

Reverse Engineering a Suspicious Program: An Investigation into its Functionality and Potential Threats

Authors: Camila Fonseca, Rodrigo Lima

DETI, University of Aveiro

June 16, 2023

Contents

1	Introduction	2
2	Program Analysis	3
2.1	Initial Probing	3
2.2	Static Analysis	3
2.3	Dynamic Analysis	6
2.3.1	Network Analysis/Packet Capture with Wireshark	6
2.3.2	Strace	6
2.4	Conclusion	10
3	Malware Analysis	11
3.1	Main Function	11
3.2	Process Directory Function	11
3.3	Exfiltrate Files Function	14
3.4	Exfiltrate File Contents Function	14
3.5	Encrypt Files Function	14
3.6	Encrypt Or Decrypt File Function	15
3.7	Files recovery	16
3.8	Conclusion	17
4	Conclusions and Recommendations	19

1 Introduction

This report aims to detail the steps taken in the reverse engineering of a provided executable, possibly a malware sample, with the aim of understanding if it is a malicious program, mapping out its functionality, and if possible, using the knowledge of its inner workings to undo possible damages it may have caused, such as decrypting files.

2 Program Analysis

2.1 Initial Probing

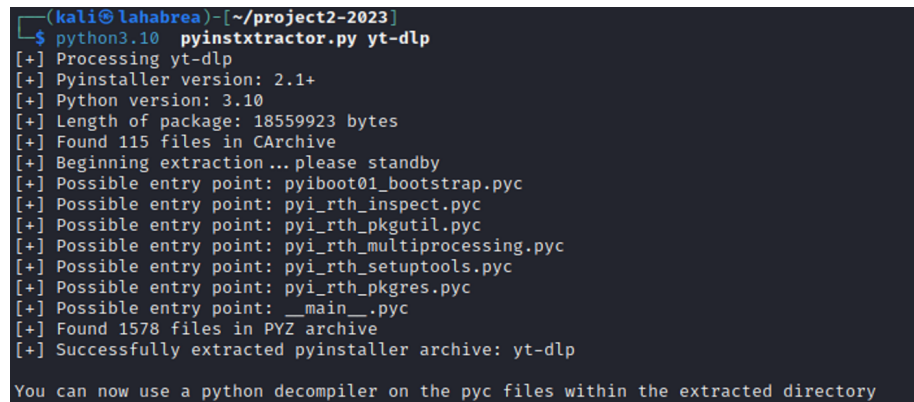
An initial analysis of the **yt-dlp** file shows it to be a stripped x86-64 executable (an ELF). Opening it up in ghidra reveals a very non-standard ELF, with no readily identifiable `main()` function nor any obvious entry points and unusual code structures such as function calls with an exceedingly high amount of arguments.

However, in the program memory/data segments there are a number of references to python libraries with respective functions such as `cryptodome`, as well as a reference to `libpython3.10.so`. These strings, together with the program very clearly not being a regular binary with its odd structure, lead to the suspicion of this being a compiled Python executable.

With this in mind, static analysis is the next step taken as an initial approach to understanding the program behaviour.

2.2 Static Analysis

To get started with the analysis work, it is needed to unpack the compiled binary. For this end, the tool **pyinstxtractor**[?] can be used, with care to use the same python version as the binary does, which in this case is thought to be 3.10.



```
(kali@lahabrea) ~/project2-2023
$ python3.10 pyinstxtractor.py yt-dlp
[+] Processing yt-dlp
[+] Pyinstaller version: 2.1+
[+] Python version: 3.10
[+] Length of package: 18559923 bytes
[+] Found 115 files in CArchive
[+] Beginning extraction... please standby
[+] Possible entry point: pyiboot01_bootstrap.pyc
[+] Possible entry point: pyi_rth_inspect.pyc
[+] Possible entry point: pyi_rth_pkgutil.pyc
[+] Possible entry point: pyi_rth_multiprocessing.pyc
[+] Possible entry point: pyi_rth_setuptools.pyc
[+] Possible entry point: pyi_rth_pkgres.pyc
[+] Possible entry point: __main__.pyc
[+] Found 1578 files in PYZ archive
[+] Successfully extracted pyinstaller archive: yt-dlp

You can now use a python decompiler on the pyc files within the extracted directory
```

Figure 1: Unpacking the ELF

After unpacking, we are left with .pyc files, C shared libraries (.so), an archive file containing python's base libraries (base_library.zip) and an executable archive that contains the code that is actually executed by yt-dlp.

```
(kali@lahabrea)~/project2-2023/pydataextract_test/yt-dlp_extracted/PYZ-00.pyz_extracted
$ ls
aix_support.pyc      _compression.pyc  ftplib.pyc        markupbase.pyc    _py_abc.pyc       sitecustomize.pyc  tracemalloc.pyc
argparse.pyc         concurrent.pyc     _future_.pyc      mimetypes.pyc     py_compile.pyc    site.pyc           tty.pyc
ast.pyc              configparser.pyc   getopt.pyc        multiprocessing    pycparser         socket.pyc         typing.pyc
asyncio.pyc          contextlib.pyc     getpass.pyc       mutagen           _pydecimal.pyc   socketserver.pyc  uuid.pyc
base64.pyc           copy.pyc           glob.pyc          netrc.pyc         pydoc_data        ssl.pyc            uu.pyc
bdb.pyc              cryptodome.pyc     gzip.pyc          nturl2path.pyc   pydoc.pyc         statistics.pyc     webbrowser.pyc
bisect.pyc           csv.pyc            hashlib.pyc       numbers.pyc       queue.pyc          stringprep.pyc    websockets
_bootsubprocess.pyc  ctypes.pyc         hmac.pyc          opcode.pyc        quopri.pyc        string.pyc        xml
brotli.pyc          dataclasses.pyc   http.pyc          optparse.pyc      random.pyc        _strptime.pyc    xmlrpc
bz2.pyc             datetime.pyc      importlib.pyc     _osx_support.pyc  _strptime.pyc     subprocess.pyc    yt_dlp
calendar.pyc        decimal.pyc       inspect.pyc       pickle.pyc        sysconfigdata__x86_64-linux-gnu.pyc  zipfile.pyc
certifi.pyc         distutils.pyc     json.pyc          pkg_resources     selectors.pyc      sysconfig.pyc     zipimport.pyc
cffi.pyc            distutils_hack.pyc  logging.pyc       platform.pyc      setuptools         tarfile.pyc
cgi.pyc             email.pyc          lzma.pyc          plistlib.pyc      shlex.pyc         tempfile.pyc
cmd.pyc             fileinput.pyc      lzma.pyc          pplistlib.pyc    shutil.pyc        textwrap.pyc
code.pyc            fractions.pyc       lzma.pyc          pprint.pyc        signal.pyc         _threading_local.pyc
compat_pickle.pyc   _compat_pickle.pyc  lzma.pyc          pty.pyc           _sitebuiltins.pyc  threading.pyc
```

Figure 2: Extracted Files

This PYZ file is just an executable zip/archive, and as such it too can be unzipped in order to have its contents analyzed.

```
(kali@lahabrea)~/project2-2023
$ python3.10 pyinstxtractor.py yt-dlp
[+] Processing yt-dlp
[+] Pyinstaller version: 2.1+
[+] Python version: 3.10
[+] Length of package: 18559923 bytes
[+] Found 115 files in CArchive
[+] Beginning extraction... please standby
[+] Possible entry point: pyiboot01_bootstrap.pyc
[+] Possible entry point: pyi_rth_inspect.pyc
[+] Possible entry point: pyi_rth_pkgutil.pyc
[+] Possible entry point: pyi_rth_multiprocessing.pyc
[+] Possible entry point: pyi_rth_setuptools.pyc
[+] Possible entry point: pyi_rth_pkgres.pyc
[+] Possible entry point: __main__.pyc
[+] Found 1578 files in PYZ archive
[+] Successfully extracted pyinstaller archive: yt-dlp

You can now use a python decompiler on the pyc files within the extracted directory
```

Figure 3: Unzipped PYZ

The .pyc files contain Python bytecode, and can be further analyzed after disassembly and decompilation, using the tool `Decompyle++`^[?], which can provide both human-readable python bytecode and an attempt at reconstructing the original python code.

```

29 [Constants]
30 0
31 None
32 'frozen'
33 False
34 'main'
35 [Disassembly]
36 0 LOAD_CONST 0: 0
37 2 LOAD_CONST 1: None
38 4 IMPORT_NAME 0: sys
39 6 STORE_NAME 0: sys
40 8 LOAD_NAME 1: __package__
41 10 LOAD_CONST 1: None
42 12 IS_OP 0 (i:)
43 14 POP_JUMP_IF_FALSE 43 (to 86)
44 16 LOAD_NAME 2: getattr
45 18 LOAD_NAME 0: sys
46 20 LOAD_CONST 2: 'frozen'
47 22 LOAD_CONST 3: False
48 24 CALL_FUNCTION 3
49 26 POP_JUMP_IF_TRUE 43 (to 86)
50 28 LOAD_CONST 0: 0
51 30 LOAD_CONST 1: None
52 32 IMPORT_NAME 3: os.path
53 34 STORE_NAME 4: os
54 36 LOAD_NAME 4: os
55 38 LOAD_ATTR 5: path
56 40 LOAD_METHOD 6: realpath
57 42 LOAD_NAME 4: os
1 # Source Generated with Decompyle++
2 # File: __main__.pyc (Python 3.10)
3
4 import sys
5 if not __package__ is None and getattr(sys, 'frozen', False):
6     import os.path as os
7     path = os.path.realpath(os.path.abspath(__file__))
8     sys.path.insert(0, os.path.dirname(os.path.dirname(path)))
9 import yt.dlp
10 if __name__ == '__main__':
11     yt.dlp.main()
12     return None
13

```

Figure 4: Decompyle++ applied to main

However, `Decompyle++` isn't always successful in decompiling the code, and probing the disassembled code in search of the malware code (That at this point in time had its behaviour known, due to Dynamic Analysis that will be covered in the next section.) did not yield results, mainly due to the sheer size and scope of the `yt-dlp` tool, whose vast amount of libraries and functionalities made searching for anything concrete quite difficult.

With static analysis reaching a dead-end, dynamic analysis was then employed.

2.3 Dynamic Analysis

2.3.1 Network Analysis/Packet Capture with Wireshark

As an initial analysis, the yt-dlp was executed inside a vmware virtual machine and its traffic captured.

In this packet capture, there are two top level domains resolved by DNS: youtube and dropbox.

Querying the Youtube domain was expected behaviour of a video downloading tool, but the same cannot be written for Dropbox. The domain queried for it was:

- ‘uc9f5ff7ecd10e603966873989b3.dl.dropboxusercontent.com’

However, accessing this domain on a browser yields no results. These following googlevideo domains were also queried, but are likely to be for video downloading purposes as well:

- ‘rr3—sn-2vgu0b5auxaxjvh-v2vz.googlevideo.com’
- ‘rr5—sn-apn7en7s.googlevideo.com’

All the other traffic present in the packet capture is HTTPS and therefore encrypted, and useless to analyse here.

2.3.2 Strace

Reverse engineering the malware behaviour was most successful when analyzed with this tool. It is used to analyze and log every syscall done by the program during execution, and can give a wide insight as to what the program is doing, as syscalls are used for I/O operations such as operations concerning the filesystem, sockets (Including TCP/IP communications.) as well as observing fork/clone operations, and following the syscalls done in those. As such, and despite not being able to observe the internal operations of the malware, it was possible to gain a lot of understanding of its behaviour and purpose.

To start off, the executable is ran without any extra arguments. This seemingly does not trigger the malware, (The infection indicators were not present afterwards.) and allows a small map of its normal functionality.

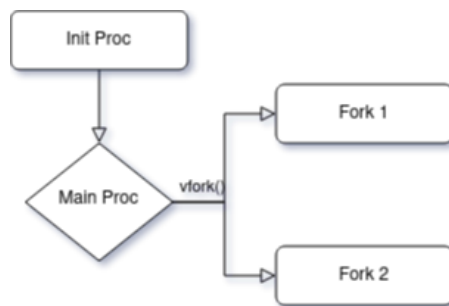


Figure 5: Barebones yt-dlp Execution

Afterwards, the executable was ran with an youtube link as an argument. This did trigger the malware, and below is presented a full analysis of its behaviour and functionality visible from Strace. The numbers on each box, which corresponds to a process/thread, are its respective PID, displayed and referred to throughout the analysis for ease of understanding. These will **not** remain constant throughout executions, but do show the order in which the processes were created, reflected in the diagram as well. (Time not to scale.)

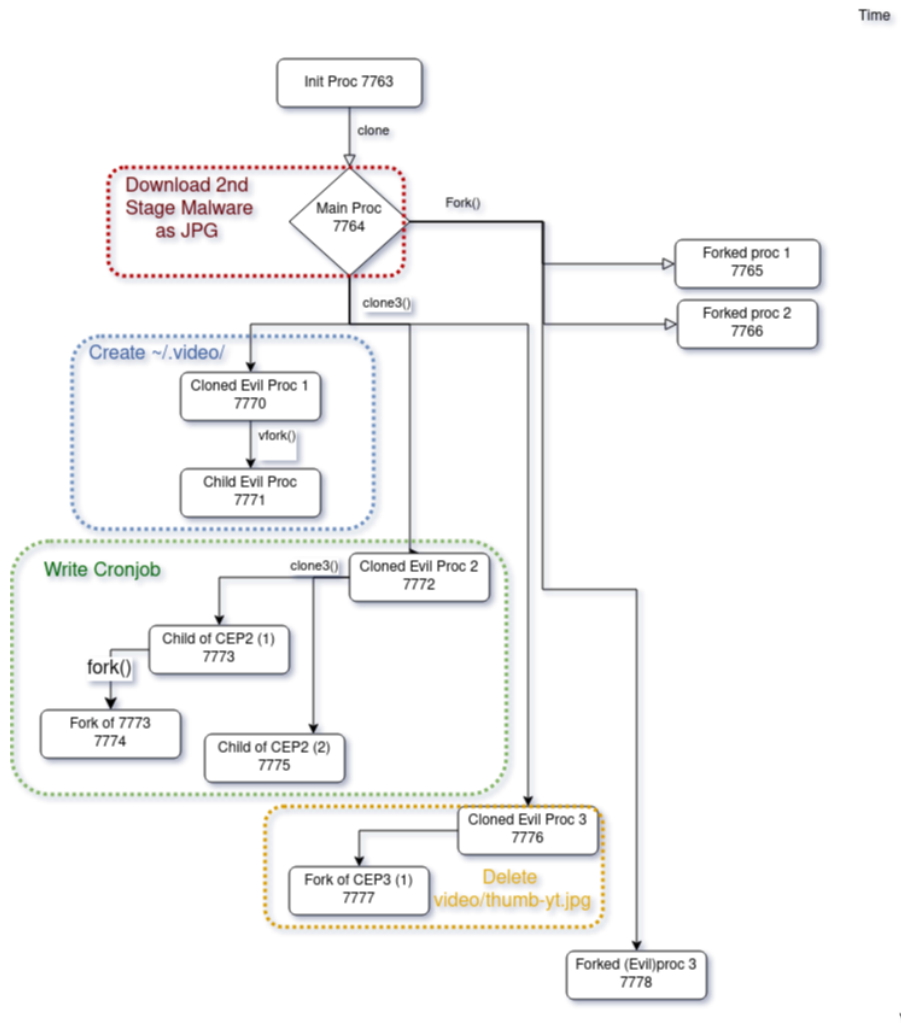


Figure 6: Malware Dropper Execution Flow

The executable was analyzed both with and without elevated privileges. The functionality differs at times, and these will be noted throughout.

The Main Process (PID 7764) runs just the same as in the previous run, up until the point where it clones itself into 7770 (detailed below):

Cloned Evil Proc 1 (PID 7770 and 7771)

This thread (and its fork, 7771) serve to create the `~/ .video` directory, which is part of the malware flow and not default *yt-dlp* behaviour:

```
mkdir -p /home/ubuntu/.video/
```

Which, as can be seen in 7771's trace, is attempted to be done with full permissions, 0777.

```
mkdir("/home", 0777)          = -1 EEXIST (File exists)
chdir("/home")                = 0
mkdir("ubuntu", 0777)         = -1 EEXIST (File exists)
chdir("ubuntu")               = 0
mkdir(".video/", 0777)        = 0
```

This process then exits, which brings the execution back to **Main Proc (7764)**.

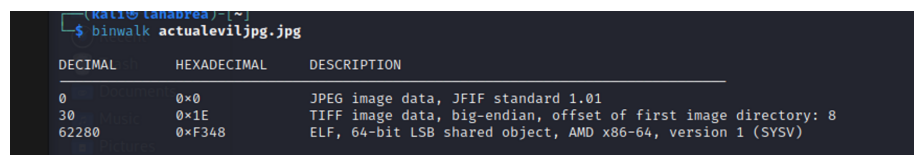
Main Proc 1 (PID 7764)

This process proceeds to do four DNS requests, such as this one, retrieved from the strace socket transmissions and decoded using python:

```
;QUESTION
www.dropbox.com. IN A
;ANSWER
www.dropbox.com. 39 IN CNAME www-env.dropbox-dns.com.
www-env.dropbox-dns.com. 39 IN A 162.125.68.18
```

A bit below it then writes/creates a file, most likely having been downloaded from dropbox (which cant be verified as the communications with dropbox use HTTPS and as such are encrypted).

This file, seemingly a JPG, is written to `/home/ubuntu/.video/thumb-yt.jpg`. However, in closer analysis:



DECIMAL	HEXADECEMAL	DESCRIPTION
0	0x0	JPEG image data, JFIF standard 1.01
30	0x1E	TIFF image data, big-endian, offset of first image directory: 8
62280	0xF348	ELF, 64-bit LSB shared object, AMD x86-64, version 1 (SYSV)

Figure 7: A polyglot JPG containing an ELF

This JPG is in reality a polyglot, and contains embedded an ELF file. The malware extracts this ELF from the JPG, stores it as `~/video/check_version`.

This is when it can be made sure this is a malware being examined. It specifically is a dropper, a first-stage in the attack that will retrieve the payload that contains the actual damaging code.

It then clones itself:

Cloned Evil Proc 2(PID=7772,7773 and 7775)

This process aims to execute a very suspicious command:

```
execve("/bin/sh", ["sh", "-c", "(crontab_l2>/dev/null;  
echo \"*/5 * * * * $HOME/.video/check_version\") \"  
____|_crontab_"], [(. . .)]) = 0
```

This creates a cronjob that runs every 5 minutes and executes the (malicious) ELF previously downloaded.

It does so in the following manner: Process 7772 forks into 7773, which attempts to get information about several possible cron binary paths, in order to find which one is valid/existing on the system (Which in this case is `/usr/bin/crontab`.)

```
newfstatat(ATFDCWD, "/usr/local/sbin/crontab",  
           0x7ffd624cd310, 0)  
= -1 ENOENT (No such file or directory)  
newfstatat(ATFDCWD, "/usr/local/bin/crontab",  
           0x7ffd624cd310, 0)  
= -1 ENOENT (No such file or directory)  
newfstatat(ATFDCWD, "/usr/sbin/crontab",  
           0x7ffd624cd310, 0)  
= -1 ENOENT (No such file or directory)  
newfstatat(ATFDCWD, "/usr/bin/crontab", {(. . .)}, 0) = 0
```

7773 then forks into 7774, and executes the binary found by its parent, `/usr/bin/crontab`.

It also reads `/etc/passwd`, then changes directory to `/var/spool/cron` (which is where the malicious cronjob is created), attempts to read `/etc/cron.allow` and `/etc/cron.deny` and successfully reads `/etc/localtime`.

It then attempts to open multiple possible locations of 'libc.mo' but none of them exist (It's unclear as to what its purpose is.), then tries to use 'fopen' to write to 'crontabs' but is denied permission and exits with code 1.

Back in 7773, after this fork returns, 7773 attempts to write a cronjob, but fails as the pipe has been broken (Never existed, 7773 failed to get a file descriptor for the cronjob-containing-file.)

HOWEVER, if the executable is being ran with elevated privileges, the cronjob is successfully written.

In process 7775, it will attempt to write to `/var/spool/cron` in a similar manner to 7774, but via tmp files (specifically, `mkstemp()`) but fails as well, if it isn't running with elevated privileges, in which case it succeeds at opening `crontabs/tmp.x` (x is a random string of 6 characters), and writes the following into it:

```
# DO NOT EDIT THIS FILE — edit the master and reinstall.
# (— installed on Fri Apr 28 18:15:46 2023)
# (Cron version — $Id: ...)
*/5 * * * * /root/.video/check_version
```

It then renames the file into `crontabs/root`.

Process 7770 will wait for 7775 to end, and then 7773, at which point it too ends, going back to 7764, which clones itself again.

Cloned Evil Proc 3(PID=7776, 7777)

This process, 7776, aims to delete the previously created `~/.video/thumb-yt.jpg`, and does retrieve a number of data values relating to itself:

- uid - gid - pid - euid - ppid - egid - current directory (pwd)

and then forks itself into 7777, which seems to do the actual deleting part of the previous process.

After 7776 ends, going back to 7764, which it forks itself once again, into 7778, which seems to be part of the normal function of `yt-dlp`. From this point on, normal execution resumes.

2.4 Conclusion

To sum up, the binary **yt-dlp** acts as a malware dropper. It fetches an executable disguised as a JPG, extracts it, then creates a cronjob that will execute it every five minutes. It has multiple mechanisms that will attempt the creation of the cronjob using different strategies.

Signs it executed are the presence of `~/.video/`, with the **check-version** file inside, which is the actual malware downloaded, and the existence of the cronjob.

3 Malware Analysis

The malicious program is located at the path `/root/.video/check.version`. Upon dissection, the code appears to be a malicious program designed to infiltrate a target system and exfiltrate sensitive data.

3.1 Main Function

The program is an ELF file and begins by initializing some variables and taking in command-line arguments through the main function. The program initializes some variables, one of these being a sha256 digest of the hostname of the target.

```
int32_t hash_len = 0x20
char hostname_end = 0
void hostname
gethostname(name: &hostname, len: 0x3ff)
void host_hash
sha256(&hostname, &host_hash, &hash_len)
```

After variable initialization it creates two threads using `pthread_create`, and sets up message queues using `msgget`.

```
union pthread_attr_t thread_attr0
pthread_attr_init(attr: &thread_attr0)
union pthread_attr_t thread_attr1
pthread_attr_init(attr: &thread_attr1)
pthread_t thread_ID0
pthread_create(&thread_ID0, &thread_attr0,
               exfiltrate_files, &msg_chan_key0)
pthread_t thread_ID1
pthread_create(&thread_ID1, &thread_attr1,
               encrypt_files, &msg_chan_key1)
pthread_attr_destroy(attr: &thread_attr0)
pthread_attr_destroy(attr: &thread_attr1)
```

These threads are responsible for exfiltrating files from the target system and encrypting them, respectively. The program, after thread creation, calls the `process_directory` function with the path to the user's HOME directory, the `host_hash` key data, and the two message queue keys.

This function is responsible to scan the user's directories for files to exfiltrate and encrypt. The code then sends exit messages to the message queues and joins the two threads using `pthread_join`. Finally, the program checks if the `fsbase` pointer has been modified by the malware and triggers a stack-smashing protection error and terminates the program if necessary.

3.2 Process Directory Function

The code takes a path to a directory as input and performs several operations on the directory.

It recursively traverses the directory and encrypts certain files, then writes a README.txt file in the directory containing instructions for the victim to pay a ransom in order to recover their files.

The function process_directory() takes in four arguments: the path to the directory to process, two message queue IDs (exfil_queue and encrypt_queue) and a key (key_data).

The function begins by getting the address of the thread-local storage base pointer and saving its value to rax.

The function is also responsible for generating the IV to be used on the encryption process. The IV is generated based on operations made to the timestamp from when the function was run.

```
time_t current_time = time(NULL);
srand(x: current_time.d)
```

The generation of the IV can be seen as the following function:

```
void generate_iv(char *iv_string) {
    unsigned char iv[16];

    time_t current_time = time(NULL);
    srand((unsigned)current_time);

    for (int i = 0; i < 16; i++) {
        iv[i] = rand() & 0xff;
    }

    for (int i = 0; i < 16; i++) {
        sprintf(iv_string + i * 2, "%02x", iv[i]);
    }
}
```

After IV generation the program opens the directory at the given path using opendir(). If the directory is successfully opened, the function generates some random data, fills a block of memory with zeroes, and then writes a file named "README.txt" in the directory. The contents of the file are encrypted using a XOR cipher, with the key being the value 0x13 (1337).

The function then closes the "README.txt" file and proceeds to process each file in the directory.

The code reads each file in the directory using readdir(), then checks the file's extension to see if it matches any of several pre-defined extensions (".priv", ".pub", ".p12", ".pem", ".crt", "id_rsa", "id_dsa", "id_ecdsa").

```
if (file_extension != 0)
    if (strcasecmp(file_extension, ".priv") == 0)
        label_409f:
        msgsnd(msqid: exfil_queue,
              msgp: temp_var2_ptr,
```

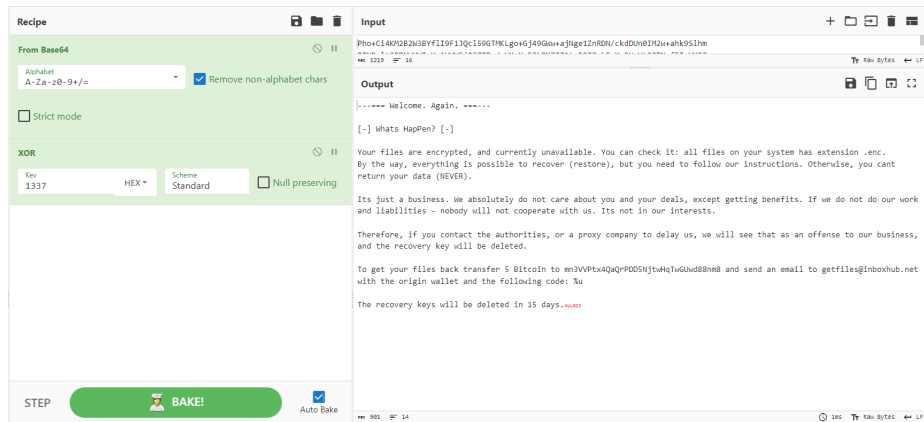


Figure 8: README text

```

        msgsz: zx.q(strlen(temp_var4) + 0x31),
        msgflg: 0)
else
    if (strcasemp(file_extension, ".pub") == 0)
        goto label_409f
    if (strcasemp(file_extension, ".p12") == 0)
        goto label_409f
    if (strcasemp(file_extension, ".pem") == 0)
        goto label_409f
    if (strcasemp(file_extension, ".crt") == 0)
        goto label_409f
    if (strcasemp(file_extension, "id_rsa") == 0)
        goto label_409f
    if (strcasemp(file_extension, "id_dsa") == 0)
        goto label_409f
    if (strcasemp(file_extension, "id_ecdsa") == 0)
        goto label_409f

```

If the file has a matching extension, the function sends the file data to `exfil_queue` using `msgsnd()`. If the file matches instead the not critical extensions present within the memory of the program, the function sends the file data to `encrypt_queue`.

```

int32_t compare_result
do
    compare_result = strcasemp(file_path,
                               *extension_ptr)

    if (compare_result == 0)
        break
    extension_ptr = extension_ptr + 8

```

```

while (&temp_var9 != extension_ptr)
if (compare_result != 0)
    break
msgsnd(msqid: encrypt_queue,
       msgp: temp_var2_ptr,
       msgsz: zx.q(strlen(temp_var4) + 0x31),
       msgflg: 0)

```

If the file is a directory, the code checks to see if it is either the current directory (".") or the parent directory (".."), and skips processing it if it is. Otherwise, the function recursively calls itself with the path of the subdirectory.

If the end of the directory is reached, the function returns result, which indicates success or failure of the operation.

3.3 Exfiltrate Files Function

The function first initializes some variables, including a variable to store the type of received message, and gets the current value of a register using the "fsbase" keyword.

It then enters a loop where it waits for messages to be received on the message queue identified by the input argument. When a message is received, the function calls another function called "exfiltrate_file_contents" with the name of the input file.

3.4 Exfiltrate File Contents Function

The function exfiltrates the file contents provided through arguments. The exfiltration process involves several steps. Firstly, the input for the filename given is opened. Secondly, the filename is base64 encoded and any '/' characters are replaced with '-'. Next, a subdomain is constructed with the format "0-base64_encoded_filename.max.xyz" and a DNS query is sent to it. The file is then read in 32-byte chunks and for each chunk, the chunk is base64 encoded and any '/' characters are replaced with '-'. A subdomain is constructed with the format "N-base64_encoded_chunk.max.xyz" (where N is the chunk number) and a DNS query is sent to it. This process is repeated until the entire file has been read.

3.5 Encrypt Files Function

The function retrieves a message queue ID from a pointer passed to it as a parameter. If a global variable data_8778 is not equal to 0, the function enters a loop where it waits for a message to arrive on the message queue with type 1. Once it receives a message, it proceeds to encrypt the file named in the message.

The encryption process involves allocating memory for a new encrypted filename, encrypting the original file with a symmetric key algorithm, removing the original file, and freeing the memory allocated for the new encrypted filename.

The encrypted filename is constructed by appending a fixed suffix of ".enc" to the original filename, and the encryption key and initialization vector are retrieved from global variables named key and iv. To encrypt the file contents the function calls the "encrypt_or_decrypt_file" function passing the input file name, the encrypted file name, a key and an initialization vector (IV). The mode parameter is set to 1 to indicate encryption.

After retrieving the original filename, the code checks its length to determine how many bytes are needed to store the encrypted filename. If the length is greater than or equal to 8 bytes, the code copies the contents of the original filename into the encrypted filename array and pads the remaining bytes with random data. This is done using a loop that copies 8 bytes at a time from the original filename to the encrypted filename array. If there are any remaining bytes in the original filename, they are padded with random data.

If the length of the original filename is less than 8 bytes, the code checks if the length is greater than or equal to 4 bytes. If it is, the code copies the contents of the original filename into the first 4 bytes of the encrypted filename array and pads the remaining bytes with random data. If the length is less than 4 bytes, the code copies the contents of the original filename into the first byte of the encrypted filename array and pads the remaining bytes with random data.

After the encrypted filename is generated, a null byte is added to terminate the string, and a specific suffix is added to the filename. The suffix is ".enc" followed by a null byte. This suffix is used to identify files that have been encrypted by the malware.

Once the encrypted filename is generated, the file contents are encrypted using a function called "encrypt_or_decrypt_file". This function takes as input the original filename, the encrypted filename, a key, an initialization vector (IV), and a mode. The mode parameter is set to 1, indicating that the function should encrypt the file contents.

After the file contents are encrypted, the original file is deleted using the "remove" function. Finally, the memory used to store the encrypted filename is freed using the "free" function. The function continues to loop until a flag variable named "data_8778" is set to a non-zero value.

3.6 Encrypt Or Decrypt File Function

The function either encrypts or decrypts the contents of a file, based on the specified mode, and writes the result to a new file. The function takes in four arguments: the input filename, the output filename, a key, an initialization vector (IV), and a mode (0 for decryption, 1 for encryption).

First, the function opens the input file in read mode and the output file in write mode. If either file fails to open, the function returns an error code of 0xffffffff.

Next, the function creates a new cipher context using the EVP library, which is used to perform the encryption or decryption operation. If the context fails to be created, the function returns an error code of 0xffffffff.

The function then initializes the cipher context based on the specified mode and reads the contents of the input file in blocks of 1024 bytes. The input data is passed to the EVP function along with the cipher context, and the resulting output is written to the output file. This process is repeated until the entire input file has been processed.

Finally, the function closes both the input and output files, frees the cipher context, and returns either 0 or 0xffffffff depending on whether the operation was successful or not.

Overall, this function appears to be a generic file encryption/decryption routine that is used by other parts of the malware to encrypt or decrypt sensitive files.

3.7 Files recovery

In order to recover the original files without paying to the attackers, we first need to generate the necessary key for decryption, this key is a digest of the hostname of the target computer.

```
$ echo -n "devbox" | openssl sha256
SHA2-256(stdin)=
2f...ce
```

We then generate the initialization vector (IV) that was used to encrypt the data. This IV is based on the time the program was run, this can be seen as the code given in the README, this code will be used as the seed for the `srandom()` function and later used to generate random numbers to be operated on to generate the IV.

```
$ ./gen_iv
IV: 91...5b
```

After getting the IV, what remains is decrypting the encrypted files and fixing their internal structure to their appropriate file type.

```
$ openssl enc -aes-256-cbc -d -in image.jpg.enc
-out image.dat -K 2f...ce -iv 91...5b
$ echo -en
'\xff\xd8\xff\xe1\x00\x0c\x45\x78\x69\x66\x00\x00'
| cat - image.dat > image.jpg
```



Figure 9: recovered image

3.8 Conclusion

The analyzed code shows a malware designed to encrypt the files on the victim's system using the AES-256 encryption algorithm in CBC mode with a random IV. The malware encrypts the filenames and adds a specific suffix to them ".enc". The malware also retrieves critical files from the target through exfiltration via DNS queries to the ".max.xyz" domain. This domain can be considered as an indicator of compromise (IOC) that can be used to identify the systems that have been infected by this malware.

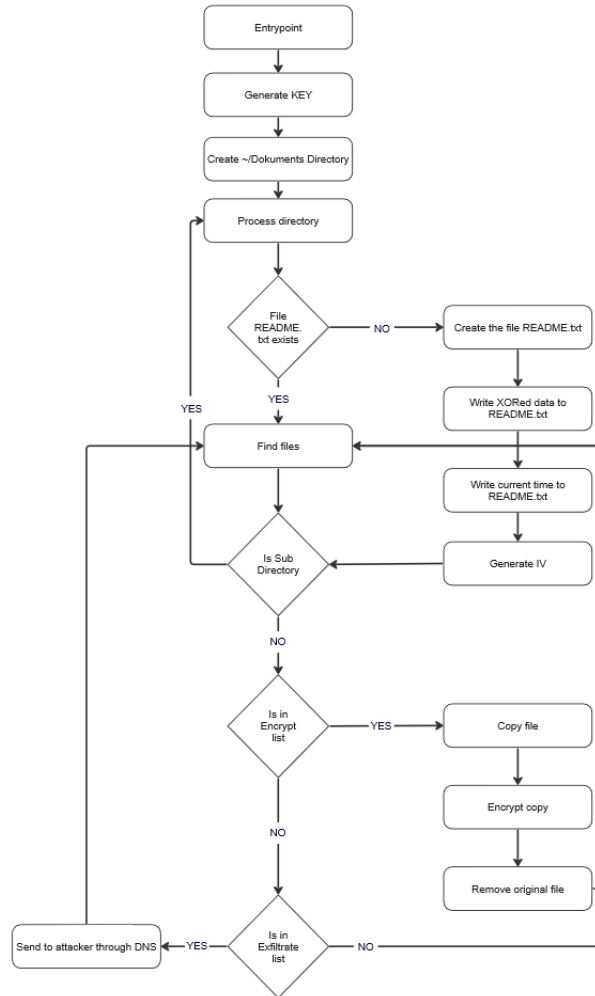


Figure 10: Malware Flow

The malware employs an encryption key derived from the SHA256 hash of the victim's system hostname, which is a distinct value for each affected system.

In conjunction with an initialization vector (IV) created based on the timestamp when the malware is executed, this key and IV render file recovery impossible without possessing both. However, due to their inherent characteristics, the system owner can potentially obtain the required information to regenerate the key and IV.

4 Conclusions and Recommendations

The provided executable was proven to be a malware, made up of a dropper and a ransomware payload as a second-stage. It retrieves a payload masquerading as a JPG in the form of a polyglot file and executes it using a created cronjob.

The ransomware payload was proven to have two main objectives - exfiltration of critical files used for authentication through DNS queries and file encryption of the target's documents. The decryption of the files can be made with the hostname and the Unix timestamp from when the program was run, enabling file recovery without paying the ransom.

There was no evidence found of the malware having capabilities to spread to other hosts. However it is recommended to carefully analyze the contents present on the target machine to minimize the use of the exfiltrated authentication files present within it onto other hosts.

A system reinstall would be preferred to make sure all traces of the malware are gone, but if such isn't possible, it's needed to delete the `~/.video` (or `/root/.video`, if ran with `sudo`) and the created cronjobs. It is also advisable to remove the `Dokuments` folder and the files within, but these can be kept if so is desired, as they're harmless.