Talos Vulnerability Report

TALOS-2022-1574

Hancom Office 2020 Hword Docx XML parsing heap underflow vulnerability

OCTOBER 4, 2022

CVE NUMBER

CVE-2022-33896

SUMMARY

A buffer underflow vulnerability exists in the way Hword of Hancom Office 2020 version 11.0.0.5357 parses XML-based office files. A specially-crafted malformed file can cause memory corruption by using memory before buffer start, which can lead to code execution. A victim would need to access a malicious file to trigger this vulnerability.

CONFIRMED VULNERABLE VERSIONS

The versions below were either tested or verified to be vulnerable by Talos or confirmed to be vulnerable by the vendor.

Hancom Office 2020 Hancom Office 2020 11.0.0.5357

PRODUCT URLS

Hancom Office 2020 - https://office.hancom.com/

CVSSV3 SCORE

7.8 - CVSS:3.0/AV:L/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H

CWE

CWE-124 - Buffer Underwrite (‘Buffer Underflow’)

DETAILS

Hancom Office is considered one of the more popular Office suites used within South Korea.

After enabling PageHeap for the application and HwordApp.dll, attempting to open the malicious file results in the following exception in the debugger:

```
(150c.1f44): Access violation - code c0000005 (first/second chance not available)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=131d1490 ebx=00ef9fa4 ecx=dcbaaaaa edx=00000001 esi=00ef9f64 edi=00ef9ea0
eip=6a55d80c esp=00ef9dac ebp=00ef9de4 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00200246
HwordApp!SetInitFontCallbackFunc+0x4f619c:
6a55d80c 8b01            mov     eax,dword ptr [ecx]  ds:002b:dcbaaaaa=????????
```

A few assembly instructions prior, we see the following code (we assume the HwordApp.dll is loaded at the base address of `0x6a010000`):

```
.text:6A55D803 loc_6A55D803:
.text:6A55D803 mov     eax, [esi+0A30h]
.text:6A55D809 mov     ecx, [eax-4]                                    (1)
.text:6A55D80C mov     eax, [ecx]                                      (2)
.text:6A55D80E call    dword ptr [eax+10h]                             (3)
```

The pointer is decremented by 4 at location (1) and is dereferenced at (2) where the code eventually crashes.

At location (1), `eax` holds the pointer to memory allocated in the heap. Since we have enabled PageHeap, we can see metadata related to this heap allocation, like allocation size, the stack trace of the allocating code, etc. In memory, these metadata reside just before the heap allocation returned to the application. We can examine the metadata with the following command:

```
0:002> dt _DPH_BLOCK_INFORMATION eax-20
ntdll!_DPH_BLOCK_INFORMATION
   +0x000 StartStamp       : 0xabcdaaaa
   +0x004 Heap             : 0x84801000 Void
   +0x008 RequestedSize    : 4
   +0x00c ActualSize       : 0x2c
   +0x010 FreeQueue        : _LIST_ENTRY [ 0x48010c0 - 0x12bd4230 ]
   +0x010 FreePushList     : _SINGLE_LIST_ENTRY
   +0x010 TraceIndex       : 0x10c0
   +0x018 StackTrace       : 0x050f2284 Void
   +0x01c EndStamp         : 0xdcbaaaaa
```

The code attempts to dereference a pointer with the erroneous value of `0xdcbaaaaa`, which is the `EndStamp` magic value at the end of PageHeap metadata.

As evident in the `RequestedSize` field, the application requested an allocation of 4 bytes. However, the code effectively attempts to dereference a pointer that resides 4 bytes prior to that allocation.

After reversing the code, it was found that the application crashes in an attempt to parse XML data, specifically the `</w:p>` XML tag. The malicious file is in the .docx format, which is a .zip file containing various XML files. Unzipping the malicious file, we see the following in `word/document.xml`:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
...
<w:body></w:p></w:body>
```

The `</w:p>` XML tag denotes the end of a paragraph in the .docx format. Providing the open tag `<w:p>` before the end tag, the application does not crash. From our analysis, it was evident that the application keeps a vector-type data structure containing pointers of objects. When there is an open tag followed by a close tag, the vector holds two pointers with a total size of 8 bytes (2 pointers of 4 bytes each) and the code works correctly. When there is only the close tag, the code assumes there is another element in the vector, decrements the pointer to the vector by 4 and attempts to dereference and call the erroneous pointer.

At location (3) in the code listing above, we see this pointer being used as a target to an indirect call, in a typical behaviour to a virtual function table call in C++. As a result, the code will attempt to make a call to an erroneous pointer that resides in memory.

Using heap spraying techniques, an attacker could force the application to allocate memory that resides immediately before the buffer and make it contain arbitrary data. As a result, the attacker could be able to control the pointer that will eventually be used for the indirect call, resulting in arbitrary code execution.

TIMELINE

2022-07-14 - Vendor Disclosure
2022-09-30 - Vendor Patch Release
2022-10-04 - Public Release

CREDIT

Discovered by Marcin 'Icewall' Noga and Dimitrios Tatsis of Cisco Talos.

VULNERABILITY REPORTS

PREVIOUS REPORT
TALOS-2022-1517

NEXT REPORT
TALOS-2022-1587