Writing a decryptor for Jaff ransomware

clairelevin.github.io/malware/2023/02/14/jaff.html

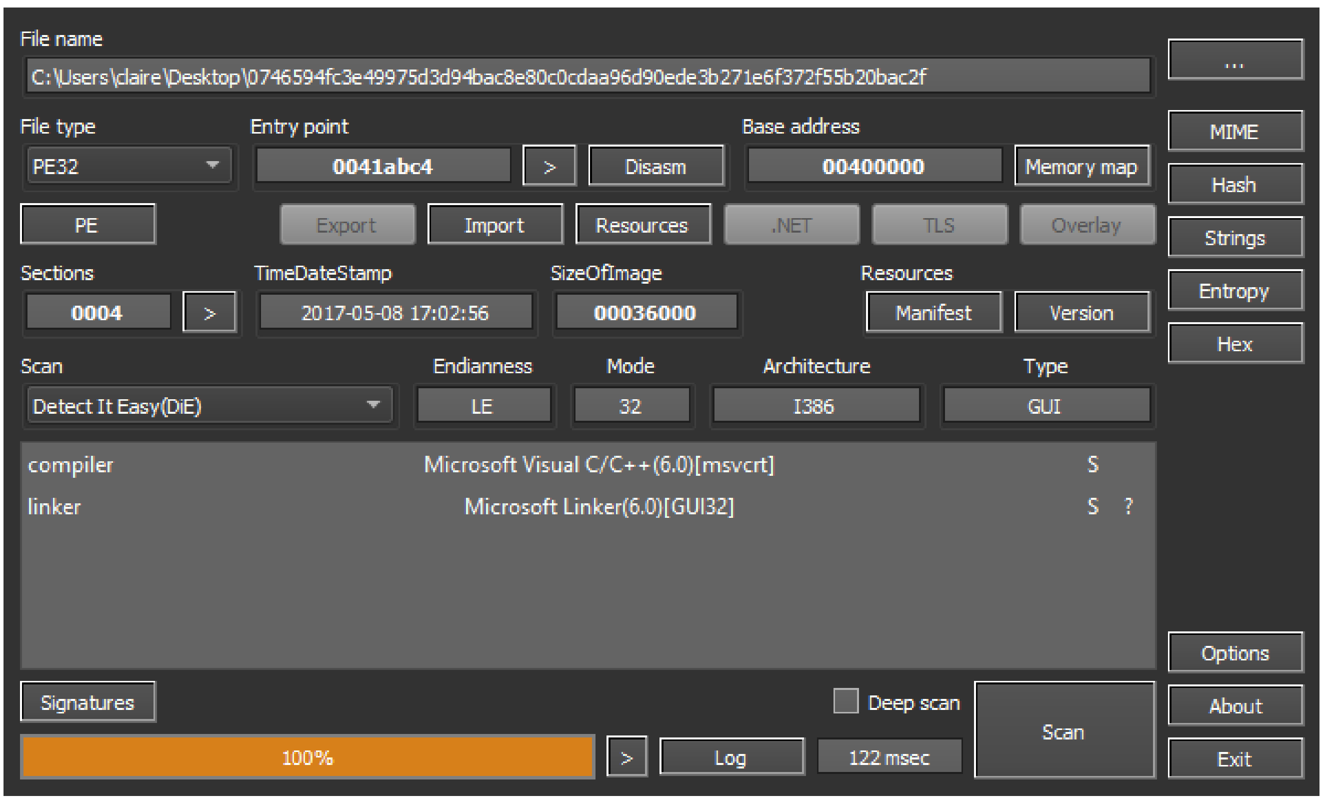
February 14, 2023

Overview

Recently, I've been trying to learn more about reverse engineering ransomware. Jaff is ransomware from a campaign dating back to 2017, and I was told that it had a vulnerability that would make it possible to write a decryptor. I analyzed a sample to see if I could rediscover the vulnerability myself.

You can find the sample I used on MalShare, and its SHA256 hash is `0746594fc3e49975d3d94bac8e80c0cdaa96d90ede3b271e6f372f55b20bac2f`.

Initial Observations

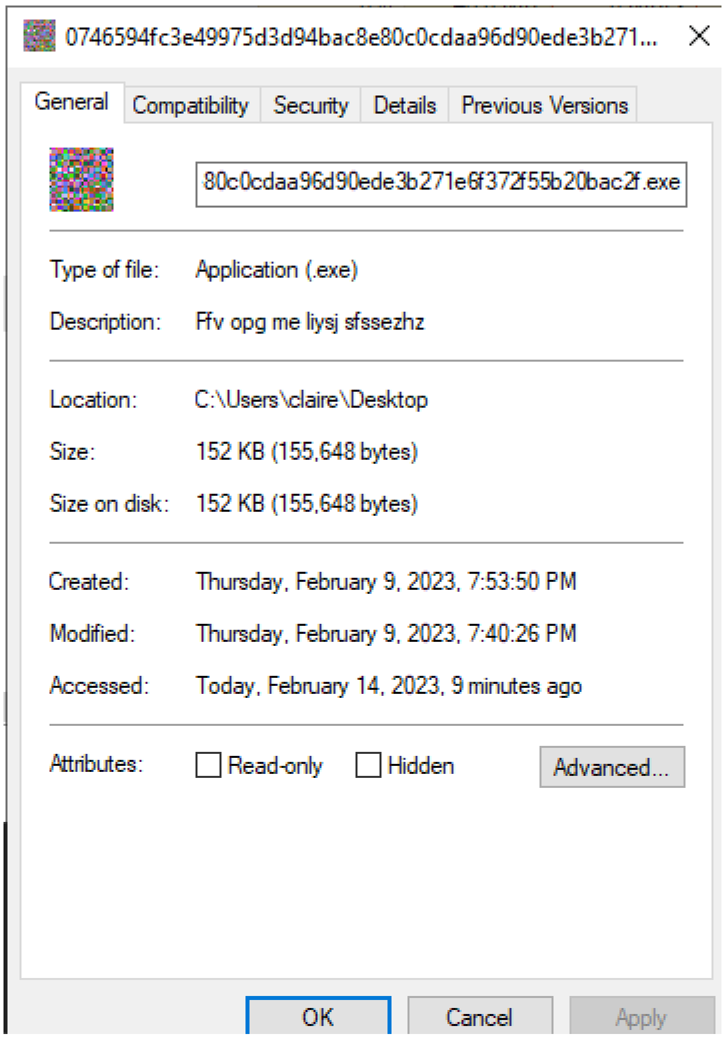


The sample is a 32-bit PE excutable written in C++. The executable did not seem to import any functions related to cryptography, and it contained a very long chunk of encrypted data. This meant that the most important functions of this program were likely being decrypted dynamically.

By setting breakpoints on `VirtualAlloc` and `VirtualProtect`, I kept track of each time a RWX segment of memory was allocated. After several calls to `VirtualAlloc` and `VirtualProtect`, the program wrote a PE file to one of these segments, which I dumped from memory. This turned out to be the actual encryptor, and it's what I'll be focusing on for the remainder of my analysis.

Behaviors

When run, the sample calls itself `Ffv opg me liysj sfssezhz`:



Additionally, a GET request is made to `fkksjobnn43[.]org/a5/`. As I don't have access to this C2 server, I have no way of knowing what was expected from this server or whether the encryption process would have proceeded differently if I'd been able to connect.

```
GET /a5/ HTTP/1.1
Host: fkksjobnn43.org
```

Strings, Imports, and Resources

The binary I dumped from memory imports cryptography-related functions such as `CryptEncrypt`, `CryptExportKey`, and `CryptGenKey`, as well as file enumeration functions such as `FindFirstFileW` and `FindNextFileW`. This is how I knew I was looking at the actual encryptor.

Additionally, there were several resources containing data used in the encryption process:

- `#105`: The string representation of the numbers `35326054031861368139563306184134167018130718569482731666001650817539108744401016633231304437224730790638615766740272106403143256` and `35326054031861368139563306184134167018130718569482731666001650829864568371094444203557601170206844003631101722202233367975968667`.

- `#106`: The file extensions to encrypt:

```
.xlsx .acd .pdf .pfx .crt .der .cad .dwg .MPEG .rar .veg .zip .txt .jpg .doc
.wbk .mdb .vcf .docx .ics .vsc .mdf .dsr .mdi .msg .xls .ppt .pps .obd .mpd .dot
.xlt .pot .obt .htm .html .mix .pub .vsd .png .ico .rtf .odt .3dm .3ds .dxf .max
.obj .7z .cbr .deb .gz .rpm .sitx .tar .tar.gz .zipx .aif .iff .m3u .m4a .mid
.key .vib .stl .psd .ova .xmod .wda .prn .zpf .swm .xml .xlsm .par .tib .waw
.001 .002 003. .004 .005 .006 .007 .008 .009 .010 .contact .dbx .jnt .mapimail
.oab .ods .ppsm .pptm .prf .pst .wab .1cd .3g2 .7ZIP .accdb .aoi .asf .asp. aspx
.asx .avi .bak .cer .cfg .class .config .css .csv .db .dds .fif .flv .idx .js
.kwm .laccdb .idf .lit .mbx .md .mlb .mov .mp3 .mp4 .mpg .pages .php .pwm .rm
.safe .sav .save .sql .srt .swf .thm .vob .wav .wma .wmv .xlsb .aac .ai .arw .c
.cdr .cls .cpi .cpp .cs .db3 .docm .dotm .dotx .drw .dxb .eps .fla .flac .fxg
.java .m .m4v .pcd .pct .pl .potm .potx .ppam .ppsx .ps .pspimage .r3d .rw2
.sldm .sldx .svg .tga .wps .xla .xlam .xlm .xltm .xltx .xlw .act .adp .al .bkp
.blend .cdf .cdx .cgm .cr2 .dac .dbf .dcr .ddd .design .dtd .fdb .fff .fpx .h
.iif .indd .jpeg .mos .nd .nsd .nsf .nsg .nsh .odc .odp .oil .pas .pat .pef .ptx
.qbb .qbm .sas7bdat .say .st4 .st6 .stc .sxc .sxw .tlg .wad .xlk .aiff .bin .bmp
.cmt .dat .dit .edb .flvv .gif .groups .hdd .hpp .log .m2ts .m4p .mkv .ndf
.nvram .ogg .ost .pab .pdb .pif .qed .qcow .qcow2 .rvt .st7 .stm .vbox .vdi .vhd
.vhdx .vmdk .vmsd .vmx .vmxf .3fr .3pr .ab4 .accde .accdt .ach .acr .adb .srw
.st5 .st8 .std .sti .stw .stx .sxd .sxg .sxi .sxm .tex .wallet .wb2 .wpd .x11
.x3f .xis .ycbcra .qbw .qbx .qby .raf .rat .raw .rdb rwl .rwz .s3db .sd0 .sda
.sdf .sqlite .sqlite3 .sqlitedb .sr .srf .oth .otp .ots .ott .p12 .p7b .p7c .pdd
.pem .plus_muhd .plc .pptx .psafe3 .py .qba .qbr.myd .ndd .nef .nk .nop .nrw
```

- `#109`: The ransom note in HTML form, with the string `[ID5]` in place of the victim's decryption ID.

- `#110`: The string `.jaff`, which is the extension appended to encrypted files.

- `#111`: The URL `fkksjobnn43[.]org/a5/`.

- `#112`: The ransom note in text form, again with `[ID5]` in place of the ID.

- `#113`: A string of bytes which, when XORed with the second number in `#106`, gives the strings `ReadMe.txt`, `ReadMe.bmp`, and `ReadMe.html`.

Additionally, the string `cmd /C del /Q /F %s` found in the program suggests that it is intended to delete itself once encryption is complete.

The Encryption Process

The sample uses 256-bit AES to encrypt files. For debugging purposes, I set a breakpoint on `CryptImportKey` to read the key blob from memory:

```
Address  | Hex                                             | ASCII
013CB940 | 08 02 00 00 10 66 00 00 20 00 00 00 DE 48 17 E3 | .....f.. ...þH.ã
013CB950 | E4 0C 28 DE 1B 28 34 53 FF 8F 9D FE E7 28 68 E0 | ä.(þ.(4Sÿ..þç(hà
013CB960 | A6 A2 AF 59 81 8F 14 11 24 43 07 AA AB AB AB AB | ¦¢¯Y....$C.ª««««
013CB970 | AB AB AB AB 00 00 00 00 00 00 00 00 00 00 00 00 | ««««............
```

A new key is generated using `CryptGenKey` each time the program is run.

Beginning with the root directory, the program enumerates all files and subdirectories and uses `CryptEncrypt` to AES encrypt each file. The program uses `GetLogicalDrives` to find all drives connected to the system, and encrypts all drives that are not CD-ROM drives (possibly because a CD-ROM drive would make a noticeable noise as it started up).

The `.jaff` extension is appended to the encrypted file, and the AES-encrypted bytes are written. We can see that there are multiple `WriteFile` calls to the encrypted file, revealing that something else is appended to the `.jaff` file before the encrypted data:

```
ReadFile(hFile: file, lpBuffer: eax_10, nNumberOfBytesToRead: edi_3, lpNumberOfBytesRead: var_230, lpOverlapped: nullptr)
var_230 = nullptr
SetFilePointer(hFile: file, lDistanceToMove: 0, lpDistanceToMoveHigh: nullptr, dwMoveMethod: FILE_BEGIN)
var_230 = &bytes_written
WriteFile(hFile: file, lpBuffer: &header, nNumberOfBytesToWrite: 4, lpNumberOfBytesWritten: var_230, lpOverlapped: nullptr)
var_230 = &bytes_written
WriteFile(hFile: file, lpBuffer: number_str, nNumberOfBytesToWrite: header, lpNumberOfBytesWritten: var_230, lpOverlapped: nullptr)
var_230 = &bytes_written
WriteFile(hFile: file, lpBuffer: &size, nNumberOfBytesToWrite: 4, lpNumberOfBytesWritten: var_230, lpOverlapped: nullptr)
var_230 = &bytes_written
WriteFile(hFile: file, lpBuffer: ciphertext, nNumberOfBytesToWrite: size, lpNumberOfBytesWritten: var_230, lpOverlapped: nullptr)
```

The appended value turned out to be the ASCII representation of a large number.

```
Input     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  02 01 00 00 33 33 35 34 33 37 31 34 31 30 31 32
00000010  32 39 31 38 32 33 35 30 32 33 30 30 32 37 35 35
00000020  30 38 33 32 32 30 34 39 37 30 32 33 38 37 35 32
00000030  33 38 33 30 37 36 31 30 37 34 37 31 32 38 39 30
00000040  38 39 32 35 30 33 35 33 37 35 38 39 38 30 36 38
00000050  36 39 37 33 31 38 30 33 31 37 36 30 30 32 32 35
00000060  33 32 38 30 35 30 35 39 39 35 32 33 30 31 38 39
00000070  36 32 36 31 34 34 30 33 35 36 32 37 38 32 32 31
00000080  39 37 34 35 20 31 37 35 32 38 32 39 34 32 37 36
00000090  34 34 36 34 31 35 31 32 33 34 30 38 39 37 31 33
000000A0  36 33 38 30 38 37 37 35 31 36 35 32 35 36 36 33
000000B0  33 36 33 30 37 36 38 33 37 30 34 39 38 37 38 35
000000C0  35 30 36 37 37 33 34 31 31 30 30 36 36 38 36 36
000000D0  33 33 37 31 30 34 37 34 35 39 39 36 36 37 35 37
000000E0  38 36 34 39 39 39 34 39 37 31 38 37 34 30 33 33
000000F0  39 35 36 30 31 33 36 32 33 33 30 36 33 37 39 36
00000100  37 35 33 36 33 20 20 27 00 00 C6 00 14 CA 40 B5
00000110  9D 16 AA A6 DC F2 8B 1F 67 BA EB BF 90 C9 C4 2A
00000120  58 34 7B C7 50 44 42 5B C5 C6 FB 53 03 63 4E 9B
00000130  71 CA 3D 19 04 E7 4D 23 06 28 0D C4 33 EA 73 48
00000140  65 ED 70 20 77 9C 4A A6 34 C9 B0 60 73 69 79 D1
00000150  A1 D6 A0 C1 B2 2A FD 2C CF 24 C7 C6 56 E6 B8 2E
00000160  FD C4 AE 6B 2A 12 38 A9 CE D6 BE AA 22 36 1B D9
00000170  95 0F 73 94 D9 45 8F 70 BF E4 61 1A 29 74 82 B8
00000180  BF A3 63 06 3D F7 48 1C 50 DA A3 27 46 D4 86 10
00000190  FA B0 BA 39 3F EA 04 72 D7 59 EA 18 41 8A F6 50
```

testfile.txt.jaff

Additionally, the ransom note is dropped in each encrypted directory. The note is dropped in text, HTML, and image forms, with file names of `ReadMe.txt`, `ReadMe.html`, and `ReadMe.bmp` respectively.

jaff decryptor system

Files are encrypted!

To decrypt flies you need to obtain the private key.
The only copy of the private key, which will allow you to decrypt your files, is located on a secret server in the Internet

❶ You must install Tor Browser: https://www.torproject.org/download/download-easy.html.en

❷ After instalation, run the Tor Browser and enter address: http://rktazuzi7hbln7sy.onion/

Follow the instruction on the web-site.

Your decrypt ID: 0709158138

A new victim ID is generated each time the program is run.

Encryption of the AES Key

I suspected that the long number appended before the encrypted data in the `.jaff` files was likely an encryption of the AES key. A new AES key was generated for each victim, so the program would need some way to store it.

Representing The Key Bytes

I found that the AES key was being passed as an argument to `sub_402d70`. When passed into this function, the AES key blob was being stored as a decimal representation in little-endian format, with each decimal digit being stored as a 16-bit integer. Each byte of the key blob was converted to three decimal digits; for instance, `08` would be stored as `008` and `8A` would be stored as `138`. Additionally, the digit "1" was appended to the sequence:

Address	Hex	ASCII
013DA208	06 00 03 00 00 00 07 00 01 00 00 00 00 00 02 00	....
013DA218	00 00 03 00 04 00 01 00 09 00 02 00 01 00 09 00	....
013DA228	08 00 00 00 05 00 07 00 01 00 02 00 06 00 01 00	....
013DA238	06 00 06 00 01 00 04 00 02 00 02 00 04 00 00 00	....
013DA248	01 00 00 00 04 00 00 00 01 00 03 00 02 00 04 00	....
013DA258	05 00 02 00 07 00 05 00 01 00 03 00 04 00 01 00	....
013DA268	05 00 05 00 02 00 03 00 08 00 00 00 02 00 05 00	....
013DA278	00 00 00 00 04 00 00 00 07 00 02 00 00 00 02 00	....
013DA288	02 00 02 00 00 00 04 00 00 00 02 00 01 00 00 00	....
013DA298	08 00 02 00 02 00 07 00 02 00 02 00 03 00 02 00	....
013DA2A8	00 00 02 00 07 00 00 00 02 00 02 00 02 00 00 00	....
013DA2B8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
013DA2C8	02 00 03 00 00 00 00 00 00 00 00 00 00 00 00 00	....
013DA2D8	00 00 02 00 00 00 01 00 06 00 01 00 00 00 00 00	....
013DA2E8	00 00 00 00 00 00 00 00 00 00 02 00 00 00 00 00	....
013DA2F8	08 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00	....

For example, during one run of the program, the original AES key blob was the following:

```
08 02 00 00 10 66 00 00 20 00 00 00 52 8A A4 D0 46 E3 4F FE E8 C6 A0 F5 91 0C 25 81
03 0E 5C 3C 57 F6 A0 43 08 32 C9 83 2C 01 FC 95
```

It was stored as the sequence of bytes

```
04 00 04 00 00 00 01 00 03 00 01 00 01 00 00 00 02 00 00 00 05 00 00 00 08 00 00 00
00 00 07 00 06 00 00 00 00 00 06 00 01 00 06 00 04 00 02 00 07 00 08 00 00 00 00 00
06 00 00 00 02 00 09 00 00 00 04 00 01 00 00 00 03 00 00 00 00 00 09 00 02 00 01 00
07 00 03 00 00 00 02 00 01 00 00 00 05 00 04 00 01 00 05 00 04 00 02 00 00 00 06 00
01 00 08 00 09 00 01 00 02 00 03 00 02 00 04 00 05 00 02 00 09 00 07 00 00 00 07 00
02 00 02 00 00 00 07 00 00 00 08 00 00 00 02 00 04 00 06 00 01 00 08 00 03 00 01 00
02 00 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 00 03 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00 01 00 06 00 01 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 02 00 00 00 00 00 08 00 00 00 00 00 01
```

which corresponds to the number

```
10080020000000161020000000320000000000082138164208070227079254232198160245145012037129
003014092060087246160067008050201131044
```

To convert this representation back into bytes, I used the following function:

```
def convert_from_decimal(s):
        result = b''
        s_fixed = s[1:]
        for i in range(0, len(s_fixed) ,3):
                curr_num = s_fixed[i:i+3]
                result += int(curr_num).to_bytes(1, 'little')
        return result
```

Encrypting The Key

At this point, it was time to look at what `sub_402d70` was actually doing. The arguments to the function were the AES key, an array of bytes that were either 1 or 0, and the decimal representation of the number `35326054031861368139563306184134167018130718569482731666001650829864568371094444203555760117020684400363110172220223336797596866 7`. Note that this is one of the two numbers that appeared in resource `#105`.

By experimenting with this subrouting in a debugger, I found that the program was calling functions that performed multiplication and division on arbitrarily large numbers. Sepecifically, the AES key was being squared over and over, and something different was done with the result based on the values in the array of 1s and 0s.

```
00402f25        do
00402dda            void* var_ac_7
00402dda            if (*(i + ones_and_zeros) == 0)
00402e92                void* var_9c
00402e92                void** eax_17 = multiply(arg1, &var_9c, arg1)
00402e9d                int32_t var_18_1 = 0
00402ea4                int32_t var_14_1 = 0xa
00402eab                char var_10_1 = 1
00402eb8                decimal_key_cpy = HeapAlloc(hHeap: GetProcessHeap(), dwFlags: HEAP_ZERO_MEMORY, dwE
00402ec8                void* var_5c
00402ec8                divide(dividend: eax_17, &var_5c, divisor: &arg_10, remainder: &decimal_key_cpy)
00402ed6                HeapFree(hHeap: GetProcessHeap(), dwFlags: HEAP_NONE, lpMem: var_5c)
00402ee0                if (eax_17[3].b == 0)
00402ee2                    char var_10_2 = 0
00402eec                void* var_7c
00402eec                copy_arr(&decimal_key_cpy, arg1, &var_7c)
00402f00                HeapFree(hHeap: GetProcessHeap(), dwFlags: HEAP_NONE, lpMem: var_7c)
00402f0b                HeapFree(hHeap: GetProcessHeap(), dwFlags: HEAP_NONE, lpMem: decimal_key_cpy)
00402f13                var_ac_7 = var_9c
00402df4            else
00402df4                void* var_6c
00402df4                void* var_4c
00402df4                void** eax_7 = multiply(multiply(arg1, &var_4c, arg1), &var_6c, decimal_key)
00402dff                int32_t var_28_1 = 0
00402e06                int32_t var_24_1 = 0xa
00402e0d                char var_20_1 = 1
00402e1a                void* mod_result = HeapAlloc(hHeap: GetProcessHeap(), dwFlags: HEAP_ZERO_MEMORY, dv
00402e2d                void* var_8c
00402e2d                divide(dividend: eax_7, &var_8c, divisor: &arg_10, remainder: &mod_result)
00402e3e                HeapFree(hHeap: GetProcessHeap(), dwFlags: HEAP_NONE, lpMem: var_8c)
00402e48                if (eax_7[3].b == 0)
00402e4a                    char var_20_2 = 0
00402e54                void* var_3c
00402e54                copy_arr(&mod_result, arg1, &var_3c)
00402e68                HeapFree(hHeap: GetProcessHeap(), dwFlags: HEAP_NONE, lpMem: var_3c)
```

This proved to be the repeated-squaring method for modular exponentiation. The AES key was being raised to an exponent, which was passed as an argument in binary form in order to aid in the repeated-squaring algorithm. The modulus was the long number stored in the resource.

The use of modular exponentiation immediately suggested that RSA was being used. Normally, this would mean we wouldn't be able to decrypt the AES key, as we need the private key for that.

However, resource `#105` contains two numbers, and we've only used one so far. One of them is the public modulus n, and the other number is very close to it. It seemed possible that the second number was phi(n), which is needed to compute the private exponent d from the public exponent e. I wrote the following script to test it:

```
def rsa_decrypt(msg, e, n, phi_n):
        d = pow(e, -1, phi_n)
        return pow(msg, d, n)
```

Sure enough, passing in the second number as phi(n) returned the decrypted AES key! Since the RSA key was hard-coded, this meant that we had enough information to write a decryptor for any files encrypted with this sample, even if the AES key changed each time.

The Public Exponent

To generate the private exponent for the decryptor, I not only needed phi(n), but also the public exponent. However, the program generated a new public exponent each time it was run.

Upon closer inspection, I found that the public exponent was usually close to the victim ID given in the ransom note. Sometimes they matched exactly, but sometimes the exponent was slightly more than the ID, and occasionally they didn't seem to match at all.

Eventually, I found that the victim ID seemed to be randomly generated. If a negative number was generated, the bits were negated in order to produce a positive result.

```
00404a41    int32_t id = *id_ptr
00404a45    if (id s< 0)
00404a47        id = neg.d(id)
00404a4b    int32_t id_mask = id & 0x80000001
00404a4b    bool cond:0 = id_mask != 0
00404a51    if (id_mask s< 0)
00404a57        cond:0 = ((id_mask - 1) | 0xfffffffe) != 0xffffffff
00404a58    if (not(cond:0))
00404a5a        id = id + 1
```

After correcting for this, I found that either the victim ID or its negation was always close to the exponent, but there didn't seem to be much of a pattern to the exact difference.

It turned out that the victim ID sometimes needed to be modified before it could work as a public exponent. In RSA, the public exponent needs to be invertible modulo phi(n), meaning that the exponent and phi(n) need to be relatively prime. However, the process that generated the victim IDs did not guarantee a result that was relatvely prime to phi(n).

(This is just speculation, but my guess is that this is why phi(n) was hard-coded in the executable - they needed to guarantee that they had a valid public exponent, so they had to check whether the ID and phi(n) were relatively prime. However, this also gives us enough information to decrypt the files ourselves!)

By incrementing the victim ID until I got a number that was relatively prime to phi(n), I managed to retrieve the public exponent.

```
def get_relatively_prime(e, phi_n):
         while(math.gcd(e, phi_n) != 1):
                e += 2
         return e
```

Putting It All Together

We now have enough information to write a decryptor that decrypts the victim’s files using only the encrypted `.jaff` file and the ID number in the ransom note.

```
import binascii
import math
from Crypto.Cipher import AES
from struct import pack, unpack

phi_n =
3532605403186136813956330618413416701813071856948273166600165081753910874440101663323
1304437224730790638615766740272106403143256
n =
3532605403186136813956330618413416701813071856948273166600165082986456837109444420355
7601170206844003631101722202233367975968667

def convert_from_decimal(s):
        result = b''
        s_fixed = s[1:]
        for i in range(0, len(s_fixed) ,3):
                curr_num = s_fixed[i:i+3]
                result += int(curr_num).to_bytes(1, 'little')
        return result

def rsa_decrypt(msg, e, n, phi_n):
        d = pow(e, -1, phi_n)
        return pow(msg, d, n)

def get_relatively_prime(e, phi_n):
         while(math.gcd(e, phi_n) != 1):
                e += 2
         return e

def aes_decrypt(ciphertext, blob):
        iv = b'\x00'*16
        key_bytes = blob[12:]
        key = AES.new(key_bytes, AES.MODE_CBC, iv)

        padded_text = ciphertext + b'\x00'*(16 - len(ciphertext)%16)

        return key.decrypt(padded_text)

def decrypt(filename, id):

        #parse the encrypted AES key and data from the file

        enc_file = open(filename, 'rb').read()
        num_size = unpack('<I', enc_file[0:4])[0]
        key_str = enc_file[4:num_size+4]
        ciphertext = enc_file[num_size+8:]

        keys = [int(i) for i in key_str.split()]
        aes_key = []


        #test both the victim ID and its negation for a valid public exponent
```

```
exp1 = get_relatively_prime(id | 1, phi_n)
for k in keys: aes_key.append(rsa_decrypt(k, exp1, n, phi_n))
if(str(aes_key[0]))[0:6] != '100800':
        aes_key = []
        not_id = ~id & 0xffffffff
        exp2 = get_relatively_prime(not_id | 1, phi_n)
        for k in keys: aes_key.append(rsa_decrypt(k, exp2, n, phi_n))


#decode the key blob from its decimal representation

aes_key_bytes = b''
for k in aes_key: aes_key_bytes += convert_from_decimal(str(k))

return aes_decrypt(ciphertext, aes_key_bytes)
```