Analysis of Microsoft CVE-2022-21907

fortinet.com/blog/threat-research/analysis-of-microsoft-cve-2022-21907

February 15, 2022

Threat Research

By Tim Lau | February 15, 2022

On January 11th, 2022 Microsoft released a patch for CVE-2022-21907 as part of Microsoft's Patch Tuesday. CVE-2022-21907 attracted special attentions from industry insiders due to the claim that the vulnerability is worm-able. In this analysis we will look at the cause of the vulnerability and how attackers can exploit it.

Affected Platforms: Windows Server 2022, Windows Server 2019, Windows 10
Impacted Users: Any organization with affected Windows system
Impact: Denial of service to affected systems
Severity Level: High

CVE-2022-21907 is a remote code execution vulnerability in Windows' Internet Information Services (IIS) component. More specifically, it affects the kernel module inside http.sys that handles most of the IIS core operations. At a minimum, the vulnerability can lead to denial of service conditions on the victim's machine by crashing the operating system. It might also be possible to combine this vulnerability with another vulnerability to enable remote code execution.

We used Windows 2022 Server 10.0.20348.143 as the base of our analysis. IIS is also present on Windows 10. We also looked at the Windows 10 (2H 2021) http.sys and confirmed that the same vulnerable code path exists. However, since IIS is not enabled by default on Windows 10, the chance of Windows 10 systems being exploited is significantly less.

First, we performed a binary differential between the vulnerable http.sys and the patched http.sys (10.0.20348.469). The program Bindiff compared the two binary files and highlighted the functions that have been modified. While a few functions were heavily modified, we were interested in two particular functions—http!UlpAllocateFastTracker() and http!UlFastSendHttpResponse() .

(As an aside, we did our initial analysis on Windows 10 http.sys, and these two functions are the only ones patched on Windows 10.)

In http!UlpAllocateFastTracker(), we see the following differences:

Figure 1: Differences between the original (top) and the patched function (bottom)

One curious thing to note is that memset() is called twice to zero out the buffer: once for a hardcoded first 0x1e0 bytes of the buffer, and the other starting at 0x2e for 0x50 bytes.

Figure 2: memset(value = 0, size=0x1e0)

Figure 3: memset(value = 0, size=0x50)

The difference between these two memset() calls is that memset(0x1e0) is for a freshly allocated buffer from nt!ExAllocatePool3(), and memset(0x50) is for buffers from both nt!ExAllocatePool3() and ExpInterlockedPopEntrySList(). (Internally, it uses a Windows single-linked list structure LIST_ENTRY, basically reusing previously allocated buffers.)

If we only look at the modifications to http!UlpAllocateFastTracker(), we can deduce that the non-zero entries in the buffer (let's call it a Tracker) might cause some unpleasant side effects. Furthermore, since the buffer can be allocated directly from nt!ExAllocatePool3(), if the system is experiencing memory pressure it's possible to spray attacker-controlled data into the memory and have the attacker-controlled data show up in the newly-minted Tracker buffer.

The fact that memset() is called twice to the same Tracker buffer is also curious. Apparently, the developer felt that it was necessary to zero-out a particular segment of the Tracker (from 0x2E-0x7E), even if the buffer was retrieved from a LookAside link list (where the initial allocation would have already zeroed out all the attacker-controlled data). This means that whatever non-zero value that triggers the bug, it should be inside the 0x2e-0x7e part of the Tracker structure.

At this point, we have a few ideas we can try. First, we need to know under what conditions http!UlpAllocateFastTracker() would be called. This turns out to be very easy to determine. A single command in Ghidra (Find References to UlpAllocateFastTracker) or IDA (Jump to xref) both show that only one function in http.sys could call UlpAllocateFastTracker().

Figure 4: Two calls to http!UlpAllocateFastTacker()

After writing some python scripts blasting HTTP requests to IIS, we determined that http!UlFastSendHttpResponse() does what its name suggests—the function is responsible for sending an http response back to the client. The Tracker object we saw earlier is a structure that keeps track of various states and pointers related to that response. When we snoop around, we can even find the response data in one of the pointers.

Figure 5: A Tracker object and the corresponding response data

After we determined how to access the initialization code, we decided to 'help' the exploit by pre-writing a non-zero value to Tracker. Using Windbg, pykd and some python scripting, we managed to inject pre-determined values into the part of Tracker that are likely to be affected before http!UlpAllocateFastTracker() returns.

Sadly, no matter how much we ran our 'fuzzer', the test system remained stable and responsive. We did, however, noticed that most of the calls to http!UlpAllocateFastTracker() were from UlFastSendHttpResponse+0x2F0 (>90%), and only a few allocation calls were made from UlpAllocateFastTracker+0xe99. We did a bindiff on http!UlFastSendHttpResponse() on Windows 10's http.sys and there's a gigantic code change.

Figure 6: Comparison between the patched http!UlFastSendHttpResponse() and the original http!UlFastSendHttpResponse()

At this point we were unable to trigger the crash so we took to Twitter to look for a POC.

The Crash

Armed with a new PoC, we resumed our analysis (this time on the Windows 2022 Server) and we soon discovered the cause of the crash that was being patched.

The crash happens at the end of the clean up phrase of http!UlFastSendHttpResponse(), with a call to nt!MmUnampLockedPages() that tries to access invalid memory.

Figure 7: A stack trace of the crash

According to Microsoft, nt!MmUnmapLockedPages() is a Windows kernel routine that releases a mapping between a virtual memory address and a physical memory address. The mapping is described by a kernel structure called the Memory Descriptor List (MDL).

Figure 8: Function signature of MmUnmapLockedPages()

When we set a breakpoint on the call to nt!MmUnmapLockedPages() we started to see all sorts of invalid memory addresses being passed in as the BaseAddress (virtual memory address).

Figure 9: Invalid arguments for MmUnmapLockedPages()

But now the question became, what did the PoC do differently to trigger the vulnerability? To get to the bottom of this, we needed to decompile http!UlFastSendHttpResponse() and look at the code.

Figure 10: Decompiled vulnerable code

We were able to immediately make some guesses. We could see that the pointer v19 is freed by UlpFreeFastTracker(). This told us that v19 is the pointer to the Tracker buffer. Indeed, when we scrolled up and checked the two calls to http!UlpAllocateFastTracker(), we could see that v19 is the return value from that function.

Figure 11: v19 as the returned Tracker object

At the same time, we knew the second argument for MmUnmapLockedPages() is the MDL struct. If we check the definition of MDL (http.sys uses an internal struct definition of MDL, but it's the same as the kernel's), we could see that the 0x00a field is the MdlFlags, and that the routine checked to see if the flag's 0th bit is 1. Finally, the 0x018 (24 in decimal) field is the MappedSystemVa.

Figure 12: MDL definition

As an aside, according to wdm.h from Windows SDK, 0x0001 is MDL_MAPPED_TO_SYSTEM_VA, ie. the memory mapping described by this MDL is valid.

Figure 13: MdlFlags bit field definition from Windows SDK

With these two pieces of information, we were able to construct the pseudo-code-

Figure 14: Pseudo-code of vulnerable code. The analysis will call Tracker->80 as 'some_mdl' from now on.

So, looking back at our initial guess, it was pretty good. We guessed that Tracker's 0x2e-0x7e needed to be non-zero, and indeed, the 0x50 pointer does have to be non-zero for this if statement to go through.

So now, we have four new questions, ranked from the most to the least obvious:

1. Can we control the 'some_mdl' MDL struct data?
2. What is Tracker->member_0x50?
3. Why can the PoC reach this code when our driver couldn't?
4. Is remote code execution possible?

Can we control the 'some_mdl' MDL struct data?

It turns out that, yes, the attacker does have control of the bytes in the MDL in certain situations! We went back to http!UlpAllocateFastTracker() and stepped through every single line of instruction, and while there are multiple MDL pointers in Tracker, the MDL at offset 0x80 is never initialized. The allocation routine simply picks sequential memory spaces after Tracker's struct location and has the Tracker's MDL pointers point to these addresses. This makes sense, as when nt!ExAllocatePool3() is called, the bytes requested are much larger than the deduced size of the Tracker struct (Remember that the patched memset() only writes 0s to the first 0x1e0 of the buffer.)

Figure 15: nt!ExAllocatePool3() allocated 0xc85 bytes of buffer

Figure 16: http!UlInitializeFastTrackerPool() assigning addresses to Tracker pointers

We know that 0x68, 0x70, 0x88, and 0x80 are all MDL pointers (via IDA heuristics), but only 0x68 is initialized with MmBuildMdlForNonPagedPool() in the initialization routine.

Figure 17: Tracker->0x68 pointer being initialized as a valid MDL object

Once control is returned to http!UlFastSendHttpResponse(), additional MDLs are eventually initialized. However, the PoC discovered a code path where the initialization is skipped, with disastrous consequences.

What is Tracker->member_0x50?

We tried to dive deeper into the code to figure out what member_0x50 does, but the object is created outside of http!UlFastSendHttpResponse(), and was passed-by-reference to the routine as a pointer argument.

Since the code assumes that if member_0x50 is valid then some_mdl should also be valid, perhaps the MDL's memory range is the backing storage for member_0x50, and both elements should be valid (or be invalid) together.

During our testing, however, the argument that leads to member_0x50 is always null with both our driver and the PoC. We decided to leave it at that.

Why can the PoC reach this code when our driver couldn't?

As mentioned before, we need the execution to take a path that would not initialize some_mdl. The PoC takes advantage of this by sending identical malformed HTTP packets in quick succession.

There are two calls to http!UlpAllocateFastTracker(). The first call is used more than 90% of the time. Once the Tracker structure is allocated, http!UlFastSendHttpResponse() takes over and continues the struct initialization process. Most importantly, during the process the member_0x50 element is zero-ed out, thus ensuring the bug would never be executed during normal execution.

Figure 18: member_0x50 pointer is zeroed out

However, when IIS receives multiple malformed packets in quick succession, a different code path is taken. According to our code analysis, IIS eventually abandons its (our guess) caching mechanisms. In particular, while the first call to http!UlpAllocateFastTracker() still happens, the allocated Tracker structure is quickly deallocated.

We are not sure why, but during the first call, a single value at Tracker->0x148 changes from 0x14 (in previous calls) to 0x0, which causes the new Tracker structure to be deallocated, and a call to http!UlpAllocateFastTracker() is made with a different second argument.

Figure 19: First call, 0x200

Figure 20: Second call, 0x0

(By the way, judging from the code and the PDB (program database), two other variables "UlH3ExtraHeaderCount" and "_UX_DUO_COLLECTION" are used to determine the value. If anyone knows what these variables do, please let us know.)

After the second allocation, a call is made to http!UlGenerateFixedHeaders(). However, the call is cut short. The sixth argument to http!UlGenerateFixedHeaders()—the same 0x0 (normally 0x200) variable as the allocation call, causes an early check fail, resulting in an error code 0xC000000D and a quick return.

Figure 21: The 0x0 value in the argument makes http!UlGenerateFixedHeaders() returns with an error

After the routine returns with the error code, the execution skips most of the http!UlFastSendHttpResponse() and goes straight to cleanup phrase. As part of the cleanup, the vulnerable call to nt!MmUnmapLockedPages() is made, and the system crashes.

As we said earlier, a malformed HTTP packet is needed to trigger this bug. We then wondered if other forms of malformed HTTP packets would also work. To our surprise, almost all our test samples would crash the system. This poses a serious issue as there are potentially many ways to attack the victim system.

Is remote code execution possible?

Since we have control of mapping any attacker-controlled memory, there is a risk of remote code execution. Constructing such a remote code execution, however, would require more research into what the Tracker fields do. The attacker would need to spray the memory with fake MDLs and fake Tracker pointers (this might require another vulnerability that leaks the kernel address info) or take advantage of the fact that there are other fields in Tracker that are also not initialized properly.

We tried to combine the PoC with our driver program to spray the kernel memory with attacker-controlled data. However, the probability of the sprayed content being reallocated again and showing up in the vulnerable code is rather low; A successful remote code execution chain might require a more accurate way to spray the memory.

Based on this analysis, we at FortiGuard have modified our IPS signatures to account for potential malicious traffic.

Conclusion

Due to the claim that the CVE is wormable, initially there was concerns that CVE-2022-21907 could potentially have a high impact. However, the combination that IIS is seldom enable on Windows 10, and the fact that the attacker does not have a direct way to create read or write primitives into kernel memory, lessen the risk somewhat.

Fortinet Protections

ForiGuard IPS protects against all known exploits associated with the CVE with the following signature:

MS.Windows.HTTP.Protocol.Stack.CVE-2022-21907.Code.Execution

However, due to the unpredictable nature of malformed HTTP packets, we strongly urge organizations to apply the corresponding patches as quickly as possible to avoid service disruption. FortiGuard Labs will continue to monitor the CVE and apply new countermeasures when necessary.

Appendix:

https://msrc.microsoft.com/update-guide/vulnerability/CVE-2022-21907

https://twitter.com/wdormann/status/1488148028317917186 https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-mmunmaplockedpages

Related Posts