A Test Harness for Fuzzing Font Parsing Engines in Web Browsers

James Fell, james.fell@alumni.york.ac.uk

Originally published in 2600: The Hacker Quarterly, Volume 34, Number 3, Autumn 2017

This article presents a cross-platform test harness written in Python that assists the user in searching for vulnerabilities in web browsers, specifically by fuzzing their font parsing functionality. The tool automates the delivery of test cases (font files in this context) into a web browser. The source code for the test harness should be available to download at **`https://www.2600.com/code/`**. To get the most out of this article it is recommended to have the source code open to refer to at the same time.

Fuzzing

Fuzzing is an established software testing process consisting of repeatedly delivering malformed input to an application while monitoring it for evidence of abnormal behaviour. Various memory corruption bugs such as use-after-free, double free and buffer overflows can be revealed in this way. Fuzzing is one of the most common methods for detecting vulnerabilities in software today. It is a form of dynamic analysis, as the software is being tested whilst it is executing. This is in contrast to static analysis which covers methods of examining an application's source code or a disassembly of the application's binary, without actually executing it.

There are two fundamental approaches to fuzzing based on how the malformed test cases are created; mutation and generation fuzzing. Mutation fuzzing takes one or more valid sample inputs and makes changes to them in some way, such as flipping bits. For example, a selection of PNG image files downloaded from the web could be randomly modified in order to fuzz an image viewer. Generation fuzzing on the other hand uses a specification of the format or protocol being fuzzed in order to generate test cases from scratch. For example, a grammar describing the JavaScript language could be used to generate slightly incorrect scripts to use as test cases when fuzzing a JavaScript interpreter.

In the context of this article, and the supplied test harness, we are using malformed font files to fuzz web browsers and it does not really matter how they were created. A simple example of applying mutation to sample font files is given later in the section titled Corpus Preparation, but many other approaches are possible.

Although the concept of fuzzing sounds quite simple (merely loading dodgy input into an application and seeing if it crashes) once you start trying to actually do it (and do it well) it often has a way of becoming complex. Issues such as creating good test cases, dealing with checksums or compression, delivering test cases to the target application, maximising code coverage, analysing crashes and so on can actually be quite tricky. The more advanced approaches to fuzzing also make use of techniques like taint analysis, symbolic execution and genetic algorithms to create better test cases. For anyone wanting to read more about the topic, Chapter 17 of The Shellcoder's Handbook [6] and Chapters 8-10 of Gray Hat Python [7] are good starting points.

Two excellent open source fuzzing tools that the reader should also download and take a look at are American Fuzzy Lop (AFL) [1] and Radamsa [2]. Reading their documentation and then experimenting with these two tools is a good way to get started with practical fuzzing and learn more.

Font Rendering

Most web browsers can read custom fonts from a web site in various formats including OTF (OpenType Font), TTF (TrueType Font) and WOFF (Web Open Font Format) files. The font can then be used for rendering some or all of the text that appears on that web site. The specific list of supported font formats varies from browser to browser and can generally be found in their documentation. In any case, the functionality in the browser responsible for parsing the font file after reading it from the remote web server can of course contain vulnerabilities. In such a case, a specially crafted font file can possibly cause arbitrary code to be executed on the target's computer.

As an example of this kind of vulnerability, back in 2011 the state-sponsored Duqu malware made use of a 0-day vulnerability (now assigned CVE-2011-3402) in the TrueType font parsing engine in win32k.sys on Microsoft Windows [3]. Duqu itself exploited this by having a malicious font file embedded in a Word document, but the same vulnerability could be exploited by convincing the target to visit the attacker's web page using Internet Explorer and delivering the TTF file in that web page.

The reader will probably have noticed that since win32k.sys is handling the font parsing in the example above this means that the vulnerability was not actually in Internet Explorer itself, but rather in the Windows XP kernel. Similarly, on the modern Windows 10 operating system the Edge web browser uses the DirectWrite library Dwrite.dll to handle fonts rather than having its own custom functionality. This may raise the question, why not just write a wrapper to dwrite.dll or the equivalent library if you intend to fuzz it, instead of processing fuzzed fonts through a web browser? However, that approach requires a separate test harness for each font parsing library on each platform. Also, some browsers actually do use their own custom font engines instead of passing the job to the operating system. The test harness presented here can be used unmodified to fuzz the font parsing functionality of any web browser on any OS, as long as there is a Python interpreter available. It is not the only way to approach the task but having one test harness that can be used for many targets seems like a good thing.

Corpus Preparation

The test harness that is presented here deals with injecting a corpus of malformed font files into a web browser and causing the browser to attempt to parse each font. Before this is described, it is worth giving a quick explanation of how such a set of font files could be created. There are many ways of achieving this, and it is not the focus of this article, but here is one example of how a corpus of malformed TTF files could be created.

First, it is necessary to obtain some samples of valid TTF files from somewhere. A simple Google search will be a good start, but the more variety in the sample files the better.

The user should then install Radamsa [2] on a Linux system and use it to mutate the valid TTF files as shown below.

```
radamsa -o output/test-%n.ttf -r input -n 50000
```

This will instruct Radamsa to read all of the valid TTF files in the directory called input. The tool will then create 50,000 new, mutated TTF files in the directory called output. These font files will each be slightly invalid in interesting ways that may trigger bugs when used. The precise ways in which Radamsa mutates input are described in the tool's own documentation.

Because Radamsa is a general purpose mutation fuzzer and is not aware of the specific format that it is mutating, some work now needs to be done to fix up the checksums inside the 50,000 mutated TTF files. Otherwise, the font parser being fuzzed will most likely reject each font file immediately and the only thing to be tested will be the bit of code that inspects checksums. In order to have our mutated files be fully

processed and potentially trigger bugs, we need to ensure that they will pass the basic checks that are likely to be carried out. Fortunately, there is a tool available on Windows called MsFontsFuzz [4] that can be used for this, at least when dealing with OTF and TTF fonts.

After copying the 50,000 mutated TTF files in the output directory over to a Windows system (or perhaps just use wine on the Linux system, I didn't check but it would probably work), the user can run the following from the command prompt. This assumes that the mutated fonts are now in c:\fonts on the Windows system.

```
for /f %%f in ('dir /b c:\fonts\') do msfontsfuzz test c:\fonts\%%f --fix-crcs
```

The command above will fix the checksums in each TTF file so that when they are loaded into the target they should not be immediately rejected. Once this command has finished, the contents of c:\fonts should be a corpus of 50,000 mutated TTF files now with valid checksums ready to be used in fuzzing.

Test Harness

The test harness presented here is essentially a web server written in Python that accepts connections from web browsers and delivers web pages to them containing malformed font files that are read from a filesystem directory. The user specifies two command line options when starting the harness; the path to the directory where the font corpus is stored and the TCP port to bind to on localhost.

This is the point in the article where it will be really helpful for you to download and open the source code and take a look at it.

Upon startup, the corpus directory is scanned and a list data structure containing all the font files in it is created. The Twisted framework [5] is then used to create a HTTP server listening on the requested port. If you do not already have this Python library, it can be installed by running **`pip install twisted`**.

The **`render_GET`** function contains the code that will be executed every time a HTTP GET request is received from the browser being fuzzed. It is in here that we must build up the web page to return to the browser and make sure that it uses a new font file each time.

The **`render_GET`** function handles three different cases of HTTP request URLs. When the document root (**`/`**) is requested we return the full web page. When the font (**`/font`**) is requested we read a font file from the corpus and return its contents. When any other URL is requested (for example the browser might request **`/favicon.ico`** or something automatically) we simply return an empty string in the HTTP response.

First we look at how to build a suitable web page when the document root (**`/`**) is requested. It is possible to load a custom font file into a web browser and use it for displaying text by using the @font-face CSS rule in a web page. The following snippet of CSS illustrates this.

```
@font-face {
     font-family: 'fuzzFont';
     src: url(/font);
}
```

This can be followed with further CSS to cause all text in the body of the web page to be rendered using that specified custom font.

```
body {
     font-family: 'fuzzFont';
}
```

Placing some text in the HTML body will now result in it being rendered using the font that is retrieved from the web server using **`/font`** as the URL.

A couple more things need to be added to the HTML web page before it is ready to be given to the browser. The harness places two meta tags into the web page header. The first causes the web browser to reload the page after one second, which in turn causes the web server to read and deliver the next font file, and causes the process to continue until all font files have been parsed.

```
<meta http-equiv="refresh" content="1">
```

The second is a meta tag to instruct the web browser to disable caching.

```
<meta http-equiv="cache-control" content="no-cache">
```

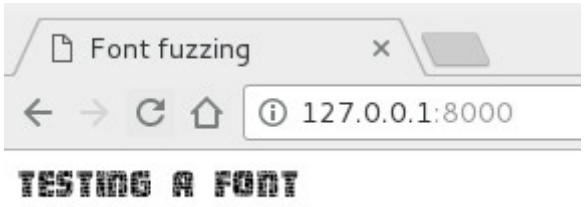
This is used simply to make sure that when the browser reloads the page it does not use any cached content (especially the font) but instead requests it all again from the web server, and hence receives the next font.

Whenever the **`render_GET`** function receives a request for **`/font`** it reads the next font file from disk and returns its contents in a HTTP response. The index into the font list data structure is incremented each time this happens until we have eventually served up all the fonts and reached the end of the list.

The screenshot below shows the test harness being started up.

```
user@laptop03:~/2600$ ./fontharness.py ./ttf-scraped-radamsa-fixed/ 8000
Found 10000 files in corpus
Starting HTTP Server. Please point the web browser to be tested at http://127.0.0.1:8000
```

At this point we would start up the web browser that we would like to fuzz and put the URL **`http://127.0.0.1:8000`** into the address bar to get the process started. The screenshot below shows this happening.



At this point you will see the web browser reloading the same page repeatedly every second. This is due to the refresh meta tag mentioned earlier. Each time the browser reloads the page it is receiving, and attempting to parse, a new font file from the corpus directory. It will also attempt to render the string “Testing a font” using the current font. We are now hoping that one of these mutated, malformed font files will crash the web browser when it attempts to make use of it. This would indicate a bug in the browser's font engine, and potentially an exploitable security vulnerability.

Now that the harness is running it is also writing to a log file. This is created in the same directory as the Python script and has the filename **`fontharness-log-n.txt`** where n is replaced with whichever TCP port you chose. In our example it would be 8000. Each time a new font file is served to the browser, its filename is appended to the log file. This is necessary for determining which font caused the browser to crash, when this eventually occurs.

Browser Instrumentation

Some final words are needed regarding instrumenting the web browser. The test harness does not handle this due to its requirement for being cross-platform and so the user must take care of it herself. Without instrumentation you will not be able to see what is happening inside the browser process and you will not detect bugs unless the entire browser actually crashes. The available options depend mostly upon which operating system you are using at the time.

On Linux systems, when fuzzing an open source web browser it is best to compile it using Asan (AddressSanitizer) [8] as this is excellent for detecting memory errors. This can be done simply by adding the **`-fsanitize=address`** option for gcc or clang on the command line when you compile it. You can also download precompiled Asan builds of both Chromium and Firefox from their respective web sites making it even easier for those two. If you have trouble getting the target browser to compile with Asan or you do not have the source code another option is to simply start it up and then attach gdb (GNU debugger).

On Windows it is good practice to enable Page Heap for the specific browser process before you start it. This can be done by typing the following command in an Administrator command prompt.

```
gflags /p /enable c:\path\to\browser.exe /full
```

This acts a little bit like Asan on Linux by causing an exception to be raised if any heap memory corruption occurs. Unfortunately some browsers implement their own memory management instead of using the operating system and so Page Heap has no effect on them. Either way, you can then attach a debugger such as WinDbg or Immunity Debugger to the running browser process before you begin fuzzing.

Conclusion

The Python software presented in this article allows the user to cause a web browser to sequentially process each font file in a given directory. When combined with a corpus of mutated and malformed font files this allows the testing of the font parsing functionality in any web browser on any operating system, as long as a Python interpreter is available. By attaching a debugger or other suitable instrumentation to the web browser, error states can be detected and investigated. These can potentially be exploitable security vulnerabilities.

The test harness is pretty simple and can certainly be improved upon, but it is good enough to get started with. My hope is that the tool and this article will help more people to get started in fuzzing. I am happy to receive feedback or questions by email.

References

[1] – Michal Zalewski, “American Fuzzy Lop”. http://lcamtuf.coredump.cx/afl/

[2] – Oulu University Secure Programming Group, “Radamsa”. https://github.com/aoh/radamsa

[3] – Mitre, “CVE-2011-3402”. https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2011-3402

[4] - Oleksiuk Dmytro, “MsFontsFuzz: OpenType font format fuzzer for Windows”. https://github.com/Cr4sh/MsFontsFuzz

[5] – Twisted Matrix Labs, “Twisted”. https://twistedmatrix.com/

[6] – Chris Anley et al, “The Shellcoder's Handbook: Discovering and Exploiting Security Holes”, Second Edition. Wiley Publishing, 2007.

[7] – Justin Seitz, “Gray Hat Python: Python Programming for Hackers and Reverse Engineers”. No starch press, 2009.

[8] – Google, “AddressSanitizer”. https://github.com/google/sanitizers/wiki/AddressSanitizer