UEFI Utility to Read TPM 2.0 PCRs

Finnbarr P. Murphy

(fpm@fpmurphy.com)

In a previous post, I discussed how to retrieve Platform Configuration Register (PCR) values from a discrete TPM (Trusted Platform Module) 1.2 chip (*dTPM 1.2*) and provided source code for a UEFI shell utility to display the digests from the first 16 PCRs. In this post, I discuss a number of key TPM 2.0 features and provide the source code for a UEFI shell utility to display the digests from the first 24 PCRs of a TPM 2.0 implementation.

What is driving the move to TPM 2.0? Simple, TPM 1.2 ((*ISO/IEC 11889*) only supports one hash algorithm, i.e. *SHA1*, and one asymmetric algorithm, i.e. *RSA*, and no block ciphers. This is no longer acceptable for a number of reasons including:

- SHA1 is being deprecated by NIST due to weakness.
- Standards organisations in different geographies are mandating different algorithms.
- Infrastructure asymmetric cryptography is migrating from RSA to ECC.

In support of the above, TPM 2.0 supports multiple algorithms including *RSA*, *ECC*, *ECC-DAA*, *ECDH*, *SHA-1*, *SHA-256*, *HMAC*, and *AES*. Manufacturers can add additional algorithms such as *SHA512* or *SM3* by registering algorithm IDs with TCG and implementing the algorithms in their product. The TCG Algorithm Registry "lists each algorithm assigned an identifier, allowing it to be unambiguously defined and referenced by other TCG specifications." For example, *TPM_ALG_SHA*, value 0x004, unambiguously denotes the SHA1 algorithm.

The TPM 2.0 specifications permit a TPM to have multiple banks (groupings) of PCRs.

- Banks do not have to have the same number of PCRs.
- A PCR can be a member of multiple banks.
- Within a bank, all PCRs are extended using the same algorithm.

Another wrinkle to retrieving TPM 2.0 PCR values is the fact that Trusted Computing Group (TCG) describes five different types of TPM 2.0 implementations: discrete TPMs (*dTPM*), firmware TPMs (*fTPM*), integrated TPMs, software TPMs, and virtual TPMs. Each of these can have different low level programming interfaces.

Discrete TPMs are semiconductor chips that implement TPM functionality in hardware to resist software patching and bugs and to provide tamper resistance. They are regarded as the most secure type of TPM and were the only type of implementation supported by the initial TPM specifications. Firmware TPMs are software-only solutions that run in a CPU's trusted execution environment. Such TPMs are vulnerable to software bugs in their implementation and in the trusted execution environment. Intel PTT (Platform Trust Technology) implements a fDTM 2.0 and is a popular TPM 2.0 solution in laptops as it eliminates a chip. See the section on TPM Implementations in this Wikipedia article for more information.

Between support for multiple algorithms and support for multiple PCR banks, reading TPM 2.0 PCR values is a lot more difficult to do than for TPM 1.2. Just look at the huge difference in SLOC (Lines of Source Code) and complexity between my *ShowPCR12* utility and the source code shown below.

Here is the source for a UEFI shell utility which outputs digests for the first 24 PCRs:


```
//
//  Copyright (c) 2017  Finnbarr P. Murphy.   All rights reserved.
//
//  Retrieve and print TPM 2.0 PCR digests
//
//  License: BSD License
//
//  Routines containing an underbar in the name were ported from Intel Open Source Technol
ogy
//  Center tpm2.0-tools and modified for UDK2015 environment and my requirements.
//
//  Original tpm2.0-tools license is:
//
//  Copyright (c) 2015, Intel Corporation
//
//  Redistribution and use in source and binary forms, with or without
//  modification, are permitted provided that the following conditions are met:
//
//      * Redistributions of source code must retain the above copyright notice,
//        this list of conditions and the following disclaimer.
//      * Redistributions in binary form must reproduce the above copyright
//        notice, this list of conditions and the following disclaimer in the
//        documentation and/or other materials provided with the distribution.
//      * Neither the name of Intel Corporation nor the names of its contributors
//        may be used to endorse or promote products derived from this software
//        without specific prior written permission.
//
//  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
//  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
//  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
//  DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE
//  FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
//  DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
//  SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
//  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
//  OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
//  OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
#include <Uefi.h>
#include <Library/UefiLib.h>
#include <Library/ShellCEntryLib.h>
#include <Library/ShellLib.h>
#include <Library/BaseMemoryLib.h>
#include <Library/UefiBootServicesTableLib.h>
#include <IndustryStandard/UefiTcgPlatform.h>
#include <Protocol/EfiShell.h>
#include <Protocol/LoadedImage.h>
#include <Protocol/TcgService.h>
#include <Protocol/Tcg2Protocol.h>
#include <IndustryStandard/UefiTcgPlatform.h>
#define UTILITY_VERSION L"0.8"
#define MAX_PCR 24
typedef struct {
    UINT64 count;
    TPML_DIGEST pcr_values[MAX_PCR];
} tpm2_pcrs;
typedef struct {
    TPML_PCR_SELECTION pcr_selections;
    tpm2_pcrs pcrs;
} pcr_context;
#pragma pack(1)
typedef struct {
    TPM2_COMMAND_HEADER   Header;
    TPML_PCR_SELECTION    PcrSelectionIn;
} TPM2_PCR_READ_COMMAND;
typedef struct {
    TPM2_RESPONSE_HEADER   Header;
    UINT32                 PcrUpdateCounter;
```

```
    TPML_PCR_SELECTION     PcrSelectionOut;
    TPML_DIGEST            PcrValues;
} TPM2_PCR_READ_RESPONSE;
#pragma pack()
typedef struct {
    TPMI_ALG_HASH alg;
    CHAR16 *desc;
    BOOLEAN supported;
} tpm2_algorithm;
tpm2_algorithm algs[] = {
    { TPM_ALG_SHA1, L"TPM_ALG_SHA1", FALSE },
    { TPM_ALG_SHA256, L"TPM_ALG_SHA256", FALSE },
    { TPM_ALG_SHA384, L"TPM_ALG_SHA384", FALSE },
    { TPM_ALG_SHA512, L"TPM_ALG_SHA512", FALSE },
    { TPM_ALG_SM3_256, L"TPM_ALG_SM3_256", FALSE },
    { TPM_ALG_NULL, L"TPM_ALG_UNKNOWN", FALSE }
};
// prototypes
EFI_STATUS Tpm2PcrRead ( TPML_PCR_SELECTION *, UINT32 *, TPML_PCR_SELECTION  *, TPML_DIGES
T *);
EFI_TCG2_PROTOCOL *Tcg2Protocol;
CONST CHAR16 *
Get_Algorithm_Name( TPMI_ALG_HASH alg_id)
{
    UINT32 i;
    for (i = 0; algs[i].alg != TPM_ALG_NULL; i++) {
        if (algs[i].alg == alg_id) {
            break;
        }
    }
    return algs[i].desc;
}
VOID
Set_PcrSelect_Bit( TPMS_PCR_SELECTION *s,
                   UINT32 pcr)
{
    s->pcrSelect[((pcr) / 8)] |= (1 << ((pcr) % 8));
}
VOID
Clear_PcrSelect_Bits( TPMS_PCR_SELECTION *s)
{
    s->pcrSelect[0] = 0;
    s->pcrSelect[1] = 0;
    s->pcrSelect[2] = 0;
}
VOID
Set_PcrSelect_Size( TPMS_PCR_SELECTION *s,
                    UINT8 size)
{
    s->sizeofSelect = size;
}
BOOLEAN
Is_PcrSelect_Bit_Set( TPMS_PCR_SELECTION *s,
                      UINT32 pcr)
{
    return (s->pcrSelect[((pcr) / 8)] &amp; (1 << ((pcr) % 8)));
}
BOOLEAN
Unset_PcrSections(TPML_PCR_SELECTION *s)
{
    UINT32 i, j;
    for (i = 0; i < s->count; i++) {
        for (j = 0; j < s->pcrSelections[i].sizeofSelect; j++) {
            if (s->pcrSelections[i].pcrSelect[j]) {
                return FALSE;
            }
        }
    }
```

```
    return TRUE;
}
VOID
Update_Pcr_Selections( TPML_PCR_SELECTION *s1,
                       TPML_PCR_SELECTION *s2)
{
    UINT32 i, j, k;
    for (j = 0; j < s2->count; j++) {
        for (i = 0; i < s1->count; i++) {
            if (s2->pcrSelections[j].hash != s1->pcrSelections[i].hash) {
                continue;
            }
            for (k = 0; k < s1->pcrSelections[i].sizeofSelect; k++) {
                s1->pcrSelections[i].pcrSelect[k] &amp;= ~s2->pcrSelections[j].pcrSelect[k
];
            }
        }
    }
}
VOID
Show_Pcr_Values( pcr_context *context)
{
    UINT32 vi = 0, di = 0, i;
    for (i = 0; i < context->pcr_selections.count; i++) {
        CONST CHAR16 *alg_name = Get_Algorithm_Name( context->pcr_selections.pcrSelections
[i].hash);
        Print(L"\nBank (Algorithm): %s (0x%04x)\n\n", alg_name,
                context->pcr_selections.pcrSelections[i].hash);
        for (UINT32 pcr_id = 0; pcr_id < MAX_PCR; pcr_id++) {
            if (!Is_PcrSelect_Bit_Set(&amp;context->pcr_selections.pcrSelections[i], pcr_i
d)) {
                continue;
            }
            if (vi >= context->pcrs.count || di >= context->pcrs.pcr_values[vi].count) {
                Print(L"ERROR: Trying to output PCR values but nothing more to output\n");
                return;
            }
            Print(L"[%02d] ", pcr_id);
            for (UINT32 k = 0; k < context->pcrs.pcr_values[vi].digests[di].size; k++)
                Print(L" %02x", context->pcrs.pcr_values[vi].digests[di].buffer[k]);
            Print(L"\n");
            if (++di < context->pcrs.pcr_values[vi].count) {
                continue;
            }
            di = 0;
            if (++vi < context->pcrs.count) {
                continue;
            }
        }
        Print(L"\n");
    }
}
BOOLEAN
Read_Pcr_Values( pcr_context *context)
{
    TPML_PCR_SELECTION pcr_selection_tmp;
    TPML_PCR_SELECTION pcr_selection_out;
    UINT32 pcr_update_counter;
    EFI_STATUS Status;
    CopyMem(&amp;pcr_selection_tmp, &amp;context->pcr_selections, sizeof(pcr_selection_tmp)
);
    context->pcrs.count = 0;
    do {
        Status = Tpm2PcrRead( &amp;pcr_selection_tmp,
                              &amp;pcr_update_counter,
                              &amp;pcr_selection_out,
                              &amp;context->pcrs.pcr_values[context->pcrs.count]);
        if (EFI_ERROR (Status)) {
```

```
            Print(L"ERROR: Tpm2PcrRead failed [%d]\n", Status);
            return FALSE;
        }
        // unmask pcrSelectionOut bits from pcrSelectionIn
        Update_Pcr_Selections(&amp;pcr_selection_tmp, &amp;pcr_selection_out);
        // goto step 2 if pcrSelctionIn still has bits set
    } while (++context->pcrs.count < MAX_PCR &amp;&amp; !Unset_PcrSections(&amp;pcr_select
ion_tmp));
    // hack - this needs to be re-worked
    if (context->pcrs.count >= MAX_PCR &amp;&amp; !Unset_PcrSections(&amp;pcr_selection_tm
p)) {
        Print(L"ERROR: Reading PCRs. Too much PCRs found [%d]\n", context->pcrs.count);
        return FALSE;
    }
    return TRUE;
}
//
// Modified from original UDK2015 SecurityPkg routine
//
EFI_STATUS
Tpm2PcrRead( TPML_PCR_SELECTION  *PcrSelectionIn,
             UINT32              *PcrUpdateCounter,
             TPML_PCR_SELECTION  *PcrSelectionOut,
             TPML_DIGEST         *PcrValues)
{
    EFI_STATUS             Status;
    TPM2_PCR_READ_COMMAND  SendBuffer;
    TPM2_PCR_READ_RESPONSE RecvBuffer;
    UINT32                 SendBufferSize;
    UINT32                 RecvBufferSize;
    UINTN                  Index;
    TPML_DIGEST            *PcrValuesOut;
    TPM2B_DIGEST           *Digests;
    // Construct the TPM2 command
    SendBuffer.Header.tag = SwapBytes16(TPM_ST_NO_SESSIONS);
    SendBuffer.Header.commandCode = SwapBytes32(TPM_CC_PCR_Read);
    SendBuffer.PcrSelectionIn.count = SwapBytes32(PcrSelectionIn->count);
    for (Index = 0; Index < PcrSelectionIn->count; Index++) {
        SendBuffer.PcrSelectionIn.pcrSelections[Index].hash =
            SwapBytes16(PcrSelectionIn->pcrSelections[Index].hash);
        SendBuffer.PcrSelectionIn.pcrSelections[Index].sizeofSelect =
            PcrSelectionIn->pcrSelections[Index].sizeofSelect;
        CopyMem (&amp;SendBuffer.PcrSelectionIn.pcrSelections[Index].pcrSelect,
            &amp;PcrSelectionIn->pcrSelections[Index].pcrSelect,
            SendBuffer.PcrSelectionIn.pcrSelections[Index].sizeofSelect);
    }
    SendBufferSize = sizeof(SendBuffer.Header) + sizeof(SendBuffer.PcrSelectionIn.count) +
        sizeof(SendBuffer.PcrSelectionIn.pcrSelections[0]) * PcrSelectionIn->count;
    SendBuffer.Header.paramSize = SwapBytes32 (SendBufferSize);
    RecvBufferSize = sizeof (RecvBuffer);
    Status = Tcg2Protocol->SubmitCommand( Tcg2Protocol,
                                          SendBufferSize,
                                          (UINT8 *)&amp;SendBuffer,
                                          RecvBufferSize,
                                          (UINT8 *)&amp;RecvBuffer);
    if (EFI_ERROR (Status)) {
        Print(L"ERROR: SubmitCommand failed [%d]\n", Status);
        return Status;
    }
    if (RecvBufferSize < sizeof (TPM2_RESPONSE_HEADER)) {
        Print(L"ERROR: RecvBufferSize [%x]\n", RecvBufferSize);
        return EFI_DEVICE_ERROR;
    }
    if (SwapBytes32(RecvBuffer.Header.responseCode) != TPM_RC_SUCCESS) {
        Print(L"ERROR Tpm2 ResponseCode [%x]\n", SwapBytes32(RecvBuffer.Header.responseCod
e));
        return EFI_NOT_FOUND;
    }
```



```
    // Response - PcrUpdateCounter
    if (RecvBufferSize < sizeof (TPM2_RESPONSE_HEADER) + sizeof(RecvBuffer.PcrUpdateCounte
r)) {
        Print(L"Tpm2PcrRead - RecvBufferSize Error - %x\n", RecvBufferSize);
        return EFI_DEVICE_ERROR;
    }
    *PcrUpdateCounter = SwapBytes32(RecvBuffer.PcrUpdateCounter);
    // Response - PcrSelectionOut
    if (RecvBufferSize < sizeof (TPM2_RESPONSE_HEADER) + sizeof(RecvBuffer.PcrUpdateCounte
r) +
        sizeof(RecvBuffer.PcrSelectionOut.count)) {
        Print(L"Tpm2PcrRead - RecvBufferSize Error - %x\n", RecvBufferSize);
        return EFI_DEVICE_ERROR;
    }
    PcrSelectionOut->count = SwapBytes32(RecvBuffer.PcrSelectionOut.count);
    if (RecvBufferSize < sizeof (TPM2_RESPONSE_HEADER) + sizeof(RecvBuffer.PcrUpdateCounte
r)
        + sizeof(RecvBuffer.PcrSelectionOut.count)
        + sizeof(RecvBuffer.PcrSelectionOut.pcrSelections[0]) * PcrSelectionOut->count) {
        Print(L"Tpm2PcrRead - RecvBufferSize Error - %x\n", RecvBufferSize);
        return EFI_DEVICE_ERROR;
    }
    for (Index = 0; Index < PcrSelectionOut->count; Index++) {
        PcrSelectionOut->pcrSelections[Index].hash =
            SwapBytes16(RecvBuffer.PcrSelectionOut.pcrSelections[Index].hash);
        PcrSelectionOut->pcrSelections[Index].sizeofSelect =
            RecvBuffer.PcrSelectionOut.pcrSelections[Index].sizeofSelect;
        CopyMem (&amp;PcrSelectionOut->pcrSelections[Index].pcrSelect,
                 &amp;RecvBuffer.PcrSelectionOut.pcrSelections[Index].pcrSelect,
                 PcrSelectionOut->pcrSelections[Index].sizeofSelect);
    }
    // Response - return digests in PcrValue
    PcrValuesOut = (TPML_DIGEST *)((UINT8 *)&amp;RecvBuffer + sizeof (TPM2_RESPONSE_HEADER)
                   + sizeof(RecvBuffer.PcrUpdateCounter) + sizeof(RecvBuffer.PcrSelectionO
ut.count)
                   + sizeof(RecvBuffer.PcrSelectionOut.pcrSelections[0]) * PcrSelectionOut
->count);
    PcrValues->count = SwapBytes32(PcrValuesOut->count);
    Digests = PcrValuesOut->digests;
    for (Index = 0; Index < PcrValues->count; Index++) {
        PcrValues->digests[Index].size = SwapBytes16(Digests->size);
        CopyMem (&amp;PcrValues->digests[Index].buffer, &amp;Digests->buffer,
                 PcrValues->digests[Index].size);
        Digests = (TPM2B_DIGEST *)((UINT8 *)Digests + sizeof(Digests->size)
                  + PcrValues->digests[Index].size);
    }
    return EFI_SUCCESS;
}
VOID
Init_Pcr_Selection( pcr_context *context,
                    TPMI_ALG_HASH alg)
{
    TPML_PCR_SELECTION *s = (TPML_PCR_SELECTION *) &amp;(context->pcr_selections);
    s->count = 1;
    s->pcrSelections[0].hash = alg;
    Set_PcrSelect_Size(&amp;s->pcrSelections[0], 3);
    Clear_PcrSelect_Bits(&amp;s->pcrSelections[0]);
    for (UINT32 pcr_id = 0; pcr_id < MAX_PCR; pcr_id++) {
        Set_PcrSelect_Bit(&amp;s->pcrSelections[0], pcr_id);
    }
}
BOOLEAN
CheckForTpm12()
{
    EFI_STATUS Status = EFI_SUCCESS;
    EFI_TCG2_PROTOCOL *TcgProtocol;
    EFI_GUID gEfiTcgProtocolGuid = EFI_TCG_PROTOCOL_GUID;
    Status = gBS->LocateProtocol( &amp;gEfiTcgProtocolGuid,
```

```
                              NULL,
                              (VOID **) &amp;TcgProtocol);
    if (EFI_ERROR (Status)) {
        return FALSE;
    }
    return TRUE;
}
VOID
Usage(CHAR16 *Str)
{
    Print(L"Usage: %s [--version] [algorithm]\n", Str);
    Print(L"\nPossibly supported algorithms:\n");
    for (UINT32 i = 0; algs[i].alg != TPM_ALG_NULL ; i++) {
        if (algs[i].alg == TPM_ALG_SHA1) {
            Print(L"  %s (default)\n", algs[i].desc);
        } else {
            Print(L"  %s\n", algs[i].desc);
        }
    }
    Print(L"\n");
}
INTN
EFIAPI
ShellAppMain(UINTN Argc, CHAR16 **Argv)
{
    EFI_STATUS Status = EFI_SUCCESS;
    EFI_GUID gEfiTcg2ProtocolGuid = EFI_TCG2_PROTOCOL_GUID;
    TPMI_ALG_HASH alg = TPM_ALG_SHA1;    // default algorithm
    pcr_context context;
    if (Argc > 2) {
        Usage(Argv[0]);
        return Status;
    }
    if (Argc == 2) {
        if ((!StrCmp(Argv[1], L"--version")) ||
            (!StrCmp(Argv[1], L"-V"))) {
            Print(L"Version: %s\n", UTILITY_VERSION);
            return Status;
        }
        if ((!StrCmp(Argv[1], L"--help")) ||
            (!StrCmp(Argv[1], L"-h"))) {
            Usage(Argv[0]);
            return Status;
        }
        if ((!StrCmp(Argv[1], L"SHA1")) ||
            (!StrCmp(Argv[1], L"sha1"))) {
            alg = TPM_ALG_SHA1;
        } else if ((!StrCmp(Argv[1], L"SHA256")) ||
            (!StrCmp(Argv[1], L"sha256"))) {
            alg = TPM_ALG_SHA256;
        } else if ((!StrCmp(Argv[1], L"SHA384")) ||
            (!StrCmp(Argv[1], L"sha384"))) {
            alg = TPM_ALG_SHA384;
        } else if ((!StrCmp(Argv[1], L"SHA512")) ||
            (!StrCmp(Argv[1], L"sha512"))) {
            alg = TPM_ALG_SHA512;
        } else if ((!StrCmp(Argv[1], L"SM3")) ||
            (!StrCmp(Argv[1], L"sm3"))) {
            alg = TPM_ALG_SM3_256;
        } else {
            Usage(Argv[0]);
            return Status;
        }
    }
    Status = gBS->LocateProtocol( &amp;gEfiTcg2ProtocolGuid,
                              NULL,
                              (VOID **) &amp;Tcg2Protocol);
    if (EFI_ERROR (Status)) {
```



```
        if (CheckForTpm12()) {
            Print(L"ERROR: Platform configured for TPM 1.2, not TPM 2.0\n");
        } else {
            Print(L"ERROR: Failed to locate EFI_TCG2_PROTOCOL [%d]\n", Status);
        }
        return Status;
    }
    Init_Pcr_Selection(&amp;context, alg);
    if (Read_Pcr_Values(&amp;context))
        Show_Pcr_Values(&amp;context);
    return Status;
}
```

I named the utility *ShowPCR20* (the TPM 1.2 version was named *ShowPCR12*) but you can name it anything you like in your build environment. As before, I assume that you are familiar with UEFI internals and the UDK2015 build environment. Sorry but I will NOT help you with build issues; I simply do not have time to do so.

Along with my earlier brief overview of TPM 2.0 support for multiple algorithms and multiple PCR banks, the following structure *typedefs* may help you to better understand the interrelationship between multiple algorithms and multiple PCR banks in the above code:

```
typedef struct {
  TPM2_COMMAND_HEADER       Header;
  TPML_PCR_SELECTION        PcrSelectionIn;
} TPM2_PCR_READ_COMMAND;

typedef struct {
  TPM_ST tag;
  UINT32    paramSize;
  TPM_CC    commandCode;
} TPM2_COMMAND_HEADER;

typedef struct {
   UINT32              count
   TPMS_PCR_SELECTION  pcrSelections[HASH_COUNT]
} TMPL_PCR_SELECTION;

typedef struct {
   TPMI_ALG_HASH  hash;
   UINT8          sizeofSelect;
   BYTE           pcrSelect[PCR_SELECT_MAX];
} TPMS_PCR_SELECTION;
```

If you are familiar with retrieving TPM 1.2 PCR values, you will immediately notice that retrieving TPM 2.0 PCRs is a lot more difficult because of the number of structures involved, some with similar structure and member names. I would also point out that the UDK2015 source is not very useful for gaining an understanding of TPM 2.0. It is better to examine the Intel TPM2 Tools source and the TPM 2.0 specifications.

Here is a suitable UDK2015 build *.INF* for this utility:

```
[Defines]
  INF_VERSION                    = 0x00010006
  BASE_NAME                      = ShowPCR20
  FILE_GUID                      = 4ea87c51-7395-4ccd-0355-747111f3ce51
  MODULE_TYPE                    = UEFI_APPLICATION
  VERSION_STRING                 = 0.1
  ENTRY_POINT                    = ShellCEntryLib
  VALID_ARCHITECTURES            = X64
```



```
[Sources]
  ShowPCR20.c

[Packages]
  MdePkg/MdePkg.dec
  ShellPkg/ShellPkg.dec

[LibraryClasses]
  ShellCEntryLib
  ShellLib
  BaseLib
  BaseMemoryLib
  UefiLib

[Protocols]

[BuildOptions]

[Pcd]
```

Here is sample output from the utility when I ran it on an Lenovo T450 with the security chip setting set to *Intel PTT* (an fTPM 2.0):

```
FS1> ShowPCR20.efi --version
Version: 0.8

FS1> ShowPCR20.efi --help
Usage: FS1:\ShowPCR20.efi [--version] [algorithm]

Possibly supported algorithms:
  TPM_ALG_SHA1 (default)
  TPM_ALG_SHA256
  TPM_ALG_SHA384
  TPM_ALG_SHA512
  TPM_ALG_SM3_256

FS1> ShowPCR20.efi

Bank (Algorithm): TPM_ALG_SHA1 (0x0004)

[00]  D3 35 8E 13 88 EF 41 2C 26 07 A0 68 BA D9 71 6C 07 DA 7B 50
[01]  61 28 6E 8F DD 95 05 DC 13 EF 5F 71 E0 21 EA 8D 9E F5 87 D6
[02]  B2 A8 3B 0E BF 2F 83 74 29 9A 5B 2B DF C3 1E A9 55 AD 72 36
[03]  B2 A8 3B 0E BF 2F 83 74 29 9A 5B 2B DF C3 1E A9 55 AD 72 36
[04]  13 9F E0 36 DA 84 86 FA A9 D8 DB A1 4C B9 8B E3 6F 0E 6D EE
[05]  45 A3 23 38 2B D9 33 F0 8E 7F 0E 25 6B C8 24 9E 40 95 B1 EC
[06]  A9 CD BE 97 0A A5 DD AA A8 C2 07 28 A0 EB 16 44 DC 4D 50 FE
[07]  DD 38 9D 19 91 FA 9A D1 D9 D9 CF 41 B4 94 8E 39 39 DD 7D BC
[08]  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[09]  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[10]  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[11]  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[12]  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[13]  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[14]  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[15]  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[16]  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[17]  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
[18]  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
[19]  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
[20]  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
[21]  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
[22]  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
[23]  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```



```
FS1> ShowPCR20.efi SHA256
ERROR: Reading PCRs. Too much PCRs found [24]

FS1>
```

This code has only been tested on my Lenovo T450 which has a dTPM 1.2 and an Intel PTT fTPM 2.0. As a result, I am sure that there are a fairly large number of errors in the code and missed edge cases. Please let me know of any problems you find within the source code.

As an aside, the Intel PTT fTPM 2.0 is implemented in the Intel *Management Engine* (ME), also known as *Manageability Engine*, which resides in what is currently known as the Platform Configuration Hub (PCH). Igor Skochinsky of Hex-Rays gave a very interesting presentation on the internals of Intel ME at RECON 2014 Montreal. The presentation is entitled *Intel ME Secrets – Hidden code in your chipset and how to discover what exactly it does* and can easily be found on the Internet. It is a *must read* for anybody interested in Intel ME internals including PTT and the fTPM.

Enjoy!

