The Mystery of Metador | Unpicking Mafalda's Anti-Analysis Techniques

sentinelone.com/labs/the-mystery-of-metador-unpicking-mafaldas-anti-analysis-techniques/

Aleksandar Milenkoski

Overview

At the inaugural LabsCon, we unveiled Metador, a previously unreported threat actor that targets telecommunications, internet service providers, and universities in the Middle East and Africa. We observed Metador using two versions of a feature-rich backdoor, dubbed 'Mafalda', one of which features anti-analysis techniques to make analysis challenging.

In this article, we provide a deep dive into the anti-analysis techniques that Mafalda implements. This article complements our previous report on Metador and offers a deeper understanding of how Mafalda tries to hinder analysis and make detection and attribution more challenging for analysts.

The implementation of Mafalda suggests that the malware is maintained and developed by a dedicated team. Mafalda includes comprehensive backdoor command documentation with comments for a separate group of operators. In addition, Mafalda implements an execution log that the malware maintains when it runs on an infected system. The log provides detailed information about the execution of the malware on the system and therefore is a rich resource to analysts. Our previous report discusses the functionalities of Mafalda in greater detail.

Throughout our analysis, we retrieved and analyzed two variants of Mafalda, which we refer to as 'Mafalda clear build 144' (compiled with a timestamp of April 2021) and its successor, 'obfuscated Mafalda variant' (compiled with a timestamp of December 2021). The newer, obfuscated Mafalda variant extends the backdoor functionalities that the older variant provides and implements the anti-analysis techniques that we cover in this article.

String Obfuscation

Mafalda uses obfuscated strings for different purposes, for example, to dynamically resolve library function addresses through library and library export names, or to store content in the execution log that Mafalda maintains. Mafalda obfuscates strings by:

- Splitting the strings into multiple portions, with a maximum portion length of 9 characters.

- Encrypting and encoding each string portion. Mafalda encodes a portion of an obfuscated string using the bitmask 0x7F and XOR-encrypts the portion using a portion-specific XOR key of one byte.

Therefore, to restore an obfuscated string into a valid string, Mafalda first decodes and decrypts each of the string's portions, and then concatenates the string portions together.

The figure below depicts a snippet of the function that Mafalda executes to decode and decrypt a portion of an obfuscated string (a2 is a portion of an obfuscated string, v2 is an XOR key).

```
[...]
for ( i = 0i64; i < 9; ++i )
{
  v5 = a2 & 0x7F;
  if ( (a2 & 0x7F) != 0 )
    v5 ^= v2;
  v8[i] = v5;
  a2 >>= 7;
}
[...]
```

Mafalda's function for decoding and decrypting string portions

String Encryption

In addition to the string obfuscation approach, Mafalda works with encrypted versions of strings that may represent an information source to malware analysts. Such strings include segments of the execution log and debugger messages that Mafalda generates.

We noted that Mafalda prints encrypted debugger messages if the name of the computer where it executes is `WIN-K4C3EKBSMMI`, possibly indicating the name of the computer used by the developers.

```
0:000> g
DbG: ?vhho0G~IcUdkQXD$M-}C44!sE${%k(HwvLR8+!HRwvLXi>41m~T$oB2;Sn{,16:9_e        woLY#gxd7&Jz?
ModLoad: 00007ff8`d1990000 00007ff8`d19a1000   C:\Windows\System32\kernel.appcore.dll
DbG: ?<U_aS]RZ4Rvlh$B,^"FM~\w{'
6%]sd?1994768?↑uM&J:1H(wcHt'kw:wS6GT?0DbG: ?~!KP<Dq OZ"^h?E004197EB89EEAA0B097C9F4DA2F9A9EDbG: ?<U_aS]RZ4Rvlh$B,^"FM~\w{'
+msi~7FLb/?Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0; KTXN)
ModLoad: 00007ff8`c1ba0000 00007ff8`c2081000   C:\Windows\SYSTEM32\wininet.DLL
ModLoad: 00007ff8`c6a30000 00007ff8`c6cd8000   C:\Windows\SYSTEM32\iertutil.dll
[...]
ModLoad: 00007ff8`c66a0000 00007ff8`c6877000   C:\Windows\SYSTEM32\urlmon.dll
DbG: [-] ?Fx;aBco}G3v,ZA6H|BH? Error: 12029 DbG: ?Oz$yJtD?D↑e?N?DbG: [-] ?Fx;aBco}G3v,ZA6H|BH? Error: 12029 DbG: ?Oz$yJtD?D↑e?N?
```

Encrypted debugger messages

In contrast to the Mafalda clear build 144, the obfuscated Mafalda variant writes encrypted strings to its execution log. Given that this log provides extensive information about the operation of the malware, encrypting the execution log serves to hinder analysis.

```
0:008> db 00000261`ad72fbb0 L0x300
00000261`ad72fbb0  01 00 76 00 68 00 68 00-6f 00 30 00 47 00 7e 00  ..v.h.h.o.0.G.~.
00000261`ad72fbc0  49 00 63 00 55 00 64 00-6b 00 51 00 58 00 44 00  I.c.U.d.k.Q.X.D.
00000261`ad72fbd0  24 00 4d 00 2d 00 7d 00-43 00 34 00 34 00 21 00  $.M.-.}.C.4.4.!.
00000261`ad72fbe0  73 00 45 00 24 00 7b 00-25 00 6b 00 28 00 48 00  s.E.$.{.%.k.(.H.
00000261`ad72fbf0  77 00 76 00 4c 00 52 00-38 00 2b 00 21 00 48 00  w.v.L.R.8.+.!.H.
00000261`ad72fc00  52 00 77 00 76 00 4c 00-58 00 69 00 3e 00 34 00  R.w.v.L.X.i.>.4.
00000261`ad72fc10  31 00 6d 00 7e 00 54 00-24 00 6f 00 42 00 32 00  1.m.~.T.$.o.B.2.
00000261`ad72fc20  3b 00 53 00 6e 00 7b 00-2c 00 31 00 36 00 3a 00  ;.S.n.{.,.1.6.:.
```

```
[-] VirtualFree 1 failed Error: 487 Attempt to access invalid address.
[-] VirtualFree 2 failed Error: 87 The parameter is incorrect.
Obfuscated thread creation disabled (todo!)
added vectored exception handler
Obfuscated thread creation disabled (todo!)
load bin file C:\Windows\system32\ntdll.dll
ntdll hash: E004197EB89EEAA0B097C9F4DA2F9A9E
load bin file C:\Windows\system32\msv1_0.dll
msv1_0.dll hash: ED2D037C307D2BE5C526E8DF48299FCD
http_connect_to_c2( 5.2.77.52 )
[-] HttpSendRequest: Error: 12029
GET failed, retrying <- Failed C2 connection
```

Encrypted (top) and plain text (bottom) Mafalda execution log

We did not discover evidence of functionality within Mafalda for decrypting the strings it encrypts. This suggests that string decryption takes place at Metador's command-and-control servers – a simple yet effective technique for hindering analysis.

Function Parameter Obfuscation

Mafalda often obfuscates numerical function parameters by calculating parameter values prior to function execution using arithmetics and bitwise operations. It may also first calculate a value using arithmetics and bitwise operations. If the computed value does or does not match a predefined value, Mafalda assigns the correct values to the obfuscated parameters. The alternative branch assigns wrong values to the obfuscated parameters.

Mafalda applies this obfuscation approach when it executes the function that the implant uses to decode and decrypt portions of obfuscated strings (labeled `j_str_resolve_sub_18014FE4D` in the figure below).

```
[...]
if ( ((HIDWORD(qword_1802DE238) + dword_1802DE150) ^ dword_1802DE0D4 ^ dword_1802DE1E0)
   + (dword_1802DE134 ^ dword_1802DE224)
   + dword_1802DE154
   + dword_1802DE244 == 0x1BC8A )
{
  v51 = v47 | 0x80;
  v52 = v3 + 752;
  v53 = 0x2DF9B1CF79AEA2F2i64;
}
else
{
  v51 = v47 | 0x40;
  v52 = v3 + 784;
  v53 = 0i64;
}
v54 = j_str_resolve_sub_18014FE4D(v52, v53);
[...]
```

Function parameter obfuscation; *v53* is a portion of an obfuscated string

This obfuscation technique may direct emulation tools to wrong execution branches and function parameter values – analysts may use emulation to automate the decryption and decoding of portions of obfuscated strings across the whole implementation of Mafalda. For example, the `iterateAllPaths` feature of the flare-emu tool attempts to emulate all execution paths to a given function and the function itself. For automated deobfuscation, malware analysts typically use this feature to emulate functions that deobfuscate strings at runtime. When we used the `iterateAllPaths` function to emulate `j_str_resolve_sub_18014FE4D`, Mafalda often directed the tool to the wrong values of the function's obfuscated parameters. This resulted in incorrect string decoding and decryption. In the figure below, `rn` and `9` are incorrectly decoded and decrypted strings.

```
[...]
v5 = qword_1802DF6F8 == 194519434;
*(_DWORD *)(v1 - 95 + 111) = 0;
if ( v5 )
{
  if ( dword_1802DE228 >= (unsigned int)dword_1802DE0E8 )
  {
    v6 = 2;
    v7 = v1 - 104;
    v8 = 226807i64;
  }
  else
  {
    v6 = 1;
    v7 = v3 + 23;
    v8 = 0x6DF1E38CEi64;
  }
  v9 = j_str_resolve_sub_18014FE4D(v7, v8);   // rn
  if ( dword_1802DE09C == 58472 )
  {
    v11 = v6 | 8;
    v10 = v3 - 73;
    v12 = 39i64;
  }
  else
  {
    v10 = v3 - 41;
    v11 = v6 | 4;
    v12 = (unsigned int)dword_1802DE1E8 ^ 0x62C37E6DBC7A8Ci64;
  }
  v13 = j_str_resolve_sub_18014FE4D(v10, v12);// 9
  [...]
```

Incorrect string decoding and decryption

However, when we used the flare-emu `emulateRange` functionality for emulating only specific implementation regions in which Mafalda invokes `j_str_resolve_sub_18014FE4D`, the tool was more accurate in assigning correct function parameter values. This resulted in correct string decoding and decryption. In the figure below, `Sleep` and `kernel32` are correctly decoded and decrypted strings – Mafalda uses these strings to invoke the Sleep function that is implemented in the `kernel32.dll` library file.

```
[...]
v5 = qword_1802DF6F8 == 194519434;
*(_DWORD *)(v1 - 95 + 111) = 0;
if ( v5 )
{
  if ( dword_1802DE228 >= (unsigned int)dword_1802DE0E8 )
  {
    v6 = 2;
    v7 = v1 - 104;
    v8 = 226807i64;
  }
  else
  {
    v6 = 1;
    v7 = v3 + 23;
    v8 = 0x6DF1E38CEi64;
  }
  v9 = (_QWORD *)j_str_resolve_sub_18014FE4D(v7, v8);// Sleep
  if ( HIDWORD(qword_1802DE098) == 58472 )
  {
    v11 = v6 | 8;
    v10 = v3 - 73;
    v12 = 39i64;
  }
  else
  {
    v10 = v3 - 41;
    v11 = v6 | 4;
    v12 = (unsigned int)dword_1802DE1E8 ^ 0x62C37E6DBC7A8Ci64;
  }
  v13 = j_str_resolve_sub_18014FE4D(v10, v12);// kernel32
  [...]
```

Correct string decoding and decryption

Execution Flow Obfuscation

Mafalda is obfuscated at implementation-level such that the compiled code of the implant consists mainly of obfuscated and non-obfuscated code segments. The majority of the non-obfuscated code segments are functions that implement Mafalda functionalities. The obfuscated code segments contain heavily obfuscated code that serves no purpose but to confuse analysis tools and increase cognitive load.

In most cases, Mafalda directs execution to the obfuscated code segments through thunk functions – functions that implement only a single `JMP` (jump) instruction that directs execution to a destination location. An obfuscated code segment ultimately returns execution to a location that is in the relative vicinity of the appropriate thunk function. This location is the beginning of a non-obfuscated code segment — often the prologue of a function that implements Mafalda functionalities. In summary, the obfuscated code segments effectively obfuscate the invocation of non-obfuscated functions.

The figure below depicts an instance of execution flow obfuscation through thunk functions. The thunk function `entryRoutine` directs execution to the location `entryRoutine_0`, which marks the beginning of an obfuscated code segment. This code segment ultimately returns the execution to a non-obfuscated code segment – the prologue of the function `sub_17808D17767`.

```
debug019:0000017808D17758                        ; Attributes: thunk
debug019:0000017808D17758
debug019:0000017808D17758                        ; void entryRoutine()
debug019:0000017808D17758                        public entryRoutine
debug019:0000017808D17758                        entryRoutine proc near
debug019:0000017808D17758 000 E9 DE 5E 10 00     jmp     entryRoutine_0
debug019:0000017808D17758                        entryRoutine endp
debug019:0000017808D17758
debug019:0000017808D1775D                        ; ---------------------------------------------------------------------------
debug019:0000017808D1775D 9B                     wait
debug019:0000017808D1775E 85 54 17 D3            test    [rdi+rdx-2Dh], edx
debug019:0000017808D17762 39 D6                  cmp     esi, edx
debug019:0000017808D17764 04 63                  add     al, 63h ; 'c'
debug019:0000017808D17764                        ; ---------------------------------------------------------------------------
debug019:0000017808D17766 53                     db  53h ; S
debug019:0000017808D17767
debug019:0000017808D17767                        ; =============== S U B R O U T I N E =======================================
debug019:0000017808D17767
debug019:0000017808D17767
debug019:0000017808D17767                        ; __int64 __fastcall sub_17808D17767(__int64, __int64, __int64)
debug019:0000017808D17767                        sub_17808D17767 proc near
debug019:0000017808D17767
debug019:0000017808D17767                        var_3A8= qword ptr -3A8h
debug019:0000017808D17767                        var_398= qword ptr -398h
debug019:0000017808D17767                        var_390= qword ptr -390h
debug019:0000017808D17767                        var_388= qword ptr -388h
debug019:0000017808D17767                        var_378= qword ptr -378h
debug019:0000017808D17767                        var_370= qword ptr -370h
debug019:0000017808D17767                        var_368= qword ptr -368h
debug019:0000017808D17767                        var_358= qword ptr -358h
debug019:0000017808D17767                        var_350= qword ptr -350h
debug019:0000017808D17767                        var_28= byte ptr -28h
debug019:0000017808D17767
debug019:0000017808D17767 000 55                 push    rbp
debug019:0000017808D17768 008 41 54              push    r12
debug019:0000017808D1776A 010 41 55              push    r13
debug019:0000017808D1776C 018 41 56              push    r14
debug019:0000017808D1776E 020 41 57              push    r15
debug019:0000017808D17770 028 48 8D A8 38 FD FF FF lea   rbp, [rax-2C8h]
debug019:0000017808D17777 028 48 81 EC A0 03 00 00 sub   rsp, 3A0h
                                                 [...]
```

```
[...]
debug025:0000017808E1D63B                   entryRoutine_0:
debug025:0000017808E1D63B 66 93             xchg    ax, bx
debug025:0000017808E1D63D 48 C1 C6 00       rol     rsi, 0
debug025:0000017808E1D641 66 93             xchg    ax, bx
debug025:0000017808E1D643 48 8B C4          mov     rax, rsp
debug025:0000017808E1D646 74 0C             jz      short loc_17808E1D654
debug025:0000017808E1D648 7A 03             jp      short loc_17808E1D64D
debug025:0000017808E1D64A 7B 01             jnp     short loc_17808E1D64D
debug025:0000017808E1D64C 59                pop     rcx
debug025:0000017808E1D64D
debug025:0000017808E1D64D                   loc_17808E1D64D:
debug025:0000017808E1D64D
debug025:0000017808E1D64D EB 02             jmp     short loc_17808E1D651
debug025:0000017808E1D64D                   ; ----------------------------
debug025:0000017808E1D64F 9D                db 9Dh
debug025:0000017808E1D650 E3                db 0E3h
debug025:0000017808E1D651                   ; ----------------------------
debug025:0000017808E1D651
debug025:0000017808E1D651                   loc_17808E1D651:
debug025:0000017808E1D651 75 03             jnz     short loc_17808E1D656
debug025:0000017808E1D653 95                xchg    eax, ebp
[...]
```

Execution flow obfuscation through a thunk function

Next, we discuss some of the obfuscation techniques that the developers of Mafalda have applied to the obfuscated code segments.

Purposeless Instruction(s)

The obfuscated code segments in Mafalda contain instructions that serve no purpose in the execution of the code. These instructions exist only to increase the cognitive load when an analyst analyzes the instruction stream. In Mafalda, purposeless instructions are placed sequentially or are intertwined with other instructions.

The table below lists the majority of the purposeless instructions that we encountered in Mafalda’s obfuscated code segments (p denotes an instruction parameter).

Instructions	Description
rol p,0 / ror p,0	Rotates p left or right by 0 bits.
xchg p1, p2 xchg p1, p2	Swaps p1 and p2 two times.
xchg p, p	Swaps p with itself.

pause	Provides a spin-wait loop hint to the processor. The Mafalda developers have placed this instruction very often in the obfuscated code segments to increase cognitive load.
bswap p bswap p	Reverses the byte order of p twice.
push p pop p	First preserves p on the stack and then restores p from the stack without modifying p between these actions.
pushfq popfq	First preserves the RFLAGS register on the stack and then restores RFLAGS from the stack without modifying RFLAGS between these actions.

```
[...]
debug025:0000017808E1D63B 66 93          xchg    ax, bx
debug025:0000017808E1D63D 48 C1 C6 00    rol     rsi, 0
debug025:0000017808E1D641 66 93          xchg    ax, bx
[...]
```

An example of some purposeless instructions in Mafalda

Opaque Predicates

The obfuscated code segments in Mafalda implement simple opaque predicates. They involve first issuing the `cmp` instruction for comparing a value against itself, which always evaluates to TRUE, and then evaluating the `ZF`, `PF`, or the `SF` flag to direct the execution to a given execution branch.

The table below lists the majority of the opaque predicates that we encountered in Mafalda's obfuscated code segments. `p` denotes an instruction parameter and `addr` a memory address mapped to Mafalda: a virtual address or a parameter to a conditional or unconditional jump instruction.

Instructions	Description
cmp p, p JNP/JNZ/JNE/JS [addr1] [addr2]: [. . .]	The branch at address [addr1] is never taken, the branch at address [addr2] is always taken.
cmp p, p JP/JZ/JE/JNS [addr1] [addr2]: [. . .]	The branch at address [addr1] is always taken, the branch at address [addr2] is never taken.

cmp p, p JNP/JNZ/JNE/JS [addr1] JMP [addr2] [addr3]: [. . .]	The branch at address [addr1] is never taken, the branch at address [addr2] is always taken, the branch at address [addr3] is never taken.

The execution branches that are always or never taken may contain any instructions, such as the purposeless instructions mentioned above.

```
[...]
debug025:0000017808E1D6B2 38 FF
debug025:0000017808E1D6B4 75 14
debug025:0000017808E1D6B6 EB 02
[...]
```

```
cmp     bh, bh
jnz     short loc_17808E1D6CA
jmp     short near ptr loc_17808E1D6B9+1
```

An opaque predicate

Unconditional Jump (JMP) Obfuscations

The obfuscated code segments in Mafalda contain instructions that obfuscate unconditional jumps to locations in the memory mapped to Mafalda. This involves:

- Conditional execution based on a flag value in the `RFLAGS` register, for example, the `ZF` or the `PF` flag, such that any of the possible flag values (0 or 1) result in the execution of the code at a given destination location; or
- Use of multiple, instead of one, unconditional jumps (trampolines) to direct execution to a given destination location.

The table below lists the majority of the unconditional jump obfuscations sets that we encountered in Mafalda's obfuscated code segments (`addr` denotes a memory address mapped to Mafalda: a virtual address or a parameter to a conditional or unconditional jump instruction).

Instructions	**Description**
JP [addr1] JNP [addr1] [addr2]: [. . .]	The branch at address [addr1] is always taken, the branch at address [addr2] is never taken.
JS [addr1] JNS [addr1] [addr2]: [. . .]	The branch at address [addr1] is always taken, the branch at address [addr2] is never taken.

JB [addr1] JNB [addr1] [addr2]: [. . .]	The branch at address [addr1] is always taken, the branch at address [addr2] is never taken.
[addr]: call $ + [offset] [. . .] [addr+offset]: [. . .]	Executes the instructions placed at the offset [offset] from the address [addr] where the call instruction resides. The instructions between [addr] and [addr+offset] are never executed.
JMP [addr1] [. . .] [addr2]: JMP [addr3] [. . .] [addr1]:: JMP [addr2] [. . .] [addrN]: JMP [dest_addr] [. . .] [dest_addr]: [. . .]	Directs execution to multiple locations (addresses [addr1] to [addrN]) through trampolines until the final destination location at address [dest_addr] is reached. The instructions between the trampolines are never executed. We observed up to 17 trampolines as part of such an unconditional jump obfuscation.

```
[...]
debug025:0000017808E1D648 7A 03                    jp      short loc_17808E1D64D
debug025:0000017808E1D64A 7B 01                    jnp     short loc_17808E1D64D
debug025:0000017808E1D64C 59                       pop     rcx
debug025:0000017808E1D64D
debug025:0000017808E1D64D                          loc_17808E1D64D:
[...]
```

```
[...]
debug025:0000017808E1D701
debug025:0000017808E1D701 EB 10
debug025:0000017808E1D703
debug025:0000017808E1D703
debug025:0000017808E1D703
debug025:0000017808E1D703 EB 06
debug025:0000017808E1D705
debug025:0000017808E1D705
debug025:0000017808E1D705
debug025:0000017808E1D705 EB 02
debug025:0000017808E1D707
debug025:0000017808E1D707
debug025:0000017808E1D707
debug025:0000017808E1D707 EB 06
debug025:0000017808E1D709
debug025:0000017808E1D709
debug025:0000017808E1D709
debug025:0000017808E1D709 EB FC
debug025:0000017808E1D70B
debug025:0000017808E1D70B
debug025:0000017808E1D70B
debug025:0000017808E1D70B EB F4
debug025:0000017808E1D70D
debug025:0000017808E1D70D
debug025:0000017808E1D70D
debug025:0000017808E1D70D EB 02
debug025:0000017808E1D70F
debug025:0000017808E1D70F
debug025:0000017808E1D70F
debug025:0000017808E1D70F EB F2
debug025:0000017808E1D711
debug025:0000017808E1D711
debug025:0000017808E1D711
debug025:0000017808E1D711 EB F2
debug025:0000017808E1D713
[...]
```

```
loc_17808E1D701:
jmp     short loc_17808E1D713
; ---------------------------------------

loc_17808E1D703:
jmp     short loc_17808E1D70B
; ---------------------------------------

loc_17808E1D705:
jmp     short loc_17808E1D709
; ---------------------------------------

loc_17808E1D707:
jmp     short loc_17808E1D70F
; ---------------------------------------

loc_17808E1D709:
jmp     short loc_17808E1D707
; ---------------------------------------

loc_17808E1D70B:
jmp     short loc_17808E1D701
; ---------------------------------------

loc_17808E1D70D:
jmp     short loc_17808E1D711
; ---------------------------------------

loc_17808E1D70F:
jmp     short loc_17808E1D703
; ---------------------------------------

loc_17808E1D711:
jmp     short loc_17808E1D705
; ---------------------------------------
```

Unconditional jump obfuscations

Conclusion

Mafalda's anti-analysis techniques make the analysis of the malware challenging, which helps the Metador threat actor to delay effective defensive actions against its operations. Metador takes a number of measures at infrastructure- and network-level to hide and protect its operation from defenders. The techniques that this article discusses add to these measures at an executable, malware-implementation level.

By complementing our previous publication on Metador, we hope that this post will encourage collaboration towards further unveiling the mystery of this threat actor.