

DECODING AND INTERPRETING INTEL PT TRACES FOR VULNERABILITY ANALYSIS

2020

MATT OH, **DARUNGRIM**

WHO AM I?

- Matt Oh
 - Author of DarunGrim Patch Diffing Tool
 - Now reviving the project as a **Binary Reverse Engineering Data Science Kit** - work in progress...
 - Ex Microsoft Security Researcher
 - Now Start-up Starter - focusing on threat detection and analysis technology

WHAT IS INTEL PT (PROCESSOR TRACE)?

Intel PT (Processor Trace) is a technology that is part of the recent Intel CPUs. Intel Skylake and later CPU models comes with this feature.

WHAT CAN YOU DO WITH INTEL PT?

- You can trace code execution at instruction level with triggering and filtering capabilities.
 - Filtering: CPL, CR3, IP Ranges
- Usage
 - Post-mortem analysis - crashes
 - Debugging performance issues
 - Enriching call-stack information
 - ...

THIS PRESENTATION

With this presentation, we want to explore the practical application of this technology in vulnerability analysis.

USING INTEL PT ON WINDOWS

For recording Intel PT records on Windows mainly three methods are available.

Name	Description
WindowsIntelPT	Works for Windows 10 pre-RS6
WinIPT	Windows 10 Post-RS6. Uses ipt.sys interface
Intel® Debug Extensions for WinDbg* for Intel® Processor Trace	Needs physical kernel debugging connection (ex. USB debugging)

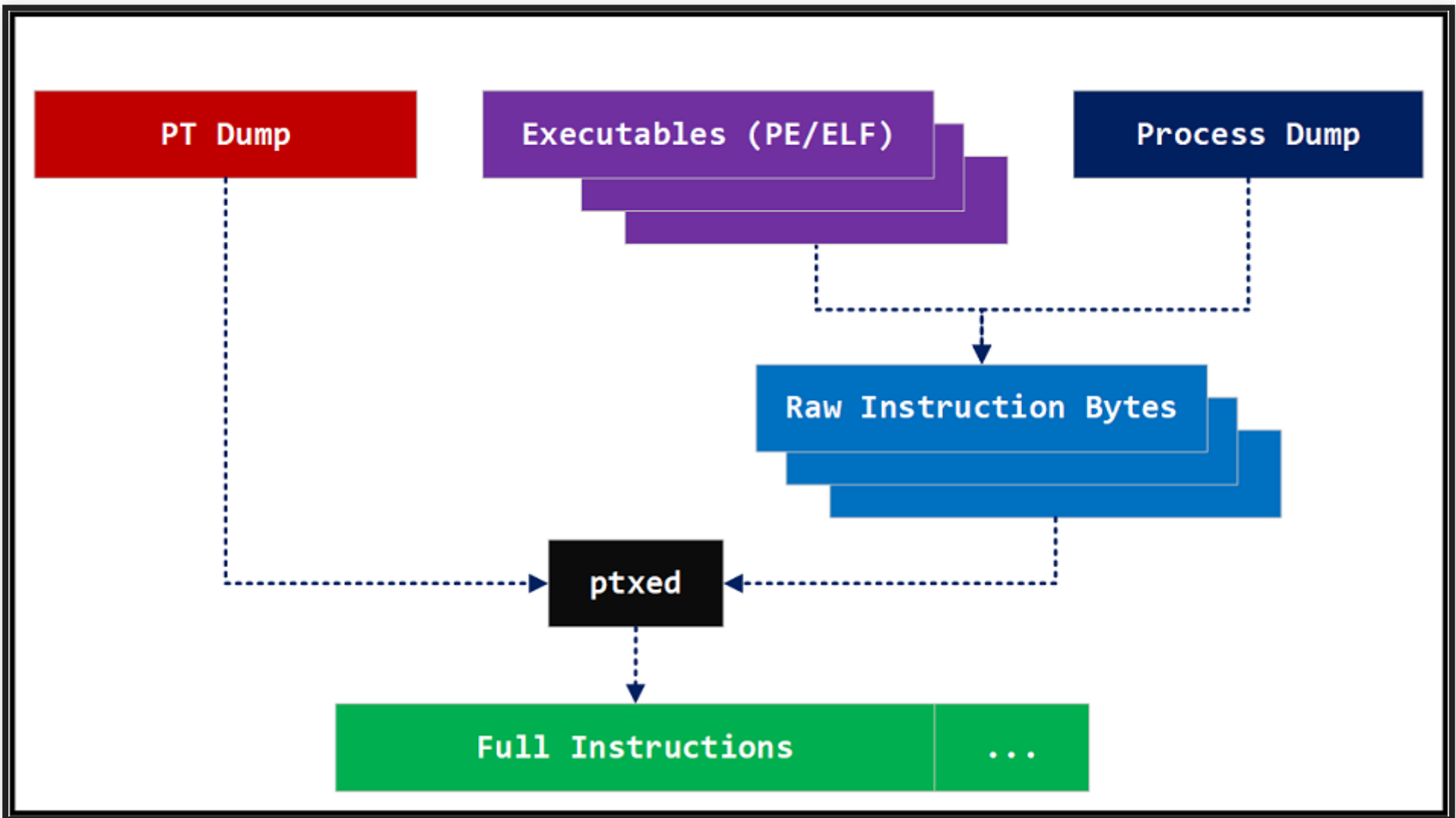
- For Linux
 - Cheat sheet for Intel Processor Trace with Linux perf and gdb

DECODING TRACE: LIBIPT

- For analysis of the recorded packets, you can use **libipt** from Intel.
- Libipt is a standard library that can decode Intel PT packets. It provides basic tools like ptdump and ptxed.

INSTRUCTION SOURCES

- Intel PT logs only control flow changes. To decode Intel PT trace, we need image file where the instructions are executed.
 - Because Intel PT doesn't save instruction bytes or memory contents, you need to provide the instruction bytes for each IPs (Instruction Pointers).



MISSING INSTRUCTION SOURCE

If we don't have matching image for certain regions of the code execution, we might lose some execution information.

- JIT code execution: there is no static image file available
- Shellcode: the shellcode instructions is not static in many cases

COMPRESSED RECORDING

One barrier in utilizing Intel PT in real world is the huge CPU time requirements to process Intel PT trace file.

- The trace file is compressed and it needs to be decompressed before used for any purposes.
- Libipt library can be used for decoding process but it is more of single threaded operation.

BRANCHES

- Similar to **LBR**, Intel PT works by recording branches.
 - At runtime, when CPU encounters any branch instructions like "je", "call", "ret", it will record the actions taken with the branch. With conditional jump instructions, it will record taken (T) or not taken (NT) using 1 bit.

INDIRECT CALLS AND JUMPS

With indirect calls and jumps, it will record with target addresses.

UNCONDITIONAL BRANCHES

For unconditional branches like jumps or calls, it will not record the change because you can deduce the target jump address from the instructions.

IP COMPRESSION

The IP (Instruction Pointer) to be recorded will be compared with last IP recording using one of the FUP, TIP, TIP.PGE or TIP.PGD packets. If upper parts of the address bytes overlap between them, those matching bytes will be suppressed in the current packet. Also, for the near return instructions, if the return target is the next instruction of the call instruction, it will not be recorded because it can be deduced from the control flow.

PACKETS

Descriptions on the packets used in IPT compression can be found from [Intel® 64 and IA-32 Architectures Software Developer's Manual](#).

There are many packets used to implement the recording mechanism. But, there are few important packet types that play main roles.

PSB (PACKET STREAM BOUNDARY)

The PSB packet works as a synchronization point for a trace-packet decoding. It is the boundary in the trace log where the decompression process can be performed independently without any side effects. This offset is referred as "sync offset" in libipt library code because this is an offset in the trace file where you can safely start decoding the following packets.

TIP (TARGET IP)

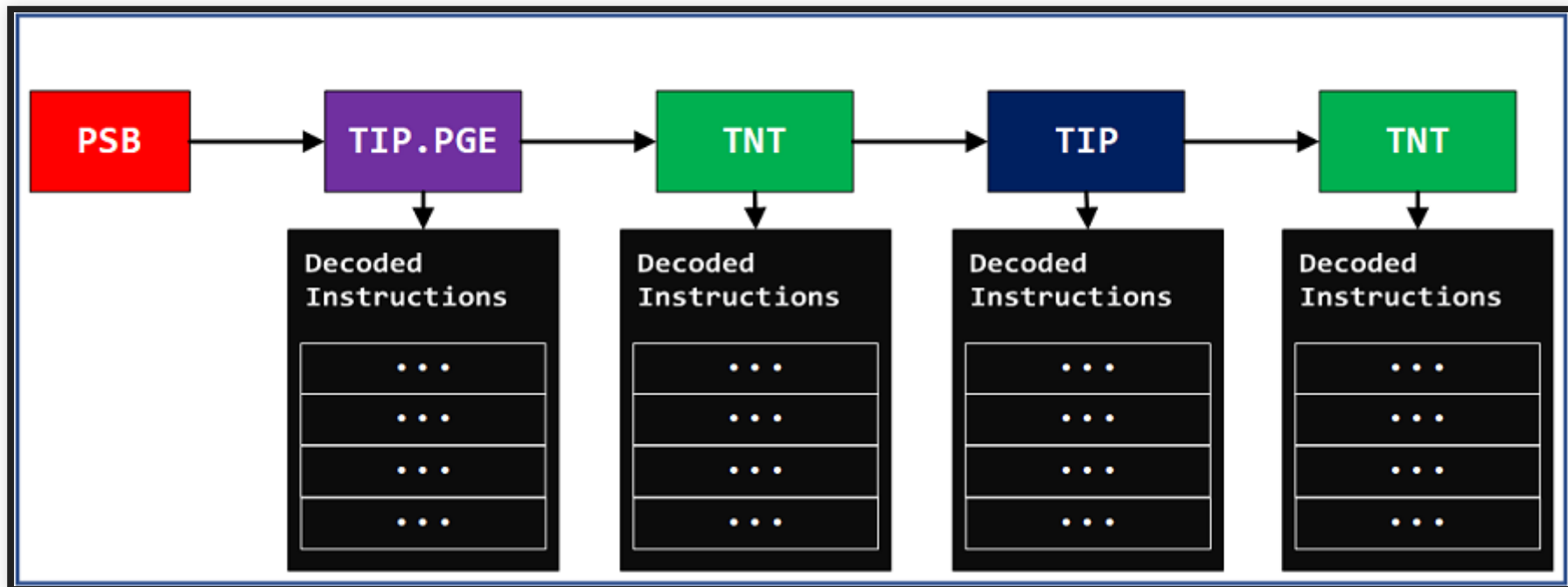
TIP packets indicate the target IPs. This information can be used as the base point of instruction pointer.

TNT (TAKEN NOT-TAKEN)

- TNT packet is used to indicate whether conditional branch is taken or not.
 - Any unconditional branch jumps will not be recorded because those flow control can be deduced from the process image.

COMPRESSION OVERVIEW

The IntelPT log can be used to reconstruct full instruction executions and control flow changes with help from instruction bytes. Without instruction bytes, it only gives partial view of full instruction executions.



EXAMPLE TRACE LOG

- Here is a snippet of a IPT trace log, which is converted to text form using ptdump from libipt.

```
0000000000000001c  psb
0000000000000002c  pad
0000000000000002d  pad
0000000000000002e  pad
```

- It starts with PSB packet which indicates the position where you can safely decode following packets.
 - There are some paddings and timing related packets, too.

TIP.PGE

At offset 3db, there is a tip.pge packet. It means the instruction pointer is located at the location indicated by the packet which is 00007ffbb7d63470.

```
...  
000000000000003db  tip.pge    3: 00007ffbb7d63470  
000000000000003e2  pad  
000000000000003e3  pad
```

INSTRUCTIONS FOR TIP.PGE LOCATION

From the process image, we can identify the address 00007ffbb7d63470 of tip.pge points to the following instructions.

```
seg000:00007FFBB7D63470      mov     rcx, [rsp+20h]
seg000:00007FFBB7D63475      mov     edx, [rsp+28h]
seg000:00007FFBB7D63479      mov     r8d, [rsp+2Ch]
seg000:00007FFBB7D6347E      mov     rax, gs:60h
seg000:00007FFBB7D63487      mov     r9, [rax+58h]
seg000:00007FFBB7D6348B      mov     rax, [r9+r8*8]
seg000:00007FFBB7D6348F      call    sub_7FFBB7D63310
```

TIP.PGE

The tip packet indicates that the code started execution from address 00007ffbb7d63470 and continued execution until it encountered call instruction at 00007FFBB7D6348F. Because the call is not indirect one, the call destination is pre-determined at compile time, so this tip.pge packet expands to the inside call instructions. The additional instructions from call target address 00007FFBB7D63310 will be decoded.

```
seg000:00007FFBB7D63310      sub     rsp, 48h
seg000:00007FFBB7D63314      mov     [rsp+48h+var_28], rcx
seg000:00007FFBB7D63319      mov     [rsp+48h+var_20], rdx
seg000:00007FFBB7D6331E      mov     [rsp+48h+var_18], r8
seg000:00007FFBB7D63323      mov     [rsp+48h+var_10], r9
seg000:00007FFBB7D63328      mov     rcx, rax
seg000:00007FFBB7D6332B      mov     rax, cs:7FFBB7E381E0h
seg000:00007FFBB7D63332      call    rax
```


INDIRECT CALLS

At this point, there is a indirect call happens at address 00007FFBB7D63332. The next tip packet will give the necessary information where this call is jumping. The compression removes first 4bytes of address to save space. From the packet at 3ee, we can deduce that the call target is 00007ffbb7d4fb70.

```
...  
000000000000003ee tip      2: ????????b7d4fb70  
000000000000003f3 pad  
...
```

DECODING CONTINUES

The decoding continues from 00007ffbb7d4fb70 until it encounters a conditional jump instruction at 00007FFBB7D4FB8C.

```
seg000:00007FFBB7D4FB70      mov     rdx, cs:7FFBB7E38380h
seg000:00007FFBB7D4FB77      mov     rax, rcx
seg000:00007FFBB7D4FB7A      shr     rax, 9
seg000:00007FFBB7D4FB7E      mov     rdx, [rdx+rax*8]
seg000:00007FFBB7D4FB82      mov     rax, rcx
seg000:00007FFBB7D4FB85      shr     rax, 3
seg000:00007FFBB7D4FB89      test    cl, 0Fh
seg000:00007FFBB7D4FB8C      jnz     short loc_7FFBB7D4FB95
seg000:00007FFBB7D4FB8E      bt      rdx, rax
seg000:00007FFBB7D4FB92      jnb     short loc_7FFBB7D4FBA0
seg000:00007FFBB7D4FB94      retn
```

TNT

At this point, the tnt packet will give you information whether the conditional jump is taken or not taken. The following tnt.8 packet with 2 ".." means, it didn't take two unconditional jumps.

```
000000000000003fe  tnt.8  ..
```

TNT

Next, it will encounter ret instruction at
00007FFBB7D4FB94.

...

```
seg000:00007FFBB7D4FB8C      jnz     short loc_7FFBB7D4FB95
seg000:00007FFBB7D4FB8E      bt      rdx, rax
seg000:00007FFBB7D4FB92      jnb     short loc_7FFBB7D4FBA0
seg000:00007FFBB7D4FB94      retn
```

RETURN ADDRESS RECORDING -> TIP

The return address can't be reliably determined from the image itself even though it can calculate with some emulation. Basically, "ret" is an indirect jump, where it retrieves jump address from the current SP (stack pointer). The next tip packet will give you the address where this ret instruction is returning.

```
000000000000003ff tip 2: ????????b7d63334
```

DECODING

The returned address disassembles like following and the code execution continues.

```
seg000:00007FFBB7D63334      mov     rax, rcx
seg000:00007FFBB7D63337      mov     rcx, [rsp+48h+var_28]
seg000:00007FFBB7D6333C      mov     rdx, [rsp+48h+var_20]
seg000:00007FFBB7D63341      mov     r8, [rsp+48h+var_18]
seg000:00007FFBB7D63346      mov     r9, [rsp+48h+var_10]
seg000:00007FFBB7D6334B      add     rsp, 48h
```

IPTANALYZER: PROBLEM STATEMENT

The IPT compression mechanism is very efficient and it needs help from disassembly engine to reconstruct full instructions. Even short amount of IPT trace recording can take a lot of CPU resources to decompress. One way, you can apply IP filterings to limit the output to minimize the amount of trace output. Sometimes huge trace log is inevitable for research purposes.

IPTANALYZER

IPTAnalyzer is a tool to perform parallel processing of the IPT trace logs. The tool can process Intel PT trace using Python multiprocessing library and create a basic blocks cache file. This block information can be useful in overall analysis of the control flow changes. For example, if you want to collect instructions from specific image or address range, you can query this basic block cache file to find the locations that falls into the range before retrieving full instructions.

CASE STUDY: CVE-2017-11882

CVE-2017-11882 is a vulnerability in Equation Editor in Microsoft Office. This can be a good exercise target to exercise how IPT can be used for exploit analysis. We will explain how you can use IPT and IPTAnalyzer to perform exploit analysis efficiently.

IPT LOG COLLECTION

You can use various approaches to generate IPT trace logs. I used **WinIPT** to generate trace log.

OFFICE MALWARE SAMPLE

We used malicious sample

`abbdd98106284eb83582fa08e3452cf43e22edde9e86ffb8e9380`

to reproduce the exploit condition.

IPTTOOL.EXE

Run ipttool.exe with process id and log file name.

IPTTOOL.EXE

Ex) The process id 2736 is the vulnerable Equation Editor process. The trace output will be saved into EQNEDT32.pt file.

```
C:\Analysis\DebuggingPackage\TargetMachine\WinIPT>ipttool.exe --trace 2736 EQNEDT32.pt
/-----\
|=== Windows 10 RS5 1809 IPT Test Tool ===|
|=== Copyright (c) 2018 Alex Ionescu ===|
|=== http://github.com/ionescu007 ===|
|=== http://www.windows-internals.com ===|
\-----/

[+] Found active trace with 1476395324 bytes so far
    [+] Trace contains 11 thread headers
        [+] Trace Entry 0 for TID 2520
            Trace Size: 134217728                [Ring Buffer Offset: 4715184]
            Timing Mode: MTC Packets                [MTC Frequency: 3, ClockTsc Ratio: 83]
        [+] Trace Entry 1 for TID 1CA8
            Trace Size: 134217728                [Ring Buffer Offset: 95936]
```

TAKING PROCESS MEMORY DUMP

You can use **ProcDump** or **Process Explorer** or even Windbg to take memory dump of the Equation Editor (EQNEDT32.exe). Instead of supplying individual image files to the libipt, IPTAnalyzer can use process memory dump to retrieve instruction bytes automatically.

RUNNING IPTANALYZER








For convenience, set %IPTANALYZERTOOL% as the root of the IPTAnalyzer folder in the following examples. By using decode_blocks.py, a block cache file can be generated. You need to provide -p option with IPT trace file name and -d option with process memory dump file.

```
python %IPTANALYZER%\pyipttool\decode_blocks.py -p PT\EQNEDT32.pt -d ProcessMemory\EQNEDT32.dmp -c block.cache
```

PARALLEL PROCESSING

The following shows the parallel Python processes working to decode the trace file.

python.exe	< 0.01	1,500,032 K	1,503,856 K	1688 Python	Python Software Foundation
python.exe	3.04	29,736 K	40,852 K	26484 Python	Python Software Foundation
python.exe	3.03	62,956 K	62,556 K	23788 Python	Python Software Foundation
python.exe	2.92	32,416 K	41,568 K	13748 Python	Python Software Foundation
python.exe	3.04	65,616 K	65,176 K	4516 Python	Python Software Foundation
python.exe	3.11	41,120 K	47,228 K	22392 Python	Python Software Foundation
python.exe	3.11	34,104 K	41,224 K	23436 Python	Python Software Foundation
python.exe	3.09	37,596 K	44,284 K	19216 Python	Python Software Foundation
python.exe	3.12	34,120 K	43,276 K	20032 Python	Python Software Foundation
python.exe	3.09	35,364 K	45,100 K	360 Python	Python Software Foundation
python.exe	3.00	63,288 K	62,948 K	20668 Python	Python Software Foundation
python.exe	3.06	33,660 K	43,104 K	3160 Python	Python Software Foundation
python.exe	2.98	62,980 K	62,256 K	20836 Python	Python Software Foundation
python.exe	3.12	37,404 K	43,760 K	22568 Python	Python Software Foundation
python.exe	3.09	32,564 K	42,900 K	2040 Python	Python Software Foundation
python.exe	3.03	33,020 K	40,896 K	17764 Python	Python Software Foundation
python.exe	3.11	34,204 K	44,080 K	16820 Python	Python Software Foundation
python.exe	3.09	33,664 K	42,420 K	26436 Python	Python Software Foundation
python.exe	3.07	34,764 K	43,812 K	1580 Python	Python Software Foundation
python.exe	2.90	62,956 K	62,388 K	2344 Python	Python Software Foundation
python.exe	3.11	33,020 K	41,296 K	3024 Python	Python Software Foundation
python.exe	3.09	35,448 K	44,016 K	25584 Python	Python Software Foundation
python.exe	3.10	34,772 K	42,396 K	23420 Python	Python Software Foundation
python.exe	3.00	61,956 K	61,776 K	6480 Python	Python Software Foundation

 python.exe	2.97	50,904 K	50,440 K	23328 Python	Python Software Foundation
 python.exe	2.84	55,468 K	53,012 K	14128 Python	Python Software Foundation
 python.exe	3.06	65,652 K	65,172 K	23752 Python	Python Software Foundation
 python.exe	2.98	33,812 K	42,612 K	1480 Python	Python Software Foundation
 python.exe	2.95	60,644 K	59,920 K	25960 Python	Python Software Foundation
 python.exe	3.00	35,284 K	44,648 K	25536 Python	Python Software Foundation
 python.exe	3.07	61,932 K	61,420 K	22804 Python	Python Software Foundation
 python.exe	3.10	34,440 K	44,292 K	17360 Python	Python Software Foundation
 python.exe	3.12	33,936 K	44,604 K	5488 Python	Python Software Foundation

DUMP EQNEDT32 MODULE BLOCKS

Because the EQNEDT32 main module has the vulnerability and an abnormal code execution pattern will happen inside or around the module address range, we want to enumerate blocks inside EQNEDT32 main module range, which is between 00400000 and 0048e000.

```
0:011> lmvm EQNEDT32
Browse full module list
start          end          module name
00000000`00400000 00000000`0048e000  EQNEDT32  (deferred)
...
```

DUMP_BLOCKS.PY: ENUMERATE BASIC BLOCKS

The dump_blocks.py tool can be used to enumerate any basic blocks inside specific address range.

```
python %IPTANALYZER%\pyipttool\dump_blocks.py -p PT\EQNEDT32.pt -d ProcessMemory\EQNEDT32.dmp -C 0 -c blocks.c
```

FULL LOG OF BASIC BLOCKS IN ADDRESS RANGE

The command will generate a full log of basic blocks matching the address range. Probably the transition into shellcode will happen at the end of the code execution from the vulnerable module, we focus on the basic block patterns at the end of the log. Notice the "sync_offset=2d236c" shows the location of PSB packet for these last basic block hits. This sync_offset value can be used to retrieve instructions around that point.

FULL LOG OF BASIC BLOCKS IN ADDRESS RANGE

```
...
> 00000000004117d3 () (sync_offset=2d236c, offset=2d26f4)
    EQNEDT32!EqnFrameWinProc+0x2cf3:
00000000`004117d3 0fbf45c8      movsx    eax,word ptr [rbp-38h]

> 000000000041181e () (sync_offset=2d236c, offset=2d26f4)
    EQNEDT32!EqnFrameWinProc+0x2d3e:
00000000`0041181e 0fbf45fc      movsx    eax,word ptr [rbp-4]

> 0000000000411869 () (sync_offset=2d236c, offset=2d26f4)
    EQNEDT32!EqnFrameWinProc+0x2d89:
00000000`00411869 33c0          xor      eax,eax

> 000000000042fad6 () (sync_offset=2d236c, offset=2d26fc)
    EQNEDT32!MFEnumFunc+0x12d9:
```

DUMP EQNEDT32 MODULE INSTRUCTIONS

Now, we know that the last basic blocks from EQNEDT32 module were executed inside "sync_offset=2d236c" PSB block. The dump_instructions.py script can be used to dump full instructions. Options like -S (start sync_offset) and -E (end sync_offset) can be used to specify sync_offset range.

```
python %IPTANALYZER%\pyipttool\dump_instructions.py -p ..\PT\EQNEDT32.pt -d ..\ProcessMemory\EQNEDT32.dmp -S 0
```

LOCATING THE CODE TRANSITION

With the output from `dump_instructions.py`, you can easily identify where the code transition from EQNEDT32 to shellcode happens.

```
...
Instruction: EQNEDT32!EqnFrameWinProc+0x2d8b:
00000000`0041186b e900000000      jmp      EQNEDT32!EqnFrameWinProc+0x2d90 (00000000`00411870)
Instruction: EQNEDT32!EqnFrameWinProc+0x2d90:
00000000`00411870 5f          pop      rdi
Instruction: EQNEDT32!EqnFrameWinProc+0x2d91:
00000000`00411871 5e          pop      rsi
Instruction: EQNEDT32!EqnFrameWinProc+0x2d92:
00000000`00411872 5b          pop      rbx
Instruction: EQNEDT32!EqnFrameWinProc+0x2d93:
00000000`00411873 c9          leave
Instruction: EQNEDT32!EqnFrameWinProc+0x2d94:
00000000`00411874 c3          ret
Instruction: EQNEDT32!MFEnumFunc+0x12d9:
00000000`0042fad6 c3          ret
```

RET INSTRUCTIONS

From the above instruction listing, you can notice that there are two "ret" instructions at 00411874 and 0042fad6.

```
Instruction: EQNEDT32!EqnFrameWinProc+0x2d94:  
00000000`00411874 c3          ret  
Instruction: EQNEDT32!MFEnumFunc+0x12d9:  
00000000`0042fad6 c3          ret
```


CODE CONTROL TRANSFER

After these two "ret" instructions, the code transfers into a non-image address space.

```
Instruction: 00000000`0019ee9c bac342baff    mov     edx,0FFBA42C3h
Instruction: 00000000`0019eea1 f7d2        not     edx
Instruction: 00000000`0019eea3 8b0a        mov     ecx,dword ptr [rdx]
Instruction: 00000000`0019eea5 8b29        mov     ebp,dword ptr [rcx]
```

Notice that the instruction at 00000000`0019ee9c doesn't have any matching module name retrieved which means, it has a high probability of being shellcode loaded inside dynamic memory.

NEXT STAGE SHELLCODE

Following the shellcode, we can locate the position where next stage shellcode is executed at 0019eec1 with "jmp rax" instruction. Basically, we have full listing of shellcode execution in the Intel PT log.

```
Instruction: 00000000`0019eeb7 0567946d03    add    eax,36D9467h
Instruction: 00000000`0019eebc 2d7e936d03    sub    eax,36D937Eh
Instruction: 00000000`0019eec1 ffe0        jmp    rax
```

DUMP_INSTRUCTIONS.PY: DUMP INSTRUCTION IN ADDRESS RANGE

These are the next stage shellcode dumped by dump_instructions.py script.

```
Instruction: 00000000`00618111 9c          pushfq
Instruction: 00000000`00618112 56          push    rsi
Instruction: 00000000`00618113 57          push    rdi
Instruction: 00000000`00618114 eb07        jmp     00000000`0061811d
Instruction: 00000000`0061811d 9c          pushfq
Instruction: 00000000`0061811e 57          push    rdi
Instruction: 00000000`0061811f 57          push    rdi
Instruction: 00000000`00618120 81ef40460000 sub    edi,4640h
Instruction: 00000000`00618126 81ef574b0000 sub    edi,4B57h
Instruction: 00000000`0061812c 8dbfbc610000 lea     edi,[rdi+61BCh]
Instruction: 00000000`00618132 81c73b080000 add     edi,83Bh
Instruction: 00000000`00618138 5f          pop     rdi
Instruction: 00000000`00618139 5f          pop     rdi
```

CONCLUSIONS

Intel PT is a very useful technology that can be used for defensive and offensive security research. IPTAnalyzer is a tool that uses libipt library to speed up analysis using IPT trace logs. The exploit example here shows the benefits of using IPTAnalyzer tool to generate block cache file and use it for basic exploit investigation. Without help from Intel PT, this process can be tedious and might rely more on the instinct of the researchers. With Intel PT, there are potentials of automating this process and detecting malicious code activities automatically.