Project Zero

News and updates from the Project Zero team at Google

Friday, July 1, 2016

A year of Windows kernel font fuzzing #2: the techniques

Posted by Mateusz Jurczyk of Google Project Zero

In part #1 of the series (see here), we discussed the motivation and outcomes of our year long fuzzing effort against the Windows kernel font engine, followed by an analysis of two bug collisions with Keen Team and Hacking Team that ensued as a result of this work. While the bugs themselves are surely amusing, what we find even more interesting are the techniques and decisions we made to make the project as effective as it turned out to be. Although several methods we used were very specific to the particular target, we believe that most of the ideas are still applicable to a majority of fuzzing efforts, as we have developed them through years of experience not only limited to the Windows kernel. Enjoy!

Technical details

While fuzzing is in principle based on a very simple concept (*mutate input, run target, catch crashes, repeat*), it's also the kind of "easy to learn, hard to master" activity. It is composed of a number of different stages, and the final result is a product of the effectiveness of each of them. Thus, in this blog post, we would like to explain how we tried to optimize each part of the process based on our experience with fuzzing other software in the past.

Preparing the input corpus

When the term "fuzzing" is used in relation to the testing of native applications processing input files, it's typically in the context of mutational fuzzing. This means that an initial corpus of diverse and usually valid input files is prepared beforehand, and then particular samples are slightly mutated during the fuzzing process. This is to retain the general file structure expected by the tested program, but also to insert random changes not expected by the developers. Historically, this approach has proven to be very effective, as it allowed a large number of different program states to be reached at a low cost without having to generate all the required (often complex) structures from scratch. Even in coverage-guided fuzzing with an evolving corpus, it is still beneficial to use a high-quality initial data set, as it may save a ton of wall-clock and CPU time which otherwise would be wasted on the brute-force generation of basic constructs in the inputs.

Also in this case, mutational fuzzing was fundamental and has contributed to the discovery of most of the reported bugs. For reasons discussed later on, we decided not to use coverage guidance, which increased the importance of a good input corpus even more. Specifically, we were interested in the following file types:

- TrueType fonts (.TTF extension, handled by win32k.sys),
- OpenType fonts (.OTF extension, handled by ATMFD.DLL),
- Bitmap fonts (.FON, .FNT extensions, handled by win32k.sys),
- Type 1 fonts (.PFB, .PFM, .MMM extensions, handled by ATMFD.DLL).

However, we decided to eventually leave Type 1 out for several reasons:

- Windows requires at least two corresponding files (.PFB and .PFM) to load a single Type 1 font.
- The format is structurally very simple, with the most complex part being CharStrings, which were already thoroughly audited by ourselves.
- Most of the font handling logic is shared in ATMFD.DLL between both Type 1 and OpenType formats.

When it comes to Bitmap fonts, we did initially include them in the fuzzing, but soon discovered that the only crash being (constantly) hit was an unhandled divide-by-zero exception. Therefore, we decided to exclude it from future fuzzing, and instead focus on the TrueType and OpenType formats, which were most interesting anyway.

Since we didn't intend to perform Windows kernel code coverage measurements, we couldn't run the typical corpus distillation of a large number of fonts generated automatically by various tools or crawled off the web. However, thanks to the previous FreeType2 fuzzing, we had several corpora generated for that project. We picked the one with high coverage redundancy (up to 100 samples per a single basic block pair) and extracted just the TTF and OTF fonts, which resulted in a collection of 19507 files – 14848 TrueType and 4659 OpenType ones – consuming about 2.4G of disk space.

Using a corpus tailored for one piece of software for fuzzing another involves a risk that some features present in the latter are not covered by samples for the former, and with no coverage guidance, they could never be discovered on their own. In this case I accepted the trade-off, as the risk was partially mitigated by the redundancy in the corpus, and the fact that FreeType has probably the most comprehensive support for both TrueType and OpenType formats, which should cover most or all of the Windows font engine features. Lastly, using the readily available corpus saved us a lot of time, by enabling us to start the fuzzer as soon as the rest of the infrastructure was functional.

Mutating TTF & OTF

In our experience, an optimal choice of the mutation strategy is one of the most crucial parts of effective fuzzing, so it's usually worth spending some time adjusting the configuration, or even writing format-specific processing. Conveniently, both TrueType and OpenType formats share a common chunk structure called SFNT. In short, it's a file layout consisting of multiple tables identified by 4-byte tags, each structured in a documented way and serving a different purpose. The design is backwards compatible, but also trivial to extend with vendor-specific tables. Presently, there are a total of around 50 tables in existence, but only ~20 can be considered somewhat important, and even fewer are obligatory. Furthermore, they can also be categorized depending on whether they are TrueType-specific, OpenType-specific, or common between those two. Detailed information on the most fundamental tables can be found in Microsoft's OpenType specification.

The essential observation about SFNT tables is that they are all completely different. They differ in length, structure, importance, nature of stored data etc. With this in mind, it only seems reasonable to treat each of them individually rather than equally. However, nearly every publicly discussed font fuzzer approached the mutation step in the opposite way: by first mutating the TTF/OTF file as a whole (not considering any of its internal structure), and then fixing up the table checksums in the header, so Windows wouldn't immediately refuse to load them. In our opinion, this was vastly suboptimal.

In order to get some concrete numbers, we went through our font corpus to discover that on average, there were ~10 tables whose modification affected the success of a font's loading and displaying. Intuitively, a mutation strategy is most powerful when the resulting files are successfully processed by the tested software 50% of times, and likewise fail to parse the other 50% of times. This indicates that the test cases are *on the verge* of being valid, and shows that the configuration is neither too aggressive, nor too loose. In the case of SFNT fonts, if one of the tables gets damaged too severely, the entire file will fail to load, regardless of the contents of the other tables. If we denote the probability of each table succeeding to load with a specific mutation strategy as $P(table_i)$, then the probability for the whole file is $P(table_1 \cap table_2 \cap \ldots \cap table_n)$, or:

$$P(font) = \prod_{i=1}^{n} P(table_i)$$

or if we assume that the probabilities for all tables should be equal (we want to fuzz them uniformly):

$$P(font) = P(table)^n$$

If we insert the averaged out $n = 10$ and an ideal $P(font) = 0.5$, then:

$$P(table) = \sqrt[n]{P(font)} = \sqrt[10]{0.5} \approx 0.93$$

So, as a result, we wanted to configure the mutator such that for each table, its mutated form could still be successfully processed ~93% of the time (at least on average).

The dumb algorithms we chose to mutate the tables were **Bitflipping** (flips 1-4 consecutive bits in randomly chosen bytes), **Byteflipping** (replaces randomly chosen bytes with other values), **Chunkspew** (copies data over from one location in the input to another), **Special Ints** (inserts special, *magic* integers into the input) and **Add Sub Binary** (adds and subtracts random integers from binary data). Since each of the algorithms affected data in a different way, we had to determine the average most optimal mutation ratio for each table/algorithm pair.

To facilitate the task, we developed a program which loaded each file in the corpus, iterated through each table and algorithm, and determined the mutation ratio which would result in a ~93% font loading success rate through bisection. After processing all files, we averaged the numbers, set some generic mutation ratios for tables whose mutations did not appear to affect font loading (but they could potentially still be parsed somewhere), and ended up with the following table:

Table / mutation	Bitflipping	Byteflipping	Chunkspew	Special Ints	Add Sub Binary
`hmtx`	0.1	0.8	0.8	0.8	0.8
`maxp`	0.009766	0.078125	0.125	0.056641	0.0625
`OS/2`	0.1	0.2	0.4	0.2	0.4
`post`	0.004	0.06	0.2	0.15	0.03
`cvt`	0.1	0.1	0.1	0.1	0.1
`fpgm`	0.01	0.01	0.01	0.01	0.01
`glyf`	0.00008	0.00064	0.008	0.00064	0.00064
`prep`	0.01	0.01	0.01	0.01	0.01
`gasp`	0.1	0.1	0.1	0.1	0.1
`CFF`	0.00005	0.0001	0.001	0.0002	0.0001

EBDT	0.01	0.08	0.2	0.08	0.08
EBLC	0.001	0.001	0.001	0.001	0.001
EBSC	0.01	0.01	0.01	0.01	0.01
GDEF	0.01	0.01	0.01	0.01	0.01
GPOS	0.001	0.008	0.01	0.008	0.008
GSUB	0.01	0.08	0.01	0.08	0.08
hdmx	0.01	0.01	0.01	0.01	0.01
kern	0.01	0.01	0.01	0.01	0.01
LTSH	0.01	0.01	0.01	0.01	0.01
VDMX	0.01	0.01	0.01	0.01	0.01
vhea	0.1	0.1	0.1	0.1	0.1
vmtx	0.1	0.1	0.1	0.1	0.1
mort	0.01	0.01	0.01	0.01	0.01

Even with the accurate numbers listed above, they are just average results based on the entirety of the corpus, so using the exact same values for all fonts wouldn't work out great. This is especially true since the size and complexity of input samples varies heavily, ranging from 8 kilobytes to 10 megabytes. Consequently, data density and distribution in the tables vary as well, requiring individual mutation ratios to reach the desired parsing success rate. In order to address that and provide the fuzzer with even more freedom, instead of using a fixed ratio in every iteration, in each pass the fuzzer would choose a temporary ratio from a range of $[0, 2\times R]$, R being the calculated ideal rate.

With the above configuration and a trivial piece of code to disassemble, modify and reassemble SFNT files (written in C++ in our case), we were now able to mutate fonts in a *smart-dumb* way. While we were still just randomly flipping bits and employing similarly simple algorithms, we had some degree of control of how Windows would react to the average mutated font.

15 out of 16 vulnerabilities discovered during the effort were found with the above mutation configuration.

Generating TTF programs

Adding some "smart" logic into an otherwise completely dumb fuzzing process is a significant improvement, but it's important to be aware that it might still be insufficient to discover certain classes of font handling bugs. A large part of most TTF and OTF files, sometimes even a majority, is consumed by glyph outline programs written for dedicated TrueType / PostScript virtual machines. These virtual machines are quite fragile in the context of fuzzing, as they abort program execution as soon as an invalid instruction is encountered. Consequently, dumb algorithms operating on "black-box" bitstreams with no knowledge of their structure would exercise a very limited portion of the VM, as each program would be very likely to crash after executing just a few instructions.

This is not to say that *conventional* mutations can't be useful for finding VM bugs. Vulnerabilities which could be triggered by changing a single instruction to another, or flipping an instruction argument could very well be discovered that way, and actually were in the past. This is why we allowed mutations in the "prep", "fpgm" and "glyf" tables, which contained TrueType programs. In fact, a vulnerability in the "IUP" instruction handler was indeed discovered with a dumb fuzzer.

However, we suspected that there might have also been bugs which required specific, long constructs of valid instructions in order to manifest themselves. The suspicion was also supported by the security review of CharString processing code in ATMFD, where most crashes could only be provoked with three or more concrete instructions. Such constructs are extremely unlikely (to the point of being unrealistic within reasonable time) to be generated by a dumb algorithm, especially with missing coverage guidance. As I had already spent weeks auditing the virtual machine implemented in ATMFD, there was no reason to fuzz-test it further. The problem only had to be addressed for TTF, and our idea to do it was to use a dedicated TrueType program generator, which would output streams of structurally valid instructions (with random arguments) and embed them into existing, legitimate TTF fonts. Below, we proceed to describe how we managed to achieve that effect.

SFNT formatted fonts (both TTF and OTF) can be dissected into human-readable .ttx files (XML-structured) with the ttx.py utility as part of the FontTools project, and reassembled back to binary form. This is greatly useful for convenient inspection of internal font structures and modifying them in any desired way. The project also understands TrueType instructions, and is able to disassemble and assemble back whole programs written in that language. The instruction streams are located within `<assembly></assembly>` tags, as shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<ttFont sfntVersion="\x00\x01\x00\x00" ttLibVersion="2.4">
…
 <glyf>
…
   <TTGlyph name=".notdef" xMin="128" yMin="0" xMax="896" yMax="1638">
…
     <instructions><assembly>
       PUSH[ ]  /* 16 values pushed */
       6 111 2 9 7 111 1 8 5 115 3 3 8 6 115 1
       SVTCA[0]
       MDAP[1]
```

```
        MIRP[01101]
        SRP0[ ]
…
      </assembly></instructions>
    </TTGlyph>
…
  </glyf>
…
</ttFont>
```

What we started with was a bit of pre-processing: we converted all .ttf files into their corresponding .ttx versions, and wrote a short Python script (using `xml.etree.ElementTree`) to remove the body of all `<assembly>` tags. Embedding generated programs into existing, legitimate fonts is largely superior to inserting them into a single, minimalistic TTF template. The TrueType instructions are tightly connected to other characteristics of their *hosting* fonts (points, contours, other settings), so having a diverse set of them (similarly to mutational fuzzing) couldn't hurt us, but could only help.

The next step was to write the actual generator, which would both generate the instructions and embed them in the right places of the .ttx files. Most of the work involved reading through the TrueType instruction set specification, and implementing a generator of each instruction individually. Some of the instructions (such as RTHG, ROFF or FLIPON) were trivial to handle, while others were much more complex. Overall, in order to make sure the programs were correct and wouldn't bail out very quickly, we also had to add processing of other portions of the font:

- Count the number of contours and points in each glyph, in order to be able to generate their valid indexes as instruction arguments.
- Follow the current zone pointers (ZP0, ZP1, ZP2) set by the SZP0, SZP1, SZP2, SZPS instructions, also to be able to generate valid contour and point indexes.
- Expand the "CVT" table to 32768 elements with random values.
- Adjust many fields in the "maxp" table for our convenience, so that we wouldn't hit any limits arbitrarily set by the hosting font.

After a few days worth of development and testing, we ended up with a `tt_generate.py` script, which took .ttx files as input, and created other .ttx files with all structures properly adjusted and containing long streams of mostly correct instructions for each glyph on output. After recompilation with ttx.py, the font could be loaded and displayed similarly to test cases created through dumb fuzzing.

The first vulnerability the generator found was obviously the "IUP" bug, as it only required setting ZP2 to 0 through the SZP2/SZPS instructions, and then invoking the IUP instruction. As this sequence was generated very frequently, we had to actively avoid it in the code (by always setting ZP2 to 1 before emitting IUP) before a patch for the issue was released. Then, the generator found one more unique bug (this time missed by the dumb approach) in the 2nd iteration of the fuzzing (issue #507): a pool-based buffer overflow in text drawing functions (`win32k!or_all_N_wide_rotated_need_last` and other related routines).

With one unique, serious bug and one collision with the result of mutational fuzzing, we still believe the effort put into writing the TTF generator was worthwhile. :-)

Kernel fuzzing in Bochs

One of the fundamental assumptions of the project was to make the fuzzing scalable, preferably on Linux systems with no virtualization support available. This premise considerably limited our choice of infrastructure, since we are talking about fuzzing the Windows kernel, which must run as a whole, but also requires some hypervisor above it to be able to detect crashes, save offending test cases and so forth. The qemu project sounded like an intuitive idea, but anyone familiar with our Bochspwn research knows that we greatly enjoy the Bochs x86 emulator and its instrumentation support. Systems running in full emulation mode are very slow (up to ~100 MHz effective frequency), but with potentially thousands of machines, the 20-50x slowdown is still easily scaled against. Furthermore, we already had extensive experience in running Windows on Bochs, so we could start implementing the fuzzing logic in the instrumentation without worrying about any unexpected entry-level problems.

In order to save cycles inside the guest and speeds things up, we decided that the font mutation/generation would take place outside of Windows, within the Bochs instrumentation. The only thing that the emulated system would be responsible for is requesting input data from the hypervisor, loading it in the system and displaying all glyphs in various styles and point sizes. The entire guest-side logic was included in a single *harness.exe* program, which automatically started at boot time and began testing fonts as soon as possible.

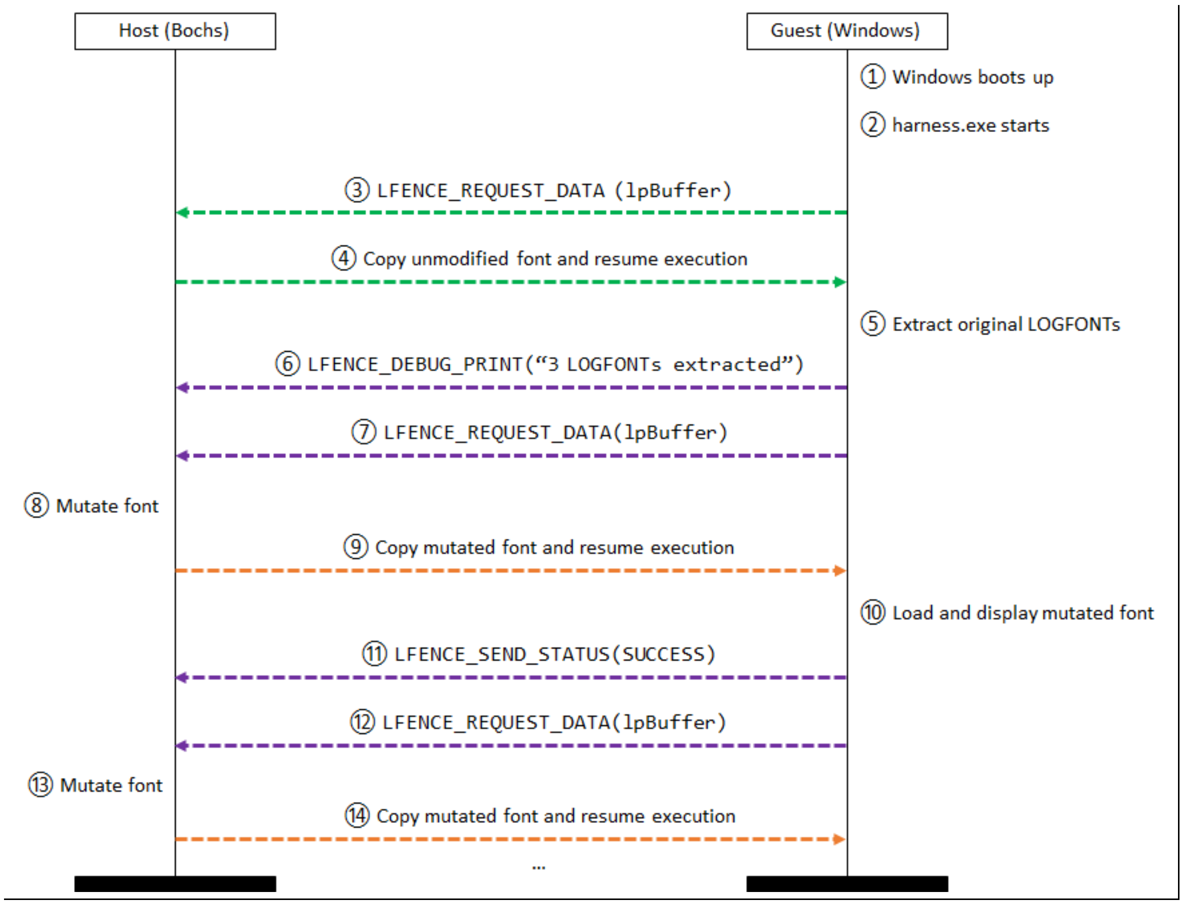
The above design required some channel of communication between the harness and Bochs instrumentation. How? If we realize that we're running an emulator, and thus control every part of the virtual CPU's execution, the answer becomes obvious: by detecting some special state which never occurs during normal execution, but can be easily set by our process inside the VM. In our case, we decided to use the LFENCE x86 instruction as a communication channel between the two environments. The opcode is effectively a no-op and is (almost) never used during regular system operation, making it a perfect candidate for this feature. Its execution can be detected with the `BX_INSTR_AFTER_EXECUTION` callback:

```
void bx_instr_after_execution(unsigned cpu, bxInstruction_c *i);
```

In our design, the operation code was passed through the EAX register, and input/output parameters through the ECX/EDX registers. We implemented a total of three operations facilitating the copying of font data to the guest, informing the host about the success or failure of its loading, and sending textual debug messages:

- `REQUEST_DATA(lpBuffer)` – sent by the harness to receive fresh font data to test the kernel against. The lpBuffer argument contains a virtual address of the buffer where data should be copied by the hypervisor.
- `SEND_STATUS(dwStatus)` – sent by the harness to inform the host if loading the most recent font succeeded or not.
- `DBG_PRINT(lpMessage)` – sent by the harness to log a textual message.

In addition, the data received in response to the first `REQUEST_DATA` message was always the original, non-mutated data of the sample file. This was to make it possible for the harness to extract valid `LOGFONT` structures describing fonts contained within the file, which were then used to load and display them in all subsequent iterations. The workflow is illustrated in the diagram below:


Host (Bochs)
Guest (Windows)
① Windows boots up
② harness.exe starts
③ LFENCE_REQUEST_DATA (lpBuffer)
④ Copy unmodified font and resume execution
⑤ Extract original LOGFONTs
⑥ LFENCE_DEBUG_PRINT("3 LOGFONTs extracted")
⑦ LFENCE_REQUEST_DATA(lpBuffer)
⑧ Mutate font
⑨ Copy mutated font and resume execution
⑩ Load and display mutated font
⑪ LFENCE_SEND_STATUS(SUCCESS)
⑫ LFENCE_REQUEST_DATA(lpBuffer)
⑬ Mutate font
⑭ Copy mutated font and resume execution
...

As you can see, the scheme is very simple. Sending the debug messages and status information made it possible to aggregate runtime information from the fuzzing sessions, and occasionally make sure that they were running correctly, and the goal of an average font loading ratio of 50% was really met. The font mutation/generation step implemented in the instrumentation was of course synchronous, meaning that the execution of the LFENCE instruction in the guest would not return before the sample data was copied to memory. The mutation stage was rather fast, as it was written in C++ and executed in the same process. The TTF program generation variant was much slower, as it involved starting an out-of-process Python generator, and then the ttx.py script to assemble the font back to binary form. However, compared to the time period of a single font test inside the guest (ranging from 0.5s to several minutes), it was still a neglectable overhead.

Another interesting point is that while copying the font data into the user-mode process memory inside of the VM was generally quite simple (using a `write_lin_mem` function similar to Bochspwn's `read_lin_mem`, see the implementation in mem_interface.cc), we had to make sure that the virtual address passed to the hypervisor had a corresponding physical memory mapping (i.e. was committed and not paged out). Memory allocated in Windows userland through the standard allocators or VirtualAlloc are by default pageable, which can be addressed by either using the VirtualLock function, or simply writing data to the allocated memory area, which also ensures that it will remain mapped in the near future.

With this design, we could now scale the Windows kernel font fuzzing to any number of machines, regardless of the host operating system (as long as it could run Bochs) and whether hardware virtualization support was available or not. Again, the only significant problem was the speed of the guest systems, but with the resources at our disposal, it was easy to balance the emulator's overhead and go far beyond the abilities of a single high-performance PC. We also undertook a number of steps to optimize the tested Windows installation as much as possible (both in terms of size and background execution), which is detailed in one of the next sections.

The client harness

Next to mutating and generating fonts in a way that makes it most likely for the tested software to crash, exercising the input-processing code paths in an exhaustive manner is probably the most crucial part of fuzzing. Even if we have the smartest mutators, most effective error detection mechanisms and millions of machines to use, automated testing won't accomplish much if it never executes the vulnerable code. Furthermore, even if the code does execute but only touches 1% of the input data in the process, our chances of triggering a crash are vastly (and unnecessarily) limited. This is why we believed that much care had to be put into implementing the client harness, in order to ensure that all font-related attack surfaces would be tested against all data included in each input font. The approach was a complete opposite of some other fuzzing efforts in the past, whose entire font testing consisted of firing up the default *Windows Font Viewer* program against malformed fonts, or only using it to display characters within the printable ASCII range, with the default style and point size.

To start with, every file can contain multiple fonts, whose number is returned by a successful call to the AddFontResource function. In order to enumerate them in a way they can be distinguished by the operating system, we used an undocumented GetFontResourceInfoW function (exported by gdi32.dll) with the `QFR_LOGFONT` parameter. The function operates on files saved to disk, and with the specific argument used, it returns an array of `LOGFONT` structures, each describing one embedded font. This is why the initial *dry run* iteration was necessary: to extract the descriptors from the original, unmodified font file (guaranteeing consistency of the information).

The dry run was also the only time the harness operated on the file system. For the actual loading of the fonts for testing them, we called the AddFontMemResourceEx function, which installs fonts directly from memory instead of files, which in this case resulted in an enormous performance improvement. The `LOGFONT` structures were passed directly as parameter of the CreateFontIndirect function, which created GDI font objects and returned handles of type `HFONT`. The handles were then passed to SelectObject, setting the font as the current one in the specified *Device Context*. In our harness, we created five font objects per each extracted `LOGFONT`: one with their original contents, and four with the *height*, *width*, *italic*, *underline* and *strikeout* properties chosen semi-randomly (with a constant srand() seed).

For each such font object, we wanted to display all supported glyphs. As we quickly found out, the GetFontUnicodeRanges function was designed exactly for this task: listing out the unicode ranges of characters which have corresponding outlines in the font. With this information, we called the DrawText API for every 10 subsequent characters. In addition, we also called the GetKerningPairs function once per font, and GetGlyphOutline with various parameters for every glyph. Overall, the harness was very fast – all operations were performed in-process, with hardly any interaction with the system other than through the syscalls operating on the tested fonts and related graphical objects. On the other hand, we believe that it managed to exercise most of the relevant attack surface, which is somewhat confirmed by the fact that the 16 crashes were triggered through four different system calls: `NtGdiGetTextExtentExW`, `NtGdiAddFontResourceW`, `NtGdiGetCharABCWidthsW` and `NtGdiGetFontUnicodeRanges`.

Optimizing the system and detecting bugchecks

Considering that the entire operating system was running in full software x86 emulation mode, which incurred an enormous overhead, every single emulated CPU cycle was extremely precious. Consequently, we wanted to make sure that as few irrelevant instructions were executed as possible, to make some breathing room for our harness, and most importantly the kernel drivers performing the actual font processing. Furthermore, another goal was to reduce the hard drive image with the system to the smallest possible size, so that it could be quickly distributed to all machines running the fuzzer.

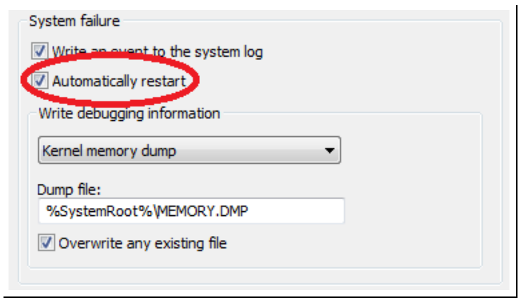
We undertook a number of actions to optimize the speed and size of the system:

- Changed the graphical theme to Classic.
- Disabled all services which were not absolutely necessary for the system to work (only left a few critical ones).
- Set the boot mode to Minimal/Safe Boot with VGA, so that only the core kernel-mode drivers would be loaded.
- Uninstalled all default Windows components (games, web browser etc.).
- Set the "Adjust for best performance" option in System Properties.
- Changed the default shell in registry from explorer.exe to the fuzzing harness, so that Explorer would never start in the first place.
- Removed all items from Autostart.
- Disabled disk indexing.
- Disabled paging.
- Disabled hibernation.
- Removed most unnecessary files and libraries from the C:\Windows directory.

With all of the above changes, we managed to get the disk image size down to around 7 gigabytes, and have it start in Bochs in under five minutes. Of course, when preparing images for the 2nd and 3rd iterations, many of the changes had to be reverted in order to update the operating system, and then applied back.

Interestingly, one of our optimization changes affected the reproducibility of issue #684. While the bug was reported to Microsoft on the same day as other results of the 3rd iteration, after two weeks the vendor informed us that they were unable to get a repro. Following some back-and-forth communication, we took a closer look into the possible configuration differences between the default installation and our testing environment. We then quickly realized that as part of setting the "Adjust for best performance" option in System Properties, the "Smooth edges of screen fonts" setting was unchecked, while it is enabled by default. It turned out that indeed, the vulnerability only manifested with the option disabled. When Microsoft confirmed they could now reproduce the problem, we updated the *Reported* date in the tracker, and the vendor fixed the bug within the new 90-day window.

When it comes to detecting system bugchecks, we chose the easiest solution, instructing Windows to automatically restart when a critical system crash occurred:


System failure

[✓] Write an event to the system log

[✓] Automatically restart

Write debugging information

Kernel memory dump

Dump file:

%SystemRoot%\MEMORY.DMP

[✓] Overwrite any existing file

On the Bochs instrumentation side, a system restart could be trivially detected with the following callback:

```
void bx_instr_reset(unsigned cpu, unsigned type);
```

We could rely on a system restart as an indicator of the crash, because it was the only reason Windows could possibly shut down: the harness was set to work indefinitely, and hardly any other services or processes ran in the background (i.e. no Windows 10 Upgrade Tool). One major drawback of this solution was that at the point of a system restart, no context information (registers, flags, stack trace) was available anymore, not to mention having access to a system crash dump or any other crash-related information. The only trace of the bugcheck was the font file itself, but thanks to mechanisms such as Special Pools, and having an exact copy of the fuzzing environment prepared for reproduction, this turned out to be mostly sufficient. We were able to reproduce nearly all crashes occurring in Bochs based solely on the saved test cases.

Of course, there is a lot of room for improvement in this area. A better instrumentation could try to filter kernel exceptions and acquire CPU context information from there, hook into disk access in order to intercept the crash dump file being saved to the filesystem, or at the very least make a screenshot of the Blue Screen of Death, to get some basic idea about the crash (bugcheck type, four arguments). However, since the simplistic implementation worked so reliably for us and only took a few minutes to set up, we didn't feel the need to complicate it.

Reproducing crashes

Crash reproduction was generally one of the easiest stages of the process. We used a VirtualBox VM with the exact same environment as the fuzzer, and modified the font-loading harness to load the font from a file instead of requesting it through the LFENCE instruction. Then, we also had to write a simple "reproducer" program, which followed a very simple scheme:

1. Load the name of the last processed sample from the *progress.txt* file (if it exists).
2. Check if there are any crash dumps saved under C:\Windows\Minidumps.
   a. If there is one, copy it into a VirtualBox shared directory, under the name of the last font file.
   b. Start WinDbg, use it to generate a textual report of the crash dump (`!analyze -v`), and copy it into the same directory.
   c. Remove the crash dump locally.
3. List files in the input directory, skipping them until the last processed one is found. Choose the next one and save its name in *progress.txt*.
4. Start the modified harness several times (just to be sure, one should generally be enough).
   a. If a crash occurs, the system automatically saves a crash dump and restarts.
5. If we're still running at this point, it means that the crash is non-reproducible. In order to make sure that no corrupted system state persists when processing the next sample, restart the system "manually" (through the ExitWindowsEx API).

Thanks to the above algorithm, once the reproduction was over, we conveniently ended up with a list of directories corresponding to the crashing samples, and containing both binary crash dumps and their textual representations that we could easily grep through or process in other ways. By doing a full system reboot, we could also be 100% sure that each system crash was indeed caused by the one particular font, as no previously loaded ones could have affected the result. The downside was of course the increased time the overall process takes, but with an extremely optimized system which took <50 seconds to boot and usually another <10 seconds to test the font, the throughput was somewhere at ~1500 samples a day, which was acceptable to process the output of the first (most *crashy*) run in just a few days.

The only tweaks applied to the reproduction system as compared to the fuzzing one were to add the reproducer program to Startup, reduce the time spent in the "Windows Error Recovery" boot screen (appearing after a system crash) to 1 second, set up Shared Folders and install WinDbg to be able to generate textual crash dump excerpts within the guest itself. We also prepared a corresponding Windows 7 64-bit reproduction VM with more RAM, to potentially give Special Pools more memory to work with, but it didn't turn out to be very useful, as crashes generally either reproduced on both builds, or didn't on any of them.

Minimizing samples

There are two types of sample minimization which can be performed in the context of SFNT fonts: table-granular and byte-granular (within the scope of each table). Of course, both of them only make sense for the results of mutational fuzzing. When reporting the crashes to Microsoft, we only performed first-level, table granular minimization, which served two purposes: to satisfy our curiosity and to help deduplicate the crashes. Especially in the first iteration, when we had to deal with dozens of various stack traces, it was largely helpful to know which bugcheck was caused by mutations in which table(s).

The minimization tool was trivial in design: it used our existing C++ SFNT library to load both the original and mutated samples, and display a list of equal and differing tables. The user could then *restore* a chosen table (insert its original contents in place of the malformed one) and test if the new font would still trigger a crash. If yes, then mutations in the table were irrelevant and could be skipped; otherwise, at least some of them were essential for triggering the bug and had to be preserved. After a few repetitions, we were left with the 1-3 tables which were essential to the crash.

Byte granular minimization was only performed for several specific samples, in order to find out which fields in the font structures were the culprit of the crashes. Since none of the mutation algorithms changed the size of any of the tables (only modified data in a "static" way), a very simple algorithm could be used: in each iteration, half of the currently differing bytes would be restored in the mutated file. If a bugcheck still occurred, these bytes were unessential and could be forgotten about. If there was no crash anymore, some of the bytes must have been important, so the iteration's changes were reverted, and another subset of bytes was chosen to be tested. Minimization was complete when the set of differing bytes could no longer be reduced after a large number of attempts.

A separate minimization algorithm could be potentially devised for outputs of the TTF program generator; however, since there was only one such unique crash, and we didn't intend to perform its detailed root cause analysis, we didn't develop it in the end.

Future work

While the project has been more fruitful than we originally expected, in large part thanks to the individual approach to the file format and tested software, there are still a number of directions for further improvement. We've summarized them in the list below:

1. **Coverage-guided fuzzing.**
   The entire fuzzing process ran with no code coverage feedback enabled, which limited its ability to discover new code paths and have the corpus evolve towards a more comprehensive one. As AFL, LibFuzzer and other coverage-guided fuzzers have recently shown, this single feature can improve fuzzing results by an order of magnitude, depending on the target software, nature of input data and quality of the initial corpus.

   In our specific scenario, however, adding coverage recording and/or analysis to the Bochs instrumentation was not a feasible option, as it would incur further, significant overhead to a system which was already running extremely slowly. There are also other problems to address, such as how to distinguish instructions processing the input data from other kernel code, etc.

   The idea is related to the more general concept of performing coverage-guided Windows kernel fuzzing (Linux already has it, see syzkaller), regardless of whether it's fonts, system calls or some other attack surface. This is, however, a subject for a whole separate research, probably best implemented with real virtualization and VMI (*Virtual Machine Introspection*) for enhanced performance.

2. **Improved mutations.**
   While the mutation strategy used in the project was somewhat smarter than the average one observed in the past, there are still many ways to make it better. This includes adding more diverse dumb mutators, adding the ability to chain them together, splicing the tables of multiple samples into a single font, or determining mutation ratios on a per-font basis (instead of averaging out the results for the entire corpus).

   Other ideas include challenging the assumption that we manually found all the bugs in CharString processing and writing an instruction stream generator for OTF fonts, finding a way to fuzz-test SFNT tables which we previously deemed to be too fragile to mutate (`cmap`, `head`, `hhea`, `name`, `loca` and so on), or mutating fonts based on a representation other than the compiled, binary form (e.g. the XML .ttx files generated by FontTools).

3. **Fuzzing on other Windows versions.**
   The project was run on Windows 7 from the beginning to the end, with the assumption that it could contain the most bugs (e.g. those not backported from Windows 8.1 and 10). However, it is possible that newer platforms may include some added functionality, and fuzzing them could reveal vulnerabilities not present in Windows 7.

4. **Improved Bochs crash detection.**
   As mentioned in the previous section, crash detection was originally implemented based on a very simple indicator in the form of a machine reboot. This effectively limited the amount of crash context information we could collect to none, which made it impossible to investigate crashes which didn't reproduce with the saved sample. One potential solution could be to set up a callback for the `BX_INSTR_EXCEPTION` event, in order to catch CPU exceptions as they occur, and extract context information from there. This option also has the additional advantage that it could be used to signal kernel-mode exceptions which are typically masked by generic exception handlers, but still manifest

real bugs.

Other options, such as hooking into disk access in order to intercept crash dumps being saved to the filesystem, or making screenshots of BSoDs displayed by the kernel are also viable choices, but much less elegant and still quite limited in terms of the scope of information being obtained.

Summary

As a software testing technique, fuzzing has a very low entry bar and may be used to achieve satisfying results with little expertise or invested effort. However, it is still not a silver bullet in vulnerability hunting, and there are many stages which may require careful configuration or individual tailoring for a specific target or file format, especially for non-trivial targets such as closed-source operating system kernels. In this post, we have demonstrated how we attempted to enhance the process of Windows kernel font fuzzing to the maximum extent within the available time resources. We especially put a lot of energy into mutating, generating and exercising the inputs in a decently effective way, and into scaling the fuzzing process to thousands of machines, through the development of a dedicated Bochs instrumentation and aggressive optimization of the operating system. The outcome of the work, in the form of 16 high-severity vulnerabilities, has shown that the techniques were effective and improved upon previous work.

Considering how much potential fuzzing has and how broad the subject is, we look forward to seeing it grow further and be used to accomplish even more impressive effects, while ceasing to be perceived as a *voodoo* technique which "just works" regardless of the technical details behind it. In the upcoming weeks and months, we are also planning to share more of our experience and thoughts in this field.