Advanced x86:
BIOS and System Management Mode Internals
UEFI Reverse Engineering

Xeno Kovah && Corey Kallenberg

LegbaCore, LLC

All materials are licensed under a Creative Commons “Share Alike” license.

http://creativecommons.org/licenses/by-sa/3.0/

You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

Under the following conditions:

Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

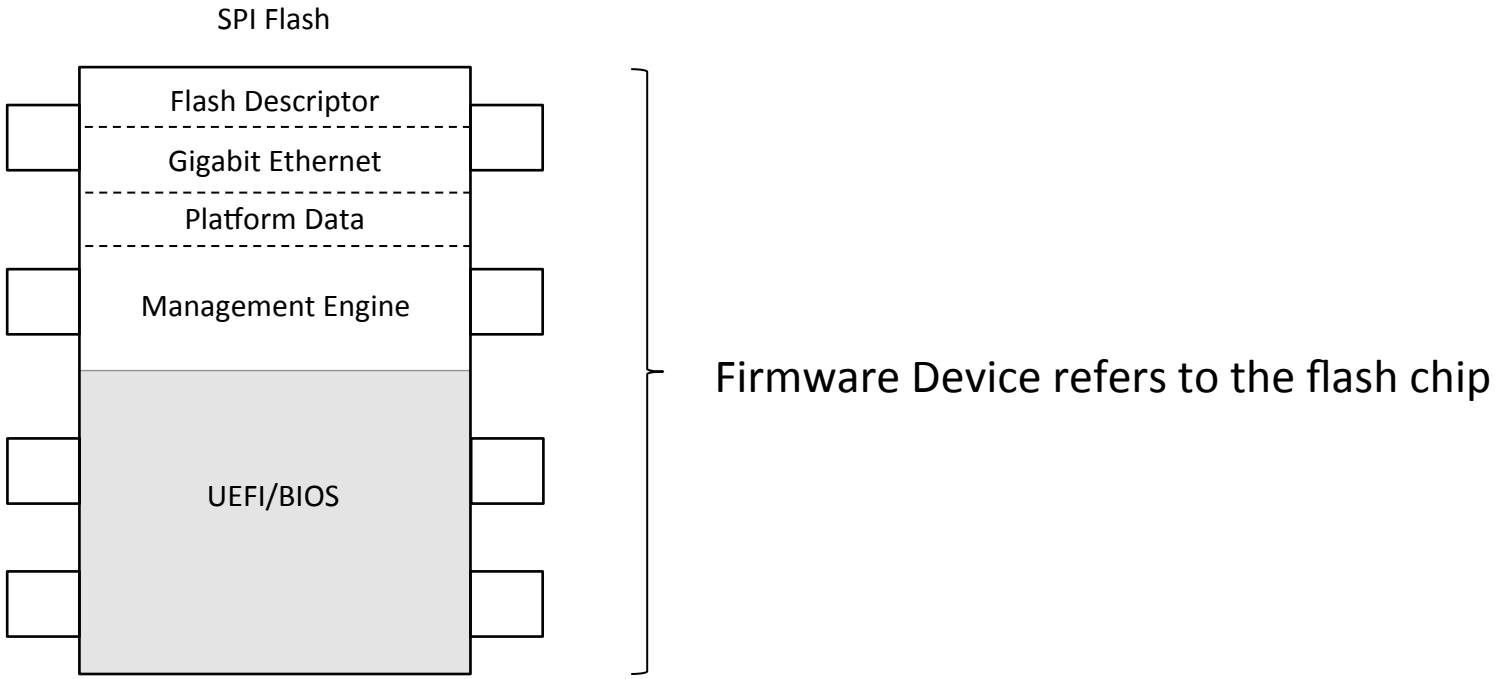
Attribution condition: You must indicate that derivative work
"Is derived from John Butterworth & Xeno Kovah’s ’Advanced Intel x86: BIOS and SMM’ class posted at http://opensecuritytraining.info/IntroBIOS.html”

And the people yelled:

Simmer down y'all.
I reckon what ya best do is…

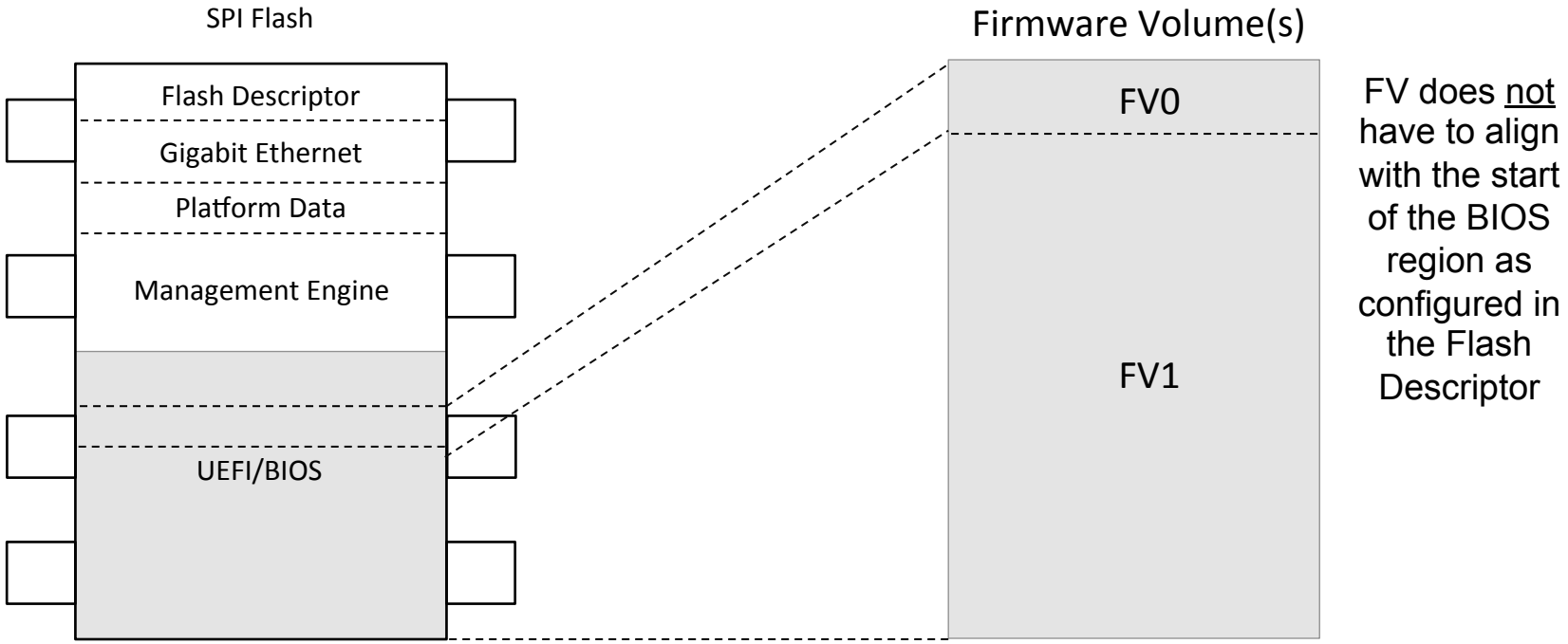
- Find some subset of interesting code
  - You *could* search for B/D/F address of interest
- But better is to narrow down what you want to look at, by slicing and dicing the firmware filesystem with one of:
- EFIPWN
  - https://github.com/G33KatWork/EFIPWN
- UEFITool
  - https://github.com/LongSoft/UEFITool
- UEFI Firmware Parser
  - https://github.com/theopolis/uefi-firmware-parser
  - We're not going to cover this for now, since I haven't built it on Windows yet

Firmware Storage



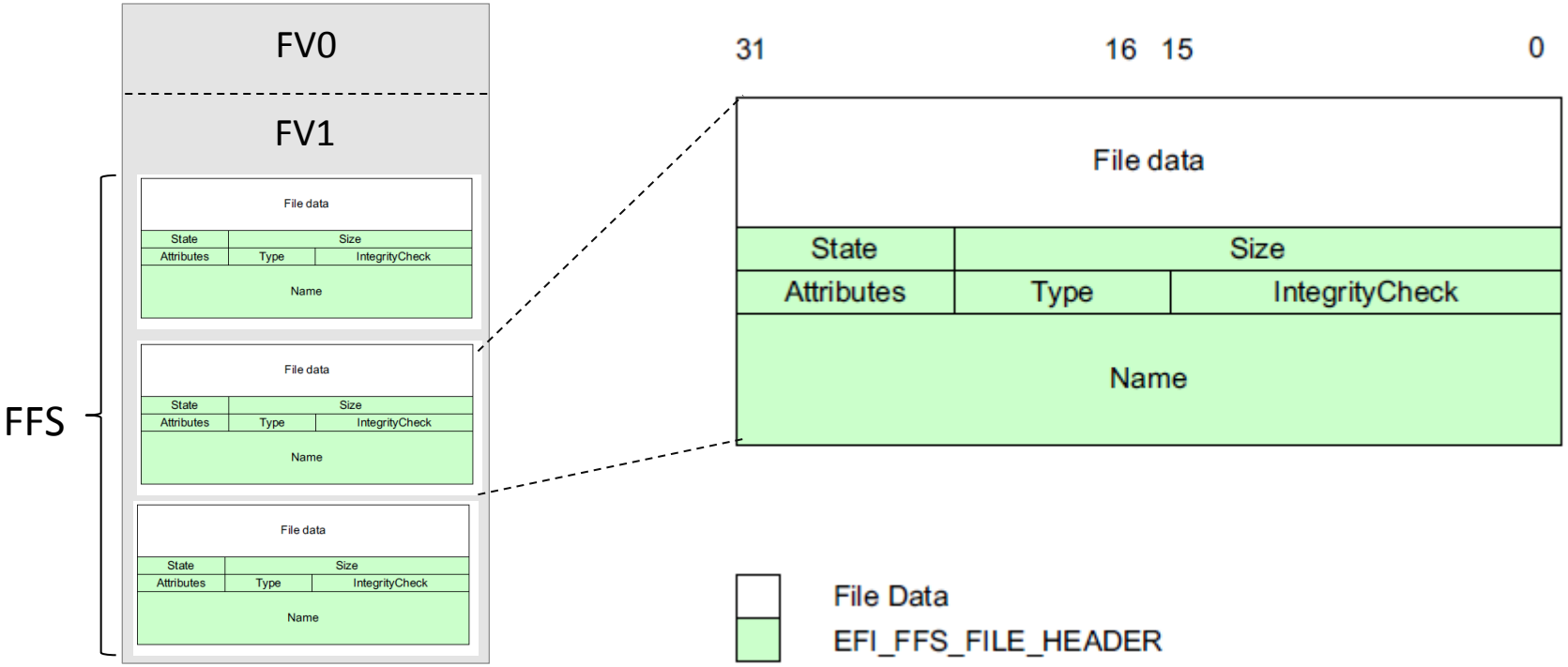
- UEFI utilizes the physical flash device as a storage repository
- Comprised of 4 basic components:
  - Firmware Device
  - Firmware Volume
  - Firmware File System
  - Firmware Files

Firmware Volumes (FVs)



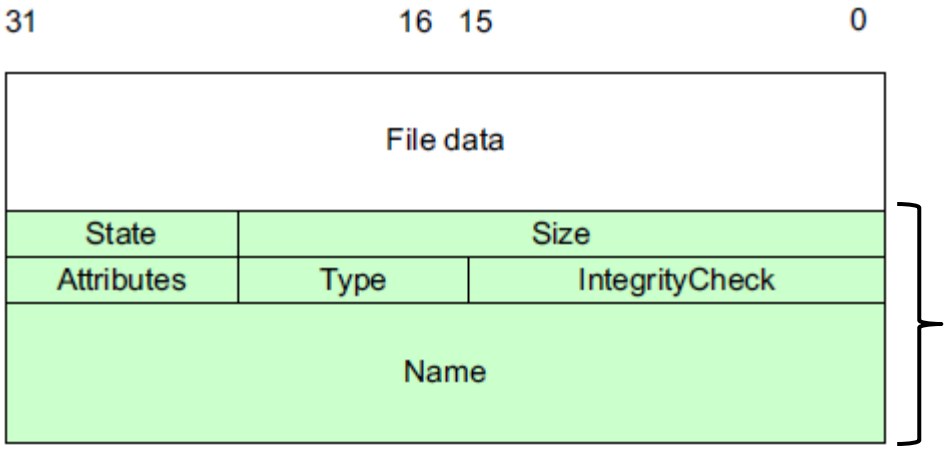
- A Firmware Device is a physical component such as a flash chip.
- We mostly care about Firmware Volumes (FVs)
- We often see separate volumes for PEI vs. DXE code
  - And occasional “duplicate” volumes for restore-from-backup
- FVs can contain multiple firmware volumes (nesting)
- FVs are organized into a Firmware File System (FFS)
- The base unit of a FFS is a file

Firmware File System (FFS)



- FVs are organized into a Firmware File System (FFS)
- A FFS describes the organization of files within the FV
- The base unit of a FFS is a file
- Files can be further subdivided into sections

Firmware Files




```
typedef struct {
  EFI_GUID                  Name;
  EFI_FFS_INTEGRITY_CHECK   IntegrityCheck;
  EFI_FV_FILETYPE           Type;
  EFI_FFS_FILE_ATTRIBUTES   Attributes;
  UINT8                     Size[3];
  EFI_FFS_FILE_STATE        State;
} EFI_FFS_FILE_HEADER;
```

File Data
EFI_FFS_FILE_HEADER

- We mostly care about file sections that are in PE (Portable Executable) file format
  - Alternatively can be a TE (Terse Executable) which is a “minimalist” PE

Oh, how interesting! My BIOS uses "Windows" executables? I know how to analyze those!

Yay Standardization!

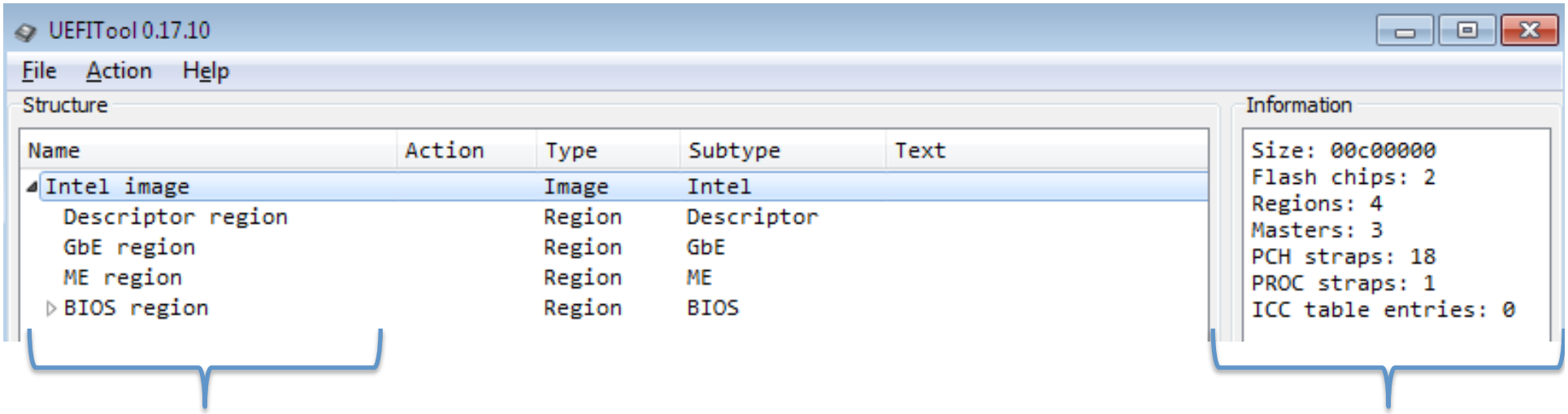
A standard way of putting together the firmware filesystem, with nice human readable names, makes it easier for me to find my way around to the likely locations I want to attack

A standard way of putting together the firmware filesystem, with nice human readable names, makes it easier for me to understand the context of what might have been attacked if I see a difference there

UEFITool/UEFIExtract

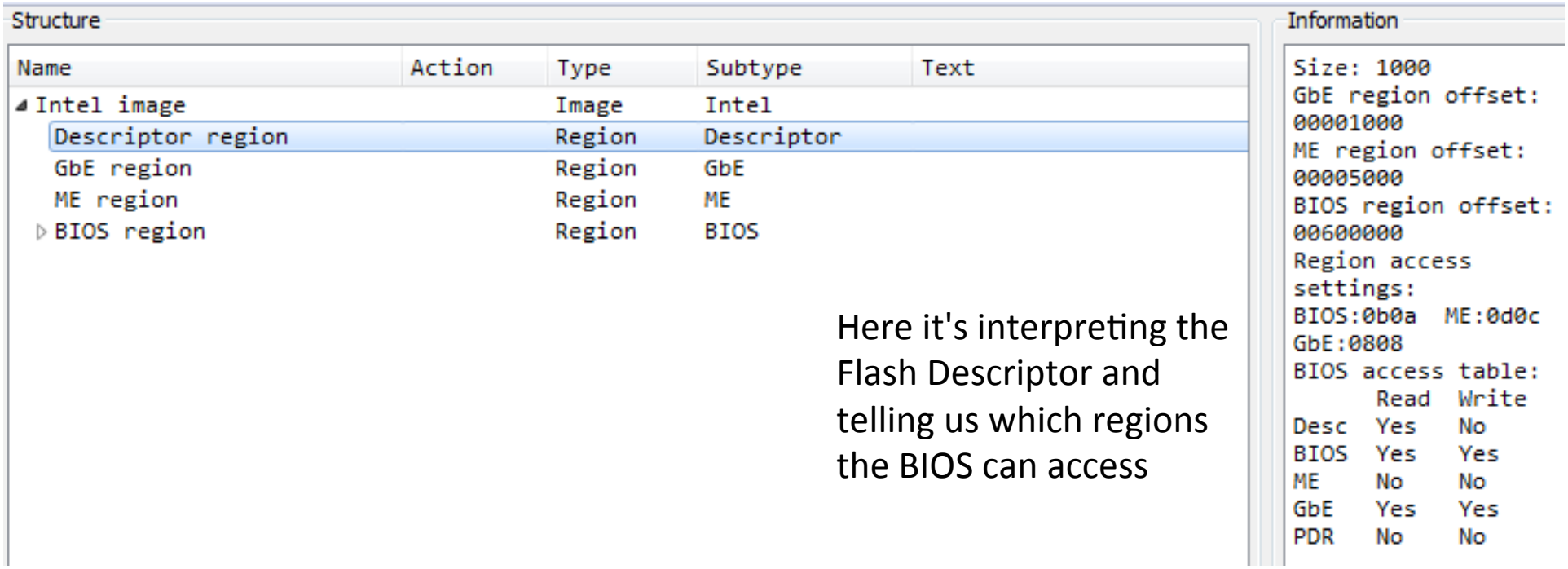
- The best and most up-to-date firmware filesystem parser

Go to File->Open and select the file dump (I selected the "e6430A03.bin")



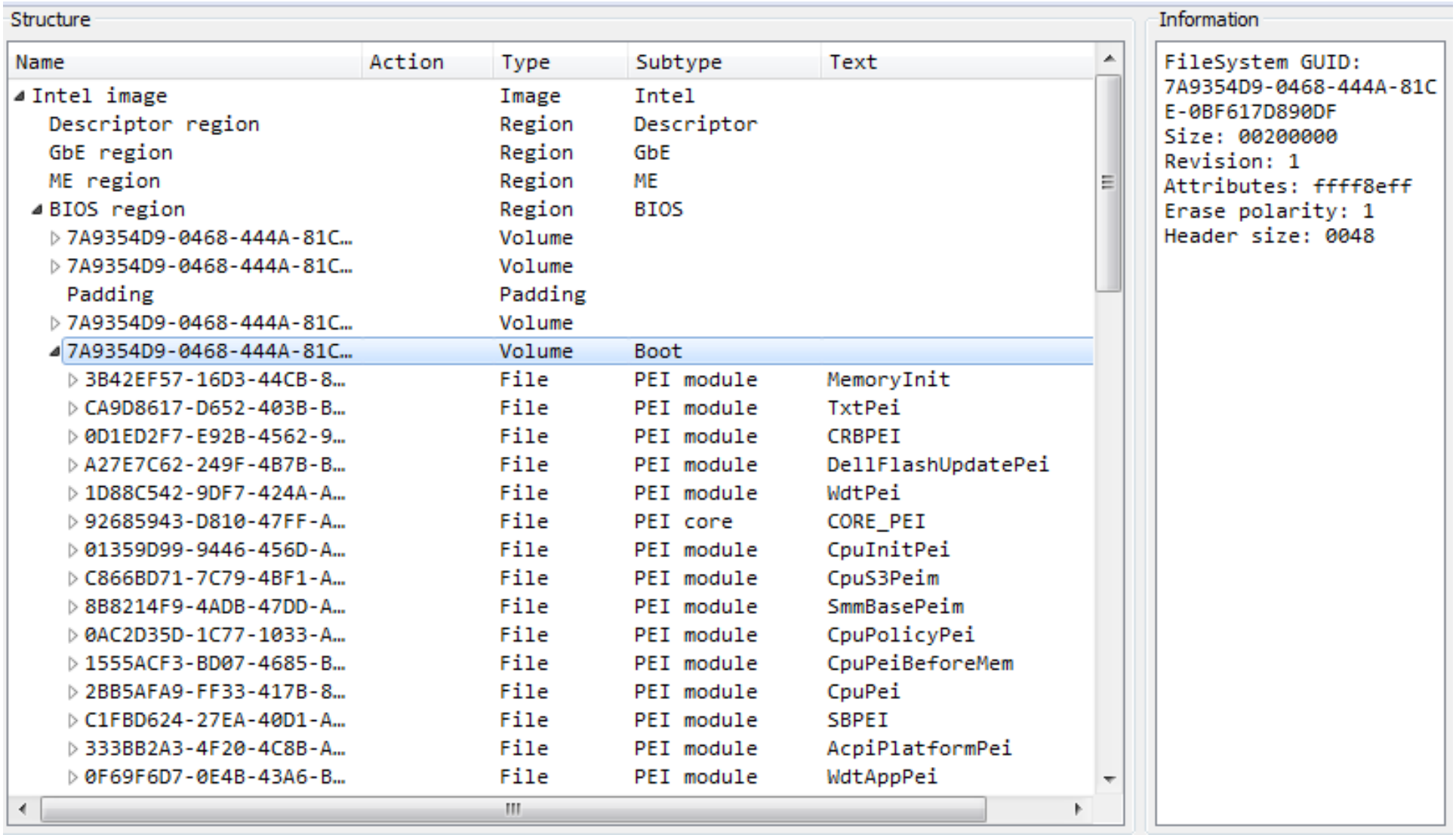
Navigation by expanding portions here

Parsed metadata here



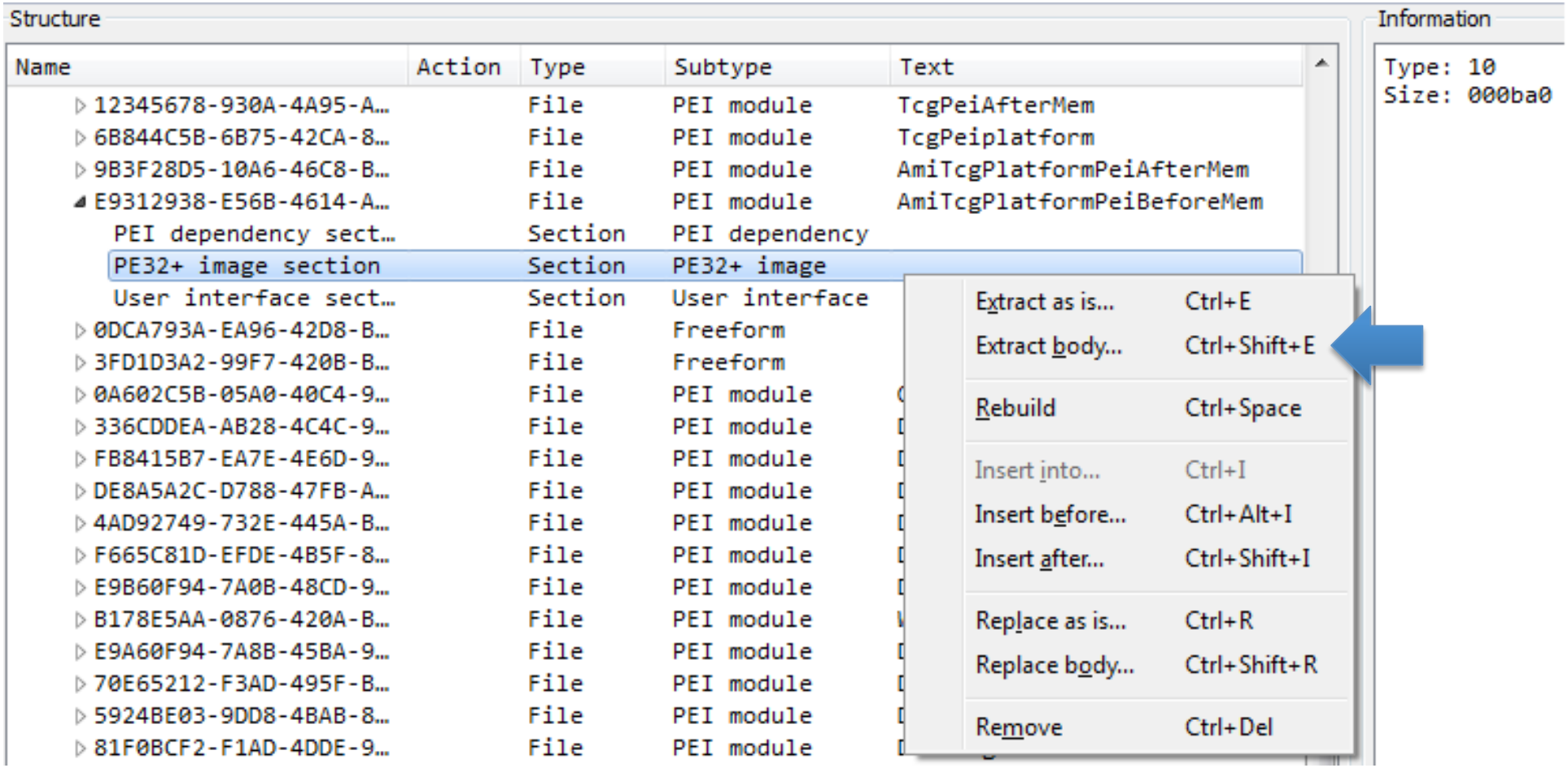
Here it's interpreting the Flash Descriptor and telling us which regions the BIOS can access

This volume holds a bunch of PEIMs (and the one above it a bunch of DXE drivers.)



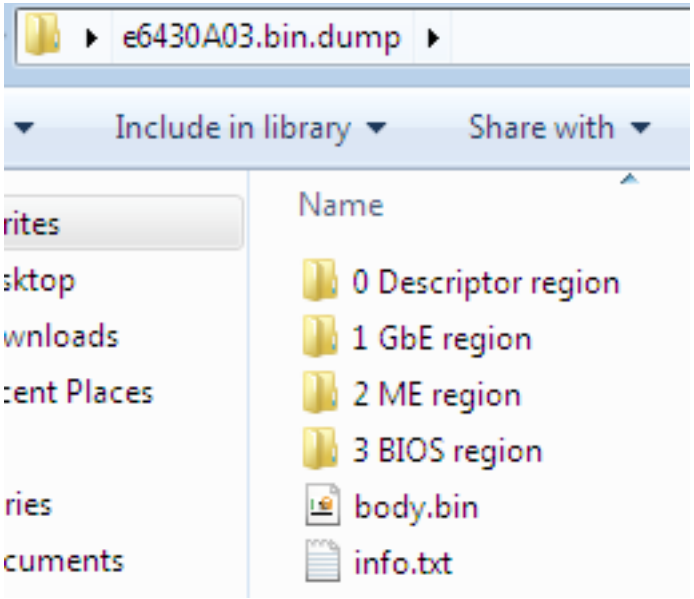
"AmgTcgPlatformPeiBeforeMem" is the PEIM we're going to be interested in shortly

To get a well-formed PE file, we extract it by right clicking and selecting "Extract body"



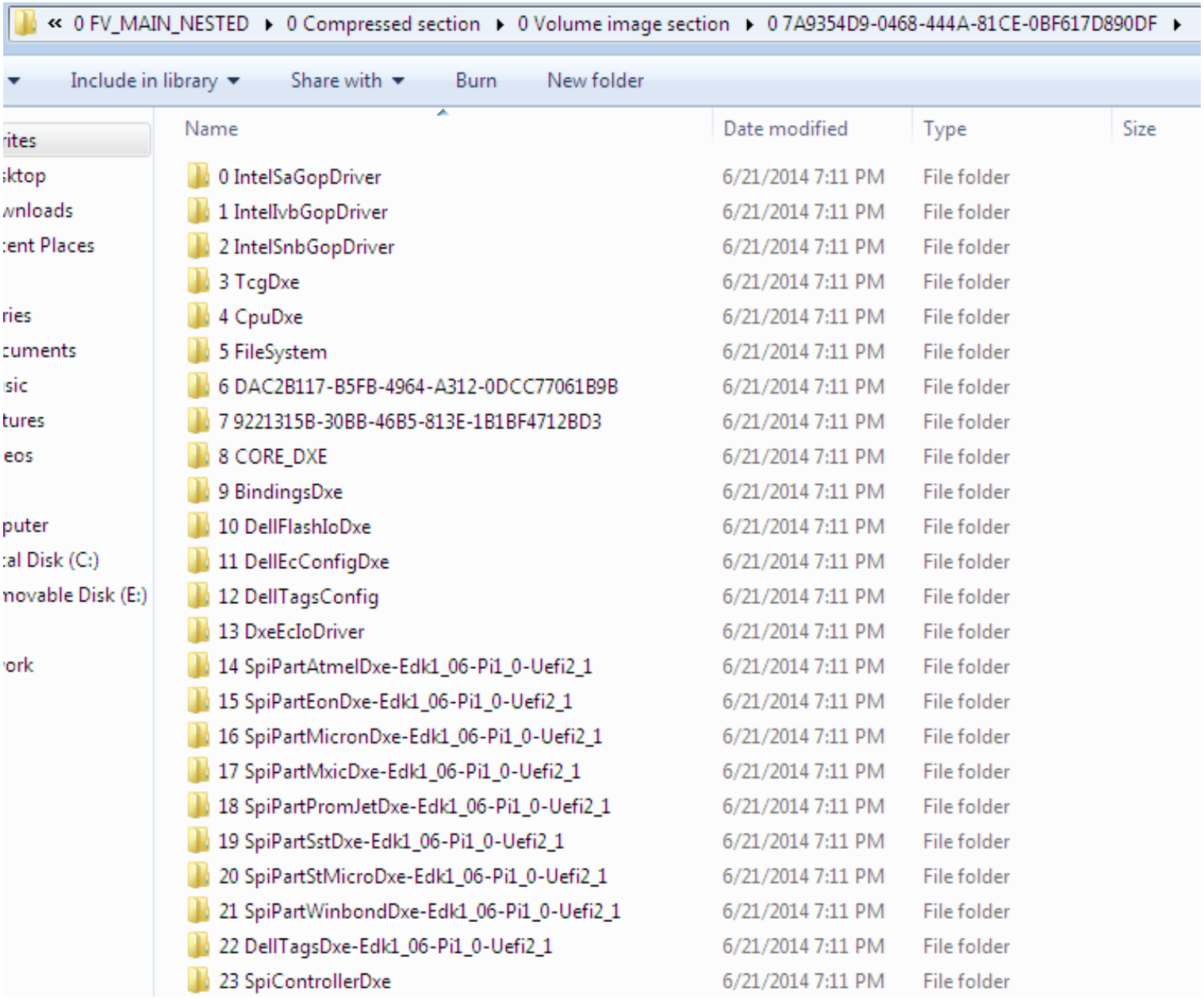
UEFIExtract is a simple command line tool that just dumps everything out to the filesystem instead of making it navigable from a GUI

```
C:\Users\student\Desktop>UEFIExtract.exe
UEFIExtract 0.2

Usage: uefiextract imagefile


C:\Users\student\Desktop>UEFIExtract.exe C:\Users\student\Desktop\e6430A03.bin
parseRegion: ME region version is unknown, it can be damaged
parseVolume: 17088572-377F-44EF-8F4E-B09FFF46A070, unaligned file
```



The metadata will be stored off to the side in .txt files

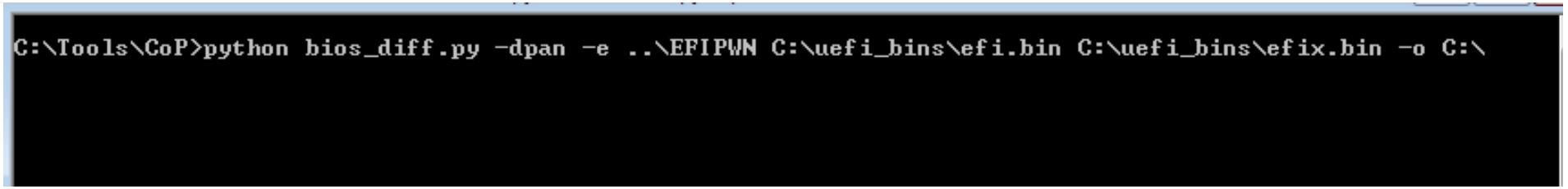
This is good if you want to search all the files for a pattern. But it's less easy to navigate if you want to just get a single file (in that case just use the GUI)



Identifying Changes in BIOS (bios_diff.py)

- So as we know, Copernicus provides us the full dump of the BIOS flash
  - Repeated from previous: Copernicus maintains the FLA offsets for each region by reading even those which the CPU/BIOS master has no permissions to read (like the Management Engine, typically)
  - Any BIOS dump should work as long as it's a UEFI BIOS (structured for better parsing)
- Comparing BIOS dumps over a period of time can provide change detection
- How this differs from observing the TPM PCR registers is this:
- When a PCR tells you a change has been made, it cannot tell you where the change has been made
- Bios_diff.py uses the decomposition capability of EFIPWN to tell us the particular module(s) in which the change(s) is/are located

Identifying Changes in BIOS (bios_diff.py)



- This script uses EFIPWN to parse and diff the modules between two BIOS dumps
- EFIPWN decomposes the BIOS into its firmware volumes (FVs) and then decomposes each into the files/modules that comprise it
- In this example we're analyzing an earlier "known-good" BIOS with one which we notice has changed
  - We took a known good and purposefully made a small change in the "suspicious" one

Identifying Changes in BIOS (bios_diff.py)

```
C:\Tools\CoP>python bios_diff.py -dpan -e C:\EFIPWN-sam\EFIPWN "F:\UEFI Binaries\e6430A03.bin" "F:\U
FI Binaries\e6430A03_haxed.bin" -o .
Differing file found:
.\e6430A03.bin\fv3\e9312938-e56b-4614-a252-cf7d2f377e26\PE32_73 (AmiTcgPlatformPeiBeforeMem)
7 unique bytes out of 2976
1036,1042
PE Information:
Section .text
RVA 0x40c
VA 0xffe6d090
.\e6430A03_haxed.bin\fv3\e9312938-e56b-4614-a252-cf7d2f377e26\PE32_73 (AmiTcgPlatformPeiBeforeMem)
7 unique bytes out of 2976
1036,1042
PE Information:
Section .text
RVA 0x40c
VA 0xffe6d090
```

- The script has found a difference located in firmware volume 3
- Some files/modules have user-friendly names and if this is the case the script outputs this name
- AmiTcgPlatformPeiBeforeMem
- Tcg could be Trusted Computing Group and this is likely a PEIM that executes before memory is established

Identifying Changes in BIOS (bios_diff.py)

```
C:\Tools\CoP>python bios_diff.py -dpan -e C:\EFIPWN-sam\EFIPWN "F:\UEFI Binaries\e6430A03.bin" "F:\U
FI Binaries\e6430A03_haxed.bin" -o .
Differing file found:
.\e6430A03.bin\fv3\e9312938-e56b-4614-a252-cf7d2f377e26\PE32_73 (AmiTcgPlatformPeiBeforeMem)
7 unique bytes out of 2976
1036,1042
PE Information:
Section .text
RVA 0x40c
VA 0xffe6d090
.\e6430A03_haxed.bin\fv3\e9312938-e56b-4614-a252-cf7d2f377e26\PE32_73 (AmiTcgPlatformPeiBeforeMem)
7 unique bytes out of 2976
1036,1042
PE Information:
Section .text
RVA 0x40c
VA 0xffe6d090
```

- If more than 1 diff is found they will all be listed here in this manner
- In this case it is just a single diff found
- Diff was found at offset 0x40C in the file “AmiTcgPlatformPeiBeforeMem”
- The length of the diff is 7 bytes

Identifying Changes in BIOS (bios_diff.py)

```
C:\Tools\CoP>python bios_diff.py -dpan -e C:\EFIPWN-sam\EFIPWN "F:\UEFI Binaries\e6430A03.bin" "F:\U
FI Binaries\e6430A03_haxed.bin" -o .
Differing file found:
.\e6430A03.bin\fv3\e9312938-e56b-4614-a252-cf7d2f377e26\PE32_73 (AmiTcgPlatformPeiBeforeMem)
7 unique bytes out of 2976
1036,1042
PE Information:
Section .text
RVA 0x40c
VA 0xffe6d090
.\e6430A03_haxed.bin\fv3\e9312938-e56b-4614-a252-cf7d2f377e26\PE32_73 (AmiTcgPlatformPeiBeforeMem)
7 unique bytes out of 2976
1036,1042
PE Information:
Section .text
RVA 0x40c
VA 0xffe6d090
```

- Files in the UEFI Flash File System are in the PE format (or TE [Terse Executable], which is a minimalist PE file)
  - But still PE
- For this reason we can identify whether diffs are located in the .data or .text (code) sections of a given file
  - In this case the change occurs in the code section

Identifying Changes in BIOS (bios_diff.py)

```
C:\Tools\CoP>python bios_diff.py -dpan -e C:\EFIPWN-sam\EFIPWN "F:\UEFI Binaries\e6430A03.bin" "F:\U
FI Binaries\e6430A03_haxed.bin" -o .
Differing file found:
.\e6430A03.bin\fv3\e9312938-e56b-4614-a252-cf7d2f377e26\PE32_73 (AmiTcgPlatformPeiBeforeMem)
7 unique bytes out of 2976
1036,1042
PE Information:
Section .text
RVA 0x40c
VA 0xffe6d090
.\e6430A03_haxed.bin\fv3\e9312938-e56b-4614-a252-cf7d2f377e26\PE32_73 (AmiTcgPlatformPeiBeforeMem)
7 unique bytes out of 2976
1036,1042
PE Information:
Section .text
RVA 0x40c
VA 0xffe6d090
```

- Also from the PE file we can get the Virtual Address of the change in the file
- From this we can derive both the Flash Linear Address of the change on the serial flash (provided the size of the BIOS region) and therefore its location in mapped high-memory
- The output also identifies the Relative-Virtual Address (RVA), which is the segment offset from the start of the PE file

Identifying Changes in BIOS (bios_diff.py)

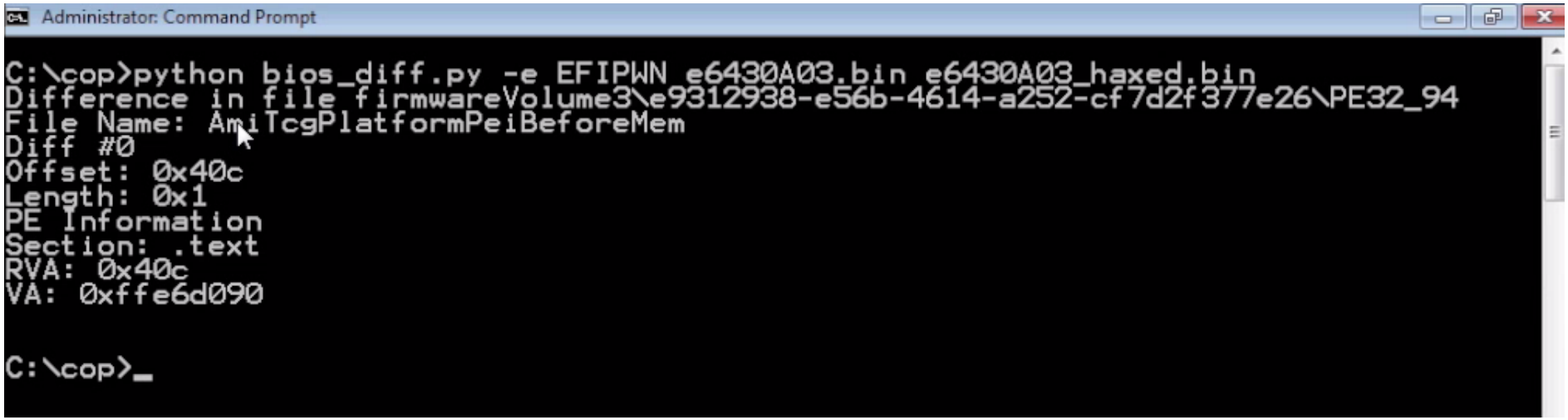
```
C:\Tools\CoP>python bios_diff.py -dpan
FI Binaries\e6430A03_haxed.bin" -o .
Differing file found:
.\e6430A03.bin\fv3\e9312938-e56b-4614-a
7 unique bytes out of 2976
1036,1042
PE Information:
RVA 0x40c
VA 0xffe6d090
```

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00A6CC80	A4	0B	00	10	4D	5A	00	00	00	00	00	00	00	00	00	00	¤...MZ.....
00A6CC90	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00A6CCA0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00A6CCB0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00A6CCC0	C8	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	È.....
00A6CCD0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....

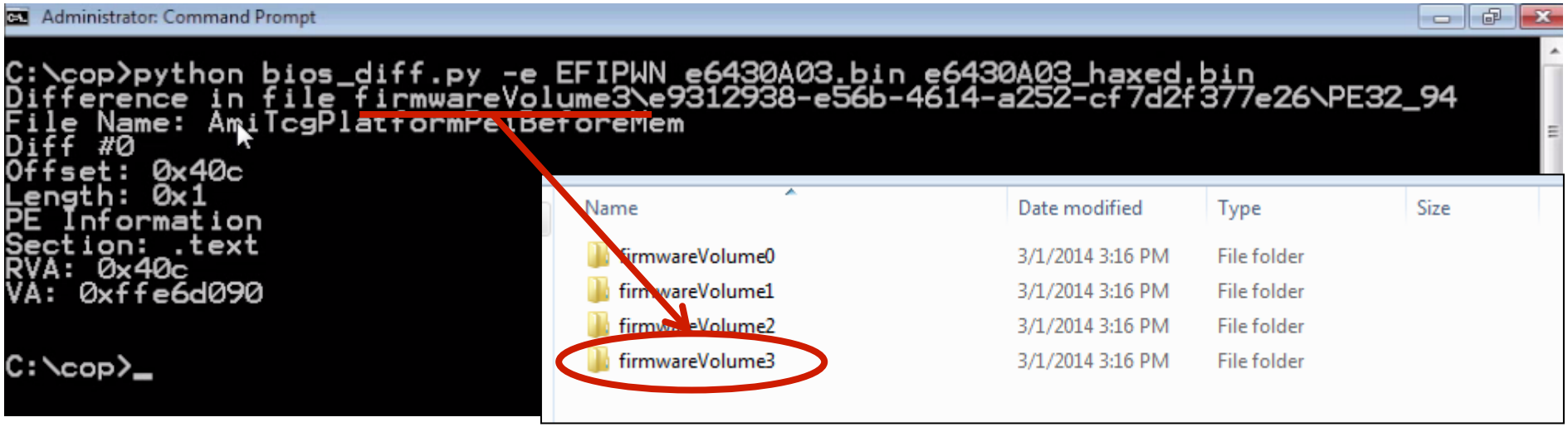
- We can use the VA and RVA information to locate this PE file in the BIOS hex dump
- VA – RVA = beginning of PE file
- But first let's convert that VA to a flash linear address:
- FFFF_FFFFh – FFE6_D090h = 19_2F6Fh
- <.bin size> - 19_2F6Fh = BF_FFFFh - 19_2F6Fh = A6_D090h
- A6_D090h – 40C = **A6_CC84h**

Analyzing UEFI Files with IDA
(Search for “MITRE Copernicus Analyzing BIOS Differences with IDA Pro”)

Analyzing UEFI Files

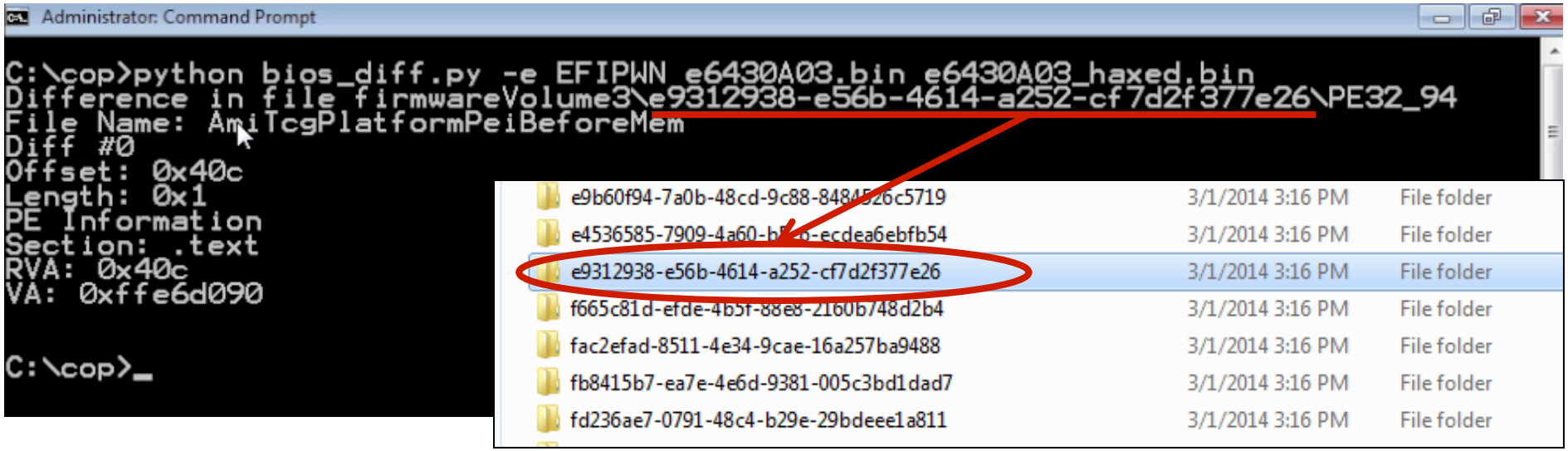


- Following our example of finding a “diff” across multiple BIOS, let’s find out how to analyze the change using IDA
- This should strike a sharp contrast to trying to analyze a legacy BIOS which does not follow public standards
  - Not to say they don’t have internal standards, just that those standards are not public
- The free version of IDA will be adequate for these purposes



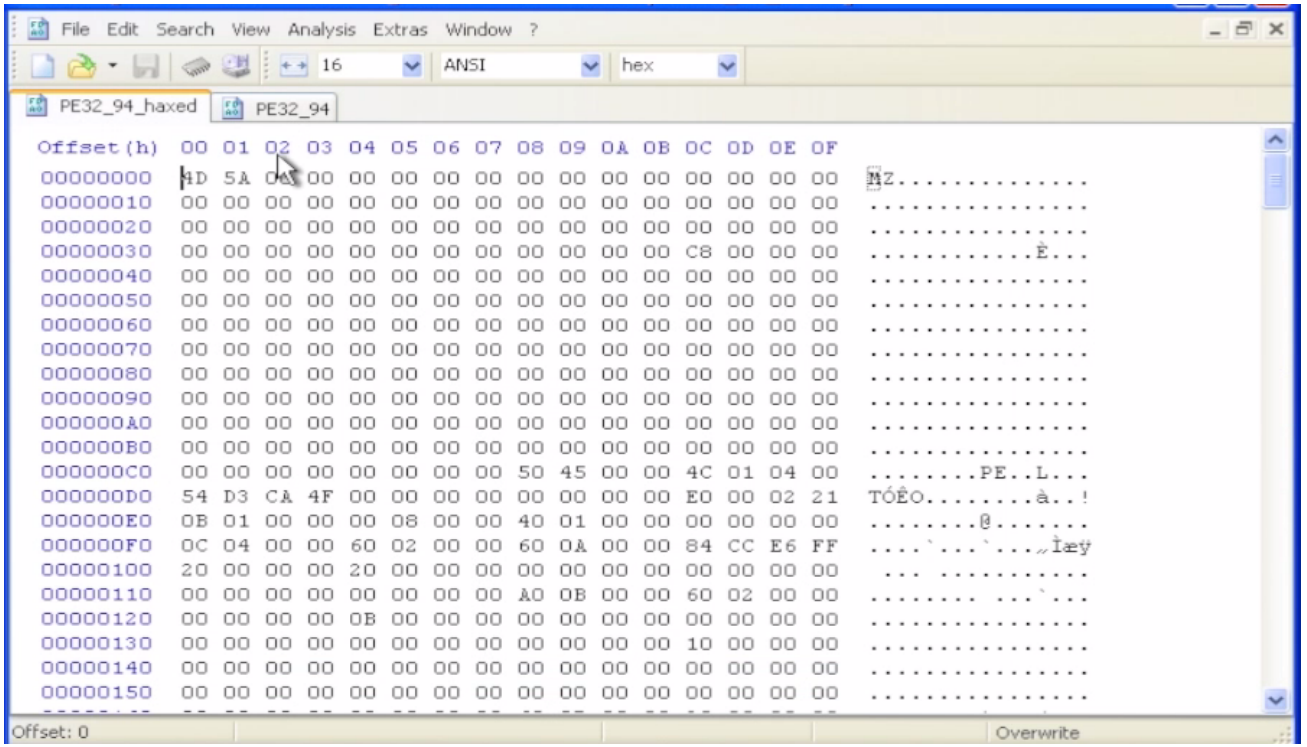
- The first step having identified a change between two BIOS dumps is to first locate the specific files in which the change(s) were detected
- In our example, the changes occur in Firmware Volume 3
- Find the directory where EFIPWN decomposed the UEFI binary and go to firmwareVolume3

Analyzing UEFI Files



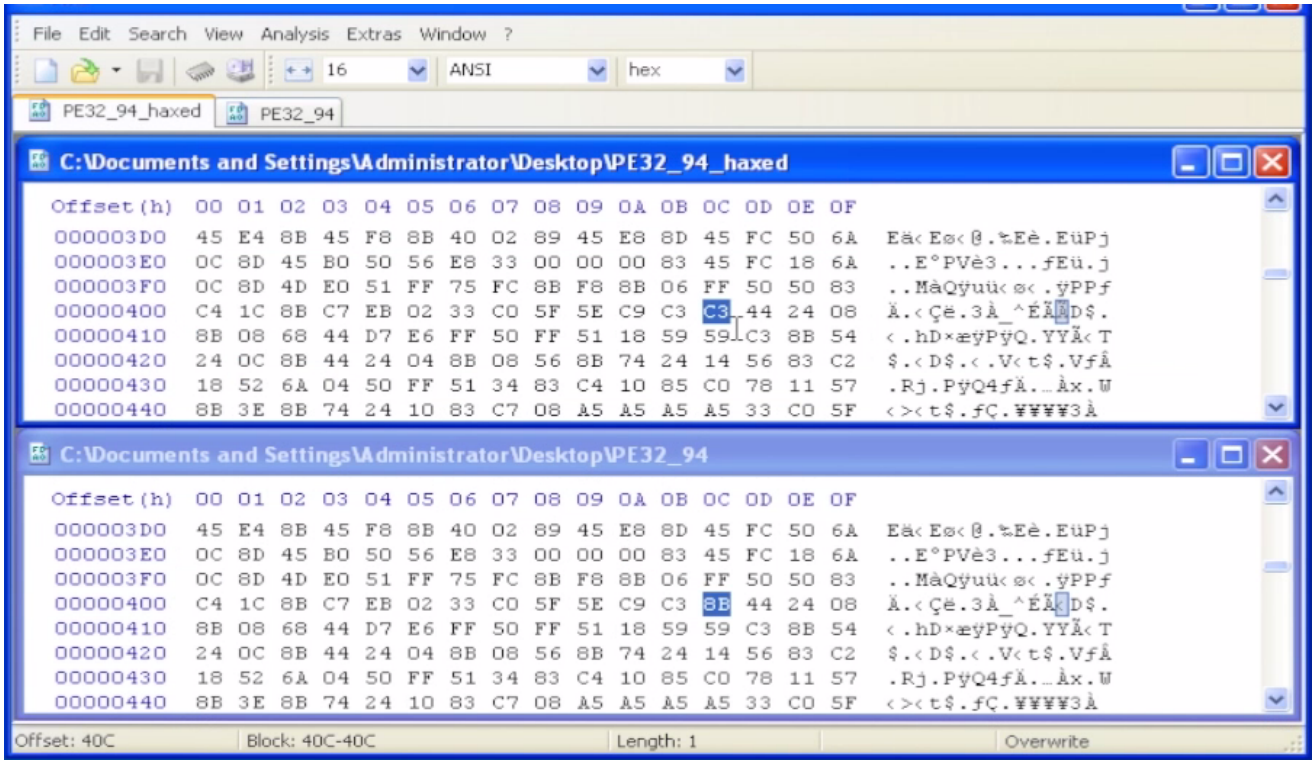
- Inside the firmwareVolume3 directory is a directory listing of GUIDS
- Find the GUID in which this diff was detected
- In this case it is GUID:
  - e9312938-e56b-4614-a252-cf7d2f377e26
- Inside this directory you will find the PE32_94 file which contains the file that has changed
- You can locate both of these files in this manner: the previous one which is assumed to be good, and the new one in which the change has been observed

Quick look in hex editor



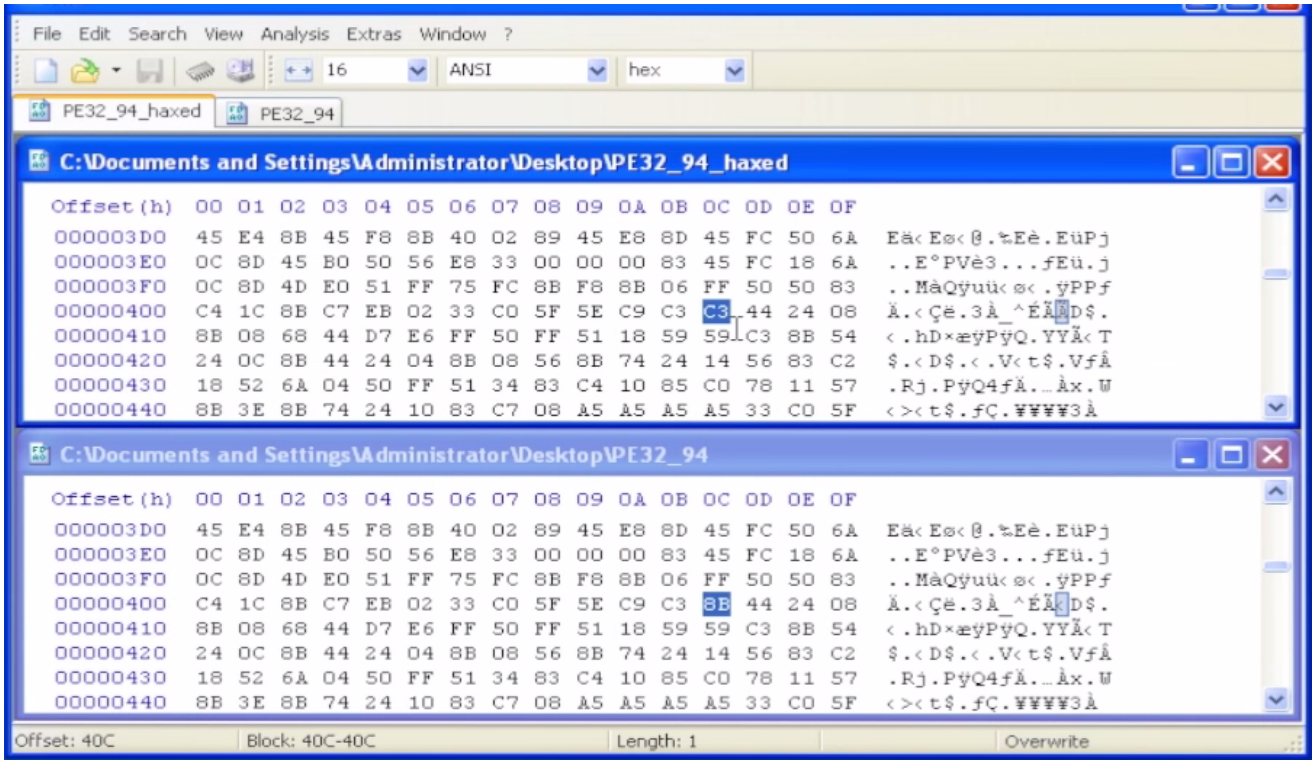
- One of the first things you can do upon acquiring both files is to observe them in a hex editor
- HxD allows you to easily perform binary comparisons between 2 files (Analysis > File-Compare > Compare, and then select the 2 files you want to compare)

Quick look in hex editor



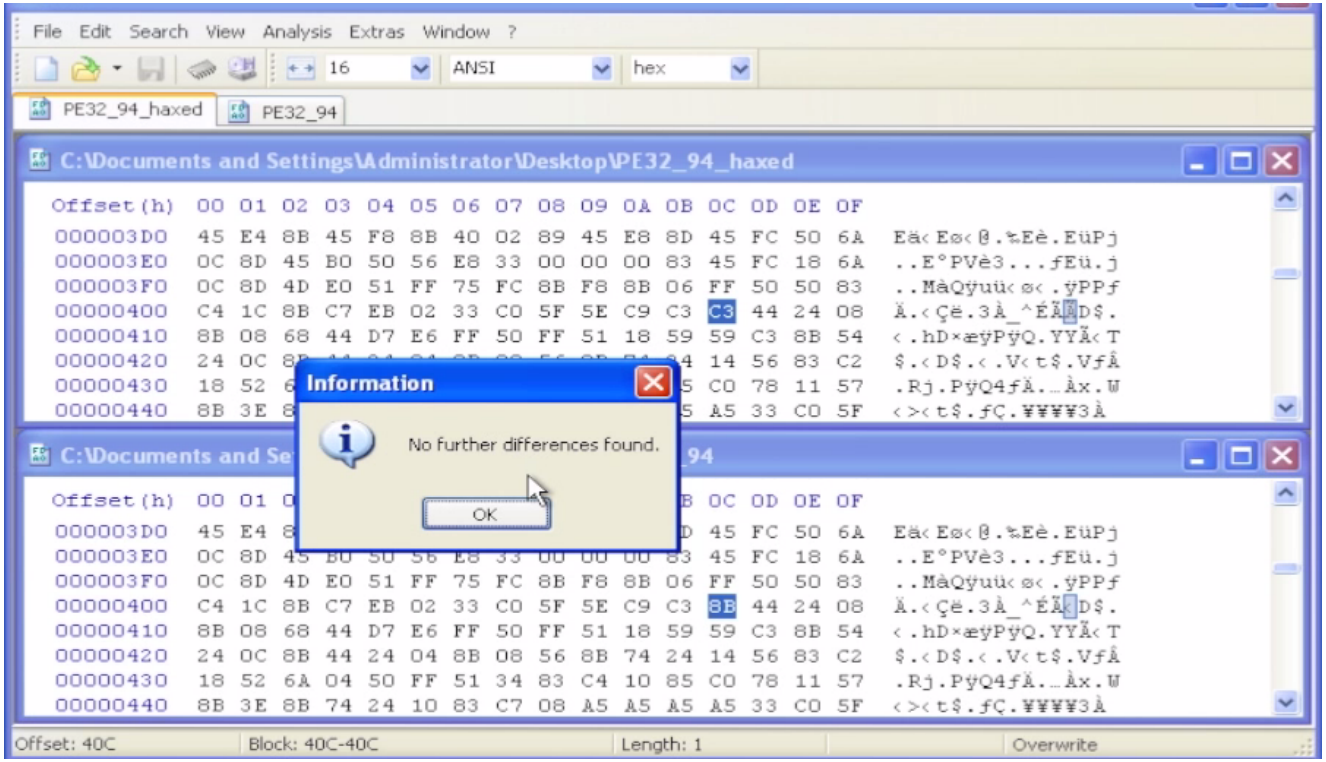
- HxD's file comparison compares each file in parallel and highlights each byte that differs
- It's a quick way to "eyeball" changes which have been detected
- This is less helpful when the file-sizes differ and the area where you want to analyze the change occurs at an offset other than where it usually does

Quick look in hex editor



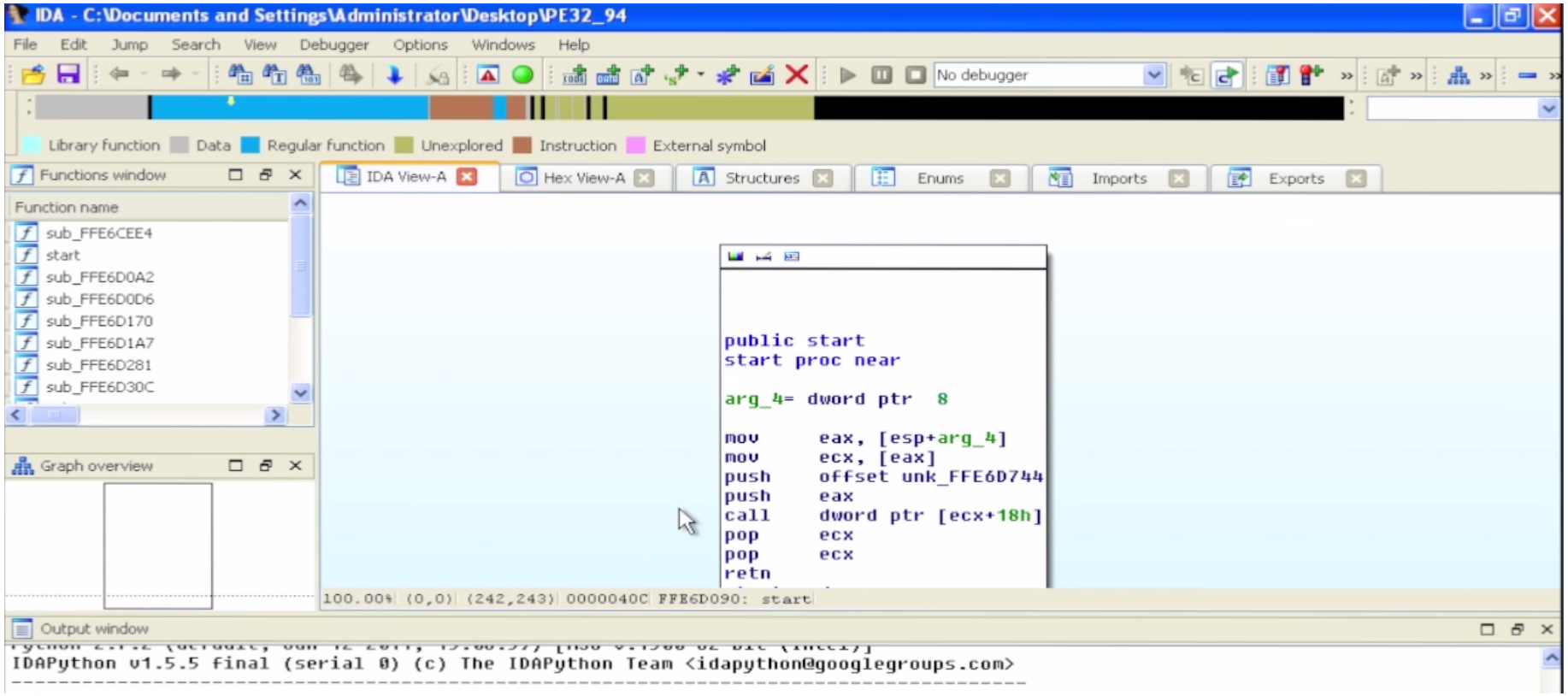
- In this simple example, the “haxed” version of the PE file has opcode 0xC3 at offset 0x40C while the original file has 0x8B
- Those who are familiar with the x86 instruction set may recognize the 0xC3 opcode as the RET (return) instruction
- Note that at the bottom of the HxD window it shows the file offset of the highlighted diff byte (“Block 40C-40C”)
- This corresponds to the information outputted by our bios_diff.py

Quick look in hex editor



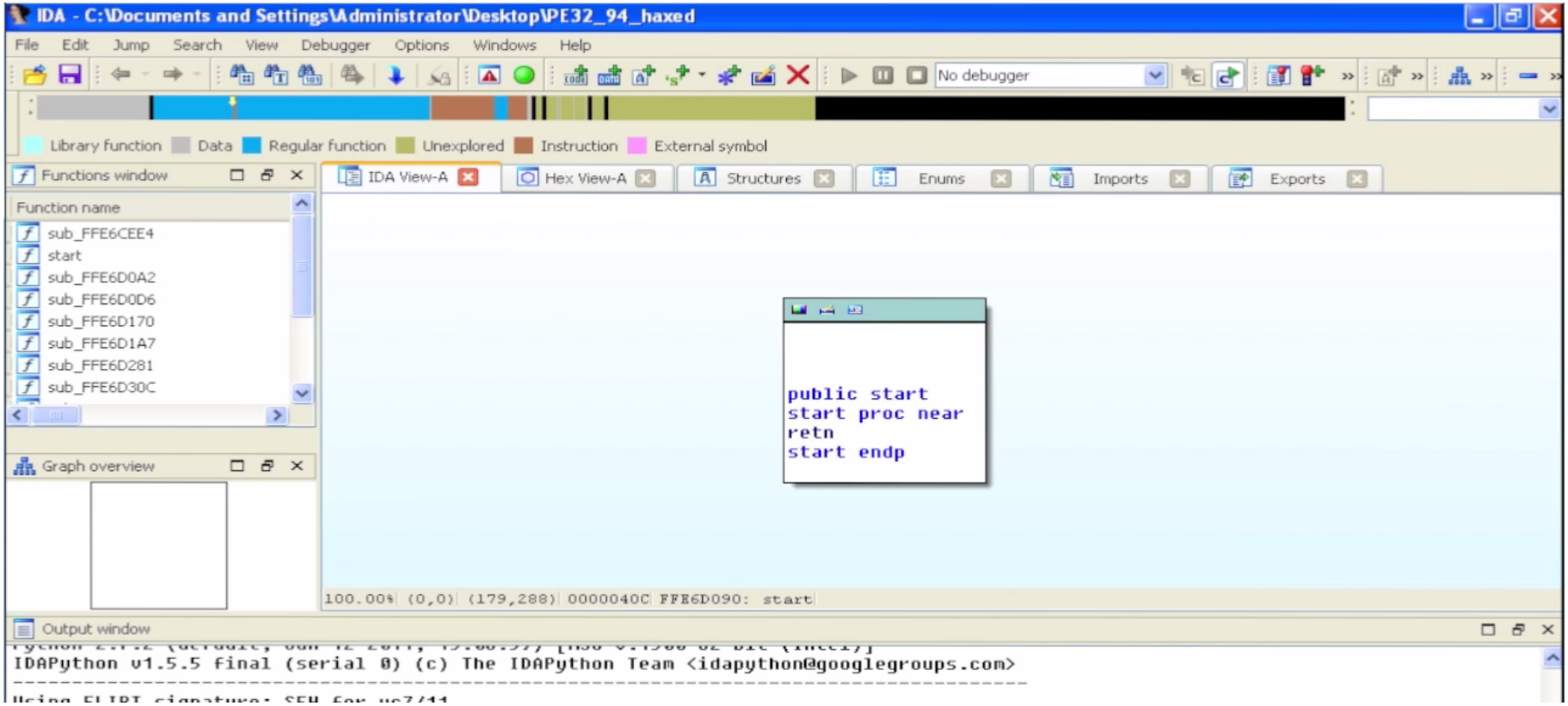
- You can cycle through each byte that is different by pressing ‘F6’ (Next Difference)
- In this simple example, there is only this single byte that is different

Analyzing UEFI Files with IDA



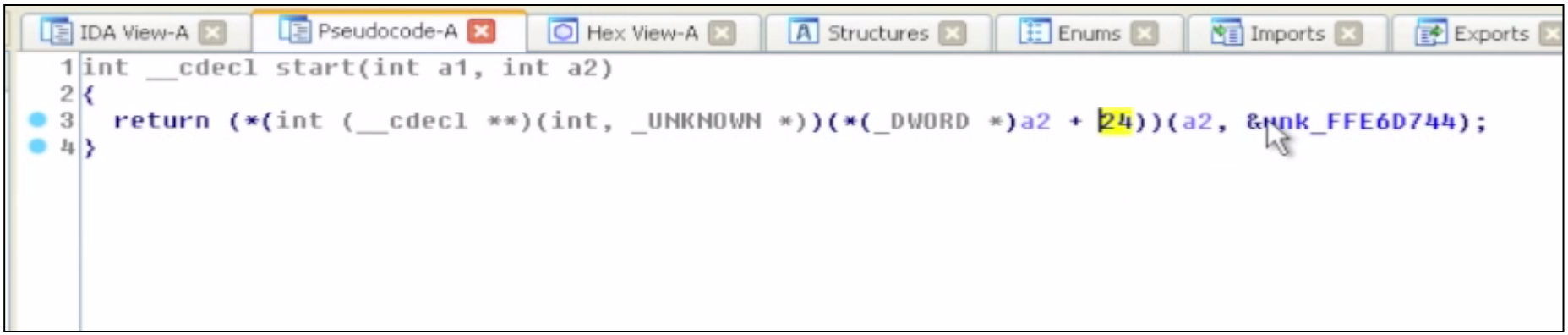
- Now we'll actually take a look at these files in IDA
  - Free version is mostly adequate, minus the Hex-Rays pseudo-code view
- Notice IDA recognizes the PE file format and opens the file accordingly
  - IDA 6.7 will recognize UEFI files! (but can't distinguish between PEI and DXE drivers, and so just applies a DXE entry point definition in both cases)
- Shown here is the non-hacked version of the TPM driver showing real instructions at the entry point

Analyzing UEFI Files with IDA



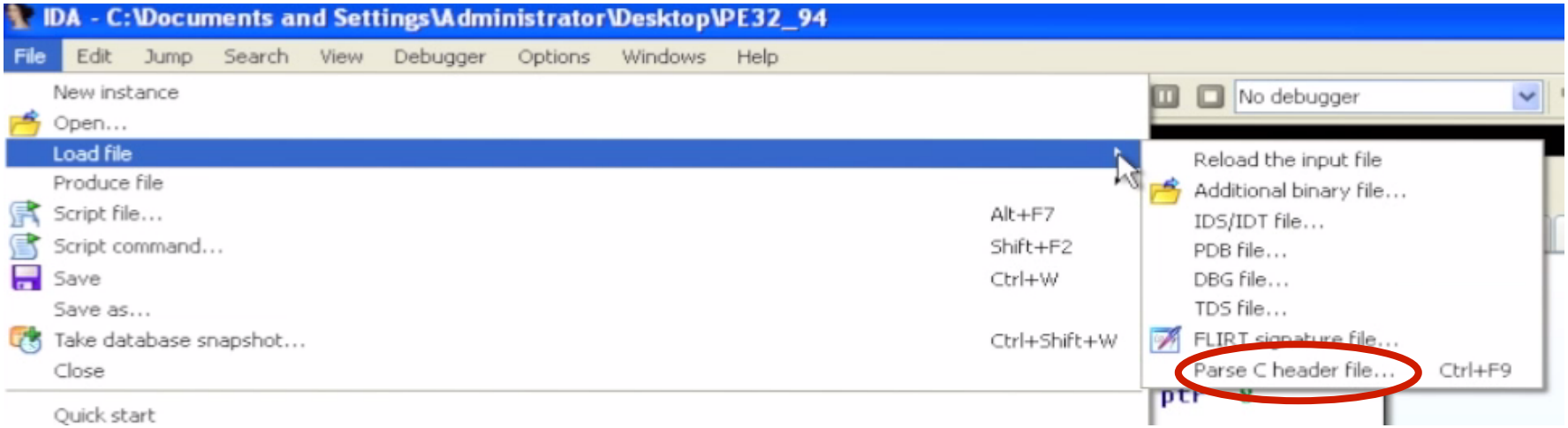
- Shown above is the hacked file with just the RET at the entry point
- This simple example assumes the attacker has placed this instruction here so that the TPM driver never performs any of its activities

Analyzing UEFI Files with IDA



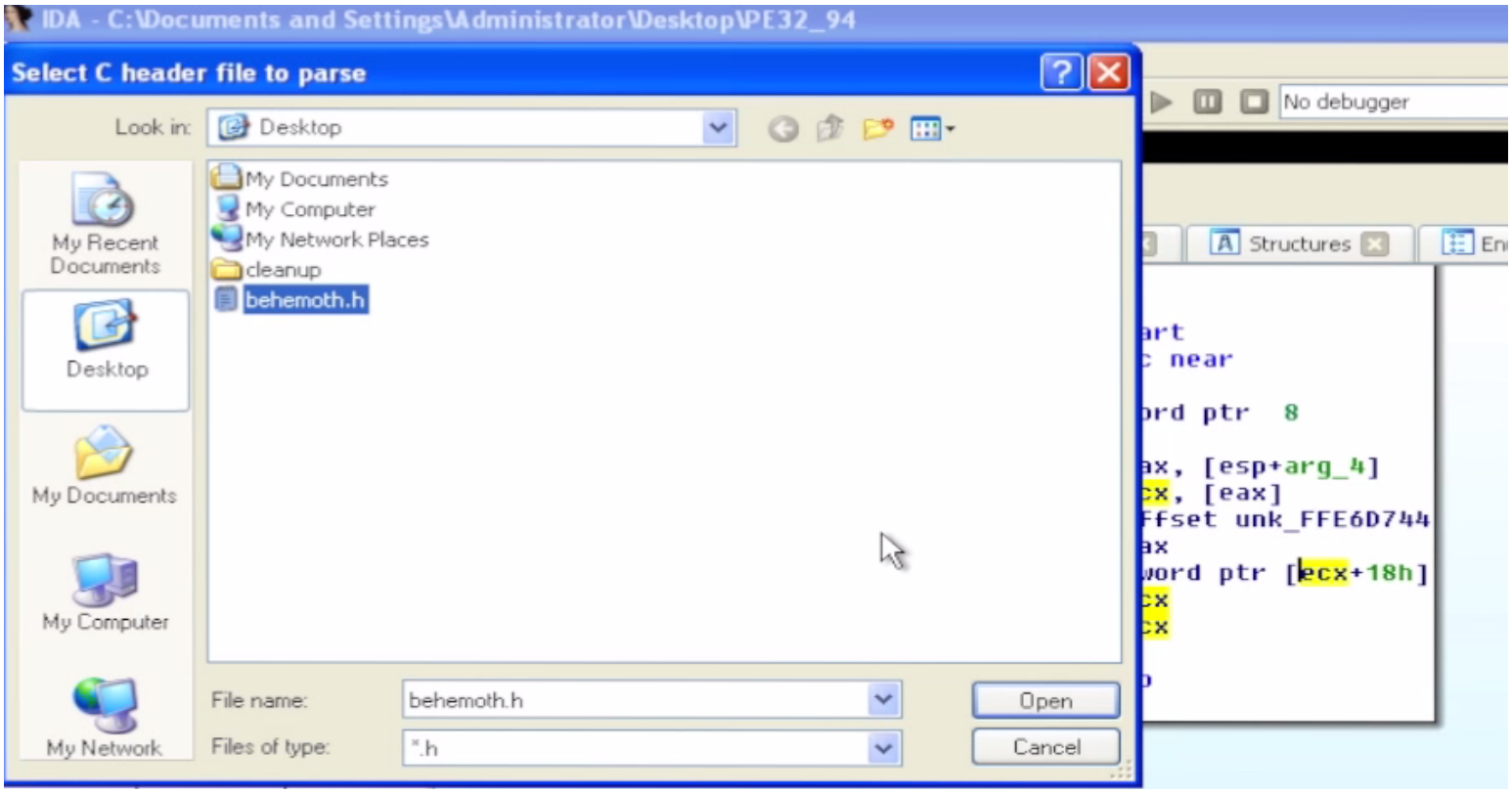
- To see the pseudo-code you will need the full version of IDA Pro with Hex-Rays
- The non-hacked file is dereferencing a DWORD at offset 24 of arg 2
  - IDA displays offsets in base 10 by default; 24 is 0x18
- The dereference is followed by a call: (a2, &unk_FFE6D744)
- So this appears to be calling a function pointer from out of a table

Applying UEFI Structure Definitions



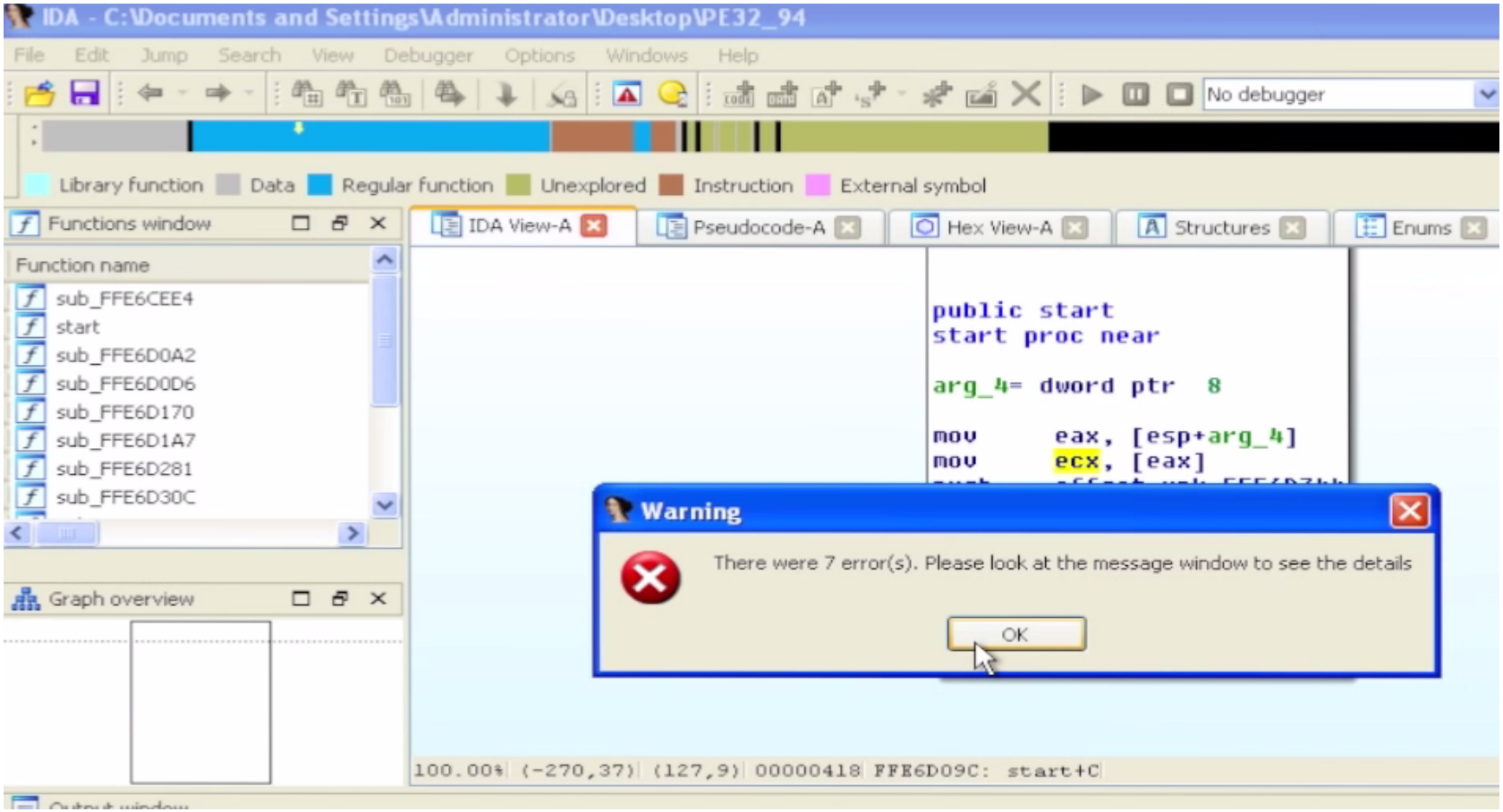
- UEFI uses publically-defined data structures
- We're going to import 'behemoth.h' which was created by Snare (using scripts)
  - https://github.com/snarez/ida-efiutils/blob/master/behemoth.h
  - Snare has done a talk on attacking Apple's EFI implementation
  - Black Hat USA 2012: http://ho.ax/downloads/De_Mysteriis_Dom_Jobsivs_Black_Hat_Slides.pdf
  - White Paper: http://ho.ax/De_Mysteriis_Dom_Jobsivs_Black_Hat_Paper.pdf

Applying UEFI Structure Definitions



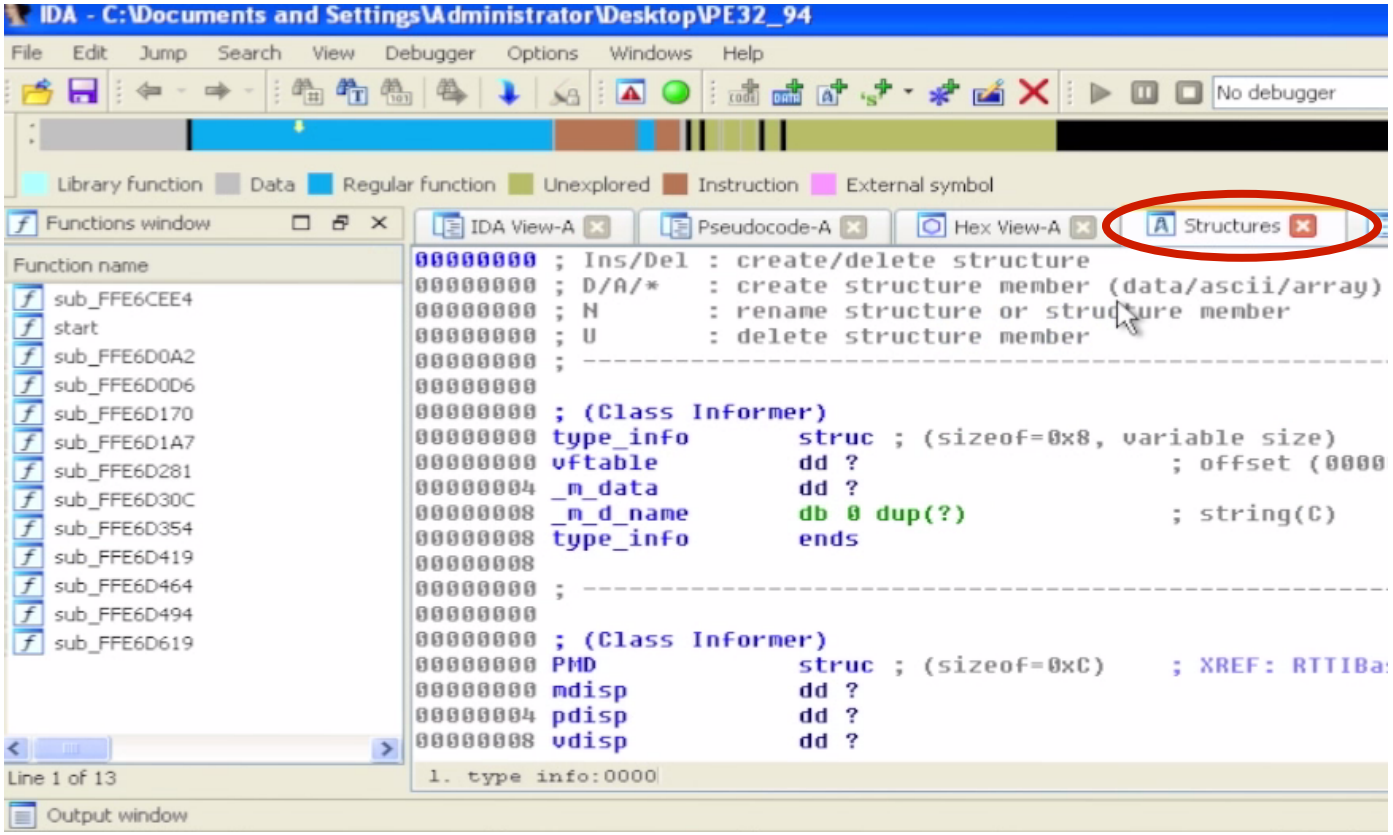
- Our behemoth.h file is located in the C:\Tools\ directory
- It contains a lot of structure definitions from the EFI Specification
  - Plus enumerated values and types

Applying UEFI Structure Definitions



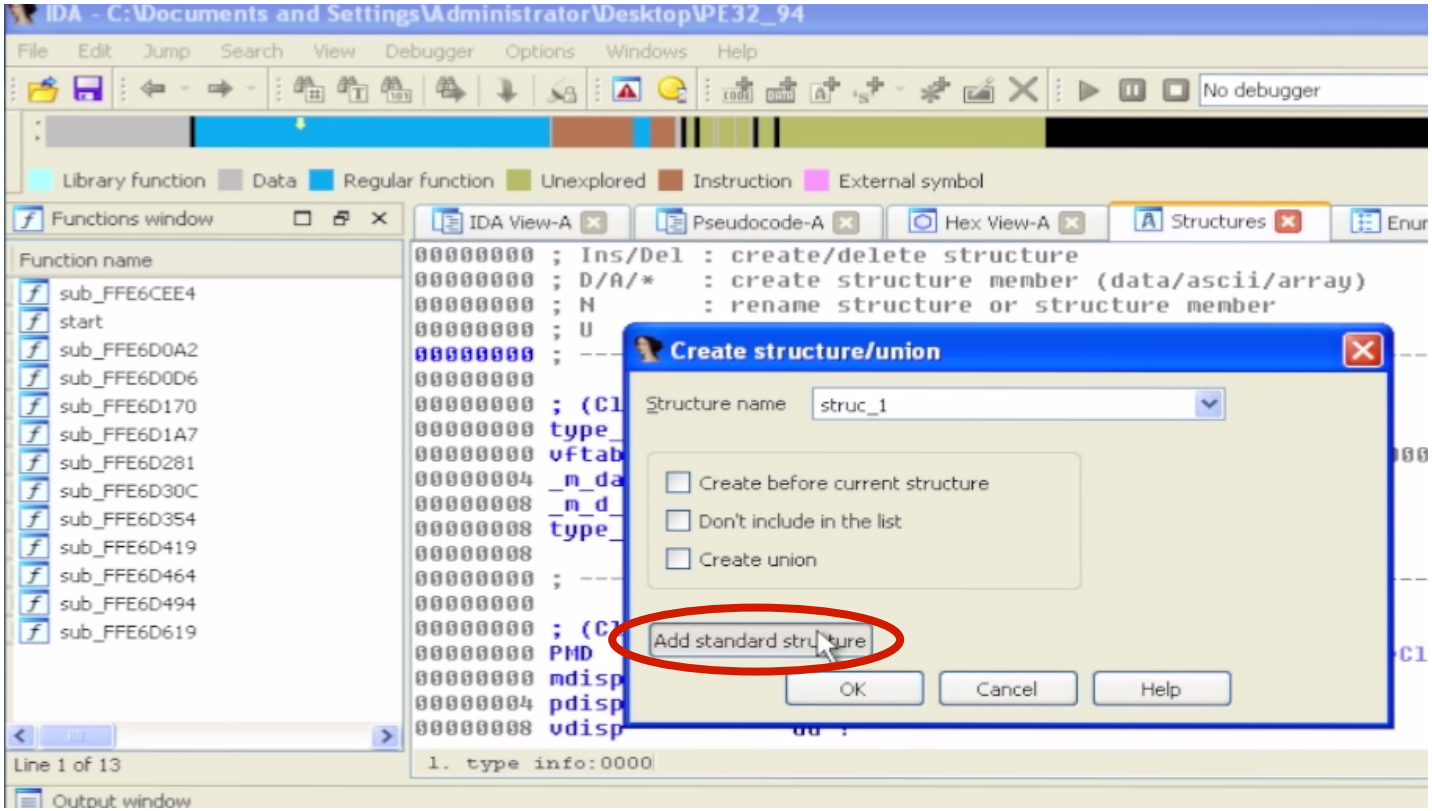
- Ignore any errors you see when importing this file
  - Importing the structures we use will still work

Applying UEFI Structure Definitions

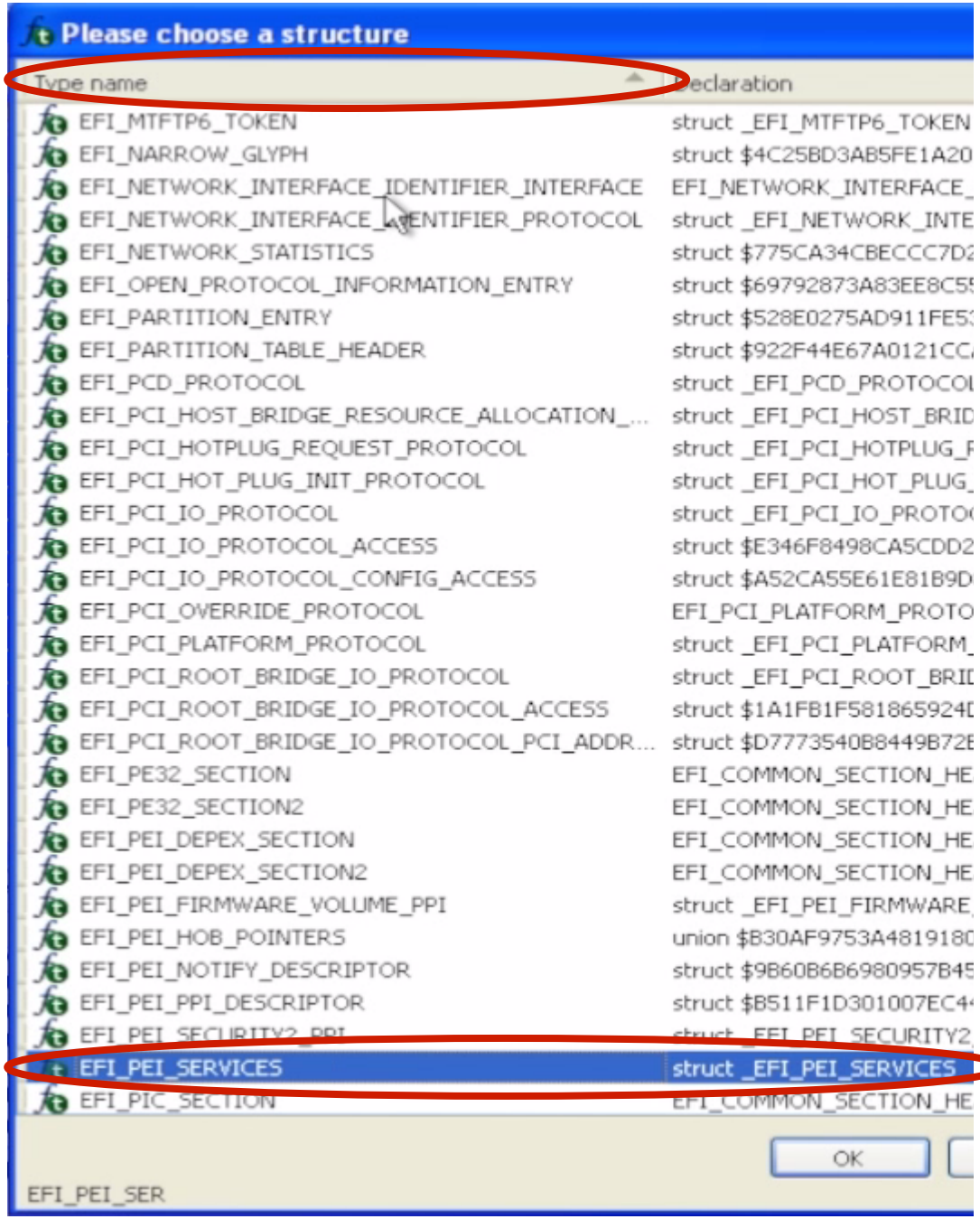


- Now go to the Structures tab
- Hit ‘Insert’

Applying UEFI Structure Definitions



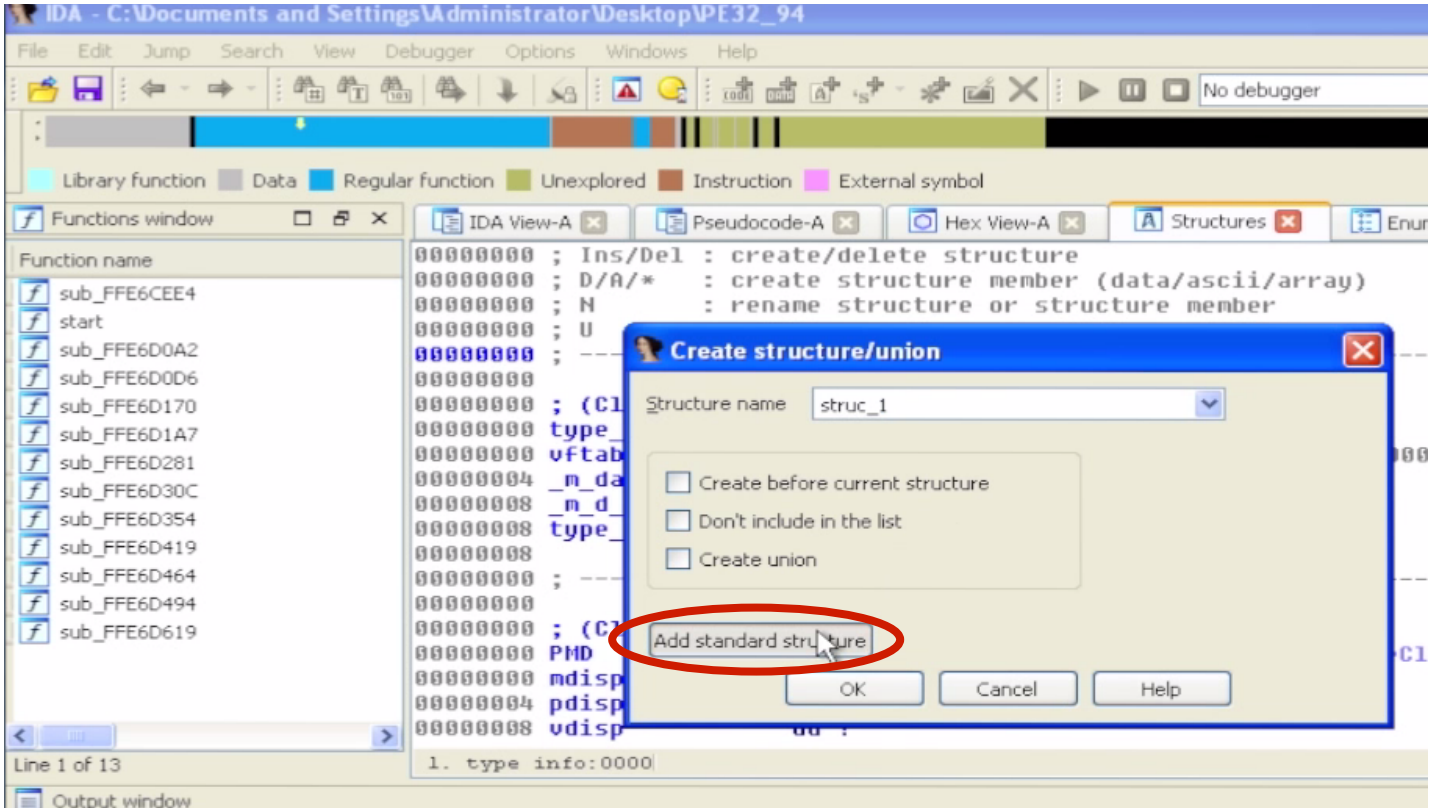
- Select ‘Add a Standard Structure’



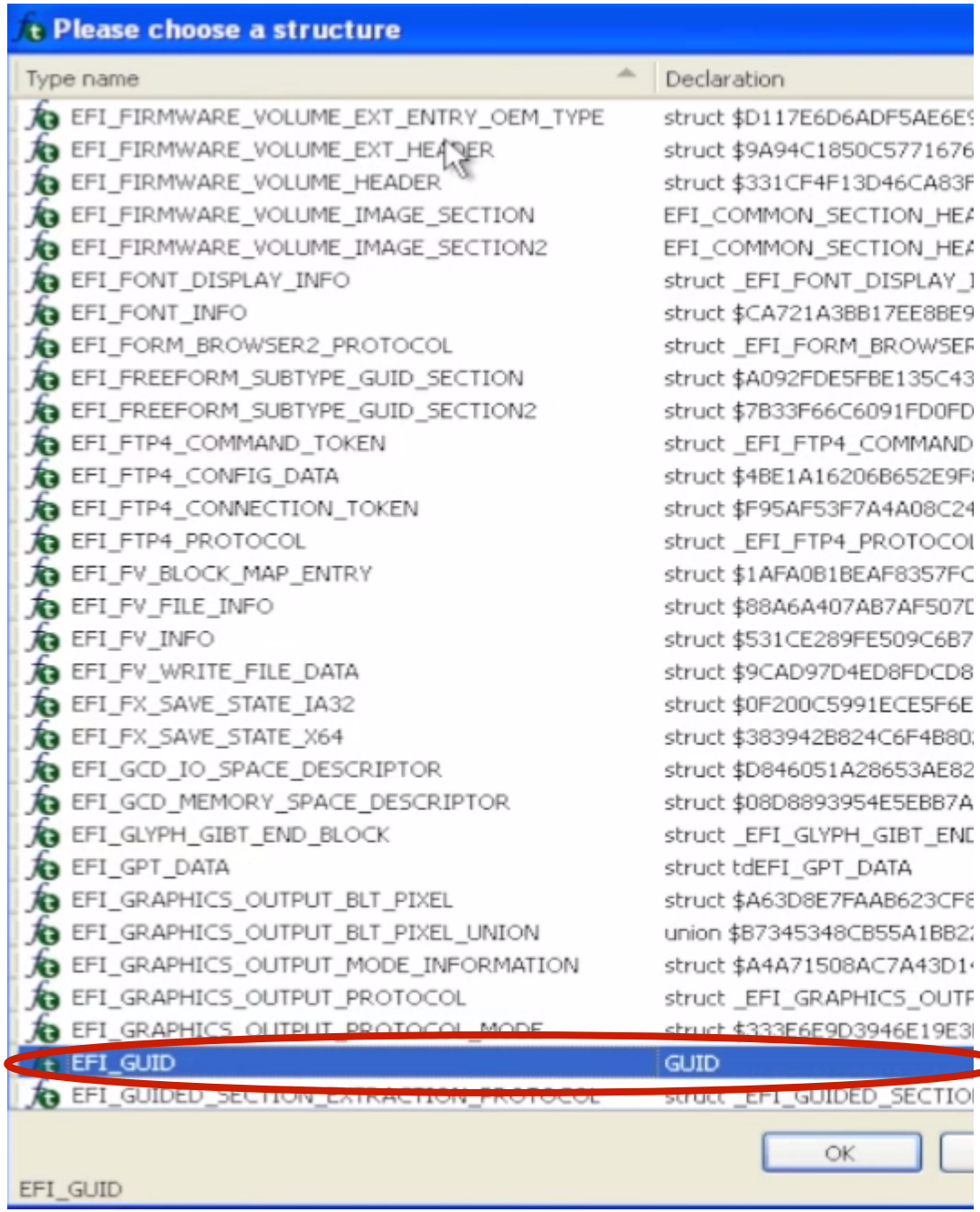
- We can sort the structures by name to make search easier
- We're looking for EFI_PEI_SERVICES
- These are services used by PEIMs during the PEI phase
- An (incomplete) sampling is below:

```
EFI_PEI_SERVICES struc ; (sizeof=0x78)
Hdr             EFI_TABLE_HEADER ?
InstallPpi      dd ?
ReInstallPpi    dd ?
LocatePpi       dd ?
NotifyPpi       dd ?
GetBootMode     dd ?
SetBootMode     dd ?
GetHobList      dd ?
CreateHob       dd ?
FfsFindNextVolume dd ?
FfsFindNextFile dd ?
FfsFindSectionData dd ?
InstallPeiMemory dd ?
```

Applying UEFI Structure Definitions



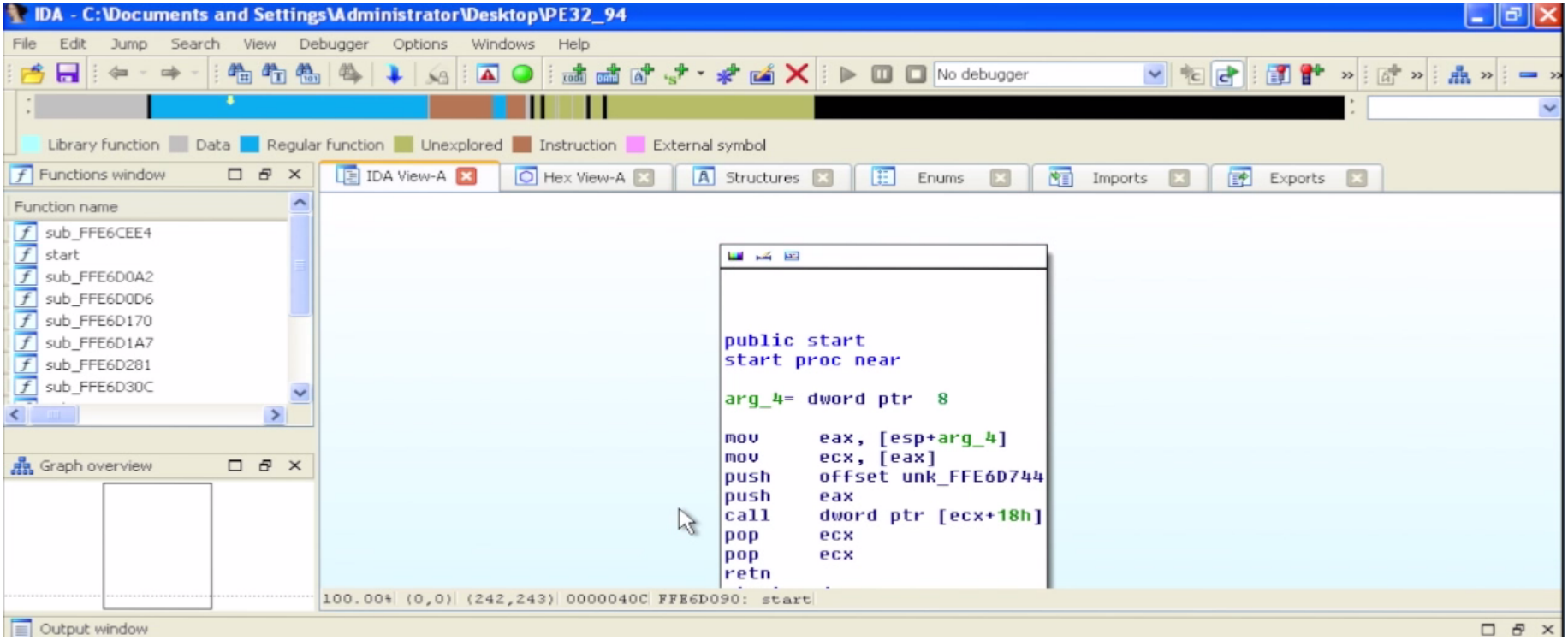
- Now we're going to add an EFI_GUID structure



- A GUID is a 16-byte data structure used as a name for many of the EFI objects:
  - Dword
  - Word
  - Word
  - Char array[8]

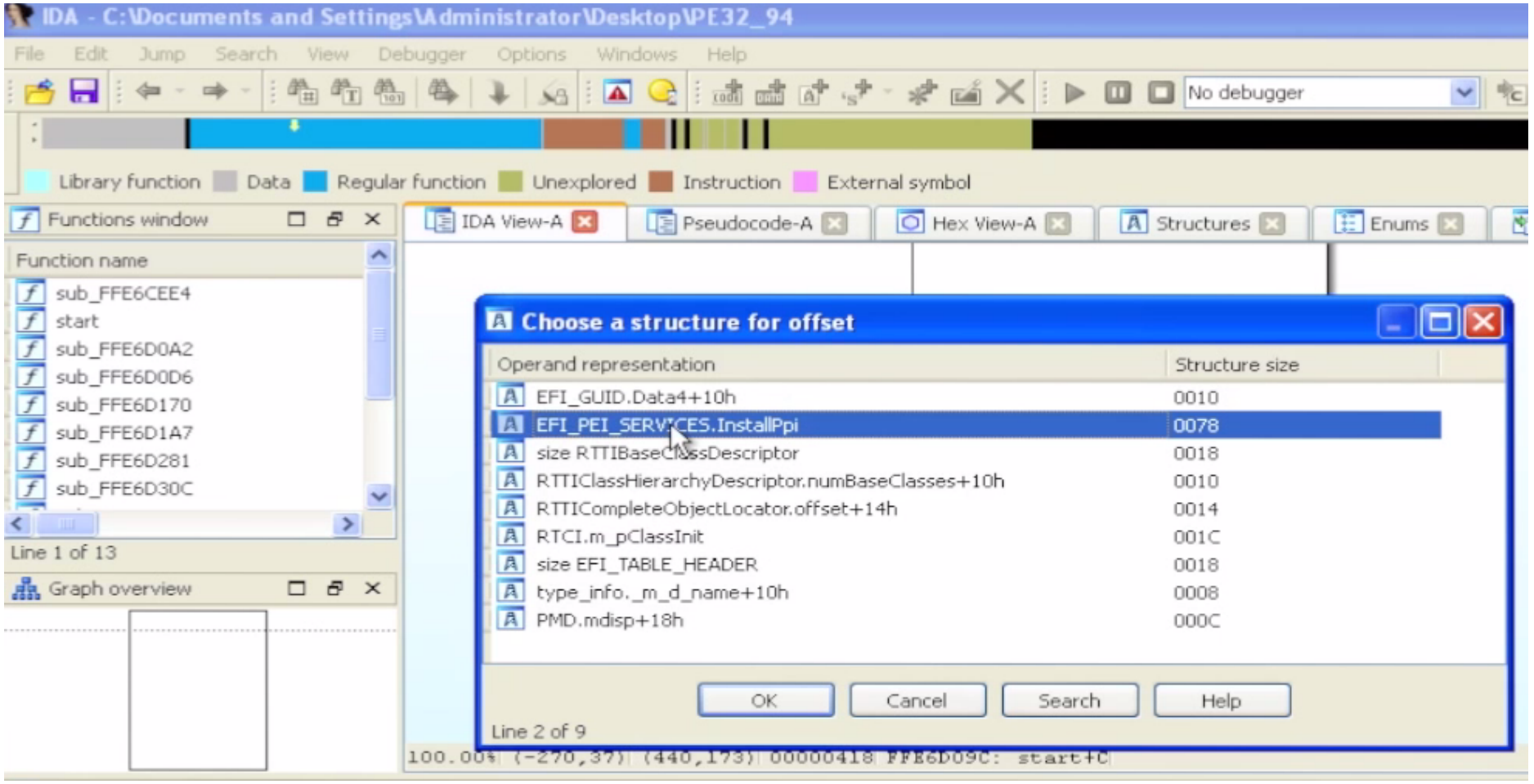
```
EFI_GUID        struc ; (sizeof=0x10)
Data1           dd ?
Data2           dw ?
Data3           dw ?
Data4           db 8 dup(?)
EFI_GUID        ends
```

Applying UEFI Structure Definitions



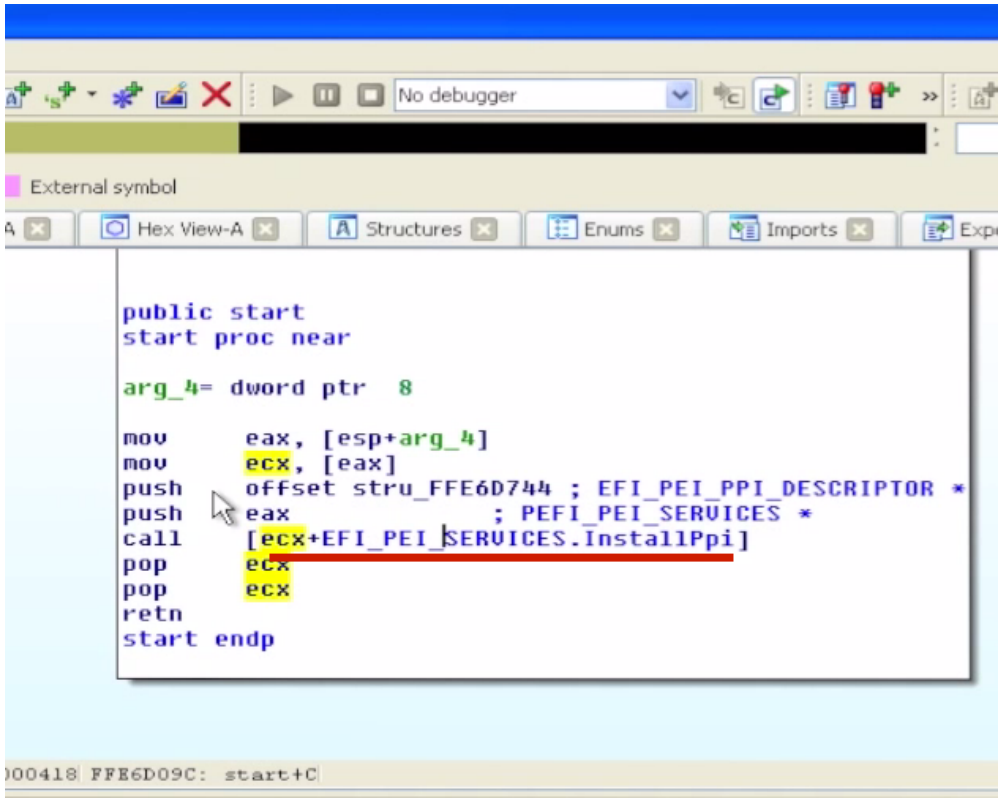
- Likely this file will be using the PEI Services table:
- The name of the file is ‘AmiTcgPlatformPeiBeforeMem’
- It’s a common structure used during the PEI phase so PEIMs can use common services

Applying UEFI Structure Definitions



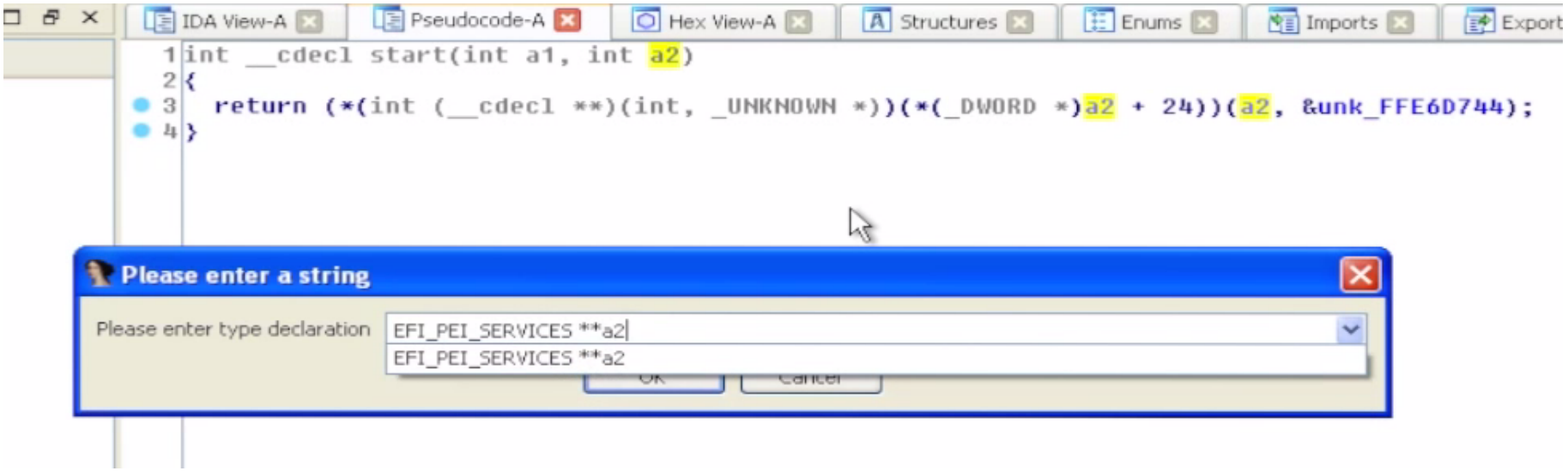
- Hit ‘t’ to have IDA interpret that value as a structure
- Select EFI_PEI_SERVICES based on our hypothesis

Applying UEFI Structure Definitions



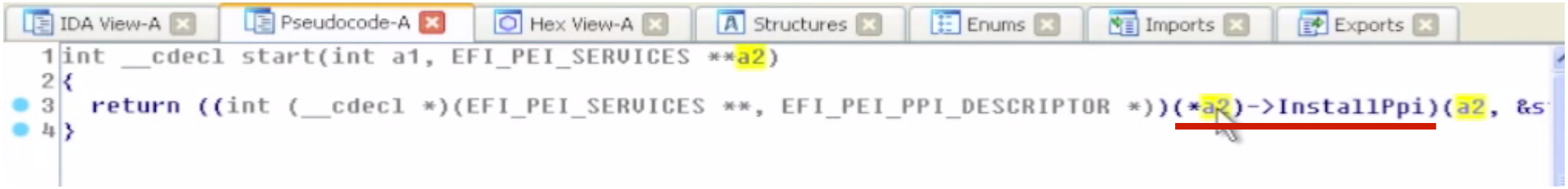
- Hit Ok or ‘y’ to accept this definition
- IDA does not have an undo, so it’s always good to save first
  - But we have a hunch that this is the right object

Applying UEFI Structure Definitions



- In the pseudo-code view you can do the same thing
- Select the a2 argument and hit ‘t’
- Select the EFI_PEI_SERVICES structure
- When we enter the above, we see the code simplifies:

Applying UEFI Structure Definitions



- We see that this function immediately calls the InstallPpi() PEI Service
- InstallPpi() takes 2 arguments:
  - The EFI_PEI_SERVICES structure
  - Some Unknown argument
- Per the EFI Specification, InstallPpi installs an interface in the PEI PEIM-to-PEIM Interface (PPI) database by GUID
- We could look up the prototype in the spec:

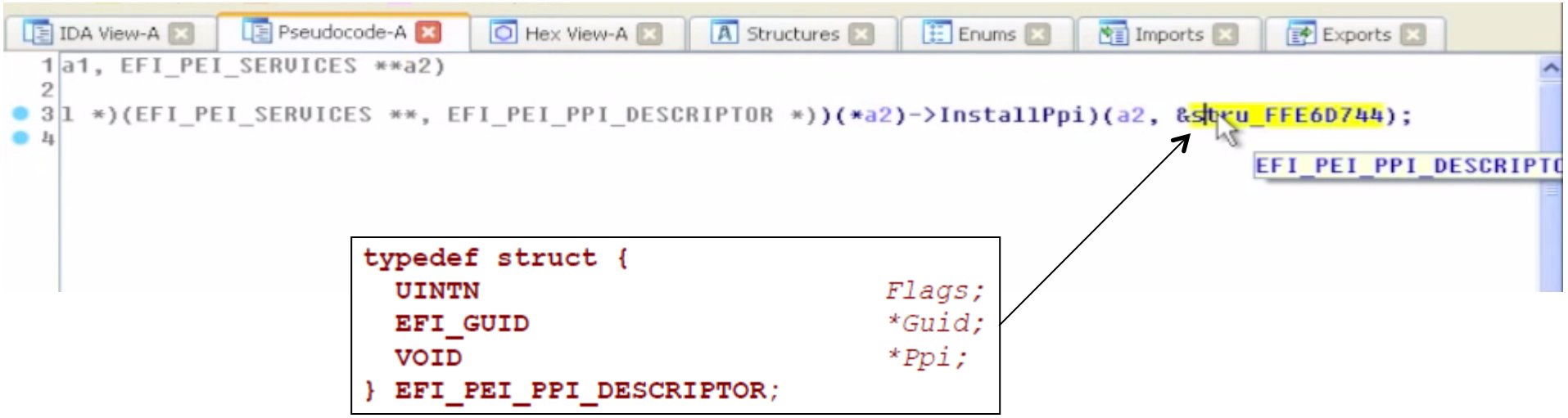
```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_INSTALL_PPI) (
  IN CONST EFI_PEI_SERVICES            **PeiServices,
  IN CONST EFI_PEI_PPI_DESCRIPTOR      *PpiList
  );
```

Always let the GUIDs be your GUIDe

- UEFI uses a lot of “GUIDs” – Globally Unique IDentifiers.
- Used to identify files on the filesystem
  - Filesystem GUIDs often reused between EDK & production systems. Or between the same IBV code on different OEMs’ systems
- Used to identify structures (PPIs in PEI phase, Protocols in DXE phase) that contain data and/or function pointers

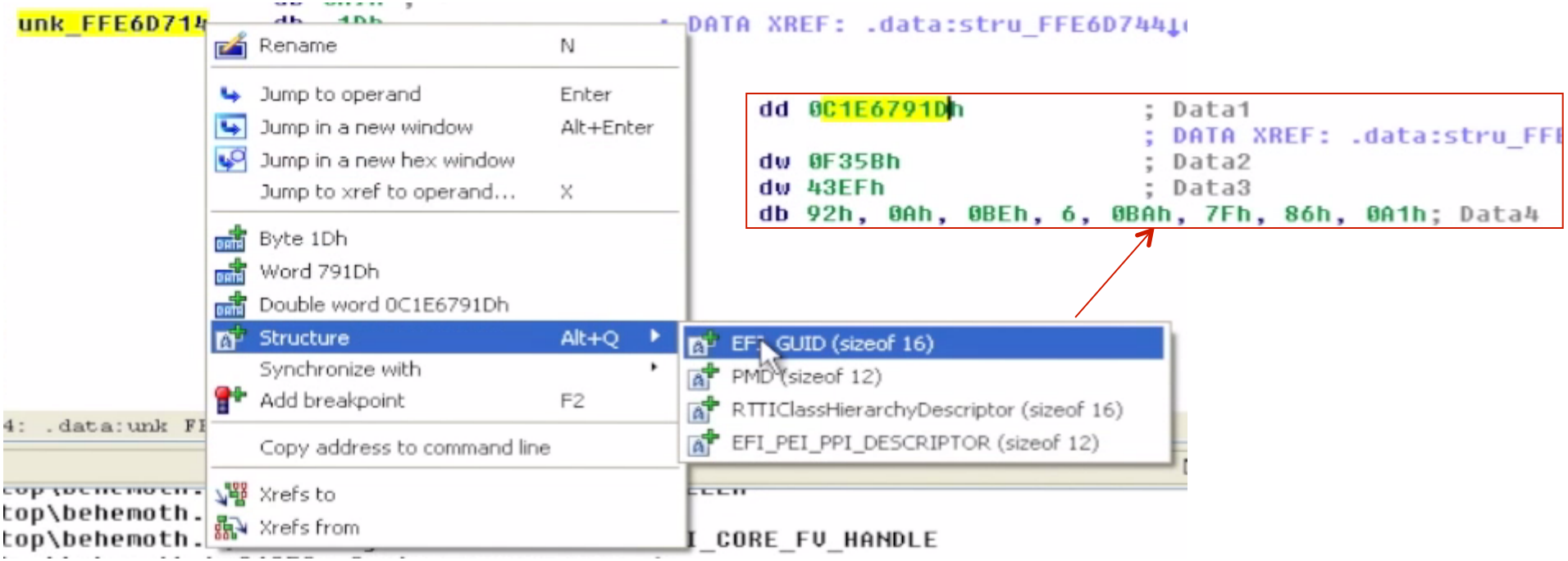

EFI_ARP_PROTOCOL_GUID
EFI_DEBUG_AGENT_GUID
EFI_ACPI_TABLE_GUID
EFI_DHCP4_PROTOCOL_GUID
DXE_CORE_FILE_NAME_GUID
EFI_CERT_TYPE_PKCS7_GUID
PERTH
KALGOORLIE
2127km 234 00'
CANBERRA
1897km 148 34'
KUALA LUM
4829km 298 02'
FRANKFUR
14213km 321 07'
EQUATOR
PORT MORESBY
SOUTH

Tracing PPIs



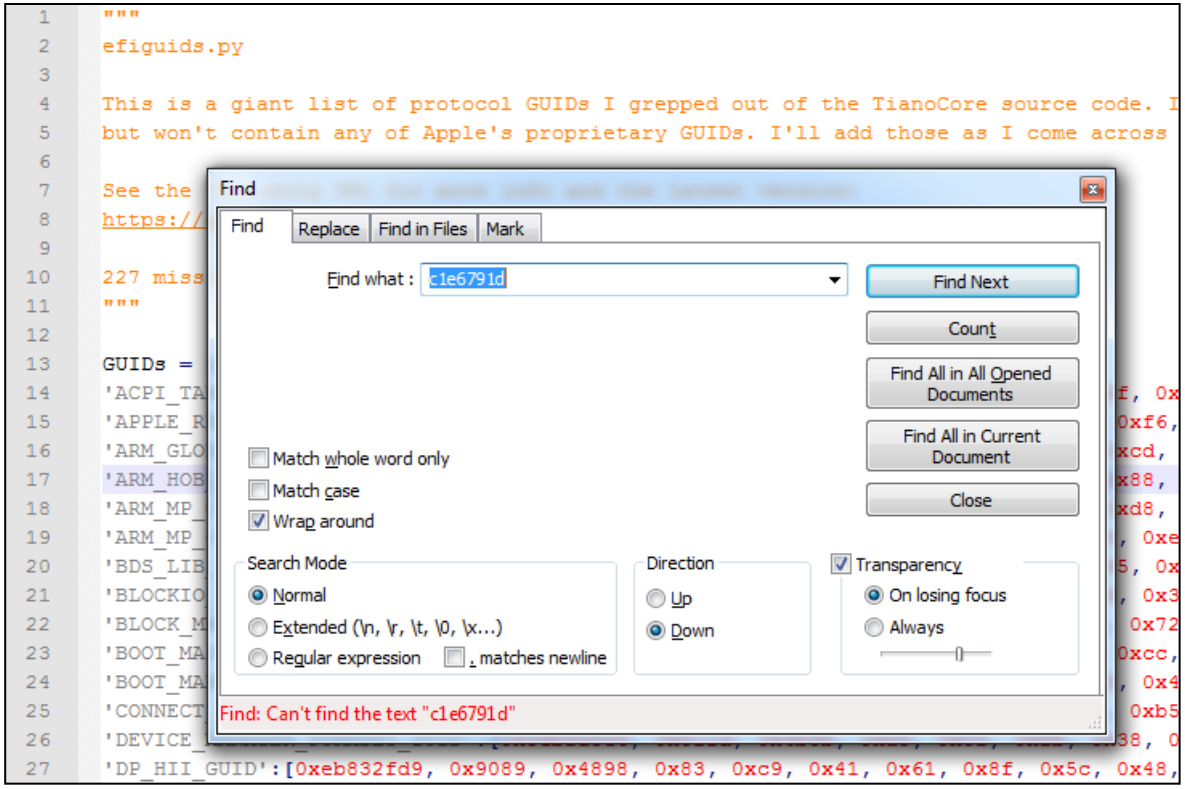
- But in this case IDA also recognizes this structure
- We can double-click on it to see that IDA has identified it as an EFI_PEI_PPI_DESCRIPTOR :
  - First is the Flags 80000010h
  - Second is the pointer to the GUID
  - Third is the pointer to the PPI that will be installed

Tracing PPIs



- Select the GUID structure
- One thing we can do is try and determine if this is a known-GUID or an unknown GUID
  - The UDK defines a lot of GUIDS, these would likely be the same across all vendors
  - Vendors also implement their own proprietary GUIDS

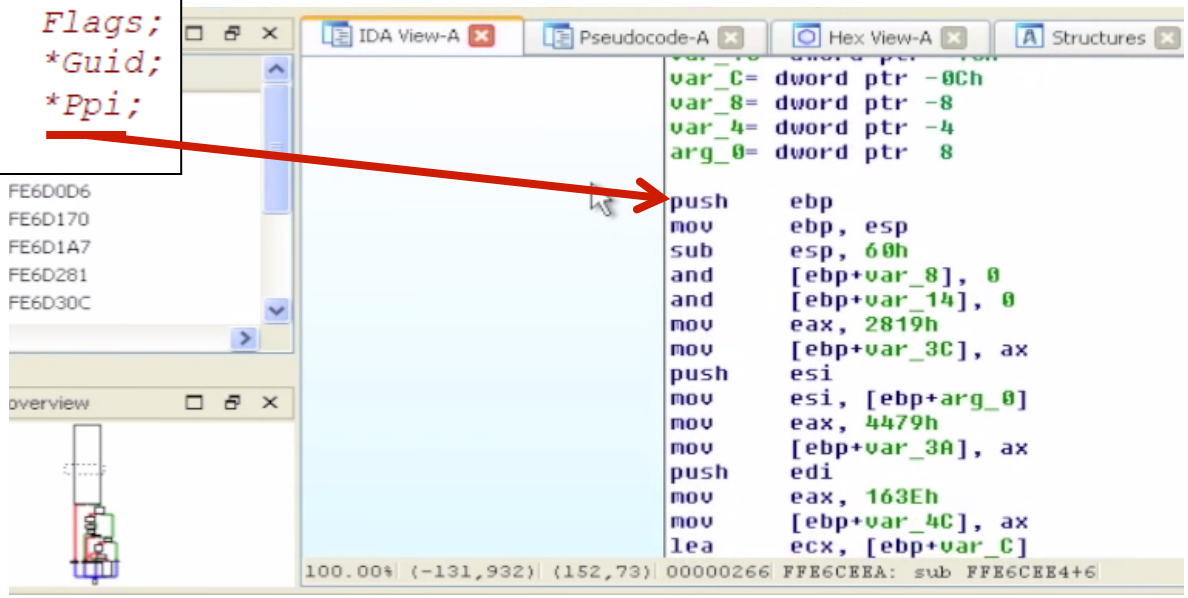
Tracing PPIs



- Snare also provides the efiguids.py file which contains GUIDs he pulled out of the UDK
- Our efiguids.py is located in C:\Tools\ and contains previously identified GUIDs
- In this case it is not in this file. We can name it ‘UnknownGuid1’

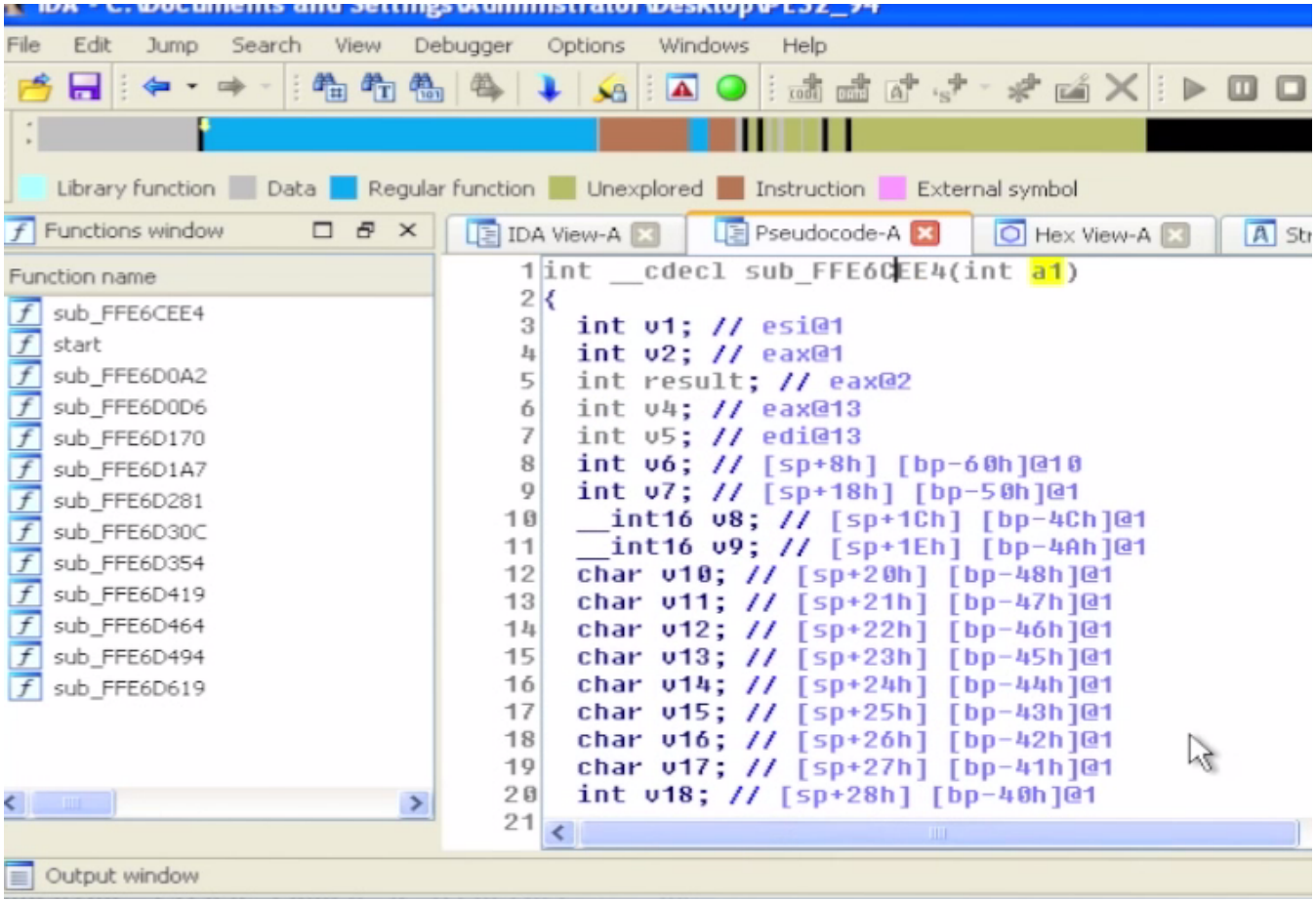
Tracing PPIs

```
typedef struct {
  UINTN                 Flags;
  EFI_GUID              *Guid;
  VOID                  *Ppi;
} EFI_PEI_PPI_DESCRIPTOR;
```



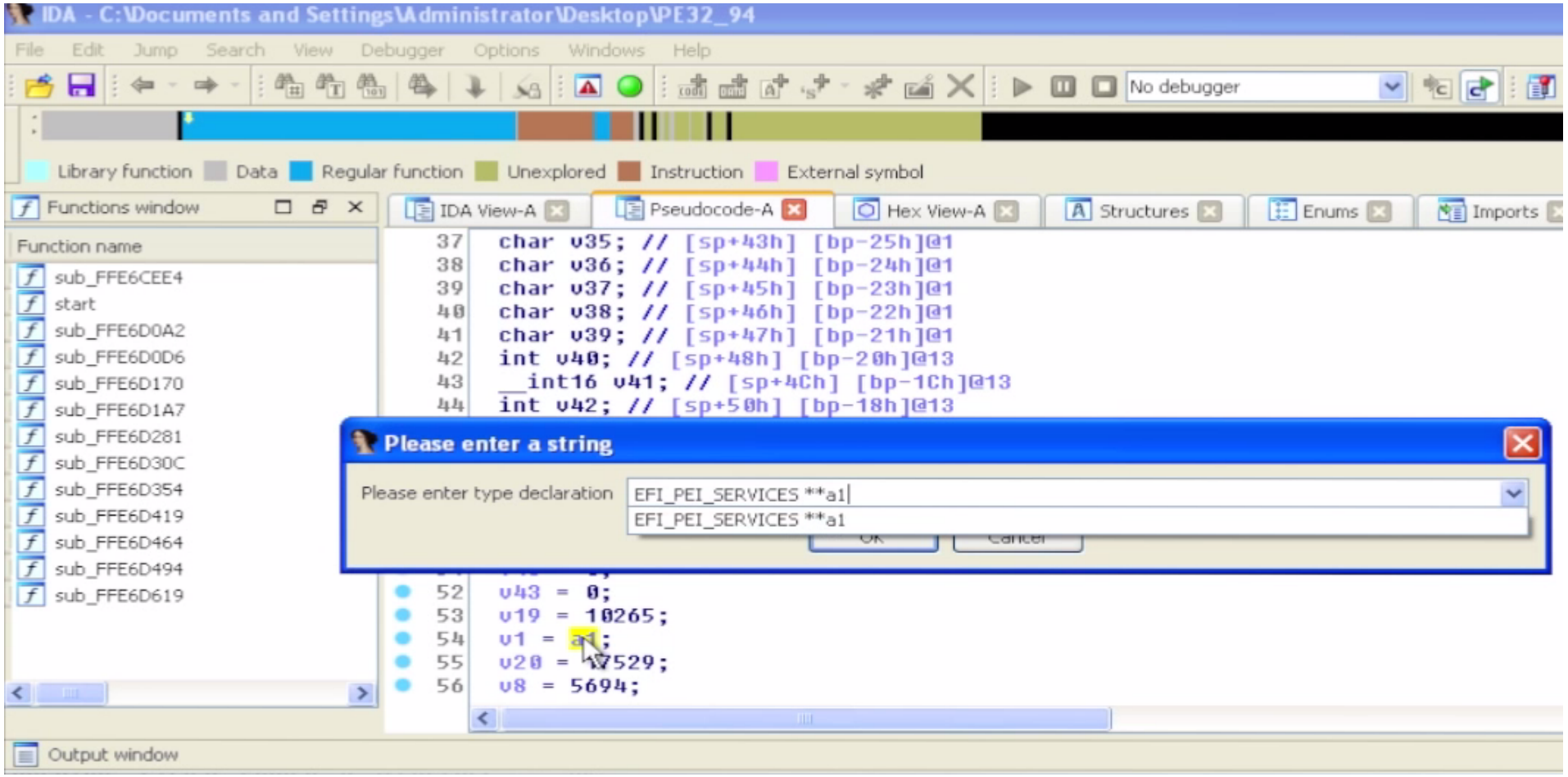
- Now if we follow the pointer it will take us to the PPI that is going to be installed
- This function is what will get called when someone uses this PPI

Recurse & define



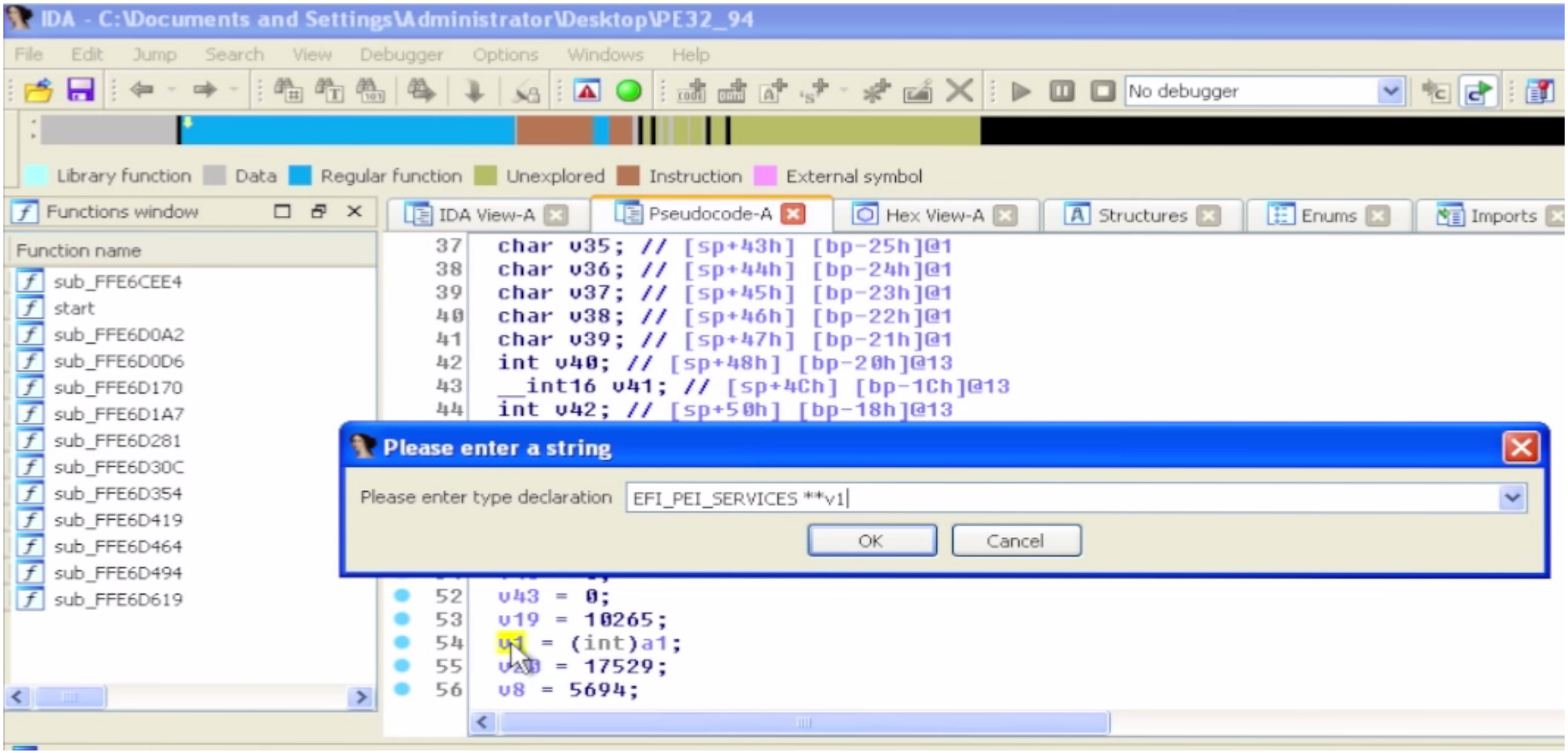
- We can analyze this is pseudo-code or the main view
- Since it accepts one argument we can hypothesize again that it takes in an instance of the EFI_PEI_SERVICES structure

Recurse & define



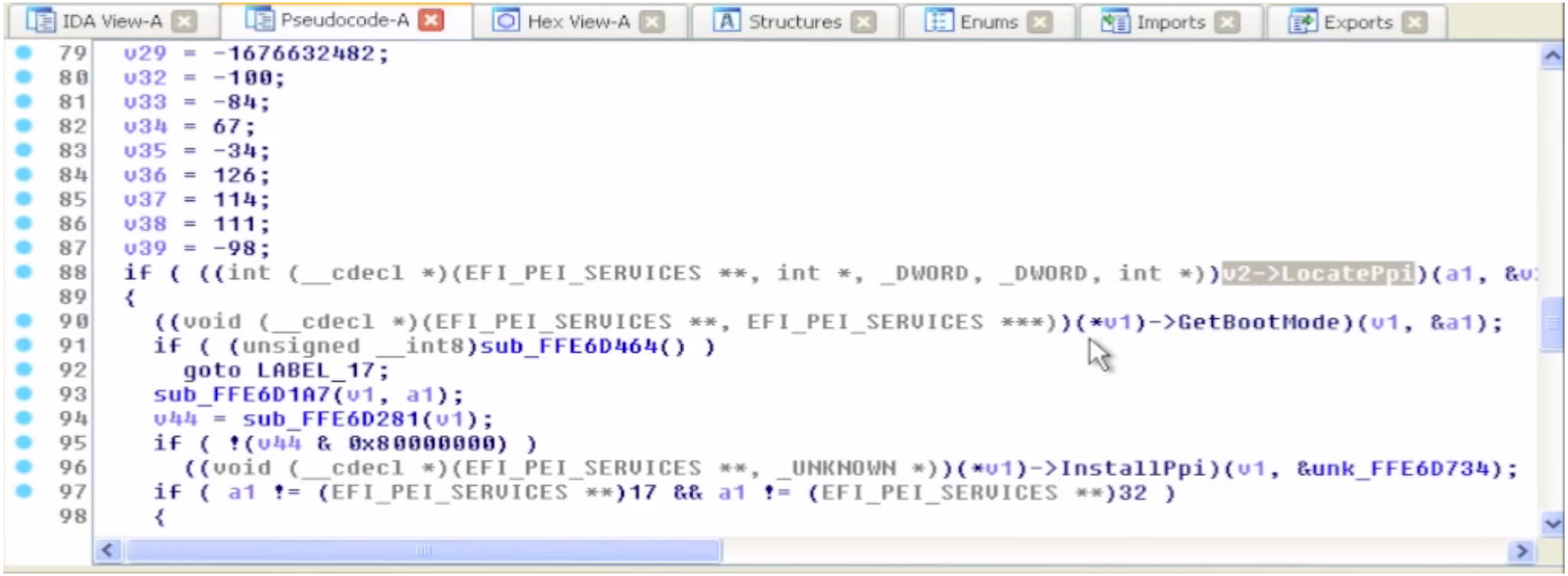
- As before, we can define this as EFI_PEI_SERVICES**a1

Recurse & define



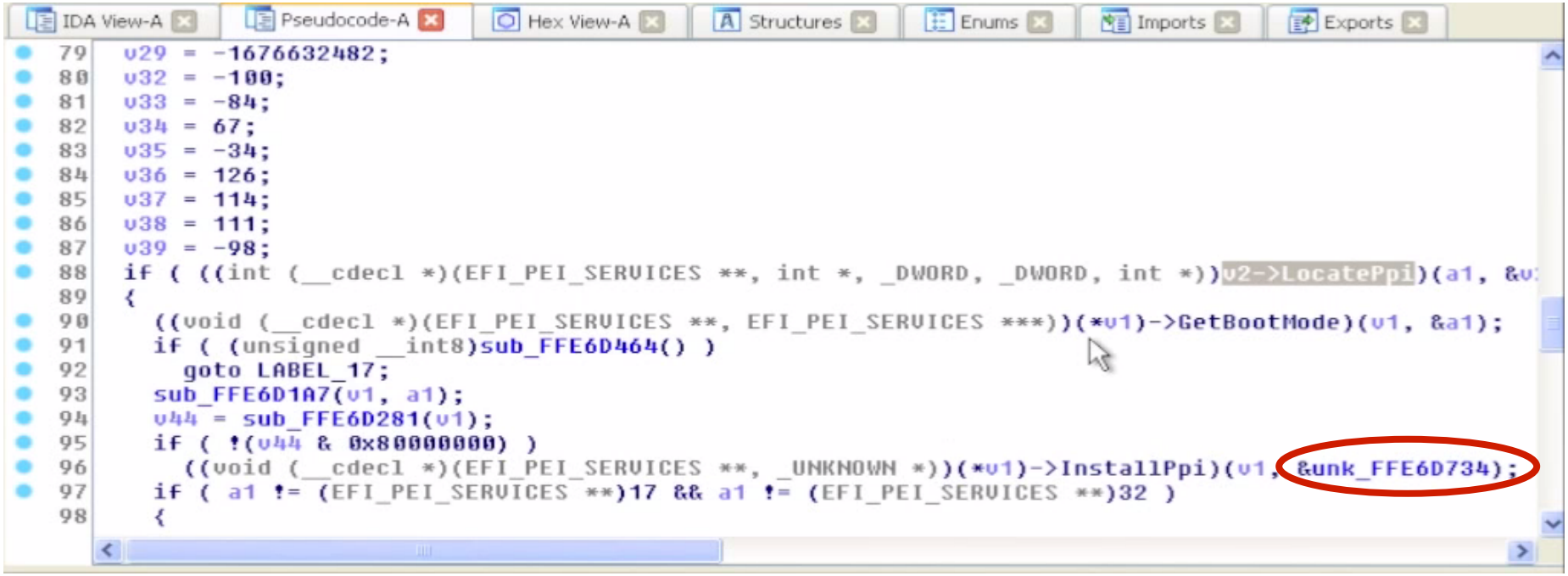
- Also we can define v1 in the same way since its equal to a1
- EFI_PEI_SERVICES**v1

Recurse & define



- Now we can scroll down and see that we were right in assuming this was an instance of a EFI_PEI_SERVICES
- We see a call to LocatePpi(), and then GetBootMode(), followed by InstallPpi()
- This series of EFI services “makes sense”

Recurse & define



- We can look up the definitions for the new services LocatePpi(), GetBootMode()
- Can we identify the GUID located in the EFI_PEI_PPI_DESCRIPTOR passed into InstallPpi?

```
'PEI_TPM_INITIALIZED_PPI_GUID':[0xe9db0d58, 0xd48d, 0x47f6, 0x9c, 0x6e, 0x6f, 0x40, 0xe8, 0x6c, 0x7b, 0x41],
```

Analyzing UEFI Files with IDA

- So from here the strategy would be to use the same methodology to identify and “fill out” LocatePpi(), GetBootMode(), etc.
- For you, cross-correlating where the PPIs are defined that you see getting called later will take a bit of grunt work (grepping for guids, finding their usage, etc)…
- For us, it’s already scripted ;)

Further GUID-based analysis strategies

- If you binary grep for a GUID (or search by GUID in UEFITool), you may find that it is specifically referenced/loaded by some other file.
- Pick a GUID in the spec that you're interested in. E.g. EFI_DHCP4_PROTOCOL_GUID
- If you grep for it, you'll find everywhere that particular protocol/PPI is used (to include installation, lookup, and things that have registered to be notified when it's available)
  - Then you just have to sift through the results

TODO:

- Add discussion of diffing things against EDK & against other known stuff
- Here comes a new challenger!
- http://joxeankoret.com/blog/2015/03/13/diaphora-a-program-diffing-plugin-for-ida-pro/

UEFI/Secure Boot Summary

- Secure boot can help you protect your firmware
  - If your BIOS is UEFI but Secure Boot isn't used, you can self-sign keys and turn it on
- But if the SPI flash isn't locked down, secure boot doesn't provide any protection
  - And neither does System Management Mode, or signed firmware updates, or TPM Measured Boot…
- UEFI does add complexity to locking down the SPI flash SPI Protected Range (PR) registers can be used to lock down the UEFI executable firmware
- But the NVRAM variables must remain writeable

A Locked Down UEFI/BIOS Does the Following:

- Has a properly-configured flash descriptor
  - Read-only, provides proper Flash Master permissions
- Protects the UEFI executable code using the PR registers
- Locks down the SPI flash configuration registers (FLOCKDN)
- Uses BIOS_CNTL to protect the flash
- Implements signed firmware updates
- Implements Secure Boot
- Ensures SMM_BWP is asserted so that the flash is writeable only when the processor is in SMM
- Ensures SMRAM is locked down (D_LCK is set and SMRR are used)
- Ensures SMI's are enabled and cannot be suppressed
- If possible uses Measured Boot and observes PCRs
- Sounds simple enough…

```
csc\fvsc\fv3\43172851-cf7e-4345-9fe0-d7012bb17b88\csc iFfsSmm
csc\fvsc\fv3\5552575a-7e00-4d61-a3a4-f7547351b49e\csc SmmBaseRuntime
csc\fvsc\fv3\59287178-59b2-49ca-bc63-532b12ea2c53\csc PchSmbusSmm
csc\fvsc\fv3\6869c5b3-ac8d-4973-8b37-e354dbf34add\csc CmosManagerSmm
csc\fvsc\fv3\753630c9-fae5-47a9-bbbf-88d621cd7282\csc SmmChildDispatcher
csc\fvsc\fv3\77a6009e-116e-464d-8ef8-b35201a022dd\csc DigitalThermalSensorSmm
csc\fvsc\fv3\7fed72ee-0170-4814-9878-a8fb1864dfaf\csc SmmRelocDxe
csc\fvsc\fv3\8d3be215-d6f6-4264-bea6-28073fb13aea\csc SmmThunk
csc\fvsc\fv3\921cd783-3e22-4579-a71f-00d74197fcc8\csc HeciSmm
csc\fvsc\fv3\9cc55d7d-fbff-431c-bc14-334eaea6052b\csc SmmDisp
csc\fvsc\fv3\a0bad9f7-ab78-491b-b583-c52b7f84b9e0\csc SmmControl
csc\fvsc\fv3\abb74f50-fd2d-4072-a321-cafc72977efa\csc SmmRelocPeim
csc\fvsc\fv3\acaeaa7a-c039-4424-88da-f42212ea0e55\csc PchPcieSmm
csc\fvsc\fv3\bc3245bd-b982-4f55-9f79-056ad7e987c5\csc AhciSmm
csc\fvsc\fv4\025b3ec4-28dc-44ae-8c94-d07563be743f\csc DellFnUsbEmulationSmm
csc\fvsc\fv4\0369cd67-fa74-45a3-bdcb-d25675d5ffde\csc DellOA30CtrlSmm-Edk1_06-Pi1_0-Uefi2_1
csc\fvsc\fv4\08abe065-c359-4b95-8d59-c1b58eb657b5\csc IntelLomSmm
csc\fvsc\fv4\099fd87f-4b39-43f6-ab47-f801f99209f7\csc DellDcpRegisterSmm-Edk1_06-Pi1_0-Uefi2_1
csc\fvsc\fv4\09d2cb46-c303-42c2-9726-5704a1fdfbbd\csc DellVariableSmmWrapper
csc\fvsc\fv4\0d28c529-87d4-4298-8a54-40f22a9fe24a\csc DellDaHddProtectionSmm-Edk1_06-Pi1_0-Uefi2_1
csc\fvsc\fv4\0d81fdc5-cb98-4b9f-b93b-70a9c0663abe\csc DellDccsSmmDriver
csc\fvsc\fv4\0dde9636-8321-4edf-9f14-0bfca3b473f5\csc DellIntrusionDetectSmm
csc\fvsc\fv4\1137c217-b5bc-4e9a-b328-1e7bcd530520\csc DellThermalDebugSmmDriver
csc\fvsc\fv4\1181e16d-af11-4c52-847e-516dd09bd376\csc DellCenturyRolloverSmm
csc\fvsc\fv4\119f3764-a7c2-4329-b25c-e6305e743049\csc DellSecurityVaultSmm-Edk1_06-Pi1_0-Uefi2_1
csc\fvsc\fv4\12963e55-5826-469e-a934-a3cbb3076ec5\csc SmmSbAcpi
csc\fvsc\fv4\1478454a-4584-4cca-b0d2-120ace129dbb\csc DellMfgModeSmmDriver
csc\fvsc\fv4\166fd043-ea13-4848-bb3c-6fa295b94627\csc DellVariableSmm-Edk1_06-Pi0_9-Uefi2_1
csc\fvsc\fv4\16c368fe-f174-4881-92ce-388699d34d95\csc SmmGpioPolicy
csc\fvsc\fv4\1afe6bd0-c9c5-44d4-b7bd-8f5e7d0f2560\csc DellDiagsSbControlSmm
csc\fvsc\fv4\26c04cf3-f5fb-4968-8d57-c7fa0a932783\csc SbServicesSmm
csc\fvsc\fv4\2a502514-1e81-4cda-9b50-8970fa4ac311\csc R5U242Smm
csc\fvsc\fv4\2aeda0eb-1392-4232-a4f9-c57a3c2fa2d9\csc BindingsSmm
```

- Oh but vendors also need to ensure that none of the code they implement in SMRAM is buggy
- On the Dell Latitude E6430, ~144 out of 495 EFI modules appear to contribute code to SMM …

Backup

- Used EFIPWN to backup because we don't recommend its use as a primary tool anymore (but it is still used behind the scenes for Copernicus' bios_diff.py)

EFIPWN

https://github.com/G33KatWork/EFIPWN

Setting up EFIPWN

- This describes using a version of EFIPWN modified by Sam Cornwell who added some improvements:

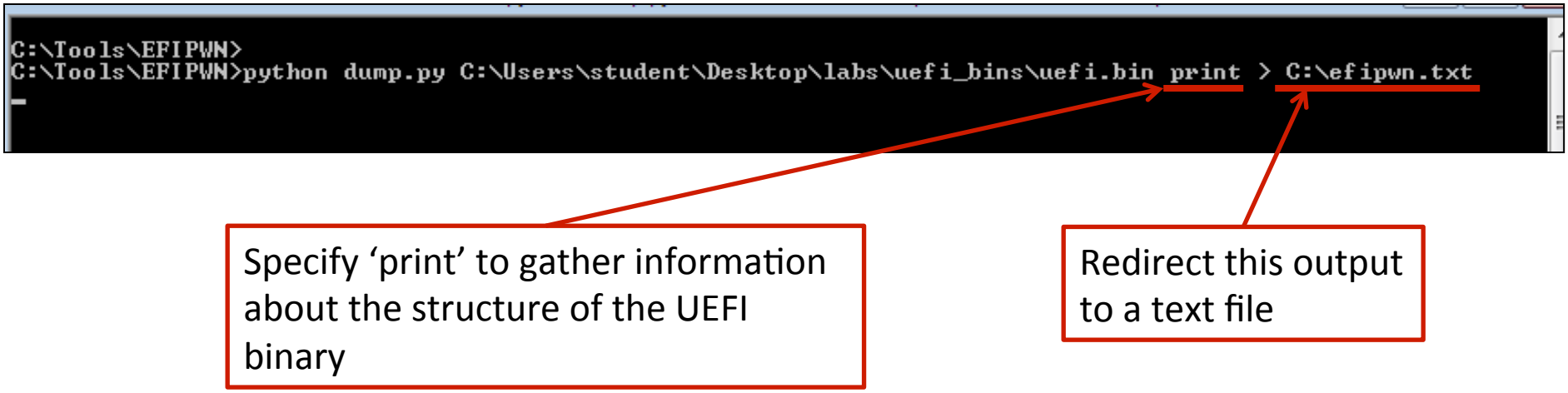
- EFIPWN requires the following:
- Python (I use 2.7.x-something)
- Mako: http://www.makotemplates.org/
- ArgParse: https://pypi.python.org/pypi/argparse
- Pylzma: http://www.joachim-bauch.de/projects/pylzma/
- I have an easier time downloading the source and installing using "python setup.py install"
- You will also need the 'xz' utility
  - Mac and Linux: you get it either automatically or by easy download
  - Windows: http://tukaani.org/xz/
  - The pre-built binaries work fine. I tested it by putting the bin_x86-x64 version into the local EFIPWN directory and it worked fine

Testing EFIPWN Functionality

```
C:\Tools\EFIPWN>python dump.py -h
usage: dump.py [-h] [-d] file {print,dump,genfdf} ...

EFI Firmware exploration tool

positional arguments:
  file                  The firmware file

optional arguments:
  -h, --help            show this help message and exit
  -d, --debug           Display debug information (DEBUG)

Operations:
  {print,dump,genfdf}
    print               Print a tree of the structure of the EFI firmware image
    dump                Dump all files in an EFI firmware image into a
                        directory structure
    genfdf              Try to create a EDK2 FDF file for generating a firmware
                        image out of a dump

C:\Tools\EFIPWN>
```

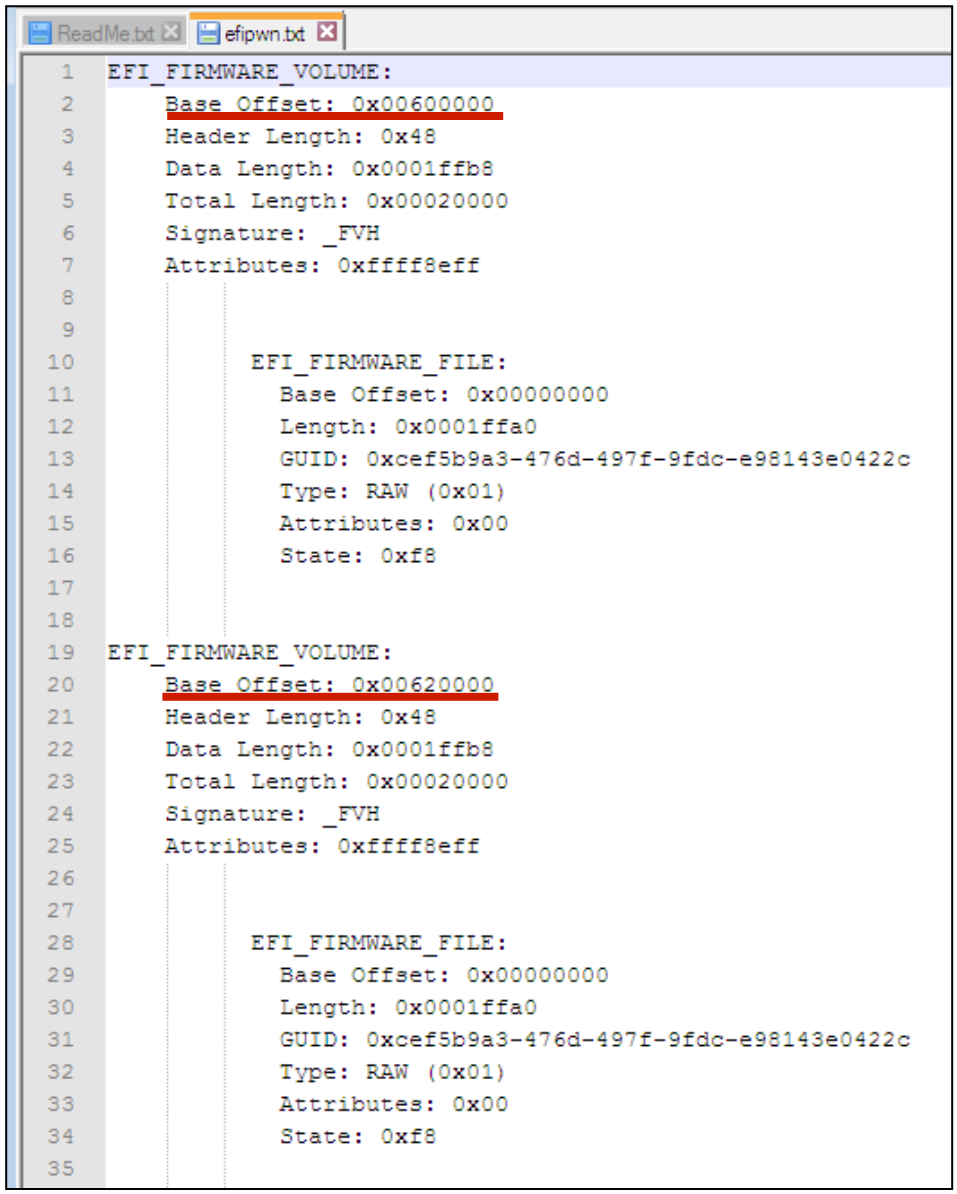
- Once you have all the dependencies installed, typing the following ‘python dump.py –h’ should yield the above output
- The arguments are a little confusing for EFIPWN, as a general rule they go like this:
- Python dump.y <file> <print, dump> <output>
- * The genfdf function does not work yet

EFIPWN ‘print’



- Before we decompose a UEFI binary, we’ll use the ‘print’ functionality to print a text file containing the UEFI firmware volume information and the PE files/modules contained therein

EFIPWN ‘print’: Firmware Volume



- The base offset is the Flash Linear Address (FLA) in the file where the volume begins
- This page shows one FV beginning at 60_0000h and another immediately following it at 62_0000h

EFIPWN ‘print’: Firmware Volume

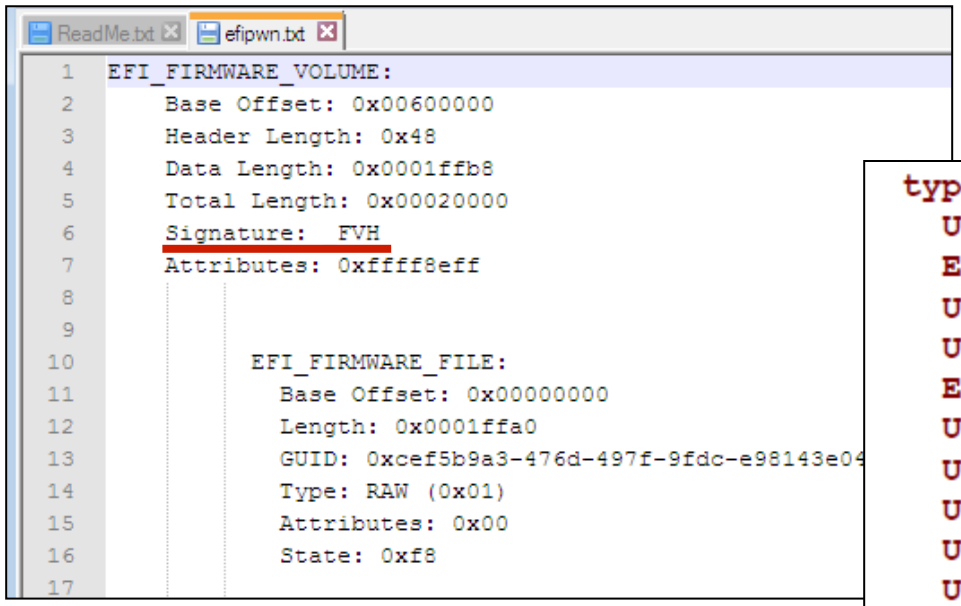
ReadMe.txt | efipwn.txt

```
1  EFI_FIRMWARE_VOLUME:
2      Base Offset: 0x00600000
3      Header Length: 0x48
4      Data Length: 0x0001ffb8
5      Total Length: 0x00020000
6      Signature: _FVH
7      Attributes: 0xffff8eff
8
9
10         EFI_FIRMWARE_FILE:
11           Base Offset: 0x00000000
12           Length: 0x0001ffa0
13           GUID: 0xcef5b9a3-476d-497f-9fdc-e98143e04
14           Type: RAW (0x01)
15           Attributes: 0x00
16           State: 0xf8
17
```

```
typedef struct {
  UINT8                 ZeroVector[16];
  EFI_GUID              FileSystemGuid;
  UINT64                FvLength;
  UINT32                Signature;
  EFI_FVB_ATTRIBUTES_2  Attributes;
  UINT16                HeaderLength;
  UINT16                Checksum;
  UINT16                ExtHeaderOffset;
  UINT8                 Reserved[1];
  UINT8                 Revision;
  EFI_FV_BLOCK_MAP      BlockMap[];
} EFI_FIRMWARE_VOLUME_HEADER;
```

- The Header length refers to the length in bytes of the FV header
- The Data length refers to the length in bytes of the FV minus the header
- The Total length refers to the total length of the FV including the header

EFIPWN ‘print’: Firmware Volume




```
typedef struct {
  UINT8                 ZeroVector[16];
  EFI_GUID              FileSystemGuid;
  UINT64                FvLength;
  UINT32                Signature;
  EFI_FVB_ATTRIBUTES_2  Attributes;
  UINT16                HeaderLength;
  UINT16                Checksum;
  UINT16                ExtHeaderOffset;
  UINT8                 Reserved[1];
  UINT8                 Revision;
  EFI_FV_BLOCK_MAP      BlockMap[];
} EFI_FIRMWARE_VOLUME_HEADER;
```

- The Signature of a firmware volume is {‘_’, ‘F’, ‘V’, ‘H’}
- The signature field only applies to Firmware Volumes

EFIPWN ‘print’: Firmware Volume

ReadMe.txt | efipwn.txt

```
1  EFI_FIRMWARE_VOLUME:
2      Base Offset: 0x00600000
3      Header Length: 0x48
4      Data Length: 0x0001ffb8
5      Total Length: 0x00020000
6      Signature: _FVH
7      Attributes: 0xffff8eff
8
9
10         EFI_FIRMWARE_FILE:
11           Base Offset: 0x00000000
12           Length: 0x0001ffa0
13           GUID: 0xcef5b9a3-476d-497f-9fdc-e98143e04
14           Type: RAW (0x01)
15           Attributes: 0x00
16           State: 0xf8
17
```

```
typedef struct {
  UINT8                 ZeroVector[16];
  EFI_GUID              FileSystemGuid;
  UINT64                FvLength;
  UINT32                Signature;
  EFI_FVB_ATTRIBUTES_2  Attributes;
  UINT16                HeaderLength;
  UINT16                Checksum;
  UINT16                ExtHeaderOffset;
  UINT8                 Reserved[1];
  UINT8                 Revision;
  EFI_FV_BLOCK_MAP      BlockMap[];
} EFI_FIRMWARE_VOLUME_HEADER;
```

- The attributes field declares capabilities and power-on defaults for the firmware volume

EFIPWN ‘print’: Firmware Volume

```
// Attributes bit definitions
#define EFI_FVB2_READ_DISABLED_CAP 0x00000001
#define EFI_FVB2_READ_ENABLED_CAP 0x00000002
#define EFI_FVB2_READ_STATUS 0x00000004
#define EFI_FVB2_WRITE_DISABLED_CAP 0x00000008
#define EFI_FVB2_WRITE_ENABLED_CAP 0x00000010
#define EFI_FVB2_WRITE_STATUS 0x00000020
#define EFI_FVB2_LOCK_CAP 0x00000040
#define EFI_FVB2_LOCK_STATUS 0x00000080
#define EFI_FVB2_STICKY_WRITE 0x00000200
#define EFI_FVB2_MEMORY_MAPPED 0x00000400
#define EFI_FVB2_ERASE_POLARITY 0x00000800
#define EFI_FVB2_READ_LOCK_CAP 0x00001000
#define EFI_FVB2_READ_LOCK_STATUS 0x00002000
#define EFI_FVB2_WRITE_LOCK_CAP 0x00004000
#define EFI_FVB2_WRITE_LOCK_STATUS 0x00008000
#define EFI_FVB2_ALIGNMENT 0x001F0000
#define EFI_FVB2_WEAK_ALIGNMENT 0x80000000
#define EFI_FVB2_ALIGNMENT_1 0x00000000
#define EFI_FVB2_ALIGNMENT_2 0x00010000
#define EFI_FVB2_ALIGNMENT_4 0x00020000
#define EFI_FVB2_ALIGNMENT_8 0x00030000
#define EFI_FVB2_ALIGNMENT_16 0x00040000
#define EFI_FVB2_ALIGNMENT_32 0x00050000
#define EFI_FVB2_ALIGNMENT_64 0x00060000
#define EFI_FVB2_ALIGNMENT_128 0x00070000
#define EFI_FVB2_ALIGNMENT_256 0x00080000
#define EFI_FVB2_ALIGNMENT_512 0x00090000
```

```
#define EFI_FVB2_ALIGNMENT_1K 0x000A0000
#define EFI_FVB2_ALIGNMENT_2K 0x000B0000
#define EFI_FVB2_ALIGNMENT_4K 0x000C0000
#define EFI_FVB2_ALIGNMENT_8K 0x000D0000
#define EFI_FVB2_ALIGNMENT_16K 0x000E0000
#define EFI_FVB2_ALIGNMENT_32K 0x000F0000
#define EFI_FVB2_ALIGNMENT_64K 0x00100000
#define EFI_FVB2_ALIGNMENT_128K 0x00110000
#define EFI_FVB2_ALIGNMENT_256K 0x00120000
#define EFI_FVB2_ALIGNMENT_512K 0x00130000
#define EFI_FVB2_ALIGNMENT_1M 0x00140000
#define EFI_FVB2_ALIGNMENT_2M 0x00150000
#define EFI_FVB2_ALIGNMENT_4M 0x00160000
#define EFI_FVB2_ALIGNMENT_8M 0x00170000
#define EFI_FVB2_ALIGNMENT_16M 0x00180000
#define EFI_FVB2_ALIGNMENT_32M 0x00190000
#define EFI_FVB2_ALIGNMENT_64M 0x001A0000
#define EFI_FVB2_ALIGNMENT_128M 0x001B0000
#define EFI_FVB2_ALIGNMENT_256M 0x001C0000
#define EFI_FVB2_ALIGNMENT_512M 0x001D0000
#define EFI_FVB2_ALIGNMENT_1G 0x001E0000
#define EFI_FVB2_ALIGNMENT_2G 0x001F0000
```

- Defined in Vol. 3 Shared Architectural Elements

EFIPWN ‘print’: Firmware Files

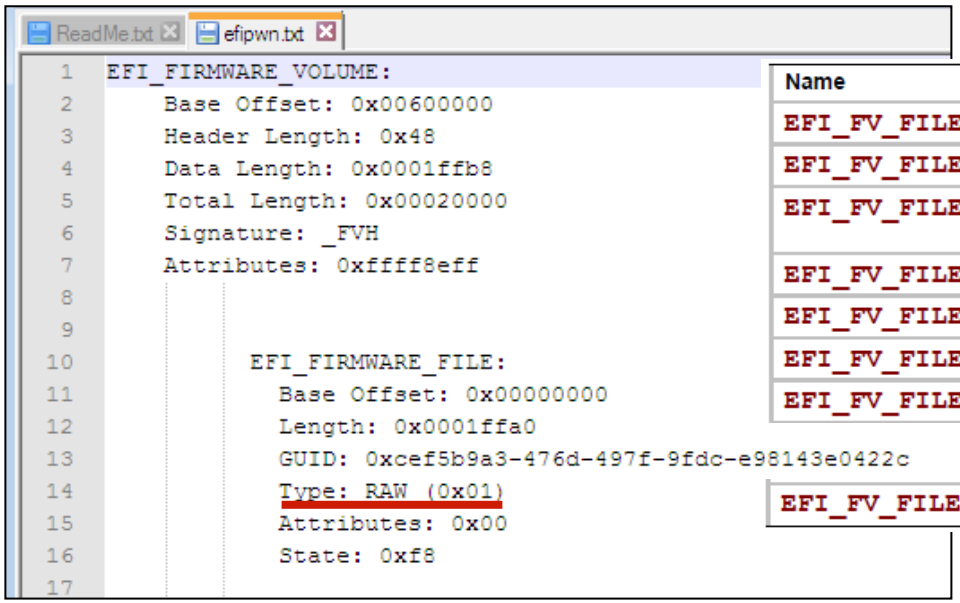
ReadMe.txt | efipwn.txt

```
1  EFI_FIRMWARE_VOLUME:
2      Base Offset: 0x00600000
3      Header Length: 0x48
4      Data Length: 0x0001ffb8
5      Total Length: 0x00020000
6      Signature: _FVH
7      Attributes: 0xffff8eff
8
9
10         EFI_FIRMWARE_FILE:
11           Base Offset: 0x00000000
12           Length: 0x0001ffa0
13           GUID: 0xcef5b9a3-476d-497f-9fdc-e98143e0422c
14           Type: RAW (0x01)
15           Attributes: 0x00
16           State: 0xf8
17
```

```
typedef struct {
  EFI_GUID                  Name;
  EFI_FFS_INTEGRITY_CHECK   IntegrityCheck;
  EFI_FV_FILETYPE           Type;
  EFI_FFS_FILE_ATTRIBUTES   Attributes;
  UINT8                     Size[3];
  EFI_FFS_FILE_STATE        State;
} EFI_FFS_FILE_HEADER;
```

- Firmware files are code and/or data stored within firmware volumes
- Combined, Firmware Files are described/contained within a Firmware File System
- Base offset refers to its relative location within the volume
- Length refers to the length of the file
- GUID is its ID

EFIPWN ‘print’: Firmware File



Name	Value	Description
EFI_FV_FILETYPE_RAW	0x01	Binary data
EFI_FV_FILETYPE_FREEFORM	0x02	Sectioned data
EFI_FV_FILETYPE_SECURITY_CORE	0x03	Platform core code used during the SEC phase
EFI_FV_FILETYPE_PEI_CORE	0x04	PEI Foundation
EFI_FV_FILETYPE_DXE_CORE	0x05	DXE Foundation
EFI_FV_FILETYPE_PEIM	0x06	PEI module (PEIM)
EFI_FV_FILETYPE_DRIVER	0x07	DXE driver
• • •		
EFI_FV_FILETYPE_FFS_PAD	0xF0	Pad File For FFS

- There are different enumerated types of Firmware Files
- Defined in Vol3 Shared Architectural Elements Section 2.1.4.1

EFIPWN ‘print’: Firmware Files

ReadMe.txt | efipwn.txt

```
1  EFI_FIRMWARE_VOLUME:
2      Base Offset: 0x00600000
3      Header Length: 0x48
4      Data Length: 0x0001ffb8
5      Total Length: 0x00020000
6      Signature: _FVH
7      Attributes: 0xffff8eff
8
9
10         EFI_FIRMWARE_FILE:
11           Base Offset: 0x00000000
12           Length: 0x0001ffa0
13           GUID: 0xcef5b9a3-476d-497f-9fdc-e98143e0422c
14           Type: RAW (0x01)
15           Attributes: 0x00
16           State: 0xf8
17
```

```
// FFS File Attributes
#define FFS_ATTRIB_LARGE_FILE           0x01
#define FFS_ATTRIB_FIXED                0x04
#define FFS_ATTRIB_DATA_ALIGNMENT       0x38
#define FFS_ATTRIB_CHECKSUM             0x40
```

- Firmware Files have attributes like Firmware Volumes do (and are the same)
- The State of the file is intended to preserve integrity

EFIPWN ‘print’: Firmware Files

ReadMe.txt | efipwn.txt

```
1  EFI_FIRMWARE_VOLUME:
2      Base Offset: 0x00600000
3      Header Length: 0x48
4      Data Length: 0x0001ffb8
5      Total Length: 0x00020000
6      Signature: _FVH
7      Attributes: 0xffff8eff
8
9
10         EFI_FIRMWARE_FILE:
11           Base Offset: 0x00000000
12           Length: 0x0001ffa0
13           GUID: 0xcef5b9a3-476d-497f-9fdc-e98143e0422c
14           Type: RAW (0x01)
15           Attributes: 0x00
16           State: 0xf8
17
```

```
// FFS File State Bits
#define EFI_FILE_HEADER_CONSTRUCTION      0x01
#define EFI_FILE_HEADER_VALID             0x02
#define EFI_FILE_DATA_VALID               0x04
#define EFI_FILE_MARKED_FOR_UPDATE        0x08
#define EFI_FILE_DELETED                  0x10
#define EFI_FILE_HEADER_INVALID           0x20
```

Bits 6, 7 are reserved bits.

- You can see that it includes the provision for marking files as deleted, which is kind of interesting. But unlike filesystem forensics, all these tools should basically show you all files, whether they’re deleted or not