Securing AMD SEV

Phillip Mestas
Oregon State University

Abstract

AMD SEV allows for the creation of fully encrypted virtual machines. This allows cloud computing tenants' data to be secret to the cloud computing provider. However, it has been shown that the encryption scheme used by AMD can easily be broken . The attacker can create a copy of the virtual machine, and perform some malicious operations to gain a secret value used in the encryption scheme. They can then use this value to write and read encrypted data to and from the target virtual machine. To prevent this, we propose wrapping the insecure encryption scheme with a stronger encryption scheme that is proven to be resilient to this kind of attack. In addition, we discuss other trusted execution environment vulnerabilities that have been discovered, as well as how they relate to AMD SEV and our proposed defense.

1 The Attack and Proposed Defense

1.1 Overview

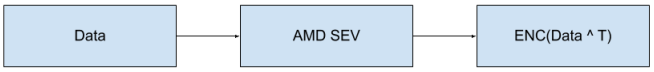

Figure 1: When data is written to a virtual machine's memory, it is xored with the memory pages tweak value, and then encrypted with AES-128 encryption.

Secure Encrypted Virtualization (SEV) is AMD's implementation of a trusted execution environment [1]. It is capable of encrypting an entire virtual machine, protecting it's execution and data from a malicious hypervisor. Whenever data is written to memory, it is xored with the tweak value for the page that it is being stored in, and the result is encrypted with AES-128 encryption using a key that is private to the virtual machine. This operation is represented in figure 1. When the data is read from memory, this operation is reversed to get the original data. This scheme encrypts all of the data that is stored in memory, preventing the hypervisor from writing and reading information to and from the program's memory space. However, the encryption scheme used by AMD has been shown to be insecure [6]. Using the following method, the hypervisor is able to defeat the defense and both read and insert arbitrary data.

1.2 Reading From and Writing To Memory

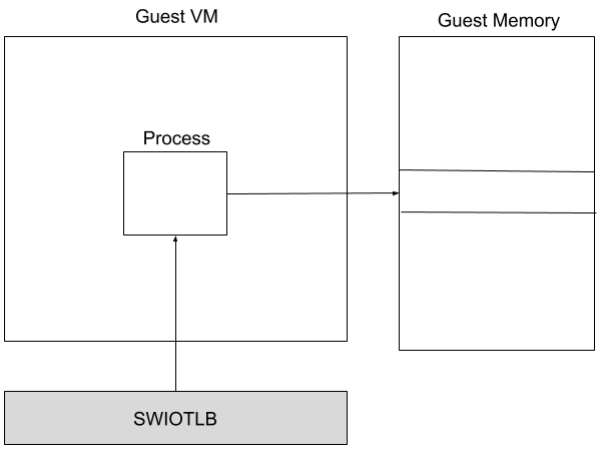

Figure 2: Data goes from the shared SWIOTLB to the private memory space of the process inside the virtual machine, and then into that virtual machine's private memory.

In order to perform this attack, the hypervisor must be able to read and write raw data, as well as encrypt and decrypt some data in the program's memory space. Reading and writing raw data is easy. The hypervisor is capable of altering the program's memory space at any time. AMD's implementation provides the hypervisor with decryption and encryption oracles as well and that allow this attack to be possible. Direct memory access operations are unable to write and read data directly to and from the virtual machine's memory. The data

must pass through a memory space shared between the hypervisor and the virtual machine shown in figure 2 called the SWIOTLB. Information will be encrypted as it is transferred from SWIOTLB to a buffer in the program memory, and decrypted as it is transferred from that buffer to the SWIOTLB. To encrypt aribitrary data to the buffer, the hypervisor can stop the program before a write and change the value in the SWIOTLB to be the data to be encrypted. Similarly, the hypervisor can overwrite the data in the buffer before a read to decrypt arbitrary data from the buffer.

1.3 Stealing the Tweak Value

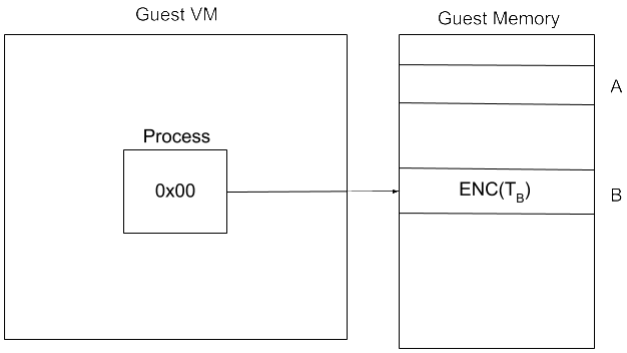

Figure 3: When the user stores 0 in the virtual machine's memory, the result of the operation performed by AMD SEV will be the buffer pages tweak value encrypted.

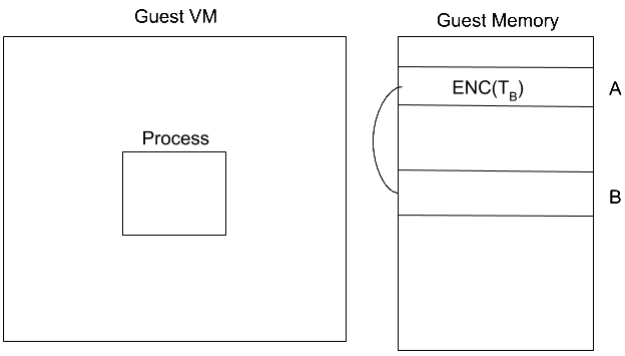

Figure 4: The hypervisor can copy the raw data in any memory address to another. To find the tweak value, the attacker should use this ability to copy the inserted data to the target memory page.

If the tweak value was the same across the entire virtual machine, the attack would be easy. To read some data stored in by the virtual machine, the hypervisor would just need to move the encrypted data to the buffer before a read, and read the data from the SWIOTLB after the read. However, each

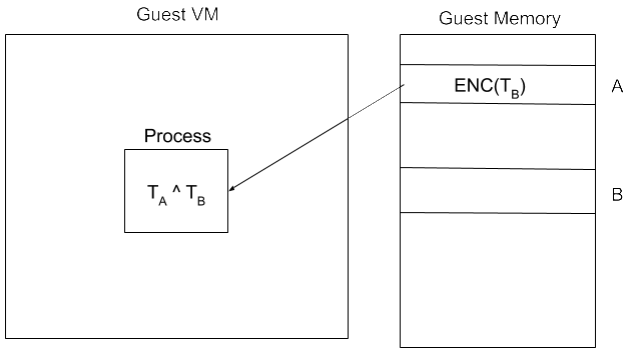

Figure 5: Now that the malicious data is stored in the target page, reading the target page will give the tweak value of the buffer page xored with the tweak value of the target page.

memory page has a unique tweak value, so it is not that simple. The attacker needs to be able to tweak data with the correct value to properly store it in encrypted memory. Likewise, they must use the correct tweak value to decrypt data stored by the program. Luckily for the attacker, the tweak values will be the same for each memory page on a different virtual machine with the same configuration. Since the attacker has control of the hypervisor, they can spin up their own virtual machine with the same configuration. The tweak values can then be determined with the following procedure. First, the hypervisor has an application write all 0s to the memory page that contains the buffer. 0 will be xored with the tweak value, resulting in just the tweak value for the buffer page being encrypted, as shown in figure 3. This value will then be copied over to the target memory page by the hypervisor, represented in figure 4. The application will then read the data from the target page. When the data is read, it will undo the encryption, leaving the tweak value of the buffer page, and xor it with the tweak value for the target page. The result is the tweak value for the buffer page xored with the tweak value for the target page, as displayed in figure 5. This powerful value can be used by the attacker to write and read arbitrary data to and from the original virtual machine.

1.4 Reading Encrypted Data

Once the attacker knows the result of xoring the tweak value of the buffer page and the tweak value of the target page, they are capable of decrypting the encrypted data in the target page. First, they move the encrypted target value to the buffer address before a read. When the read happens, the data will be decrypted, leaving the desired plaintext xored with the target pages tweak value. This value will be xored with the buffer pages tweak value, and the result will be the plaintext xored with the target pages tweak value and the buffer pages tweak value. This is represented in figure 6. The attacker can grab that value from the SWIOTLB, and xor it with the value

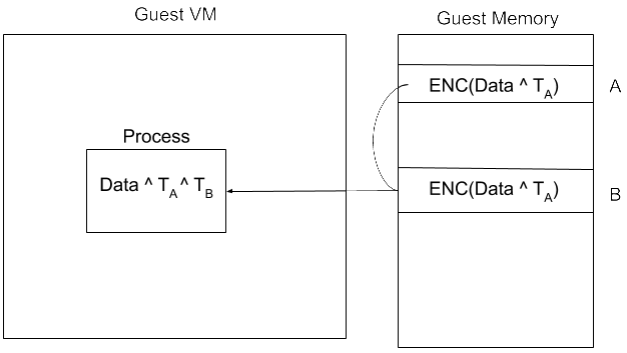

Figure 6: By copying the encrypted secret data to the buffer, the attacker can get the value of the plaintext xored with the tweak values of both the buffer page and the target page.

that was determined in the previous step. Because xoring a value by the same value again gives the original value, this operation will result in the decrypted plaintext! The attacker can apply this procedure to any memory page to read any data they desire.

1.5 Inserting Encrypted Data

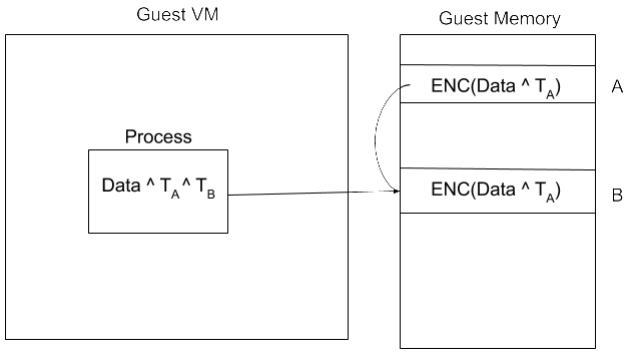

Figure 7: When the attacker xores their malicious data with the tweak value of the target page and the tweak value of the buffer page and then inserts that value into the buffer page, the result will be the encryption of their original data that is appropriate for the target page.

The attacker can use a similar procedure to write arbitrary data to encrypted memory. First, they will xor their plaintext data with the value determined previously. They will then put this data into the SWIOTLB before a write. When the write happens, the data will be xored with the tweak value of the buffer page, encrypted, and stored in the buffer. The result of this is the encryption of the plaintext data xored with the tweak value of the target page as shown in figure 7. This is the correct way to store the encrypted plaintext in the target page! The hypervisor can then copy the raw data in the buffer to the target page. The next time the victim reads this page, it will be decrypted into the hypervisor's data, so the victim will read whatever value the hypervisor desires.

1.6 Defending the Attack

The cause of the attack is that the attacker has an encryption oracle and a decryption oracle, and the encryption scheme used is not strong enough to defend against an attacker with these tools. We cannot remove the oracles or strengthen the encryption scheme without making significant changes to the AMD CPUs that support SEV. Our proposed defense to this attack is to wrap the weak encryption scheme with a stronger one that cannot be defeated with the tools available to the attacker. The current encryption scheme is an XE encryption scheme, which has been proven to be insecure with the presence of both an encryption oracle and a decryption oracle. We will wrap it with an XEX encryption scheme, which is shown to be secure in this scenario. This change will be implemented in the linux kernel. Whenever a write occurs from the SWIOTLB, we will encrypt the data with the secure encryption scheme, and send that value to be encrypted with the insecure encryption scheme. When data is read into the SWIOTLB, we will reverse the operation to get the original value back before writing it to the SWIOTLB. One challenge that we will face is encrypting network traffic. Because the encryption needs to happen in order, we will have to store any network traffic that arrives out of order and wait until the next in order data arrives so that this data can be encrypted first. When data is being written from the disk, we must take into account that there may be simultaneous writes from different parts of the disk. We will separate each of these writes, so that the encrypted data is not mixed together. Our methods for handling these challenges may not be the most efficient way, but our goal is just to prove that the idea works to defend from the attack. We will leave developing efficient methods for future engineering work.

2 Related Work

2.1 RAM Based Exploits

Intel Software Guard eXtension (SGX) is a trusted execution environment available on Intel CPUs that stores sensitive code and data in an isolated environment known as an enclave. It does not encrypt an entire virtual machine like AMD SEV, but does provide similar security features for the data that is protected by the enclave. Intel SGX provides a mechanism for ensuring that the data within the enclave is not physically interfered with. If a change is detected between reads that was not caused from within the enclave, the entire system will shut down. This security feature is effective for stopping physical attackers, but can cause problems if the data can be

changed without physical access, or using the enclave. The rowhammer attack allows malicious users to flip arbitrary bits in physical memory from their own memory space by alternating between memory accesses to different rows in the same vulnerable DRAM bank [3]. If the attacker is able to execute code on the same physical machine, and the machine uses vulnerable DRAM, the attacker can launch this attack to change some of the memory protected by the enclave. When Intel SGX detects this change, it will shut down the entire physical machine [2]. Although Intel SGX is vulnerable to this denial of service attack, AMD SEV is not impacted. The lack of integrity checking prevents such attacks from being effective.

2.2 Timing Side Channel Attacks

Side channel attacks are a powerful class of attacks that use outside information to infer something about the programs execution. Timing based side channel attacks have been shown to be very powerful in certain contexts. They involve measuring the time it takes for a processor to accomplish some task, and using that information to infer some secret information. The flush and reload attack is performed by flushing the L3 cache of a CPU [10]. Then attacker then waits for the victim to perform some operation. After, the attacker will attempt to reload all of flushed cache blocks. Any blocks with a slow access time were not accessed by the victim, while any that are read quickly were accessed by the victim. This attack can be used in combination with flaws within the CPU microarchitecture to yield some powerful results. For example, the meltdown vulnerability allowed the entire kernel space to be read from user space on Intel CPUs [7]. Sceptre is a similar attack to meltdown. It relies on speculative execution and timing based side channel attacks to read secret data from another user's program space [4]. The attack requires a vulnerability in the victim's code that is known to the attacker, but could theoretically be used across the boundary of AMD SEV if the victim and the attacker are sharing the same L3 cache. Our defense does nothing to mitigate these attacks, and AMD SEV users should be aware of their existence.

2.3 Memory Access Side Channel Attacks

Another useful side channel is memory access [9]. If the attacker has control of the operating system, they can alter the page table to trigger page faults when specific pages are accessed by the victim. When the victim attempts to access one of the target pages, the operating system will be alerted with information about the page that the victim tried to access. If the victim has the source code of the program that the attacker is running, they can analyze the branches in the code and identify areas where they can infer some data in the program by knowing which branch was taken. One proposed defense to this vulnerability is program obfuscation [8]. This technique involves creating confusing memory access patterns that do not leak the program's data. However, this approach is not practical yet because the current state of the art implementations still introduce a large amount of overhead. Developers working with AMD SME should be aware of these attacks, as their programs may be vulnerable. Those working with AMD SEV should not need to worry about this side channel, as the operating system is encrypted and protected.

2.4 Other

Memory corruption vulnerabilities are common in any non-trivial program. Numerous techniques have been developed to exploit them to achieve arbitrary code execution with the victim program's permissions. Dark ROP is a set of tools to exploit these vulnerabilities in code running inside of an Intel SGX enclave [5]. Even though the code is encrypted, the attack can still be launched. Programs running under AMD SEV will likely be vulnerable to similar strategies. Developers must be aware of these attacks, and avoid implementations that allow for them to happen. In addition, more work should be done to mitigate this class of attacks.

References

[1] Jeremy Powell David Kaplan and Tom Woller. Amd memory encryption. 2016.

[2] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. Sgx-bomb: Locking down the processor via rowhammer attack. In *Proceedings of the 2Nd Workshop on System Software for Trusted Execution*, SysTEX'17, pages 5:1–5:6, New York, NY, USA, 2017. ACM.

[3] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *SIGARCH Comput. Archit. News*, 42(3):361–372, June 2014.

[4] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *CoRR*, abs/1801.01203, 2018.

[5] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 523–539, Vancouver, BC, August 2017. USENIX Association.

[6] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting unprotected i/o operations in amd's secure encrypted virtualization. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1257–1272, Santa Clara, CA, August 2019. USENIX Association.

[7] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, August 2018. USENIX Association.

[8] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 431–446, Washington, D.C., August 2015. USENIX Association.

[9] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 640–656, Washington, DC, USA, 2015. IEEE Computer Society.

[10] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 719–732, Berkeley, CA, USA, 2014. USENIX Association.