Erledigt "Du hast ja Alles" - hmmm vielleicht, wenn ich einen Laptop habe.

Beitrag von „Brumbaer“ vom 1. Februar 2018, 14:21

Aller Anfang ist schwer
Der heutige Exkurs führt uns in die unheimlichen Gefilde der Maschinensprache.
Was hier beschrieben wird ist es eher komplex und bedarf eines soliden Grundwissens. Ich weiß nicht so recht ob ich Evas Begleiter rufen und bei ihnen beginnen soll, oder nur das Ergebnis präsentieren, weil es eh keinen interessiert und zu wenig Infos ein Nachverfolgen des Vorgehens sowieso unmöglich macht.
Ich lasse Adam in Ruhe und gehe einen Mittelweg. Wem's zu viel wird einfach aufhören, nächstes Mal wird es wieder weniger "speziell". Aber vielleicht regt es den einen oder anderen an mal etwas tiefer in die Materie ein zu steigen.

Nach dieser obskuren Einleitung sei gesagt, dass ich nicht weiß ob das Problem mit einem der tausend gebetsmühlenartig angewendeten Standard Rehab DSDT Patches nicht aufgetreten wäre. Das spielt auch keine Rolle, da ich sie bewußt nicht angewandt habe, konnten sie auch nichts verhindern.

Panic auf der Titanic
Dank AptioMemoryFix, funktioniert das NVRam und macos liefert den, in meiner letzten Post gezeigten, Fehlerbericht.

Kerneltrap sagt einem, wo das Problem aufgetreten ist. Das ist oft nicht sehr aussagekräftig. Und ohne Anhaltspunkte sagt einem der Wert auch nicht viel.
Interessanter ist da der Backtrace.

Aus Computersicht werden immer nur Unterprogramme ausgeführt. Ob App, Programm, Funktion, Methode, Subroutine oder was auch immer für die CPU ist es ein Unterprogramm, das ausgeführt wird und dann dorthin zurückkehrt von wo es aufgerufen wurde.

Babuschka
Jedes Unterprogramm hat eine Aufgabe zu erfüllen. Ist die Aufgabe komplex oder hat man andere Unterprogramme, die Teile der Aufgabe lösen können, so bekommt ein solches eine

Teilaufgabe zugewiesen und wird aufgerufen. Und es selbst ruft vermutlich wieder Unterprogramme auf. Unterprogramme, können sich sogar selbst aufrufen. Wie in einer virtuellen Babuschka hat man Unterprogramme in Unterprogrammen in Unterprogrammen. Dabei entsteht eine Hierarchie. Ein aufgerufenes Unterprogramm ist tiefer als das aufrufende Unterprogramm. Grob kann man sagen je tiefer, desto Detailarbeit. Es gilt zu beachten, dass ein Unterprogramm im Sinne von Quellcode an sich keine Position in der Hierarchie hat, sondern erst der Aufruf die Position festlegt. Unterprogramme können so je nach Programm weiter oben oder weiter unten oder sogar mehrmals in der Hierarchie vorkommen.

Errötend folgt er ihren Spuren

Jede Zeile im Backtrace zeigt uns in der ersten Spalte, die Adresse des Stackframes und in der zweiten Spalte die Rücksprungadresse eines Unterprogramms.
Die Rücksprungadresse in der ersten Zeile zeigt wohin das gerade ausgeführte Unterprogramm zurückspringen wird, wenn es fertig ist. Die nächste Zeile zeigt uns wohin das Unterprogramm, das dieses Unterprogramm aufgerufen hat, springen wird, wenn es fertig ist usw.. Je tiefer ein Unterprogramm in der Hierarchie steht, desto weiter vorne steht es im Backtrace.
Wir können also sehen, dass der Computer zum Zeitpunkt des Absturzes ein Unterprogramm ausgeführt hat, das danach zur Adressen 0xffffff8002e505f6 zurückkehren würde.
Boah, Boah, Boa Constrictor, 0xffffff8002e505f6 meine Güte, da bin ich aber ... sprachlos.

Das beste kommt zum Schluß

Zugegebenermaßen klingt 0xffffff8002e505f6 nicht hilfreich.
Aber am Ende des Tunnels ist Licht - keine Angst, wir gehen nicht drauf zu, wir schauen es uns nur an.
"Kernel Extensions in backtrace:" sagt uns in welchen Kexten die Rücksprungadressen liegen und somit welche Kexte die Unterprogramme aufgerufen haben, die am Ende zum Crash führten.
Die dependency Zeilen sagen uns in welchen Kexten das Kext, das in deren Abhängigkeit steht, Unterprogramme aufruft, aber nicht in welchen Kexten, die Kexte wieder Unterprogramme aufrufen. Aber es ist an dieser Stelle irrelevant.
Wenn wir die dependency Zeilen streichen, bleiben 3 Zeilen übrig. Am Anfang steht die Kennung des Kextes und am Ende welchen Speicherbereich das Kext belegt. Rücksprungadressen, die in diesem Speicherbereich liegen, zeigen also auf die Stelle an der das Kext ein Unterprogramm aufgerufen hat.
Hier habe ich die Rücksprungadressen, die sich einem der Kexte zuordnen lassen farblich markiert.

Backtrace (CPU 2), Frame : Return Address

0xffffff91201531a0 : 0xffffff8002e505f6
0xffffff91201531f0 : 0xffffff8002f7d604
0xffffff9120153230 : 0xffffff8002f6f0f9
0xffffff91201532b0 : 0xffffff8002e02120
0xffffff91201532d0 : 0xffffff8002e5002c
0xffffff9120153400 : 0xffffff8002e4fdac
0xffffff9120153460 : 0xffffff8002f6f2e9
0xffffff91201535e0 : 0xffffff8002e02120
0xffffff9120153600 : 0xffffff7f85621d08
0xffffff9120153720 : 0xffffff7f85639388
0xffffff9120153760 : 0xffffff7f856390f0
0xffffff91201537b0 : 0xffffff7f85639037
0xffffff9120153810 : 0xffffff7f856367e5
0xffffff9120153850 : 0xffffff7f8563ba0c
0xffffff9120153880 : 0xffffff7f85640088
0xffffff91201538d0 : 0xffffff7f85661134
0xffffff9120153950 : 0xffffff7f856624d9
0xffffff9120153990 : 0xffffff7f856631c2
0xffffff91201539e0 : 0xffffff7f8565a68c
0xffffff9120153a20 : 0xffffff7f8565eded
0xffffff9120153a80 : 0xffffff7f85617283
0xffffff9120153b10 : 0xffffff7f83b97e99
0xffffff9120153b60 : 0xffffff7f83b9734c
0xffffff9120153b80 : 0xffffff7f83b974ca
0xffffff9120153bb0 : 0xffffff7f84516c80
0xffffff9120153bf0 : 0xffffff7f84517352
0xffffff9120153c70 : 0xffffff7f8450f158
0xffffff9120153cc0 : 0xffffff8003464cff
0xffffff9120153d10 : 0xffffff80034c0ee7
0xffffff9120153d70 : 0xffffff8002f292f2
0xffffff9120153dc0 : 0xffffff8002e55c30
0xffffff9120153e10 : 0xffffff8002e32cbd
0xffffff9120153e60 : 0xffffff8002e45b7b
0xffffff9120153ef0 : 0xffffff8002f5952d
0xffffff9120153fa0 : 0xffffff8002e02926
Kernel Extensions in backtrace:
com.apple.iokit.IOACPIFamily(1.4)[8794C760-FDD9-3664-ADED-4A9BBEC6E517]@0xffffff7f83b96000->0xffffff7f83b9efff
com.apple.driver.AppleACPIPlatform(6.1)[1804645B-B360-305E-B1BE-916F5E3E1CC4]@0xffffff7f85611000->0xffffff7f856acfff
dependency: com.apple.iokit.IOACPIFamily(1.4)[8794C760-FDD9-3664-ADED-4A9BBEC6E517]@0xffffff7f83b96000
dependency: com.apple.driver.AppleSMCRTC(1.0)[3F01C7A4-754F-39DD-A872-

B4FAF0442276]@0xffffff7f84be4000
dependency: com.apple.iokit.IOPCIFamily(2.9)[C08F7FC1-78A4-3A1B-BFE2-C07080CF2048]@0xffffff7f83734000
dependency: com.apple.driver.AppleSMC(3.1.9)[259F0A4B-0AAB-30F3-8896-629A102CBD70]@0xffffff7f83b9f000
com.apple.iokit.IOGraphicsFamily(517.22)[2AEA02BF-2A38-3674-A187-E5F610FD65B7]@0xffffff7f84500000->0xffffff7f84546fff
dependency: com.apple.iokit.IOPCIFamily(2.9)[C08F7FC1-78A4-3A1B-BFE2-C07080CF2048]@0xffffff7f83734000

Es ist alles relativ

Wie bekommen wir nun raus, was da passiert ? Der Programmcode eines Kextes befindet sich für gewöhnlich in der Datei im MacOS Ordner des Kextes. Der Name der Datei steht in der Info.plist des Kextes und stimmt meist mit dem Namen des Kextes überein.
Wie kommt man nun von einer Rücksprungadresse zu einer Stelle im Programm bzw. der Datei ?
Man berechnet den Abstand zwischen der Rücksprungadresse und dem Anfang des Kextes. Dazu zieht man einfach die Startadresse des Kextes von der Rücksprungadresse ab.

Das sieht dann so aus

Backtrace (CPU 2), Frame : Return Address
0xffffff91201531a0 : 0xffffff8002e505f6
0xffffff91201531f0 : 0xffffff8002f7d604
0xffffff9120153230 : 0xffffff8002f6f0f9
0xffffff91201532b0 : 0xffffff8002e02120
0xffffff91201532d0 : 0xffffff8002e5002c
0xffffff9120153400 : 0xffffff8002e4fdac
0xffffff9120153460 : 0xffffff8002f6f2e9
0xffffff91201535e0 : 0xffffff8002e02120

0xffffff9120153600 : 0x10d08
0xffffff9120153720 : 0x28388
0xffffff9120153760 : 0x280f0
0xffffff91201537b0 : 0x28037
0xffffff9120153810 : 0x257e5
0xffffff9120153850 : 0x2Aa0c
0xffffff9120153880 : 0x2F088
0xffffff91201538d0 : 0x50134
0xffffff9120153950 : 0x514d9
0xffffff9120153990 : 0x521c2
0xffffff91201539e0 : 0x4968c
0xffffff9120153a20 : 0x4dded
0xffffff9120153a80 : 0x062830
0xffffff9120153b10 : 0x1e99
0xffffff9120153b60 : 0x134c
0xffffff9120153b80 : 0x14ca
0xffffff9120153bb0 : 0x16c80
0xffffff9120153bf0 : 0x17352
0xffffff9120153c70 : 0x0f158
0xffffff9120153cc0 : 0xffffff8003464cff
0xffffff9120153d10 : 0xffffff80034c0ee7
0xffffff9120153d70 : 0xffffff8002f292f2
0xffffff9120153dc0 : 0xffffff8002e55c30
0xffffff9120153e10 : 0xffffff8002e32cbd
0xffffff9120153e60 : 0xffffff8002e45b7b
0xffffff9120153ef0 : 0xffffff8002f5952d
0xffffff9120153fa0 : 0xffffff8002e02926
Kernel Extensions in backtrace:
com.apple.iokit.IOACPIFamily(1.4)[8794C760-FDD9-3664-ADED-4A9BBEC6E517]@0xffffff7f83b96000->0xffffff7f83b9efff
com.apple.driver.AppleACPIPlatform(6.1)[1804645B-B360-305E-B1BE-916F5E3E1CC4]@0xffffff7f85611000->0xffffff7f856acfff
dependency: com.apple.iokit.IOACPIFamily(1.4)[8794C760-FDD9-3664-ADED-4A9BBEC6E517]@0xffffff7f83b96000
dependency: com.apple.driver.AppleSMCRTC(1.0)[3F01C7A4-754F-39DD-A872-B4FAF0442276]@0xffffff7f84be4000
dependency: com.apple.iokit.IOPCIFamily(2.9)[C08F7FC1-78A4-3A1B-BFE2-C07080CF2048]@0xffffff7f83734000
dependency: com.apple.driver.AppleSMC(3.1.9)[259F0A4B-0AAB-30F3-8896-629A102CBD70]@0xffffff7f83b9f000
com.apple.iokit.IOGraphicsFamily(517.22)[2AEA02BF-2A38-3674-A187-E5F610FD65B7]@0xffffff7f84500000->0xffffff7f84546fff
dependency: com.apple.iokit.IOPCIFamily(2.9)[C08F7FC1-78A4-3A1B-BFE2-

C07080CF2048]@0xffffff7f83734000

Lesbar ist anders

In der Datei stehen einfach nur ein Haufen Bytes. Hin und wieder lässt sich ein Text erkennen, aber alles in allem kommt einem das spanisch vor, außer man ist Engländer, dann kommt es einem griechisch vor. Weiß jemand wie es einem Spanier oder einem Griechen vorkommt ?

Disassembler machen aus Maschinencode Assemblercode: Der ist besser lesbar, hält aber dem Vergleich mit C oder einer anderen Hochsprache nicht stand.
Ich verwende Hopper um die Datei zu öffnen und zu disassemblieren, aber es geht auch mit Bordmitteln. Damit jeder die hiesigen Schritte nachvollziehen kann verwende ich deshalb den Terminal Befehl otool statt Hopper.

Alles eine Frage der Perspektive

Ok, wo fangen wir an ? Tief in der Hierarchie sind wir dem Fehler näher, aber erkennen möglicherweise vor lauter Bäumen den Wald nicht. Ganz oben sind wir möglicherweise so weit vom Wald weg, dass wir keine Bäume sehen. Also ..... eeene meeene Miste..... oben, warum nicht, das sollte uns zumindest einen groben Hinweise geben worum es geht.
Die "oberste" Rücksprungadresse in einem Kext ist 0x0f158 in IOGraphicsFamily. Das ist ein systemeigenes Kext und somit in /System/Library/Extensions zu finden. Mit einem Rechtsclick auf das Kext, kann man sich den Paketinhalt anzeigen lassen. Und wie immer wenn man etwas zeigen will, verwendet dieses Kext nicht die "normale" Kext-Struktur, sondern schmeißt alle Dateien zusammen. Der Finder zeigt uns zwei ausführbare Dateien iogdiagnose und IOGraphicsFamily. Wir könnten nun in der Info.plist nachschauen, welche Datei, den Programmcode für das Kext enthält, aber ich wage es zu behaupten, dass es die Datei mit dem Namen des Kextes ist.

Zerleger

"otool, otool bitte, auf die Bühne". Wir öffnen das Terminal. Und schreiben

Code

1. otool -tV

Nach dem V folgt ein Leerschritt. -tV bedeutet Text-Segment anzeigen und mit Symbolen disassemblieren. Das Text Segment beinhaltet, anders als er der Name vermuten lässt,

Programmcode.

Nun ziehen wir die IOGraphicsFamily Datei aus dem Finder auf das Terminal Fenster, das spart es und den Pfad zu tippen.

Code

```
1. otool -tV /System/Library/Extensions/IOGraphicsFamily.kext/IOGraphicsFamily
```

Dahinter kommt noch ein Leerschritt und die Angabe der Datei in die wir den Assemblercode geschrieben haben wollen. Da wir hier ein Unix Derivat haben kommt ein > vor den Dateinamen.

Code

```
1. otool -tV /System/Library/Extensions/IOGraphicsFamily.kext/IOGraphicsFamily >~/IOGF.txt
```

Drücken wir nun Eingabe haben wir das Assemblerlisting in der Datei IOGF.txt im Benutzerverzeichnis. Die Tilde ~bedeutet Benutzerverzeichnis.

Zuviel ist zuviel

Öffnen wir die Datei werden wir von der schieren Anzahl von Zeilen erschlagen. Aber Gott sei Dank, müssen wir uns nicht alle Zeilen anschauen, sondern nur die in der näheren Umgebung unserer Rücksprungadressen. 0x0f158 ist die Rücksprungadresse die uns Interessiert also suchen wir nach 000f158 im Text. das 0x lassen wir weg, weil das Format der Adressen in der Datei kein 0x enthält und die Nullen fügen wir hinzu um die Suche einzugrenzen.

Eine Code Zeile beginnt mit der Adresse, gefolgt vom Befehl und dessen Parametern und gegebenenfalls einem Kommentar.

Kalt

Code

```
1. Schnipp
2.
3.
```

4. 000000000000f147 callq __ZN13IOFramebuffer18setPlatformConsoleEP8PE_Videojy ## IOFramebuffer::setPlatformConsole(PE_Video*, unsigned int, unsigned long long)
5. 000000000000f14c movq (%r14), %rax
6. 000000000000f14f movq %r14, %rdi
7. 000000000000f152 callq *0x978(%rax)
8. ----------
9. 000000000000f158 movl %eax, %ebx
10. ----------
11. 000000000000f15a testl %ebx, %ebx
12. 000000000000f15c jne 0xf077
13. 000000000000f162 movq %r12, %rdi
14. 000000000000f165 callq __ZN23IOFramebufferUserClient8withTaskEP4task ## IOFramebufferUserClient::withTask(task*)
15. 000000000000f16a jmp 0xf182
16.
17.
18. Schnipp

Alles anzeigen

Rücksprungadressen zeigen hinter den Aufruf, denn dort soll das Programm ja weiter machen, wenn es aus dem Unterprogramm zurückkommt. Aufrufe von Unterprogrammen heißen call irgendwas.
Wir scrollen nun solange nach oben, bis wir einen Namen sehen, wo sonst Adressen stehen. Das ist der Name des Unterprogrammes dessen Code wir gerade sehen entdecken.

Code

```
1. Schnipp
2.
3.
4. 000000000000eeb0 popq %rbp
5. 000000000000eeb1 retq
6. __ZN13IOFramebuffer13newUserClientEP4taskPvjPP12IOUserClient:
7. 000000000000eeb2 pushq %rbp
8. 000000000000eeb3 movq %rsp, %rbp
9. 000000000000eeb6 pushq %r15
10.
11.
12. Schnipp
```

Alles anzeigen

__ZN13IOFramebuffer13newUserClientEP4taskPvjPP12IOUserClient ist C++ Compiler Gibberish. Es geht los mit der Klasse IOFramebuffer, dann kommt der Unterprogrammname, newUserClient und dann der Typ des Parameters, IOUserClient.
Wir wissen nun der Aufruf erfolgt sobald ein Framebuffer einen neuen UserClient anlegt. Die restlichen Texte zwischen diesem Namen und dem Aufruf sind genauso wenig hilfreich. Probieren wir es mit der nächst tieferen Rücksprungadresse.

Warm
Das ist die 0x17352. Suchen nach 00017352 und nach oben scrollen zum Unterprogrammnamen zeigt uns.

Code

```
1. Schnipp
2.
3.
4. __ZN13IOFramebuffer4openEv:
5. 0000000000017198 pushq %rbp
6. 0000000000017199 movq %rsp, %rbp
7. 000000000001719c pushq %r15
8. 000000000001719e pushq %r14
9. 00000000000171a0 pushq %r13
10. 00000000000171a2 pushq %r12
11. 00000000000171a4 pushq %rbx
```

12. 00000000000171a5 subq $0x48, %rsp
13.
14.
15. Schnipp
16.
17.
18. 0000000000017302 cmpq $0x0, 0x1e066(%rip)
19. 000000000001730a jne 0x17348
20. 000000000001730c movq 0xfdc5(%rip), %rax
21. 0000000000017313 movq (%rax), %rbx
22. 0000000000017316 leaq 0xcc56(%rip), %rdi ## literal pool for: "AppleClamshellState"
23. 000000000001731d xorl %esi, %esi
24. 000000000001731f callq 0x17324
25. 0000000000017324 leaq __ZN13IOFramebuffer16clamshellHandlerEPvS0_P9IOServiceP10IONotifier(%rip), %rdx ## IOFramebuffer::clamshellHandler(void*, void*, IOService*, IONotifier*)
26. 000000000001732b xorl %ecx, %ecx
27. 000000000001732d xorl %r8d, %r8d
28. 0000000000017330 movl $0x2710, %r9d
29. 0000000000017336 movq %rbx, %rdi
30. 0000000000017339 movq %rax, %rsi
31. 000000000001733c callq 0x17341
32. 0000000000017341 movq %rax, __ZL20gIOFBClamshellNotify(%rip) ## gIOFBClamshellNotify
33. 0000000000017348 movl $0xc, %edi
34. 000000000001734d callq __ZN13IOFramebuffer18readClamshellStateEy ## IOFramebuffer::readClamshellState(unsigned long long)
35. ----------
36. 0000000000017352 cmpl $-0x1, 0x1d78b(%rip)
37. ----------
38. 0000000000017359 jne 0x174a9
39. 000000000001735f leaq 0xb05f(%rip), %rdi ## literal pool for: "backlight"
40. 0000000000017366 xorl %esi, %esi
41.
42.
43. Schnipp

Alles anzeigen

Höre ich da ein Aaah ? Wir sind in der Open Methode eines FrameBuffers. Aber wichtiger ist der Aufruf der zu unserer Rücksprungadresse gehört heißt readClamshellState. Entweder erkundigt sich macos nach der Qualität der Muscheln oder es hat etwas mit dem Zuklappen des Laptops zu tun.

Also auf zum nächsten Rücksprung, 0x16c80.

Heiß

Code

1. Schnipp
2.
3.
4. __ZN13IOFramebuffer18readClamshellStateEy:
5. 0000000000016bc6 pushq %rbp
6. 0000000000016bc7 movq %rsp, %rbp
7. 0000000000016bca pushq %r15
8. 0000000000016bcc pushq %r14
9. 0000000000016bce pushq %r13
10. 0000000000016bd0 pushq %r12
11. 0000000000016bd2 pushq %rbx
12. 0000000000016bd3 pushq %rax
13. 0000000000016bd4 movq %rdi, %r14
14. 0000000000016bd7 movl __ZL23gIOFBLastClamshellState(%rip), %r15d ## gIOFBLastClamshellState
15. 0000000000016bde movq __ZZN13IOFramebuffer18readClamshellStateEyE9lidDevice(%rip), %rdi ## IOFramebuffer::readClamshellState(unsigned long long)::lidDevice
16. 0000000000016be5 testq %rdi, %rdi
17. 0000000000016be8 jne 0x16c64
18. 0000000000016bea leaq 0xdacf(%rip), %rdi ## literal pool for: "PNP0C0D"
19. 0000000000016bf1 xorl %esi, %esi
20. 0000000000016bf3 callq 0x16bf8 laut Hopper IOService::nameMatching(char const*, OSDictionary*)
21. 0000000000016bf8 movq %rax, %rbx
22. 0000000000016bfb movq %rbx, %rdi
23. 0000000000016bfe callq 0x16c03 laut Hopper IOService::getMatchingServices(OSDictionary*)
24. 0000000000016c03 movq %rax, %r13
25. 0000000000016c06 testq %rbx, %rbx
26. 0000000000016c09 je 0x16c14
27. 0000000000016c0b movq (%rbx), %rax

28. 0000000000016c0e movq %rbx, %rdi
29. 0000000000016c11 callq *0x28(%rax)
30. 0000000000016c14 testq %r13, %r13
31. 0000000000016c17 je 0x16c58
32. 0000000000016c19 movq (%r13), %rax
33. 0000000000016c1d movq %r13, %rdi
34. 0000000000016c20 callq *0x128(%rax)
35. 0000000000016c26 movq %rax, %r12
36. 0000000000016c29 leaq 0xda98(%rip), %rsi ## literal pool for: "IOACPIPlatformDevice"
37. 0000000000016c30 movq %r12, %rdi
38. 0000000000016c33 callq 0x16c38 laut Hopper OSMetaClassBase::metaCast(char const*) const
39. 0000000000016c38 testq %rax, %rax
40. 0000000000016c3b je 0x16c4e
41. 0000000000016c3d movq %r12, __ZZN13IOFramebuffer18readClamshellStateEyE9lidDevice(%rip) ## IOFramebuffer::readClamshellState(unsigned long long)::lidDevice
42. 0000000000016c44 movq (%r12), %rax
43. 0000000000016c48 movq %r12, %rdi
44. 0000000000016c4b callq *0x20(%rax)
45. 0000000000016c4e movq (%r13), %rax
46. 0000000000016c52 movq %r13, %rdi
47. 0000000000016c55 callq *0x28(%rax)
48. 0000000000016c58 movq __ZZN13IOFramebuffer18readClamshellStateEyE9lidDevice(%rip), %rdi ## IOFramebuffer::readClamshellState(unsigned long long)::lidDevice
49. 0000000000016c5f testq %rdi, %rdi
50. 0000000000016c62 je 0x16c93
51. 0000000000016c64 movq (%rdi), %rax
52. 0000000000016c67 leaq 0xda6f(%rip), %rsi ## literal pool for: "_LID"
53. 0000000000016c6e leaq -0x2c(%rbp), %rdx
54. 0000000000016c72 xorl %ecx, %ecx
55. 0000000000016c74 xorl %r8d, %r8d
56. 0000000000016c77 xorl %r9d, %r9d
57. 0000000000016c7a callq *0x8d0(%rax)
58. ----------
59. 0000000000016c80 testl %eax, %eax
60. ----------
61. 0000000000016c82 jne 0x16c93
62. 0000000000016c84 xorl %eax, %eax
63. 0000000000016c86 cmpl $0x0, -0x2c(%rbp)
64. 0000000000016c8a sete %al

65.
66.
67. Schnipp

Alles anzeigen

Den Unterprogrammnamen brauchen wir nicht suchen, denn den kennen wir schon vom Aufruf, readClamshellState. Das sagt schon alles. Die Frage ist wie das Unterprogramm das macht. Wir versuchen nicht den Code zu analysieren, wir halten uns erst mal nur an die Texte.
Wie jeder DSDT Junky weiß ist "PNP0C0D" die Kennung für ein Lid Device. Und wer es nicht weiß kann es googeln. Lid ist ein Deckel, eine Klappe, aber wir waren ja schon soweit, dass es was mit der Klappe zu tun hat. Wir können mit dem IORegistryEditor oder in der DSDT nachschauen, ob unser Rechner so ein klappriges Gerät hat. Er hat.
Wir gehen also davon aus, dass das Programm nach dem Gerät mit dieser Kennung sucht - warum sonst würde die Kennung hier erwähnt.
Nun stellt sich die Frage ob es das richtige Gerät ist, "IOACPIPlatformDevice" lässt vermuten, dass hier nachgeschaut wird ob es sich bei dem Gerät um ein IOACPIPlatformDevice handelt.
Das sind ziemlich viele Vermutungen, aber sie lassen sich mit Hopper erhärten. Hopper liefert mehr Informationen und zeigt, dass mit "PNP0C0D" IOService::nameMatching(char const*, OSDictionary*) und mit dessen Ergebnis IOService::getMatchingServices(OSDictionary*) aufgerufen wird. Das ist die Art, wie man nach einem Service(Gerät) mit dieser Kennung gesucht.
Hopper bestätigt auch, dass "IOACPIPlatformDevice" für einen DynamicCast und somit für eine Überprüfung des Gerätetyps verwendet wird.
Der nächste Text ist "_LID". Unterstrich und 3 Buchstaben schreit nach einem DSDT Namen. Also die DSDT geöffnet und danach gesucht und siehe da es gibt eine Methode namens _LID in dem Gerät LID0, das vom Typ PNP0C0D ist. Es scheint nicht zu gewagt zu sein, zu behaupten, dass hier die Methode _LID ausgeführt wird. Leider gibt auch Hopper an der Stelle keinen Namen aus. Das ist aber kein Problem, da die nächste Rücksprungadresse im Backtrace innerhalb des aufgerufenen Unterprogrammes liegt. Die nächste Rücksprungadresse ist 0x14ca in IOACPIFamily.
Wenn man nun AppleACPIPlatform mit otool disassembliert, wird man feststellen, dass die Rücksprungadresse in evaluateInteger liegt. Es wird also _LID ausgeführt und das Ergebnis als Integer interpretiert.

Land in Sicht

An der Stelle können wir sagen, dass der Crash dann erfolgt, wenn die _LID Methode des Gerätes vom Typ "PNP0C0D" aufgerufen wird. Das ist wie Amerika entdecken. Die Frage ist gehen wir an Land und erforschen den neuen Kontinent ? Wasser und Vorräte müssen wir nicht ergänzen, Anzeichen von Skorbut gibt es auch keine, wir müssen also nicht, wir können Amerika auch Columbus überlassen.

Einfach nur die Klappe halten

Da wir wissen wo es kracht, können wir überlegen wie wir es verhindern. Wenn das Gerät keine _LID Methode hat, kann es mit der _LID Methode keine Probleme geben. Oder man lässt ihn erst gar kein passendes Gerät finden.

Ich entscheide mich für die Methode mit der Methode.

Damit _LID nicht aufgerufen werden kann, wird die Methode einfach umbenannt. Mit einem passenden ACPI Patch macht das Clover sehr schön. Wir wollen _LID durch etwas anderes, sagen wir, BLID ersetzen. Der zu findende Wert ist 5F4C4944. Den ersetzen wir mit 424C4944.

Der Rechner wird jetzt nicht mehr auf ein Schließen der Klappe reagieren, aber hoffentlich nicht mehr crashen.

Und immer wieder geht die Sonne auf

Also config.plist angepasst. Neustart und - alles ist gut und Neustart und immer noch gut.

Der Fehler tritt nicht mehr auf, aber wir haben ihn nicht behoben, wir haben nur das Symptom bekämpft. Genau genommen haben wir nicht mal den Fehler gefunden, denn der tritt in einem der tieferen Unterprogramme auf, aber ein bootfähiges ist mir im Moment erstmal genug.

Hexenwerk

Der letzte Schritt, das Feststellen was passiert, scheint wie Hexenwerk und vielleicht etwas willkürlich. Das ist es aber nicht.

Wenn man sich aus dem Assemblercode einen Pseudocode (etwas das aussieht wie Quellcode, aber keiner ist) bastelt bekommt man so etwas wie

Code

```
1. Unterprogramm __ZN13IOFramebuffer18readClamshellStateEy
2.
3.
4. ZuSuchen = IOService::nameMatching ("PNP0C0D")
5. ListeDerGefundenGeräte = IOService::getMatchingServices(ZuSuchen)
6.
7.
8. ErsterEintragInDerListe bestimmen
9.
10.
11. Wenn es keinen gibt hau ab.
```

```
12.
13.
14. ACPIPlatformDevice = OSDynamicCast (IOACPIPlatformDevice, ErsterEintragInDerListe)
15.
16.
17. Ist der Null dann hau ab
18.
19.
20. ACPIPlatformDevice.evaluateInteger("_LID");
```

Alles anzeigen

Wenn man Erfahrung mit IOKit hat, erkennt man die Vorgehensweise. Das ist Standard, wie man halt ein Gerät eines Gerätetypes, der in der DSDT vorkommt, findet, hat also nicht mit Gerate oder Hellsehen zu tun.
Raten oder "Sich-Denken" muss man nur, wen einem das Wissen fehlt, führt aber zum selben Ziel - zumindest, wenn man richtig rät.

Auf Regen folgt der Sonnenschein
Nächstes Mal wird's wieder einfacher und alltagstauglicher.