REKOOBE APT-31 Linux Backdoor Analysis

By techevo

Published: 2024-11-30 · Archived: 2026-04-05 21:40:05 UTC

In this post I will be taking a look at a Linux backdoor known as **REKOOBE**[1]

Reporting suggests this and previous iterations have been used by APT-31 against a variety of victims.

This post will go over both static and dynamic analysis techniques, as well as provide some *primitive* scripts to automate extracting the C2 details.

The sample for this analysis can be found here and here with the SHA1: `23e0c1854c1a90e94cd1c427c201ecf879b2fa78`.

As with previous posts, it might be beneficial to follow along, and hopefully the post is structured in a way that makes that possible.

Output from commands and scripts used for this post can be found in this Github repository.

Static Analysis

The start of any analysis should be to verify what it is that needs analyzing.

Using the `file` [2] command, the output shows the target file is a dynamically linked 64-bit ELF executable.

This is a hopeful start as any imported functions should be visible to us, unless the sample is packed.

```
file rekoobe.elf
```

```
rekoobe.elf: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
   interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.18,
   BuildID[sha1]=025ab2845d244964abc35fb2cffadf388408fa14, stripped
```

One additional take-away from the file output is the “GNU/Linux” version that is referenced: `2.6.18`.

Whilst compiling code on modern compilers will generally result in older versions being targeted for compatibility reasons, this version is well beyond expected values.

There are at least two reasons for this:

1) The binary was compiled on a very old Linux system.

2) The binary is designed to be deployed on potentially very old Linux systems.

For reference, version 2.6.10 of the Linux kernel was released in 2006[3]

The output of the `strings` command also hints this sample was compiled using a version of GCC from 2012[4].

```
strings rekoobe.elf
```

```
 ...
GCC: (GNU) 4.4.7 20120313 (Red Hat 4.4.7-4)
...
```

Before wading into the depths of functions, reviewing the required shared libraries shows that anything imported is pretty standard. No additional functionality in custom shared libraries as is sometimes the case with Windows malware.

```
readelf -d rekoobe.elf
```

```
 Dynamic section at offset 0x14028 contains 23 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libutil.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [librt.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libpthread.so.0]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
...
```

I have radare2[5] installed so will be making use of various tools from the framework. You do **not** have to use the same tools, any tools for interrogating ELF files should work fine.

Using `rabin2` to list the *imports*, shows that this sample makes use of the `execv` function, which allows execution of arbitrary system commands.

The output below is truncated, the full output can be viewed here.

In addition to `execv`, the output also showed `execl`, `recv`, `setsockopt`, `bind`, and `openpty`, which all seem a little suspicious. These functions resemble the basis for a backdoor, and certainly should raise some eyebrows.

```
rabin2 -i rekoobe.elf
```

```
 [Imports]
nth vaddr      bind   type   lib name
―――――――――――――――――――――――――――――――――――――――――――――――――――――
```

```
1   0x00401790 GLOBAL FUNC       daemon
2   0x004017a0 GLOBAL FUNC       chmod
3   0x004017b0 GLOBAL FUNC       dup2
4   0x004017c0 GLOBAL FUNC       execv
5   0x004017d0 GLOBAL FUNC       memset
6   0x004017e0 GLOBAL FUNC       setsid
7   0x004017f0 GLOBAL FUNC       shutdown
...
```

You shouldn't take my word for it either!

Rather than going through every imported function and reading the documentation, Capa[6] provides a nice way to scan for functionality of binaries.

```
./capa ./rekoobe.elf
```

MBC Objective	MBC Behavior
ANTI-STATIC ANALYSIS	Executable Code Obfuscation::Argument Obfuscation [B0032.020]
	Executable Code Obfuscation::Stack Strings [B0032.017]
COMMAND AND CONTROL COMMUNICATION	C2 Communication::Receive Data [B0030.002]
	DNS Communication::Resolve [C0011.001]
	Socket Communication::Create TCP Socket [C0001.011]
	Socket Communication::Get Socket Status [C0001.012]
	Socket Communication::Receive Data [C0001.006]
	Socket Communication::Send Data [C0001.007]
	Socket Communication::Set Socket Config [C0001.001]
CRYPTOGRAPHY	Cryptographic Hash::SHA1 [C0029.002]
	Encrypt Data::AES [C0027.001]
	Encrypt Data::RC4 [C0027.009]
	Generate Pseudo-random Sequence::RC4 PRGA [C0021.004]
DATA	Encode Data::Base64 [C0026.001]
	Encode Data::XOR [C0026.002]
DEFENSE EVASION	Obfuscated Files or Information::Encoding-Standard Algorithm [E1027.m02]
	Obfuscated Files or Information::Encryption-Standard Algorithm [E1027.m05]
DISCOVERY	File and Directory Discovery [E1083]
	System Information Discovery [E1082]
FILE SYSTEM	Create Directory [C0046]
	Delete File [C0047]
	Move File [C0063]
	Read File [C0051]
	Set File Attributes [C0050]
	Writes File [C0052]
IMPACT	Remote Access::Reverse Shell [B0022.001]
PROCESS	Create Process [C0017]
	Create Thread [C0038]

Figure 1: Capa output for Rekoobe sample.

The output in *Figure 1* shows **Remote Access::Reverse Shell**, which pretty much sums it up, case closed.

The Capa output shows that there are references to RC4 and AES encryption routines, which might be interesting to take a look into. The full output from Capa can be found here.

Let's start exploring the binary in a disassembler.

The `main` symbol is exported so should be quickly identifiable in other tools such as Ghidra[7] or IDA.

The following command will **p**rint **d**issasembly located at the `main` **f**unction.

```
r2 -AA -q -c 'pdf @ main;' rekoobe.elf
```

Figure 2: Radare2 main routine.

The output shows that the process first calls the imported symbol `daemon`, allowing the execution to continue in the background. A function labeled `fcn.00404568` is called, and the return value in `EAX` is checked before calling another function labeled `fcn.00404415`.

Static Analysis: fcn.00404568

Starting with `fcn.00404568` the command below prints the first 27 instructions of the function. Why 27? Because it looked nice in the screen shot.

```
r2 -AA -q -c 'pd 27 @ fcn.00404568' rekoobe.elf
```

```
            ; CALL XREF from main @ 0x404974(x)
┌ 1007: fcn.00404568 (int64_t arg1);
│ `- args(rdi) vars(42:sp[0x128..0x1a96])
│           0x00404568      4156           push r14
│           0x0040456a      4155           push r13
│           0x0040456c      4154           push r12
│           0x0040456e      55             push rbp
│           0x0040456f      53             push rbx
│           0x00404570      4881ec701a..   sub rsp, 0x1a70
│           0x00404577      4889fb         mov rbx, rdi              ; arg1
│           0x0040457a      488dbc2470..   lea rdi, [dest]
│           0x00404582      b800000000     mov eax, 0
│           0x00404587      b920000000     mov ecx, 0x20             ; 32
│           0x0040458c      f348ab         rep stosq qword [rdi], rax
│           0x0040458f      488dbc2470..   lea rdi, [var_1870h]
│           0x00404597      b120           mov cl, 0x20              ; 32
│           0x00404599      f348ab         rep stosq qword [rdi], rax
│           0x0040459c      c684246018..   mov byte [var_1860h], 0x72  ; 'r'
│                                                                  ; [0x1860:1]=255 ; string "r0st__"
│           0x004045a4      c684246118..   mov byte [var_1861h], 0x30  ; '0'
│                                                                  ; [0x1861:1]=255
│           0x004045ac      c684246218..   mov byte [var_1862h], 0x73  ; 's'
│                                                                  ; [0x1862:1]=255
│           0x004045b4      c684246318..   mov byte [var_1863h], 0x74  ; 't'
│                                                                  ; [0x1863:1]=255
│           0x004045bc      c684246418..   mov byte [var_1864h], 0x40  ; [0x1864:1]=255
│           0x004045c4      c684246518..   mov byte [var_1865h], 0x23  ; '#'
│                                                                  ; [0x1865:1]=255
│           0x004045cc      c684246618..   mov byte [var_1866h], 0x24  ; '$'
│                                                                  ; [0x1866:1]=255
│           0x004045d4      c684246718..   mov byte [var_1867h], 0    ; [0x1867:1]=255
│           0x004045dc      803d5d0121..   cmp byte [0x00614740], 0    ; [0x614740:1]=1
│       ┌─< 0x004045e3      0f8459030000   je 0x404942
│       │   |
│       │   0x004045e9      488dac2470..   lea rbp, [var_1870h]
│       │   0x004045f1      0fb7154901..   movzx edx, word [0x00614741] ; [0x614741:2]=12
│       │   0x004045f8      66895500       mov word [rbp], dx
```

Figure 3: Radare2 fcn.00404568 disassembly.

Starting at `0x0040459c`, there is a sequence of 8 `mov byte` instructions. The 8 bytes are ASCII characters depicted as shown:

The `\0` (NULL) byte terminates the character array.

```
 0x72 = r
0x30 = 0
0x73 = s
0x74 = t
0x40 = @
0x23 = #
0x24 = $
0x00 = '\0'
```

Following the `mov byte` instructions there is then a value comparison with a byte located at `0x00614740` located in the `.data` section of the ELF file.

If the value is set to `0`, then the `je`, jumps to the end of the function before returning.

This value turns out to be quite important later on…

The Capa output told us there was stack strings in use, and this is one of them. At this stage it is not important *what* this string is used for, however if there are more, it would be nice to recover them.

I created a script to recover these strings, which you can view here The output shown is truncated. A copy of the full output can be viewed here

```
python3 ./recover_stack_strings.py ./rekoobe.elf
```

```
 %02x
%02X
r0st@#$
/etc//etc/issue.net
/etc/issue
/proc/ve/etc/issue.net
/etc/issue
/proc/version
r.
/
.
/
%s/%s
.
..
rb
a+b
a+b
/usr/usr/include/sdfwex.h
/tmp/.l
...
```

Whilst the output is far from perfect and not production ready, you can see it located the `r0st@#$` string correctly, as well as some interesting file paths.

Continuing on, a WORD (2 bytes) is read from `0x00614741` into `EDX` with the value of `12`.

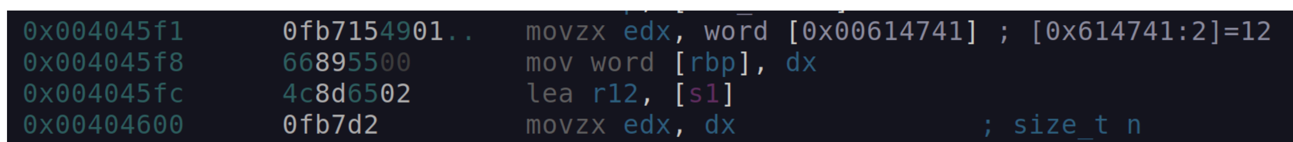

Figure 4: Radare2 size parameter read.

Zooming out, shows the use of this parameter more clearly.

A memory address `0x614743` is stored into `ESI`, before both are passed into `memcpy`, to copy **12** bytes from the location stored in `ESI` into a buffer labeled `s1`.

After the `memcpy` function returns the stack string we recovered earlier located at `[var_1860h]`, the value **12** and the address of the `s1` buffer as passed to a function called `fcn.00402af9`.

```
0x004045f1      0fb7154901..   movzx edx, word [0x00614741] ; [0x614741:2]=12
0x004045f8      66895500       mov word [rbp], dx
0x004045fc      4c8d6502       lea r12, [s1]
0x00404600      0fb7d2         movzx edx, dx                ; size_t n
0x00404603      be43476100     mov esi, 0x614743            ; 'CGa' ; "U<_\xff\xec\x8aR\xc96\xc8\xd9\x02" ;
*s2
0x00404608      4c89e7         mov rdi, r12                 ; void *s1
0x0040460b      e820d4ffff     call sym.imp.memcpy          ; void *memcpy(void *s1, const void *s2, size_t

0x00404610      488d942460..   lea rdx, [var_1860h]         ; int64_t arg3
0x00404618      0fb77500       movzx esi, word [rbp]        ; signed int64_t arg2
0x0040461c      4c89e7         mov rdi, r12                 ; int64_t arg1
0x0040461f      e8d5e4ffff     call fcn.00402af9
0x00404624      488dac2470..   lea rbp, [dest]
0x0040462c      4c89e6         mov rsi, r12                 ; const char *src
0x0040462f      4889ef         mov rdi, rbp                 ; char *dest
0x00404632      e8c9d4ffff     call sym.imp.strcpy          ; char *strcpy(char *dest, const char *src)
```

Figure 5: Radare2 string operations.

Static Analysis: fcn.00402af9

The functionality of `fcn.00402af9` is an implementation of the RC4[8] cipher.

The parameters passed to `fcn.00402af9`, are shown in the function prototype.

```
void fcn.00402af9(
 char    *buffer,
 int64_t length,
 char    *key
)
```

The `buffer` contains the ciphered data on input, and on output contains the original clear-text data.

The length contains the length of the data stored in the buffer, as `\0` (NULL) bytes will not be used to terminate the data.

Finally, the `key`, in this `call` is the `r0st@#$` string.

We can quickly test this out taking the various inputs and using the RC4 CyberChef recipe.

First extract the 12 input bytes from `0x614743.`

```
r2 -AA -q -c 'px0 12 @ 0x614743' rekoobe.elf
```

```
553c5fffec8a52c936c8d902
```

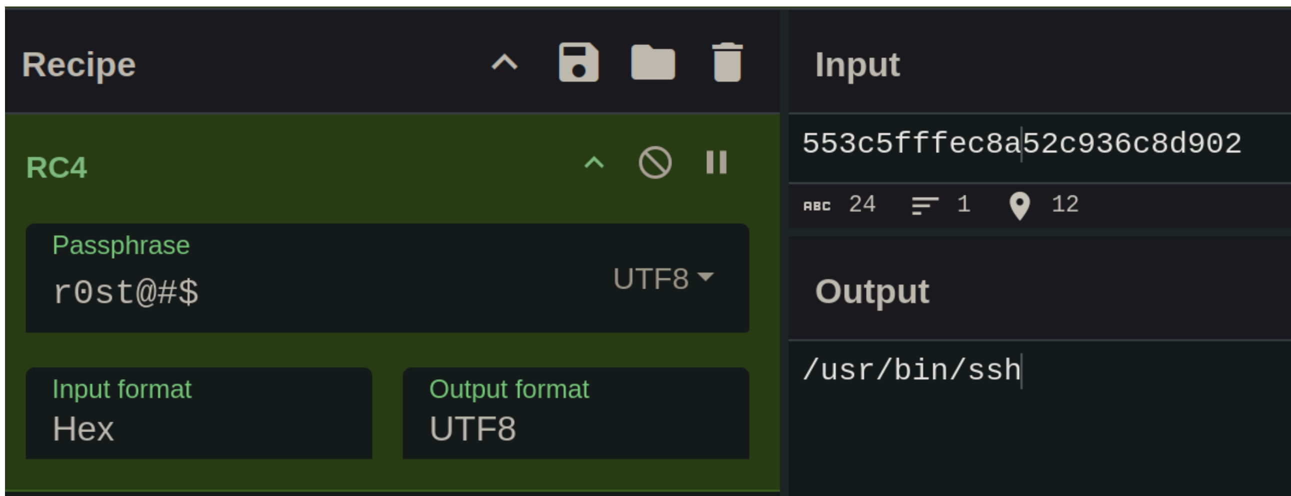

Figure 6: CyberChef RC4

As the RC4 code was its own function, we can find cross-references to this routine to locate more values being decrypted that might be useful in later analysis.

The `axt` command shows there are 10 calls to this RC4 function, which are worthy of further exploration.

```
r2 -AA -q -c 'axt @ 0x00402af9;' rekoobe.elf
```

```
 fcn.0040225c 0x4022d3 [CALL:--x] call fcn.00402af9
fcn.00404568 0x40461f [CALL:--x] call fcn.00402af9
fcn.00404b27 0x404dc8 [CALL:--x] call fcn.00402af9
fcn.00404b27 0x404ea4 [CALL:--x] call fcn.00402af9
fcn.00404f06 0x405130 [CALL:--x] call fcn.00402af9
fcn.00404f06 0x40525d [CALL:--x] call fcn.00402af9
fcn.0040ba91 0x40bad4 [CALL:--x] call fcn.00402af9
fcn.0040ba91 0x40bb10 [CALL:--x] call fcn.00402af9
fcn.0040bbe3 0x40bc5e [CALL:--x] call fcn.00402af9
fcn.0040bbe3 0x40bcf2 [CALL:--x] call fcn.00402af9
```

Static Analysis: fcn.00404568 (continued)

Returning (pun intended) back to `fcn.00404568`, we now have a decrypted string:

```
/usr/bin/ssh
```

Figure 6 shows a call to `strcpy` (`0x0040467f`), which shows the value stored in `RBP` moved into `RSI` as the source of the string copy operation. The screen shot shows that `RBP` contains the buffer address used to decrypt the string using the RC4 decryption routine.

```
0x0040461f      e8d5e4ffff      call fcn.00402af9
0x00404624      488dac2470..    lea rbp, [dest]
0x0040462c      4c89e6          mov rsi, r12                ; const char *src
0x0040462f      4889ef          mov rdi, rbp                ; char *dest
0x00404632      e8c9d4ffff      call sym.imp.strcpy         ; char *strcpy(char *dest, const char *src)
0x00404637      4c8b03          mov r8, qword [rbx]
0x0040463a      48c7c6ffff..    mov rsi, 0xffffffffffffffff
0x00404641      4c89c7          mov rdi, r8
0x00404644      b800000000      mov eax, 0
0x00404649      4889f1          mov rcx, rsi
0x0040464c      f2ae            repne scasb al, byte [rdi]
0x0040464e      48f7d1          not rcx
0x00404651      488d1431        lea rdx, [rcx + rsi]
0x00404655      4889ef          mov rdi, rbp
0x00404658      4889f1          mov rcx, rsi
0x0040465b      f2ae            repne scasb al, byte [rdi]
0x0040465d      4889ce          mov rsi, rcx
0x00404660      48f7d6          not rsi
0x00404663      4883ee01        sub rsi, 1
0x00404667      4839f2          cmp rdx, rsi
0x0040466a      7222            jb 0x40468e
|
0x0040466c      be00000000      mov esi, 0                  ; int c
0x00404671      4c89c7          mov rdi, r8                 ; void *s
0x00404674      e857d1ffff      call sym.imp.memset         ; void *memset(void *s, int c, size_t n)
]> pdfrOBA|     ||  0x00404679      4889ee          mov rsi, rbp                ; const char *src
0x0040467c      488b3b          mov rdi, qword [rbx]        ; char *dest
0x0040467f      e87cd4ffff      call sym.imp.strcpy         ; char *strcpy(char *dest, const char *src)
```

Figure 6: Radare2 string operations.

Using the Ghidra plugin[9] for Radare2 with the command `pdga` , *Figure 7* shows the destination more clearly, as `*param_1` .

```
0x00404674 call sym.imp.memset                |          *(puVar11 + -8) = 0x404679;
0x00404674 call sym.imp.memset                |          sym.imp.memset(pcVar21, 0);
0x0040467c mov rdi, qword [rbx]               |          pcVar21 = *param_1;
0x0040467f call sym.imp.strcpy                |          *(puVar11 + 0 + -8) = 0x404684;
0x0040467f call sym.imp.strcpy                |          sym.imp.strcpy(pcVar21, pcVar20);
0x00404956 ret                                |          return 1;
                                              |      }
```

Figure 7: Radare2 Ghidra disassemble .

Going back to see what was passed into this function shown in *Figure 8*, we see from `main` that `argv` is the only parameter supplied (`0x00404971`).

```
    0x00404957      53              push rbx
    0x00404958      4881ec100a..    sub rsp, 0xa10
    0x0040495f      4889f3          mov rbx, rsi                ; argv
    0x00404962      be00000000      mov esi, 0
    0x00404967      bf00000000      mov edi, 0
    0x0040496c      e81fceffff      call sym.imp.daemon
    0x00404971      4889df          mov rdi, rbx                ; int64_t arg1
    0x00404974      e8effbffff      call fcn.00404568
    0x00404979      85c0            test eax, eax
┌─< 0x0040497b      7408            je 0x404985
│   |
│   0x0040497d      4889e7          mov rdi, rsp                ; int64_t arg1
│   0x00404980      e890faffff      call fcn.00404415
│   |
│   ; CODE XREF from main @ 0x40497b(x)
└─> 0x00404985      b801000000      mov eax, 1
    0x0040498a      4881c4100a..    add rsp, 0xa10
    0x00404991      5b              pop rbx
    0x00404992      c3              ret
```

Figure 8: Radare2 call fcn.00404568 .

Overwriting `argv` will for all intents and purposes alter the process name, allowing the process to avoid detection. When executed, this process will appear to be named `/usr/bin/ssh` , when commands such as `ps` and `top` are used to inspect the system.

This function contains more capabilities to copy and rename itself based on the value that is checked, however in this sample, it returns to `main` setting the return code to `1` which allows execution to continue into `fcn.00404415` shown in *Figure 8*.

Static Analysis: fcn.0040225c

From the `main` function, `fcn.00404415` is called which performs some value checks before calling `fcn.0040225c` .

The start of the function builds the same stack string `r0st@#$` as previously seen, and calls the same RC4 wrapper. The input length is stored at `0x6144e0` and contains decimal **42**.

The 42 bytes of input is located at `0x6144e2` , again in the `.data` section.

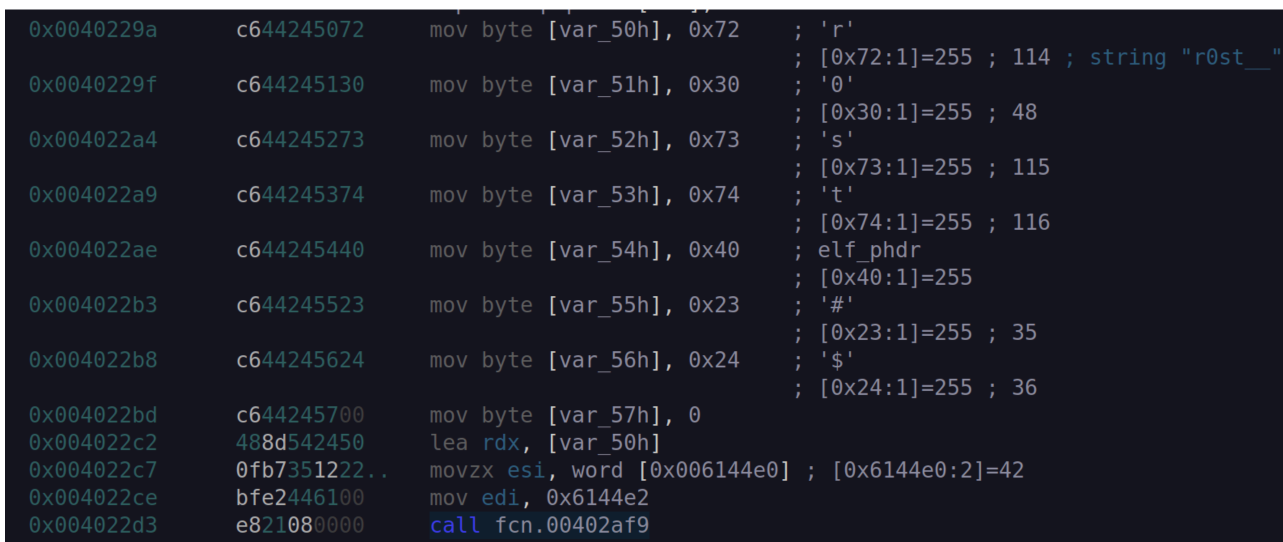

Figure 9: Radare2 decrypt configuration.

The following command will extract the hexadecimal stream to be decrypted.

```
r2 -AA -q -c 'px0 42 @ 0x6144e2' rekoobe.elf
```

```
42671ebcfbc60295378a98593b13a7e9721f03aac47781891b5f10926882a5239c6d961129b3d32ca620
```

Using the same CyberChef recipe as before, it shows an IPv4 address and port, as well as some binary flag values.

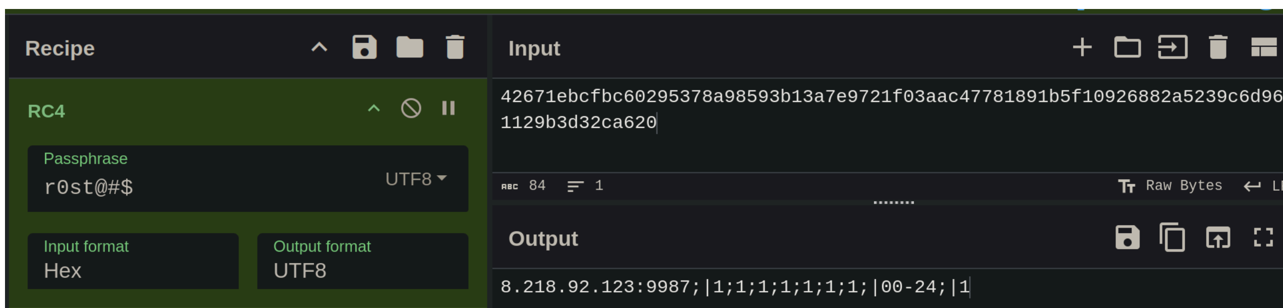

Figure 10: CyberChef decrypt configuration.

There are 4 sections in this configuration, delimited by `|` values. These sections are identified using the `strstr` function by the malware.

Configurations options are then further split using `;`, before being parsed using `strtol` to convert the string values “1” to a long integer.

Static Analysis: fcn.00401db4

Before heading into some dynamic analysis, I thought it was worth highlighting the function `fcn.00401db4`.

The script to recover the stack strings highlighted some interesting file paths common on Linux systems. This function is where they reside and it responsible for collecting information regarding the infected system.

The stack strings, reveal the following file paths:

- `/etc/issue.net`
- `/etc/issue`
- `/proc/version`

First `/etc/issue.net` is passed to `fopen` and if that fails then `/etc/issue` is opened. The procfs file `/proc/version` is opened and `strstr` us used to locate the value `x86_64`, which determines the host architecture.

A call to `gethostname` is fairly self explanatory, gathering the hostname.

A call to `getifaddrs` returns a structure containing a linked-list, which is traversed gathering the IP address from each network interface.

Dynamic Analysis

From the static analysis, the command and control IPv4 was determined. Unfortunately at the time of analysis no response on the provided port was returned.

To see how the sample would have interacted with the server, we need to provide a route to the IP address: `8.218.92[.]123`.

This can be achieved using the `lo` loopback interface as shown.

```
sudo ip addr add 8.218.92[.]123 dev lo
```

Once the IP address has been added, a `nc` netcat listener can be setup on the required port.

```
nc -l -p 9987 > output.bin
```

Using the `ltrace` [10] program, it is possible to trace the library calls of this dynamically linked executable, saving the output into the `ltrace.out` file. A copy of the full output can be found here

```
ltrace -fbS -o ltrace.out ./rekoobe.elf
```

Figure 11 shows the output of `ltrace` revealing the configuration strings.

```
memcpy(0x7ffd7fecd2e2, "U<_\377\354\212R\3116\310\331\002", 12)          = 0x7ffd7fecd2e2
strcpy(0x7ffd7fecd3e0, "/usr/bin/ssh")                                   = 0x7ffd7fecd3e0
memset(0x7ffd7fecf47a, '\0', 13)                                         = 0x7ffd7fecf47a
strcpy(0x7ffd7fecf47a, "/usr/bin/ssh")                                   = 0x7ffd7fecf47a
memcpy(0x7ffd7fecd190, "8.218.92.123:9987;|1;1;1;1;1;1;1"..., 42)       = 0x7ffd7fecd190
strstr("8.218.92.123:9987;|1;1;1;1;1;1;1"..., "|")                      = "|1;1;1;1;1;1;1;|00-24;|1"
strstr("1;1;1;1;1;1;1;|00-24;|1", "|")                                  = "|00-24;|1"
strstr("00-24;|1", "|")                                                 = "|1"
memcpy(0x7ffd7fecce70, "8.218.92.123:9987;", 18)                        = 0x7ffd7fecce70
strtol(0x7ffd7fecce7d, 0, 10, 18)                                       = 9987
memcpy(0x7ffd7fecd514, "8.218.92.123", 12)                              = 0x7ffd7fecd514
memcpy(0x7ffd7fecce70, "1;1;1;1;1;1;1;", 14)                            = 0x7ffd7fecce70
memcpy(0x7ffd7fecce70, "00-24;", 6)                                     = 0x7ffd7fecce70
strstr("00-24;", "-")                                                   = "-24;"
strtol(0x7ffd7fecce73, 0, 10, 6)                                        = 24
strtol(0x7ffd7fecce70, 0, 10, 6)                                        = 0
strtol(0x7ffd7fecd1b9, 0, 10, 1)                                        = 1
```

Figure 11: ltrace decrypt configuration.

Figure 12 shows the various files being opened to gather information regarding the host. It also shows a call to the `socket` and `bind` functions, indicating a listing port being established.

```
fopen("/etc/issue.net", "rb" <unfinished ...>
SYS_openat(0xffffff9c, 0x7ffd7feca7d0, 0, 0)                             = 4
<... fopen resumed> )                                                    = 0x14072c0
fgets( <unfinished ...>
SYS_newfstatat(4, 0x7fb0c07ff44f, 0x7ffd7feca590, 4096)                  = 0
SYS_read(4, "Ubuntu 22.04.5 LTS\n", 4096)                                = 19
<... fgets resumed> "Ubuntu 22.04.5 LTS\n", 200, 0x14072c0)              = 0x7ffd7feca8b0
fclose(0x14072c0 <unfinished ...>
SYS_close(4)                                                             = 0
<... fclose resumed> )                                                   = 0
strcpy(0x7ffd7fecd250, "Ubuntu 22.04.5 LTS")                             = 0x7ffd7fecd250
fopen("/proc/version", "rb" <unfinished ...>
SYS_openat(0xffffff9c, 0x7ffd7feca7b0, 0, 0)                             = 4
<... fopen resumed> )                                                    = 0x14072c0
fgets( <unfinished ...>
SYS_newfstatat(4, 0x7fb0c07ff44f, 0x7ffd7feca590, 4096)                  = 0
SYS_read(4, "Linux version 5.15.0-27-generic "..., 1024)                 = 170
<... fgets resumed> "Linux version 5.15.0-27-generic "..., 200, 0x14072c0)   = 0x7ffd7feca7e0
fclose(0x14072c0 <unfinished ...>
SYS_close(4)                                                             = 0
<... fclose resumed> )                                                   = 0
strstr("Linux version 5.15.0-27-generic "..., "x86_64")                  = nil
gethostname( <unfinished ...>
SYS_uname(0x7ffd7feca5b0)                                                = 0
<... gethostname resumed> "malware", 256)                                = 0
getifaddrs(0x7ffd7feca778, 0x7ffd7feca5f7, 8, 0 <unfinished ...>
SYS_socket(16, 0x80003, 0, 0x7fb0c063b880)                               = 4
SYS_bind(4, 0x7ffd7feca65c, 12, 0x7fb0c063b880)                          = 0
SYS_getsockname(4, 0x7ffd7feca65c, 0x7ffd7feca658, 0x7fb0c063b880)       = 0
SYS_sendto(4, 0x7ffd7feca630, 20, 0)                                     = 20
```

Figure 12: ltrace decrypt configuration.

The dynamic analysis confirms the findings from the static analysis.

In a slightly modified lab setup, I was able to capture the network communications between the malware and the C2 server.

The PCAP file is available here, and shows that 548 bytes were sent over the TCP socket. The data in both directions is binary data, rather than encapsulated in HTTP.

From the analysis performed, both the process name and configuration string were stored in the `.data` section.

Using `radare2`, locating the `.data` virtual address, and printing the hexdump shows the encrypted strings.

```
iS~.data
```

```
s 0x006144c0
```

```
pxs 810
```

```
[0x006144c0]> iS~.data
15   0x0000d760   0x50b0 0x0040d760   0x50b0 -r-- PROGBITS    .rodata
24   0x000144c0    0x3e0 0x006144c0    0x3e0 -rw- PROGBITS    .data
[0x006144c0]> s 0x006144c0
[0x006144c0]> pxs 810
- offset -   C0C1 C2C3 C4C5 C6C7 C8C9 CACB CCCD CECF  0123456789ABCDEF
0x006144c0   0000 0000 0000 0000 0000 0000 0000 0000  ................
0x006144d0   0000 0000 0000 0000 0000 0000 0000 0000  ................
0x006144e0   2a00 4267 1ebc fbc6 0295 378a 9859 3b13  *.Bg......7..Y;.
0x006144f0   a7e9 721f 03aa c477 8189 1b5f 1092 6882  ..r....w..._..h.
0x00614500   a523 9c6d 9611 29b3 d32c a620 0000 0000  .#.m..)..,. ....
0x00614510   0000 0000 0000 0000 0000 0000 0000 0000  ................
 ...
0x00614520   0000 0000 0000 0000 0000 0000 0000 0000  ................
0x00614730   0000 0000 0000 0000 0000 0000 0000 0000  ................
0x00614740   010c 0055 3c5f ffec 8a52 c936 c8d9 0200  ...U<_...R.6....
0x00614750   0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Figure 13: Radare2 .data section hex dump.

Using this information, I have developed a configuration extractor which can be found here

Executing the script, and providing the RC4 key outputs JSON document containing the C2 details.

```
python3 ./rekoobe_config.py rekoobe.elf r0st@#$
```


```
{
   "c2": "8.218.92.123:9987",
   "flags": {
       "unknown_0": 1,
       "unknown_1": 1,
       "unknown_2": 1,
       "unknown_3": 1,
       "unknown_4": 1,
       "unknown_5": 1,
       "unknown_6": 1
   },
   "hours": "00-24",
```

```
    "process_change": 1,
    "process_name": "/usr/bin/ssh",
    "unknown": 1
}
```


Conclusion

In this post we have explored the initial workings of the **REKOOBE** backdoor, identifying how the command and control details are retrieved and shown a Python script to extract the details.

There is more to this sample, however the internals of this backdoor have been researched in prior work. Some notable research from AhnLab and hunt.io among others.

If you enjoyed reading or learnt something new, let me know!

You can find me on Twitter (currently known as X) as well as BlueSky.

Until next time, keep evolving.

techevo

References

1. https://malpedia.caad.fkie.fraunhofer.de/details/elf.rekoobe ↩
2. https://linux.die.net/man/1/file ↩
3. https://kernelnewbies.org/Linux_2_6_18 ↩
4. https://gcc.gnu.org/gcc-4.4/ ↩
5. https://rada.re/n/ ↩
6. https://linux.die.net/man/3/execv ↩
7. https://github.com/mandiant/capa ↩
8. https://en.wikipedia.org/wiki/RC4 ↩
9. https://github.com/radareorg/r2ghidra ↩
10. https://man7.org/linux/man-pages/man1/ltrace.1.html ↩

Source: https://blog.techevo.uk/analysis/linux/2024/11/30/rekoobe-apt31-linux-backdoor.html