Project Zero

News and updates from the Project Zero team at Google

Thursday, January 30, 2020

Part II: Returning to Adobe Reader symbols on macOS

Posted by Mateusz Jurczyk, Project Zero

In a blog post titled "The story of Adobe Reader symbols" published in October 2019, I presented an analysis of the debug symbols shipped with some older versions of Adobe Reader for Unix-family systems released between 1997-2013. Such symbols can prove immensely useful for both understanding the inner workings of various Reader components while reverse engineering, and for automated tasks such as crash classification and deduplication in fuzzing projects. In that blog post, I only briefly mentioned the macOS Reader packages as containing up-to-date symbols for the JP2K and 3D components, but missing any other interesting information beyond that. Upon a closer and more thorough inspection of the Mac releases, I have discovered that in fact more debug metadata made its way to the publicly available installers, some of which is unique (i.e. not present in other symbol sources) and/or more recent than the latest Unix symbols. Considering the added value of this new data, I wish to share it with you here in this follow-up post. As a reminder, legacy builds of Adobe Reader dating back to the '90s can be downloaded from the official ftp.adobe.com FTP server and the corresponding paths on the ardownload.adobe.com HTTP server.

Review of core Acrobat Mach-O images

Below is a breakdown of the six core components of Reader for macOS, and whether they included symbols in each of the existing software releases. For brevity and readability, the table starts with version 7.x from 2005, as this is where the debug information started making first appearances:

	Adobe Reader for Mac OS					
	7.x	8.x	9.x	10.x	11.x	DC
	2005-2009	2007-2011	2008-2013	2011-2014	2012-2017	2015-2019
Acrobat	Stripped	Not stripped [1]	Stripped	Stripped	Stripped	Stripped
AGM	Stripped	Not stripped [1]	Not stripped [2]	Stripped	Stripped	Not stripped [3]
CoolType	Stripped	Not stripped [1]	Not stripped [2] (private)	Stripped	Stripped	Stripped
BIB	Stripped	Not stripped [1]	Not stripped [2]	Stripped	Stripped	Stripped
JP2K	Stripped	Not stripped	Not stripped	Not stripped	Not stripped	Not stripped
3D	Not stripped (private)	Not stripped	Not stripped	Not stripped	Not stripped	Not stripped

[1] See detailed 8.x version breakdown
[2] See detailed 9.x version breakdown
[3] Up to version 15.016.20045, stripped afterwards

As mentioned in the annotations, the presence of symbols for some modules varied greatly between minor versions of the 8.x and 9.x releases (the reason for which is unclear to me). Detailed tables for these two major versions are shown below:

	Adobe Reader 8.x for Mac OS			
	8.0, 8.1.x, 8.2.0, 8.2.1	8.2.2, 8.2.3	8.2.4	8.2.5, 8.2.6, 8.3.x
Acrobat	Stripped	Not stripped	Stripped	Not stripped
AGM	Stripped	Not stripped	Not stripped	Not stripped
CoolType	Stripped	Not stripped	Stripped	Not stripped
BIB	Stripped	Not stripped	Not stripped	Not stripped
JP2K	Not stripped	Not stripped	Not stripped	Not stripped
3D	Not stripped	Not stripped	Not stripped	Not stripped

	Adobe Reader 9.x for Mac OS				
	9.0, 9.1.x, 9.2, 9.3.x, 9.4.0, 9.4.1	9.4.2, 9.4.3	9.4.4	9.4.5	9.4.6, 9.5.x

Search This Blog

Search

Pages

- About Project Zero
- Working at Project Zero
- 0day "In the Wild"
- Vulnerability Disclosure FAQ

Archives

2020

- Part II: Returning to Adobe Reader symbols on macO... (Jan)
- Remote iPhone Exploitation Part 3: From Memory Cor... (Jan)
- Remote iPhone Exploitation Part 2: Bringing Light ... (Jan)
- Remote iPhone Exploitation Part 1: Poking Memory v... (Jan)
- Policy and Disclosure: 2020 Edition (Jan)

2019

- Calling Local Windows RPC Servers from .NET (Dec)
- SockPuppet: A Walkthrough of a Kernel Exploit for ... (Dec)
- Bad Binder: Android In-The-Wild Exploit (Nov)
- KTRW: The journey to build a debuggable iPhone (Oct)
- The story of Adobe Reader symbols (Oct)
- Windows Exploitation Tricks: Spoofing Name... (Sep)
- A very deep dive into iOS Exploit chains found in ... (Aug)
- In-the-wild iOS Exploit Chain 1 (Aug)
- In-the-wild iOS Exploit Chain 2 (Aug)
- In-the-wild iOS Exploit Chain 3 (Aug)
- In-the-wild iOS Exploit Chain 4 (Aug)
- In-the-wild iOS Exploit Chain 5 (Aug)
- Implant Teardown (Aug)
- JSC Exploits (Aug)
- The Many Possibilities of CVE-2019-8646 (Aug)
- Down the Rabbit-Hole... (Aug)
- The Fully Remote Attack Surface of the iPhone (Aug)
- Trashing the Flow of Data (May)
- Windows Exploitation Tricks: Abusing the User-Mode... (Apr)
- Virtually Unlimited Memory: Escaping the Chrome Sa... (Apr)
- Splitting atoms in XNU (Apr)
- Windows Kernel Logic Bug Class: Access Mode Mismat... (Mar)
- Android Messaging: A Few Bugs Short of a Chain (Mar)
- The Curious Case of Convexity Confusion

Acrobat	Stripped	Stripped	Stripped	Stripped	Stripped
AGM	Not stripped	Stripped	Stripped	Not stripped	Not stripped
CoolType	Not stripped (private)	Stripped	Not stripped (private)	Not stripped (private)	Not stripped (private)
BIB	Not stripped	Stripped	Stripped	Stripped	Not stripped
JP2K	Not stripped	Not stripped	Not stripped	Not stripped	Not stripped
3D	Not stripped	Not stripped	Not stripped	Not stripped	Not stripped

Clearly, there is a lot of useful information hidden in the Reader Mach-O executables, with versions 8.x and 9.x being the most ripe with debug data. I find the following pieces of information to be the most useful and unique compared to the data previously extracted from the Unix files:

- Private symbols in the stabs format for the 3D module (version 7.x) and the CoolType font engine (most 9.x versions). This includes not only function and global variable names, but also source file paths, source line numbers corresponding to blocks of assembly code, local variable names, definitions of structures, enums etc. This kind of metadata provides us with some extremely detailed and accurate insight into how the code works. To my best knowledge, such private symbols for these two libraries are not known to be available from any other public source and may therefore be a great assistance to anyone interested in their inner workings.
- The symbols for the Acrobat, AGM, CoolType and BIB modules in versions 8.x and 9.x are generally more recent than the most recent symbols on the Unix side. Furthermore, for the purpose of crash deduplication or root cause analysis, it is arguably easier to run Reader 9 on macOS than Reader 9 on SunOS/Solaris, or Reader 7 on AIX/HP-UX, simply because it is the more common platform.
- Lastly, the AGM component was not stripped in the DC versions between 15.006.30033 (March 2015) to 15.016.20045 (June 2016), which is also a much more recent source of information than what was previously accessible (AIX/HP-UX symbols for versions 7.x from 2007).

Out of all of them, I believe that the private CoolType symbols carry the most interesting and practical information. What makes it especially attractive is the fact that it is not only applicable to Reader and other Adobe products, but also to all other libraries and system components which share the same font rasterization code – such as the Windows kernel drivers (win32k.sys, atmfd.dll), the Windows DirectWrite library (dwrite.dll), the macOS font libraries (libTrueTypeScaler.dylib, libType1Scaler.dylib), and the JRE t2k font rasterizer (libt2k.so etc.). Let's have a deeper look into the kind of data found in the debug builds of AdobeCoolType and how we can extract it and use it in a meaningful way.

Digging into the CoolType symbols

In total, I have found 17 unique AdobeCoolType files in various versions of Reader 9.x which contain debug data in the stabs format. The format is a largely textual one, so if we glance at the end of the analyzed file, we should see data similar to the following dump:

```
parseSeacComponent:f(0,1).h:P(0,17).stdcode:P(0,1).offset:r(0,3)....t1cParse
:F(0,1).offset:p(0,3).aux:p(0,27).glyph:p(0,28).h:(0,19).offset:r(0,3).aux:r
(0,27).glyph:r(0,28)....t1cDecrypt:F(0,1).lenIV:p(0,1).length:p(2,29).cipher
:p(4,36).plain:p(4,36).lenIV:r(0,1).length:r(2,29).r:r(0,9).cnt:(0,3)..c1:(0
,11).....t1cUnprotect:F(0,1).lenIV:p(0,1).length:p(2,29).cipher:p(4,36).plai
n:p(4,36).lenIV:r(0,1).length:r(2,29).single:r(0,11).r1:(0,9).r2:(0,9).cnt:(
0,3)..c1:r(0,11).....:t(0,35)=*(0,36).ctlVersionCallbacks:t(0,36)=(2,19)._er
rstrs.4964.t1cErrStr:F(4,36).errcode:p(0,1).errstrs:V(0,37).errcode:r(0,1)..
.:t(0,37)=ar(9,27);0;18;(4,36)...
```

That particular snippet shows names from the open-source AFDKO library, which is compiled into CoolType. Other parts of the debug section reveal more interesting and less public data, such as the detailed stack layout of `Type1InterpretCharString`, the main Postscript CharString interpreter function shown to be subject to a number of bugs in 2015:

```
Type1InterpretCharString:F(5,29).inst:p(38,37).pProcs:p(41,28).pCharDataDesc
:p(0,48).pCharIO:p(41,15).pRunData:p(41,13).pPathProcs:p(0,49).pT1Args:p(0,5
0).etod_path:(38,117).etod_arg:(45,3).status:(0,51).temp_flags:(5,18).did_re
try:(5,1)..moveto_lineto_index:r(5,29)...Exception:(13,3)..pPathProcs:(42,6)
..substack:(0,52).psstack:(0,53).flexCds:(0,54).hints:(0,55).hint3:(0,56).vS
temCount:(5,29).op_stk:(41,10).op_sp:r(5,37).sp:r(38,46).cp:(5,39).wid:(5,39
).tempfcd:(5,39).dcp:(5,39).delta:(5,39).i:r(5,15).indx:(5,15).sby:(5,35).dm
in:(5,35).subindex:(5,15).tfmLockPt:(0,57).flags:(5,18).accentClipBBox:(5,47
).pTempClipBBox:(42,5).prevcp:(5,39).initcp:(5,39).
```

The amount of information is stunning – within a 6.35 MB AdobeCoolType file from Reader 9.5.5, around

1.78 MB is consumed just by debug strings of various kinds. Unfortunately, IDA Pro or Ghidra don't seem to parse stabs, so while we will see the names of functions and globals in these disassemblers, types (structures, unions, enums, typedefs) and local variable names unfortunately won't be imported. If I missed some way to have these symbols loaded into the database, let me know!

The two tools I have successfully used to load stabs are the old OS X objdump and gdb utilities, with the latter being especially powerful. In my test environment, I have Adobe Reader 9.5.5 installed on Mac OS 10.6.8 (Snow Leopard). When I then start Reader under gdb, I see the following output:

```
tests-Mac:~ test$ gdb /Applications/Adobe\ Reader\ 9/Adobe\
Reader.app/Contents/MacOS/AdobeReader

[...]

(gdb) r
Starting program: /Applications/Adobe Reader 9/Adobe
Reader.app/Contents/MacOS/AdobeReader
Reading symbols for shared libraries
.+++.....................................................................
.................. done
Reading symbols for shared libraries ...

[...]

warning: Could not find object file
"/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/build/MT
_Release_Symbols/cooltype5.build/Universal/MT_Release_Symbols.build/Objects-
normal/i386/CT_CTS_OpenType.o" - no debug information available for
"/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../..
/source/cooltype/source/cts/cts_glue/CT_CTS_OpenType.cpp".

warning: Could not find object file
"/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/build/MT
_Release_Symbols/cooltype5.build/Universal/MT_Release_Symbols.build/Objects-
normal/i386/CT_CTS_StreamFromBuffer.o" - no debug information available for
"/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../..
/source/cooltype/source/cts/cts_glue/CT_CTS_StreamFromBuffer.c".

....... done
Reading symbols for shared libraries . done
Reading symbols for shared libraries . done
Reading symbols for shared libraries . done
Reading symbols for shared libraries . done
Reading symbols for shared libraries . done
Reading symbols for shared libraries . done
Reading symbols for shared libraries . done
Reading symbols for shared libraries . done
Reading symbols for shared libraries .. done
Reading symbols for shared libraries . done
Reading symbols for shared libraries . done
Reading symbols for shared libraries . done
Reading symbols for shared libraries ... done
```

The debugger successfully loads the application and embedded symbols, but it complains about missing object and source files – expecting them to exist as such a build would typically be used for debugging on a developer's machine. At this point, we can conveniently dump various types of general information with the following gdb commands:

- info functions – list all functions found in the loaded modules, or print information about a specific function
- info line – print information about a specific line in the source code file (i.e. which assembly regions it corresponds to)
- info sources – list source file names
- info types – list all known types
- info variables – list all global and static variable names

This already gives us plenty of information; for example, let's go back to our favourite function and look it up:

```
(gdb) info functions Type1InterpretCharString
All functions matching regular expression "Type1InterpretCharString":

File
```

```
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/buildch/sources/t1interp.c:
IntX Type1InterpretCharString(PFontInst, BCProcs *, T1CharDataDesc *, CharIO
*, RunRec *, PathProcs * volatile, T1Args * volatile);
(gdb)
```

We can set a breakpoint on it and learn in which line in the source code it is declared:

```
(gdb) b Type1InterpretCharString
Breakpoint 1 at 0x7d853e5a: file
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/buildch/sources/t1interp.c, line 7037.
(gdb)
```

We can get some more information on that specific line:

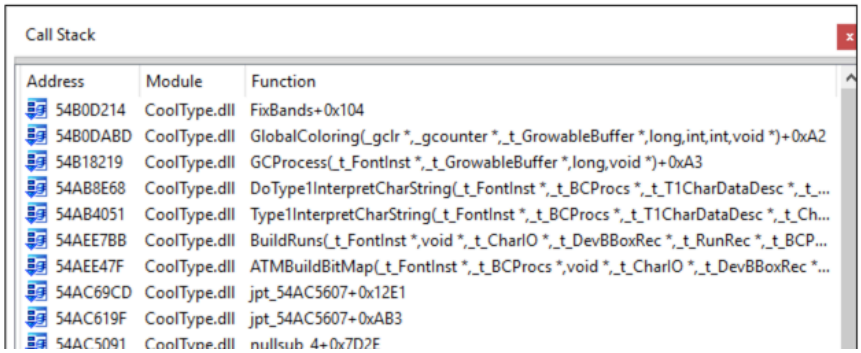
```
(gdb) info line
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/buildch/sources/t1interp.c:7037
Line 7037 of
"/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../..
/source/cooltype/source/atm/base/buildch/sources/t1interp.c" starts at
address 0x7d853e5a <Type1InterpretCharString+77> and ends at 0x7d853e80
<Type1InterpretCharString+115>.
(gdb)
```

Finally, we can check the definition of one of the function argument types, for example the `PFontInst` structure pointer:

```
(gdb) ptype PFontInst
type = struct _t_FontInst {
    Card16 gridRatio;
    Card16 xAlign;
    Fixed flatnessFactor;
    FixMtx tfmMtx;
    Card32 lenStemSnapV;
    Fixed stemSnapV[48];
    Card32 lenStemSnapH;
    Fixed stemSnapH[48];
   [...]
    Fixed accentHintedWidth;
    Fixed accentCSadjust;
    Fixed accentDSdx;
    Fixed blueStdHW;
    Card32 instructions;
    boolean boostNeeded;
    Card16 boostPhase;
    Card16 fontType;
} *
(gdb) print sizeof(struct _t_FontInst)
$2 = 1020
(gdb)
```

Analyzing issue #1888 (CVE-2019-8048)

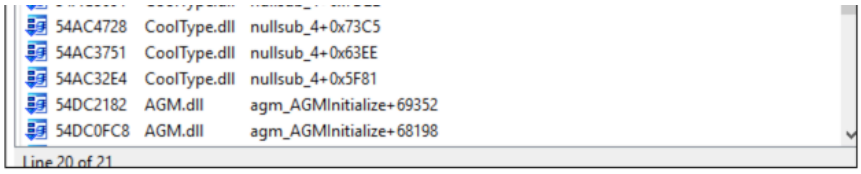
Enumerating the global information may prove very informative and bring answers to a number of questions one could have had while reverse engineering font rasterizers. However, the true power of the symbols lies in the context information available during live debugging. In the previous blog post, we tried to recover the stack trace upon triggering a crash with the PoC for issue #1888. This is how far we got:



- When 'int' is the new 'short' (Jul)
- What is a "good" memory corruption vulnerability? (Jun)
- Analysis and Exploitation of an ESET Vulnerability... (Jun)
- Owning Internet Printing - A Case Study in Modern ... (Jun)
- Dude, where's my heap? (Jun)
- In-Console-Able (May)
- A Tale of Two Exploits (Apr)
- Taming the wild copy: Parallel Thread Corruption (Mar)
- Exploiting the DRAM rowhammer bug to gain kernel p... (Mar)
- Feedback and data-driven updates to Google's discl... (Feb)
- (^Exploiting)\s*(CVE-2015-0318)\s*(in)\s*(Flash$) (Feb)
- A Token's Tale (Feb)
- Exploiting NVMAP to escape the Chrome sandbox - CV... (Jan)
- Finding and exploiting ntpd vulnerabilities (Jan)

2014

- Internet Explorer EPM Sandbox Escape CVE-2014-6350... (Dec)
- pwn4fun Spring 2014 - Safari - Part II (Nov)
- Project Zero Patch Tuesday roundup, November 2014 (Nov)
- Did the "Man With No Name" Feel Insecure? (Oct)
- More Mac OS X and iPhone sandbox escapes and kerne... (Oct)
- Exploiting CVE-2014-0556 in Flash (Sep)
- The poisoned NUL byte, 2014 edition (Aug)
- What does a pointer look like, anyway? (Aug)
- Mac OS X and iPhone sandbox escapes (Jul)
- pwn4fun Spring 2014 - Safari - Part I (Jul)
- Announcing Project Zero (Jul)



When we load the same PDF in Adobe Reader 9.5.5 for macOS, here is what we will see under gdb:

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0xff978034
0x7d87e018 in FixOnePath (maxStem=void, cs=0x7d9dbe98, expansionFactor=3932)
at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/buildch/sources/glbclr.c:787
787
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/buildch/sources/glbclr.c: No such file or
directory.
        in
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/buildch/sources/glbclr.c

(gdb) bt
#0  0x7d87e018 in FixOnePath (maxStem=void, cs=0x7d9dbe98,
expansionFactor=3932) at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/buildch/sources/glbclr.c:787
#1  0x7d87ed67 in GlobalColoring (stems=void, counters=0x0, b3=0xbfffa858,
expansionFactor=3932, numCounters=15, numStems=14, pClientHook=0xbfffa764)
at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/buildch/sources/glbclr.c:608
#2  0x7d85891b in Type1InterpretCharString (inst=0xd6a704,
pProcs=0xbfffa748, pCharDataDesc=0xbfffa910, pCharIO=0xbfffa8e0,
pRunData=0xbfffa728, pPathProcs=0xbfffa544, pT1Args=0xbfffa6c8) at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/buildch/sources/t1interp.c:3874
#3  0x7d83b1d8 in BuildRuns (inst=0xd6a704, pCharDataDesc=0xbfffa910,
pCharIO=0xbfffa8e0, pClipBBox=0x0, pRunData=0xbfffa728,
pCallbackProc=0xbfffa748, pClientHook=0xbfffa764, rFlags=33) at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/buildch/sources/atmbuild.c:384
#4  0x7d83b39c in ATMBuildBitMap (pFontInst=0xd6a704,
pCallbackProc=0x7d9d9e24, pCharDataDesc=0xbfffa910, pCharIO=0xbfffa8e0,
pClipBBox=0x0, pBitMap=0xbfffa994, pClientHook=0xb83ab4) at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/buildch/sources/atmbuild.c:625
#5  0x7d73bc9f in ATMCGetGlyph () at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/truetype/sources/interp.c:2636
#6  0x7d73b517 in Render::RenderToGlyphMap () at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/truetype/sources/interp.c:2636
#7  0x7d739d0c in FontInstanceCache::GetGlyph () at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/truetype/sources/interp.c:2636
#8  0x7d739239 in FontInstanceCache::GetGlyphMaps () at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/truetype/sources/interp.c:2636
#9  0x7d738670 in Text::GetTextGlyphs () at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/truetype/sources/interp.c:2636
#10 0x7d738149 in _CTTextGetTextGlyphs_GetTextGlyphs () at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/truetype/sources/interp.c:2636
[...]
```

Now that is a much more detailed call stack – we can see the names of the ancestors of `ATMBuildBitMap`, and we have access to almost complete type/locals information.

Analyzing issue #1891 (CVE-2019-8042)

As another example, let's take a CoolType crash in TrueType font processing reported in issue #1891. When we open poc.pdf in Reader, we get the following crash:

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x0a68d4cc
0x7d72a310 in fsg_QueryTwilightElement (pPrivateFontSpace=0xa69b008 "",
PrivateSpaceOffsets=0x8caad0) at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/truetype/sources/fsglue.c:915
915
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/truetype/sources/fsglue.c: No such file or
directory.
        in
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/truetype/sources/fsglue.c
```

Once again, we can obtain a very verbose stack trace:

```
(gdb) bt
#0  0x7d72a310 in fsg_QueryTwilightElement (pPrivateFontSpace=0xa69b008 "",
PrivateSpaceOffsets=0x8caad0) at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/truetype/sources/fsglue.c:915
#1  0x7d729c93 in fs__NewTransformation (inputPtr=0xbfffc040,
outputPtr=0xbfffbf70, useHints=1, pFontInst=0x83238d4) at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/truetype/sources/fscaler.c:474
#2  0x7d729b19 in fs_NewTransformation (inputPtr=0xbfffc040,
outputPtr=0xbfffbf70, pFontInst=0x83238d4) at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/truetype/sources/fscaler.c:411
#3  0x7d72997e in SetGlyph (pCharDataDesc=0xbfffc264, inst=0x83238d4,
procs=0x7d9d9e24, mem=0xbfffc1c0, in=0xbfffc040, out=0xbfffbf70) at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/truetype/sources/ttbuild.c:392
#4  0x7d73c331 in TTBuildBitMap (pFontInst=0x83238d4,
pCallbackProc=0x7d9d9e24, pCharDataDesc=0xbfffc264, pCharIO=0xbfffc210,
pClipBBox=0x0, pBitMap=0xbfffc2c4, pClientHook=0xb67fac) at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/truetype/sources/ttbuild.c:1128
#5  0x7d73bc9f in ATMCGetGlyph () at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/truetype/sources/interp.c:2636
#6  0x7d73b517 in Render::RenderToGlyphMap () at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/truetype/sources/interp.c:2636
#7  0x7d739d0c in FontInstanceCache::GetGlyph () at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/truetype/sources/interp.c:2636
#8  0x7d739239 in FontInstanceCache::GetGlyphMaps () at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/truetype/sources/interp.c:2636
#9  0x7d738670 in Text::GetTextGlyphs () at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/truetype/sources/interp.c:2636
#10 0x7d738149 in _CTTextGetTextGlyphs_GetTextGlyphs () at
/Volumes/BuildDisk_mbs16/Acro_root_nsp/bravo/build/cooltype/xcode2/../../../
source/cooltype/source/atm/base/truetype/sources/interp.c:2636
[...]
```

If we wish to dig deeper and analyze the root cause of the problem, we may start with establishing the origin of the unmapped 0x0a68d4cc address. The arguments of the top-level `fsg_QueryTwilightElement` function are:

```
pPrivateFontSpace=0xa69b008 "", PrivateSpaceOffsets=0x8caad0
```

The 0x0a68d4cc and 0xa69b008 addresses are close to each other, with a difference of 56124 bytes between them, so we might expect that the former is calculated based on the latter. Let's look into the second parameter to see if we find any clues that would align with the existing data:

```
(gdb) ptype PrivateSpaceOffsets
type = struct fsg_PrivateSpaceOffsets {
    uint32 offset_storage;
    uint32 offset_functions;
    uint32 offset_instrDefs;
    uint32 offset_controlValues;
    uint32 offset_globalGS;
    uint32 offset_FontProgram;
```

```
    uint32 offset_FontProgram;
    uint32 offset_PreProgram;
    uint32 offset_TwilightZone;
    uint32 offset_TwilightOutline;
    fsg_OutlineFieldInfo TwilightOutlineFieldOffsets;
} *
(gdb) print *(fsg_PrivateSpaceOffsets *)0x8caad0
$2 = {
  offset_storage = 0,
  offset_functions = 192,
  offset_instrDefs = 1096,
  offset_controlValues = 1096,
  offset_globalGS = 3764,
  offset_FontProgram = 4124,
  offset_PreProgram = 7149,
  offset_TwilightZone = 4294911172,
  offset_TwilightOutline = 4294911220,
  TwilightOutlineFieldOffsets = {
    x = 65304,
    y = 326488,
    ox = 587672,
    oy = 848856,
    oox = 1110040,
    ooy = 1371224,
    onCurve = 0,
    sp = 65296,
    ep = 65298,
    f = 1632408,
    fc = 65300,
    maxPointIndex = 65296
  }
}
(gdb)
```

Two of the structure fields seem unusually large, what do they look like as signed numbers?

```
(gdb) print (int)((fsg_PrivateSpaceOffsets *)0x8caad0)->offset_TwilightZone
$3 = -56124
(gdb) print (int)((fsg_PrivateSpaceOffsets *)0x8caad0)-
>offset_TwilightOutline
$4 = -56076
(gdb)
```

We found the culprit! If we open the `fsg_QueryTwilightElement` function in a disassembler, we can confirm that this is indeed how the invalid pointer is constructed prior to being dereferenced. We could now decide to continue the analysis to learn why the two fields are assigned out-of-bounds values and what the root cause of the problem is, or leave it at that and provide the extra bit of information to the vendor while reporting the bug.

Conclusion

As we can see, working with debug metadata can make a world of difference during the security research of a closed-source application. I am hoping that these newly uncovered Reader symbols will make the life of some security engineers easier in the future. For me, it is yet another piece of evidence that reconnaissance is an essential part of vulnerability research and may yield results far exceeding expectations.

On the other hand, the recon step wouldn't be necessary had Adobe made the symbols available to everyone upfront. The fact that the debug data on macOS hasn't been previously discussed means that only the most determined offensive security researchers might have known about it in the past. We would therefore encourage closed-source companies to improve the inspectability of their software by proactively sharing symbols, similarly to what Microsoft does with their public symbol server. Such an approach would level the playing field and base the security of client applications on openness and not obscurity, which we believe would consequently benefit the users in the long term.

https://googleprojectzero.blogspot.com/	January 30, 2020 at 2:49:25 PM
Unknown Version	10.15.2