A gente descobre depois

João Gabriel Trombeta
João Paulo Taylor Ienczak Zanette
Ranieri Schroeder Althoff

8 de Abril de 2018

Conteúdo

Capítulo 1

Máquinas Virtuais

1.1 Sobre Máquinas Virtuais

De maneira sucinta, máquinas virtuais (VM — *Virtual Machines*) são computadores sendo executados por outros computadores. Chama-se de **Guest** a máquina virtual em si, e de **Host** o *hardware* que oferece recursos para executar a VM. No host, uma camada de software chamada hypervisor permite a execução de múltiplas máquinas virtuais independente sem uma única máquina física, cada uma executando seu próprio SO.

1.2 Hypervisor

Também chamado de Monitor de Máquina Virtual (VMM — *Virtual Machine Monitor*), um **Hypervisor** é um componente (seja *hardware*, *software* ou *firmware*) responsável por criar e executar uma VM, sendo o *Host* o computador em que o Hypervisor é executado.

O Hypervisor é responsável pela camada de abstração entre o host e os guests, realizando o gerenciamento de recursos, uma vez que cada guest trabalha na ilusão de que todos os recursos de hardware são seus. Cada VM deve ser isolada para evitar que uma VM possa comprometer o funcionamento de outra, por isso toda interação com o meio físico é intermediada pelo Hypervisor, que é fortemente protegido das VM.

1.3 Falhas em Hypervisors

1.3.1 Xen Hypervisor

Uma falha de segurança detectada com relação a Hypervisors foi explorada no Xen Hypervisor (criado pelo Xen Project, composto por membros da The Linux Foundation), em que é possível chamar uma função arbitrária alterando a tabela de *Hypercalls* (semelhante a uma *vtable*). Uma *Hypercall* é *software trap*

do Hypervisor para executar operações privilegiadas (como atualizar tabelas de página).

Para explorar a falha, primeiramente é necessário descobrir a localização da tabela de *Hypercalls*. Para isso, deve-se procurar pela assinatura da página (dada pelo *checksum* de seu conteúdo). Porém, como essa página não possui um conteúdo tão previsível, é difícil de localizá-la. Em compensação, a tabela de argumentos dos *Hypercalls* possui um formato previsível, já que seu conteúdo — que é o número de argumentos de cada *Hypercall* — é fixo, e portanto seu *checksum* também é previsível. Além disso, a tabela de argumentos sempre se encontra na página seguinte à tabela de *Hypercalls*, e portanto, ao encontrar uma, se tem a localização da outra. Aliado à possibilidade de leitura e escrita de código arbitrário, feito através de falhas nas regras de verificação de segurança de escrita em páginas do *Hypervision*, é possível então efetuar escape de máquina virtual (i.e. acessar recursos do *Host* que não pertencem à máquina virtual).

Capítulo 2

AMD Memory Encryption

2.1 AMD Secure Processor

Grande parte das funções de criptografia executadas em um AMD utilizam um processador dedicado e independente, o AMD Secure Processor (AMD-SP, antigamente chamado de Platform Security Processor), que garante que componentes sensíveis à segurança não recebam interferência do software dos processadores principais.

O AMD-SP roda um kernel seguro de código fechado, que pode executar tarefas do sistema assim como tarefas de terceiros confiáveis, tendo o administrador controle sobre quais tarefas de terceiros são designadas ao processador. Além disso, o Secure Processor possui uma SRAM dedicada e acesso direto ao CCP, que é composto por um gerador de números aleatórios, várias engines para o processamento de algoritmos de criptografia, e um bloco para o armazenamento de chaves.

2.2 Security Memory Encryption

O AMD Secure Memory Encryption (SME) é um mecanismo que pode ser utilizado para criptografar os dados que vão para a DRAM, com a finalidade de evitar ataques físicos. Durante o boot, o AMD Secure Processor gera uma chave que será usada para criptografar e descriptografar os dados que transitam pela DRAM. Como o a engine de criptografia está dentro do chip o impacto das operações é pequeno.

A figura 2.1 mostra como ocorre a criptografia. A engine é posicionada entre o OS/Hypervisor, onde dada a chave o dado é criptografado antes de ser salvo na DRAM. Ao passar da DRAM para o SO novamente, os dados são descriptografados.

Esse tipo de segurança não previne ataques advindos de um Hypervisor comprometido, uma vez que ele possui acesso aos dados de maneira direta, esse tipo de segurança tem com o objetivo evitar ataques como probe attack na

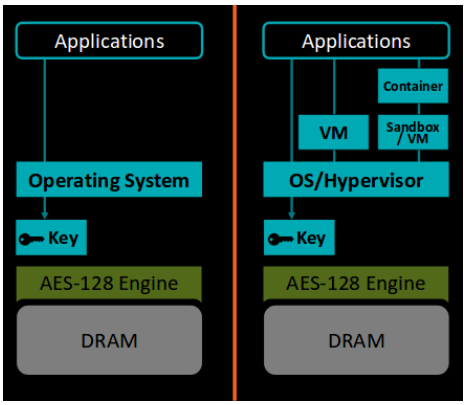

Figura 2.1: Método de criptografia do SME

DRAM, instalação de hardware que possa acessar a memória do guest, ataques que possam capturar dados de DIMM e NVDIMM.

Para utilizar SME, é necessário verificar se o processador possui suporte para esse recurso, o que pode ser verificado através da chamada de CPUID Fn8000_001F, e que durante o boot o bit 23 de SYSCFG MSR esteja definido como 1 para sinalizar que esse recurso está habilitado. Após isso, ao fazer acesso à DRAM, é visto o último bit mais significativo do endereço, chamado de C-bit, que define se o dado deve ou não ser criptografado.

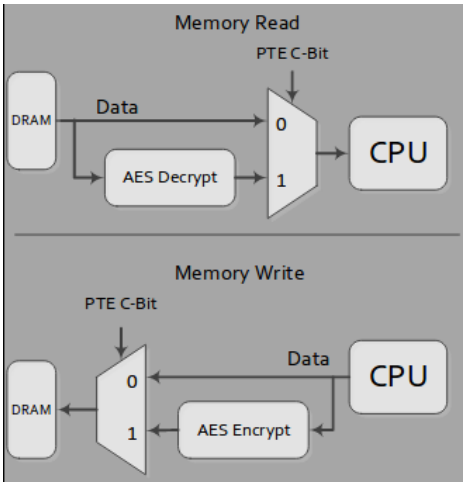

Figura 2.2: Leitura e escrita no SME

Como visto em 2.2, para a leitura, antes do dado passar para CPU, duas versões do dado são inseridas como entrada de um mux, um com o dado como estava na DRAM e outra com o dado após passar pelo circuito responsável pela

criptografia. O controle do mux recebe o bit mais significativo do endereço, o C-bit, caso seja 1 significa que o dado está criptografado e a CPU precisa da informação descriptografada, caso seja 0 significa que o dado pode ser passado direto para a CPU.

Para a escrita a lógica é a mesma, caso o C-bit seja 1 o dado deve ser criptogrado antes de ser inserido na DRAM, caso seja 0 o dado pode ser salvo diretamente.

Ainda existe uma variação chamada Transparent SME, onde tudo é criptografado. Nesse caso não é necessário suporte do SO, tendo em vista que não é preciso fazer o controle de quais endereços serã criptografados e quais não. O processo de acesso à memória ocorre da mesma forma que em SME.

2.3 Secure Encrypted Virtualization

2.4 Utilização do SME e SEV

2.4.1 *Set-up* e comunicação

A comunicação entre o AMD-SP e o x86 se dá por meio de registradores MMIO (*Memory Mapped IO*), chamados de *"mailbox registers"*. Um *mailbox* essencial *"Command buffer"*, que será lido/escrito pelo *Driver* responsável pela implementação do SEV para enviar comandos relacionados ao SEV para o *firmware*. Primeiramente é necessário, além de alterar o MSR, definir o endereço de memória em que ficará o *Command Buffer*, separado em dois registradores: `CmdBufAddr_Hi` e `CmdBufAddr_Lo` (32 bits mais e menos significativos, respectivamente). Internamente, o registrador do *Command Buffer* é chamado de `CmdResp`, e é utilizado da forma:

1. O driver (em x86) altera o `CmdResp` com:
   - Bit 31 em 0 (simbolizando a emissão de um comando);
   - Bits 30 a 26 mantidos em 0 (pois são reservados);
   - Bits 25 a 16 com o ID do comando a ser emitido;
   - Bits 15 a 1 também são mantidos em 0 (por serem reservados);
   - Bit 0 pode ser colocado em 1 para informar o *firmware* para habilitar uma interrupção no x86 ao completar o comando, do contrário pode ser mantido em 0.
2. O AMD-SP executa o comando;
3. Após completar o comando, o AMD-SP escreve no `CmdResp` com o resultado da execução do comando, da forma:
   - Bit 31 em 1 para indicar que se trata de uma resposta;
   - Bits 15 a 0 com o código de status descritos na tabela 2.1.

Status	Código
SUCCESS	0000h
INVALID_PLATFORM_STATE	0001h
INVALID_GUEST_STATE	0002h
INVALID_CONFIG	0003h
INVALID_LENGTH	0004h
ALREADY_OWNED	0005h
INVALID_CERTIFICATE	0006h
POLICY_FAILURE	0007h
INACTIVE	0008h
INVALID_ADDRESS	0009h
BAD_SIGNATURE	000Ah
BAD_MEASUREMENT	000Bh
ASID_OWNED	000Ch
INVALID_ASID	000Dh
WBINVD_REQUIRED	000Eh
DFFLUSH_REQUIRED	0009h
INVALID_GUEST	0010h
INVALID_COMMAND	0011h
ACTIVE	0012h
HWERROR_PLATFORM	0013h
HWERROR_UNSAFE	0014h
UNSUPPORTED	0015h
INVALID_PARAM	0016h

Tabela 2.1: Relação dos códigos de status dados pelo AMD-SP. Mais detalhes sobre a tabela estão em [1]

2.4.2 Comandos

INIT

O comando `INIT`, intuitivamente, inicializa a plataforma. Isso é feito carregando os dados de persistência relacionados ao SEV, vindos de algum armazenamento não-volátil, e inicializando o contexto da plataforma. Geralmente, os casos em que este não é o primeiro comando executado são os que primeiramente utilizam comandos como `PLATFORM_STATUS` para determinar a versão da API.

Requisitos: O estado da plataforma deve ser `PSTATE.UNINIT`;

Ações: 1. O *firmware* carrega os dados para sua memória privada;

2.

Parâmetros:

Bibliografia

[1] AMD, “Secure encrypted virtualization api version 0.14,” tech. rep.