INTEL VIRTUALIZATION EXTENSIONS

INTRO TO VTX/VTD/EPT

Diego Porras

AGENDA

- Disclaimer
- UEFI primer
- VTx
- VMX mode
- ~~VTd~~
  ~~EPT~~

GOALS

- Motivate to continue exploring low level topics (intel specific), particularly virtualization and its application to security
- Set the base to introduce confidential compute later (TXT, TDX, SGX)
- Set the base to discuss security model of virtualization technologies

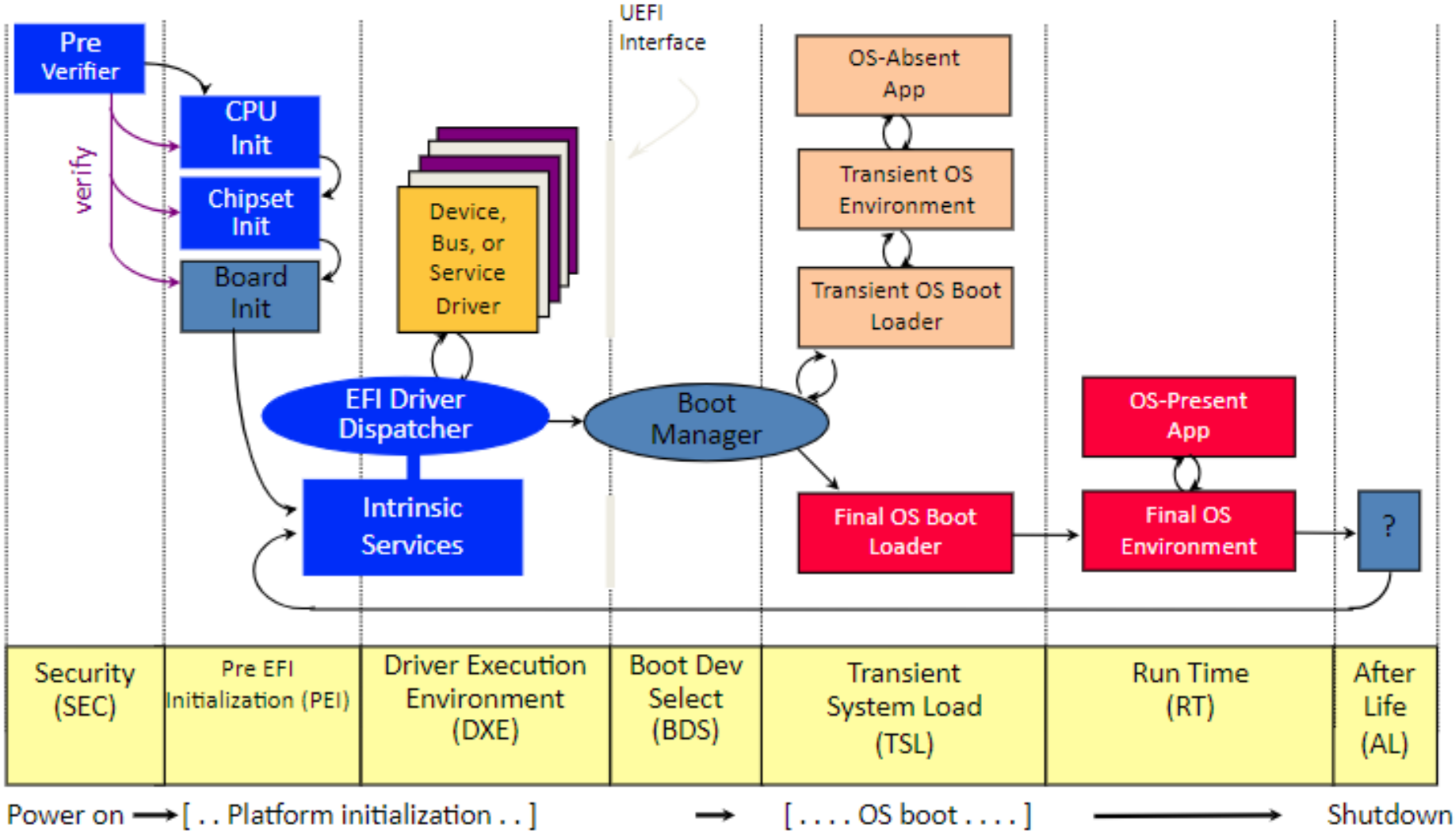
DISCLAIMER

- These are entirely my opinions and doesn't reflect those of my employer. This talk is worked and shared in my own free time.
- This is for illustrative purposes and I'm not responsible to any use the attendees can make of this information
- This is a conceptual introduction, no coding today. I hope to give you a roadmap of pre-reqs so we can have a formal class later :D

UEFI

- UEFI is a specification.
- A reference implementation can be found: EDK2 is a very popular one.

UEFI



https://raw.githubusercontent.com/tianocore/tianocore.github.io/master/images/PI_Boot_Phases.JPG

```
370     EFI_STATUS
371     EFIAPI
372     UefiMain (
373         IN EFI_HANDLE ImageHandle,
374         IN EFI_SYSTEM_TABLE* SystemTable
375         )
376     {
377         EFI_STATUS efiStatus;
378
379         //
380         // Find the PI MpService protocol used for multi-processor startup
381         //
382         efiStatus = gBS->LocateProtocol(&gEfiMpServiceProtocolGuid,
383                                         NULL,
384                                         &_gPiMpService);
385         if (EFI_ERROR(efiStatus))
386         {
387             Print(L"Unable to locate the MpServices protocol: %r\n", efiStatus);
388             return efiStatus;
389         }
390
391         //
392         // Call the hypervisor entrypoint
393         //
394         return ShvOsErrorToError(ShvLoad());
395     }
396
```

HTTPS://GITHUB.COM/IONESCU007/SIMPLEVISOR/BLOB/MASTER/UEFI/SHVOS.C#L372

```
46  INT32
47  ShvLoad (
48      VOID
49      )
50  {
51      SHV_CALLBACK_CONTEXT callbackContext;
52
53      //
54      // Attempt to enter VMX root mode on all logical processors. This will
55      // broadcast a DPC interrupt which will execute the callback routine in
56      // parallel on the LPs. Send the callback routine the physical address of
57      // the PML4 of the system process, which is what this driver entrypoint
58      // should be executing in.
59      //
60      callbackContext.Cr3 = __readcr3();
61      callbackContext.FailureStatus = SHV_STATUS_SUCCESS;
62      callbackContext.FailedCpu = -1;
63      callbackContext.InitCount = 0;
64      ShvOsRunCallbackOnProcessors(ShvVpLoadCallback, &callbackContext);
65
66      //
67      // Check if all LPs are now hypervised. Return the failure code of at least
68      // one of them.
69      //
70      // Note that each VP is responsible for freeing its VP data on failure.
71      //
72      if (callbackContext.InitCount != ShvOsGetActiveProcessorCount())
73      {
74          ShvOsDebugPrint("The SHV failed to initialize (0x%lX) Failed CPU: %d\n",
75                          callbackContext.FailureStatus, callbackContext.FailedCpu);
76          return callbackContext.FailureStatus;
77      }
```

https://github.com/ionescu007/SimpleVisor/blob/989d33b1bc6569965d7aad3bd50a8d35fa4c359e/shv.c#L47

WHAT DOES IT TAKE TO RUN A HYPERVISOR?

- VTx support : Check for existence of the feature
- VMX Mode/VMXON – This requires some fixed values in CR0 and CR4 and VMXON region
- VMCS structure – Data structure used to configure individual vCPU, this is indicates what the hypervisor will support ie. Instructions that trap conditionally, who handles specific interrupts.
- VMWRITE – This changes values in VMCS, we use it a lot. Always do a VMCLEAR on your VMCS region before starting setup
- VMPTRLD/VMPRSC - Used to read VMCS and load it to the current CPU, this is the step that gets us ready for the nex stem
- VMLAUNCH -> Whit the vCPU ready this enters the guest life-cycle in VMX-nonRoot mode
- VMEXIT -> This is where the magic happens. There are conditional VMEXITs and instructions that ALWAYS cause VMEXIT, this is useful to ie. Write virtualization aided debuggers or anti-anti-virtualization hypervisor

VMXON ENTER

VMXON—Enter VMX Operation

Opcode/ Instruction	Op/En	Description
F3 0F C7 /6 VMXON m64	M	Enter VMX root operation.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Puts the logical processor in VMX operation with no current VMCS, blocks INIT signals, disables A20M, and clears any address-range monitoring established by the MONITOR instruction.[1]

The operand of this instruction is a 4KB-aligned physical address (the VMXON pointer) that references the VMXON region, which the logical processor may use to support VMX operation. This operand is always 64 bits and is always in memory.

INTEL MANUALS 3B, 3C, 3D ARE YOUR BEST FRIENDS

- Pay particular attention to the following:
  - Chapter 24 — Introduction to Virtual Machine Extensions.
  - Chapter 25 — Virtual Machine Control Structures.
  - Chapter 26 — VMX Non-Root Operation.
  - Chapter 27 — VM Entries. Describes VM entries.
  - Chapter 28 — VM Exits.
  - Chapter 29 — VMX Support for Address Translation
  - Chapter 31 — VMX Instruction Reference to multiple guest software environments.

PMT4, CR0, CR3, CR4, PAGING, IDT,GDT, MSR…

- If you are interested in hypervisor development, but still struggle with some of the concepts:

  - OST2 – Architecture 1001: x86-64 Assembly LINK (If you are completely new to low-level)
  - **OST2 - Architecture 2001: x86-64 OS Internals LINK**
  - That OSDev UEFI - LINK
  - OST2 - Architecture 4021: Introductory UEFI LINK (Difficult to follow, but great to setup a debugging environment)

THANKS AND NEXT STEPS

- This was a VERY theoretical introduction to hypervisors based on Intel virtualization extensions.

  Next chapter will be some coding and fun.
  Please watch pre-reqs, ask questions and comments and hope to see you next time