Notes on Windows Uniscribe Fuzzing

googleprojectzero.blogspot.com/2017/04/notes-on-windows-uniscribe-fuzzing.html

Posted by Mateusz Jurczyk of Google Project Zero

Among the total of 119 vulnerabilities with CVEs fixed by Microsoft in the March Patch Tuesday a few weeks ago, there were 29 bugs reported by us in the font-handling code of the Uniscribe library. Admittedly the subject of font-related security has already been extensively discussed on this blog both in the context of manual analysis [1][2] and fuzzing [3][4]. However, what makes this effort a bit different from the previous ones is the fact that Uniscribe is a little-known user-mode component, which had not been widely recognized as a viable attack vector before, as opposed to the kernel-mode font implementations included in the win32k.sys and ATMFD.DLL drivers. In this post, we outline a brief history and description of Uniscribe, explain how we approached at-scale fuzzing of the library, and highlight some of the more interesting discoveries we have made so far. All the raw reports of the bugs we're referring to (as they were submitted to Microsoft), together with the corresponding proof-of-concept samples, can be found in the official Project Zero bug tracker [5]. Enjoy!

Introduction

It was November 2016 when we started yet another iteration of our Windows font fuzzing job (whose architecture was thoroughly described in [4]). At that point, the kernel attack surface was mostly fuzz-clean with regards to the techniques we were using, but we still like to play with the configuration and input corpus from time to time to see if we can squeeze out any more bugs with the existing infrastructure. What we ended up with a several days later were a bunch of samples which supposedly crashed the guest Windows system running inside of Bochs. When we fed them to our reproduction pipeline, none of the bugchecks occurred again for unclear reasons. As disappointing as that was, there also was one interesting and unexpected result: for one of the test cases, the user-mode harness crashed itself, without bringing the whole OS down at the same time. This could indicate either that there was a bug in our code, or that there was some unanticipated font parsing going on in ring-3. When we started digging deeper, we found out that the unhandled exception took place in the following context:

```
(4464.11b4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0933d8bf ebx=00000000 ecx=09340ffc edx=00001b9f esi=0026ecac edi=00000009
eip=752378f3 esp=0026ec24 ebp=0026ec2c iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
USP10!ScriptPositionSingleGlyph+0x28533:
752378f3 668b4c5002      mov     cx,word ptr [eax+edx*2+2] ds:002b:09340fff=????
```

Until that moment, we didn't fully realize that our tools were triggering any font-handling code beyond the well-known kernel implementation (despite some related bugs having been publicly fixed in the past, e.g. CVE-2016-7274 [6]). As a result, the fuzzing system was not prepared to catch user-mode faults, and thus any such crashes had remained completely undetected in favor of system bugchecks, which caused full machine restarts.

We quickly determined that the usp10.dll library corresponded to “Uniscribe Unicode script processor” (in Microsoft’s own words) [7]. It is a relatively large module (600-800 kB depending on system version and bitness) responsible for rendering Unicode-encoded text, as the name suggests. From a security perspective, it’s important that the code base dates back to Windows 2000, and includes a C++ implementation of the parsing of various complex TrueType/OpenType structures, in addition to what is already implemented in the kernel. The specific tables that Uniscribe touches on are primarily Advanced Typography Tables (“GDEF”, “GSUB”, “GPOS”, “BASE”, “JSTF”), but also “OS/2”, “cmap” and “maxp” to some extent. What’s equally significant is that the code can be reached simply by calling the DrawText [8] or other equivalent API with Unicode-encoded text and an attacker-controlled font. Since no special calls other than the typical ones are necessary to execute the most exposed areas of the library, it makes for a great attack vector in applications which use GDI to render text with fonts originating from untrusted sources. This is also evidenced by the stack trace of the original crash, and the fact that it occurred in a program which didn’t include any usp10-specific code:

```
0:000> kb
ChildEBP RetAddr
0026ec2c 09340ffc USP10!otlChainRuleSetTable::rule+0x13
0026eccc 0133d7d2 USP10!otlChainingLookup::apply+0x7d3
0026ed48 0026f09c USP10!ApplyLookup+0x261
0026ef4c 0026f078 USP10!ApplyFeatures+0x481
0026ef98 09342f40 USP10!SubstituteOtlGlyphs+0x1bf
0026efd4 0026f0b4 USP10!SubstituteOtlChars+0x220
0026f250 0026f370 USP10!HebrewEngineGetGlyphs+0x690
0026f310 0026f370 USP10!ShapingGetGlyphs+0x36a
0026f3fc 09316318 USP10!ShlShape+0x2ef
0026f440 09316318 USP10!ScriptShape+0x15f
0026f4a0 0026f520 USP10!RenderItemNoFallback+0xfa
0026f4cc 0026f520 USP10!RenderItemWithFallback+0x104
0026f4f0 09316124 USP10!RenderItem+0x22
0026f534 2d011da2 USP10!ScriptStringAnalyzeGlyphs+0x1e9
0026f54c 0000000a USP10!ScriptStringAnalyse+0x284
0026f598 0000000a LPK!LpkStringAnalyse+0xe5
0026f694 00000000 LPK!LpkCharsetDraw+0x332
0026f6c8 00000000 LPK!LpkDrawTextEx+0x40
0026f708 00000000 USER32!DT_DrawStr+0x13c
0026f754 0026fa30 USER32!DT_GetLineBreak+0x78
0026f800 0000000a USER32!DrawTextExWorker+0x255
0026f824 ffffffff USER32!DrawTextExW+0x1e
```

As can be seen here, the Uniscribe functionality was invoked internally by user32.dll through the lpk.dll (Language Pack) library. As soon as we learned about this new attack vector, we jumped at the first chance to fuzz it. Most of the infrastructure was already in place, since both user- and kernel-mode font fuzzing share a large number of the pieces. The extra work that we had to do was mostly related to filtering the input corpus, fiddling with the mutator configuration, adjusting the system configuration and implementing logic for the detection of user-mode crashes (both in the test harness and Bochs instrumentation). All of these steps are discussed in detail below. After a few days, we had everything working as planned, and after another couple, there were already over 80 crashes at unique addresses waiting for triage. Below is a summary of the issues that were found in the first fuzzing run and reported to Microsoft in December 2016.

Results at a glance

Since ~80 was still a fairly manageable number of crashes to triage manually, we tried to reproduce each of them by hand, deduplicating them and writing down their details at the same time. When we finished, we ended up with 8 separate high-severity issues that could potentially allow remote code execution:

Tracker ID	Memory access type at crash	Crashing function	CVE
1022	Invalid write of n bytes (memcpy)	usp10!otlList::insertAt	CVE-2017-0108
1023	Invalid read / write of 2 bytes	usp10!AssignGlyphTypes	CVE-2017-0084
1025	Invalid write of n bytes (memset)	usp10!otlCacheManager::GlyphsSubstituted	CVE-2017-0086
1026	Invalid write of n bytes (memcpy)	usp10!MergeLigRecords	CVE-2017-0087
1027	Invalid write of 2 bytes	usp10!ttoGetTableData	CVE-2017-0088
1028	Invalid write of 2 bytes	usp10!UpdateGlyphFlags	CVE-2017-0089

1029	Invalid write of n bytes	usp10!BuildFSM and nearby functions	CVE-2017-0090
1030	Invalid write of n bytes	usp10!FillAlternatesList	CVE-2017-0072

All of the bugs but one were triggered through a standard DrawText call and resulted in heap memory corruption. The one exception was the #1030 issue, which resided in a documented Uniscribe-specific ScriptGetFontAlternateGlyphs API function. The routine is responsible for retrieving a list of alternate glyphs for a specified character, and the interesting fact about the bug is that it wasn't a problem with operating on any internal structures. Instead, the function failed to honor the value of the cMaxAlternates argument, and could therefore write more output data to the pAlternateGlyphs buffer than was allowed by the function caller. This meant that the buffer overflow was not specific to any particular memory type – depending on what pointer the client passed in, the overflow would take place on the stack, heap or static memory. The exploitability of such a bug would greatly depend on the program design and compilation options used to build it. We must admit, however, that it is unclear what the real-world clients of the function are, and whether any of them would meet the requirements to become a viable attack target.

Furthermore, we extracted 27 unique crashes caused by invalid memory reads from non-NULL addresses, which could potentially lead to information disclosure of secrets stored in the process address space. Due to the large volume of these crashes, we were unable to analyze each of them in much detail or perform any advanced deduplication. Instead, we partitioned them by the top-level exception address, and filed all of them as a single entry #1031 in the bug tracker:

1. usp10!otlMultiSubstLookup::apply+0xa8
2. usp10!otlSingleSubstLookup::applyToSingleGlyph+0x98
3. usp10!otlSingleSubstLookup::apply+0xa9
4. usp10!otlMultiSubstLookup::getCoverageTable+0x2c
5. usp10!otlMark2Array::mark2Anchor+0x18
6. usp10!GetSubstGlyph+0x2e
7. usp10!BuildTableCache+0x1ca
8. usp10!otlMkMkPosLookup::apply+0x1b4
9. usp10!otlLookupTable::markFilteringSet+0x1a
10. usp10!otlSinglePosLookup::getCoverageTable+0x12
11. usp10!BuildTableCache+0x1e7
12. usp10!otlChainingLookup::getCoverageTable+0x15
13. usp10!otlReverseChainingLookup::getCoverageTable+0x15
14. usp10!otlLigCaretListTable::coverage+0x7
15. usp10!otlMultiSubstLookup::apply+0x99
16. usp10!otlTableCacheData::FindLookupList+0x9
17. usp10!ttoGetTableData+0x4b4

18. usp10!GetSubtableCoverage+0x1ab
19. usp10!otlChainingLookup::apply+0x2d
20. usp10!MergeLigRecords+0x132
21. usp10!otlLookupTable::subTable+0x23
22. usp10!GetMaxParameter+0x53
23. usp10!ApplyLookup+0xc3
24. usp10!ApplyLookupToSingleGlyph+0x6f
25. usp10!ttoGetTableData+0x19f6
26. usp10!otlExtensionLookup::extensionSubTable+0x1d
27. usp10!ttoGetTableData+0x1a77

In the end, it turned out that these 27 crashes manifested 21 actual bugs, which were fixed by Microsoft as CVE-2017-0083, CVE-2017-0091, CVE-2017-0092 and CVE-2017-0111 to CVE-2017-0128 in the MS17-011 security bulletin.

Lastly, we also reported 7 unique NULL pointer dereference issues with no deadline, with the hope that having any of them fixed would potentially enable our fuzzer to discover other, more severe bugs. On March 17th, MSRC responded that they investigated the cases and concluded that they were low-severity DoS problems only, and would not be fixed as part of a security bulletin in the near future.

Input corpus, mutation configuration and adjusting the test harness

Gathering a solid corpus of input samples is arguably one of the most important parts of fuzzing preparation, especially if code coverage feedback is not involved, making it impossible for the corpus to gradually evolve into a more optimal form. We were lucky enough to already have had several font corpora at our disposal from previous fuzzing runs. We decided to use the same set of files that had helped us discover 18 Windows kernel bugs in the past (see the “Preparing the input corpus” section of [4]). It was originally generated by running a corpus distillation algorithm over a large number of fonts crawled off the web, using an instrumented build of the FreeType2 open-source library, and consisted of 14848 TrueType and 4659 OpenType files, for a total of 2.4G of disk space. In order to tailor the corpus better for Uniscribe, we reduced it to just the files that contained at least one of the “GDEF”, “GSUB”, “GPOS”, “BASE” or “JSTF” tables, which are parsed by the library. This left us with 3768 TrueType and 2520 OpenType fonts consuming 1.68G on disk, which were much more likely to expose bugs in Uniscribe than any of the removed ones. That was the final corpus that we worked with.

The mutator configuration was also pretty similar to what we did for the kernel: we used the same five standard bitflipping, byteflipping, chunkspew, special ints and binary arithmetic algorithms with the precalculated per-table mutation ratio ranges. The only change made specifically for Uniscribe was to add mutations for the “BASE” and “JSTF” tables, which were previously not accounted for.

Last but not least, we extended the functionality of the guest fuzzing harness, responsible for invoking the tested font-related API (mostly displaying all of the font's glyphs at various point sizes, but also querying a number of properties etc.). While it was clear that some of the relevant code was executed automatically through user32!DrawText with no modifications required, we wanted to maximize the coverage of Uniscribe code as much possible. A full reference of all its externally available functions can be found on MSDN [9]. After skimming through the documentation, we added calls to ScriptCacheGetHeight, ScriptGetFontProperties, ScriptGetCMap, ScriptGetFontAlternateGlyphs, ScriptSubstituteSingleGlyph and ScriptFreeCache. This quickly proved to be a successful idea, as it allowed us to discover the aforementioned generic bug in ScriptGetFontAlternateGlyphs. Furthermore, we decided to remove invocations of the GetKerningPairs and GetGlyphOutline API functions, as their corresponding logic was located in the kernel, while our focus had now shifted strictly to user-mode. As such, they wouldn't lead to the discovery of any new bugs in Uniscribe, but would instead slow the overall fuzzing process down. Apart from these minor modifications, the core of the test harness remained unchanged.

By taking the measures listed above, we hoped that they were sufficient to trigger most of the low hanging fruit bugs. With this assumption, the only part left was to make sure that the crashes would be reliably caught and reported to the fuzzer. This subject is discussed in the next section.

Crash detection

The first step we took to detect Uniscribe crashes effectively was disabling Special Pools for win32k.sys and ATMFD.DLL (which caused unnecessary overhead for no gain in user-mode), while enabling the PageHeap option in Application Verifier for the harness process. This was done to improve our chances at detecting invalid memory accesses, and make reproduction and deduplication more reliable.

Thanks to the fact that the fuzz-tested code in usp10.dll executed in the same context as the rest of the harness logic, we didn't have to write a full-fledged Windows debugger to supervise another process. Instead, we just set up a top-level exception handler with the SetUnhandledExceptionFilter function, which then got called every time a fatal exception was generated in the process. The handler's job was to send out the state of the crashing CPU context (passed in through ExceptionInfo->ContextRecord) to the hypervisor (i.e. the Bochs instrumentation) through the "debug print" hypercall, and then actually report that the crash occurred at the specific address.

In the kernel font fuzzing scenario, crashes were detected by the Bochs instrumentation with the BX_INSTR_RESET instrumentation callback. This approach worked because the guest system was configured to automatically reboot on bugcheck, consequently triggering the bx_instr_reset handler. The easiest way to integrate this approach with user-mode fuzzing would be therefore to just add a ExitWindowsEx call in the epilogue of the exception handler, making everything work out of the box without even touching the existing Bochs instrumentation. However, the method would result in losing information about the crash location, making automated deduplication impossible. In order to address this problem, we introduced a new "crash encountered" hypercall, which received the address of the faulting instruction in the argument from the guest, and passed this information further down our scalable fuzzing infrastructure. Having the crashes grouped by the exception address right from the start saved us a ton of postprocessing time, and limited the number of test cases we had to look at to a bare minimum.

This is the end of a list of differences between the Windows kernel font fuzzing setup we've been using for nearly two years now, and an equivalent setup for user-mode fuzzing that we only built a few months ago, but has already proven very effective. Everything else has remained the same as described in the "font fuzzing techniques" article from last year [4].

Conclusions

It is a fascinating but dire realization that even for such a well known class of bug hunting targets as font parsing implementations, it is still possible to discover new attack vectors dating back to the previous century, having remained largely unaudited until now, and being as exposed as the interfaces we already know about. We believe that this is a great example of how gradually rising the bar for a variety of software can have much more impact than trying to kill every last bug in a narrow range of code. It is also illustrative of the fact that the time spent on thoroughly analyzing the attack surface and looking for little-known targets may turn out very fruitful, as the security community still doesn't have a full understanding of the attack vectors in every important data processing stack (such as the Windows font handling in this case).

This effort and its results show that fuzzing is a very universal technique, and most of its components can be easily reused from one target to another, especially within the scope of a single file format. Finally, it has proven that it is possible to fuzz not just the Windows kernel, but also regular user-mode code, regardless of the environment of the host system (which was Linux in our case). While the Bochs x86 emulator incurs a significant overhead as compared to native execution speed, it can often be scaled against to still achieve a net gain in the number of iterations per second. As an interesting fact, issues #993 (Windows kernel registry hive loading), #1042 (EMF+ processing in GDI+), #1052 and #1054 (color profile processing) fixed in the last Patch Tuesday were also found with fuzzing Windows on Bochs, but with slightly different input samples, test harnesses and mutation strategies. :)

References

1. The "One font vulnerability to rule them all" series starting with https://googleprojectzero.blogspot.com/2015/07/one-font-vulnerability-to-rule-them-all.html
2. https://googleprojectzero.blogspot.com/2015/09/enabling-qr-codes-in-internet-explorer.html
3. https://googleprojectzero.blogspot.com/2016/06/a-year-of-windows-kernel-font-fuzzing-1_27.html
4. https://googleprojectzero.blogspot.com/2016/07/a-year-of-windows-kernel-font-fuzzing-2.html
5. https://bugs.chromium.org/p/project-zero/issues/list?can=1&q=product%3Auniscribe+fixed%3A2017-mar-14

6. http://blogs.flexerasoftware.com/secunia-research/2016/12/microsoft_windows_loaduvstable_heap_based_buffer_overflow_vulnerability.html
7. https://msdn.microsoft.com/pl-pl/library/windows/desktop/dd374091(v=vs.85).aspx
8. https://msdn.microsoft.com/pl-pl/library/windows/desktop/dd162498%28v=vs.85%29.aspx
9. https://msdn.microsoft.com/pl-pl/library/windows/desktop/dd374093(v=vs.85).aspx