Breaking Hardware-Enforced Security with Hypervisors

Joseph Sharkey, Ph.D.
Chief Technology Officer /
Vice President of Advanced Programs
Siege Technologies
www.siegetechnologies.com

This work was sponsored in part by the Air Force Research Laboratory (AFRL) and Air Force Office of Scientific Research (AFOSR) under contracts FA8750-C-0235, FA9550-11-1-0267, and FA9550-14-C-0019

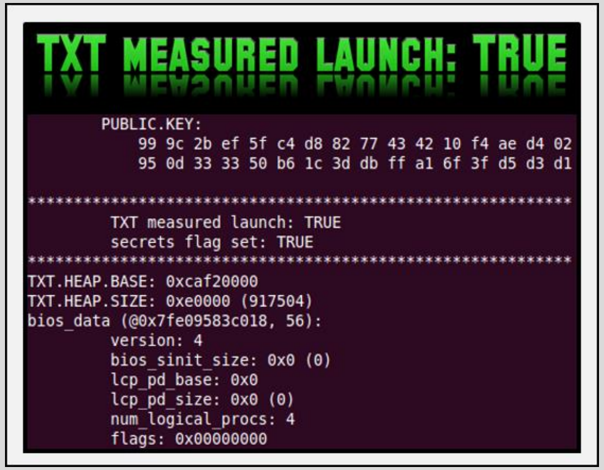
Presentation Content

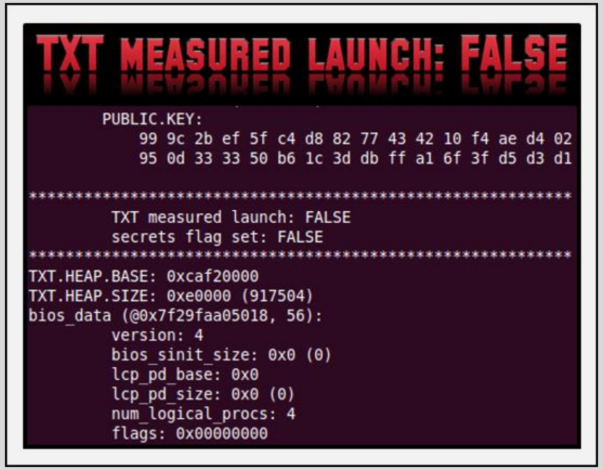
- Background on modern hardware-enforced security primitives for PC platforms
- Compromising Intel TXT with a Hypervisor rootkit
- Compromising AES-NI with a Hypervisor rootkit
- Implications
- Near Term Solutions: What can I do about it?

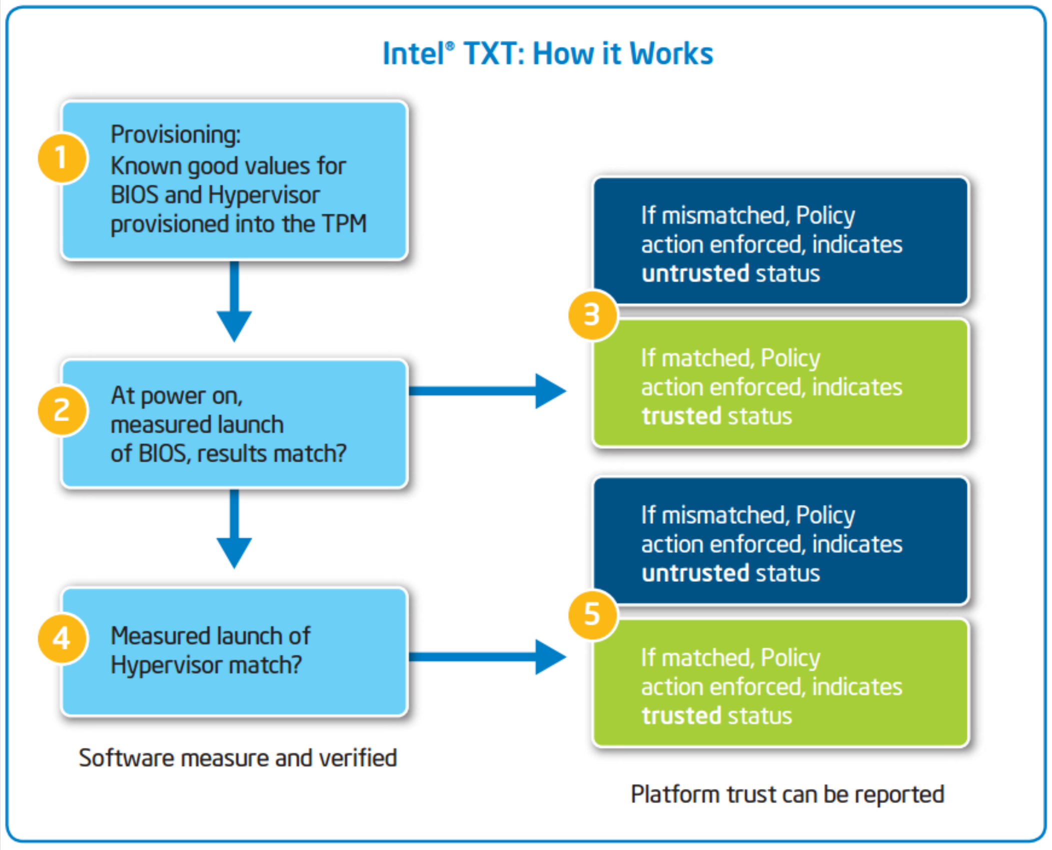
Trusted Boot (*tBoot*)

tBoot is **open source software** that makes **use of Intel's TXT**

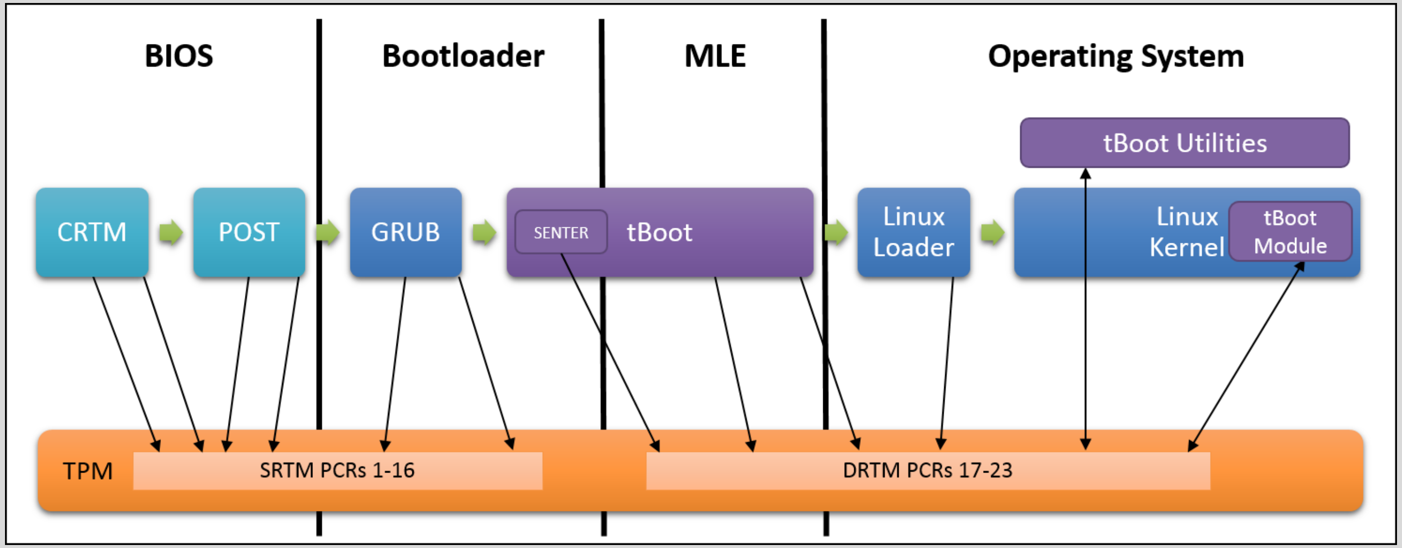
- Code written and released by Intel engineers
- *tBoot* is the de facto standard code-base for DRTM leveraging Intel's TXT extensions
  - Used by GRUB, Xen, VMWare ESXi, etc.
- Launch Control Policies (LCP) handle failed measurement; are settable by system administrator
  - ***Halt policy*** prevents boot on invalid measurement
  - ***Continuation policy*** allows boot, but notifies system of invalid state







Dynamic Root of Trust Measurement (DRTM)



DRTM is common across PC/laptop/server platforms and is used by *tboot, Xen, Vmware ESXi, etc.*

- **Establishment of trusted environment is delayed until some time after platform has booted**
  - Eliminates the need to trust early boot software, no longer need to reboot to start a chain of trust
  - Completely remove BIOS & early bootloaders from the Trusted Computing Base

- **Atomic "measure and launch"** operation ensures a clean initial state; ***TPM*** stores ***integrity measurements***, and optionally ***sealed storage and remote attestation***

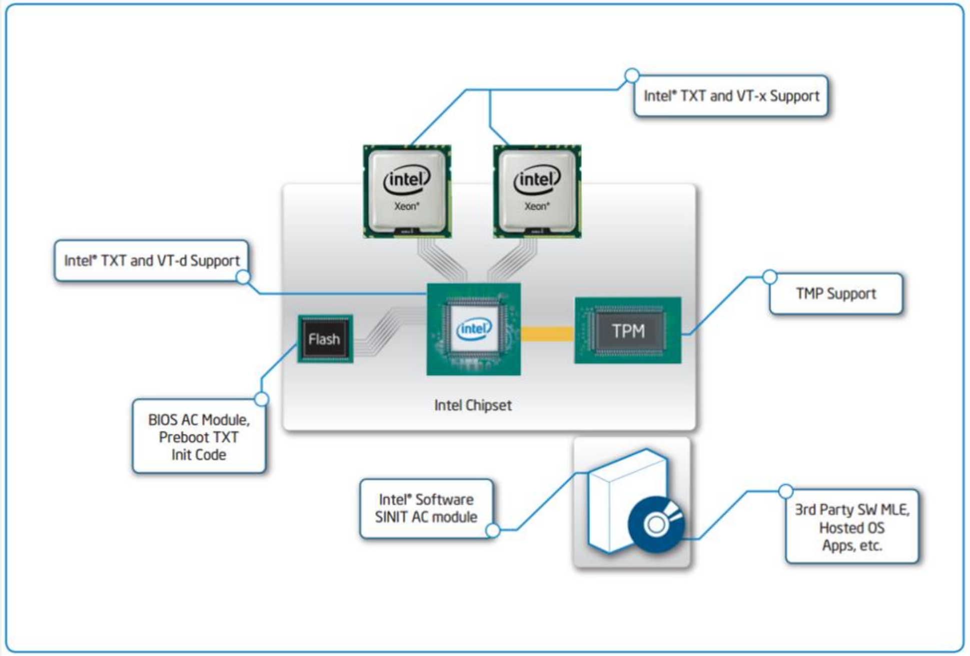
- Most popular for PC platforms; addresses challenges of Static Root of Trust approaches
  - Used by *tBoot, Xen, VMWare ESXi, etc*

DRTM Implementations

Intel’s Trusted Execution Technology (TXT)

- A set of hardware extensions and primitives
  - Safer Mode Extensions (SMX) in the CPU
  - Chipset support including VT-d, TPM v1.2, LPC Bus v1.1
- **Provides a secure way to launch a measured environment**
  - GETSEC[SENTER] instruction is used to atomically reset some chipset, TPM state, halt other cores and execute a trusted code module (SINIT)
  - SINIT module can then pass execution to a known/trusted kernel

Intel’s TXT Overview



Intel’s GETSEC Instruction Leaf Functions

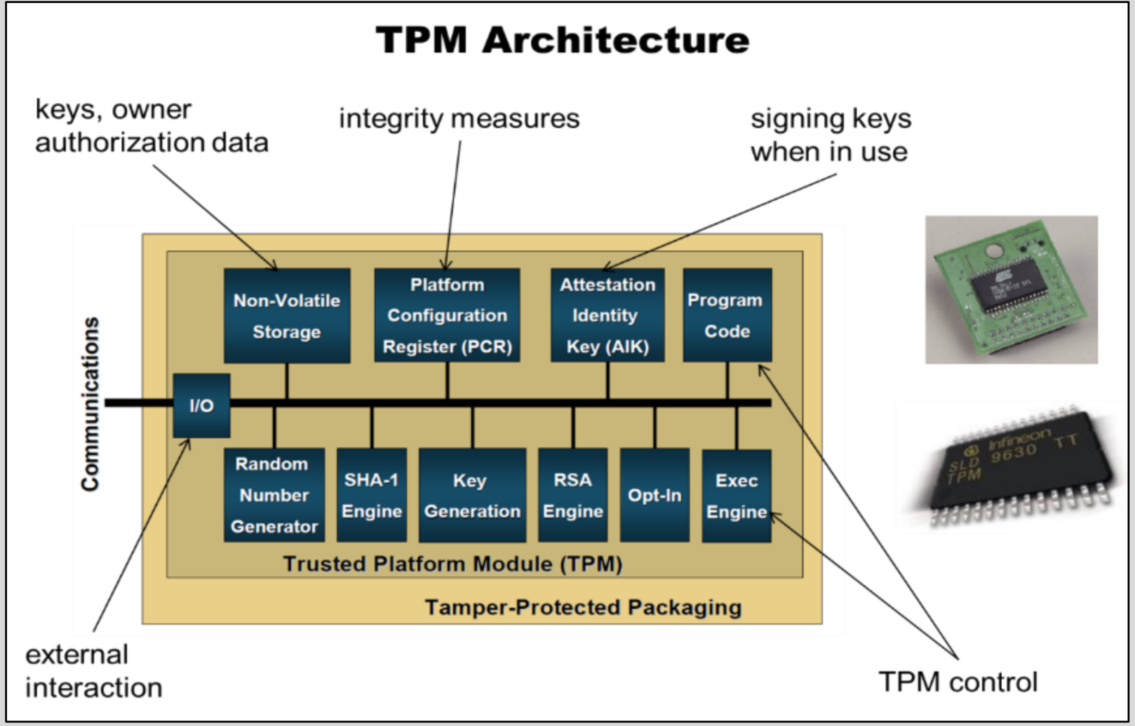
Index (EAX)	Leaf function	Description
0	CAPABILITIES	Returns the available leaf functions of the GETSEC instruction
1	Undefined	Reserved
2	ENTERACCS	Enter
3	EXITAC	Exit
4	SENTER	Launch an MLE
5	SEXIT	Exit the MLE
6	PARAMETERS	Return SMX related parameter information
7	SMCTRL	SMX mode control
8	WAKEUP	Wake up sleeping processors in safer mode
9 - (4G-1)	Undefined	Reserved

AMD’s Secure Virtual Machine (SVM)

- Very similar to Intel’s TXT, small differences
  - SKINIT – “Secure Kernel Initialization” instruction
  - Chipset extensions and support

Trusted Platform Module (TPM)

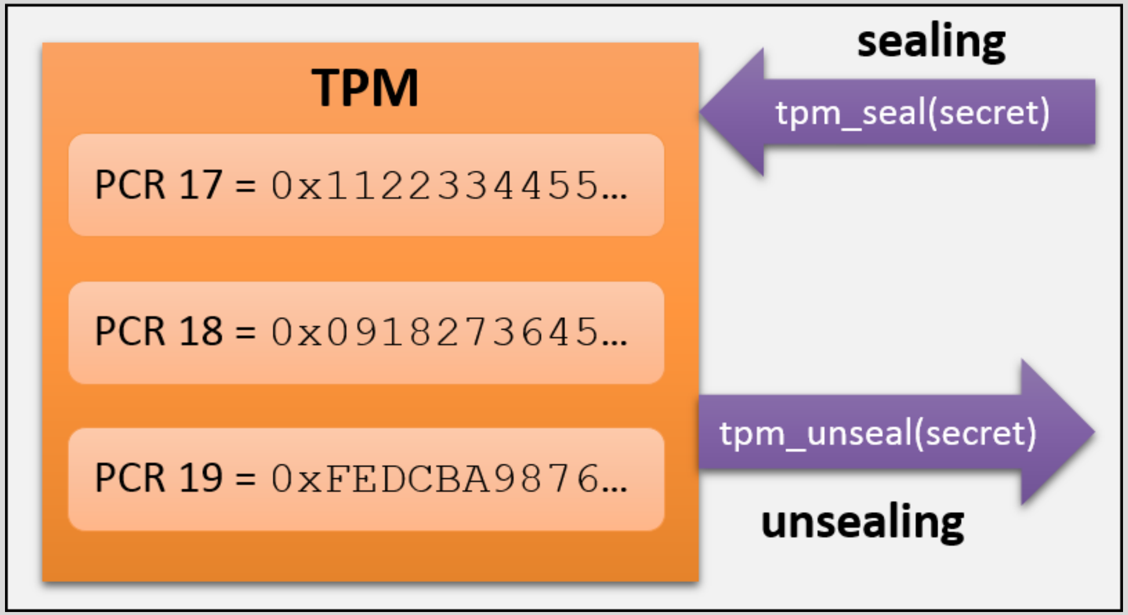
- **Services provided by the TPM include:**
  - Platform Configuration Registers (PCRs)
  - Locality based access enforcement
  - Sealed Storage
  - Remote Attestation



- **Platform configuration registers (PCRs)**
  - Hash accumulator used to track system configuration
    - $PCR_{N'}$ = SHA-1 (PCR_N | Value)
  - Computationally infeasible to set PCR to a specified value
    - (ext(A), ext(B)) ≠ (ext(B), ext(A))
  - Some registers are used for SRTM (0 - 15 and others DRTM (17 – 23), 16 is a debug PCR

Trusted Platform Module (TPM)

- **Sealed Storage**
  - Binds data to system configuration
  - Secrets can only be accessed when the TPM PCRs are in the proper state

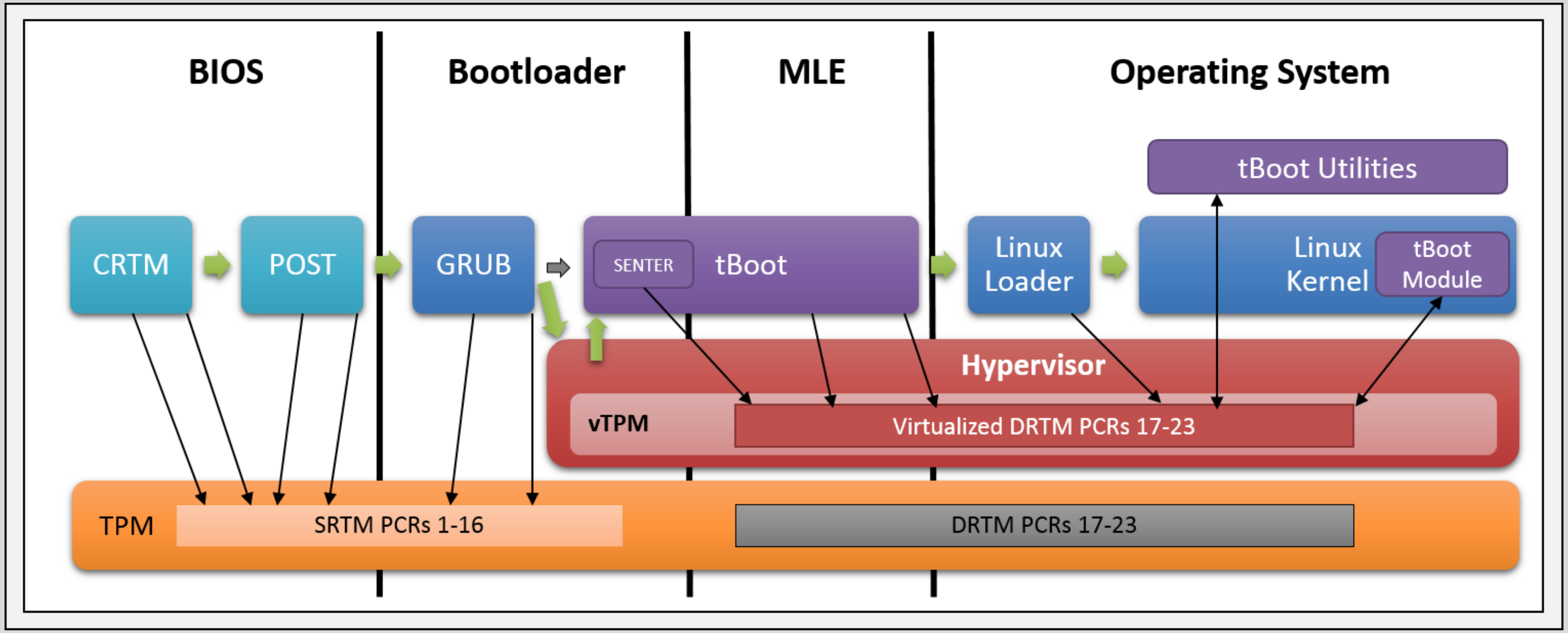


- **Remote Attestation**
  - Challenge response protocol with nonce to eliminate replay
  - TPM performs a quote operation (reports PCR values) and signs it with a key held internally (key exchange happens at setup)

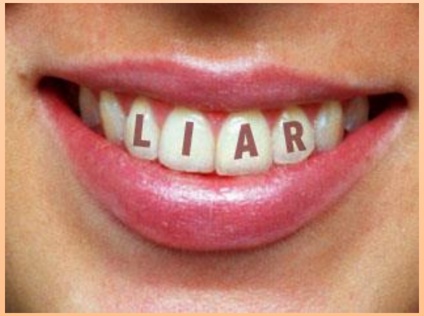
Summary of Previously Demonstrated Attacks

- *Invisible Things Lab, 2009*: **Malicious SMM** [4]
  - System Management Mode (SMM) code is not included in TXT system measurement; malicious SMM code can subvert the root of trust
  - Intel has discussed a solution to address SMM in a TXT environment (STM), but does not yet have any commercial implementations available for testing or use in trusted platforms [1].
- *Invisible Things Lab, 2009*: **TXT Chipset Misconfiguration** [2]
  - A misconfiguration in chipset VT-d settings leave MLE vulnerable to DMA attack
  - Misconfiguration issue was subsequently patched by Intel via an updated sinit software module
- *Invisible Things Lab, 2011:* **Vulnerabilities in TXT AC module** [5]
  - Buffer overflow in ACPI DMAR table allows attacker to gain code execution inside the signed executable
  - Bug was patched by Intel via updated SINIT module release
- *Johannes Winter, 2009, 2011*: **TPM hardware attacks**
  - Showed an attacker can monitor [6] and/or manipulate TPM bus communications [3]

SENTER Emulation Attack with Hypervisor Rootkit



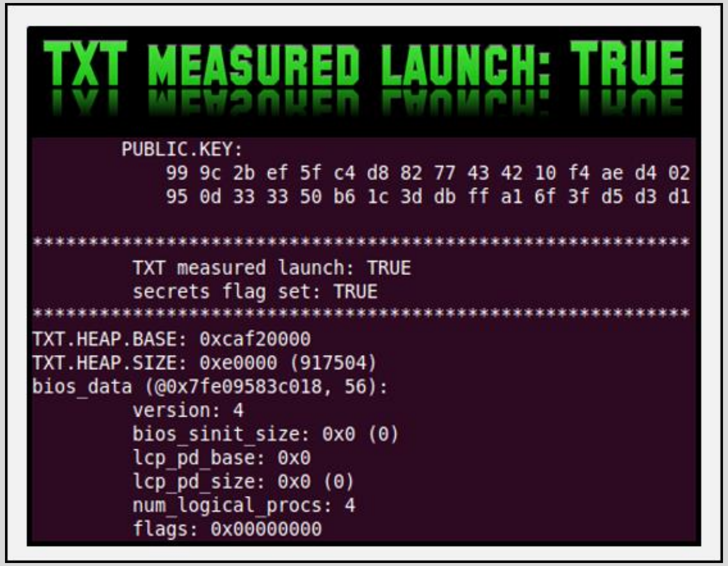
SENTER emulation attack virtualizes the DRTM establishment process, and lies about the state of the system



- Approach: Launch thin hypervisor before *tBoot* (e.g. via *GRUB* loader)
  - Intercept and emulate the GETSEC[SENTER] & other SMX instructions
  - Intercept and emulate TPM interaction to fake local attestation
  - Intercept and emulate TXT heap, private memory regions, etc.
- Results: Proof of Concept constructed against tBoot
  - *tBoot* thinks (and reports) that the **system successfully boots** into a trusted state
  - Undermines the security of *tBoot* DRTM with any policy, including the most restrictive “halt” policy

SENTER Emulation Attack Discussion

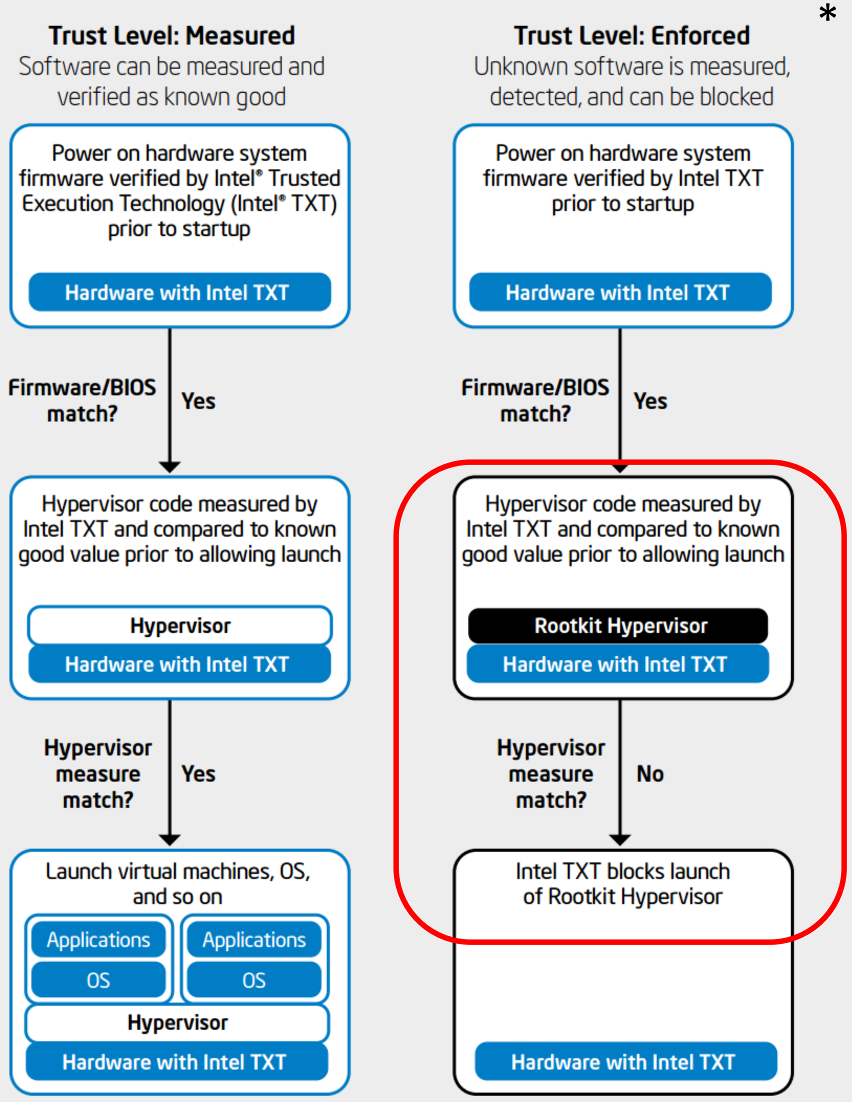
- Load ***a custom thin hypervisor rootkit*** first and then ***run tBoot inside the virtual machine*** container
  - **Trap SMX instructions (e.g. GETSEC[SENTER])**
    - Emulate those instruction's, using the pseudo-code provided by Intel in the Developer's Manual
    - Modified as desired of course ☺
  - **AC module can either be skipped** or run in the virtualized environment so the chipset is reinitialized to the specification
    - Again, modify/filter operations as desired of course ☺
  - Shadow memory is used to **emulate TPM and provide falsified PCR measurements**



- No matter what policy *tBoot* is configured with (since GETSEC[SENTER] isn't run, any Launch Control Policy can be ignored by the rootkit), it continues to boot and the *txt-stat* command reports "`TXT measured launch: TRUE`"
  - ***System thinks it is in a trusted state, even though a rogue hypervisor is running underneath the kernel!***
  - ***Dumping PCR values from within Linux shows the same exact state as when TXT succeeds***
    - *Because it isn't actually talking to the real TPM; it is talking to the hypervisor rootkit virtualized TPM*

Should this type of attack succeed?

- According to the documentation, ***NO!***
  - TXT should prevent the launch of a measured environment if a system can not be measured and verified
- Intel's show-case example states that TXT is capable of detecting the presence of a hypervisor rootkit
  - This is only possible when sealed storage or remote attestation is used
  - tBoot (written & maintained by Intel developers as a TXT reference implementation) does not use sealed storage or remote attestation out-of-the-box!...it is left for the user to implement



*

Figure 3. Intel® Trusted Execution Technology helps protect virtualized server environments.

* Whitepaper: Evolution of Integrity Checking with Intel® Trusted Execution Technology: an Intel IT Perspective
http://www.intel.com/content/dam/doc/white-paper/intel-it-security-trusted-execution-technology-paper.pdf

Fundamental Problem

- DRTM implementations require a single atomic instruction to be executed to initiate the root of trust
  - ***How can you trust an untrusted system to execute even 1 single assembly instruction safely?***
- Both AMD and Intel implementations of DTRM allow a **hypervisor to gain execution whenever a guest tries to execute a root-of-trust instructions**
  - This ***prevents a guest operating system from ousting its underlying hypervisor*** by setting up an MLE of its own
  - UNFORTUNATLEY, this design ***also allows an attacker to setup a thin-hypervisor at boot time and virtualize/emulate all TXT instructions*** and TPM interactions

Fundamental Tradeoff:

Allow attacker to kick-out trusted hypervisor by executing GETSEC[SENTER]; OR
Provide the mechanisms necessary for hypervisor rootkit to emulated GETSEC[SENTER]

AES-NI Instructions

- Improve performance of cryptographic operations by adding support directly into the CPU (more than an order of magnitude faster in some cases!)
- Use XMM (128-bit) / YMM (256-bit) CPU registers
- Round **keys & data are provided directly as a parameter** to the instructions

Instruction	Description[2]
AESENC	Perform one round of an AES encryption flow
AESENCLAST	Perform the last round of an AES encryption flow
AESDEC	Perform one round of an AES decryption flow
AESDECLAST	Perform the last round of an AES decryption flow
AESKEYGENASSIST	Assist in AES round key generation
AESIMC	Assist in AES Inverse Mix Columns
PCLMULQDQ	Carryless multiply (CLMUL).[3]

AESENC

66 0F 38 DC /r
AESENC xmm1, xmm2/m128

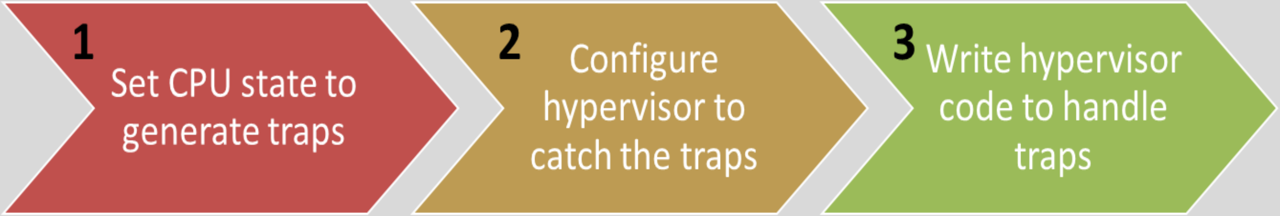
Perform one round of an AES encryption flow, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.

Applications using default crypto libraries (e.g. libcrypt & wincrypt) inherently use AES-NI when it is available whether they realize it or not

XMM/YMM registers provide the round keys as well as the data to encrypt/decrypt

Compromising AES-NI: Summary

- Leverage design features of x86/64 architecture to undermine AES-NI
- Hypervisor configures the CPU to generate an exception anytime an AES-NI is executed



- Hypervisor catches the exceptions, logs information
- This generic approach is not tailored to a specific piece of software, and is not noticeable to the OS

Use hypervisor to man-in-the-middle AES-NI operations, extracting both the encryption key as well as plain text data

Inducing VMExits

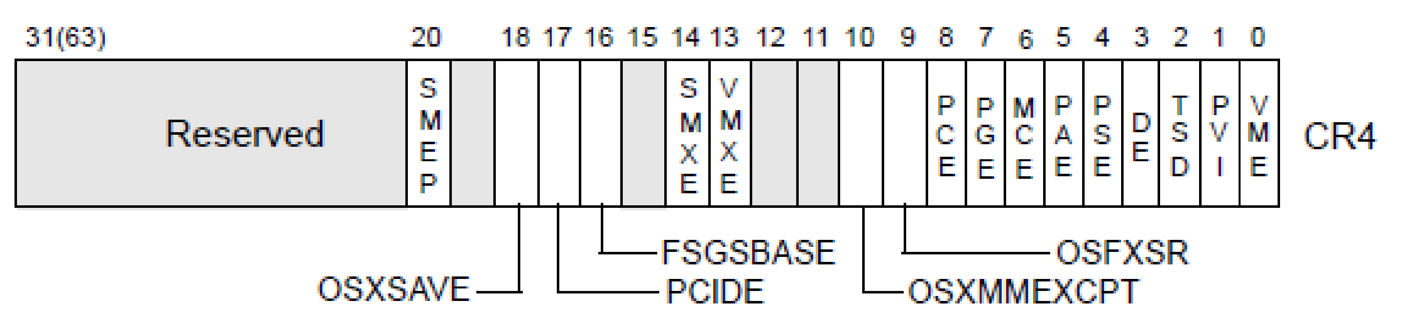
- Unlike GETSEC, the hardware does not directly provide a way to force all AES-NI instructions to trap to the hypervisor
  - Need to get creative ☺
  - All AES-NI instructions use XMM/YMM registers, and can therefore generate “Exceptions Type 4”
- Force all AES-NI instructions to trap to the hypervisor
  - Configure the CPU to trip one of the entries in the table to the right
  - Set VMCS to route the appropriate exception (#UD or #NM) to the hypervisor
  - Configure Hypervisor to catch the exception

Exceptions Type 4, from the Instruction Set Reference Manual

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix
			X	X	VEX prefix: If XFEATURE_ENABLED_MASK[2:1] != ‘11b’. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix
	X	X	X	X	If any corresponding CPUID feature flag is ‘0’
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1
Stack, SS(0)			X		For an illegal address in the SS segment
				X	If a memory address referencing the SS segment is in a non-canonical form
General Protection, #GP(0)	X	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH
Page Fault #PF(fault-code)		X	X	X	For a page fault

Inducing VMExits (continued…)

CR4.OSXSAVE/ CR4.OSXFXSR



- Setting these bits forces all instructions using SSE/AVX to cause exceptions
  - When both bits are used together ensures legacy SSE instructions and VEX prefix generate traps
- Allows the desired events (AES-NI) to be seen by the hypervisor, but also many other instructions
  - Hypervisor must look at the opcode that causes the trap to filter out which ones are AES-NI and which are not

Detailed Results Discussion

Hypervisor Output

```
1   aeskeygenassist $0x1, %xmm0, %xmm1
    key = fd747b58b0877b984473157d9f24c6d0
    RCON = 01
    DATA = c5ab4247c6aedafa387481d8fd19ad75
    RESULT = e7172146162146e7db36b47037b470db
```

⋮

```
31 aesdeclast %xmm0, %xmm2
   key = fd747b58b0877b984473157d9f24c6d0
   DATA = facb9dd822f78741c20eeb6318644f34
   RESULT = e9f847a024dee998ec556055abf32cfd
```

⋮

```
38 xorps %xmm9, %xmm2
   op1 = bd90228046bb85ec85304030d8904d8d
   op2 = e9f847a024dee998ec556055abf32cfd
   RESULT = 5468652062656c746965206573636170
```

OpenSSL Output

```
pid: 3112
salt: 3F54F5750BD88DD3
key: FD747B58B0877B984473157D9F24C6D0
iv: BD90228046BB85EC85304030D8904D8D

plaintext:
00000000  54 68 65 20 62 65 6c 74  69 65 20 65 73 63 61 70  |The beltie escap|
00000010  65 64 21 20 3a 2d 28 0a                           |ed! :-(.|
00000018

ciphertext:
00000000  e2 58 11 3b a3 67 3a 3a  db 59 ca 4a 0d 03 8a 8d  |.X.;.g::.Y.J....|
00000010  f0 be b6 21 04 93 f5 73  5e 7d b7 6b 54 c8 1b e5  |...!...s^}.kT...|
00000020

decrypted ciphertext:
00000000  54 68 65 20 62 65 6c 74  69 65 20 65 73 63 61 70  |The beltie escap|
00000010  65 64 21 20 3a 2d 28 0a                           |ed! :-(.|
00000018
```

While OpenSSL is encrypting & decrypting data, the hypervisor sees all the cryptographic operations, and readily identifies the key and plain text

Successfully used hypervisor to man-in-the-middle AES-NI operations, extracting both the encryption key as well as plain text

AES-NI Interception: What's the Catch?

- The hypervisor is able to extract the keys and grab clear text data in real time in a generic way that isn't implementation dependent
  - Surely the hypervisor could also set a breakpoint on specific library functions, but that approach would be more tailored to a specific implementation

- BUT…The devil is in the details
  - Our initial implementation incurs non-negligible performance impact (system is usable, but noticeably slower)
  - Implementation could be optimized, perhaps with some simplifying assumptions

Who is affected?

- A variety of systems
  - Laptops, desktops, workstations, servers
    - *Especially those relying on TXT for trusted boot & AES-NI for encryption*
  - Cloud computing infrastructure
    - *What if someone compromises the trusted hypervisor (e.g. via VM breakout; or malicious employee, etc.), bypasses the DRTM, and starts sniffing AES-NI operations? They can compromise SSL, VPN, disk encryption, etc. – many of the technologies that are supposed to keep you "safe" in the cloud*
  - Not Operating System specific – these issues are inherent in the architecture and can be realized on any OS

Will I know if I'm affected?

- Probably not
  - Many sysadmins (and even software developers writing the code!) don't know if they are relying on AES-NI currently
    - Generally because they rely on library calls and don't know how the library implementation is done. Most libraries now use AES-NI by default when it is available.
  - TXT is quite complex; key elements for a single implementation a modern PC platform is defined by **nine specifications** and encompasses hardware implementations from **at least three hardware vendors and eight software components**
    - Staggering complexity leaves the system administrator responsible to make configuration decisions for options that are not completely understood

NEAR-TERM SOLUTIONS:
HOW CAN WE PROTECT OURSELVES?

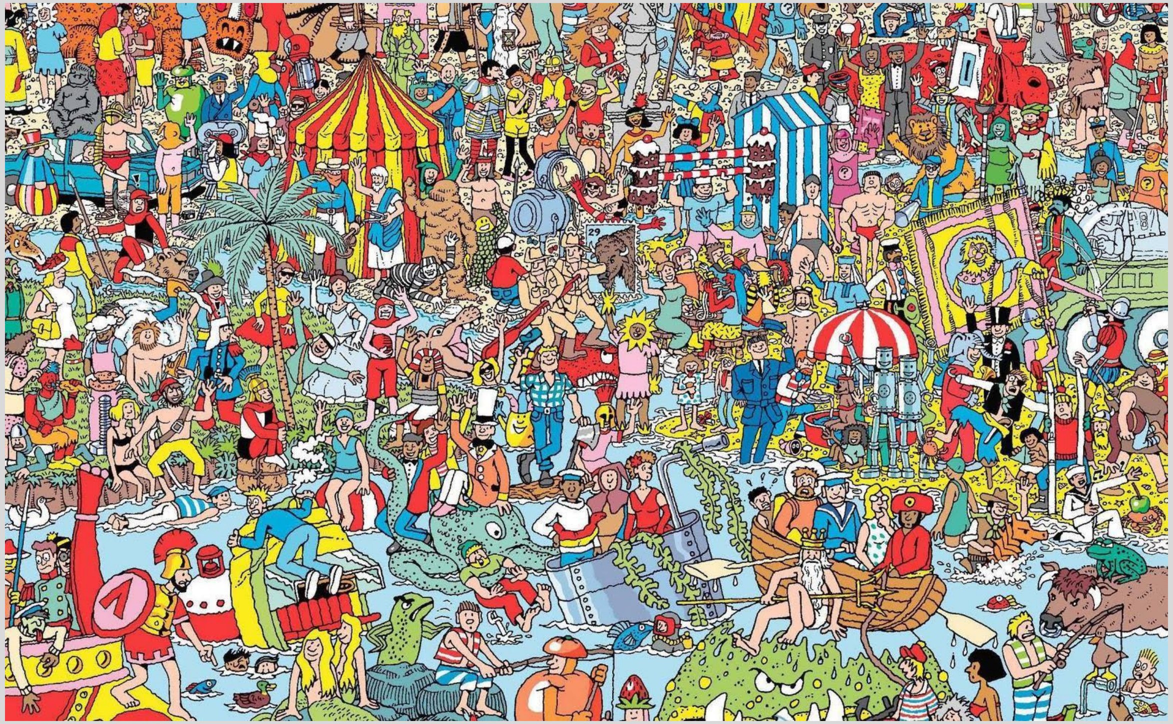
I want to encrypt data in the cloud – is it hopeless?!

- Initial experimentation indicates significant performance implications imposed by this hypervisor approach
  - Could make it less practical for wide-scale use
  - Although implementation optimizations might be able to overcome this challenge

- **To be safe, you can always use a software-only implementation of AES (not relying on the AES-NI instructions) to avoid compromise**
  - This would make it harder for a hypervisor rootkit to identify AES operations

Hide in the Noise

- Hypervisor rootkit has the privilege to see all guest operations, but must somehow find what its looking for within all the bits & bytes of the system ("semantic gap")
  - This semantic gap is the biggest challenge for hypervisor rootkits, and we can take advantage of it!



Using a software AES implementation

VS

Using AES-NI

Sealed Storage!!

- ***The attack succeeds when sealed storage and remote attestation are not implemented***
  - Attack bypasses a default installation of *tBoot* that does not leverage sealed storage or remote attestation
    - There aren’t even optional tBoot configurations to use sealed storage.
  - There are Linux command line utilities to seal/unseal secrets against the TPM, but you are on your own to script something out using them
- **CONCLUSION: Sealed storage should not be optional!!**
  - If something unique is locked (sealed) in the TPM, the attacker can lie about PCR state but will ultimately fail to produce the secret value during an unseal operation

Sealed storage and remote attestation are the ONLY mechanisms that provide trusted mechanisms to report state

Sealed Storage: Challenges

- Even if sealed storage is used, there are some pitfalls to be aware of:
  - Make sure you don’t rely on a sealed secret that can be predicted or possibly obtained by an attacker at runtime
    - If so, the attacker can report the predicted/captured value during an emulated TPM unseal operation, without ever actually having had access to the TPM!!
  - Make sure you extend PCRs after unsealing your secrets
    - Otherwise an attacker can just re-unseal your secrets at runtime!
  - Ensure your sealed secret doesn’t stay resident outside of the TPM at runtime
  - Be careful how you verify the sealed secrets at boot time
    - EXAMPLE: Use disk encryption, seal the key in the TPM, and assume we are safe if our disk gets mounted properly (if TPM unseal fails we won’t be able to decrypt the disk)
    - PITFALL: Attacker at runtime can grab the disk key from memory, and then just report it in the right place/time during boot

If an attacker can predict or obtain/access your sealed secrets, then they can emulate the unseal operation, bypassing even the protections afforded by sealed storage!!

Sealed storage: Recommendations

- You really need to seal a value that is displayed to the user ONLY to verify trusted state during boot
  - Can be text, a photo, etc. – something the system displays to the user early in the boot process
  - After the user acknowledges the “secret” (e.g. hits enter to continue) the secrets need to be scrubbed, the PCRs extended, and then the system can continue boot
- Remote attestation can be used to accomplish this for a server/headless configuration
  - Rather than attesting state to the user, state is attested to a remote server
  - The same process as above should be utilized: Perform attestation, scrub secrets, extend PCRs, continue boot

Thank You!

Questions / Comments?

Joseph Sharkey
@sharkey_joe
https://www.linkedin.com/in/joseph-sharkey-45aa0b8

References:

- [1] http://invisiblethingslab.com/resources/misc09/Quest%20To%20The%20Core%20(public).pdf
- [2] http://invisiblethingslab.com/resources/misc09/Another%20TXT%20Attack.pdf
- [3] Johannes Winter, "A Hijacker's Guide to the LPC bus", https://online.tugraz.at/tug_online/voe_main2.getvolltext?pCurrPk=59565
- [4] http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf
- [5] http://www.invisiblethingslab.com/resources/2011/Attacking_Intel_TXT_via_SINIT_hijacking.pdf
- [6] Johannes Winter, "Eavesdropping Trusted Platform Module Communication"