BlackHat 13

Smashing the Font Scaler Engine in Windows Kernel

Ling Chuan Lee & Lee Yee Chan

F13 Laboratory

Abstract

The Font Scaler engine is widely used to scale the outline font definition such as TrueType/OpenType font for a glyph to a specific point size and convert the outline into a bitmap at a specific resolution. The revolution of font in computers which is mainly used for styling purposes has made many users overlook its security issues. In fact, the Font Scaler engine can cause many security impacts especially in the Windows kernel mode.

In this paper, the basic structure of the Font Scaler engine will be discussed. This includes the conversion of an outline into a bitmap, the mathematical description of each glyph in an outline font, a set of instruction in each glyph which instructs the Font Scaler engine to modify the shape of the glyph, and the instruction interpreter etc.

Next, we introduce our smart font fuzzing method for identifying the new vulnerabilities of the Font Scaler engine. The difference between dumb fuzzing and vulnerable functions will be explained and we will prove that the dumb fuzzing technique is not a good option for Windows Font Fuzzing.

Lastly, we focus on the attack vector that could be used to launch the attacks remotely and locally. A demonstration of the new TrueType font (TTF) vulnerabilities and the attack vector on Windows 8 and Windows 7 will be shown.

1.0 Introduction

Computer uses different styles of typefaces to display text.Today, hundreds and thousands of font files have been developed in the digital world. Different categories of font are available within Microsoft Windows; for instance, GDI Fonts, Device Fonts and many more. TrueType font (TTF) is one of the common GDI fonts providing support for font management and text output. It is a digital font which includes many different kind of information used by rasterizer and the operating system software to display characters on a computer screen or print out in other devices such as a printer.

All fonts, including the TTF contain glyphs. It is a set of paths, (also known as outline) filled with pixels to create the final letter form of a character.The outline of a character in a TrueType font is made of a straight line segment and a closed curve specified using points and particular mathematics.

A few steps are required for displaying a font file on raster devices [1] and are shown as below:

1. The outline stored in the font file is scaled to the requested size.
2. Scaler converts Font Unit (FUnits) to pixel coordinates and scales outline to the size requested by the application.
3. Instructions associated with glyph are carried out by the interpreter. Interpreter executes instructions associated with glyph and grid fits.
4. The result is a grid-fitted outline for the requested glyph.
5. The outline is then scan converted to produce a bitmap that can be rendered on the targeted device.

2.0 The Font Scaler Engine

The Font Scaler engine [2] creates the necessary bitmap at a particular resolution when a specific point size is requested by an application. Typically, the Font Scaler engine consists of a set of glyph instructions that instruct the font scaler to modify the shape of the outline of a character for a particular font size for resolution display purposes.

Font scaler consists of a set of API functions. The user can pass parameters to the Font Scaler through the fs_GlyphInputType data structure and receive information from the fs_GlyphInfoType record [2]. Some important functions for the Font Scaler are shown in Table 1.

Functions	Description
win32k!fs_OpenFonts	Opens the Font Scaler
win32k!fs_Initialize	Initializes the Font Scaler
win32k!fs_NewSfnt	Retrieves data from sfnt data structure, the win32k!sfnt_DoOffsetTableMap function will be used to map offset and length table.

Functions	Description
win32k!fs__NewTransformation	Specifies the size, pixel and the used of resolution.
win32k!fs_NewGlyph	Displays a new glyph
win32k!fs_ContourScan	Required when converts the glyph into a bitmap
win32k!fs_FindBitMapSize	Calculates the amount of memory that is needed
win32k!fs_GetGlyphIDs	Returns glyph IDs for a range of character code

Table: 1 Font Scaler functions

Table 2 summarizes the routine functions that allow a glyph from a TTF file to be displayed [2].

Description	Win32k function
Engine exported interface	win32k!fs_NewGlyph win32k!fs_ContourScan win32k!fs_FindBitMapSize win32k!fs_Initialize win32k!fs_GetGlyphIDs win32k!fs_OpenFonts win32k!fs_NewSfnt win32k!fs_SetUpKey win32k!fs_WinNTGetGlyphIDs win32k!fs_ConvertGrayLevels win32k!fs_NewContourGridFit
Engine internal interface	win32k!fs__Contour win32k!fs__NewTransformation
Engine converter function	win32k!fsc_SetupScan win32k!fsc_FillBitMap win32k!fsc_CheckYReversalInSpline win32k!fsc_MeasureGlyph win32k!fsc_FillGlyph win32k!fsc_CalcLine win32k!fsc_CalcSpline win32k!fsc_OverScaleOutline win32k!fsc_BLTHoriz win32k!fsc_OverscaleToSubPixel win32k!fsc_OverscaleToBold win32k!fsc_InitializeBitMasks win32k!fsc_CheckEndPoint win32k!fsc_CalcGrayMap win32k!fsc_BeginElement win32k!fsc_CalcGrayRow win32k!fsc_AllocVMem

Description	Win32k function
	win32k!fsc_RemoveDups win32k!fsc_EndContourEndpoint
Engine support function	win32k!fsg_CreateGlyphData win32k!fsg_WorkSpaceSetOffsets win32k!fsg_ExecuteGlyph win32k!fsg_CompositeInnerGridFit win32k!fsg_CheckOutlineOrientation win32k!fsg_Embold win32k!fsg_MergeGlyphData win32k!fsg_DoScanControl win32k!fsg_RestoreContourData win32k!fsg_RunPreProgram win32k!fsg_CopyFontProgramResults win32k!fsg_GridFit win32k!fsg_InitInterpreterTrans win32k!fsg_UpdatePrivateSpaceAddresses win32k!fsg_PrivateFontSpaceSize
Bitmap related function	win32k!sbit_EmboldenSubPixel win32k!sbit_CalcDevHorMetrics win32k!sbit_GetDevAdvanceWidth win32k!sbit_GetDevAdvanceHeight win32k!sbit_GetMetrics win32k!sbit_GetBitmap win32k!sbit_EmboldenGrayFromMono win32k!sbit_NewTransform win32k!sbit_EmboldenGray win32k!sbit_SearchForBitmap win32k!sbit_Embolden win32k!sbit_ExpandGrayFromMono
Instruction virtual machine function	win32k!itrp_SHP win32k!itrp_ROUND win32k!itrp_WPV win32k!itrp_DIV win32k!itrp_SPVTCA_0 win32k!itrp_SPVTCA_1 win32k!itrp_FDEF win32k!itrp_PUSHB win32k!itrp_SROUND win32k!itrp_IF win32k!itrp_SHC win32k!itrp_NPUSHB win32k!itrp_IP win32k!itrp_WCVT win32k!itrp_MSIRP win32k!itrp_RoundToHalfGrid win32k!itrp_PUSHW win32k!itrp_RoundToHalfGridSP win32k!itrp_DUP win32k!itrp_JROT

Description	Win32k function
	win32k!itrp_SHE win32k!itrp_JROF win32k!itrp_MINDEX win32k!itrp_ELSE win32k!itrp_SDPVTL win32k!itrp_RCVT win32k!itrp_MD win32k!itrp_LOOPCALL win32k!itrp_SHPIX win32k!itrp_GETINFO (more refers to Appendix)

Table 2: Functions that allow a glyph to be displayed

3.0 TTF Fuzzing

A TTF file consists of a sequence of concatenated tables. The combination of data from different tables will be used to render the glyph data in the font. A basic font consists of multiple tables that are specified in its header. Typically, a binary TTF file begins with the Font Offset Table. The Font Offset Table is divided into five subtables, which includes the following.

sfnt version	: 65536 (0x0001 0000) for version 1.0
numTables	: Number of tables
searchRange	: (Maximum power of 2 ≤ *numTables*) x 16
entrySelector	: log2 (Maximum power of 2 ≤ *numTables*)
rangeShift	: numTables x 16 –searchRange

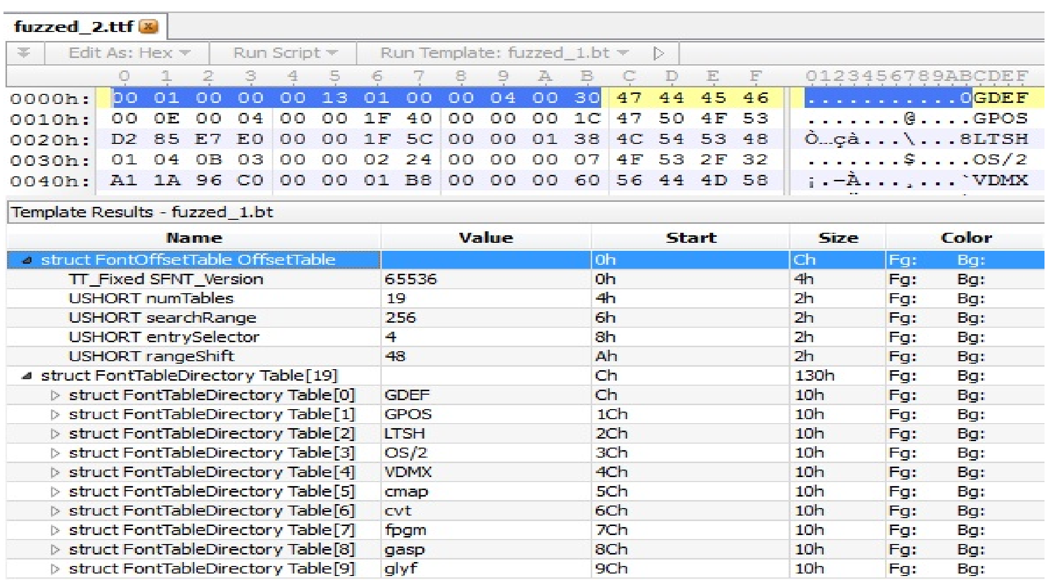

Figure 1: Font Offset Table

The FontOffset Table is followed by a sequence of tables containing the font data. These tables can be arranged in any order. A basic font is composed of multiple tables as specified in its header. Each font table directory header consists of four subtables as shown below.

tag	: 4 byte identifier
checksum	: Checksum of the table
offset	: beginning offset of the font table entry
length	: Length of the table

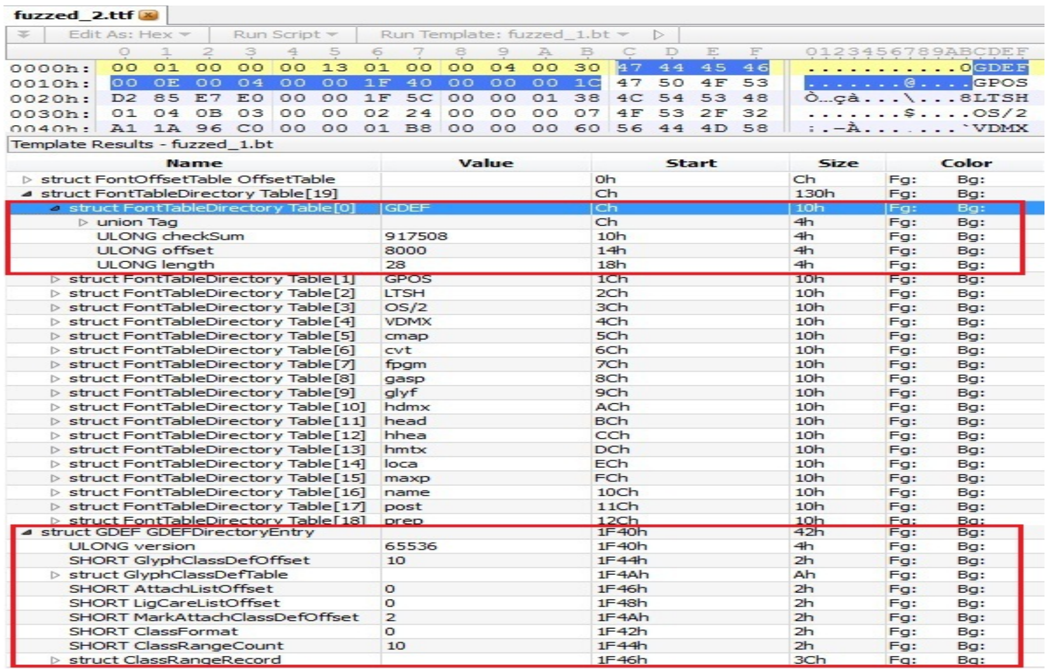

Figure 2: Font Table Directory

Table 3 shows the required table that must appeal in any valid TTF file.

Tag	**Table**
cmap	Character to glyph mapping
glyph	Glyph data
head	Font header
hhea	Horizontal header
hmtx	Horizontal metrics
loca	Index to location
maxp	Maximum profile
name	Naming table
post	PostScript information
OS/2	OS/2 and Windows specific metrics

Table 3: Required Table in TTF

In certain circumstances, due to the functionality expected of a given TTF font, optional tables may be needed. Table 4 lists the optional tables together with their tag name.

Tag	Table
cvt	Control Value Table
EBDT	Embedded bitmap data
EBLC	Embedded bitmap location data
EBSC	Embedded bitmap scaling data
fpgm	Font program
gasp	Grid-fitting and scan conversion procedure
hdmx	Horizontal device metrics
kern	kerning
LTSH	Linear threshold table
prep	CVT Program
PCLT	PCLT

Table 4: Optional Table in TTF

Due to font validation purposes, the dumb fuzzing technique is not recommended for these fields: “checkSum”, “offset”, “length” and “Table”. To reduce the number of irrelevant tests, a checksum validation program is used to determine the checksum of the “head” table.

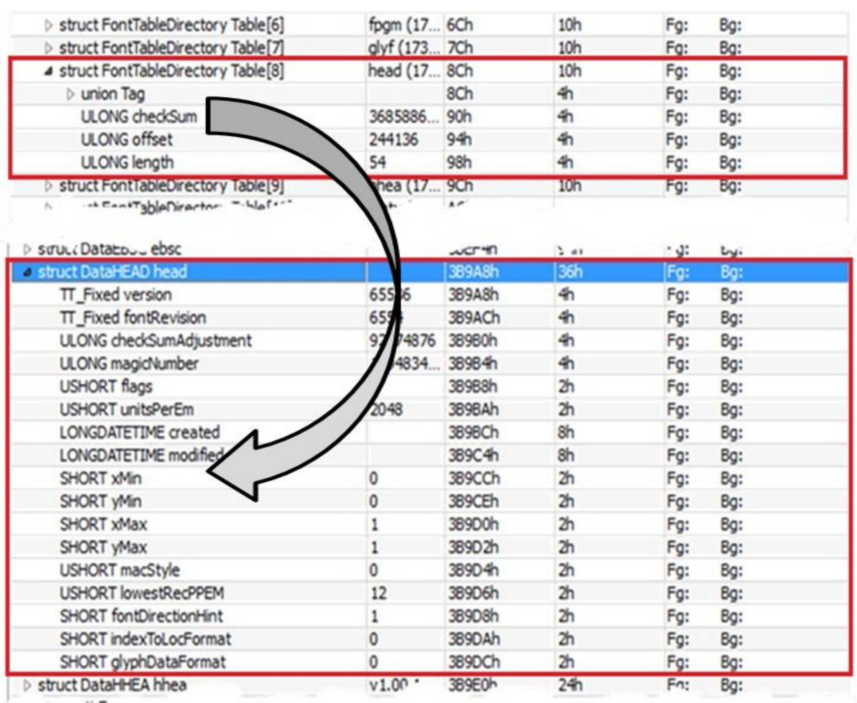

Figure 3: Validation process for TTF fuzzing

During the fuzzing process, the table checksum has to be re-computed. The checksum calculation implies a four bytes boundary as shown in the Python program below.

```
def chk(tab):
        total_data=0
        for i in range(0, len(tab), 4):
              data=unpack(“>I”,tab[i:i+4])[0]
              total_data += data
        final_data=0xFFFFFFFF &total_data
        return final_data
```

The TTF font fuzzer [3] is created to fuzz the TTF font into different sizes which enables the generation of test cases to determine the size of font in triggering the vulnerability. The overall process of the fuzzer starts with automating the installation of the crafted font in a Windows system. It will then display the font in a different size, uninstall the font type and repeat the process if no vulnerability is found.

Before using a font with a specified size and displaying it on a window, the font must be installed. Since the crafted TTF font is designed to exploit the Windows 8 Pro, the TTF font is not installed by default by Microsoft Windows during setup. The windll.gdi32.AddFontResourceExA function is used to automate the installation of the crafted font into the ‘C:\Windows\Fonts’ folder.

```
htr=windll.gdi32.AddFontResourceExA(fileFont, FR_PRIVATE,None)
```

Next, our fuzzer will need to prepare an environment by registering a window class and creating a new window to automate the display of the font text. Once the fuzzing environment is ready, a LOGFONT [4] object is created to define the attributes of a font.

```
lf=win32gui.LOGFONT()
```

Since the objective of this fuzzer is to fuzz the font into different sizes, the range of font size to be fuzzed has to be defined. The range can start and end at any number and an increment number can be specified depending on one’s preference. Just like everything else in the computer, a font must have a name. Thus, the defined name should always go to the name of the crafted TTF font. Below is a set of properties used to describe a font.

```
lf.lfHeight=fontsize
lf.lfFaceName="xxxx"
lf.lfWidth=0
lf.lfEscapement=0
lf.lfOrientation=0
lf.lfWeight=FW_NORMAL
lf.lfItalic=False
lf.lfUnderline=False
lf.lfStrikeOut=False
lf.lfCharSet=DEFAULT_CHARSET
lf.lfOutPrecision=OUT_DEFAULT_PRECIS
lf.lfClipPrecision=CLIP_DEFAULT_PRECIS
lf.lfPitchAndFamily=DEFAULT_PITCH|FF_DONTCARE
```

The fuzzer will then proceed by displaying the pre-set font with the predefined attributes. As expected from the name, a LOGFONT structure is a logical font. However, due to the application that needs to work with fonts at a lower level, it means that the target font functions are not specified and HFONT always maps to the same physical font internally. Both windll.gdi32.ExtTextOutW and ETO_GLYPH_INDEX are used as physical font APIs.

```
windll.gdi32.ExtTextOutW(
     hdc,
     5,
     5,
     ETO_GLYPH_INDEX,
     None,
     var1,
     len(var1),
     None)
```

Assuming no vulnerability has been found at a font with a specified size that has been called, the windll.gdi32.RemoveFontResourceExW function will be called to remove the fonts in the ‘C:\Windows\Fonts’ folder.

```
windll.gdi32.RemoveFontResourceExW(fileFont, FR_PRIVATE,None)
```

Another size of font in the range will be called and the same process will repeat until vulnerability is found or the list of font size elements under a loop function has all been called and no vulnerability is found.

4.0 Attack Vector

The Graphics Device Interface (GDI) is part of the core OS component. It is responsible for graphical object display and output transmission to devices such as printers.The vulnerability of a font able is to be launched via several attack vectors, both locally and remotely.

4.1 Local Font Attack Vector

Typically, the method to trigger the local attack might be different depending on the vulnerability of the font; some might need the user to select and click the file and display via fontview.exe. However, there are some vulnerable fonts that can be triggered immediately when the mouse cursor points to the font file.

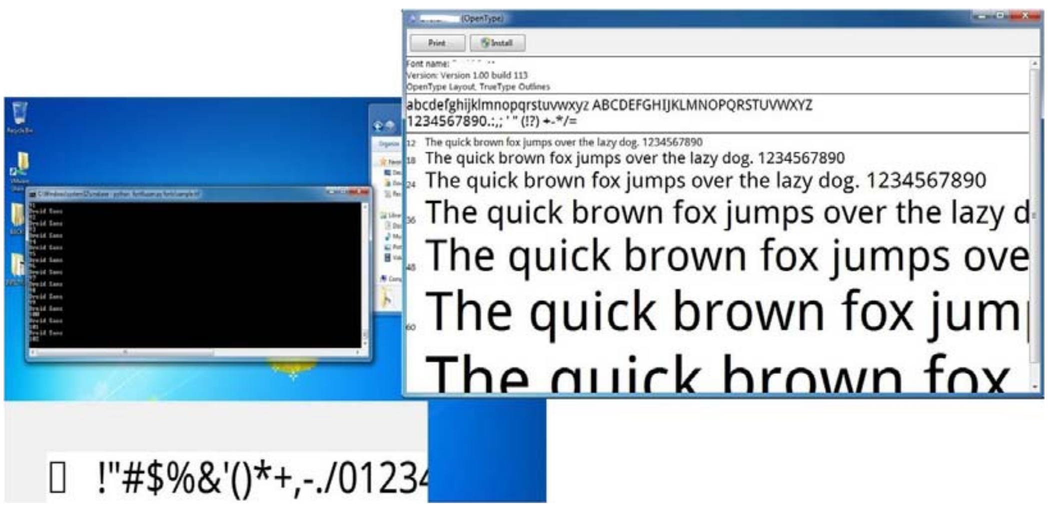

Figure 4: The attacker copies and executes a crafted font in a Windows system to raise the user privilege as a super user.

4.2 Remote Font Attack Vector

Some font vulnerabilities can allow remote attacks as long as the vulnerable font is embedded in a docx or html file. The vulnerable font can be attacked via browser (Firefox, Chrome), Microsoft Office Documents (*.docx, *.pptx) and other applications such as Adobe Portable Document format (*.pdf).

4.2.1 Remote Font Attack Vector – html:

The font vulnerability can allow remote code execution if the victim opens the crafted web page that is embedded with a TTF. Figure 5 below shows how the attacker uses the CSS @font-face property to embed crafted TTF into a web page.

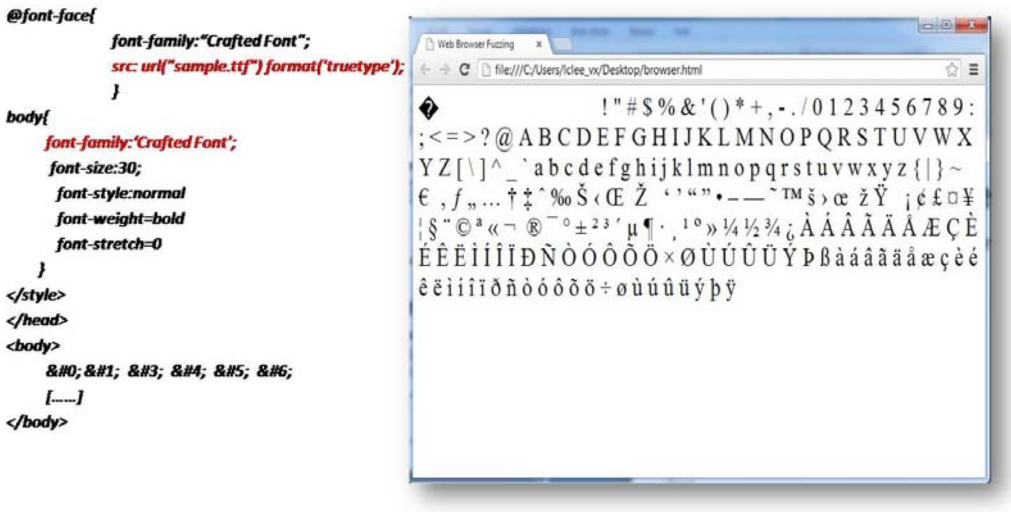

Figure 5: Embedded crafted font inahtml file

4.2.2 Remote Font Attack Vector – Docx:

The remote font attack can be launched by embedding a crafted font into a Microsoft Office document. To do this, a TTF file format has to be converted into an obfuscated TTF font file format (ODTTF) before performing the embedding process and this must satisfy the following requirements.

1. A 128-bit Global Unique Identifier (GUID) is generated.
2. An XOR operation on the first 32 byte of the TTF is performed.
3. The ODTTF font file is embedded into Microsoft Office Word.

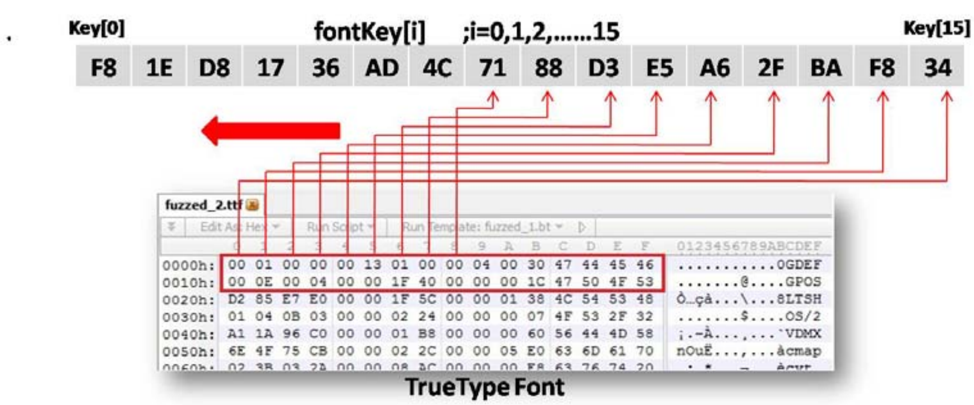

Figure 6: Performing the XOR operation on the first 32 byte of the TTF

The Python code below simplifies the obfuscation process by converting the TTF file to an ODTTF file.

```
fontKey = keys.decode("hex")
obfFontString = open(ttfFontFile, 'rb').read()
fontString = [ord(x) for x in obfFontString]
fori in range(16):
    fontString[i] = ord(obfFontString[i]) ^ ord(fontKey[15-i])
    fontString[i+16] = ord(obfFontString[i+16]) ^ ord(fontKey[15-i])
```

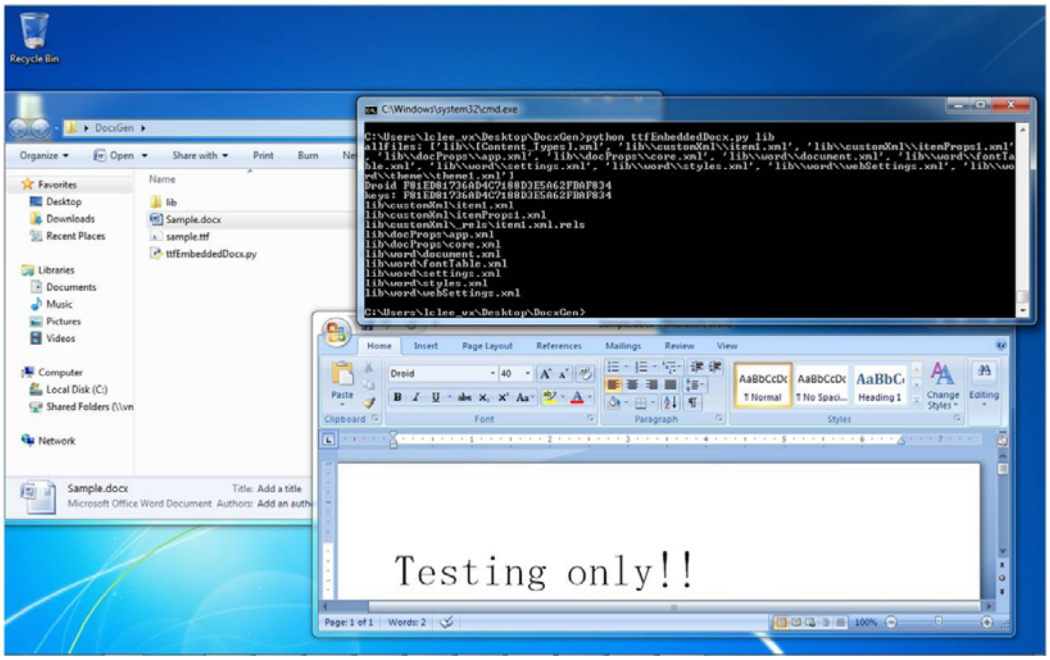

Figure 7: Successfully embedded a crafted TTF font file in Microsoft document file

Summary

Two TTF tests are included in our TTF fuzzing. One is by opening the TTF file using FontView.exe and the other is by calling the glyph index from character map and displaying the text in different sizes. It is recommended that the display test of the targeted font size is not set to start at font size 0. This is because Microsoft Office does not accept font size 0. Lastly, to ease the analysis process it is advised to focus on a few glyphs only before starting the fuzzing process.

Reference

[1] TrueType 1.0 Font File, Technical Specification Revision 1.66 August 1995

[2] Understanding Windows Kernel Font Scaler Engine Vulnerability, Wang Yu SyScan 360 2012

[3] Partial of TrueType Font Fuzzer, https://github.com/lingchuanlee/FontFuzzer/

[4] LOGFONT Structure, http://msdn.microsoft.com/en-us/library/windows/desktop/dd145037(v=vs.85).aspx

Appendix

```
;Function[xx] = TrueType Instructions
;xx is opcode from 0x00 until 0xBF
function[0x00] = itrp_SVTCA_0;
function[0x01] = itrp_SVTCA_1;
function[0x02] = itrp_SPVTCA_0;
function[0x03] = itrp_SPVTCA_1;
function[0x04] = itrp_SFVTCA_0;
function[0x05] = itrp_SFVTCA_1;
function[0x06] = itrp_SPVTL;
function[0x07] = itrp_SPVTL;
function[0x08] = itrp_SFVTL;
function[0x09] = itrp_SFVTL;
function[0x0A] = itrp_WPV;
function[0x0B] = itrp_WFV;
function[0x0C] = itrp_RPV;
function[0x0D] = itrp_RFV;
function[0x0E] = itrp_SFVTPV;
function[0x0F] = itrp_ISECT;
function[0x10] = itrp_SRP0;
function[0x11] = itrp_SRP1;
function[0x12] = itrp_SRP2;
function[0x13] = itrp_SetElementPtr;
function[0x14] = itrp_SetElementPtr;
function[0x15] = itrp_SetElementPtr;
function[0x16] = itrp_SetElementPtr;
function[0x17] = itrp_LLOOP;
function[0x18] = itrp_RTG;
function[0x19] = itrp_RTHG;
function[0x1A] = itrp_LMD;
function[0x1B] = itrp_ELSE;
function[0x1C] = itrp_JMPR;
function[0x1D] = itrp_LWTCI;
function[0x1E] = itrp_LSWCI;
function[0x1F] = itrp_LSW;
function[0x20] = itrp_DUP;
function[0x21] = itrp_POP;
function[0x22] = itrp_CLEAR;
function[0x23] = itrp_SWAP;
function[0x24] = itrp_DEPTH;
function[0x25] = itrp_CINDEX;
function[0x26] = itrp_MINDEX;
function[0x27] = itrp_ALIGNPTS;
function[0x28] = itrp_RAW;
function[0x29] = itrp_UTP;
function[0x2A] = itrp_LOOPCALL;
function[0x2B] = itrp_CALL;
function[0x2C] = itrp_FDEF;
```

```
function[0x2D] = itrp_IllegalInstruction;
function[0x2E] = itrp_MDAP;
function[0x2F] = itrp_MDAP;
function[0x30] = itrp_IUP;
function[0x31] = itrp_IUP;
function[0x32] = itrp_SHP;
function[0x33] = itrp_SHP;
function[0x34] = itrp_SHC;
function[0x35] = itrp_SHC;
function[0x36] = itrp_SHE;
function[0x37] = itrp_SHE;
function[0x38] = itrp_SHPIX;
function[0x39] = itrp_IP;
function[0x3A] = itrp_MSIRP;
function[0x3B] = itrp_MSIRP;
function[0x3C] = itrp_ALIGNRP;
function[0x3D] = itrp_RTDG;
function[0x3E] = itrp_MIAP;
function[0x3F] = itrp_MIAP;
function[0x40] = itrp_NPUSHB;
function[0x41] = itrp_NPUSHW;
function[0x42] = itrp_WS;
function[0x43] = itrp_RS;
function[0x44] = itrp_WCVT;
function[0x45] = itrp_RCVT;
function[0x46] = itrp_RC;
function[0x47] = itrp_RC;
function[0x48] = itrp_WC;
function[0x49] = itrp_MD;
function[0x4A] = itrp_MD;
function[0x4B] = itrp_MPPEM;
function[0x4C] = itrp_MPS;
function[0x4D] = itrp_FLIPON;
function[0x4E] = itrp_FLIPOFF;
function[0x4F] = itrp_DEBUG;
function[0x50] = itrp_LT;
function[0x51] = itrp_LTEQ;
function[0x52] = itrp_GT;
function[0x53] = itrp_GTEQ;
function[0x54] = itrp_EQ;
function[0x55] = itrp_NEQ;
function[0x56] = itrp_ODD;
function[0x57] = itrp_EVEN;
function[0x58] = itrp_IF;
function[0x59] = itrp_EIF;
function[0x5A] = itrp_AND;
function[0x5B] = itrp_OR;
function[0x5C] = itrp_NOT;
function[0x5D] = itrp_DELTAP1;
function[0x5E] = itrp_SDB;
```

```
function[0x5F] = itrp_SDS;
function[0x60] = itrp_ADD;
function[0x61] = itrp_SUB;
function[0x62] = itrp_DIV;
function[0x63] = itrp_MUL;
function[0x64] = itrp_ABS;
function[0x65] = itrp_NEG;
function[0x66] = itrp_FLOOR;
function[0x67] = itrp_CEILING;
function[0x68] = itrp_ROUND;
function[0x69] = itrp_ROUND;
function[0x6A] = itrp_ROUND;
function[0x6B] = itrp_ROUND;
function[0x6C] = itrp_NROUND;
function[0x6D] = itrp_NROUND;
function[0x6E] = itrp_NROUND;
function[0x6F] = itrp_NROUND;
function[0x70] = itrp_WCVTFOD;
function[0x71] = itrp_DELTAP2;
function[0x72] = itrp_DELTAP3;
function[0x73] = itrp_DELTAC1;
function[0x74] = itrp_DELTAC2;
function[0x75] = itrp_DELTAC3;
function[0x76] = itrp_SROUND;
function[0x77] = itrp_S45ROUND;
function[0x78] = itrp_JROT;
function[0x79] = itrp_JROF;
function[0x7A] = itrp_ROFF;
function[0x7B] = itrp_IllegalInstruction;
function[0x7C] = itrp_RUTG;
function[0x7D] = itrp_RDTG;
function[0x7E] = itrp_SANGW;
function[0x7F] = itrp_AA;
function[0x80] = itrp_FLIPPT;
function[0x81] = itrp_FLIPRGON;
function[0x82] = itrp_FLIPRGOFF;
function[0x83] = itrp_IDefPatch;
function[0x84] = itrp_IDefPatch;
function[0x85] = itrp_SCANCTRL;
function[0x86] = itrp_SDPVTL;
function[0x87] = itrp_SDPVTL;
function[0x88] = itrp_GETINFO;
function[0x89] = itrp_IDEF;
function[0x8A] = itrp_ROTATE;
function[0x8B] = itrp_MAX;
function[0x8C] = itrp_MIN;
function[0x8D] = itrp_SCANTYPE;
function[0x8E] = itrp_INSTCTRL;
function[0xB0] = itrp_PUSHB1;
function[0xB1] = itrp_PUSHB;
```

```
function[0xB2] = itrp_PUSHB;
function[0xB3] = itrp_PUSHB;
function[0xB4] = itrp_PUSHB;
function[0xB5] = itrp_PUSHB;
function[0xB6] = itrp_PUSHB;
function[0xB7] = itrp_PUSHB;
function[0xB8] = itrp_PUSHW1;
function[0xB9] = itrp_PUSHW;
function[0xBA] = itrp_PUSHW;
function[0xBB] = itrp_PUSHW;
function[0xBC] = itrp_PUSHW;
function[0xBD] = itrp_PUSHW;
function[0xBE] = itrp_PUSHW;
function[0xBF] = itrp_PUSHW;
```