RESEARCH

Revisiting the latest version of Andromeda/Gamarue Malware

2015/11/05 / Blueliv

Andromeda/Gamarue malware has been prevalent since it came into limelight a couple of years ago. Also, the author keeps it well updated ever since. With respect to its earlier avatars, it has gone through several changes from anti-analysis to a change in protocol format. Some excellent write-ups have already been made on it [1][2] previously, but in this blog we **will revisit and analyze the latest version**.

Andromeda-Gamarue hides itself though many layers and its default one.

Since its inception it has made use of many techniques to defeat extraction of embedded configuration (url, keys, etc.), such as using a fake encryption, fake urls, config encryption and many more.

Meanwhile **we also found a sample which had obfuscation techniques such as opaque predicates** to hinder static-analysis.

```
mov     eax, large fs:30h
mov     ebx, [eax+0Ch]
and     [ebp+78h+var_90], 0
add     esi, 56AEC017h
add     edi, 0E7FA32D5h
imul    esi, 7223CDF8h
imul    edi, 294C50BAh
mov     eax, 47479FD7h
sub     eax, esi
xor     edi, 0FF7519F5h
mov     esi, eax
and     edi, 49FEF71Ah
mov     [ebp+78h+var_C], ebx
cmp     esi, 0CB6C0E40h
jz      loc_40294C
mov     eax, [ebx+0Ch]
imul    edi, 69CCF908h
sub     esi, 55B856C3h
mov     [ebp+78h+var_C], eax
cmp     edi, 0AD2141C7h
jz      loc_402B1C
mov     [ebp+78h+var_B0], eax
```

TOPICS

API
APT
BOTNETS
COLLABORATION
COMMUNITY
CONFERENCE
COOL VENDOR
CORPORATE
COURSE
CREDENTIAL THEFT
CREDIT CARDS
CRIME SERVERS
CSP
CYBER ATTACK
CYBER CRIME
CYBER INTELLIGENCE
CYBER SECURITY
CYBER THREAT INTELLIGENCE
CYBER THREATS
DATA BREACH
DRIDEX
DYRE
EVENTS
FEED
FORENSICS
FRAUD
GARTNER
HACKTIVISM
INDUSTRY NEWS
INTERNATIONALIZATION
IP
MALWARE
MOBILE DEVICES
MOBILE SECURITY
PLUGIN
REPORT
RESEARCH FINDINGS
SPLUNK
THREAT INTELLIGENCE
TIPS
TROJAN
TUTORIAL
VULNERABILITIES
WEBINAR
WHITEPAPER

Subscribe by RSS

Follow Blueliv

LATEST TWEETS

Andromeda-Gamarue consists of two payloads, a default unpacker and a main payload. We are going to cover up both in this post.

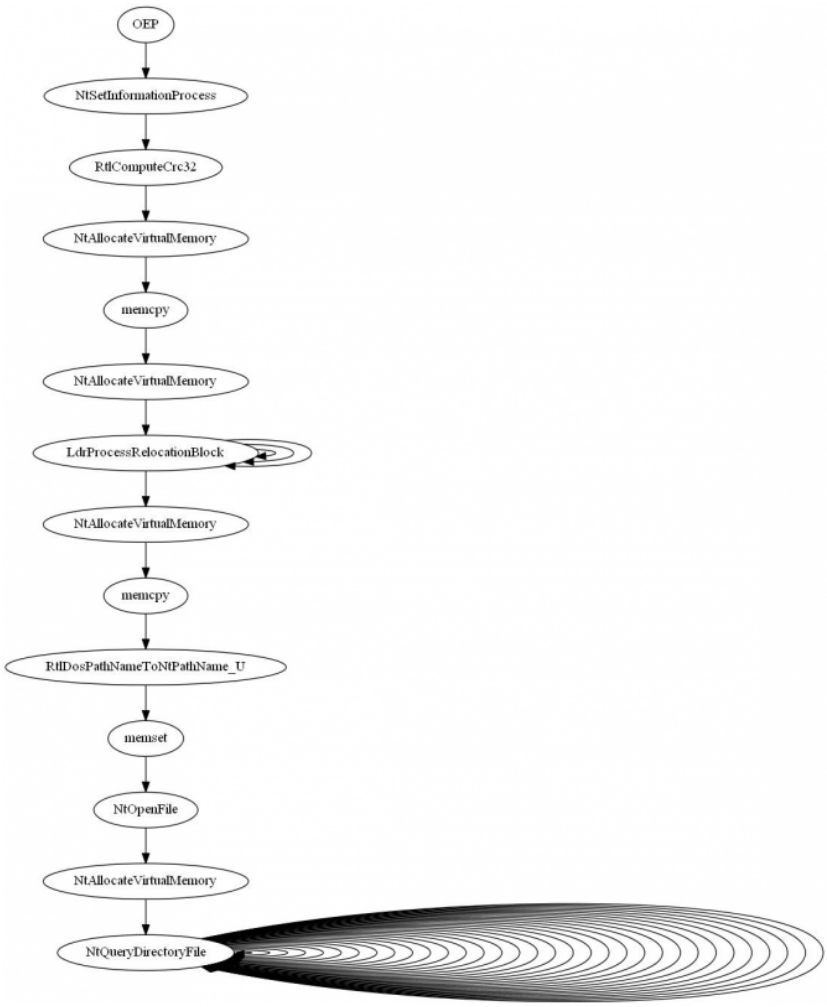
It starts with loading up some of the native functions identified by hashes using a simple hashing algorithm and stores the API address in stack variables.

```
text:00401000 ; int FnHashes[]
text:00401000 FnHashes        dd 0AB48C65h            ; DATA XREF: start+27↓r
text:00401000                                         ; start+50↓r
text:00401004                 dd 0DE604C6Ah, 925F5D71h, 0EFD32EF6h, 0B8E06C7Dh, 831D0FAAh
text:00401004                 dd 0A62BF608h, 102DE0D9h, 7CD8E53Dh, 6815415Ah, 0E7F9919Fh
text:00401004                 dd 64C4ACE4h, 28C54D3h, 82D84ED3h, 2 dup(0)
text:00401040
text:00401040 ; ============== S U B R O U T I N E =======================================
text:00401040
```

```
int __stdcall InternalHash(int buff)
{
  int ptr; // edx@1
  int result; // eax@1
  int v3; // ecx@2

  ptr = buff;
  result = 0;
  while ( *(_BYTE *)ptr )
  {
    v3 = __ROL4__(result ^ *(_BYTE *)ptr, 9);
    result = v3;
    ++ptr;
  }
  return result;
}
```

To get a basic overview of the binary we will generate a run time dynamic call graph to help us understand the functionality to some extent.



It shows some calls to LdrProcessRelocateBlock(), which gives us an indication about where and how the payload is unpacked. The binary consist of a data blob in the .rdata section of a PE file which holds information regarding the unpacked payload. It has the following structure:

```
{

    BYTE RC4Key[16]
    DWORD sizeofsection;
    DWORD crc32hash;
    DWORD UncompressedSize;
    void *OEP;
    void *RelocationTableoffset;
    void *importArry;
    BYTE BaseData[]

}
```

The integrity of the payload is checked against a hard coded crc32 hash value and, if the hash is verified, it further proceeds to decrypt and decompress the payload using a 16 byte rc4 key and APLIB decompression. This chunk is copied to an allocated heap region which is purposely created by using MEM_COMMIT or MEM_TOP_DOWN, which might be used to bypass some scanning engine or dumpers.

```
push    dword ptr [edi+10h] ; Size
lea     eax, [edi+28h]
push    eax
push    [ebp+Allocmemory]
call    [ebp+memcpy]
add     esp, 0Ch
push    dword ptr [edi+10h] ; buffersize
push    [ebp+Allocmemory] ; buffer
push    10h             ; keysize
push    edi             ; key
call    rc4_Payload
mov     eax, [edi+18h]
and     [ebp+DecompressedBlock], 0
push    PAGE_EXECUTE_READWRITE
push    ebx             ; MEM_COMMIT or MEM_TOP_DOWN
mov     [ebp+nInst], eax
lea     eax, [ebp+nInst]
push    eax
push    0
lea     eax, [ebp+DecompressedBlock]
push    eax
push    0FFFFFFFFh
call    [ebp+ZwAllocateVirtualMemory]
cmp     [ebp+DecompressedBlock], 0
jz      ExitPath2
```

The base relocations are applied on that memory region using the RelocationTableOffset field.

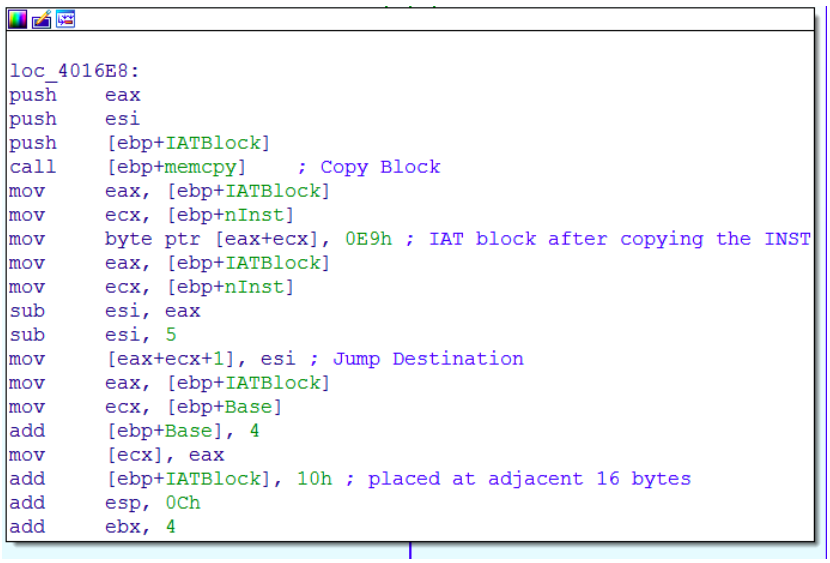
```
ProcessRelocation:
mov     ecx, [ebp+DecompressedBlock]
push    ecx             ; baseaddress = Delta
lea     edx, [esi+8]
push    edx
mov     edx, [esi+4]
sub     edx, 8
shr     edx, 1
push    edx
add     eax, ecx
push    eax
call    [ebp+LdrProcessRelocationBlock]
mov     esi, eax
mov     eax, [esi]
test    eax, eax
jnz     short ProcessRelocation
```

Another block of executable memory region of size 1000h is allocated, which will later on be used for copying stolen API code. Then, Dll and Imports are parsed. Dll names can again be found as hashes.

The first instruction is copied from an API location to this particular memory region and a succeeding jump is placed after that to the original instructions. This is done to bypass API hooking. It consists of an x86 instruction parsing subroutine.

```
.text:004018D3 regMemMode      db 11h                  ; DATA XREF: GetInstructionLength+2F↑r
.text:004018D4                 dd 4 dup(11002811h), 3 dup(11F02811h), 0F02811h, 4 dup(0)
.text:00401904                 dd 89FFFF11h, 22000023h, 22222222h, 39222222h, 11111133h
.text:00401904                 dd 111111h, 0
.text:00401920                 dd 880000C0h, 28000088h, 22000000h, 88222222h, 33888888h
.text:00401920                 dd 60391140h, 11000240h, 11002211h, 22111111h, 88222222h
.text:00401920                 dd 0F00000C2h, 1100FFh, 11110000h, 0E011h, 1103E1EEh, 1E111111h
.text:00401920                 dd 11EEEEEEh, 111E1E11h, 111111h, 0EEEE0000h, 11EEEEEEh
.text:00401920                 dd 5 dup(11111111h), 33111111h, 11101133h, 88111111h, 88888888h
.text:00401920                 dd 11888888h, 11111111h, 111111h, 113101h, 11113101h, 0EE111111h
.text:00401920                 dd 11111131h, 313331h, 0E1000000h, 3 dup(11111111h), 0E1111111h
.text:00401920                 dd 11111111h
```

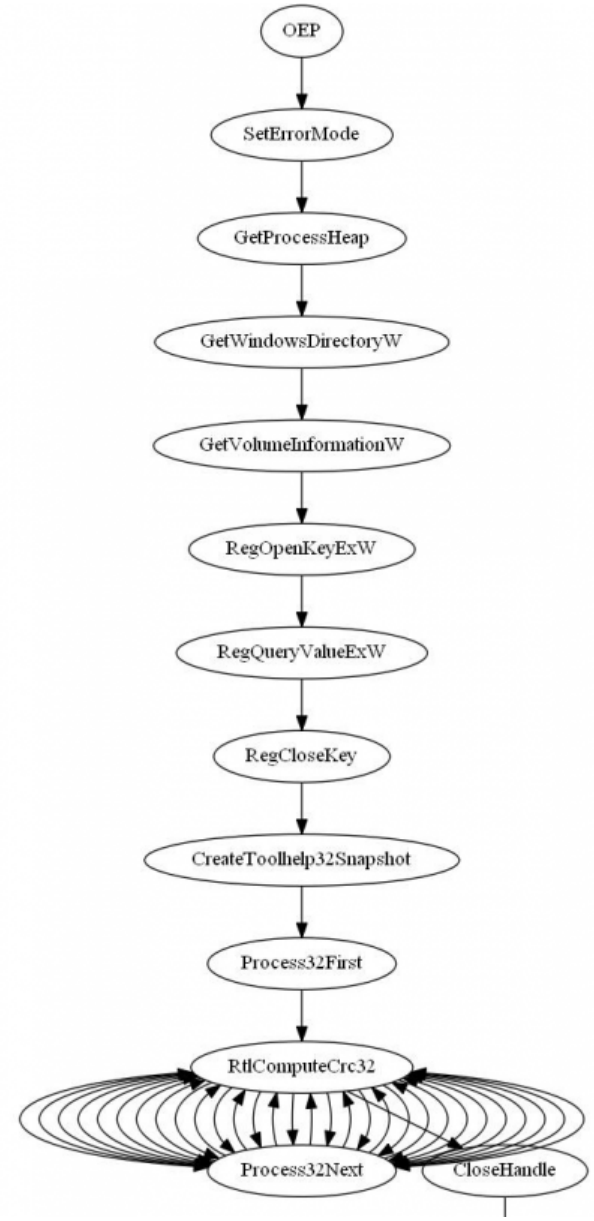
Subroutine calls and unconditional jumps follow, subsequent instructions are copied and a jump to OEP is made.



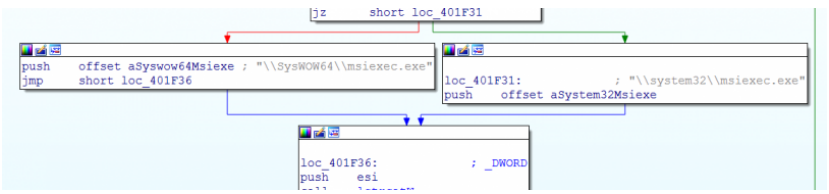
```
loc_4016E8:
push    eax
push    esi
push    [ebp+IATBlock]
call    [ebp+memcpy]    ; Copy Block
mov     eax, [ebp+IATBlock]
mov     ecx, [ebp+nInst]
mov     byte ptr [eax+ecx], 0E9h ; IAT block after copying the INST
mov     eax, [ebp+IATBlock]
mov     ecx, [ebp+nInst]
sub     esi, eax
sub     esi, 5
mov     [eax+ecx+1], esi ; Jump Destination
mov     eax, [ebp+IATBlock]
mov     ecx, [ebp+Base]
add     [ebp+Base], 4
mov     [ecx], eax
add     [ebp+IATBlock], 10h ; placed at adjacent 16 bytes
add     esp, 0Ch
add     ebx, 4
```

MAIN PAYLOAD

The main payload consists of an installer and a primary payload responsible for communicating to the command and control centre. Let's take a look to the call graph of the installer part:

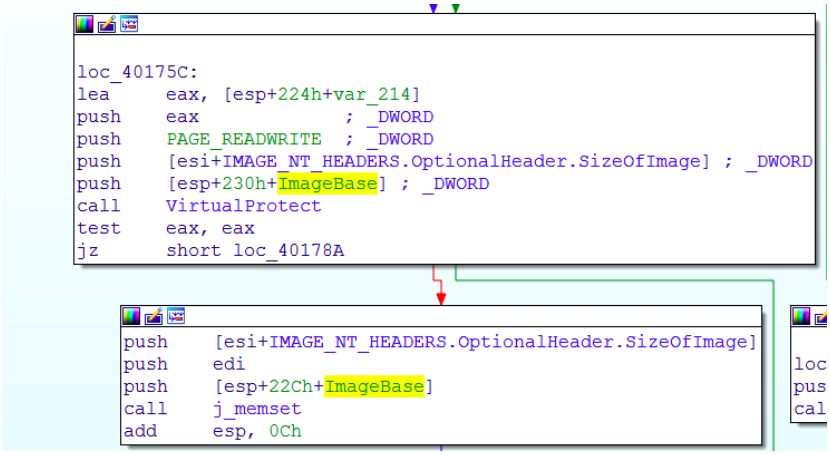


It starts by getting serial number for the root drive (which will later on be used as a part in the c2 request). It also has a function to check for the presence of certain processes and if they are found it goes in an infinite loop. These checks are bypassed if a registry key "is_not_vm" is found in HEY_LOCAL_MACHINE software\\policies. The key has to be equal to VolumeSerialNumber. An environment variable is created from xoring VolumeSerialNumber with 0x737263, which is assigned to the module file name. This environment variable acts as an indicator for the previous instance of binary. It also sets up an event named after xoring VolumeSerialNumber xoring with 0x696E6A63.

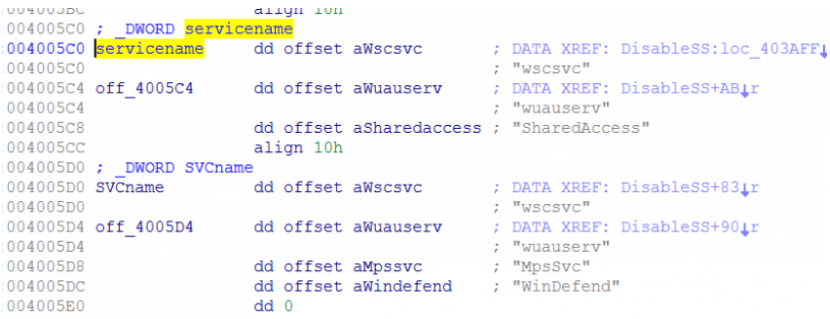


This payload is injected inside "msiexec.exe" by changing the entry point to push <base of injected code> ret and waits for the event to be triggered by the main payload.

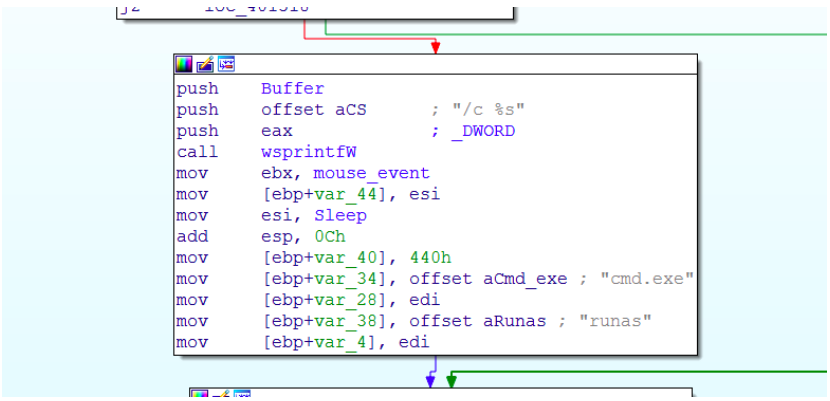
The main payload nulls the packer PE headers and sections.



Following this, it adjusts the privileges, sets TaskbarNoNotification, and disables UAC, Windows Action centre, as well as some security related services (only if the "bb" parameter is not set). Explained below:



If necessary privileges are not found, it will try to elevate the privileges by using the "Runas" verb.



C2 servers are encrypted and stored using a crc32 hash of PE data and an incremental XOR value.

After that, it makes connection to each c2 with the following json request:

{"id":%lu,"bid":%lu,"os":%lu,"la":%lu,"rg":%lu,"bb":%lu}

ID = VolumeSerialNumber

BID = botnetID

OS = OSVersion

LA = Local IP address

RG = isprivileged?

BB = islocalized (Russia, Ukraine, Belarus and Kazakhstan)

This request is encrypted using a 32 bytes rc4 key and the response is also decrypted using the same rc4 key (earlier versions would have used 4 bytes ID as a response key). The request also comes in a JSON format now. It consists of a json parser compiled from https://github.com/udp/json-parser/.

The return value from jsonparser is represented this way:

```
typedef struct _json_value
{
   struct _json_value * parent;

   int type;

   union
   {
      int boolean;
      json_int_t integer; // 8 bytes
      double dbl;

      struct
      {
         unsigned int length;
         json_char * ptr; /* null terminated */

      } string;

      struct
      {
         unsigned int length;

         json_object_entry * values;

       } object;
      struct
      {
         unsigned int length;
         struct _json_value ** values;


      } array;
```

The above JSON structure is expressed in the following format:

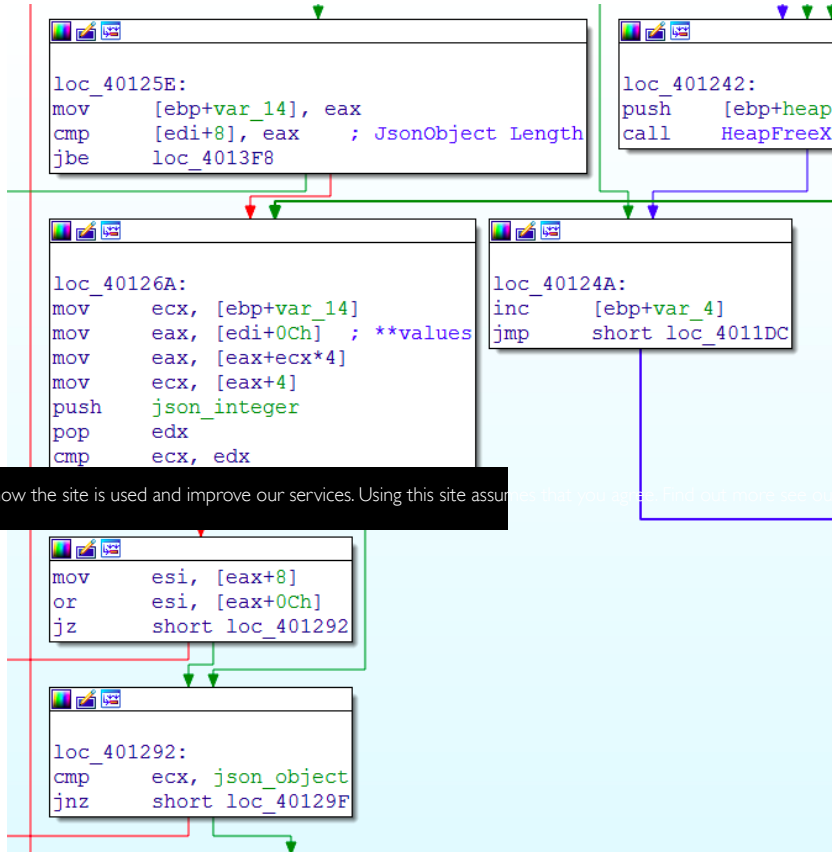
[next_request_sleeptime (minutes) ,{Unimplemented_object}, [TaskID, RequestType, 'URL'-N/A]……]

The first item in the array is the next request sleep time. It is the time frame in minutes when next iteration of calling c2 is performed.

The second in the list is an unimplemented / unused type. When this object is found, it is simply skipped.

The rest are single or multiple arrays which may consist of a url payload. TaskID is the UID of a task provided by the c2 server. This ID is sent back in a following request. The request type is an identifier of

the task type of an eg download url, plugin download or delete bot.



These urls can either be exe or plugins. Plugins are encrypted and compressed with RC4 and APlib. After completing the specified task, another request is sent back to the c2 server which has the following format:

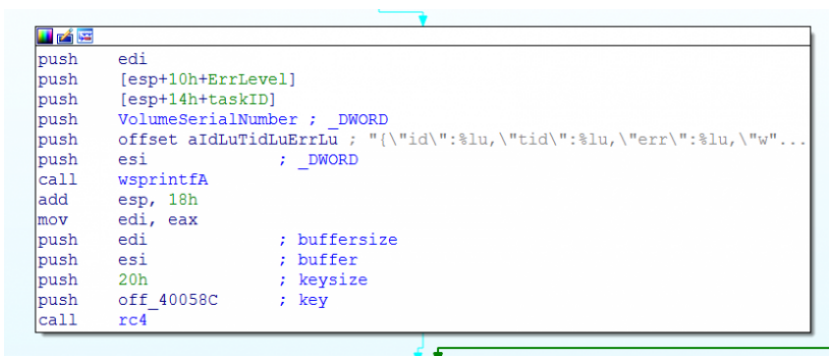
{"id":%lu,"tid":%lu,"err":%lu,"w32" :%lu}

ID:VolumeSerialNumber

TID:TaskID

ERR: Error Level on task completion (0 – no error starting from 0x10)

W32: Error Number from GetLastError()



Raashid Bhat

Malware Analyst

‹ Previous entry

Next entry ›