# AMD Secure Processor for Confidential Computing

# Security Review

Technical report by Google Project Zero & Google Cloud Security

May 2022

Cfir Cohen, James Forshaw, Jann Horn, Mark Brand

# Overview

Confidential computing (CC) protects data in use by performing computation in a hardware based trusted execution environment. These isolated environments help prevent unauthorized access or modification of applications and data while in use, thereby increasing the security assurances for organizations that manage sensitive and regulated data in hosted cloud environments.

Google Project Zero, Google Cloud Security teams, and the AMD[1] firmware team partnered to conduct a thorough security review of AMD confidential computing technology. The goal was to audit the Secure Encrypted Virtualization (SEV) firmware, improve the secure processor's (ASP) security posture, and thus further build trust in confidential computing technologies.

The review focused on the implementation of the ASP in the AMD "Milan" platform, which includes new CC features. The review spanned multiple months, covered several ASP components, and evaluated different attack vectors. We manually reviewed the design and source code implementation, wrote custom fuzzers, and ran hardware security tests. In the process, we identified security issues of varying severity, and successfully compromised CC security features. AMD was diligent in fixing all applicable issues, and now offers fixed firmware through its OEM channels. Google's AMD-based confidential compute solutions include all the fixes arising from this review.

This report, aimed at infosec practitioners and firmware authors, details our research methodologies and tools, lists interesting bug findings and exploitation techniques that are unique to cryptography heavy applications.

[1] *AMD, the AMD Arrow logo and combinations thereof are trademarks of Advanced Micro Devices, Inc.*

# Background

## Confidential computing

Confidential computing workloads run in trusted execution environments (TEE). Security is enforced by hardware, and achieved through 1) launch time attestation, and 2) run-time memory protection. Launch time attestation is a report, signed by a key rooted in hardware, which captures all code and data pages that were loaded into the TEE. This lets the workload owner remotely verify the starting configuration for the TEE. Memory protection protects workload's confidentiality and integrity. It provides run-time security protection: no workload data enters or leaves the TEE without explicit permission. This gives the owner full control over their data.

A common workflow for CC: the owner launches a non-secret loader into a TEE, remotely verifies the launch attestation, checks the intended code was loaded on a genuine TEE hardware with a given policy, then, over a secure communication channel, the owner provides secrets to the workload.

Confidential virtual machines (VMs) is the computational model supported by AMD SEV, where an entire guest VM runs in a trusted execution environment. This model provides an easy way to run native x86 applications in a confidential environment, provided that the guest VM is running an operating system designed for this use case.

## AMD Secure Processor (ASP)

The AMD secure processor (ASP), also known as platform security processor (PSP), is an isolated ARM processor that runs independently from the main x86 cores of the platform. ASP executes its own firmware, and hosts security sensitive components that can run without being affected by the main system workload.

ASP's key features are 1) platform secure boot, and 2) SEV support. ASP authenticates the initial BIOS boot code prior to starting BIOS boot. ASP supports SEV by managing guests' life cycle, generating and managing the inline memory encryption keys.

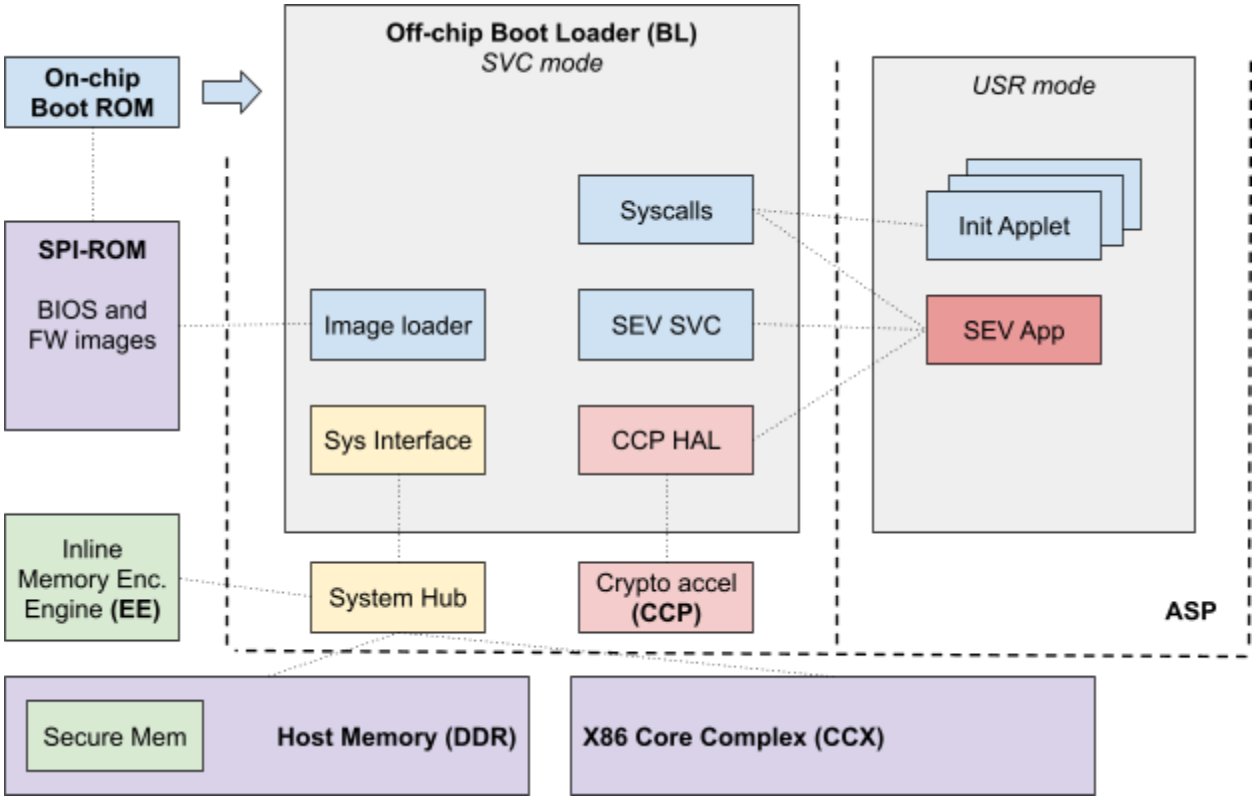


*Figure 1: ASP Architecture (Simplified)*

Follow figure 1 from left to right, top to bottom. ASP boots an immutable on-chip boot ROM, which authenticates and loads a larger off-chip bootloader image. The bootloader (BL), which runs in SVC mode, validates the x86 BIOS boot code, and releases the x86 cores for execution. The BL offers run-time services to ASP's USR mode applications. These platform specific firmware applets are responsible for silicon initialization. In its steady state, the BL waits for new SEV commands, and handles them in a separate SEV application.

The ASP is a privileged platform security component, and through its system hub interface, it has direct access to host DRAM memory, including a secure 8MiB region dedicated to ASP operations. In addition, ASP has access to the memory encryption engine, and is responsible for managing the memory encryption keys. ASP has access to the x86 core complex which is used for querying x86 state (for instance, ASP verifies *WBINVD* was executed on all active cores after decommissioning an SEV guest), and releasing x86 cores after a successful secure boot verification. Finally, the ASP has a built-in cryptographic coprocessor (CCP) used for protecting key material and accelerating cryptographic operations.

ASP's main attack vectors are: 1) parsing malformed images read from SPI-ROM, 2) handling BIOS commands read from BIOS-SP mailbox registers, 3) handling SEV commands read from SEV mailbox registers. The last further expands the attack surface, since SEV firmware processes additional data coming from untrusted SPI-ROM and host physical memory.

Our review mostly focused on the SEV firmware and CCP hal library, however, we did find and fix several issues in the BL's image processing module.

## AMD SEV

Secure Encrypted Virtualization enables running encrypted virtual machines (VMs) in which the code and data of the virtual machine are secured so that the decrypted version is available only within the VM itself.

Many hardware components make SEV work. CPU encodes the VM address space identifier (ASID) on memory accesses. The memory controller (MC) uses the inline AES encryption engine (EE), indexes the keys array using the ASID, encrypts and decrypts data written to or read from physical memory. The ASP generates random VM encryption keys using its CCP. It loads the keys into the encryption engine in a step called *guest activation*. It unloads the keys in a step called *guest deactivation*, and verifies the CPU cache has been flushed.

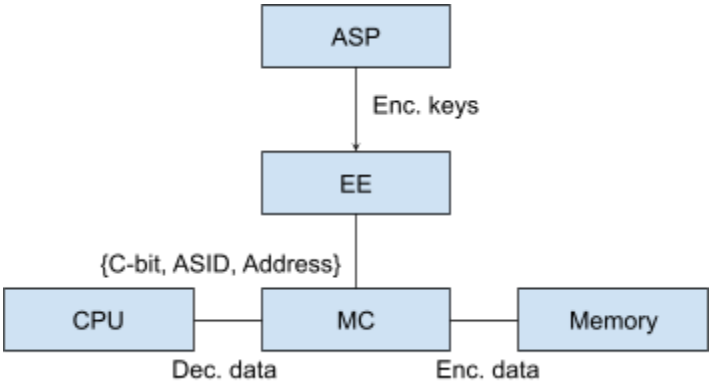


*Figure 2: Inline memory encryption*

SEV has three successive generations with improving security properties for each generation.

| **Generation** | **Security properties** | **Security assumptions** |
|---|---|---|
| Secure Encrypted Virtualization (SEV) | Memory confidentiality. | Hypervisor and host system are trusted. |
| Secure Encrypted Virtualization - Encrypted State (SEV-ES) | Memory + registers confidentiality. | Hypervisor and host system are trusted. |
| Secure Encrypted Virtualization - Secure | Memory + registers confidentiality + integrity. | Hypervisor and host system are NOT trusted. |

| | | |
|---|---|---|
| Nested Paging (SEV-SNP) | | Only AMD hardware and AMD-signed firmware are trusted.<br>Memory DIMMs are implicitly trusted. |

SEV-SNP integrity protection is implemented using an access check on memory writes.

SEV-SNP has a hypervisor + firmware managed data structure called a Reverse Map Table (RMP). The RMP tracks the state for each physical page: its owner, guest physical address and other metadata bits.

X86 uCode checks the RMP on memory write accesses, and throws an exception when an invalid access occurs, for instance, when hypervisor mode attempts to write to private guest memory. In addition, there's a process called page validation where the guest is responsible for accepting newly assigned pages.
Together, the RMP protects against data corruption / replay (only the owning context can write to a page), and memory aliasing (one page can only be mapped to one guest at a time) attacks.

Our review focused on SEV-SNP, with the assumption that the attacker is quite powerful. A malicious hypervisor can invoke SEV FW APIs with arbitrary data, and can attempt to modify / drop / replay firmware messages or guest memory pages. It can also mount software-based side channel attacks against the ASP. Any firmware that is not verified and attested as part of the boot and attestation process are untrusted, so we assume that an attacker could send rogue DMAs directly from an attached PCIe device. Note however that physical attacks are out of scope for the SNP security model, so an attacker with the access and technological capability to perform live signal modification between memory and the CPU, or to directly modify the CPU die might be able to successfully compromise SNP security features. Furthermore, architectural limitations such as the ability to repeatedly read deterministically-encrypted memory, were outside the scope of our review.

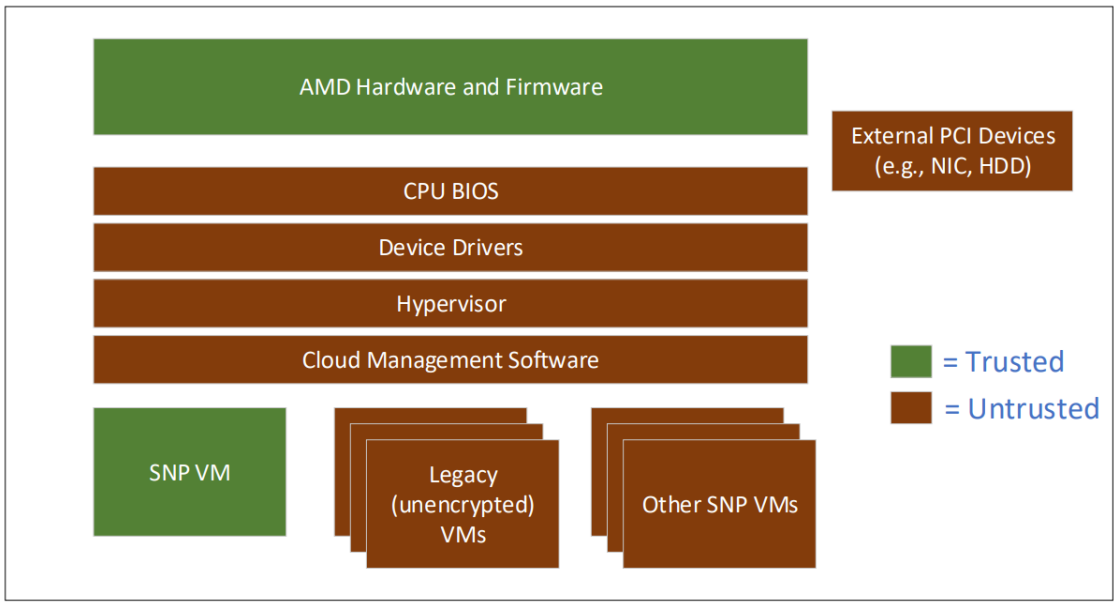


*Figure 3: SEV-SNP threat model*
*(Diagram from AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More)*

Note, guest availability cannot be guaranteed. The reason: the hypervisor is responsible for scheduling the VM for execution. In addition, the hypervisor manages the VM's life cycle and memory. It facilitates communication between the guest owner and the ASP by passing encrypted blobs between the two. In other words, the hypervisor is a necessary component, yet not a trusted component.

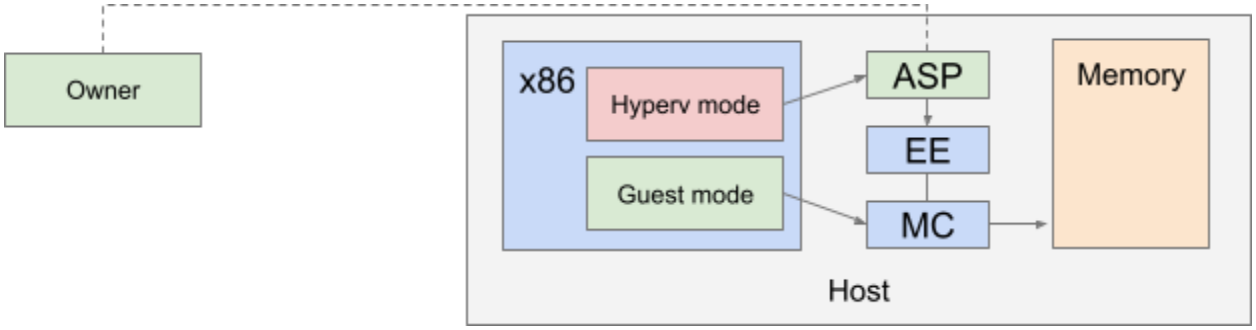


*Figure 4: Logically, VM owner and ASP form a secure channel,*
*Untrusted hypervisor facilitates communication, manages virtualization resources*

For additional information on SEV, see the AMD developers' landing page. The firmware API specifications linked on that page document the API available to the host hypervisor for management of SEV guests. The AMD64 Architecture Programmer's Manual (APM) Volume 2 lists SEV architecture details and low level structures. This KVM forum talk presents SEV-SNP, and this whitepaper describes its threat model. This whitepaper by the Confidential Computing Consortium lists the threat vectors for TEEs.

# SEV findings

The SNP firmware in “Milan” supports all SEV modes, and includes APIs for launching SEV and SEV-ES VMs. “Milan” supports running SEV VMs next to SNP VMs.

In our opinion, this represents a significant attack surface in the firmware, and received proper audit. This section presents two findings, and sets the stage for further discussion on SNP security. Note, the bugs in SEV assume a malicious hypervisor, even though it’s not strictly assumed in its threat model. We include these here, since the research methodology is interesting, and some bugs could later be leveraged to compromise SNP security or private firmware data.

## Deactivated guest reveals launch secrets

This bug demonstrates a common attack pattern in secure systems: finding an exception to system invariants. The SEV firmware maintains a set of invariants that are critical for system security, for example:

| | |
|---|---|
| **Invariant (A)** | It is impossible to assign an active ASID to two different guests. |
| **Reason** | An ASID maps to an encryption key in the EE, so a shared ASID between two guests, means shared encryption key, which means shared access to encrypted data. This is insecure since guests are mutually distrusting (different identities, different policies). |
| **Implementation** | The firmware tracks ASID state {clean, allocated, in_use, dirty} through the different API calls, and prevents activation of an owned / dirty ASID. |
| **Invariant (B)** | It is impossible to reassign an ASID (so ASID -> key X is replaced by ASID -> key Y mapping) when plaintext data still resides in CPU caches. |
| **Reason** | Failure to maintain this could leak confidential data. Data in CPU caches is tagged with the ASID. A malicious hypervisor reassigns the victim VM’s ASID to an attacker VM, then, inside the attacker’s VM, writes-back and invalidates the caches (*WBINVD*), then reads back the data from memory. The last two steps happen with the attacker’s encryption key, so they effectively leak the victim’s leftover cache data. |
| **Implementation** | The firmware tracks ASID state through the different API calls. The transition from *dirty -> clean* happens in the *DF_FLUSH* command handler, after the firmware verifies the *WBINVD* had been executed on |

| | |
|---|---|
| | the x86 cores, then it flushes the data fabric which deletes all cached data. In addition, an ASID cannot be activated unless it is in the *clean* state, see invariant (A) above. |
| **Invariant (C)** | A guest must be activated (its encrypted key loaded in the EE) when the firmware reads / writes private guest memory on behalf of the guest. |
| **Reason** | An inactive guest has an ASID = 0. This ASID is reserved for the host. The encryption key at index 0 is used for Secure Memory Encryption (SME) where host memory pages can be encrypted.<br>A firmware that writes to private guest memory using ASID 0, inadvertently writes cleartext data to physical memory. |
| **Implementation** | Firmware commands that map and access guest memory on behalf of the guest (launch update, dbg crypt, etc) check that the guest is activated. |

We identified an exception to invariant (C): *launch_secret* command handler failed to check the guest is activated. Exploitation is straightforward: a malicious hypervisor tricks the firmware into writing the launch secret in plaintext form, using ASID=0 reserved for the host, simply by deactivating the guest just before the call.

## HMAC OOB read leaks private data

The device has limited SRAM, and SEV firmware does not use a heap - there's no dynamic memory allocations. Instead, processing is done on either local stack buffers or global, fixed size buffers. A common buffer is a 20KB scratch buffer used for storing intermediate values.

For example, *launch_secret* command handler uses this buffer to store the encrypted launch secret. It decrypts the secret in place using the transport encryption key (TEK), then copies it back to guest memory, this time encrypted under the VM encryption key (VEK).

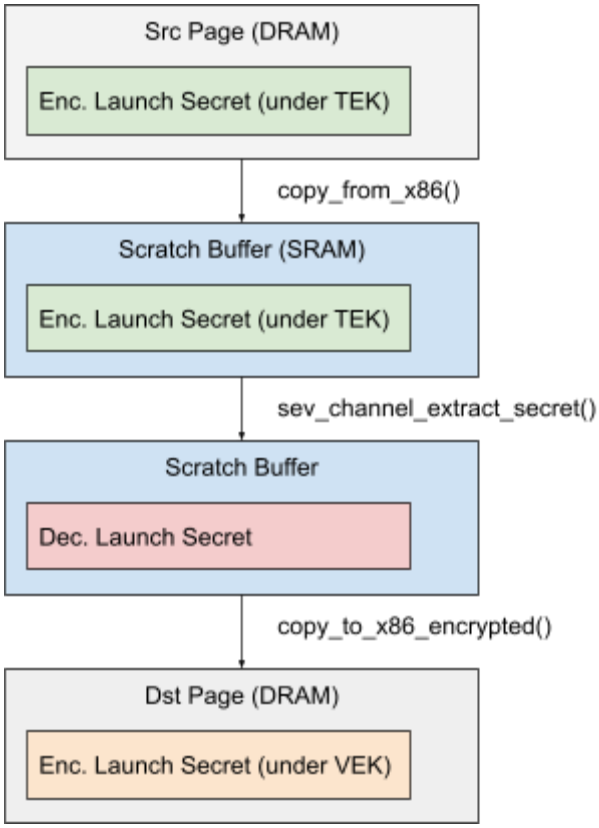


*Figure 5: Scratch buffer holds intermediate values*

An important detail, the firmware does not clear the buffer between command handling.

Similarly, *send_update* command handler decrypts guest memory into the scratch buffer, then re-encrypts it under the transport key, and copies it back to host memory.

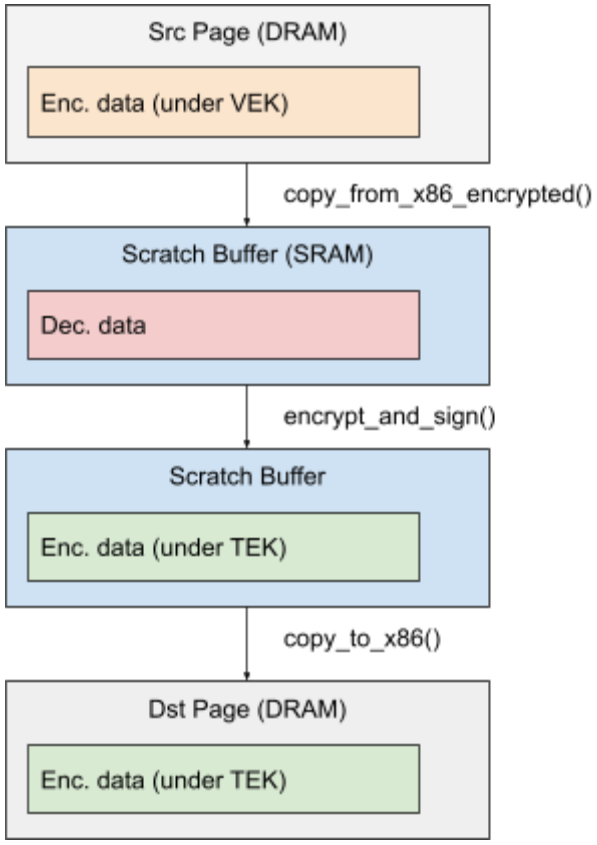


*Figure 6: send_update encrypts and macs guest data*

The authenticated encryption scheme is AES CTR + HMAC. The encryption key (TEK) and integrity key (TIK) are derived from a master session key.

We identified a subtle bug in *send_update* where the input size for encryption (*guest_len*) could be less than the input size for signing (*trans_len*). This leads to an OOB read vulnerability, where *send_update* includes leftover scratch buffer bytes in the MAC (figure 7).

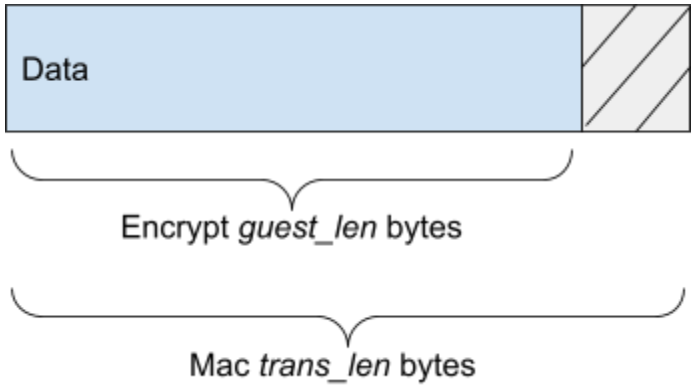


*Figure 7: send_update OOB read*

We exploited this bug to read the victim VM's private memory. At a high level, we placed the victim's private data in the scratch buffer, used the OOB read to leak a secret byte into the signature computation, and, in an offline step, brute forced all possible bytes until the correct HMAC tag was found. We used a "sliding window" that read the private data at a new offset, then repeated the attack (OOB read + HMAC bruteforce) until the entire secret was revealed.

These attacks are common in crypto applications - a leak is returned indirectly, through a new session key or a new signature blob.

There are technical challenges with the exploit above:

1. *guest_len* and *trans_len* must be 16B aligned, so we must pass at least trans_len = guest_len + 16, and read 16 bytes past the buffer.
2. Victim's data is copied to the scratch buffer using *send_update*, however, the function re-encrypts the data before it exists, see figure 6. In order to keep the secret in cleartext form we need a fail *send_update* early.
   We used *send_update* to copy guest pages instead of *launch_secret*, because the former lets us copy guest data at different offsets, and paves the way to a "sliding window" technique.

The final attack placed a secret byte next to 15 known bytes - this way we still only need to test 256 candidates, even when we read 16B past the data. We found a way to fail *send_update* early by passing an invalid page property (invalid VMSA page).

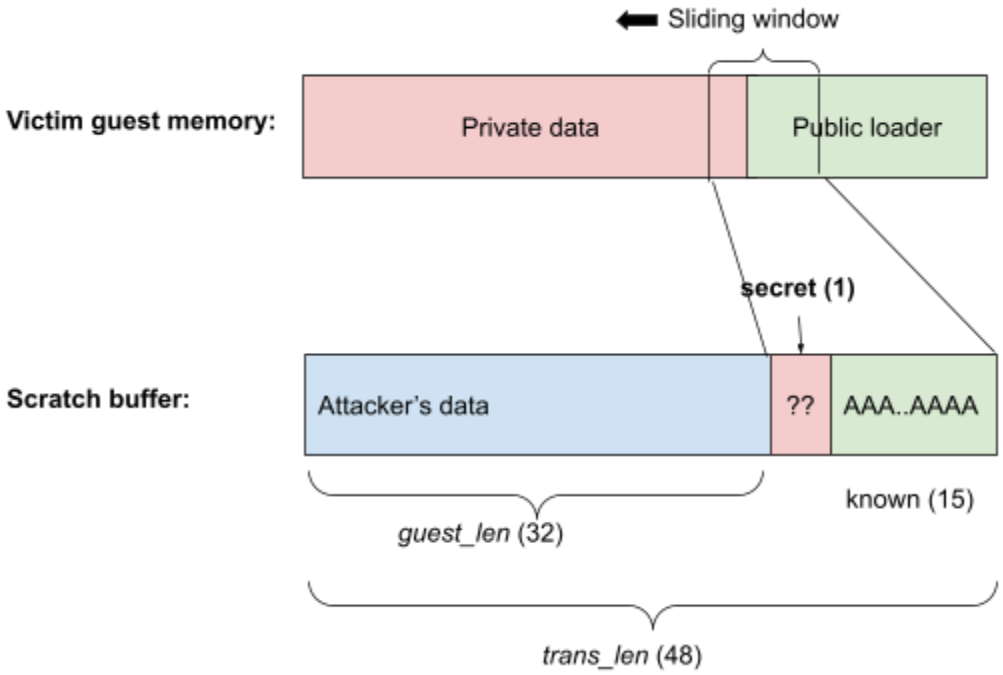


*Figure 8: Sliding window attack*

Note, after the first secret byte is recovered, we slide the window left by one, and keep the invariant “unknown secret next to 15 known bytes”. We repeat the process until the entire victim’s guest memory is recovered.

We were able to build another exploit for this bug, and leaked (parts of) the SEV identity keys - the PEK’s private ECDSA scalar. This demonstrates the impact of a simple OOB read when secret information is stored in a global scratch buffer without proper cleanup.

# Crypto review

Reviewing cryptography heavy systems for security requires non-trivial domain knowledge. In cryptography details matter: the encryption mode is sometimes more important than the underlying encryption algorithm, IV selection is just as important as key generation, small search space leads to practical attacks, and there are many other pitfalls.

Cryptographic vulnerabilities are particularly powerful due to their silent nature: no exploit mitigation blocks a compromised authentication key. These vulnerabilities don’t fall prey to fuzzers, and finding these requires dedication and expertise. Nevertheless, we believe there is a systemic way to pursue crypto reviews. In addition, there are tools that aid verification. Below we describe our process, and list some tools and methods that helped our review.

A crypto review focuses on three main layers:

| | |
|---|---|
| **Implementations** | The low-level implementation. |
| | Here we check for correctness, look for random selection bias, secret dependent operations, and so on. |
| | Keep in mind that crypto attacks target the implementation, not the math. |
| | **Tools**: manual code review, official test vectors, edge-case test vectors (see “Wycheproof” below), differential tests (encrypt with one library, decrypt with another), statistical tools (randomness tests, histogram plots). |
| **Algorithms** | Choice of basic building blocks: encryption, digest, signing algorithms and schemes. |
| | Here we look for weak algorithms (think MD5 for hashing, RC4 for encryption), weak modes (ECB), and weak schemes (unauthenticated ciphertext). We audit the key size, key generation, |

| | |
|---|---|
| | and key storage mechanism. |
| | Even strong primitives have **limitations**, for instance, AES-GCM has an upper limit on the number of messages it can encrypt using the same {key, IV}. |
| | **Tools**: established standards such as NIST provide up-to-date recommendations on algorithms, schemes and key sizes. |
| **Protocols** | The protocol layer, or how the building blocks are assembled together to form a secure communication channel. |
| | Here we look for the following security properties: secrecy, authenticity, freshness and strong identity binding. |
| | Indeed, one can build a robust implementation, use strong primitives, yet have a broken protocol. |
| | **Tools**: formal verification tools such as Proverif or Tamarin can aid verification. Interestingly, they can also synthesize attacks as counter examples. |

## Wycheproof tests

Project Wycheproof is a set of security tests that check cryptographic software libraries for known weaknesses. It is maintained by leading cryptographers Daniel Bleichenbacher and Thai Duong of Google's information security engineering (ISE) team. Wycheproof successfully found critical vulnerabilities in a dozen libraries.

Wycheproof test vectors are expansive: they cover a large number of algorithms and schemes, and include both correct and incorrect test cases. Example test vectors for an RSA signature scheme, taken from this json file, look like this:

```
"testGroups" : [
  {
    "e" : "010001",
    "keyAsn" : "3082...01",
    "keyDer" : "3082...01",
    "keyPem" : "-----BEGIN ... KEY-----",
    "keysize" : 2048,
    "mgf" : "MGF1",
    "mgfSha" : "SHA-256",
    "n" : "00a2...d5",
    "sLen" : 32,
    "sha" : "SHA-256",
    "type" : "RsassaPssVerify",
```

```
    "tests" : [
       ...
       {
        "tcId" : 62,
        "comment" : "first byte of m_hash modified",
        "msg" : "313233343030",
        "sig" : "67d1...6d",
        "result" : "invalid",
        "flags" : []
      },
      ...
      {
        "tcId" : 69,
        "comment" : "salt is all 0",
        "msg" : "313233343030",
        "sig" : "1591...bf",
        "result" : "valid",
        "flags" : []
      },
  ]
}
```

Each test has the input (key information, message, signature) and expected result, either “valid” or “invalid”.

Wycheproof intelligence is actually in its test generation code. The code chooses special keys, modifies the input in subtle ways (maybe flips bits in the message, or signs the message with different salt lengths), and encodes known weaknesses. Some examples: invalid ECDH curve points, malformed PKCS1.5 encodings that trigger padding errors, weak key parameters such as e=1 in RSA. For more information, see the issues page.

To use Wycheproof on a new crypto library we need to write a test harness. Harness reads the static test vectors, transforms and encodes the input for the target system, executes the cryptographic operation, and evaluates the result.

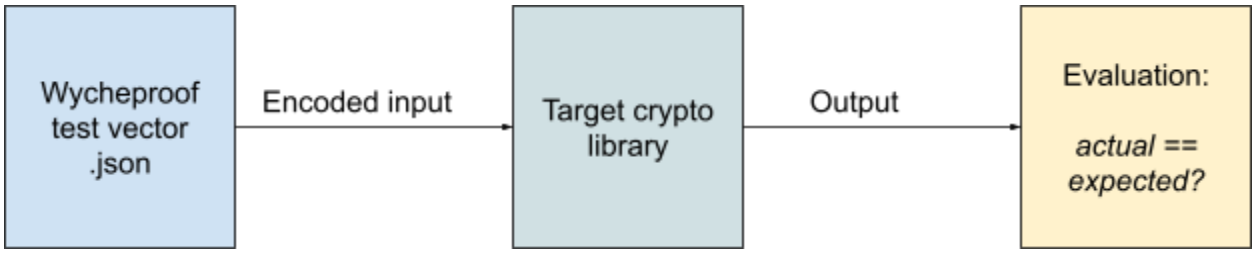


*Figure 9: Wycheproof test harness*

In our SEV review, we wrote two harnesses: a blackbox test which performed the crypto operations on an actual AMD ASP hardware, and a whitebox test which ran the tests on the source code implementation.

For the blackbox test we used the *pek_cert_import* API, and encoded the test data in a self signed *OCA* certificate.

For the whitebox test we refactored the firmware source code, and introduced low level stubs that replaced the cryptographic coprocessor (CCP). In this setup, we compiled and ran the crypto library on a x86 CPU with clang's memory sanitizers (ASAN) enabled. This proved invaluable later in our fuzzing efforts.

## RsaPssVerify out-of-bounds memory write

This bug was found using Wycheproof whitebox tests with ASAN. The RSA-PSS 4096 test case triggered a buffer overflow in the signature verification function. The function copies the seed value from the signature blob into a global, fixed size structure. The structure size was not defined correctly to accommodate for the largest possible seed, which led to a 24 bytes out-of-bounds write. Interestingly, *RsaPssVerify* is part of the boot-loader which runs in SVC mode, so a malformed certificate triggered an overflow directly in the ASP's highest privilege mode.

Exploitation analysis was interesting. The PSS scratch buffer was adjacent to the ECC command input buffer, which holds the curve parameters passed to the CCP.

```
ROM:00016820 PssScratch      RSA_PSS_SCRATCH <0>     ; DATA XREF: RsaPssVerify:loc_ADDE↑o
ROM:00016820                                         ; RsaPssVerify+130↑o ...
ROM:00016FFE ; _BYTE gEccCmdInputBuffer[576]
ROM:00016FFE gEccCmdInputBuffer DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
ROM:00016FFE                    DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
ROM:00016FFE                    DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
ROM:00016FFE                    DCB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

*Figure 10: PssScratch overflow*

An attack we had in mind was racing this overflow with an ECC point multiplication: use the bug to overwrite the first 24 bytes of the prime field of the elliptic curve, right after it's written to the input buffer, but before it is read by the CCP. Weakening the curve could lead to signature forgery and private key recovery.

However, the firmware execution model completely blocks such concurrent attacks: the SEV applet processes a single API command at a time. The design decision to have a single execution thread significantly reduces the risk from concurrent command handling.

## Unchecked memcpy with odd salt length

This bug was found using Wycheproof blackbox tests. An RSA signature with an odd salt length deterministically crashed the host. Note, the ASP fails close, so when an unrecoverable fault is triggered, the ASP throws a machine-check exception (MCE), and reboots the host.

The root cause was a low-level *memcpy* wrapper. The wrapper made sure all memory accesses are 4B aligned, however, there was a bug in the odd length case, where the loop break condition was never hit. This led to an unchecked memcpy, which quickly caused a fault.

Exploitation is prevented since the unchecked write cannot be interrupted. There's nevertheless the risk of a permanent denial of service attack if a maliciously signed image is stored on flash (SPI-ROM).

## Small IV space in swap out operation

SEV *swap_out* operation encrypts guest memory using a per-launch "offline encryption key", in AES-GCM authenticated encryption mode. This mode combines counter mode encryption with Galois (finite field arithmetic) mode of authentication.

Counter mode construction turns a block cipher into a stream cipher: input blocks "{ IV (96b) || Counter (32b) }" go through AES encryption, and produce the key stream with which the message is XOR'ed. Observations:

1. Like all stream cipher algorithms, knowledge of a {plaintext, ciphertext} pair reveals the key stream (XOR). Therefore, the same key stream should never be used twice.
2. A {key, IV} pair determines the key stream. Therefore, the IV should not be repeated for different plaintext messages.
3. The 32b counter should not be under direct attacker control. Increasing / decreasing the counter causes the key stream to *shift*, and might help recover ciphertext fields.
4. The 32b counter should not overflow so as to not repeat the key stream. This places an upper limit of 64 GiB on the size of plaintext data a single {key, IV} pair can protect.

The second limitation on nonce (IV) reuse is actually more severe in GCM - a repeated nonce reveals the GCM MAC authentication key. This "forbidden attack" was exploited in practice on GCM in TLS.

We identified an implementation bug in *swap_out* handler where only 64b of the IV were randomly set on each call. The small IV space lends itself to a practical birthday attack, where a collision is likely to occur after only 2^32 attempts.

Due to spec and hardware limitations, larger IVs could not be used. Synthetic IV construction was deemed too complex, so, instead, the fix moved to incremental 64b IVs.

## Invalid curve point variant

In 2019 we reported a critical vulnerability in the firmware (CVE-2019-9836). The ECDH key exchange implementation was vulnerable to an “invalid curve point” attack, where the firmware is tricked into doing a private key computation on an insecure, low order curve point. By trying all possible values for the small order, the attacker recovers the private scalar bits, modulo the order. The modular residues are assembled offline using the chinese remainder theorem (CRT), leading to a full key recovery.

The issue stems from how point multiplication is implemented. The arithmetic uses only two of the curve’s parameters (“a”,“p”, not “b”), which *expands* the input domain, and correctly computes the multiplication for points not on the target curve. By passing carefully chosen points, an attacker forces the vulnerable firmware to perform private key multiplication on a small order point. As stated above, private scalar bits can be learned indirectly, through the shared session key, and reassembled to recover the entire key.

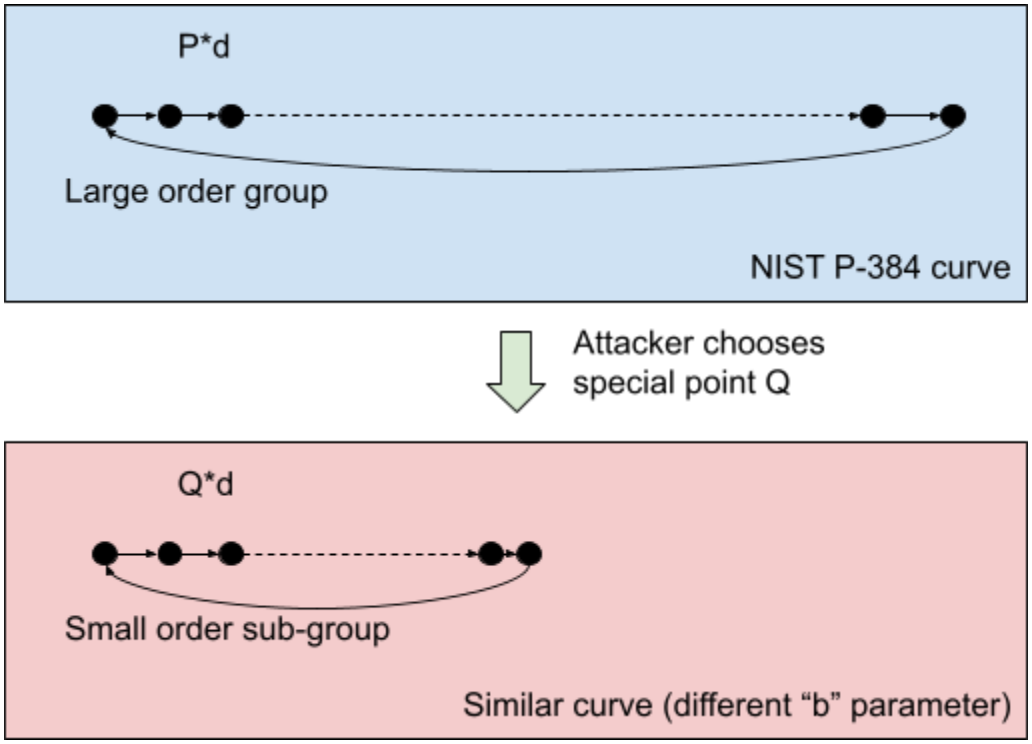


*Figure 11: Private scalar (d) multiplication on an invalid curve point (Q)*

This attack is powerful, easy to implement, and has affected many systems in production, as this recent Bluetooth bug demonstrates. A common fix is adding proper input validation: checking if the given point is on the target curve.

The firmware fix from 2019 was, unfortunately, incomplete. In SEV, the client's point is encoded in a certificate. The certificate, passed through *launch_start* API, encodes the point's (x, y) coordinates and the curve's identifier, either NIST P256 or P384. The fix verified the given point is on the client supplied curve. However, this curve could be different from the firmware's target curve (P384), where the ECDH computation is performed.

This leads to an invalid curve attack variant, with the additional constraint that the attacking point needs to also be a valid P256 point. Finding a point at the intersection of two curves (P256, invalid P384) is possible by *lifting* its coordinates using CRT. We get large (x, y) values, such that when computed modulo prime field p1 (x mod p1, y mod p1) we get a point on the curve over this prime, and when computed modulo prime field p2 (x mod p2, y mod p2) we get a point on the other curve. This is demonstrated in the following Sage script:

```
sage: # Precomputed invalid curve point.
....: inv_x =
0xb4d3dd61980728cba471e99c5b2eb87b4728394d59be5a4800926b07d8c446ffaff29b6d76e8d6d80
c8fdac70445bfd8
....: inv_y =
0xa473bee2e15f939f3653109f4ff53cc574d68df66a521a84c7d1c46122f22919d05cc79b9705bb930
420fb55c008b7
....: inv_b =
0xb97d8a1186c2f9dc1ffa6cd9ca6f58d5c0b8f69311fa1c95fc485760ad61362e408331fd7a84b2566
b351dcdae84e0ca
....:
....: P384_ec = EllipticCurve(FiniteField(P384.p), [P384.a, P384.b])
....:
....: # Same (a, p) different b.
....: inv_ec  = EllipticCurve(FiniteField(P384.p), [P384.a, inv_b])
....: inv_p  = inv_ec.point([inv_x, inv_y])
....:
....: # Low order point.
....: inv_p.order().nbits()
....:
14
sage: P256_ec = EllipticCurve(FiniteField(P256.p), [P256.a, P256.b])
....: p256_p = P256_ec.random_point()
....:
....: # Chinese Remainder Theorem.
....: lifted_x = crt(int(p256_p.xy()[0]), int(inv_p.xy()[0]), P256.p, P384.p)
....: lifted_y = crt(int(p256_p.xy()[1]), int(inv_p.xy()[1]), P256.p, P384.p)
....:
....: # Lifted point is on both curves.
....: assert(P256_ec.is_on_curve(lifted_x, lifted_y))
....: assert(inv_ec.is_on_curve(lifted_x, lifted_y))
....: assert(not P384_ec.is_on_curve(lifted_x, lifted_y))
....:
sage: # CRT results in large 256+384=640 bits numbers.
....: lifted_x.nbits()
```

```
....: lifted_y.nbits()
....:
640
640
```

We couldn't trigger this attack due to the limited coordinate size in the certificate buffer - the firmware didn't accept integers larger than 384 bits. In addition, we couldn't find an efficient way to search for small coordinates (<384b) invalid points that meet the variant constraint above.

The fix for this variant verifies the client's curve matches the expected firmware's curve ID.

## Persistent storage key reuse

The ASP exports its global state, including SEV identity keys, to the hypervisor for persistent storage. The ASP can, alternatively, write this to SPI-ROM, however, this is not always possible as SPI-ROM is often locked to prevent misuse (access is reserved for an out-of-band BMC). In either case, persistent storage must be protected since this is under attacker control.

The ASP protects its persistent storage blob using an authenticated encryption scheme. It derives two keys from a root, fused storage key, one for encryption, one for integrity. Persistent data, a fixed size structure, is encrypted using AES-CTR. Ciphertext is signed using HMAC keyed by the integrity key. {Ciphertext, tag} is stored in persistent storage. On *INIT*, the ASP reads the storage data, verifies the mac, and unwraps the data.

We identified a bug where the same, fixed IV was used to encrypt different messages. As stated above, a given {key, IV} pair generates the same AES-CTR key stream. This leads to a "two time pad attack" where plaintext properties can be learned, and even fully recovered.

This can be exploited, for instance, when the underlying data has a structure with possibly known secret data.

```
typedef struct persistent {
  uint8_t secret[256];
} persistent_t;
```

Imagine *persistent.secret* is first initialized to zeros. We export the data, and record the ciphertext. Then we issue an API call to generate a new secret, export the data, and record the second ciphertext. We recover the keystream from the first ciphertext - the plaintext is known, all zeros, so the keystream is the first ciphertext. Finally, we recover the secret by XOR'ing the second ciphertext with the first. All it took was knowledge of a single {plaintext, ciphertext} pair.

This affected the firmware in the context of the 2019 invalid curve attack, where the private identity key was compromised. An attacker mounts the attack on a vulnerable firmware, exports the data and recovers the persistent storage key stream by XOR'ing ciphertext with the compromised identity key. They upgrade the firmware to a fixed version, generate new identity keys and export the data. They decrypt the persistent data, and recover the *new* identity key from the up-to-date firmware (figure 12).

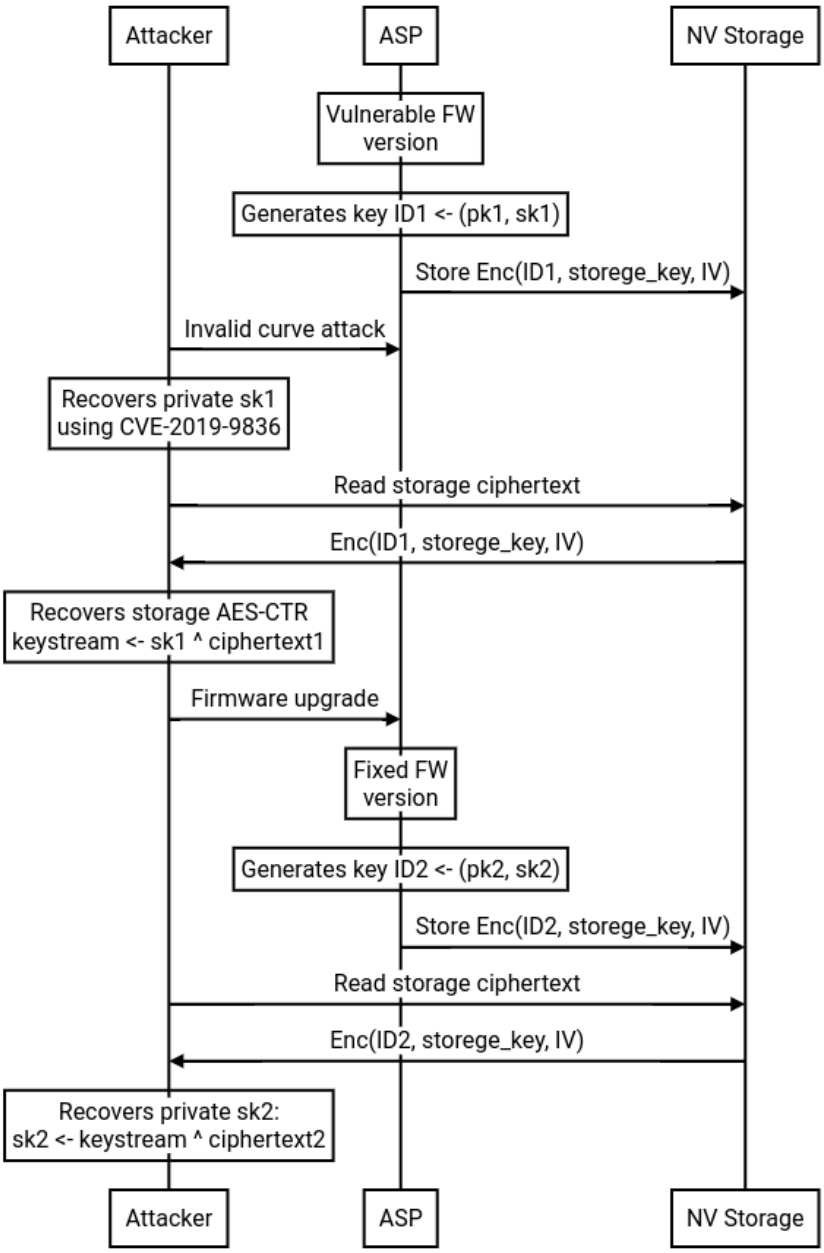


*Figure 12: Key recovery from persistent storage*

A two time pad can also reveal *related messages*, for instance, *persistent.secret* ciphertexts can be grouped into sets where the plaintext secret has the same X most significant bits. Depending on the circumstances, this can open new crypto attacks on the application.

The fix was to generate a random 128b IV each time persistent storage was exported. The IV was included under the MAC, and stored along with the ciphertext and tag.

## Secret dependent operations and side channels

A common source of crypto vulnerabilities is side channels. They come in many forms, from processing time, to changes in shared caches and subtle power measurements. Return error codes are also a form of a (not so obscure) side channel, as Bleichenbacher's padding oracle attack demonstrated. These side channel attacks (SCAs) indirectly leak secret information from the crypto system - private key bits or internal system state.

SCAs are powerful because crypto algorithms are often *fragile* in the sense that even a small leak leads to a full compromise. For example, knowledge of n/4 least significant bits of the private key, d, of an n-bits RSA modulus is sufficient to reconstruct the entire private key. ECDSA is even more fragile, where even a 1 bit nonce leakage leads to practical attacks. This Trail-of-bits blog post presents similar attacks.

SCAs have improved over the years, and researchers have learned how to use statistical tools and ML to extract useful information from noisy measurements. This explains why common crypto attacks target the implementation, not the math.

The ASP has a cryptographic co-processor (CCP) that performs low-level crypto operations, such as EC point multiplication. We confirmed the CCP runs in constant time. Furthermore, the CCP has built-in counter measures against SCAs, mostly aimed at power & radiation analysis. These SCAs require special equipment and physical access to the device, and are mostly out of scope in the SNP threat model. Nevertheless, we confirmed with AMD engineers that ASP power measurements are **not exposed** through the onboard power monitoring unit (PMU), which is available to privileged software (AMD uProf). Finally, we ran statistical tools ("dieharder"), and confirmed the CCP's random number generator (RNG) produces secure, unbiased random output.

## Crypto hygiene recommendations

We acknowledge the difficulty in using cryptography safely, and hope to raise awareness of common cryptographic vulnerabilities. This section, mostly aimed at firmware engineers, lists high level recommendations for how to build cryptography into your systems.

| | |
|---|---|
| Algorithms & Protocols | Follow established standards, such as NIST.<br>Standards have the most up to date recommendations on crypto protocols, algorithms, key sizes and schemes, and have been vetted by experts. |
| | If a custom protocol must be used, it should have proven security properties: secrecy (inc. forward secrecy), authenticity, freshness and strong identity binding. Formal tools (Proverif) can aid verification. |
| | Publish protocol details and seek peer review before implementing it. Publish implementation details and seek peer review before deploying it. |
| | Use strong primitives, but know their limitations (i.e. an upper limit on the number of messages to encrypt).<br>Prefer algorithms that are resistant to misuse. |
| Implementations | Use established, well tested cryptographic libraries (BoringSSL).<br>Prefer implementations that have been formally verified (Project Everest). |
| | If a custom library is used, build safe abstractions, see Tink, for example. It should be hard for library users to misuse cryptography. |
| | Wycheproof and differential tests can help verification. |
| Keys | Use a secure random number generator (RNG) for key generation. Statistical tools (dieharder) can help verify the source is secure and unbiased. |
| | A key should only be used for a single purpose, either encryption or signing. Its parameters (key size, digest algorithm, domain parameters) should be fixed and included in the key. Avoid in-band key/protocol negotiation.<br>If multiple keys are needed, use a secure key derivation function (KDF). |
| Encryption | Use authenticated encryption, and don't blindly trust ciphertext. Authenticate context information using additional authenticated data (AAD) construct.<br>If unique IVs are required, prefer synthetic IVs that are a MAC of the message, and avoid repeated nonces. |
| Signatures | Signatures should include their intent. Known as "The Horton principle", it states that message syntax should be unambiguous. |

| | |
|---|---|
| | Signature should be calculated over all the important data. |
| Agility | Build crypto agility, and plan for a post-quantum world. |
| Common pitfalls & things to avoid | Bias in random selection. |
| | Key + IV reuse; Nonce reuse. |
| | Unauthenticated encryption. |
| | Ambiguous message signatures. |
| | Secret dependent operations.<br>This might lead to observable side channel measurements (processing time, shared resources, power, etc). |
| | Collisions in a small search space. Be mindful of the "birthday bound". |
| | Decryption / signing oracles. Be mindful when exposing one. |
| | Custom protocols and implementations that have not been peer reviewed. |

For security engineers who want to improve their crypto review skills we recommend the following resources: 1) CTF style challenges. Cryptopals is an excellent resource, so are Google's CTF challenges. 2) Good text books. "Algorithmic Cryptanalysis" by Antoine Joux and "Introduction to Modern Cryptography" by Katz and Lindell give solid foundations. 3) Real world crypto talks.

# SEV-SNP review

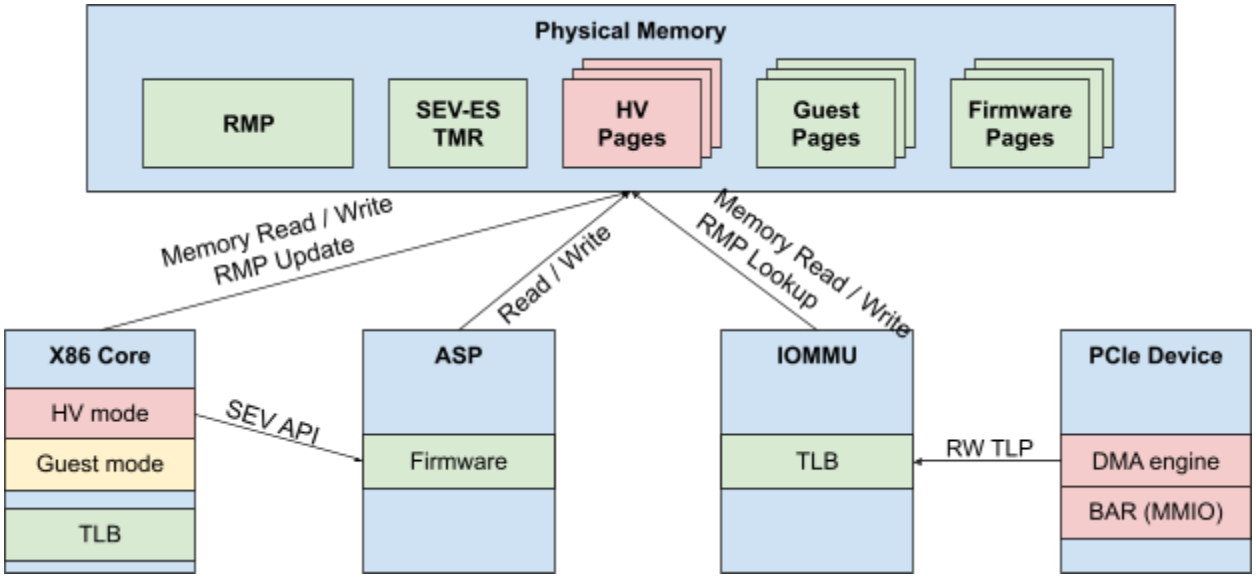


*Figure 13: SNP System Components*

Before we dive into the SNP security findings, we describe the different system components, their capabilities and trust relationship (figure 13).

| | |
|---|---|
| **Physical memory** | **RMP**: single table, tracks page state. |
| | **SEV-ES** trusted memory region (**TMR**): a 1MiB region accessible only to x86 uCode on VM entry/exit, and the firmware.<br>Holds integrity values for SEV-ES guests' registers. |
| | **Unassigned pages** (HV pages): writable pages owned by the hypervisor. Another type of writable pages are *default pages*; these, by definition, are not covered by the RMP. |
| | **Assigned pages**: pages assigned to SNP guests or the firmware. These include: encrypted guest memory, firmware pages, guest context pages, metadata pages.<br>X86 in hypervisor mode cannot write to these pages.<br>X86 in guest mode can only write to pages owned by its ASID. |
| **X86 Core** | Updates RMP through *RMPUPDATE, PVALIDATE, PSMASH, RMPADJUST* instructions. |
| | HV interacts with the firmware through its API functions. |
| | X86 reads / writes to physical memory. |
| | X86 checks RMP on memory writes. |
| | X86 uCode reads/writes integrity value in SEV-ES TMR on VM entry |

| | |
|---|---|
| | and exit. Only applicable for SEV-ES guests. |
| | RMP entry bits are cached in the X86 TLB. TLB is flushed on RMP updates (this happens in uCode). |
| **ASP** | Firmware transitions page states through RMP writes. |
| | Firmware initializes the RMP, and keeps it secure and consistent.<br>Security: RMP pages are marked “assigned firmware” pages in the RMP. This prevents modification to the RMP.<br>Consistency: there’s a 1:1 mapping from a guest physical address to a host physical address. |
| | Firmware performs many security checks on INIT: SNP is enabled on all cores, RMP addresses are programmed the same on all cores and so on. |
| | Firmware implements new SNP features: launch sequence, attestation command, secrets page. |
| **IOMMU** | Translates IO addresses to host physical addresses using a hypervisor controlled page table. |
| | IOMMU checks RMP on memory writes. IOMMU prevents writes to assigned pages. |
| | RMP entry bits are cached in the IOMMU TLB. TLB is flushed on RMP updates. |
| **PCIe device** | Issues DMA writes to IO addresses using PCIe transaction layer packets (TLPs). |
| | Exposes its base address register (BAR) over memory mapped IO (MMIO). |
| **Trust model** | Generally speaking, all components labeled in **green** in Figure 13 are considered trusted - not under direct attacker control. |
| | X86 in hypervisor mode and PCIe devices are under attacker control. So are all unassigned pages, and parameters passed to the firmware. |
| | It is also assumed the attacker controls system initialization at boot time, so, for instance, they can set RMP MSRs to any value. |

The attacks we describe below can be classified as:

- “**Confused deputy**”: hypervisor tricks the highly privileged firmware into writing data over assigned pages.

- **Misconfiguration**: hypervisor configures the system such that disparate components negatively interact with each other.
- **Race conditions**: hypervisor races security checks (TOCTOU) or non-atomic state transitions where the system is inconsistent.
- **Bad cleanups**: cached values do not reflect actual RMP.

## SEV-ES integrity pointer falls outside TMR

VM save area (VMSA) is a guest page that holds the encrypted vCPU's register contents. It holds a pointer to the integrity pool TMR, a reserved memory region managed by the firmware.

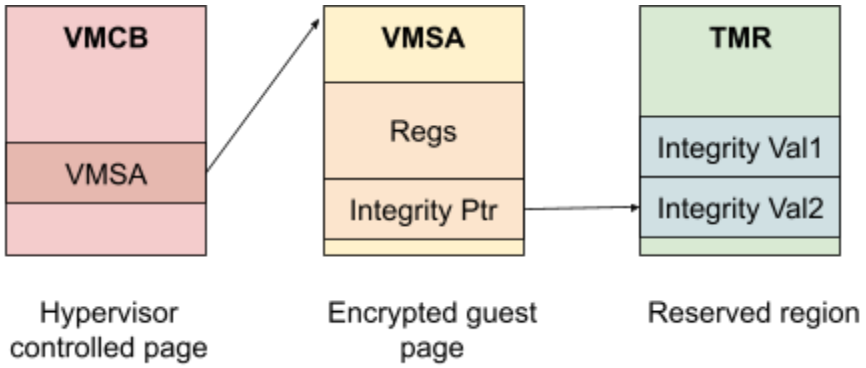


*Figure 14: VMSA integrity pointer*

X86 uCode dereferences the pointer on two events: 1) vmrun, where it reads the expected integrity value (TMR.integrity_val entry) and compares it to the actual one (checksum of vmsa.regs), 2) vmexit, where it stores the latest integrity value in the TMR.integrity_val array.

We identified a bug in the uCode where it allowed pointers in a 2MiB TMR region, where, in fact, the firmware constructs a smaller, 1MiB region. This means, we can get an unchecked write - a memory write that doesn't go through the RMP access control - to any page adjacent to the SEV-ES TMR.

We successfully exploited this to overwrite the policy field in an SNP guest context page. Exploitation is rather involved (figure 15):

1. Launch an SEV-ES guest. Update VMSA values using *dbg_encrypt* API: point integrity pointer to an unassigned page, adjacent to the TMR. Place expected integrityl value so the vCPU could run.
2. Launch vCPU. On vmrun, the integrity value is read from the hypervisor page. It matches the expected registers' checksum value, and vCPU enters execution in guest mode.
3. In guest mode, run a simple busy loop before exiting. This gives a short time window for the hypervisor to carry on the attack.

4. In the hypervisor, *RMPUpdate* the hypervisor page, and transition it to firmware state. Create a new SNP guest context page using *snp_gctx_create* API.
5. Exit the guest. On vmexit, we get an unchecked write over the new context page.

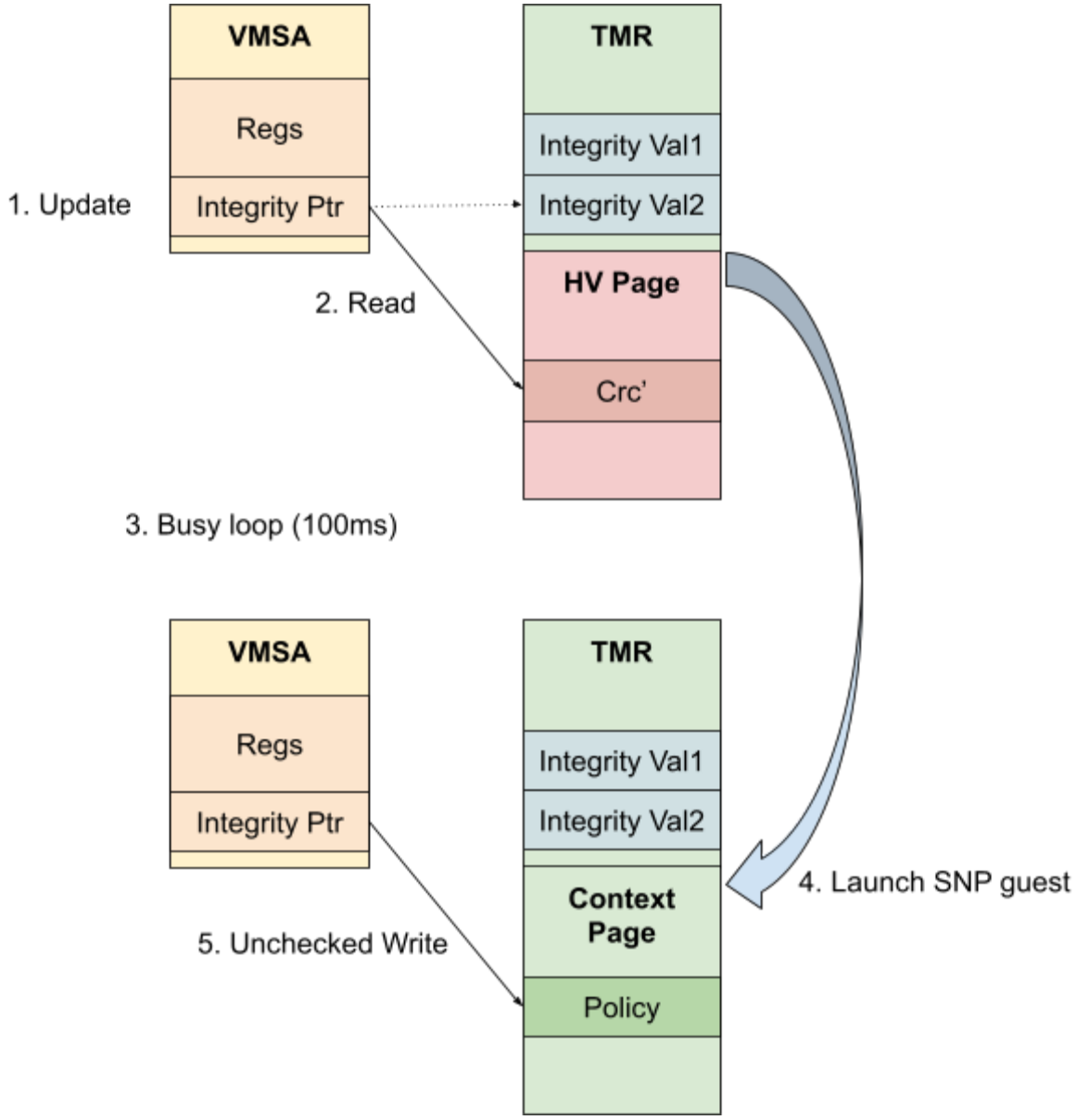


*Figure 15: Integrity pointer attack*

We further extended the attack to get a **controlled unchecked write**: by setting the vCPU's register values on exit, we force the checksum to a given value.

The fix involved the FW initializing a 2 MiB region instead of 1 MiB.

# RMP degradation attack

An unchecked write can be leveraged to mount a generic and powerful attack, “RMP degradation attack”, where the RMP’s self-protecting entries are modified to unsecure ‘hypervisor’ state. In this state, a malicious hypervisor can transition *any* page to *any* state simply by writing to the RMP. State transitions are no longer managed by the firmware, and all SNP security features are lost.

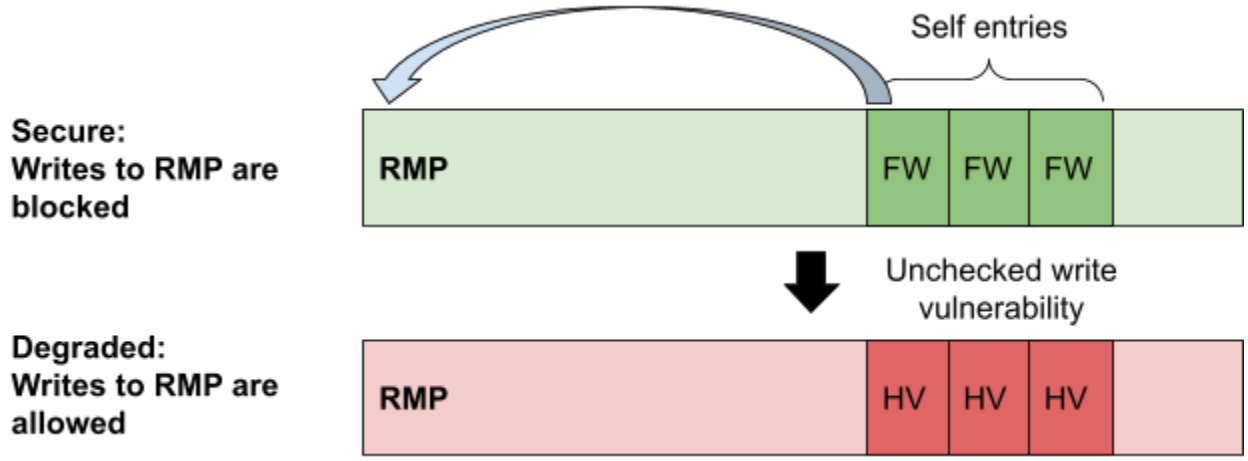


*Figure 16: RMP degradation attack*

For example, a degraded RMP enables the following attack:

1. Hypervisor creates a guest context page with a restricted policy.
2. SNP guest is launched, attestation includes a restricted policy.
3. Hypervisor writes to degraded RMP, and transitions the guest context page to ‘hypervisor’ page. This enables write access.
4. Hypervisor writes to the *snp_policy* field in the context page. With high probability, the change flips the “debug enable” bit when decrypted.
5. Hypervisor writes to degraded RMP, and transitions the guest context page back to ‘context’ page.
6. Hypervisor calls *snp_decrypt*. Firmware checks ‘debug enable’ bit in snp_policy field, and returns decrypted guest memory.

# Unsafe firmware accesses to ‘hypervisor’ pages

The SNP API specification has a fascinating diagram that depicts the SNP page state machine (figure 17). From an attacker point of view, we’re looking for unsafe transitions, where the hypervisor changes the page’s state while it is being processed by the firmware.

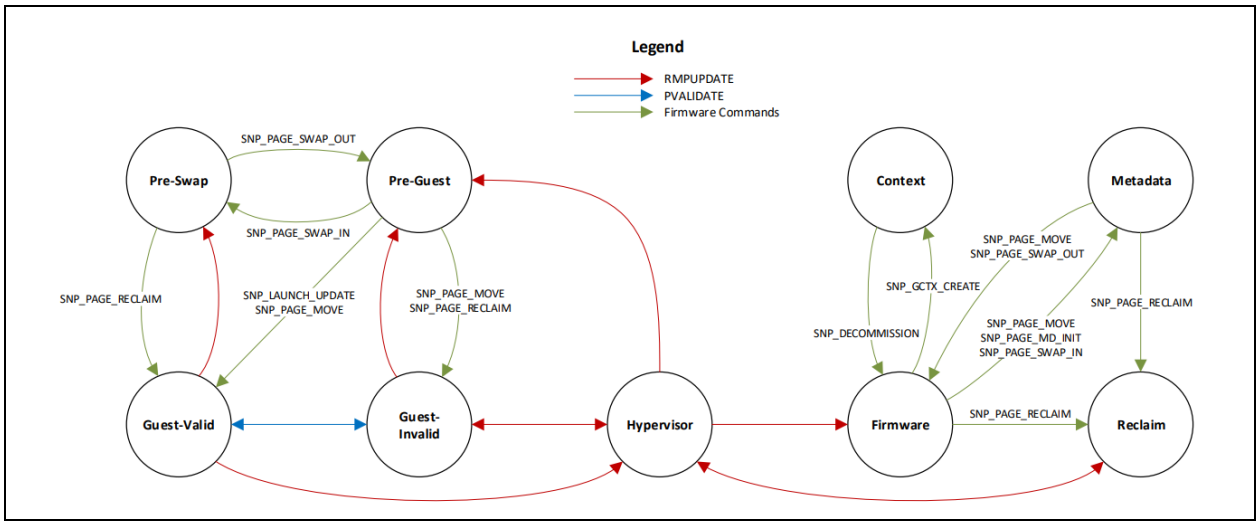


*Figure 17: SNP page state machine*
*(Diagram from AMD SEV Secure Nested Paging Firmware ABI Specification)*

Following the red and blue outbound edges in figure 17, it is clear that the host CPU can transition a ‘hypervisor’ page to ‘guest-invalid’ (RMPUPDATE), and to ‘guest-valid’ (PVALIDATE) states without involving the firmware. This means that firmware commands that accept ‘hypervisor’ output pages are susceptible to an unsafe race condition.

After we identified the potential bug in the documentation, we went out and looked for it in the code. We identified the API *get_id* command handler that accepted an output page in ‘hypervisor’ state. We confirmed this unsafe access could compromise SNP security.

1. Hypervisor calls *get_id* with an unassigned output page.
2. Firmware checks the RMP, validates the page is in ‘hypervisor’ state, and proceeds.
3. Hypervisor transitions the page to ‘guest-invalid’ using RMPUPDATE instruction.
4. Guest transitions the page to ‘guest-valid’ using PVALIDATE instruction.
5. Guest writes data to it.
6. Firmware completes *get_id* computation, writes results to the output page, and overwrites guest memory.

To avoid this bug class, the fix refactored many SEV command handlers, and verified all output pages are in assigned ‘firmware’ state before being written to.

## Firmware misidentifies *VM_HSAVE_PA* page state

The SNP API specification defines the different page states based on the RMP entry bits (figure 18).

| Page State | Assigned | Validated | ASID | Immutable | GPA | VMSA |
|---|---|---|---|---|---|---|
| Hypervisor | 0 | 0 | 0 | 0 | - | - |
| Reclaim | 1 | 0 | 0 | 0 | - | - |
| Firmware | 1 | 0 | 0 | 1 | 0 | 0 |
| Context | 1 | 0 | 0 | 1 | 0 | 1 |
| Metadata | 1 | 0 | 0 | 1 | >0 | - |
| Pre-Guest | 1 | 0 | >0 | 1 | - | - |
| Guest-Invalid | 1 | 0 | >0 | 0 | - | - |
| Pre-Swap | 1 | 1 | >0 | 1 | - | - |
| Guest-Valid | 1 | 1 | >0 | 0 | - | - |
| Default | See discussion below. | | | | | |

*Figure 18: SNP page state definitions*
*(Table from AMD SEV Secure Nested Paging Firmware ABI Specification)*

We identified a flow that results in an RMP entry that doesn’t fall into any row above. When x86 writes to the *VM_HSAVE_PA* MSR, it sets its *RMP.immutable* bit to 1 to prevent modifications by *RMPUpdates*. This results in a page with *assigned = 0* and *immutable = 1*, a special edge case for SNP firmware.

The firmware function that gets the page state is implemented in a sequence of if-else statements that match the table above. When none of the if-statements match (all possible SNP states), the function returns the ‘default’ state enum. ‘Default’ pages have special semantics: these are pages that are not covered by the RMP, and are, therefore, assumed to be owned by the hypervisor.

The last piece of the bug is the *swap_out* command handler. It supports copying either 4KiB or 2MiB pages from a source address to a destination address. The destination address can be a ‘default’ page, but in this case the *RMP.page_size* field is not checked, as default pages are not covered by the RMP.

Put together, this leads to an exploit where the firmware is tricked into writing over assigned pages:

1. Hypervisor places a ‘hypervisor’ page next to an assigned firmware page.
2. Hypervisor points *VM_HSAVE_PA* MSR to this page. X86 uCode sets its immutable bit to 1.
3. Hypervisor calls *swap_out* with *VM*_HSAVE_PA as destination page and page_size = 2MiB.
4. Firmware misidentifies dest page as ‘default’ page, and skips the destination page size check.
5. Firmware copies 2MiB data from source, and overwrites assigned pages.

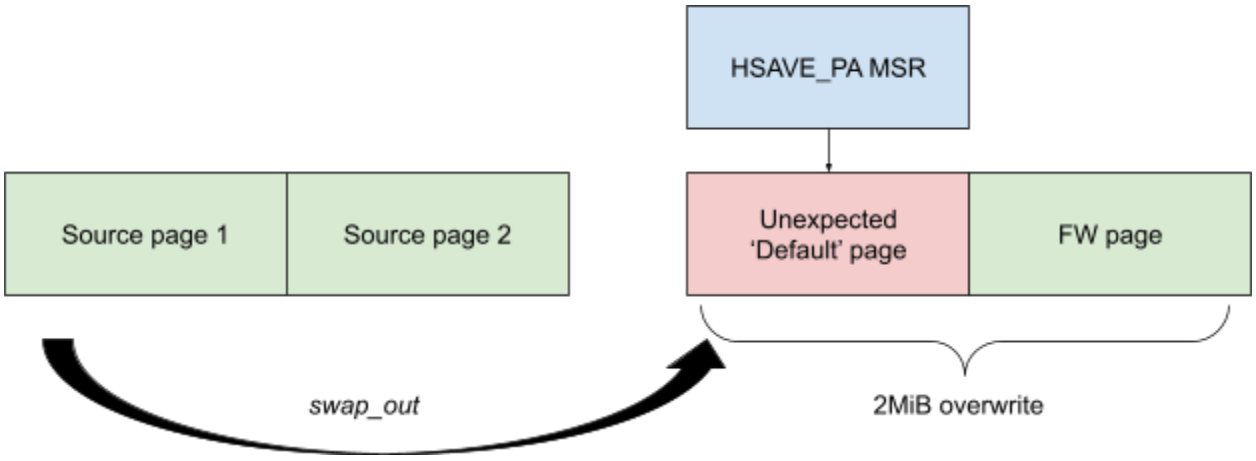


*Figure 19: Swap out overflows misidentified ‘default’ page*

This bug is interesting because on the surface, each component seems to be doing the right thing: setting the ‘immutable’ bit is safe, returning the ‘default’ state is a reasonable return code based on the table in the documentation, skipping RMP page size check is correct for pages that are not covered by the RMP. Nevertheless, a bug emerged, which led to a deterministic, attacker controlled memory overwrite.

The fix refactored the *get_state* function in the firmware, and handled all SNP state cases.

## PCIe Screamer tests

To simulate a malicious PCIe device, we used *LambdaConcept*’s PCIe screamer device. The screamer is a custom FPGA with a PCIe core, and can read and write arbitrary packets on the PCIe fabric. Its runtime firmware is a custom pcileech-fpga bitcode by Ulf Frisk.

We used two software stacks to run our security tests: LeechCore by Ulf Frisk written in C, and go-pcie-screamer and go-pcie-tlp libraries written in Go.

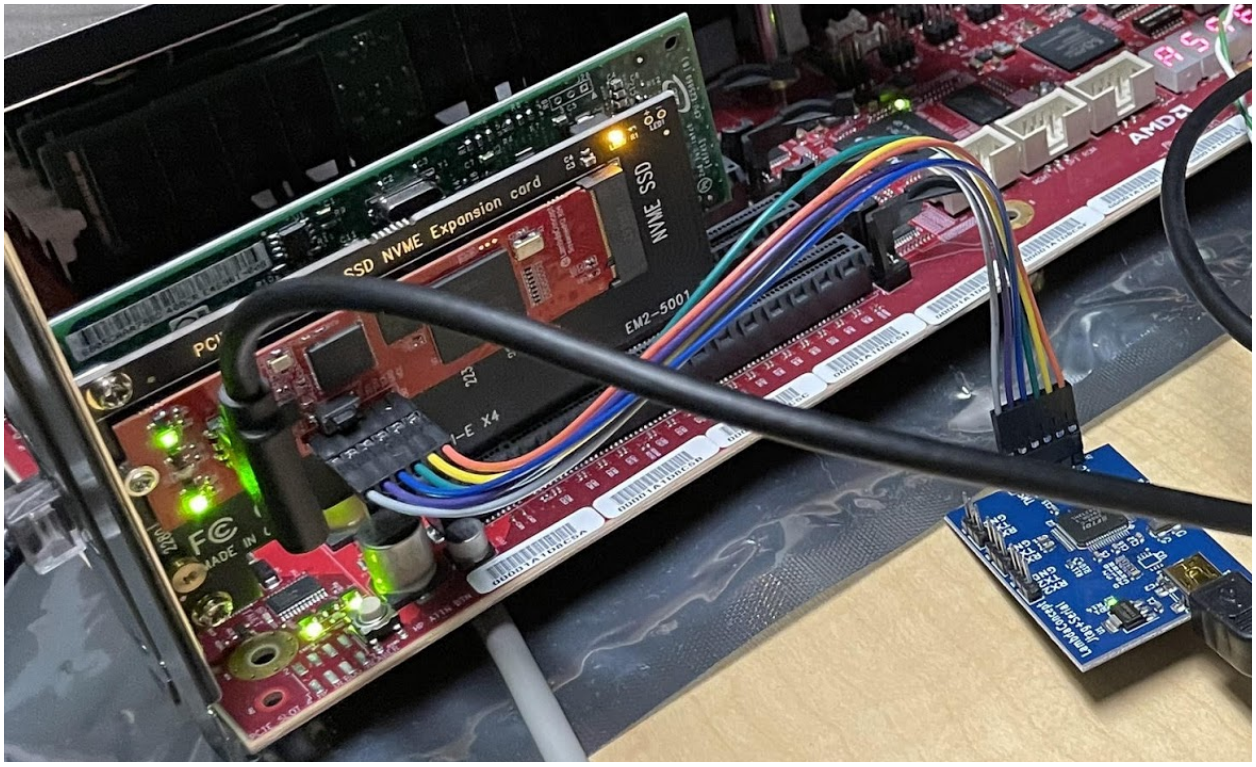


*Figure 20: PCIe hardware test equipment*

Using the screamer, we simulated two attacks: 1) device sends rogue DMAs targeting assigned pages, 2) device presents custom BAR contents over MMIO. These attacks surfaced two main vulnerabilities in the firmware, listed below.

## IOMMU TLB not flushed on SNP-INIT

The IOMMU consults the RMP on translations, and blocks write access to assigned pages. In some cases, the IOMMU caches RMP entry bits in its IOTLB. To maintain security properties, and prevent stale IOTLB entries with old RMP bits, it is critical to flush the IOTLB on RMP updates.

We were able to confirm x86 uCode flushes the IOMMU TLB on *RMPUpdate* instruction. The firmware cannot flush the IOTLB. However, this is safe since the firmware never transitions an unassigned page to an assigned state.

Nevertheless, we identified an edge case for the firmware during SNP initialization. An attacker can populate the IOTLB prior to calling *snp_init*, and get a stale translation with RMP bits that don’t reflect the actual table.

Concretely, an attacker poisons the IOTLB such that the RMP's self entries are marked 'hypervisor' state in the IOTLB - their state before *snp_init* - even after *snp_init*. This leads to an unchecked write over the RMP's self entries, and mounts the "RMP degradation attack" described above (figure 21).

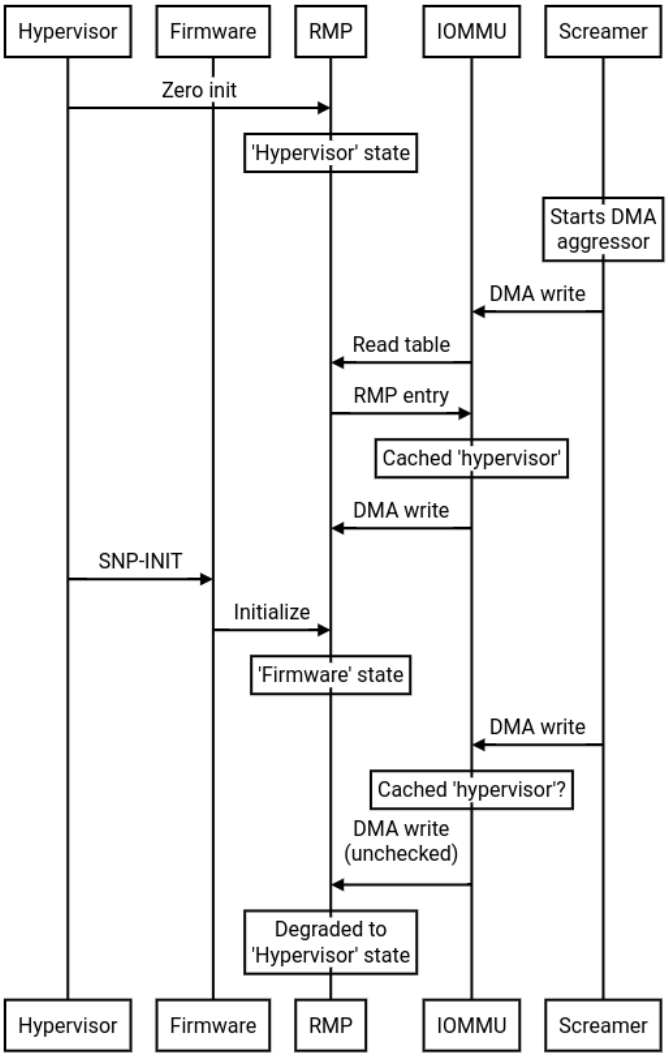


*Figure 21: Stale IOTLB entries lead to unchecked writes*

Due to hardware design limitations, the ASP cannot flush the IOTLB. Instead, the fix leaves TMRs in place to enforce that RMP memory remains read-only from the IOMMU standpoint even after *snp_init* finishes.

## Firmware accepts malleable MMIO pages

System physical address space is divided into regions of DRAM backed pages, and memory mapped IO (MMIO) pages (figure 22). The north bridge data fabric (DF) routes memory requests based on the layout configuration, which is initialized by the BIOS on reset, and may be locked.

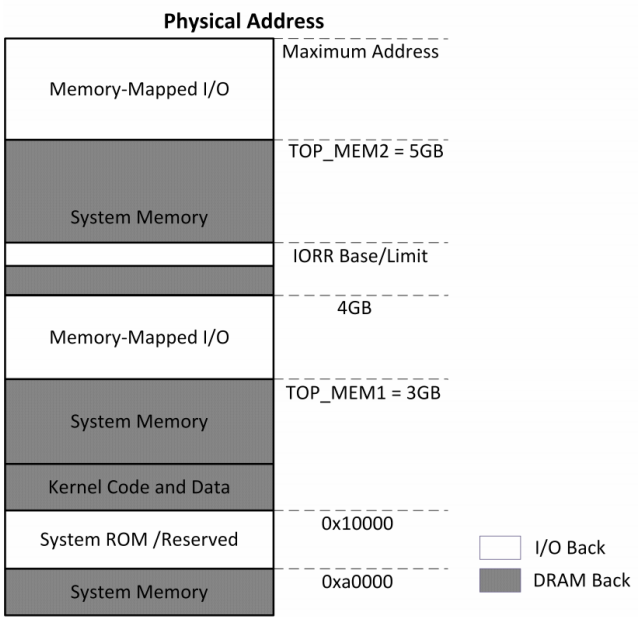


*Figure 22: Physical address layout example*

A PCIe device exposes a set of registers, called base address register (BAR), over MMIO. These pages can be accessed directly by the CPU using load/store instructions: when the CPU executes a load/store opcode targeting MMIO, the north bridge issues a memory read/write transaction layer packet (TLP) over the PCIe fabric, targeted at the PCIe endpoint. On reads, the PCIe device responds with a completion packet with the data. Writes are “posted operations”, and do not return any response (figure 23).

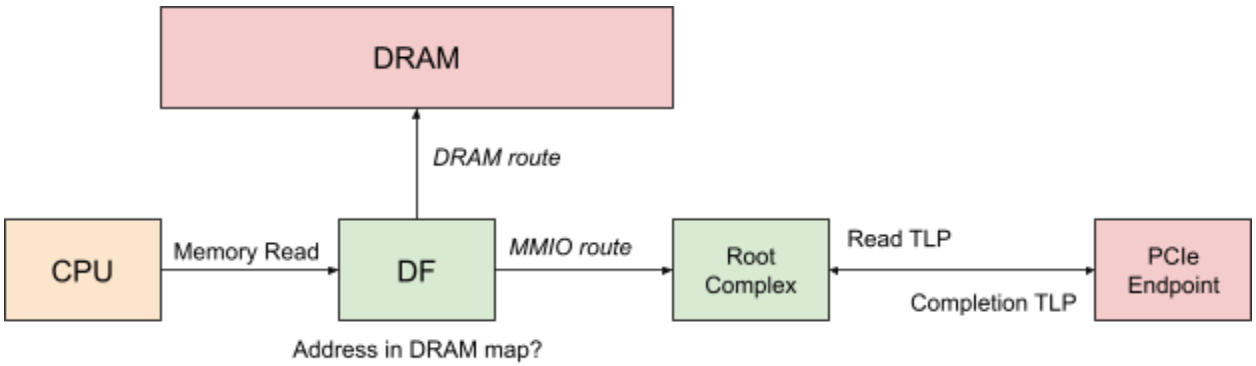


*Figure 23: Read from PCIe BAR*

Using the screamer, we can return arbitrary contents for the BAR's page, as demonstrated in the following script:

```
// A tool that returns fake BAR contents.
// It listens to memory read TLPs targeted at the device memory mapped region,
// and responses with completion TLPs.
//
// Enable memory space:
//   # setpci -d 10ee:0666 COMMAND=2
//
// Read from device BAR0:
//   # ./iotools/mmio_dump 0xfcc00000 32
//   0x00000000fcc00000: 0x41414141 0x41414141 0x41414141 0x41414141
//   0x00000000fcc00010: 0x41414141 0x41414141 0x41414141 0x41414141
//
package main

import (
        "flag"

        "github.com/golang/glog"
        "github.com/google/go-pcie-screamer/screamer"
        "github.com/google/go-pcie-tlp/pcie"
)

func main() {
        flag.Parse()

        // Enumerate and open all screamer devices on the system.
        devs, err := screamer.OpenScreamers()
        if err != nil {
                glog.Fatalf("OpenScreamers failed: %v", err)
        }

        // Close screamers on exit.
        defer func() {
                for _, d := range devs {
                        d.Close()
                }
        }()

        // Create an io.ReadWriter for reading and writing TLPs.
        rw := screamer.NewTLPController(devs[0])
        devid := pcie.NewDeviceID(devs[0].DeviceID())
```

```
        // Fixed response buffer.
        res := make([]byte, pcie.MaxTLPBuffer)
        for i := range res {
                res[i] = 0x41
        }

        // Respond to memory read TLPs.
        b := make([]byte, pcie.MaxTLPBuffer)
        for {
                n, err := rw.Read(b)
                if n == 0 {
                        continue
                }
                mrd, err := pcie.NewMRdFromBytes(b[:n])
                if err != nil {
                        glog.Warningf("Failed to parse MRd: %v", err)
                        continue
                }
                cpl, err := pcie.NewCplForMrd(devid, pcie.SuccessfulCompletion,
mrd, res[:mrd.DataLength()])
                if err != nil {
                        glog.Warningf("Failed to build Cpl: %v", err)
                        continue
                }
                n, err = rw.Write(cpl.ToBytes())
                if err != nil || n != len(cpl.ToBytes()) {
                        glog.Warningf("Failed to write Cpl: %v", err)
                        continue
                }
        }
}
```

In our research, we discovered that the firmware's physical memory accesses go through the same DF routing, meaning that MMIO addresses are respected, and the firmware might read from or write to malleable MMIO pages.

We confirmed that *RMPUpdate* instruction successfully transitions an MMIO address to 'firmware' state. We also confirmed that the firmware does not distinguish between DRAM / MMIO ranges, and successfully accepts MMIO 'firmware' pages. This bug immediately breaks SNP security features. We implemented an attack where a guest context page maps to a hypervisor controlled MMIO page. The hypervisor launches an SNP guest with a restricted policy, then issues *dbg_decrypt* and flips the "debug enable" bit by returning different page contents.

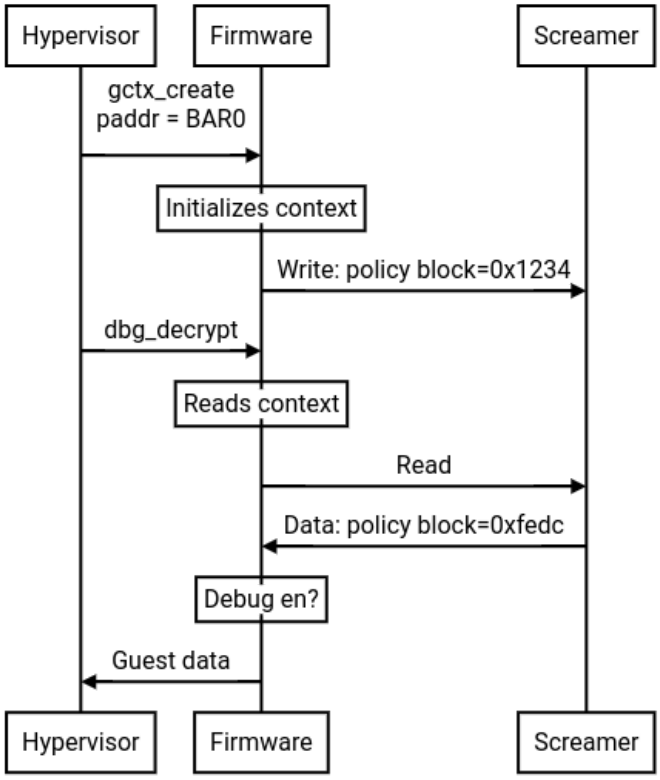


*Figure 24: Malleable guest context page*

The fix refactored the firmware's low level memory mapping functions. It validated that the DF layout is locked, and all memory addresses are in DRAM map, not MMIO.

## Untethered guest context attack

"RMP degradation attack" described a powerful attack that leverages a single unchecked write to break SNP security features. However, that attack is quite visible - the RMP's self-protecting entries are not consistent (not in 'firmware' state), which can be detected by the firmware.

We researched other, stealthier, techniques to break SNP security, given an unchecked write primitive.

The attack breaks the invariant that an active ASID cannot be assigned to multiple guests. We leverage an unchecked write to rollback the ASID field in a guest context page. Guest context pages are encrypted using the inline AES engine. They are integrity protected through RMP access checks ('assigned' bit). With an unchecked write we can modify the ASID AES block. By writing a previously seen ciphertext block, we effectively rollback the ASID field to a known value. Ciphertext rollback attack is a common technique against crypto heavy applications.

The result is an untethered guest context page: a context page with a fixed ASID value and a permissive policy (debug enable bit set) which can be reused to decrypt victims' guest memory. This attack is effective because it only needs to succeed *once*: the untethered gctx page can be reused to compromise all future SNP guests. In addition, it leaves the RMP in a consistent state.

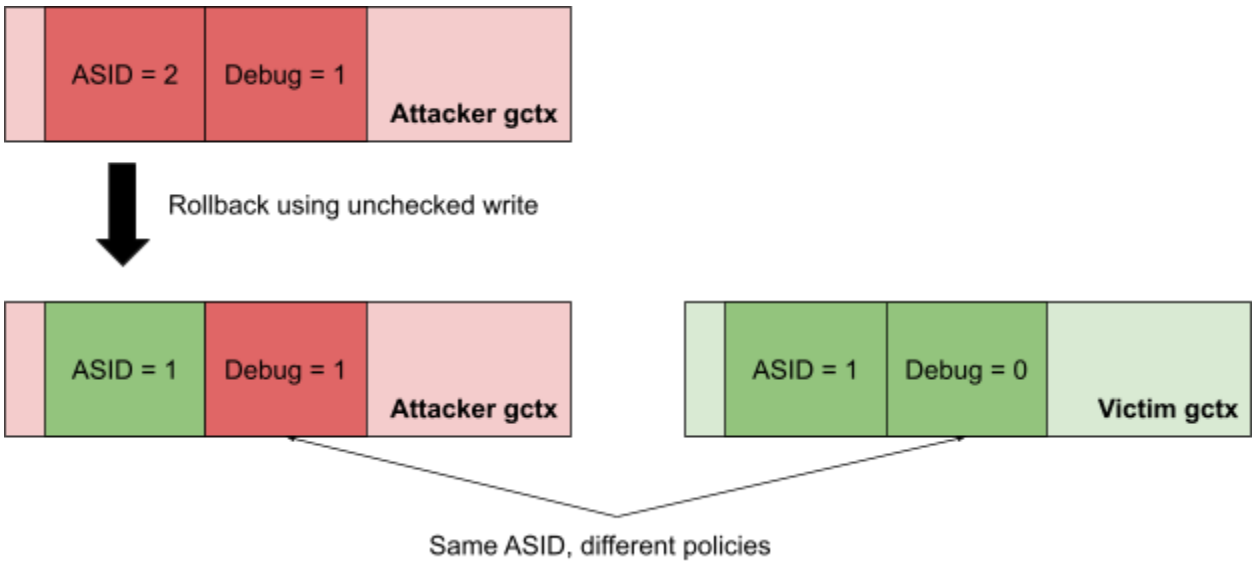


*Figure 25: Untethered guest context attack*

## Firmware races with RMPUPDATEs test

An RMP entry is a relatively large structure, 128b in size. It is updated by either 1) x86 uCode when a page transitions to 'firmware' state using the RMPUPDATE instruction, or 2) ASP firmware when a page transitions between 'assigned' states.

We confirmed that all RMP updates are atomic, such that firmware updates and CPU RMPUPDATEs *cannot* interleave, and *cannot* result in inconsistent RMP entries. This is done using an internal synchronization mechanism that guarantees atomicity.

## INVD test

AMD64 Architecture Programmer's Manual, Volume 3, states that the instruction `INVD`, when executed at CPL 0, "Invalidates all levels of cache associated with this processor", and that "No data is written back to main memory from invalidating the caches". If this was true, it would imply that a malicious hypervisor could first selectively flush some cachelines to RAM with `CLFLUSH`, then effectively revert all other changes that are still in dirty cache lines by executing

INVD. This would break some of the consistency that SNP attempts to provide - it could be abused to roll back changes to RMP entries or to corrupt guest state.

This was tested as follows:

1. initialize the RMP and enable SNP on all cores
2. shut down all CPU cores except for core 0
3. flush the caches with `WBINVD`
4. perform a memory write (tested with both a normal store and `RMPUPDATE`)
5. try to effectively roll back the change by executing `INVD`

Both with a normal store instruction and with `RMPUPDATE`, the newly written value was still visible after `INVD`. This suggests that `INVD` does not actually perform an invalidate-without-writeback, at least when SNP is enabled.

## Fuzzing efforts

As mentioned above, there are multiple different components and attack surfaces that interact with x86. One of the largest areas of ASP firmware that is (almost) directly exposed to the hypervisor are the commands used to communicate with the SEV firmware.

These commands and the associated structures can be seen in the upstream Linux kernel driver.

We built a standalone structure-aware white-box fuzzer for this SEV interface using libFuzzer, and spent some time monitoring the coverage and ensuring that the fuzzer was able to provide inputs that met the minimum preconditions for each command.

The state machine associated with the original SEV interface was already very complex, and SEV-SNP added a similar number of additional new commands. There are a lot of possible edges that are simply disallowed, and effective fuzzing requires a balancing act between enforcing too much structure on the fuzzer inputs, and too little (resulting in either uninteresting coverage, or no valid coverage at all).

When fuzzing an application that has heavy use of cryptography, it's often desirable to remove or neuter large amounts of the cryptographic code for performance reasons. In the case of the SEV firmware, this needed to be done with careful consideration, since there are also a significant number of areas in the code where this cryptography is used to ensure the integrity of data originally stored by the ASP, using keys that are only held by the ASP, where it would not be possible for the hypervisor to bypass these checks.

The fuzzer was successful in rediscovering a number of shallow issues that had been previously discovered and reported during the review, and also in finding an interesting issue detailed below that we hadn't previously spotted.

## ASID underflow

As part of the lifetime management for SEV-SNP guest, the firmware needs to know which ASIDs are currently in use, and which are available to be allocated to new guests. These states are tracked in arrays of per-CCX bitmasks, and as an optimisation in the decommission flow, the ASP can directly clear these state-tracking bits for any CCX's which never scheduled the guest, reducing the need for expensive TLB flushing operations when they are not needed.

However, in the case where a guest is created and decommissioned without being fully started, no ASID would be assigned to that guest (it would have a 0 ASID). This case was not handled correctly in the decommission path, and this would result in an off-by-one underflow where the firmware would perform these operations on out-of-bounds data.

The layout of the ASID state tracking structures mean that this resulted in unexpected state transitions for the last valid ASID, and a malicious hypervisor could use this to create multiple guests that shared the same ASID.

## Rowhammer discussion

Given that the integrity of the RMP is critical to the SEV-SNP security model, Rowhammer type attacks which allow the x86 to modify memory contents "at-a-distance" are a clear concern. The RMP is also an excellent target for Rowhammer type attacks, since a relatively small amount of control over the contents of RMP entries can lead to a complete compromise (see "RMP degradation attack" above for an explanation of why the RMP self-entries are a weak point).

In the case of the RMP self-entries, even if the entries are stored encrypted in DRAM and any corruption of this data would result in completely random plaintext, this is still very likely to result in a practically exploitable situation where the x86 can use RMPUPDATE to gain direct write-access to the RMP.

This means that effective Rowhammer mitigation is a key part of the SEV-SNP security model. The SEV-SNP architects were aware of this, and during boot the ASP ensures that the DRAM has enabled Rowhammer mitigations.

We didn't directly research the details of the Rowhammer mitigations implemented in current DDR4 server DIMMs, or Rowhammer attacks that would bypass both the existing mitigations

and modern ECC implementations, but given recent advances in the state-of-the-art, this is an area for future research.

## Kernel-to-SMM privilege escalation

As mentioned in the overview, the ASP exposes a mailbox interface to privileged BIOS software. The intent is to let BIOS inform the ASP of system state transitions like POST, and let BIOS query ASP properties like firmware versions.

The mailbox protocol is a simple message exchange over MMIO registers. Message arguments can encode physical memory addresses. Notably, this interface is available at runtime to privileged BIOS software running in System Management Mode (SMM), and to less-privileged kernel ring0.

The firmware accepts physical addresses that point to SMRAM - SMM protected memory region, however, these are accepted only if the caller is running in SMM context.

We identified an implementation issue, a TOCTOU vulnerability in the firmware, where the caller mode is checked a while after the command is pulled from the mailbox. This window gives a malicious kernel sufficient time to submit a BIOS command with address pointing to SMRAM, switch to SMM, and bypass the context check. A “boomerang” attack where a ring-0 attacker tricks the ASP into corrupting SMM memory, leading to privilege escalation.

This bug demonstrates how the ASP enabled new attack vectors against existing, privileged system components.

# Mitigating Controls

Mitigating controls are a set of tools that make exploitation harder, and help systems recover from vulnerabilities. In this last section we recap AMD Secure Processor defense strategies.

| **Strategy** | **Implementation examples** |
| --- | --- |
| Defense in depth | ASP secure boot: on-chip bootrom authenticates off-chip boot loader. BL authenticates trustlet images. In addition, the bootrom fails close on authentication failures. |

| | |
|---|---|
| | SEV firmware runs in a lower privilege mode (USR). Highly privileged keys, such as the root storage keys, are protected in the CCP. Therefore, there are multiple barriers that protect certain root keys. |
| | Firmware versions and system settings are part of the SNP attestation report. This helps the VM owner assess the security state of the firmware. This visibility also helps host providers bring the system to a good known state. |
| Attack surface reduction | Single threaded execution model removes the risk of race conditions in concurrent command processing. Masking interrupts prevents unexpected control flows, and offers better reliability. |
| | No dynamic heap allocations prevent the risks commonly associated with bad object management, such as use-after-frees. |
| | ASP power measurements are not exposed through the power monitoring unit. |
| Crypto hygiene | SNP firmware relies on NIST approved protocols and algorithms. Keys are used for a single purpose, and cryptographic signatures include their intent. |
| | Most crypto operations are deferred to a dedicated hardware accelerator (CCP). The CCP runs in constant time, has built-in side channel counter-measures, and generates true random data. |
| Secure coding practices | SNP firmware has extensive input validation and error handling. On init, the firmware validates system configuration against an allow-list, and verifies hardware settings are locked. |
| | SNP firmware is routinely audited using static analysis tools. In addition, using harnesses and memory sanitizers, the firmware is continuously being fuzzed. |
| Vulnerability recovery | Secure boot authentication key is fused, but can be revoked in case of compromise. |
| | ASP firmware has a concept of a *versioned* chip endorsement key (VCEK). It is protected using a *hash-sticks* mechanism such that older TCB versions do not have access to VCEKs of newer versions. |

# Summary

We walked through a complex security review process, discussed methods such as “invariant analysis” and layered cryptography reviews. We introduced unique tools such as *Wycheproof* for crypto tests, and the *PCIe screamer* for hardware tests. Lastly, we covered security vulnerabilities that emerged in this complex confidential computing technology.

The report highlights the challenges in building trusted execution environments that face a powerful adversary - the host operating system. Reviewing these systems for security is equally challenging, as we’re often dealing with closed source firmware and proprietary hardware components. We hope to encourage secure system designers to share more information about system design and implementation, since we believe this is the only way to truly audit and build trust in a system.

Despite the vulnerabilities listed above, we believe that AMD SNP firmware meets a high security bar. Firmware design mitigates several bug classes, and offers a way to recover from vulnerabilities. RMP based access control is a solid integrity protection, and confidential VMs are protected against a broad range of attacks.

# Acknowledgments

We are grateful for the open collaboration with AMD engineers, and wish to thank David Kaplan, Richard Relph and Nathan Nadarajah for their commitment to product security. We would also like to thank other AMD employees: Ab Nacef, Prabhu Jayanna and Mark Papermaster for their support of this joint effort.