White Paper | # TECHNICAL GUIDANCE FOR MITIGATING EFFECTS OF CIPHERTEXT VISIBILITY UNDER AMD SEV

REVISION 5.10.22

This white paper is a technical explanation of what the discussed technology has been designed to accomplish. The actual technology or feature(s) in the resultant products may differ or may not meet these aspirations. Each description of the technology must be interpreted as a goal that AMD strived to achieve and not interpreted to mean that any such performance is guaranteed to be fully achieved. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated.

INTRODUCTION

AMD Secure Encryption Virtualization (SEV) technologies are designed to help protect data inside guest virtual machines (VMs) from outside entities such as the hypervisor. AMD SEV focuses on protecting data "in-use", including data that is stored in CPU registers and private guest memory. It utilizes a hardware AES engine to encrypt and decrypt guest data as it is written to memory.

While the first generation of SEV technology focused only on protecting guest data in memory with encryption, subsequent technologies offer enhanced isolation by encrypting guest register state across VM transitions (SEV-ES) [1] and through memory integrity protection (SEV-SNP) [2]. Integrity protection ensures that untrusted entities, such as the hypervisor, cannot overwrite encrypted guest data. This protects against vectors like replay attacks.

Although access to the plaintext in SEV is limited to the VM which has the correct key, an adversary may be able to read the encrypted contents of guest memory (ciphertext) and in some cases may be able to use this information to deduce information about the guest execution or its data. This type of side-channel attack may be possible by either a malicious hypervisor in the current implementations of SEV or an attacker with physical access to the system and ability to monitor the DDR bus between the CPU and DRAM.

This whitepaper discusses techniques guest software can utilize to limit the impact of this type of side-channel, especially for highly sensitive software like cryptographic algorithms, that run in AMD SEV guests. AMD believes this information may be helpful to software developers seeking to maximize the level of security that can be obtained within an AMD SEV guest.

BACKGROUND: AMD MEMORY ENCRYPTION

Current AMD SEV technologies utilize an AES-128 engine in the on-chip memory controllers to perform in-line encryption and decryption of guest memory. Each guest is assigned a unique Address Space ID (ASID) which corresponds to a set of unique keys. When a new guest is created, memory encryption keys are created randomly using an on-die random number generator [3]. Each key is then written to one of the on-chip memory controllers and used only for accesses by the assigned ASID.

The encryption and decryption of guest data is done using an XOR-Encrypt-XOR construct. The "tweak" value used in the XOR operation is based on the physical address where the data is being stored. Additionally, starting in AMD 2nd Generation EPYC Processors, an additional random value generated at system boot time is used in the tweak generation.

As AES uses a 16B (byte) block size for encryption, the XOR-Encrypt-XOR operation ensures that identical plaintext will encrypt to different ciphertext at different 16B locations in memory. However, the same plaintext at the same location in memory will always encrypt to the same ciphertext value.

BACKGROUND: CPU REGISTER STATE

AMD SEV-SNP technology supports a feature called VMSA Register Protection described in section 15.36.17 of the AMD64 Architecture Programmer's Manual [4]. When this feature is enabled, the CPU obfuscates guest register values before saving them in memory. The obfuscation value is changed on every Automatic Exit. As a result, this helps protect values in the general-purpose registers of an SEV-SNP guest from side channel attacks based on ciphertext visibility. The rest of this document therefore focuses on protecting values stored in guest memory only.

CIPHERTEXT VISIBILITY CONCERNS

As AES is a cryptographically secure encryption algorithm, an adversary with access to ciphertext, and potentially knowledge of corresponding plaintext, cannot determine the encryption key used. Even without knowledge of the encryption key though, a malicious entity that monitors a single 16B block of memory over time may be able to infer information about guest operations based on if and when that block of data changes. For example, an adversary might notice that between two points in time, memory location A and B have been modified by the guest but memory location C has not. Information like this might enable the attacker to help guess what software the guest is running, in what is known as an inference attack.

Whether or not the ciphertext at a memory location changes can also potentially convey information. In the above example, if the attacker knows that memory location C was written but the ciphertext does not change, they can infer that the value stored at this location has not changed.

Another possibility is that the adversary might be able to gain information over time about what values certain ciphertext corresponds to. For instance, if the attacker knows that at a certain time the guest VM stores the value 0 to a memory location, they can read the ciphertext at that time and remember it. At a later time, if they observe that same memory location contains the same ciphertext, they can infer that this location again has stored the value 0.

The ability for a malicious entity to conduct a ciphertext-based attack and extract meaningful information is highly dependent on the guest software being executed. For example, software which writes memory location A when a secret value is 0, but memory location B when a secret value is 1 would be particularly susceptible to a ciphertext-based attack.

Historically, software which deals with highly sensitive data, such as cryptographic keys, have taken extra precautions against side channel attacks that can arise due to shared caches, branch predictors, etc. Ciphertext visibility by an adversary is another potential side channel which may merit consideration when writing sensitive software designed to execute inside an AMD SEV guest.

RECOMMENDED TECHNIQUES

The techniques mentioned here are presented as guidance, applicability to specific situations may be situation dependent. In all cases, the goal is for software to store a sensitive value to memory while limiting the ability for an outside entity to be able to either determine the plaintext value or in some cases, if the value was changed at all.

DATA PADDING

Data padding may be used when the sensitive data being stored in memory is less than 16B in size. In this scenario, an extra value may be stored next to the sensitive data so that the encrypted value of the entire 16B block changes in an unpredictable manner.

One way to do this is to pad data with a random value, such as from the CPU RDRAND instruction. The RDRAND instruction is available on AMD CPUs that support any flavor of SEV technology and the instruction returns a random value directly from the hardware random number generator. For instance, an 8B data value could be padded with an 8B random value. The chance of two 8B random data values repeating over time is extremely low and therefore the sensitive data stored in the lower 8B will be less susceptible to a side channel attack.

In some cases, padding could be a simple counter as shown in Figure 1. Each time the memory location is written, the counter value is incremented. In this way, the ciphertext changes on each write and will never repeat as long as the counter value never repeats. To ensure the padding values never repeat even if memory is re-used, software should obtain the initial counter value (IV) randomly.

If the data being stored is too large to be padded directly, it can be broken up until smaller pieces and "striped" across memory. For example, instead of storing a 32B value in two successive 16B locations, it could be stored in 8B chunks in four successive 16B locations with data padding at each location.

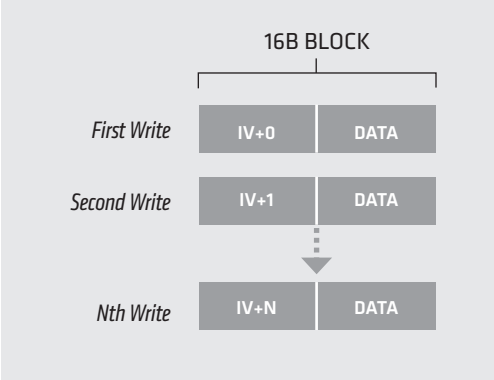

FIGURE 1: DATA PADDING WITH A COUNTER

DATA MASKING

While data padding may work in some cases, it requires extra memory usage to store the padding values. Using data masking, less additional memory may be required. This technique involves generating a random, or pseudo-random value and performing a reversible tweak on data values before they are stored in memory. For example, before storing a 128B block of data, software can generate a random 16B value and XOR each block of the data with the random value before storing it in memory. The random 16B value can itself be stored somewhere else in memory.

This technique permutes the data stored in memory in an unpredictable manner, causing ciphertext changes on every write. It works best if a large block of data is being stored at once and will be accessed later again as a large block. Each time the data block is written, the tweak value should be regenerated.

DATA ALIGNMENT

Data structures which are mis-aligned across 16B boundaries and contain secret information may be more sensitive to ciphertext visibility concerns. For example, an 8B secret which is stored from address 0x100F-0x101E spans two 16B blocks. While there are 2^{64} possible values of this secret, there are only 256 possible values for the first byte of the secret. If the data in addresses 0x1000-0x100E remains static, it may be easier for the attacker to infer the first byte of the secret since there are only 256 possible values of the ciphertext of this 16B block. If the secret data was aligned and placed at address 0x1010-0x101F then it would be much more difficult to infer any information about the secret since any change in a single byte of the secret would result in the entire 16B ciphertext changing.

Software typically aligns data structures on natural boundaries for performance reasons, but this alignment is also important to limit ciphertext visibility concerns. Note that while data alignment helps provide the maximum possible entropy for each ciphertext block, software may wish to combine this technique with one of the other techniques in this section to help ensure the ciphertext changes on every data write.

DATA MOVEMENT

Another technique that may be useful in specific cases involves moving data values in memory when they may or may not change. This is particularly useful when the fact that a value is being changed (or not changed) conveys sensitive information. While other techniques including padding or masking can also help mitigate ciphertext visibility in this case, it may be simpler to copy the data values to a different memory location after performing the appropriate operation. The ciphertext of those values at a new location will be different than the ciphertext at their previous location, regardless of any actual change in the values themselves.

When this technique is used, it is recommended that software move the data values to memory locations with different page offsets (bits 11:0) than their previous location. This ensures that a different encryption tweak value will be used, regardless of how guest memory is mapped into the physical address space.

EXAMPLE: MONTGOMERY LADDER

An example of a constant-time Montgomery Ladder for ECDSA is shown in Figure 2. This uses a CSWAP (conditional swap) operation which swaps the two inputs based on a third input. Even if CSWAP is implemented in a constant-time manner, as written, this code may be vulnerable to ciphertext side channels. As the secret key (k) is used to conditionally swap the value A and B, an attacker who can observe whether the memory locations corresponding to A and B change in each loop iteration may be able to infer the value of the secret.

To help mitigate potential ciphertext side channels, the data movement technique can be used. The CSWAP operation should write the result into a different memory location, rather than operating in-place. As the values A and B change in each iteration of the loop, an attacker would not be able to observe whether the values were swapped or not. An example of this form of the Montgomery Ladder is shown in Figure 3. In this code, the result of each CSWAP operation is written to a different memory location. This code could be further optimized, but this demonstrates how a Montgomery Ladder can be written while hiding the secret used in each CSWAP operation.

```
for (i=(n-1) down to 0)
      CSWAP(A, B, k_i)
      B = ADD(A, B)
      A = DOUBLE(A)
      CSWAP(A, B, k_i)
end for
```

FIGURE 2: BASIC MONTGOMERY LADDER

```
for (i=((n/2) – 1) down to 0)
      {C, D} = CSWAP(A, B, k_2i+1)
      D = ADD(C, D)
      C = DOUBLE(C)
      {A, B} = CSWAP(C, D, k_2i+1)
      {E, F} = CSWAP(A, B, k_2i)
      F = ADD(E, F)
      E = DOUBLE(E)
      {A, B} = CSWAP(E, F, k_2i)
end for
```

FIGURE 3: SAFER MONTGOMERY LADDER

CONCLUSION

Current versions of AMD SEV technologies offer many protections to guest VMs that wish to run in confidential environments under a potentially malicious hypervisor and subject to various forms of physical attacks. While AMD continues to invest in new security technologies and hardware-based enhancements for future designs, this paper describes techniques and considerations for software developers that wish to maximize the level of protection that can be achieved in AMD SEV environments.

REFERENCES:

[1] AMD, "Protecting Register State with SEV-ES," 2017. [Online]. Available: https://www.amd.com/system/files/TechDocs/Protecting%20VM%20Register%20State%20with%20SEV-ES.pdf.

[2] AMD, "SEV-SNP: Strengthening VM Isolation with Integrity Protection and More," 2019. [Online]. Available: https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[3] AMD, "AMD Random Number Generator," 2017. [Online]. Available: https://www.amd.com/system/files/TechDocs/amd-random-number-generator.pdf.

[4] AMD, "AMD64 Architecture Programmer's Manual, Volume 2: System Programming," November 2021. [Online]. Available: https://www.amd.com/system/files/TechDocs/24593.pdf.

REVISION 5.10.22

AMD.com/productsecurity

AMD