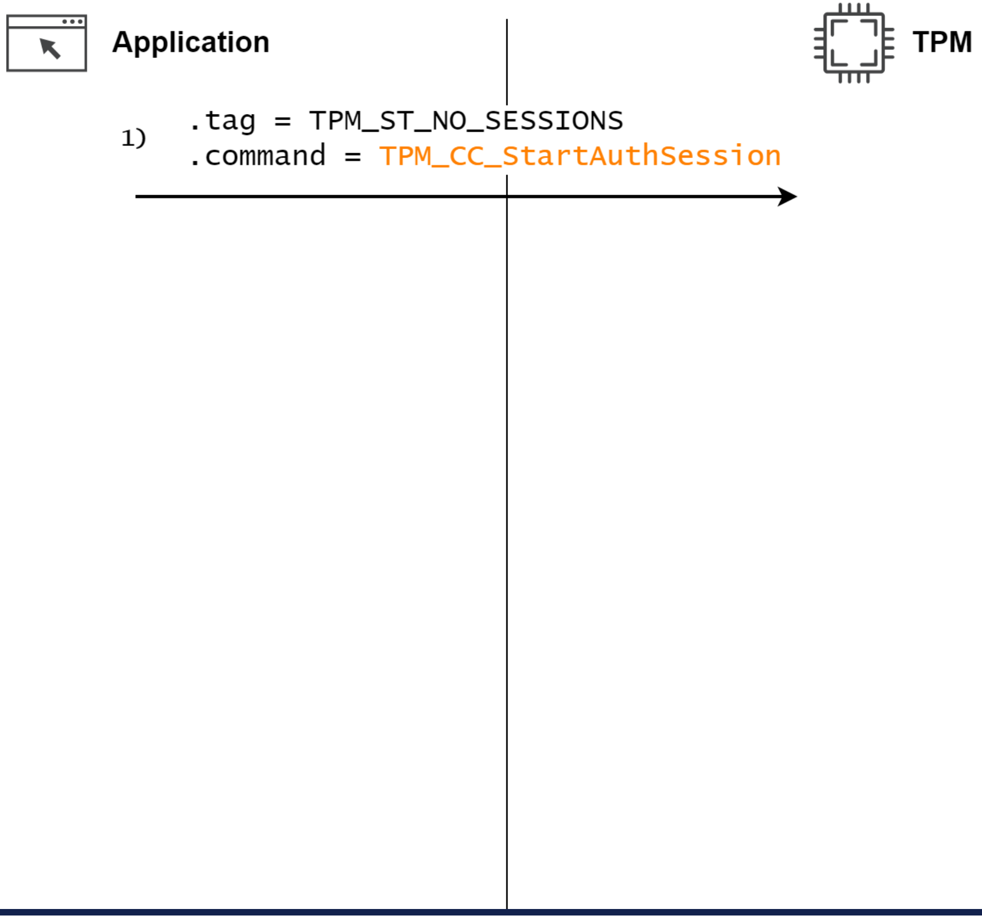

Buffer
2 bytes
cipherSize = 58
A A A A A A A A A A A A A A A
B B B B B B B B B B B B B B B B B B B B B B
C C C C C C C C C C C C C C C C C C C C C
? ?

BufferSize (length of remaining data) = **60**

This state becomes valid!

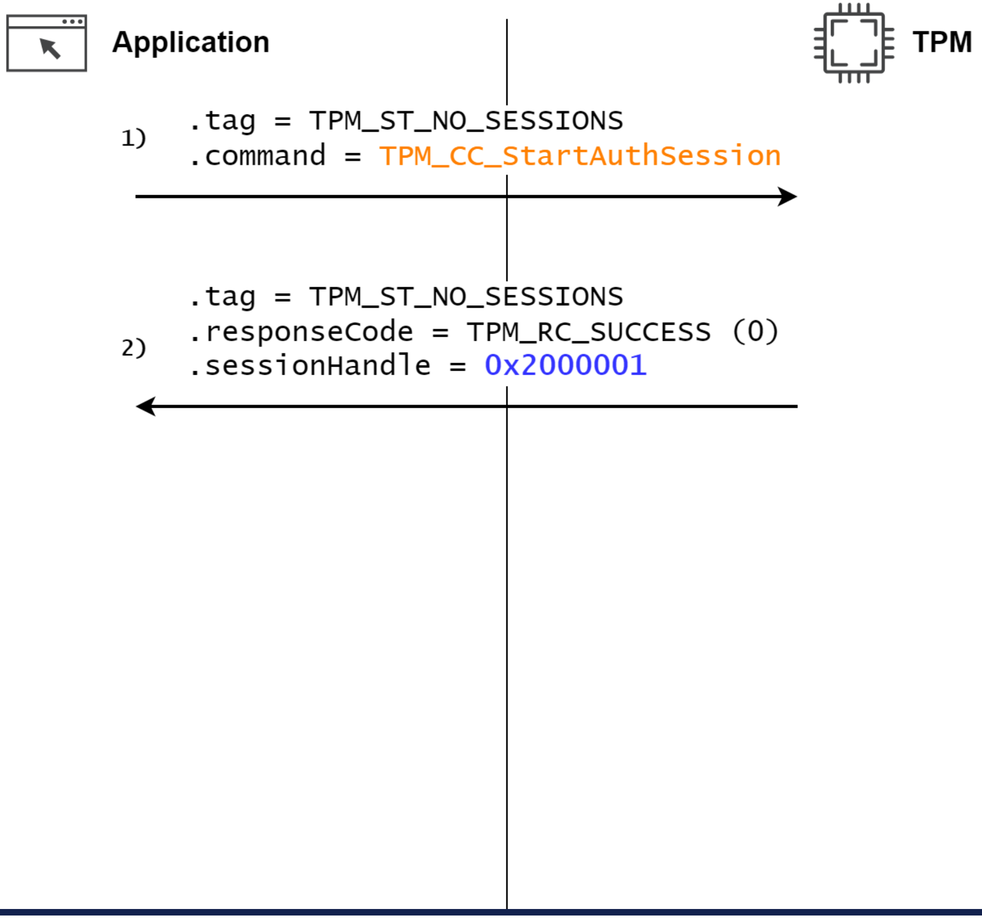

Buffer
2 bytes
cipherSize = 60
A A A A A A A A A A A A A A A
B B B B B B B B B B B B B B B B B B B B B B
C C C C C C C C C C C C C C C C C C C C C C
58 bytes
? ?
BufferSize (length of remaining data) = 60

Step 1) - Start Auth Session



```
1)  .tag = TPM_ST_NO_SESSIONS
    .command = TPM_CC_StartAuthSession
```

Step 2) - Auth Response



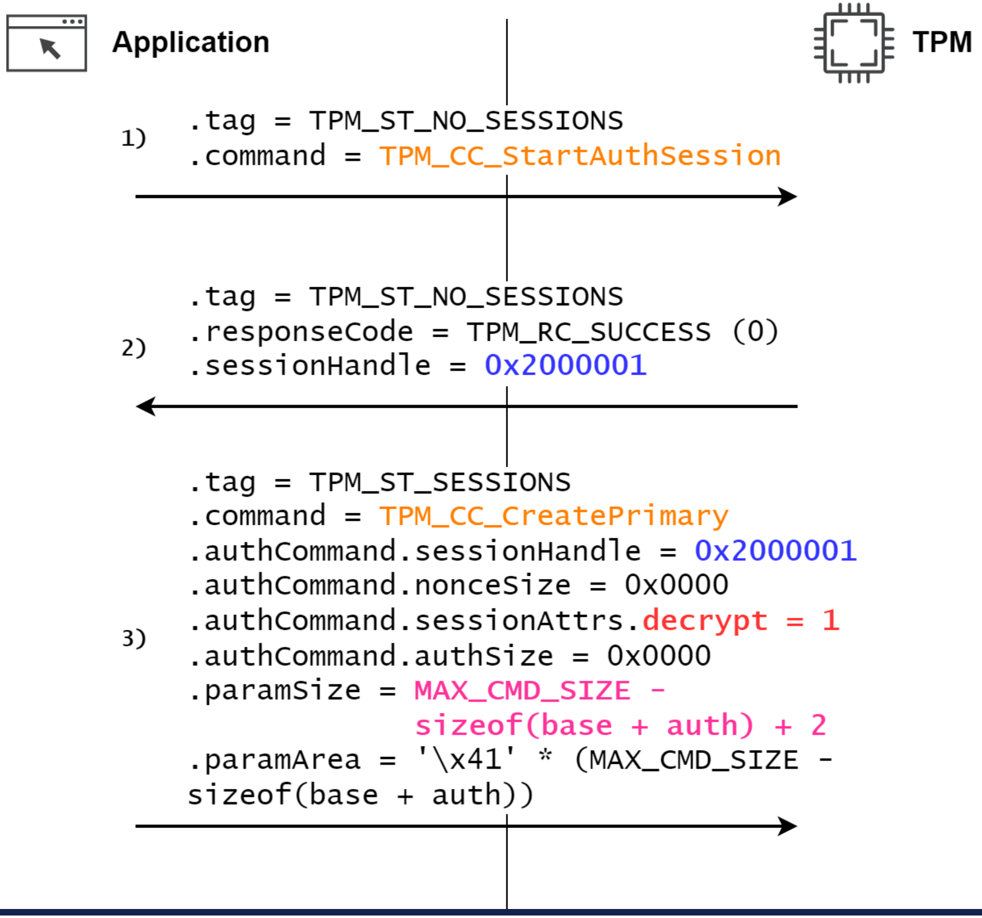
1)

```
.tag = TPM_ST_NO_SESSIONS
.command = TPM_CC_StartAuthSession
```

2)

```
.tag = TPM_ST_NO_SESSIONS
.responseCode = TPM_RC_SUCCESS (0)
.sessionHandle = 0x2000001
```

Step 3) - Create Primary with crafted paramSize



Application

TPM

1)

```
.tag = TPM_ST_NO_SESSIONS
.command = TPM_CC_StartAuthSession
```

2)

```
.tag = TPM_ST_NO_SESSIONS
.responseCode = TPM_RC_SUCCESS (0)
.sessionHandle = 0x2000001
```

3)

```
.tag = TPM_ST_SESSIONS
.command = TPM_CC_CreatePrimary
.authCommand.sessionHandle = 0x2000001
.authCommand.nonceSize = 0x0000
.authCommand.sessionAttrs.decrypt = 1
.authCommand.authSize = 0x0000
.paramSize = MAX_CMD_SIZE -
             sizeof(base + auth) + 2
.paramArea = '\x41' * (MAX_CMD_SIZE -
sizeof(base + auth))
```

Part 4.3

Impact of the vulnerabilities

1 - Impact of the OOB read

- Function `CryptParameterDecryption` in `CryptUtil.c` can read 2 bytes past the end of the received TPM command. If an affected TPM doesn't zero out the command buffer between received commands, it can result in the affected function reading whatever 16-bit value was already there from a previous command.

1 - Impact of the OOB read

- Function `CryptParameterDecryption` in `CryptUtil.c` can read 2 bytes past the end of the received TPM command. If an affected TPM doesn't zero out the command buffer between received commands, it can result in the affected function reading whatever 16-bit value was already there from a previous command.
  - This is dependant on each implementation.

2 - Impact of the OOB write

- Functions `CryptXORObfuscation`/`ParmDecryptSym` in `CryptUtil.c` (called from `CryptParameterDecryption`) can write 2 bytes past the end of the command buffer, resulting in memory corruption.

2 - Impact of the OOB write

- Functions `CryptXORObfuscation`/`ParmDecryptSym` in `CryptUtil.c` (called from `CryptParameterDecryption`) can write 2 bytes past the end of the command buffer, resulting in memory corruption.
- The chances of having something useful to overwrite adjacent to the command buffer **depend on how each implementation** allocates the buffer that receives TPM commands.

2 - Impact of the OOB write

- Functions `CryptXORObfuscation`/`ParmDecryptSym` in `CryptUtil.c` (called from `CryptParameterDecryption`) can write 2 bytes past the end of the command buffer, resulting in memory corruption.
- The chances of having something useful to overwrite adjacent to the command buffer **depend on how each implementation** allocates the buffer that receives TPM commands.
  - **SWTPM (QEMU)** uses `malloc()` to allocate a command buffer of size 0x1008 (8 bytes for a `send command prefix` that can be used to modify the locality, plus 0x1000 bytes for the maximum TPM command size).

2 - Impact of the OOB write

- Worst case scenario: OOB write ⟶ code execution on the TPM
  - VM escape in the case of a virtual TPM

2 - Impact of the OOB write

- Worst case scenario: OOB write ⟶ code execution on the TPM
  - VM escape in the case of a virtual TPM
- Corrupting TPM memory containing sensitive data such as a key

2 - Impact of the OOB write

- Worst case scenario: OOB write ⟶ code execution on the TPM
  - VM escape in the case of a virtual TPM
- Corrupting TPM memory containing sensitive data such as a key
- A DoS can cause enough trouble:
  - Failure for full disk encryption solutions relying on the TPM
  - Failure to perform boot attestation

Part 5

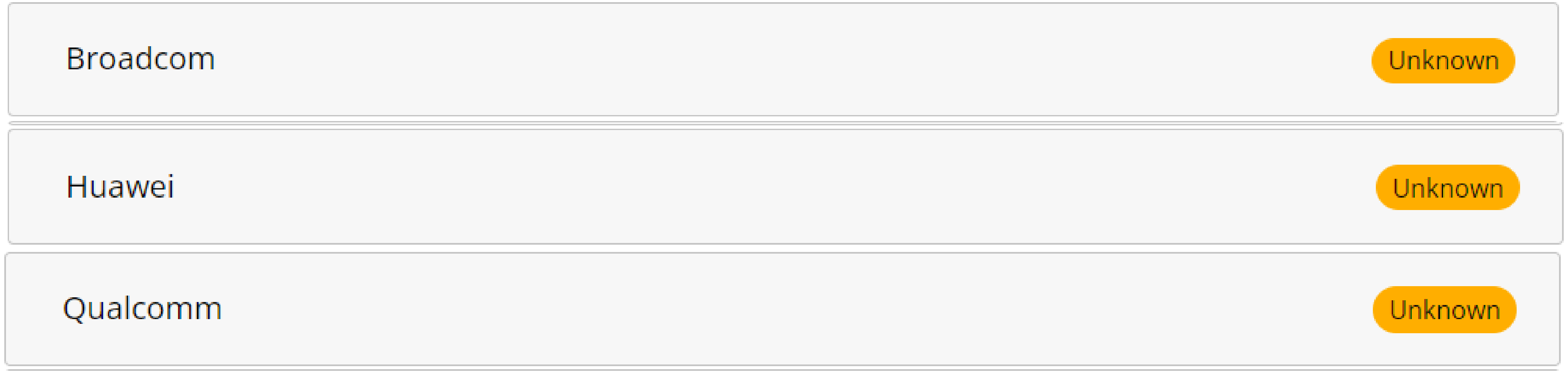
Disclosure Details

Disclosure Details

- Industry-wide disclosure process, with many parties involved.
  - Iván Arce handled it from Quarkslab's side.
  - Coordinated via US CERT/CC.
- CERT/CC granted access to the vulnerability report to **1600 vendors**.
  - Reason: several PC OEM and hardware vendors expressed interest in reaching out to other vendors up and down their supply chain.
- Google pushed the fix to a Chromium OS **public repository before embargo ended**.
- Huawei's OpenEuler Linux distribution made the vulnerability report available on its **public issue tracker**.

Disclosure Details

- Some hardware vendors reported that their products were not affected.
  - Hard to verify due to the lack of debugging/monitoring capabilities.
  - If they identified and fixed the bugs beforehand, they never reported them to TCG.
- Vulnerable status remains unknown for several hardware vendors (see *https://kb.cert.org/vuls/id/782720*)



Broadcom	Unknown
Huawei	Unknown
Qualcomm	Unknown

Part 6

Conclusions

Conclusions (1)

- Every TPM (either software or hardware implementations) whose firmware is based on the reference code published by the Trusted Computing Group is expected to be affected by these two vulnerabilities.
    - This includes the two main open source implementations available: Microsoft's and IBM's.

Conclusions (1)

- Every TPM (either software or hardware implementations) whose firmware is based on the reference code published by the Trusted Computing Group is expected to be affected by these two vulnerabilities.
  - This includes the two main open source implementations available: Microsoft's and IBM's.
- Although all affected TPMs share the exact same vulnerable function, the likeliness of successful exploitation depends on how the command buffer is implemented, and that part is left to each implementation.
  - Everyone seems to do it in a different way.

Conclusions (2)

- We were able to verify that these vulnerabilities were present in the software TPMs included in major desktop virtualization solutions, both free and non-free.
  - SWTPM (based on `libtpms`, used by QEMU) case looked dangerous (I haven't checked VirtualBox).

Conclusions (2)

- We were able to verify that these vulnerabilities were present in the software TPMs included in major desktop virtualization solutions, both free and non-free.
  - SWTPM (based on `libtpms`, used by QEMU) case looked dangerous (I haven't checked VirtualBox).
- Virtual TPMs available in the biggest cloud computing providers were also likely affected.
  - Google Cloud uses the IBM version of the reference implementation, which was affected.
  - Microsoft Azure is based on Microsoft's Hyper-V, which was affected.

Conclusions (3)

- We confirmed the OOB write in a Dell machine with a Nuvoton hardware TPM.
  - Dell Latitude E5570 with Nuvoton NPCT65x, firmware version 1.3.0.1
  - After triggering the bug, the chip would stop responding to further commands, and required a hard reboot of the computer to be operational again.

Conclusions (3)

- We confirmed the OOB write in a Dell machine with a Nuvoton hardware TPM.
  - Dell Latitude E5570 with Nuvoton NPCT65x, firmware version 1.3.0.1
  - After triggering the bug, the chip would stop responding to further commands, and required a hard reboot of the computer to be operational again.
- We expected most TPM hardware vendors to be affected too.
  - The lack of debugging capabilities in the TPM environment makes it harder to confirm the presence of vulnerabilities.

Conclusions (4)

- **Reference implementations deserve special attention, security-wise.**
  - Vulnerabilities in reference implementation code spread across diverse codebases, and may end up biting everyone.

Quarkslab

Questions?