Reverse engineering and exploiting font rasterizers

The OpenType saga

Mateusz “j00ru” Jurczyk

44CON 2015, London
PS> whoami

• Project Zero @ Google

• Low-level security researcher with interest in all sorts of vulnerability research and software exploitation

• http://j00ru.vexillium.org/

• @j00ru
Agenda

• Type 1 / OpenType font primer

• Chapter 1 – how it all started
  • FreeType arbitrary out-of-bounds stack-based write access ([CVE-2014-2240], [CVE-2014-9659])

• Chapter 2 – the Charstring research
  • Adobe Type Manager Font Driver in the Windows kernel, and shared codebases
  • Results of manual vulnerability hunting in Microsoft Windows, DirectWrite, .NET and Adobe Reader

• Chapter 3 – font fuzzing
  • Recently fixed Windows kernel TrueType and OpenType vulnerabilities
  • Bug collisions

• Final thoughts
Type 1 / OpenType font primer
Adobe PostScript fonts

- In 1984, Adobe introduced two outline font formats based on the PostScript language (itself created in 1982):
  - *Type 1*, which may only use a specific subset of PostScript specification.
  - *Type 3*, which can take advantage of all of PostScript’s features.
- Originally proprietary formats, with technical specification commercially licensed to partners.
  - Only publicly documented in March 1990, following Apple’s work on an independent font format, *TrueType*. 
Type 1 primer – general structure

Adobe Type 1 Font Format, Adobe Systems Incorporated
Type 1 Charstrings

/at ## - | { 36 800 hsbw -15 100 hstem 154 108 hstem 466 108 hstem 666 100
     hstem 445 85 vstem 155 120 vstem 641 88 vstem 0 100 vstem 275 353 rmoveto
     54 41 59 57 vhcurveto 49 0 30 -39 -7 -57 rrcurveto -6 -49 -26 -59 -62 0
     rrcurveto -49 -27 43 48 hvcurveto closepath 312 212 rmoveto -95 hlineto
     -10 -52 rlineto -30 42 -42 19 -51 0 rrcurveto -124 -80 -116 -121 hvcurveto
     -101 80 -82 88 vhcurveto 60 0 42 28 26 29 rrcurveto 33 4 callsubr 8 -31
     26 -25 28 -1 rrcurveto 48 -2 58 26 48 63 rrcurveto 40 52 22 75 0 82 rrcurveto
     0 94 -44 77 -68 59 rrcurveto -66 59 -81 27 -88 0 rrcurveto -213 -169 -168
     -223 hvcurveto -225 173 -165 215 vhcurveto 107 0 92 31 70 36 rrcurveto
     -82 65 rlineto -32 -20 -64 -12 -83 0 rrcurveto -171 -125 108 182 hvcurveto
     172 111 119 168 vhcurveto 153 0 118 -84 -9 -166 rrcurveto -5 -86 -51 -81
     -36 -4 rrcurveto -29 -3 12 43 5 24 rrcurveto closepath endchar } | -
Type 1 Charstring execution context

- **Instruction stream** – the stream of encoded instructions used to fetch operators and execute them. Not accessible by the Type 1 program itself.

- **Operand stack** – a LIFO structure holding up to 24 numeric (32-bit) entries. Similarly to PostScript, it is used to store instruction operands.
  - various instructions interpret stack items as 16-bit or 32-bit numbers, depending on the operator.

- **Transient array** or **BuildCharArray** – a fully accessible array of 32-bit numeric entries; can be pre-initialized by specifying a `/BuildCharArray` array in the Private Dictionary, and the size can be controlled via a `/lenBuildCharArray` entry of type “number”.

  The data structure is not officially documented anywhere that I know of, yet most interpreters implement it.
Type 1 Charstring operators

Officially, divided into six groups by function:

• Byte range 0 – 31:
  • Commands for starting and finishing a character’s outline,
  • Path constructions commands,
  • Hint commands,
  • Arithmetic commands,
  • Subroutine commands.

• Byte range 32 – 255:
  • Immediate values pushed to the operand stack; a special encoding used with more bytes loaded from the instruction stream in order to represent the full 32-bit range.
Type 1 Charstring operators

<table>
<thead>
<tr>
<th>Value</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>hstem</td>
</tr>
<tr>
<td>3</td>
<td>vstem</td>
</tr>
<tr>
<td>4</td>
<td>vmoveto</td>
</tr>
<tr>
<td>5</td>
<td>rlineto</td>
</tr>
<tr>
<td>6</td>
<td>hlineto</td>
</tr>
<tr>
<td>7</td>
<td>vlineto</td>
</tr>
<tr>
<td>8</td>
<td>rrcurveto</td>
</tr>
<tr>
<td>9</td>
<td>closepath</td>
</tr>
<tr>
<td>10</td>
<td>callsubr</td>
</tr>
<tr>
<td>11</td>
<td>return</td>
</tr>
<tr>
<td>12</td>
<td>escape</td>
</tr>
<tr>
<td>13</td>
<td>hsbw</td>
</tr>
<tr>
<td>14</td>
<td>endchar</td>
</tr>
<tr>
<td>21</td>
<td>rmoveto</td>
</tr>
<tr>
<td>22</td>
<td>hmoveto</td>
</tr>
<tr>
<td>30</td>
<td>vhcurveto</td>
</tr>
<tr>
<td>31</td>
<td>hvcurveto</td>
</tr>
<tr>
<td>12 0</td>
<td>dotsection</td>
</tr>
<tr>
<td>12 1</td>
<td>vstem3</td>
</tr>
<tr>
<td>12 2</td>
<td>hstem3</td>
</tr>
<tr>
<td>12 6</td>
<td>seac</td>
</tr>
<tr>
<td>12 7</td>
<td>sbw</td>
</tr>
<tr>
<td>12 12</td>
<td>div</td>
</tr>
<tr>
<td>12 16</td>
<td>callothersubr</td>
</tr>
<tr>
<td>12 17</td>
<td>pop</td>
</tr>
<tr>
<td>12 33</td>
<td>setcurrentpoint</td>
</tr>
</tbody>
</table>
Type 1 Charstring operators

• The Type 1 format dynamically changed in the first years of its presence, with various features added and removed as seen fit by Adobe.

• Even though some features are now obsolete and not part of the specification, they still remained in some implementations.
Type 1 Font Files

Several files required to load the font, e.g. for Windows it’s

.pfb + .pfm [+ .mmm]

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.mmm</td>
<td>Multiple master Type1 font resource file. It must be used with .pfm and .pfb files.</td>
</tr>
<tr>
<td>.pfb</td>
<td>Type 1 font bits file. It is used with a .pfm file.</td>
</tr>
<tr>
<td>.pfm</td>
<td>Type 1 font metrics file. It is used with a .pfb file.</td>
</tr>
</tbody>
</table>

AddFontResource function, MSDN
Type 1 Multiple Master (MM) fonts

• In 1991, Adobe released an extension to the Type 1 font format called “Multiple Master fonts”.
  • enables specifying two or more “masters” (font styles) and interpolating between them along a continuous range of “axes”.
    • weight, width, optical size, style
  • technically implemented by introducing several new DICT fields and Charstring instructions.
**Type 1 Multiple Master (MM) fonts**

<table>
<thead>
<tr>
<th>Design axis</th>
<th>Dynamic range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>![Light to Black](Light to Black)</td>
</tr>
<tr>
<td>Width</td>
<td>![Condensed to Extended](Condensed to Extended)</td>
</tr>
<tr>
<td>Optical size</td>
<td>![6-point to 72-point designs, shown at the same size for comparison](6-point to 72-point designs, shown at the same size for comparison)</td>
</tr>
<tr>
<td>Style</td>
<td>![Wedge Serif to Slab Serif](Wedge Serif to Slab Serif)</td>
</tr>
</tbody>
</table>

Type 1 Multiple Master (MM) fonts

• Initially supported in Adobe Type Manager (itself released in 1990).
  • first program to properly rasterize Type 1 fonts on screen.
• Not commonly adopted world-wide, partially due to the advent of OpenType.
  • only 30 commercial and 8 free MM fonts released (mostly by Adobe itself).
  • very sparse software support nowadays; however, at least Microsoft Windows (GDI) and Adobe Reader still support it.
OpenType/CFF primer

• Released by Microsoft and Adobe in 1997 to supersede TrueType and Type 1 fonts.

• Major differences:
  • only requires a single font file (.OTF) instead of two or more.
  • previously textual data (such as DICTs) converted to compact, binary form to reduce memory consumption.
  • the Charstring specification significantly extended, introducing new instructions and deprecating some older ones.
## Type 2 Charstring Operators

<table>
<thead>
<tr>
<th>One-byte Type 2 Operators</th>
<th>Two-byte Type 2 Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dec</strong></td>
<td><strong>Hex</strong></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
</tr>
<tr>
<td>10</td>
<td>0a</td>
</tr>
<tr>
<td>11</td>
<td>0b</td>
</tr>
<tr>
<td>12</td>
<td>0c</td>
</tr>
<tr>
<td>13</td>
<td>0d</td>
</tr>
<tr>
<td>14</td>
<td>0e</td>
</tr>
<tr>
<td>15</td>
<td>0f</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
</tr>
</tbody>
</table>

1. Reserved
Type 2 Charstring Operators

• Changes in the Charstring specs:
  • with *global* and *local* subroutines in OpenType, a new *callgsubr* instruction added,
  • multiple new hinting-related instructions introduced (*hstemhm, hintmask, cntrmask*, …),
  • new arithmetic and logic instructions (*and, or, not, abs, add, sub, neg, …*),
  • new instructions managing the stack (*dup, exch, index, roll*),
  • new miscellaneous instructions (*random*),
  • new instructions operating on the transient array (*get, put*),
  • dropped support for OtherSubrs (removed *callothersubr*).
OpenType/CFF limits specified

A good starting point for vulnerability hunting:

The following are the implementation limits of the Type 2 char-string interpreter:

<table>
<thead>
<tr>
<th>Description</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument stack</td>
<td>48</td>
</tr>
<tr>
<td>Number of stem hints (H/V total)</td>
<td>96</td>
</tr>
<tr>
<td>Subr nesting, stack limit</td>
<td>10</td>
</tr>
<tr>
<td>Charstring length</td>
<td>65535</td>
</tr>
<tr>
<td>maximum (g)subrs count</td>
<td>65536</td>
</tr>
<tr>
<td>TransientArray elements</td>
<td>32</td>
</tr>
</tbody>
</table>
Chapter 1: the beginning
FreeType

• Best and most commonly used open-source font rasterization library written in C.
  • Highly efficient and portable.

• Used on billions of devices.
  • Major clients – GNU/Linux, iOS, Android, Chrome OS.

• Supports virtually all existing font formats (BDF, PCF, PFR, OpenType, Type 1, Type 42, TrueType, FON, FNT, ...).
Perfect attack vector?

- A signedness issue leading to arbitrary PostScript operations within an internal structure in Type 1 font handling exploited by comex in 2011 as part of iOS jailbreakme v3.

- Won a Pwnie Award for “Best Client-Side Bug”.

- The security record of the project not all that great in the past, overall.
Fuzzing it myself a bit since 2012 (>50 bugs reported)

Out of bounds read of uninitialized memory in _af_latin_metrics_init_blues

Out of bounds heap-based buffer read by parsing glyph information and bitmaps for BDF fonts

Out of bounds heap-based buffer read when parsing certain SFNT strings by Type42 font parser

Out of bounds heap-based buffer read by loading properties of PCF fonts

Out of bounds heap-based buffer read by attempt to record current cell into the cell table

Out of bounds heap-based buffer read flaw in Type1 font loader by parsing font dictionary entries

Out of bounds heap-based buffer write by parsing BDF glyph information and bitmaps

Out of bounds heap-based buffer write in Type1 font parser by retrieving font’s private dictionary

Out of bounds heap-based buffer read in TrueType bytecode interpreter by executing NPUSHB and NPUSHW instructions

Out of bounds heap-based buffer write by parsing BDF glyph information and bitmaps

Out of bounds heap-based buffer read by parsing BDF font header

Out of bounds heap-based buffer read in the TrueType bytecode interpreter by executing the MIRP instruction

Array index error, leading to out-of-stack-based buffer read by parsing BDF font glyph information

Out of bounds heap-based buffer read by conversion of PostScript font objects

Out of bounds heap-based buffer read flaw by conversion of an ASCII string into a signed short integer by processing BDF fonts

Out of bounds heap-based buffer write by retrieval of advance values for glyph outlines

Integer divide by zero by performing arithmetic computations for certain fonts

Out of bounds heap-based buffer write in the TrueType bytecode interpreter by moving zone2 pointer point

NULL Pointer Dereference in _bf_free_font

Out of bounds read in _bf_parse_glyphs

Out of bounds write in _bf_parse_glyphs

Out of bounds write in _bf_parse_glyphs

Use of uninitialized memory in ps_parser_load_field, t42_parse_font_matrix and t1_parse_font_matrix

Use of uninitialized memory in tt_sbit_decoder_load_bitmap

Use of uninitialized memory in af_latin_metrics_init_blues

Out of bounds read in _bf_parse_glyphs

Out of bounds read in cff_fd_select_get

Out of bounds read in FNT_Load_Glyph

Out of bounds reads in tt_cmap0[0,2,4]_validate

BDF parsing potential heap pointer disclosure

Mac font parsing heap-based buffer overflow due to multiple integer overflows

Mac font parsing heap-based buffer overflow due to integer signedness problems

Mac FOND resource parsing out-of-bounds read from stack

PCF parsing NULL pointer dereference due to 32-bit integer overflow

PCF parsing NULL pointer dereference due to 32-bit integer overflow

SFNT parsing multiple out-of-bounds reads due to integer overflows in “cmap” table handling

WOFF parsing heap-based buffer overflow due to integer overflow

SFNT parsing integer overflow

Sbits parsing potential out-of-bounds read due to integer overflow

Sbits PNG handling heap-based buffer overflow due to integer overflow

Type42 parsing out-of-bounds read in “ps_table_add”

Type42 parsing out-of-bounds read in “tt_cmap4_validate”

Type42 parsing heap-based buffer overflow in “cff_builder_add_point”

CFString parsing heap-based buffer overflow in “cft_StreamTryRead” (embedded BDF loading)

BDF parsing NULL pointer dereference in “_bf_parse_glyphs”

CFF hintmap building stack-based arbitrary out-of-bounds write

SFNT kern parsing out-of-bounds read in “tt_face_load_kern”

TrueType parsing heap-based out-of-bounds read in “tt_face_load_hdmx”

OpenType parsing heap-based out-of-bounds read in “tt_sbit_decoder_load_image”

multiple unchecked function calls returning FT_Error

Type42 parsing invalid free in “t42_parse_sfnts”
Obviously not making everyone happy...

Looks like @j00ru killed a bunch of FreeType 0days: goo.gl/Xd012 I've been preserving some of these for the last several months..
At one point in 2013...

... FreeType actually became *fuzz clean* using my then-current methods.

After a couple of months, I saw this:
A month and a half later

Adobe CFF font rasterizer accepted by FreeType

Last month we announced that Adobe, in collaboration with Google and FreeType, contributed its CFF font rasterizer technology to FreeType. Today we are happy to let everyone know that the Adobe CFF Engine has been accepted by FreeType and the Adobe-enhanced rasterizer is now on by default.

We'd like to thank everyone who tested the Adobe CFF Engine and reported issues during the beta period. The code was released as a "mature" beta but testers did find a few issues and an improved version of the rasterizer is now being delivered to all devices that use the latest version on FreeType (version 2.5.0).
An entire new CFF rasterizer by Adobe!

• Including a lot of complex/interesting code such as Charstring handling.

• Unfortunately, most useful operators not really supported:

```c
case cf2_escGET: /* in spec */
  FT_TRACE4(( " get\n" ));

  CF2_FIXME;
  break;

case cf2_escIFELSE: /* in spec */
  FT_TRACE4(( " ifelse\n" ));

  CF2_FIXME;
  break;

case cf2_escRANDOM: /* in spec */
  FT_TRACE4(( " random\n" ));

  CF2_FIXME;
  break;
```
Let’s give it a go!

• There are still many assumptions to break in the parsing. 😊

• Restarted the fuzzer with .OTF files against the new CFF code.
  • As always, with the library built with AddressSanitizer.

• Initially no results for the first few days.

• But then...
ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fff22b36410
at pc 0x711ffe bp 0x7fff22b35e90 sp 0x7fff22b35e88
READ of size 1 at 0x7fff22b36410 thread T0

#0 0x711ffd in cf2_hintmap_build freetype2/src/cff/cf2hints.c:820
#1 0x6f54e1 in cf2_interpT2CharString freetype2/src/cff/cf2intrp.c:1201
#2 0x6e94f0 in cf2_getGlyphOutline freetype2/src/cff/cf2font.c:456
#3 0x6e5bfe in cf2_decoder_parse_charstrings freetype2/src/cff/cf2ft.c:369
#4 0x6db3e6 in cff_slot_load freetype2/src/cff/cffgload.c:2840
#5 0x69ec8c in cff_glyph_load freetype2/src/cff/cffdrivr.c:185
#6 0x4a52be in FT_Load_Glyph freetype2/src/base/ftobjs.c:726
#7 0x492ec9 in test_load ft2demos-2.5.2/src/ftbench.c:246
#8 0x493cb1 in benchmark ft2demos-2.5.2/src/ftbench.c:216
#9 0x48fdcd in main ft2demos-2.5.2/src/ftbench.c:1011
Bug analysis line by line (src/cff/cf2hints.c)

747:   FT_LOCAL_DEF( void )
748: cf2_hintmap_build( CF2_HintMap hintmap,
749:                      CF2_ArrStack hStemHintArray,
750:                      CF2_ArrStack vStemHintArray,
751:                      CF2_HintMask hintMask,
752:                      CF2_Fixed hintOrigin,
753:                      FT_Bool initialMap )
754: {
755:     FT_Byte* maskPtr;
756:     CF2_Font font = hintmap->font;
757:     CF2_HintMaskRec tempHintMask;
758:     size_t bitCount, i;
759:     FT_Byte maskByte;
...
790:     /* make a copy of the hint mask so we can modify it */
791:     tempHintMask = *hintMask;
792:     maskPtr = cf2_hintmask_getMaskPtr( &tempHintMask );
793:     /* use the hStem hints only, which are first in the mask */
794:     /* TODO: compare this to cffhintmaskGetBitCount */
795:     bitCount = cf2_arrstack_size( hStemHintArray );
Bug analysis line by line (src/cff/cf2hints.h)

46: enum
47: {
48:     CF2_MAX_HINTS = 96 /* maximum # of hints */
49: }
50:
51: /*
52: ...
53: * The maximum total number of hints is 96, as specified by the CFF
54: * specification.
55: ...
56: */
57: typedef struct CF2_HintMaskRec_
58: {
59:     FT_Error* error;
60:     FT_Bool isValid;
61:     FT_Bool isNew;
62:     size_t bitCount;
63:     size_t byteCount;
64:     FT_Byte mask[(CF2_MAX_HINTS + 7) / 8];
65: } CF2_HintMaskRec, *CF2_HintMask;
Bug analysis line by line (src/cff/cf2hints.c)

816:     /* insert hints captured by a blue zone or already locked (higher */
817:     /* priority) */
818:     for ( i = 0, maskByte = 0x80; i < bitCount; i++ )
819:     {
820:         if ( maskByte & *maskPtr )
821:         {
822:             /* expand StemHint into two `CF2 Hint' elements */
823:             CF2_HintRec bottomHintEdge, topHintEdge;
824:             ...
825:             if ( cf2_hint_isLocked( &bottomHintEdge ) ||
826:                 cf2_hint_isLocked( &topHintEdge ) ||
827:                 cf2_blues_capture( &font->blues,
828:                 &bottomHintEdge,
829:                 &topHintEdge ) )
830:             {
831:                 /* insert captured hint into map */
832:                 cf2_hintmap_insertHint( hintmap, &bottomHintEdge, &topHintEdge );
833:             }
834:             *maskPtr &= ~maskByte; /* turn off the bit for this hint */
835:         }
836:     }
837:     {
838:         if ( ( i & 7 ) == 7 )
839:         {
840:             /* move to next mask byte */
841:             maskPtr++;
842:             maskByte = 0x80;
843:         }
844:     }
845:     else
846:         maskByte >>= 1;
847:     }

controlled iteration count
out-of-bounds stack read (ASan crash)
controlled expression value
out-of-bounds stack write
The vulnerability

• Caused by an obvious lack of sanity check of the stem hint count (max. 96).

• Results in out-of-bounds read/write operations relative to a 12-byte local buffer.
  • Makes it possible to clear any chosen bit on the stack.
  • Non-continuous overwrite, can defeat stack cookies and reliably modify the return address (or any other data).

• Quite easily exploitable Remote Code Execution condition.
The Charstring trigger

1 1 hstem 1 1 hstem ... 0 0 vstem cntrmask

• >96 horizontal stems, enough to reach the desired bits on the stack.
• One hstem operator corresponds to one bit.
• Different arguments depending on the desire to clear a specific bit or not.

- a single vertical stem for correctness.
- vulnerability trigger.
For example...

**Instruction stream:** 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem ... 0 0 vstem cntrmask endchar

[Diagram showing the instruction stream and associated memory locations]
For example...

**Instruction stream:**
```
0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem
0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem
0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem
0 0 hstem 0 0 hstem 0 0 hstem ... 0 0 vstem cntrmask endchar
```

0xF75EC280
For example...

**Instruction stream:**

```
0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem ...
```

```
0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem ...
```

```
0 0 vstem cntrmask endchar
```

```
0xF75EC280 1 1 1 1 0 1 1 1 0 1 0 1 0 1 1 0 1 1 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0
```
For example...

**Instruction stream:** 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 0 0 hstem 1 1 hstem 1 1 hstem 1 1 hstem ... 0 0 vstem cntrmask endchar
For example...

Instruction stream: 1 1 hstem 1 1 hstem 1 1 hstem 1 1 hstem 0 0 hstem
1 1 hstem 1 1 hstem 1 1 hstem 0 0 vstem cntrmask endchar
For example...

**Instruction stream:** 1 1 hstem 1 1 hstem 1 1 hstem 0 0 hstem 1 1 hstem 1 1 hstem 0 0 vstem cntrmask endchar
For example...

**Instruction stream:**

```
1 1 hstem 1 1 hstem 0 0 hstem 1 1 hstem 1 1 hstem
1 1 hstem 0 0 vstem cntrmask endchar
```
For example...

**Instruction stream:** 1 1 hstem 0 0 hstem 1 1 hstem 1 1 hstem 1 1 hstem 0 0 vstem cntrmask endchar
For example...

**Instruction stream:** 0 0 hstem 1 1 hstem 1 1 hstem 1 1 hstem 0 0 vstem
cntrmask endchar
For example...

Instruction stream: 1 1 hstem 1 1 hstem 1 1 hstem 0 0 vstem cntrmask endchar
For example...

**Instruction stream:** 1 1 hstem 1 1 hstem 0 0 vstem cntrmask endchar
For example...

**Instruction stream:** `1 1 hstem 0 0 vstem cntrmask endchar`
For example...

**Instruction stream:** 0 0 vstem cntrmask endchar
For example...

**Instruction stream:** cntrmask endchar
For example...

**Instruction stream:** endchar

```
0x005EC280 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 0 1 1 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0
```
For example...

Instruction stream:
A hint in the code? (src/cff/cf2hints.c)

795:     /* TODO: compare this to cffhintmaskGetBitCount */
796:     bitCount = cf2_arrstack_size(hStemHintArray);

• Sadly not the actual root cause of the bug.
  • author seemed to realize that something *might* go wrong here.
  • the extra comparison would only be a safety net (yet an effective one).
  • other similar annotations in the code (*TODO, XXX* etc.) can be indicative of further problems.
The timeline

• Bug originally reported on 25 Feb 2014, patched in git on 28 Feb 2014, fixed in stable (FreeType 2.5.3) on 8 March 2014.

• While the patch was not obvious, the test case stopped reproducing and the crash didn’t pop out during fuzzing anymore.

• We thought that would be the end of it.
Bug rediscovery

• In November 2014, with better input font corpus and mutation algorithms, I restarted my FreeType fuzzing.

• Within minutes, I saw a very familiar crash starting to occur:

==15055==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fff2dc05b30 at pc 0x71134e bp 0x7fff2dc055b0 sp 0x7fff2dc055a8
READ of size 1 at 0x7fff2dc05b30 thread T0
#0 0x71134d in cf2_hintmap_build freetype2/src/cff/cf2hints.c:822
#1 0x7048e1 in cf2_glyphpath_moveTo freetype2/src/cff/cf2hints.c:1606
#2 0x6f5259 in cf2_interpT2CharString freetype2/src/cff/cf2intrp.c:1243
#3 0x6e8570 in cf2_getGlyphOutline freetype2/src/cff/cf2font.c:469
...

[256, 304) 'tempHintMask' <= Memory access at offset 304 overflows this variable
Bug rediscovery – timeline

• Reported again on 21 Nov 2014.

• Turned out to be the very same bug reachable via several unexpected code paths.
  • Remained improperly fixed for ~9 months. 😞

• Another, more complete patch submitted upstream on 4 Dec 2014, shipped in FreeType 2.5.4 on 6 Dec 2014.
Chapter 2: the Charstring research
Adobe Type Manager (ATM)

• Ported to Windows (3.0, 3.1, 95, 98, Me) by patching into the OS at a very low level in order to provide *native* support for Type 1 fonts.

• Windows NT made it *impossible* (?) to continue this practice.
  • Microsoft originally reacted by allowing Type 1 fonts to be converted to TrueType during system installation.
  • In Windows NT 4.0, ATM was added to the Windows kernel as a third-party font driver, becoming ATMFD.DLL.
  • It is there until today, still providing support for PostScript fonts on modern Windows.
Nowadays – shared codebases
There’s some good news...

• Various software only *based* on the same codebase.

• Living in different branches and maintained by different groups of people.

• Received a varied degree of attention from the security community.

• Don’t have to be affected by the exact same set of bugs!
... and there’s some bad news!

- Various software only *based* on the same codebase.
- Living in different branches and maintained by different groups of people.
- Received a varied degree of attention from the security community.
- Don’t have to be affected by the exact same set of bugs!

**Bindiffing anyone?**
Let’s manually audit the Charstring state machine implemented in Adobe Type Manager Font Driver.
Reverse engineering ATMFD.DLL
ATMFD.DLL: basic recon

- As opposed to Microsoft-authored system components, debug symbols for ATMFD.DLL are not available from the Microsoft symbol server.
- We have to stick with just `sub_XXXXX`. 😞
- Perhaps one of the reasons why it was less thoroughly audited as compared to the TTF font handling in `win32k.sys`?
Shared code, shared symbols?

However, since we know that DirectWrite (DWrite.dll) and WPF (PresentationCFFRasterizerNative_v0300.dll) share the same code, perhaps we could use some simple bindiffing to resolve some symbols?
There’s another way

• As Halvar Flake noticed, Adobe released Reader 4 for AIX and Reader 5 for Windows long time ago with symbols.
  
  • this includes the font engine, CoolType.dll.
  
  • the code has not fundamentally changed since then: it’s basically the same with minor patches.
  
  • it is possible to cross-diff them with modern CoolType, ATMFD or other modules to match some symbols, easing the reverse engineering process.
Functions window

Function name
-
- UFLSupport::ClearIncrGlyphList(void)
- UFLSupport::ClearIncrGlyphList(void)
- UFLEnlargePtr
- TypeMatch<s_HandlerType const * , s_Catchable>
- Type1Program::StoreSubroutine(long, uchar *, long)
- Type1Program::StoreNextCharString(char const *)
- Type1Program::GetProgramBase(void)
- Type1Program::GetMMHandler(void)
- Type1Program::GetDataRef(void)
- Type1Program::AllocSubroutines(long)
- Type1Program::AllocCharStrings(long)
ATMFD.DLL: basic recon

• On the bright side, the library is full of debug messages that we can use to find our way in the assembly.
  • variable names, function names, unmet conditions and source file paths!

• Furthermore, there are multiple Type 1 font string literals, too.
ATMFD.DLL: basic recon

Debug messages:

Type 1 string literals:
Where's Waldo?

• It is relatively easy to locate the Charstring processing routine in ATMFD.DLL.

• For one, it contains references to a lot Charstring-related debug strings:

```
.text:0003ECC4 loc_3ECC4: ; CODE XREF: sub_3A1FC+13A7fj
.text:0003ECC4 ; sub_3A1FC+13B0fj
.text:0003ECC4 push offset aFalse ; "false"
.text:0003ECC9 push offset aOperandStackUn ; "operand stack underflow"
.text:0003ECCE push 16Ah
.text:0003ECDD jmp loc_3ECCA
.text:0003ECDD ;
.text:0003ECDD loc_3ECD8: ; CODE XREF: sub_3A1FC+1434fj
.text:0003ECD8 push offset aFalse ; "false"
.text:0003ECD9 push offset aArgumentCount_0 ; "argument count error at otherNEWCOLORS"
.text:0003ECED push 1683h
.text:0003ECEF jmp loc_3F1A2
.text:0003ECEF ;
.text:0003ECEF loc_3ECEC: ; CODE XREF: sub_3A1FC+1441fj
.text:0003ECEC push offset aFalse ; "false"
.text:0003ECF1 push offset aPsstackOverflow ; "psstack overflow at otherNEWCOLORS"
.text:0003ECF6 push 1668h
.text:0003ECFD jmp loc_3F1A2
.text:0003ED08 ;
```
Where’s Waldo?

- Incidentally, the function is also by far the largest one in the whole DLL (20kB):
The interpreter function

• By looking at DirectWrite and WPF, we can see that its caller is named **Type1InterpretCharString**.

• In the symbolized CoolType, the interpreter itself is named **DoType1InterpretCharString**.

• It is essentially a giant *switch-case* statement, handling the different instructions inline.
The interpreter function

```c
BYTE op = *charstring++;
switch (op) {
    case HSTEM:
        ...
        case VSTEM:
        ...
        case VMOVETO:
        ...
    ...
}
```
Why so large?

• The same interpreter is used for both Type 1 and Type 2 (OpenType) Charstrings.
  • Type 1 fonts have access to all OpenType instructions, and vice versa! :o

• The interpreter in ATMFD.DLL still implements

  every single feature

  that was EVER part of the Type 1 / OpenType specs.

• Even the most obsolete / deprecated / forgotten ones.
Let’s get to work.
Charstring vulnerabilities

All of them affected Windows editions up to and including Windows 8.1.
# CVE-2015-0074: Unlimited Charstring execution

<table>
<thead>
<tr>
<th>Impact:</th>
<th>Denial of Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture:</td>
<td>x86, x86-64</td>
</tr>
<tr>
<td>Reproducible with:</td>
<td>Type 1, OpenType</td>
</tr>
<tr>
<td><em>google-security-research</em> entry:</td>
<td>169</td>
</tr>
<tr>
<td>Windows Kernel (ATMFD) affected:</td>
<td><strong>CVE-2015-0074</strong></td>
</tr>
<tr>
<td>DirectWrite affected:</td>
<td>No</td>
</tr>
<tr>
<td>WPF affected:</td>
<td>No</td>
</tr>
<tr>
<td>Adobe CoolType affected:</td>
<td>No</td>
</tr>
</tbody>
</table>
Let’s start simple – when does the interpreter loop stop?

1. when the `ENDCHAR` instruction is encountered.
2. when an error condition is detected during execution of a PostScript command.

There’s no hard limit over the number of instructions executed.

No loop support to exploit this, but there are subroutine calls!
CVE-2015-0074: nested subroutine calls
CVE-2015-0074: impact

• By performing a huge number of computation-heavy instructions, we can reliably and indefinitely consume 100% of one CPU.
  • multiple fonts can be used to block multiple cores.
  • the process cannot be killed, as the thread remains in kernel-mode all the time.
  • the only cure is a hard reboot.

• Remote Denial of Service vulnerability.
  • USB sticks and Explorer’s automatic thumbnailing.
  • any client application using GDI to rasterize OpenType fonts.
  • only meaningful in the Windows kernel; client application DoS not really interesting.
# CVE-2015-0087: out-of-bounds reads from the Charstring stream

<table>
<thead>
<tr>
<th>Potential impact:</th>
<th>Memory disclosure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Practical impact:</td>
<td>Denial of Service</td>
</tr>
<tr>
<td>Architecture:</td>
<td>x86, x86-64</td>
</tr>
<tr>
<td>Reproducible with:</td>
<td>Type 1, OpenType</td>
</tr>
<tr>
<td>google-security-research entry:</td>
<td>174</td>
</tr>
<tr>
<td>Windows Kernel (ATMFD) affected:</td>
<td>CVE-2015-0087</td>
</tr>
<tr>
<td>DirectWrite affected:</td>
<td>No</td>
</tr>
<tr>
<td>WPF affected:</td>
<td>No</td>
</tr>
<tr>
<td>Adobe CoolType affected:</td>
<td>CVE-2015-3095</td>
</tr>
</tbody>
</table>
• The Charstring stream is accessed by the interpreter:
  • at the beginning of the VM execution loop (to read the main opcode),
  • while reading the second byte of the “escape” instruction.
  • while reading a 8/16/32-bit value to be pushed onto the operand stack.
• In none of those cases did it check if the Charstring pointer went beyond the end of the buffer.
• Different memory regions used for different formats: kernel-mode pools (Type 1 fonts), CSRSS.EXE userland heap (OpenType fonts).
Scenario: **CSRSS.EXE** process memory disclosure

- The parser reads garbage, uninitialized or left-over data and reflects them in the form of glyph’s shape.
- Actually observed: with some valid and some empty CharStrings, the empty ones would reuse the memory of valid programs and be rasterized.
- Otherwise, extremely difficult to extract meaningful memory contents this way.
• Scenario: Blue Screen of Death due to unhandled invalid memory access.
  • Only possible with Type 1 fonts, due to ATMFD’s aggressive exception handling.
  • Requires memory to be aligned nearly perfectly with the end of a page boundary.
  • Otherwise, the interpreter will bail out with an error roughly a few bytes past the Charstring.
  • Totally viable to accomplish.
CVE-2015-0087

TRAP_FRAME: af7e6e44 -- (.trap 0xffffffffaf7e6e44)

ErrCode = 00000000

eax=00000000 ebx=00000000 ecx=00420000 edx=000000cd esi=ffffffff edi=af7e7060
eip=9956dec8 esp=af7e6eb8 ebp=af7e75bc iopl=0             nv up ei ng nz na pe cy
cs=0008 ss=0010 ds=0000 es=0023 fs=0000 gs=0023           efl=00010287

ATMFD+0x2bec8:

9956dec8 0fb60a  movzx ecx,byte ptr [edx]  ds:f000:00cd=??
WTF: ATMFD.DLL exception handling

• Most of the ATMFD code processing input data is protected with a generic exception handler.
  • Graciously handles ACCESS_VIOLATION exceptions caused by invalid user-mode memory access.
  • Poor man’s way to maintain system stability?
  • Definitely disrupts dynamic vulnerability detection – if a fuzzer ever hit a condition resulting in access to invalid user-mode memory, the researcher would never know.
**CVE-2015-0088: off-by-x out-of-bounds reads/writes relative to the operand stack**

<table>
<thead>
<tr>
<th>Potential impact:</th>
<th>Memory Disclosure, Remote Code Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Practical impact:</td>
<td>Minor Memory Disclosure</td>
</tr>
<tr>
<td>Architecture:</td>
<td>x86, x86-64</td>
</tr>
<tr>
<td>Reproducible with:</td>
<td>Type 1, OpenType</td>
</tr>
<tr>
<td>google-security-research entry:</td>
<td>175</td>
</tr>
<tr>
<td>Windows Kernel (ATMFD) affected:</td>
<td>CVE-2015-0088</td>
</tr>
<tr>
<td>DirectWrite affected:</td>
<td>No</td>
</tr>
<tr>
<td>WPF affected:</td>
<td>No</td>
</tr>
<tr>
<td>Adobe CoolType affected:</td>
<td>No</td>
</tr>
</tbody>
</table>
CVE-2015-0088

`DoType1InterpretCharString` stack frame (operand stack)

- `VOID *op_sp; @EDI`
- `ULONG op_stk[48];`
- `Saved EBP`
- `Return address`
- Callers’ stack frames

Higher addresses
CVE-2015-0088

• Most Charstring instructions expect a specific number of arguments on the operand stack.

• ATMFD did nothing to verify the assumption before executing the instructions.

• Consequently, we could get the interpreter to access up to three DWORDs directly prior the local `op_stk[48]` array on stack.
CVE-2015-0088

DoType1InterpretCharString stack frame (operand stack)

Higher addresses

CALLERS' stack frames

Saved EBP

Return address

ULONG op_stk[48];

VOID *op_sp; @EDI

...
CVE-2015-0088

Overreading instructions:

1. escape + callothersubr
2. escape + callothersubr + endflex
3. escape + callothersubr + changehints
4. escape + callothersubr + counter{1, 2}
5. escape + add
6. escape + sub
7. escape + mul
8. escape + div2
9. escape + put
10. escape + get
11. escape + ifelse
12. escape + and
13. escape + or
14. escape + eq
15. escape + roll
16. escape + setcurrentpoint
17. escape + load
18. escape + store
Prior to executing each instruction, the interpreter checks:

```c
if (op_sp < &op_stk[0]) {
    // bail out;
}
```

Makes it impossible to disclose kernel stack memory using any of the affected instructions.

- with the exception of `DUP`, which does not decrement the stack pointer.
- a 4-byte memory disclosure of the kernel stack.
- nothing too interesting there on the builds I checked.
CVE-2015-0088

Overwriting instructions:

- escape + not (off-by-1)
- escape + neg (1)
- escape + abs (1)
- escape + sqrt (1)
- escape + index (1)
- escape + exch (2)

Common scheme:

1. pop the operand(s) from stack,
2. perform corresponding calculations,
3. push the operand(s) back to stack.
CVE-2015-0088

• Potentially RCE – in practice, no interesting data stored in the 2 DWORDs directly before the stack on the builds I checked.
  • purely coincidental, but still.

• Illustrative of the general code quality of the interpreter function in ATMFD.DLL.
  • kept my hopes very high at the beginning of the process. 😊
**CVE-2015-0089: memory disclosure via uninitialized transient array**

<table>
<thead>
<tr>
<th>Impact:</th>
<th>Memory disclosure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Architecture:</strong></td>
<td>x86, x86-64</td>
</tr>
<tr>
<td><strong>Reproducible with:</strong></td>
<td>Type 1 (Windows only), OpenType</td>
</tr>
<tr>
<td><em>google-security-research</em> entries:</td>
<td>176, 259, 277</td>
</tr>
<tr>
<td><strong>Windows Kernel (ATMFD) affected:</strong></td>
<td>CVE-2015-0089</td>
</tr>
<tr>
<td><strong>DirectWrite affected:</strong></td>
<td>CVE-2015-1670</td>
</tr>
<tr>
<td><strong>WPF affected:</strong></td>
<td>CVE-2015-1670</td>
</tr>
<tr>
<td><strong>Adobe CoolType affected:</strong></td>
<td>CVE-2015-3049</td>
</tr>
</tbody>
</table>
CVE-2015-0089: the transient array

A temporary DWORD array for Charstring programs, essentially.

4.5 Storage Operators

The storage operators utilize a transient array and provide facilities for storing and retrieving transient array data.

The transient array provides non-persistent storage for intermediate values. There is no provision to initialize this array, except explicitly using the *put* operator, and values stored in the array do not persist beyond the scope of rendering an individual character.
### CVE-2015-0089: transient array size

- In Type 1 fonts, the size can be controlled via a `/lenBuildCharArray` DICT number entry (up to 65535).

- In OpenType fonts:

```markdown
<table>
<thead>
<tr>
<th>Description</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument stack</td>
<td>48</td>
</tr>
<tr>
<td>Number of stem hints (H/V total)</td>
<td>96</td>
</tr>
<tr>
<td>Subr nesting, stack limit</td>
<td>10</td>
</tr>
<tr>
<td>Charstring length</td>
<td>65535</td>
</tr>
<tr>
<td>maximum (g)subsrs count</td>
<td>65536</td>
</tr>
<tr>
<td>TransientArray elements</td>
<td>32</td>
</tr>
</tbody>
</table>
```
CVE-2015-0089: transient array access

- The array can be accessed using a number of instructions in \texttt{ATMFD.DLL}
  (some of them long gone from the official specs):

1. escape + callothersubr + storewv
2. escape + callothersubr + put(2)
3. escape + put
4. escape + callothersubr + get
5. escape + get
6. escape + load
7. escape + store
CVE-2015-0089: transient array allocation

• The array is only allocated on the first access.
  • from the kernel pools in ring-0, or user-mode heap in ring-3.

• What happens if we try to read an entry that has not been previously initialized?

• The specification addresses this matter explicitly.
„If **get** is executed prior to **put** for **i** during execution of the current charstring, the value returned is **undefined**.“

*The Type 2 Charstring Format, Technical Note #5177, Adobe Systems Incorporated, 16 March 2000, p. 27-28*
• In this case, *undefined* means „old bytes from the reused memory region”.
  • the allocation was not zero-ed out prior to letting the Charstring operate on it.

• We can place bits of uninitialized heap/pool memory on the operand stack... so what?
  • the DWORD can easily be *drawn* as a glyph, making it possible to reflect it back to an attacker or use to defeat ASLR.
  • it’s not trivial, but possible thanks to the extensive set of arithmetic / logical instructions supported by the interpreter.
CVE-2015-0089: uninitialized transient array

• With OpenType, one glyph can disclose **32 DWORDs = 128 bytes**.
  • e.g. by representing a 32x32 matrix, with each row/column describing one DWORD and each square one bit.

• With Type 1, one glyph can disclose up to **65536 DWORDs = 256 kB**.

• Possible to disclose memory of Internet Explorer, WPF and the Windows kernel with the same bug.
  • Google Chrome and Mozilla Firefox also use DirectWrite for font rasterization, but the OpenType Sanitiser disallows some of the required Charstring instructions.
  • Another „one bug to rule them all“. 😊
### Synchronization bytes:
```
aa aa aa aa 55 55 55 55 aa aa aa 55 55 55 55
aa aa aa aa 55 55 55 55 aa aa aa 55 55 55 55
aa aa aa aa 55 55 55 55 aa aa aa 55 55 55 55
aa aa aa aa 55 55 55 55 aa aa aa 55 55 55 55
aa aa aa aa 55 55 55 55 aa aa aa 55 55 55 55
aa aa aa aa 55 55 55 55 aa aa aa 55 55 55 55
```

### Disclosed bytes:
```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 6c f7 e6 9a 02 00 00 00 01 a0 00 00
40 dc 52 09 08 ed 01 00 00 dd 52 09 14 00 00 00
f0 df 99 03 01 01 00 00 48 ae 99 03 08 00 00 00
02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
48 ae 99 03 08 00 00 00 78 a6 2d 05 7f 2f 80 2f
02 00 00 00 00 00 00 00 00 78 99 33 05 48 ae 99 03
08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

### Leaked addresses:
- 0x735e1430 [image address]
- 0x0952dc40 [heap address]
- 0x0001ed08 [heap address]
- 0x0952dd00 [heap address]
- 0x0399dfff [heap address]
- 0x0399ae48 [heap address]
- 0x0399ae48 [heap address]
- 0x052da678 [heap address]
- 0x05339978 [heap address]
- 0x0399ae48 [heap address]
- 0x0399add8 [heap address]
DEMO TIME
## CVE-2015-0090: read/write-what-where in LOAD and STORE operators

<table>
<thead>
<tr>
<th>Impact:</th>
<th>Elevation of Privileges / Remote Code Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture:</td>
<td>x86, x86-64</td>
</tr>
<tr>
<td>Reproducible with:</td>
<td>Type 1, OpenType</td>
</tr>
<tr>
<td><code>google-security-research</code> entry:</td>
<td>177</td>
</tr>
<tr>
<td>Windows Kernel (ATMFD) affected:</td>
<td>CVE-2015-0090</td>
</tr>
<tr>
<td>DirectWrite affected:</td>
<td>No</td>
</tr>
<tr>
<td>WPF affected:</td>
<td>No</td>
</tr>
<tr>
<td>Adobe CoolType affected:</td>
<td>No</td>
</tr>
</tbody>
</table>
Back in the „Type 2 Charstring Format” specs from 1998, another storage available to the font programs was defined – the „Registry Object”.

- Related to *Multiple Masters* which were part of the OpenType format for a short while.
- Subsequently removed from the specification in 2000, but ATMFD.DLL of course still supports it.

- Referenced via two new instructions: **STORE** and **LOAD**.
- can transfer data back and forth between the transient array and the Registry.
The Registry provides more permanent storage for a number of items that have predefined meanings. The items stored in the Registry do not persist beyond the scope of rendering a font. Registry items are selected with an index, thus:

0  Weight Vector
1  Normalized Design Vector
2  User Design Vector

The result of selecting a Registry item with an index outside this list is undefined.
The Registry provides more permanent storage for a number of items that have predefined meanings. The items stored in the Registry do not persist beyond the scope of rendering a font. Registry items are selected with an index, thus:

0  Weight Vector
1  Normalized Design Vector
2  User Design Vector

The result of selecting a Registry item with an index outside this list is undefined.
CVE-2015-0090

• Internally, registry items are implemented as an array of `REGISTRY_ITEM` structures, inside a global font state structure.

```c
struct REGISTRY_ITEM {
    long size;
    void *data;
} Registry[3];
```

• Verification of the Registry index exists, but can you spot the bug?

```asm
.text:0003CA35  cmp eax, 3
.text:0003CA38  ja loc_3BEC4
```
CVE-2015-0090: off-by-one in index validation

• An index > 3 condition instead of index >= 3, leading to an off-by-one in accessing the Registry array.

• Using the LOAD and STORE operators, we can trigger the following memcpy() calls with controlled transient array and size:

  memcpy(Registry[3].data, transient array, controlled size);

  memcpy(transient array, Registry[3].data, controlled size);

  provided that Registry[3].size > 0.
CVE-2015-0090: use of uninitialized pointer

- The registry array is part of an overall font state structure.
  - The `Registry[3]` structure is uninitialized during the interpreter run time.
- If we can spray the Kernel Pools such that `Registry[3].size` and `Registry[3].data` occupy a previously controlled allocation, we end up with arbitrary read and write capabilities in the Windows kernel!
CVE-2015-0090

out-of-bound Registry index, culprit of the bug

offset relative to the start of the transient array

vulnerable instruction

/\ \ #\# - | \{ 3 0 0 1 store \} | -

offset relative to the start of Registry item

number of values (DWORDs) to copy
CVE-2015-0090: pointer controlled via kernel pool spraying

PAGE_FAULT_IN_NONPAGED_AREA (50)
Invalid system memory was referenced. This cannot be protected by try-except, it must be protected by a Probe. Typically the address is just plain bad or it is pointing at freed memory.
Arguments:
Arg1: aaaaaaaaa, memory referenced.
Arg2: 00000001, value 0 = read operation, 1 = write operation.
Arg3: 994f8c00, If non-zero, the instruction address which referenced the bad memory address.
Arg4: 00000002, (reserved)
### CVE-2015-0091: pool-based buffer overflow in Counter Control Hints

<table>
<thead>
<tr>
<th>Impact:</th>
<th>Elevation of Privileges / Remote Code Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture:</td>
<td>x86, x86-64</td>
</tr>
<tr>
<td>Reproducible with:</td>
<td>Type 1, OpenType</td>
</tr>
<tr>
<td>google-security-research entries:</td>
<td>178, 249</td>
</tr>
<tr>
<td>Windows Kernel (ATMFD) affected:</td>
<td>CVE-2015-0091</td>
</tr>
<tr>
<td>DirectWrite affected:</td>
<td>No</td>
</tr>
<tr>
<td>WPF affected:</td>
<td>No</td>
</tr>
<tr>
<td>Adobe CoolType affected:</td>
<td>CVE-2015-3050</td>
</tr>
</tbody>
</table>
CVE-2015-0091: passing parameters

- In the „Type 1 Font Format Supplement” document, a mechanism called „Counter Control Hint” was introduced.
- The font can provide an arbitrary number of hint parameters.
  - Packets of max. 22 integers passed via „othersubr 12”.
  - Final ≤ 22 integers passed via a terminating „othersubr 13”.
- Example (argument count, othersubr number):
  
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 22 12 callother
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 22 12 callother
  0 1 2 3 4 5 6 7 8 13 callother
CVE-2015-0091: the trigger

• The kernel allocates an array of constant size for the Counter Control Hint parameters.

• Performs no bounds checking over the total number of arguments received so far.

• With enough „[numbers] 22 12 callother“ sequences, we can easily overflow the pool-based buffer.
  • No need to have them all in verbatim – we can once again use subroutines to save some disk/memory space.
CVE-2015-0091: the trigger

dup 110 ## -| { 1094795585 1094795585 1094795585 1094795585 1094795585 1094795585 1094795585 1094795585 1094795585 1094795585 1094795585 1094795585 1094795585 1094795585 1094795585 1094795585 1094795585 22 12 callother return } |
dup 111 ## -| { 0 6 callother
    110 callsubr 110 callsubr 110 callsubr 110 callsubr
    110 callsubr 110 callsubr 110 callsubr 110 callsubr
    110 callsubr 110 callsubr 110 callsubr 110 callsubr
    110 callsubr 110 callsubr 110 callsubr 110 callsubr
    return } |
    .
    .
    .
dup 114 ## -| { 0 6 callother
    113 callsubr 113 callsubr 113 callsubr 113 callsubr
    113 callsubr 113 callsubr 113 callsubr 113 callsubr
    113 callsubr 113 callsubr 113 callsubr 113 callsubr
    113 callsubr 113 callsubr 113 callsubr 113 callsubr
    return } |
CVE-2015-0091: system bugcheck

PAGE_FAULT_IN_NONPAGED_AREA (50)
Invalid system memory was referenced. This cannot be protected by try-except, it must be protected by a Probe. Typically the address is just plain bad or it is pointing at freed memory.
Arguments:
Arg1: a8e91000, memory referenced.
Arg2: 00000001, value 0 = read operation, 1 = write operation.
Arg3: 9975d22e, If non-zero, the instruction address which referenced the bad memory address.
Arg4: 00000000, (reserved)
**CVE-2015-0092: pool-based buffer underflow due to integer overflow in STOREWV**

<table>
<thead>
<tr>
<th>Impact:</th>
<th>Elevation of Privileges / Remote Code Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture:</td>
<td>x86, x86-64</td>
</tr>
<tr>
<td>Reproducible with:</td>
<td>Type 1</td>
</tr>
<tr>
<td><code>google-security-research</code> entries:</td>
<td>179, 250</td>
</tr>
<tr>
<td>Windows Kernel (ATMFD) affected:</td>
<td>CVE-2015-0092</td>
</tr>
<tr>
<td>DirectWrite affected:</td>
<td>No</td>
</tr>
<tr>
<td>WPF affected:</td>
<td>No</td>
</tr>
<tr>
<td>Adobe CoolType affected:</td>
<td>CVE-2015-3051</td>
</tr>
</tbody>
</table>
• Otherwise known as **othersubr 19**
  
  • or rather „known”, as it’s not documented in any Type 1 specification.
  
  • perhaps Adobe introduced several new OtherSubrs such that specific CFF fonts can be fully converted back to Type 1 with all features?
  
  • interpreters such as FreeType, ATMFD, Adobe Reader still support it.
CVE-2015-0092: the STOREWV operator

• Usage: `<idx> 1 19 callother`

• Requires a `/WeightVector` array of length ≤16 to be present in the Top DICT, e.g.:

  • `/WeightVector [0.00000 0.00000 0.88077 0.11923 0.00000 0.00000 ]` def

• Copies the contents of `WeightVector` into the transient array, starting with index `idx`.

• The index is (obviously) popped from the operand stack, as a signed 16-bit value.
Before copying data, the interpreter checks that the WeightVector array will fit into the transient array at the chosen offset:

```
--op_sp;
int16_t idx = *(op_sp + 1);
if (font->master_designs + idx > font->lenBuildCharArray) {
    return -8;
}
```
CVE-2015-0092: the bounds check

```cpp
if (font->master_designs + idx > font->lenBuildCharArray)
```

- **font->master_designs**: unsigned length of WeightVector, can be 2 – 16.
- **idx**: fully controlled signed 16-bit number.
- **font->lenBuildCharArray**: unsigned length of the transient array (in items).

If **idx** is a negative number \( \geq \text{font->master_designs} \),

the bounds check can be bypassed.
CVE-2015-0092: the underflow

• Suppose `master_designs = 16, idx = -16`.
  • Results in copying 64 bytes to `&transient_array[-16]` → a pool-based buffer underflow.

```c
memcpy(&font->transient_array[idx],
       font->weight_vector,
       font->master_designs * sizeof(DWORD));
```
CVE-2015-0092: the underflow

- **ATMFD.DLL**: corruption of preceding pool headers and potentially previous allocation’s body.

- **Adobe Reader (CoolType.dll)**: corruption of Adobe’s internal allocator headers and potentially previous allocation’s body.
CVE-2015-0092: the bugcheck

KERNEL_SECURITY_CHECK_FAILURE (139)
A kernel component has corrupted a critical data structure. The corruption could potentially allow a malicious user to gain control of this machine.
Arguments:
Arg1: 00000003, A LIST_ENTRY has been corrupted (i.e. double remove).
Arg2: 81be4b54, Address of the trap frame for the exception that caused the bugcheck
Arg3: 81be4a80, Address of the exception record for the exception that caused the bugcheck
Arg4: 00000000, Reserved
## CVE-2015-0093: unlimited out-of-bounds stack manipulation via BLEN operator

<table>
<thead>
<tr>
<th>Impact:</th>
<th>Elevation of Privileges / Remote Code Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture:</td>
<td>x86</td>
</tr>
<tr>
<td>Reproducible with:</td>
<td>Type 1</td>
</tr>
<tr>
<td>google-security-research entries:</td>
<td>180, 258</td>
</tr>
<tr>
<td>Windows Kernel (ATMFD) affected:</td>
<td>CVE-2015-0093</td>
</tr>
<tr>
<td>DirectWrite affected:</td>
<td>No</td>
</tr>
<tr>
<td>WPF affected:</td>
<td>No</td>
</tr>
<tr>
<td>Adobe CoolType affected:</td>
<td>CVE-2015-3052</td>
</tr>
</tbody>
</table>
CVE-2015-0093: the BLEND operator

• Again, related to the forgotten *Multiple Master* fonts.

• Introduced in „The Type 2 Charstring Format” on 5 May 1998.

• Removed from the specs on 16 March 2000:

  Changes in the 16 March 2000 document
  
  • The information on the blend operator, and all references to multiple master fonts, were removed.

• Obviously still supported in a number of engines. 😊
CVE-2015-0093: the BLEND operator

```
blend num(1,1)...num(1,n) num(2,1)...num(k,n) n blend (16)
val1...valn
```

for *k* master designs, produces *n* interpolated result value(s) from \( n \times k \) arguments.

- Pops \( k \times n \) arguments from the stack, where:
  - \( \mathbf{k} \) = number of master designs (length of the \( /\text{WeightVector} \) table).
  - \( \mathbf{n} \) = controlled signed 16-bit value loaded from the operand stack.

- Pushes back \( n \) values to the stack.
The interpreter had a good intention to verify that the specified number of arguments are present on the stack:

```c
case BLEND:
    if ( op_sp < &op_stk[1] || op_sp > &op_stk_end ) // bail out.
    ...
    if ( master_designs == 0 && &op_sp[n] >= &op_stk_end ) // bail out.
    ...
    if ( &op_stk[n * master_designs] > op_sp ) // bail out.
    ...
    op_sp = DoBlend(op_sp, font->weight_vector, font->master_designs, n);
```
CVE-2015-0093: bounds checking

1. Is the stack pointer within the bounds of the stack buffer?

   \[ \text{op\_sp} \geq \text{op\_stk} \land \text{op\_sp} \leq \&\text{op\_stk\_end} \]

2. Is there at least one item (n) on the stack?

   \[ \text{op\_sp} \geq \&\text{op\_sp[1]} \]

3. Are there enough items (parameters) on the stack?

   \[ \&\text{op\_stk[n * master\_designs]} \leq \text{op\_sp} \]

3. Is there enough space left on the stack to push the output parameters?

   \[ \text{master\_designs} \neq 0 \lor \&\text{op\_sp[n]} < \&\text{op\_stk\_end} \]
CVE-2015-0093: debug messages

AtmfdDbgPrint("windows\core\ntgdi\fondrv\otfd\bc\t1interp.c", 6552, "stack underflow in cmdBLEND", "false");

AtmfdDbgPrint("windows\core\ntgdi\fondrv\otfd\bc\t1interp.c", 6558, "stack overflow in cmdBLEND", "false");

AtmfdDbgPrint("windows\core\ntgdi\fondrv\otfd\bc\t1interp.c", 6561, "DoBlend would underflow operand stack", "op_stk + inst->lenWeightVector*nArgs <= op_sp");
CVE-2015-0093: the `DoBlend` function

• Turns out, a negative value of \( n \) passes all the checks!

• Reaches the `DoBlend` function, which:
  • loads the input parameters from the stack,
  • performs the blending operation,
  • pushes the resulting values back.
CVE-2015-0093: the DoBlend function

From a technical point of view, what happens is essentially:

```c
op_sp -= n * (master_designs - 1) * 4
```

which is the result of popping $k*n$ values, and pushing $n$ values back.
• For a negative $n$, no actual popping/pushing takes place.
  • However, the stack pointer (op_sp) is still adjusted accordingly.
  • With controlled 16-bit $n$, we can arbitrarily increase the stack pointer, well beyond the op_stk[ ] array.
    • It is a security boundary: the stack pointer should ALWAYS point inside the one local array.
**CVE-2015-0093: we’re quite lucky!**

- At the beginning of the main interpreter loop, the function checks if `op_sp` is smaller than `op_stk[]`:

  ```c
  if (op_sp < op_stk) {
      AtmfDDbgPrint("windows\core\ntgdi\fondrv\otfd\bc\t1interp.c", 4475, "underflow of Type 1 operand stack", "op_sp >= op_stk");
      abort();
  }
  ```

- It does not check if `op_sp` is greater than the end of `op_stk[]`, making it possible to execute further instructions with the inconsistent interpreter state.
CVE-2015-0093: stack pointer control

- With \(|WeightVector|=16\), we can increase \(op\_sp\) by as much as

\[
32768 * 15 * 4 = 1966080 \ (0x1E0000).
\]

- well beyond the stack area – we could target other memory areas such as pools, executable images etc.

- With \(|WeightVector|=2\), the stack pointer is shifted by exactly \(-n*4\) \((n\) DWORDs), providing a great granularity for out-of-bounds memory access.

  - by using a two-command \(-x\ blend\) sequence, we can set \(op\_sp\) to any offset relative to the \(op\_stk[]\) array.
For example...
CVE-2015-0093

**DoType1InterpretCharString** stack frame (operand stack)

- **VOID *op_sp; @EDI**
- **ULONG op_stk[48];**
- **...**
- **Saved EBP**
- **Return address**
- **Callers’ stack frames**

**Charstring Program**

- `-349`
- `blend`
- `exch`
- `endchar`

349 DWORD distance
CVE-2015-0093

DoType1InterpretCharString stack frame (operand stack)

Saved EBP

Return address

Callers’ stack frames

Charstring Program

-349
blend
exch
endchar
CVE-2015-0093

**DoType1InterpretCharString** stack frame (operand stack)

- VOID *op_sp; @EDI
- ULONG op_stk[48];
- ...

**Charstring Program**

- -349
- blend
- exch
- endchar

Callers’ stack frames

Higher addresses
CVE-2015-0093

**DoType1InterpretCharString** stack frame (operand stack)

- **VOID *op_sp; @EDI**
- **ULONG op_stk[48];**
- **Return address**
- **Saved EBP**
- **Callers’ stack frames**

Charstring Program:

- -349
- blend
- exch
- endchar
CVE-2015-0093

DoType1InterpretCharString stack frame (operand stack)

Callers' stack frames

Charstring Program

-349
blend
exch
endchar
CVE-2015-0093: bugcheck

ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY (fc)
An attempt was made to execute non-executable memory. The guilty driver is on the stack trace (and is typically the current instruction pointer). When possible, the guilty driver's name (Unicode string) is printed on the bugcheck screen and saved in KiBugCheckDriver.

Arguments:

Arg1: 97ebf6a4, Virtual address for the attempted execute.
Arg2: 11dd2963, PTE contents.
Arg3: 97ebf56c, (reserved)
Arg4: 00000002, (reserved)
CVE-2015-0093: impact

• We can use the supported (arithmetic, storage, etc.) operators over the out-of-bounds `op_sp` pointer.
  • Possible to add, subtract, move data around on stack, insert constants etc.
  • Pretty much all the primitives requires to build a full ROP chain.

• The bug enables the creation a 100% reliable Charstring-only exploit subverting all modern exploit mitigations (stack cookies, DEP, ASLR, SMEP, ...) to execute code.
  • Both Adobe Reader and the Windows Kernel were affected.
  • Possible to create a chain of exploits for full system compromise (RCE + sandbox escape) using just this single vulnerability.
CVE-2015-0093: 64-bit

• On 64-bit platforms, the \( n \ast \text{master\_designs} \) expression is cast to \textit{unsigned int} in one of the bounds checking \textit{if statements}:

\[
\text{if } ((\text{uint64})(\&\text{op\_stk} + 4 \ast (\text{uint32})(n \ast \text{master\_designs})) > \text{op\_sp})
\]

• Consequently, the whole check fails for negative \( n \), eliminating the vulnerability from the code.

• Not to worry, there are no 64-bit builds of Adobe Reader.

• In the x64 Windows kernel, there are other font vulnerabilities to exploit for a sandbox escape 😊
DEMO TIME
Chapter 3: font fuzzing
Various approaches to font security

• The Charstring interpreter code was a perfect manual audit candidate.
  • Mostly self-contained, with a single large function to audit.
  • Relatively simple (structurally and semantically) format of processed data – binary encoded PostScript programs.
  • Known problems to look for an assumptions to violate.
  • By design pretty robust against dumb bitstream-based fuzzing.
Various approaches to font security

• That was, however, very unlike general font security research:
  • Vastly complex data structures used to describe shapes, scaling, metrics, kerning etc.
  • Multiple non-obvious relations between various settings and characteristics making up a font.
  • Extensive quantity of code to read and understand, especially difficult with no original source code available.
    • Symbols, structure definitions, comments etc. would be very useful.
Font fuzzing

• Any font security research without fuzzing would be incomplete.

• It’s the most common hotness in low-level infosec.
  • A majority of researchers have done it or considered it at some point.
  • Likewise, a majority of vulnerabilities in the past were probably discovered via fuzzing.

• Best thing is – it still works!
  • Recent example: the Windows Kernel TTF vulnerability used to break out of the Adobe Reader sandbox and win pwn2own 2015 (Keen Team).
Windows kernel font fuzzing

- I’ve been resisting it for years.
  - If so many people have successfully done it in the past, they must have found all the bugs by now, right?

- Finally gave it a shot in May 2015.
  - Dumb fuzzing TrueType and OpenType is fundamentally the same – why not do both?
  - Shared file organization (SFNT structure) and a number of common tables.
Windows kernel font fuzzing – methodology

- No rocket science, took a few simple steps to make the process as effective as possible:

1. Generated a solid initial corpus of .OTF / .TTF font files to maximize code coverage and minimize size.
2. Scaled the fuzzing process to run on several hundreds / thousands of CPU cores.
3. Applied carefully chosen per-table mutation ratios.
4. Used a variety of universal bit and byte-fiddling mutation algorithms and mixed them during fuzzing.
5. Developed a Windows harness to render all (and only) glyphs available in the font at various (but deterministic) point sizes and with various text settings.
6. Mutated and loaded fonts from memory in order to avoid expensive disk I/O operations.
7. Enabled the Special Pools mechanism for win32k.sys and ATMFD.DLL kernel modules to achieve better memory corruption detection rates.
8. Optimized Windows (turned off UI features, disabled services etc.) to reduce unrelated OS overhead.
Windows kernel font fuzzing – initial results

• 7 unique OpenType bugs and 4 TrueType bugs discovered after a few days of running.
  • caused by mutations in various tables: glyf, GPOS, maxp, hmtx, CFF, fpgm
• Initially all scheduled for the August Patch Tuesday.
• But then...
Unexpected security bulletin


MSRC Team 20 Jul 2015 11:09 AM

Today, we released a security bulletin to provide an update for Microsoft Windows. Customers who have automatic updates enabled or apply the update, will be protected.

We recommend customers apply the update as soon as possible, following the directions in the security bulletin.

More information about this bulletin can be found at Microsoft’s Bulletin Summary page.

MSRC Team

<table>
<thead>
<tr>
<th>July 2015</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS15-078</td>
</tr>
<tr>
<td>MS15-078</td>
</tr>
<tr>
<td>MS15-078</td>
</tr>
</tbody>
</table>
Collision #1: Hacking Team

• In the Hacking Team data dump, a 2nd OpenType font exploit was found targeting the Windows kernel for a sandbox escape.
  • discovered in the dump and reported to Microsoft by FireEye and TrendMicro.

• The bug was specifically in .OTF file parsing implemented by the kernel driver.
  • Resulted in a pool-based buffer overflow, facilitating a privilege escalation.

• Interesting data point: it was the most commonly hitting OTF crash during my fuzzing session.
  • basically trivial to discover.
Collision #1: culprit of the vulnerability

LPVOID lpBuffer = EngAllocMem(8 + GPOS.Class1Count * 0x20);
if (lpBuffer != NULL) {
    // Copy the first element.
    memcpy(lpBuffer + 8, ..., 0x20);

    // Copy the remaining Class1Count - 1 elements.
    ...
}

• The driver would assume that `Class1Count` (a field inside of the GPOS table) would be always non-zero.
• If it was actually zero, the code would overflow the allocated buffer by 32 (0x20) bytes.
• Since the field is a 16-bit integer, it was sufficient to set the specific 2 bytes to 0x0 in the file to trigger the condition.
Collision #1: vulnerability exploitation

- Details of the vulnerability exploitation can be found at a Chinese blog ([link]), as discussed by MJ0011 and pgboy of 360 Vulcan Team.
- The exploit was later ported to Windows 8.1 64-bit by Cedric Halbronn of NCC Group ([link]).
- In essence:
  1. Massage the kernel pool to put a `CHwndTargetProp` object directly after the overflown buffer, having its vtable corrupted and redirected into user space memory.
  2. Use another `win32k.sys` vulnerability to leak the driver’s base address.
  3. Trigger the corrupted vtable to get RIP control, hijack RSP through a stack pivot.
  4. Invoke a ROP chain to disable SMEP.
  5. Execute a privilege escalation shellcode from user-mode memory and return.
Further fixes – August 2015 Patch Tuesday

• Remaining ten vulnerabilities were fixed by Microsoft three weeks later during the regular Patch Tuesday cycle.
• Another bug collision became apparent, with Keen Team this time (CVE-2015-2455).
• It was one of the issues used during pwn2own, according to ZDI.
Collision #2: culprit of the vulnerability

The problem existed in the implementation of a TrueType **IUP** instruction.

**Interpolate Untouched Points through the outline**

IUP[a]

<table>
<thead>
<tr>
<th>Code Range</th>
<th>0x30 - 0x31</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0: interpolate in the y-direction</td>
</tr>
<tr>
<td></td>
<td>1: interpolate in the x-direction</td>
</tr>
</tbody>
</table>

Pops —

Pushes —

Uses zp2, freedom_vector, projection_vector

Considers a glyph contour by contour, moving any untouched points in each contour that are between a pair of touched points. If the coordinates of an untouched point were originally between those of the touched pair, it is linearly interpolated between the new coordinates, otherwise the untouched point is shifted by the amount the nearest touched point is shifted.

This instruction operates on points in the glyph zone pointed to by zp2. This zone should almost always be zone 1. Applying IUP to zone 0 is an error.

*source: Microsoft, The TrueType instruction set, Part 2*
Collision #2: culprit of the vulnerability

This instruction operates on points in the glyph zone pointed to by zp2. This zone should almost always be zone 1. **Applying IUP to zone 0 is an error.**
Collision #2: vulnerability trigger

```
PUSH[ ] /* 1 value pushed */
0
SZP2[ ] /* SetZonePointer2 */
IUP[0] /* InterpolateUntPts */
```

• It’s sufficient to execute the IUP instruction with zp2 (zone pointer 2) set to 0 to trigger the bug.
  • trivial to come by – a single bit flip is enough to change the SZP2 / SZPS instruction argument from 1 to 0.

• The instruction assumed it was operating on zone 1, but iterated over zone’s 0 points, leading to a multitude of out-of-bounds reads and writes, corrupting the pool memory area.
Collision #2: conclusions and data points

1. Official specifications can really hint – or even explicitly point out – where things can go wrong in file format handling.
   • already happened several times during the research, although only checked post-factum.

2. With such a trivial trigger, how did the vulnerability even make it until 2015?

3. Once again, the collided bug was the most frequently hitting TTF crash.
Font fuzzing – the future

• There’s still a lot to be done to improve font engines’ robustness through fuzzing.
  • Less dumb, more structure-aware mutation algorithms.
  • Fully code coverage driven fuzzing.
  • Better memory corruption detection ratios (e.g. against the aggressive driver exception handling).
  • Fully generative fuzzing for certain portions of the specs (e.g. the TrueType VM).

• More fixes for fuzzed out bugs are still coming up, too! 😊
Some final thoughts

• Despite a lot of attention, font vulnerabilities are still not extinct – I’d rather say the opposite.

• It’s doubtful they ever completely will – the only winning move is to remove font processing from all privileged security contexts.
  • Microsoft is already doing this with the introduction of a separated user-land font driver in Windows 10.
Some final thoughts

- Shared native codebases still exist, and are immensely scary in the context of software security.
  - especially those processing complex file formats written 20-30 years ago.
- Even in 2015 – the era of high-quality mitigations and security mechanisms, **one good bug** still suffices for a complete system compromise.
Thanks!

@j00ru
http://j00ru.vexillium.org/
j00ru.vx@gmail.com
ADDENUM:
A short recap on font history
Early 1980’s
Early 1980’s

Bitmap (raster) fonts, mostly hardcoded

MS-DOS, 1981

UNIX, 1984

86-DOS, 1980
Early 1980’s

Mac OS, 1984
Various raster font formats

- Multiple bitmap font file formats developed in the past:
  - Portable Compiled Format (PCF)
  - Glyph Bitmap Distribution Format (BDF)
  - Server Normal Format (SNF)
  - DECWindows Font (DWF)
  - Sun X11/NeWS format (BF, AFM)
  - Microsoft Windows bitmapped font (FON)
  - Amiga Font, ColorFont, AnimFont
  - ByteMap Font (BMF)
  - PC Screen Font (PSF)
  - Packed bitmap font bitmap file for TeX DVI drivers (PK)

Still supported by FreeType
Still supported by Microsoft Windows
Adobe PostScript fonts

• In 1984, Adobe introduced two outline font formats based on the PostScript language (itself created in 1982):
  • Type 1, which may only use a specific subset of PostScript specification.
  • Type 3, which can take advantage of all of PostScript’s features.

• Originally proprietary formats, with technical specification commercially licensed to partners.
  • Only publicly documented in March 1990, following Apple’s work on an independent font format, TrueType.
Type 1 Fonts

Adobe Type 1 Font Format, Adobe Systems Incorporated
Figure 2a. *Organization of a Type 1 font program*

Adobe Type 1 Font Format, Adobe Systems Incorporated
Adobe kept the details of their hinting scheme undisclosed and used a (simple) encryption scheme to protect Type 1 outlines and hints, which still persists today (although the encryption scheme and key has since been published by Adobe).

WTF #1: „encryption”

Type 1 font programs incorporate two types of encryption: character string encryption and exec encryption.

The encoded charstrings are encrypted first. This level of encryption is called *charstring encryption*; Type 1 BuildChar works only with encoded and encrypted charstrings. A section of the Type 1 font program, including the Private and CharStrings dictionaries, is again encrypted using another layer of encryption called *exec encryption*. This layer of encryption is intended to protect some of the hint information in the Private dictionary from casual inspection, and it coincidentally provides an ASCII hexadecimal form of this part of the font program so that it can be passed through communication channels that accept only 7-bit ASCII.

*Adobe Type 1 Font Format, Adobe Systems Incorporated*
WTF #1: “encryption”

This encryption step can be performed by the following C language program, where \( r \) is initialized to the key for the encryption type:

```c
unsigned short int r;
unsigned short int c1 = 52845;
unsigned short int c2 = 22719;

unsigned char Encrypt(plain) unsigned char plain;
{unsigned char cipher:
  cipher = (plain ^ (r>>8));
  r = (cipher + r) * c1 + c2;
  return cipher;
}
```

The decryption step can be performed by the following C language program, where \( r \) is initialized to the key for the encryption type:

```c
unsigned short int r;
unsigned short int c1 = 52845;
unsigned short int c2 = 22719;

unsigned char Decrypt(cipher) unsigned char cipher;
{unsigned char plain:
  plain = (cipher ^ (r>>8));
  r = (cipher + r) * c1 + c2;
  return plain;
}
```

Adobe Type 1 Font Format, Adobe Systems Incorporated
WTF #2

Because Type 1 font programs were originally produced and were carefully checked only within Adobe Systems, **Type 1 BuildChar was designed with the expectation that only error-free Type 1 font programs would be presented to it.** Consequently, **Type 1 BuildChar does not protect itself against data inconsistencies and other problems.**

Adobe Systems Incorporated 1993,
Adobe type 1 font format, Third printing, Version 1.1,
Type 1 Multiple Master (MM) fonts

• In 1991, Adobe released an extension to the Type 1 font format called “Multiple Master fonts”.
  • enables specifying two or more “masters” (font styles) and interpolating between them along a continuous range of “axes”.
    • weight, width, optical size, style
  • technically implemented by introducing several new DICT fields and Charstring instructions.
Type 1 Multiple Master (MM) fonts

- Initially supported in Adobe Type Manager (itself released in 1990).
  - first program to properly rasterize Type 1 fonts on screen.
- Not commonly adopted world-wide, partially due to the advent of OpenType.
  - only 30 commercial and 8 free MM fonts released (mostly by Adobe itself).
  - very sparse software support nowadays; however, at least Microsoft Windows (GDI) and Adobe Reader still support it.
Adobe Type Manager (ATM)

• Ported to Windows (3.0, 3.1, 95, 98, Me) by patching into the OS at a very low level in order to provide *native* support for Type 1 fonts.

• Windows NT made it *impossible* (?) to continue this practice.
  • Microsoft originally reacted by allowing Type 1 fonts to be converted to TrueType during system installation.
  • In Windows NT 4.0, ATM was added to the Windows kernel as a third-party font driver, becoming ATMFD.DLL.
  • It is there until today, still providing support for PostScript fonts on modern Windows.
Early 1990’s

• Also in 1991, Apple designed a completely new outline font format called TrueType.
  • Based on quadratic Bézier curves.
  • Offered an extensive virtual machine with a programming language for hinting, among other improvements in relation to Type 1 fonts.
  • First supported in Mac OS System 7 released in May 1991.
  • Licensed to Microsoft for free to ensure wide adoption.
    • It is generally the same code rasterizing fonts on your Windows today.
## TrueType SFNT format

<table>
<thead>
<tr>
<th>Tag</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmap</td>
<td>Character to glyph mapping</td>
</tr>
<tr>
<td>head</td>
<td>Font header</td>
</tr>
<tr>
<td>hhea</td>
<td>Horizontal header</td>
</tr>
<tr>
<td>hmtx</td>
<td>Horizontal metrics</td>
</tr>
<tr>
<td>maxp</td>
<td>Maximum profile</td>
</tr>
<tr>
<td>name</td>
<td>Naming table</td>
</tr>
<tr>
<td>OS/2</td>
<td>OS/2 and Windows specific metrics</td>
</tr>
<tr>
<td>post</td>
<td>PostScript information</td>
</tr>
<tr>
<td>cvt</td>
<td>Control Value Table</td>
</tr>
<tr>
<td>fpgm</td>
<td>Font Program</td>
</tr>
<tr>
<td>glyf</td>
<td>Glyph data</td>
</tr>
<tr>
<td>loca</td>
<td>Index to location</td>
</tr>
<tr>
<td>prep</td>
<td>CVT Program</td>
</tr>
</tbody>
</table>
Mid 1990’s

• **1994**: Apple extended TrueType with the launch of TrueType GX.
  - Added extra SFNT tables to enable *morphing* (similar to Adobe’s MM technology) and replacing sequences of characters with different designs.
  - Not widely adopted, now part of *Apple Advanced Typography* (AAT).

• **1994**: Microsoft failed to license TrueType GX and started working on a new format, *TrueType Open*.

• **1996**: Adobe joined Microsoft in these efforts, in order to create technology which would supersede both TrueType and Type 1 fonts. It was called *OpenType*.
The OpenType format

• Uses the same SFNT general structure as TrueType.
  • Requires several new tables.

• Can specify glyph outlines in either the old TrueType format ("glyf" table) or a new „Compact Font Format” (CFF).
  • CFF is essentially a compact (binary) representation of Type 1 fonts, with some additional features and an updated Charstring language (Type 2).
OpenType support

- External Adobe Type Manager was required for .OTF files on Windows 95, 98, NT and Me.
- The ATMFD.DLL library with OpenType support is bundled in the default installation since Windows 2000.
- Adobe used the same implementation in their other products (e.g. the CoolType library).
- Implementation for basic features of OTF followed in FreeType, Apple products and other software.
- OpenType became the 2nd most commonly used font format world-wide.
Late 1990’s – today

• No groundbreaking revolution since the introduction of OpenType.

• The standard has been evolving, with latest specification being version 1.6 released in 2009.

• Vendors started to make use of OTF extensibility to implement a number of new features, often with no collaboration with other major actors.
Late 1990’s – today

• Apple introduced tags enabling more advanced font features, supported by the AAT (*Apple Advanced Typography*) software in OS X.

• Microsoft introduced new math tables supported by Office, Windows 8 (RichEdit 8.0) and Gecko, among others.

• Apple, Microsoft and Google have proposed three different extensions to add support for colored Emoji fonts; each suggesting the use of different tables / formats.

• Mozilla and Adobe have proposed adding full SVG support to OpenType.

• Many more examples.
Today

<table>
<thead>
<tr>
<th>Format</th>
<th>Supported by</th>
</tr>
</thead>
<tbody>
<tr>
<td>.FON, .FNT bitmap fonts</td>
<td>Windows, FreeType</td>
</tr>
<tr>
<td>.PFB, .PFM, .MMM Type 1 fonts</td>
<td>Windows, Adobe Reader, FreeType, Java</td>
</tr>
<tr>
<td>.TTF, .TTC TrueType fonts</td>
<td>Windows, OS X, Adobe Reader, Adobe Flash, FreeType, DirectWrite, Java</td>
</tr>
<tr>
<td>.OTF OpenType Fonts</td>
<td>Windows, OS X, Adobe Reader, Adobe Flash, FreeType, DirectWrite, Java</td>
</tr>
<tr>
<td>Other, unpopular formats</td>
<td>...</td>
</tr>
</tbody>
</table>

- Most UNIX-based software (*GNU/Linux, iOS, Android, ChromeOS*) make use of the FreeType library.

- A number of Windows client programs (*Office, Explorer*, some web browsers) use the builtin Windows font support or DirectWrite.