BAZARLOADER: Unpacking an ISO File Infection

April 19, 2022

```
qmemcpy(stack_str, v16, 0xDui64);           // Kernel32.dll
v2 = 0i64;
stack_str[0xD] = 0;
do
{
  stack_str[v2] = ((0xFFFFFFC2 * (stack_str[v2] - 0x33)) % 0x7F + 0x7F) % 0x7F;
  ++v2;
}
while ( v2 != 0xD );
kernel32_dll_handle = ::GetModuleHandleA(stack_str);// "Kernel32.dll"
output->kernel32_handle = kernel32_dll_handle;
if ( kernel32_dll_handle )
{
  *&v19[8] = 0x4C0C7878;
  *v19 = 0x925110C6F334C7Ei64;
  qmemcpy(&v19[0xC], "__K", 3);
  qmemcpy(stack_str, v19, 0xFui64);
  v4 = 0i64;
  stack_str[0xF] = 0;
  do
  {
    stack_str[v4] = ((0xFFFFFFE6 * (stack_str[v4] - 0x4B)) % 0x7F + 0x7F) % 0x7F;
    ++v4;
  }
  while ( v4 != 0xF );
  GetProcAddress = ::GetProcAddress(kernel32_dll_handle, stack_str);// "GetProcAddress"
  *&v18[8] = 0x42680E7A;
  v6 = 0i64;
  output->mw_GetProcAddress = GetProcAddress;
```

- Chuong Dong
- 19th April 2022
- 1 Comment

BAZARLOADER (aka BAZARBACKDOOR) is a Windows-based loader that spreads through attachments in phishing emails. During an infection, the final loader payload typically downloads and executes a Cobalt Strike beacon to provide remote access for the threat actors, which, in a lot of cases, leads to ransomware being deployed to the victim's machine.

In this initial post, we will unpack the different stages of a BAZARLOADER infection that comes in the form of an optical disk image (ISO) file. We will also dive into the obfuscation methods used by the main BAZARLOADER payload.

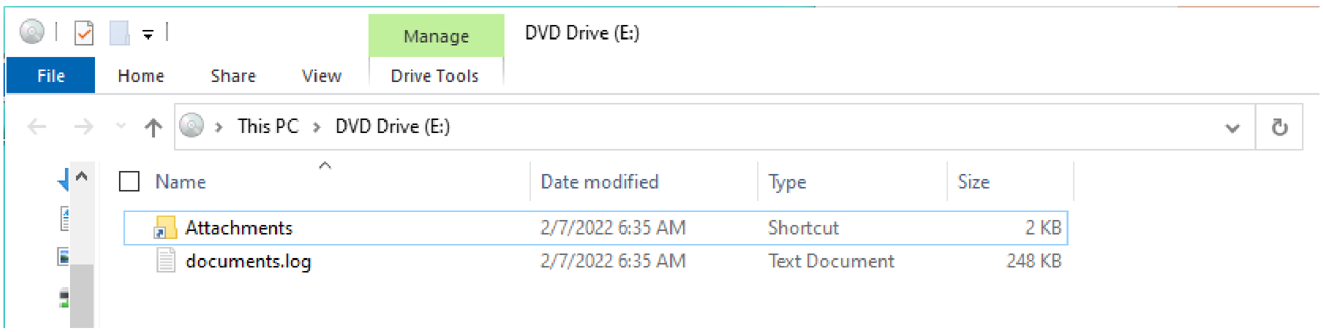
To follow along, you can grab the sample as well as the PCAP files for it on Malware-Traffic-Analysis.net.

SHA256: 0900b4eb02bdcaefd21df169d21794c8c70bfbc68b2f0612861fcabc82f28149

Step 1: Mounting ISO File & Extracting Stage 1 Executable

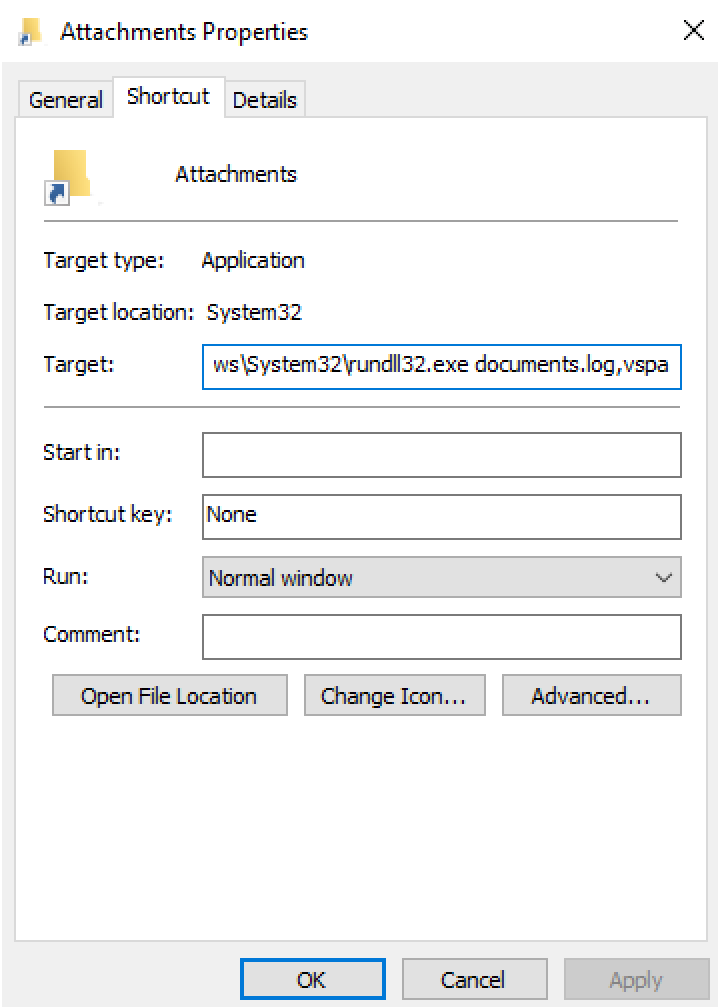
Recent BAZARLOADER samples arrive in emails containing OneDrive links to download an ISO file to avoid detection since most AVs tend to ignore this particular file type. With Windows 7 and above integrating the mounting functionality into Windows Explorer, we can mount any ISO file as a virtual drive by double-clicking on it.

When we mount the malicious ISO file, we see that a drive is mounted on the system that contains a shortcut file named **“Attachments.lnk”** and a hidden file named **“documents.log”**.



The shortcut file has to be run by the victim to begin the chain of infection. We can quickly extract the actual command being executed by this shortcut from its **Properties** window.

```
C:\Windows\System32\rundll32.exe documents.log,vspa
```



Once the victim double-clicks on the shortcut file, the command executes the Windows **rundll32.exe** program to launch the **"documents.log"** file. This lets us know that the file being launched is a DLL file, and the entry point is its export function **vspa**.

Step 2: Extracting Second Stage Shellcode

Taking a quick look in IDA, we can somewhat tell that the extracted DLL is packed since it has only a few functions and a really suspicious looking buffer of ASCII characters in its custom **.odata** section.

```
odata:000000018001D000 odata          segment para public 'DATA' use64
odata:000000018001D000                assume cs:odata
odata:000000018001D000                ;org 18001D000h
odata:000000018001D000 a46a52995f9a819 db '46a52995f9a8193ba0efa7b0724a8a0120d51ff08d25efd057bd2e82ddb7cf3b7'
odata:000000018001D000                                     ; DATA XREF: sub_180003DDA+16B↑o
odata:000000018001D000                db '8b10a67cce05ae7ba5165f1056a55e16018b4ed9209b7abab438990ad1265733d'
odata:000000018001D000                db '9f8c63da54c29ddca1b52ce9a76bb242593b05d64fbb5a3fb69597d3d585e816a'
odata:000000018001D000                db 'fb8b4732d7ce7a8e6295859b3500849a3c414a2513d8a47f5f526fc2134bcd790'
odata:000000018001D000                db '2ca04ec5f10f6ee393e94f784db16e720860b7fbdbc14627c340d97da2ec61f64'
odata:000000018001D000                db '0a76dda73907e734fc639336b9d877aea3cd4ea1e6f06da14899a0cc00be9348b'
odata:000000018001D000                db 'b8fcc24344aeae404958ac307254dde2d46c622acf094f3cf91d1749b1ec6650c'
odata:000000018001D000                db '34248e0cf8d5d9be621dce73c5485fae5284e1dc89d11a92bbada73f85452fe7e'
odata:000000018001D000                db '0f80b1949ce5b5a77bae11b971dc7cda9a4289c4f41a0e75ff7688f201ac670e5'
odata:000000018001D000                db '822582d189e6a404b6e1a4306fec1ceea8f4197751d09e9a3c7ae2a5a5777d785'
odata:000000018001D000                db '75736804b1eeb59541b14e0b2da06fbca87a8d7f182cc2bec9c2216fb5d008ec3'
odata:000000018001D000                db 'a709bdd81f443819ef59b2b76fb27b62f3a65e601d58a7ac034b06d004981cee0'
odata:000000018001D000                db '9fce980c429bfb99f57ce5aa327f3345e90025c8b4d86465e02128fb43ae3c993'
odata:000000018001D000                db '05ef4d217d4d59c9a55180957348d81dcd6d1f23228813751267c17bbdcff7cb9'
odata:000000018001D000                db '9094910fe27d00bfa451089831f6e3ba570e0d2f6acf622fccb0ab7852d59b326'
odata:000000018001D000                db 'c27b490b46adfad18756317257078fa0bcd7dc652976af11db8d80e390ff58cf4'
odata:000000018001D000                db '80767409e573309171b2e67ddc514344dc5ae51935f276acfd95b00bc98ce1376'
odata:000000018001D000                db '3bdc2580c907eaa727bedc1142133309a4e6ba1df780983d3611b1e228181587d'
odata:000000018001D000                db '1c3cb4972d35037fb8c864608baae3defd64b5be63d19bf3b969793721002cb35'
odata:000000018001D000                db '4bc685d2806e402edd6b2b0342be26b9e5867953537e136f61c79c81523c57c09'
odata:000000018001D000                db 'e7b293cb93446930f52ad4759b3db809084cf0e810bfe90398b25060023f838c0'
odata:000000018001D000                db '8babf665b844c31172b257887359be222f94c5fc49c6d4b3bad26c25fc2056061'
odata:000000018001D000                db '3523582c41233ebdb56159416efbd5df899b9cac591e93b8db3c6fc8b329ac81a'
odata:000000018001D000                db '627bd11bbc66e7ebc49d01abd0ed6e734f8995df02e3205823e99aa5efdc75025'
odata:000000018001D000                db '985ef694758f157b5d51eea17e973af119da097b54f9def2201e94bbb1523dff6'
odata:000000018001D000                db '03405df7916cf4fda8d6fc9059a2a7fb1470460cf286ac93e3928fe1d85125be2'
```

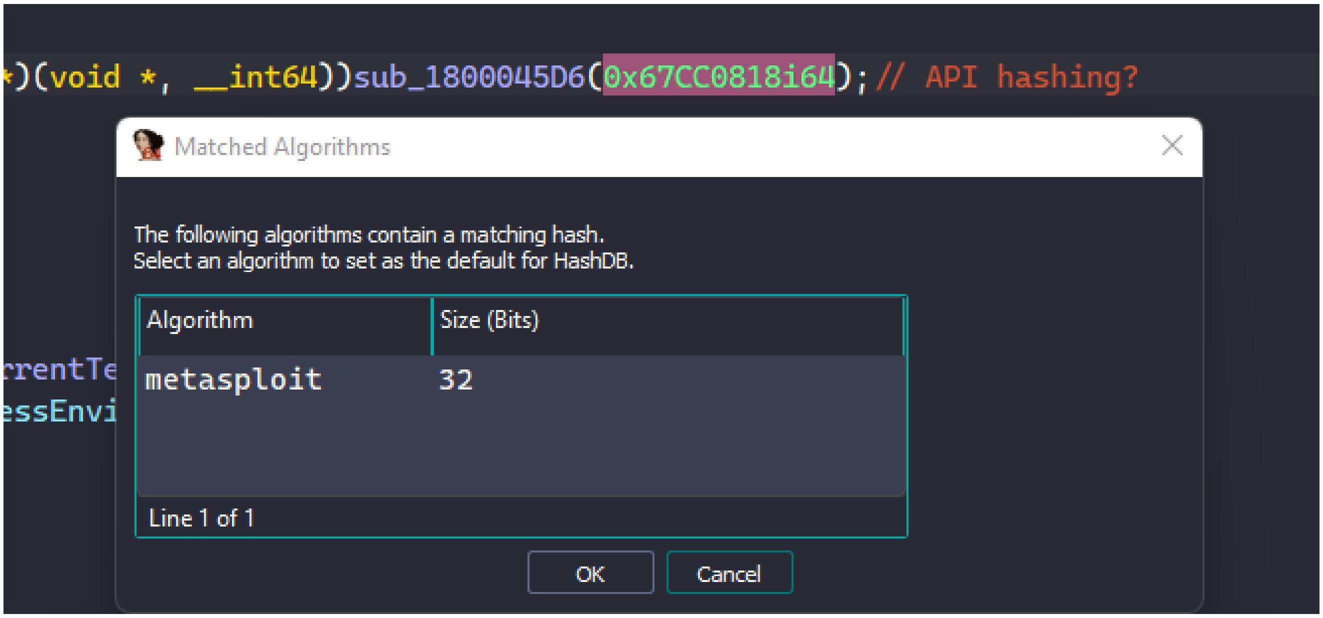
With that in mind, we will just perform some quick static analysis to determine where we can dump the next stage.

In the first function of the **vspa** export, we see **sub_1800045D6** takes a DWORD in as the parameter. This function returns a variable that contains an address to a function that is later called in the code.

```
API_func = (__int64 (__fastcall *)(void *, __int64))sub_1800045D6(0x67CC0818i64);// API hashing?
v9 = a46a52995f9a819;
for ( i = 0; !i; i = 1 )
  DWORD2(v6) += 147086;
if ( DWORD2(v6) )
{
  v7 = DWORD2(v6);
  ProcessEnvironmentBlock = NtCurrentTeb()->ProcessEnvironmentBlock;
  *(_QWORD *)&v6 = API_func(ProcessEnvironmentBlock->ProcessHeap, 8i64);
  v5 = 0;
  for ( j = 0; !j; j = 1 )
  {
    v10 = v5 + (_QWORD)v6;
    sub_180004E6D(v10);
    v5 += 147086;
  }
```

At this point, we can safely guess that **sub_1800045D6** is an API resolving function, and the parameter it takes is the hash of the API's name. Because this is still the unpacking phase, we won't dive too deep into analyzing this function.

Instead, I'll just use **OALabs's HashDB** IDA plugin to quickly reverse-lookup the hashing algorithm used from the hash. The result shows that the hash corresponds to an API name hashed with Metasploit's hashing algorithm **ROR13**.



After determining the hashing algorithm, we can use **HashDB** to quickly look up the APIs being resolved by this function. It becomes clear that this function resolves the **RtlAllocateHeap** API, calls that to allocate a heap buffer and writes the encoded ASCII data to it.

```
sub_180004DEF(&heap_buffer_1, 0i64);
RtlAllocateHeap = API_hashing(RtlAllocateHeap_0);// resolve RtlAllocateHeap
encoded_ASCII_buffer = a46a52995f9a819;
for ( i = 0; !i; i = 1 )
  heap_buffer_len_1 += 0x23E8E;
if ( heap_buffer_len_1 )
{
  heap_buffer_len = heap_buffer_len_1;
  ProcessEnvironmentBlock = NtCurrentTeb()->ProcessEnvironmentBlock;
  *(_QWORD *)&heap_buffer_1 = ((__int64 (__fastcall *)(void *, __int64, __int64))RtlAllocateHeap)(
                                ProcessEnvironmentBlock->ProcessHeap,
                                8i64,
                                heap_buffer_len);// allocate heap buffer of 0x23E8E bytes
  v5 = 0;
  for ( j = 0; !j; j = 1 )
  {
    heap_buffer = (void *)(v5 + *(_QWORD *)&heap_buffer_1);
    w_memcpy(heap_buffer, encoded_ASCII_buffer, 0x23E8Eui64);// copies encoded data to heap buffer
    v5 += 0x23E8E;
  }
}
```

From this point onward, we can guess that the packer will decode this buffer and launch it somewhere later in the code. If we skip toward the end of the **vspa** export, we see a **call** instructionon a variable that is not returned from the API resolving function, so it can potentially be our tail jump.

```
if ( (unsigned int)sub_180003FE6(&v19, (__int64)&v25, 0x40u) )
{
  return 0i64;
}
else
{
  qword_18001A9E0 = 0i64;
  v15 = 0i64;
  strcpy(v13, "vh5;");
  v13[1] += 11;
  v13[2] += 59;
  v13[3] += 38;
  v18 = v13;
  qword_18001A9E0 = v19(v26[4], qword_18001A9E8);// v19 is not returned from API_resolve?
```

The last function to modify that **v19** variable is **sub_180003FE6**, so we can quickly take a look at that.

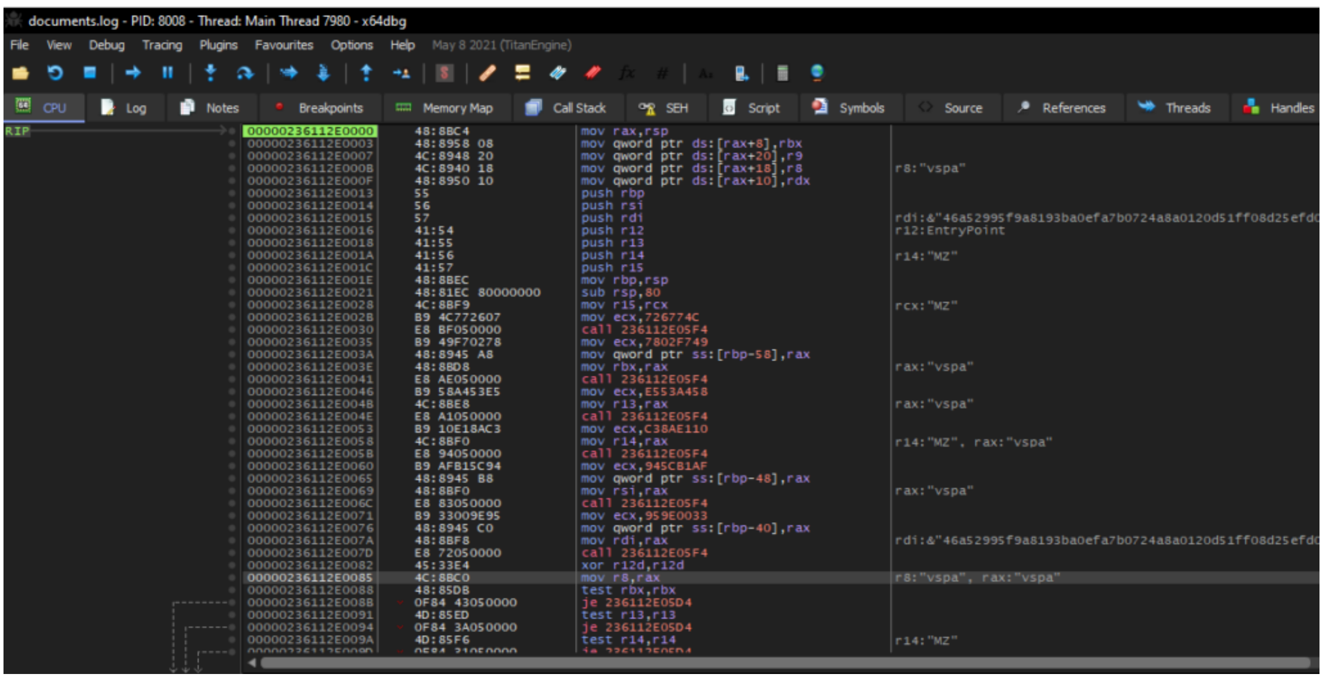
It turns out the **sub_180003FE6** function just resolves and calls **NtMapViewOfSection** to map a view of a section into the virtual address space and writes the base address of the view into the **v19** variable. Then, it just executes **qmemcpy** to copy the data in the second variable to the returned virtual base address.

```
__int64 __fastcall w_map_view_of_section(__int64 BaseAddress, __int64 ViewSize, int Win32Protect)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  NtCreateSection = API_hashing(NtCreateSection_0);
  NtMapViewOfSection = API_hashing(NtMapViewOfSection_0);
  NtClose = API_hashing(NtClose_0);
  section_handle = 0i64;
  v8 = 0i64;
  ViewSize_1 = ViewSize;
  v4 = (NtCreateSection)(&section_handle, 0xF001Fi64);
  if ( v4 ≥ 0 )
  {
    v4 = (NtMapViewOfSection)(
           section_handle,
           0xFFFFFFFFFFFFFFFFui64,
           BaseAddress,
           0i64,
           0i64,
           0i64,
           &v8,
           1,
           0,
           Win32Protect);
    (NtClose)(section_handle);
  }
  return v4;
```

```
__int64 __fastcall sub_180003FE6(_QWORD *a1, __int64 a2, int a3)
{
  int v4; // [rsp+0h] [rbp-18h]

  v4 = w_map_view_of_section(a1, *(a2 + 8), a3);
  if ( v4 ≥ 0 )
    w_qmemcpy(*a1, *a2, *(a2 + 8));
  return v4;
}
```

This tells us two things. First, our guess that the **v19** variable will contain the address to executable code is correct. Second, we know that the executable code is shellcode since the data is mapped and executed directly at offset 0 from where it is written.

From here, we can set up **x64dbg**, execute the DLL file at the **vspa** export, and break at the call instruction. After stepping into the function, we will be at the head of the shellcode.



We can now dump this virtual memory buffer to retrieve the second stage shellcode for the next unpacking step.

Step 3: Extracting The Final BAZARLOADER Payload

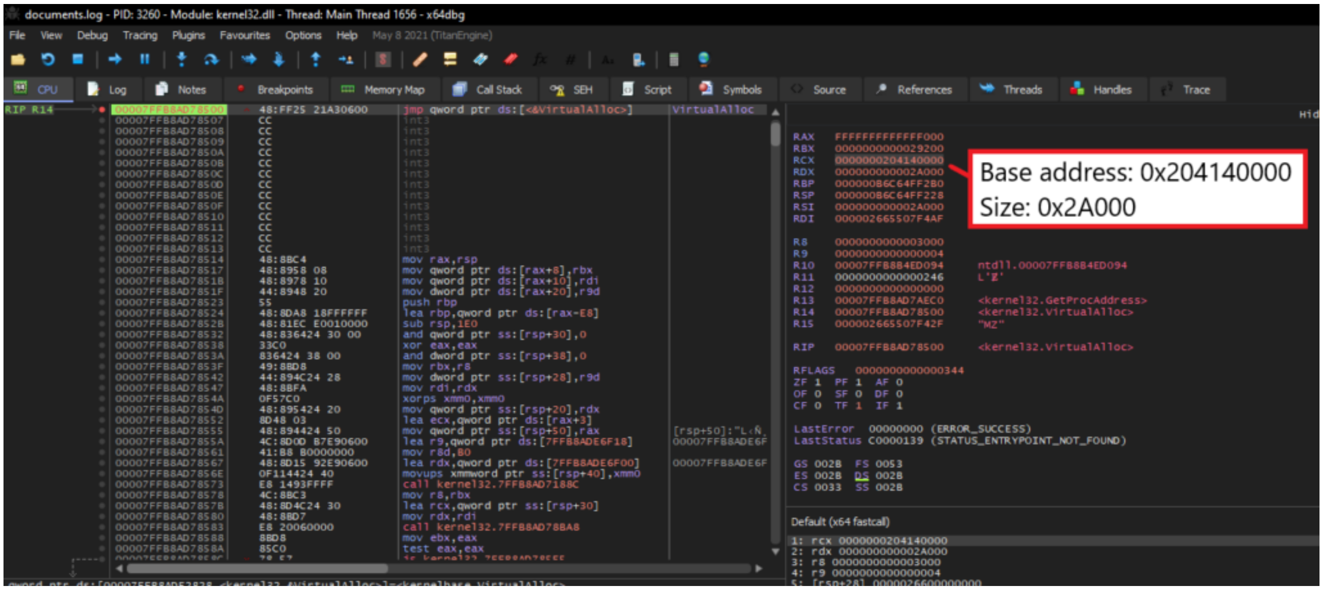
When we examine the shellcode in IDA, we can quickly use the same trick with **HashDB** above to see that the shellcode also performs API hashing with Metasploit's **ROR13**.

```
__int64 __fastcall sub_0(__int64 a1, __int64 a2, char *a3, _QWORD *a4)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  LoadLibraryA = sub_5F4(LoadLibraryA_0);
  GetProcAddress = sub_5F4(GetProcAddress_0);
  VirtualAlloc = sub_5F4(VirtualAlloc_0);
  VirtualProtect = sub_5F4(VirtualProtect_0);
  NtFlushInstructionCache = sub_5F4(NtFlushInstructionCache_0);
  GetNativeSystemInfo = sub_5F4(GetNativeSystemInfo_0);
  GetNativeSystemInfo_1 = GetNativeSystemInfo;
  if ( !LoadLibraryA )
    return 0xFFFFFFFFFFFFFFFFui64;
  if ( !GetProcAddress )
    return 0xFFFFFFFFFFFFFFFFui64;
  if ( !VirtualAlloc )
    return 0xFFFFFFFFFFFFFFFFui64;
  if ( !VirtualProtect )
    return 0xFFFFFFFFFFFFFFFFui64;
  if ( !NtFlushInstructionCache )
    return 0xFFFFFFFFFFFFFFFFui64;
  if ( !GetNativeSystemInfo )
    return 0xFFFFFFFFFFFFFFFFui64;
```

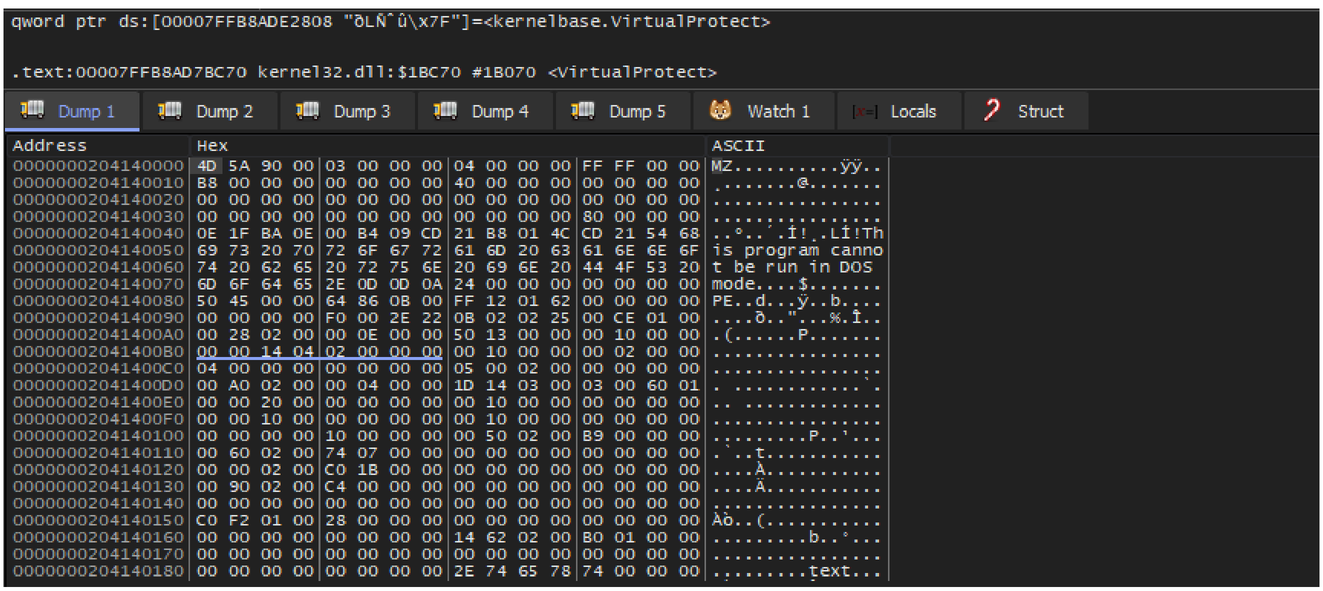
At the entry point above, the shellcode resolves a set of functions that it will call, most notably **VirtualAlloc** and **VirtualProtect**. These two functions are typically used by packers to allocate virtual memory to decode and write the next stage executable in before launching it.

With this in mind, our next step should be debugging the shellcode and setting breakpoints at these two API calls. We can pick up where we are after dumping in **x64dbg** during **Step 2**, or we can launch the shellcode directly in our debugger using OALabs's BlobRunner or similar shellcode launcher.

Our first hit with **VirtualAlloc** is a call to allocate a virtual memory buffer at virtual address 0x204140000 with the size of 0x2A000 bytes.



We can run until **VirtualAlloc** returns and start monitoring the memory at address 0x204140000. After running until the next **VirtualProtect** call, we see that a valid PE executable has been written to this memory region.



Finally, we can dump this memory region into a file to extract the BAZARLOADER payload.

Step 4: BAZARLOADER's String Obfuscation

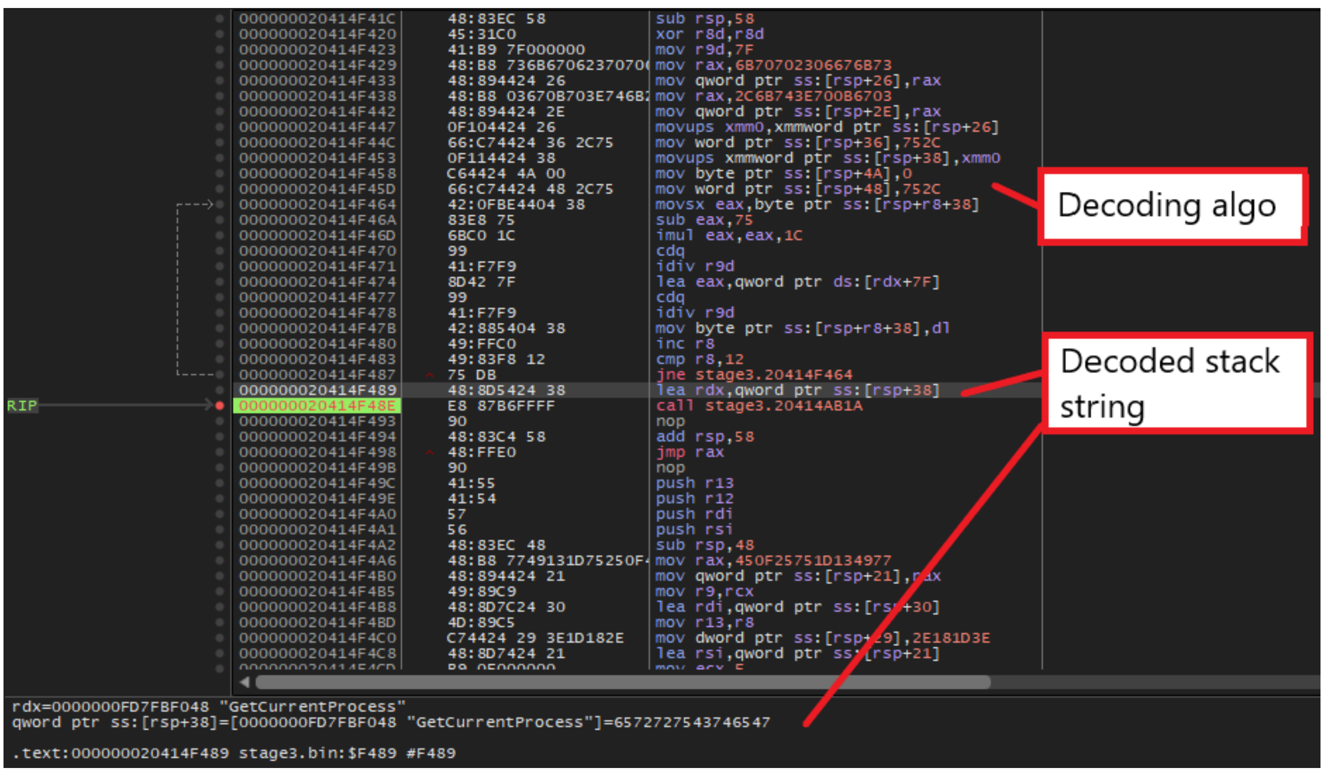
As we begin performing static analysis on BAZARLOADER, it is crucial that we identify obfuscation methods that the malware uses.

One of those methods is string obfuscation, where the malware uses encoded stack strings to hide them from static analysis.

```
v1 = 0i64;
*&v4 = 0x6B70702306676B73i64;
*(&v4 + 1) = 0x2C6B743E700B6703i64;
v5 = v4;
strcpy(v6, ",u");
do
{
  v6[v1 - 0x10] = (0x1C * (v6[v1 - 0x10] - 0x75) % 0x7F + 0x7F) % 0x7F;
  ++v1;
}
```

As shown, a typical encoded string is pushed on the stack and decoded dynamically using some multiplication, subtraction, and modulus operations.

There are different ways to resolve these stack strings, such as writing IDAPython scripts, emulation, or just running the program in a debugger and dumping the stack strings when they are resolved.



Step 5: BAZARLOADER’s API Obfuscation

BAZARLOADER obfuscates most of its API calls through a few structures that it constructs in the **DllEntryPoint** function.

First, the malware populates the following structure that contains a handle to **Kernel32.dll** and addresses to API required to load libraries and get their API addresses.

```
struct API_IMPORT_STRUCT {
    HANDLE kernel32_handle;
    FARPROC mw_GetProcAddress;
    FARPROC mw_LoadLibraryW;
    FARPROC mw_LoadLibraryA;
    FARPROC mw_LoadLibraryA2;
    FARPROC mw_FreeLibrary;
    FARPROC mw_GetModuleHandleW;
    FARPROC mw_GetModuleHandleA;
};
```

It calls **GetModuleHandle** to retrieve the handle to **Kernel32.dll**, calls **GetProcAddress** to retrieve the address of the **GetProcAddress** API, and writes those in the structure.

```
qmemcpy(stack_str, v16, 0xDui64);             // Kernel32.dll
v2 = 0i64;
stack_str[0xD] = 0;
do
{
  stack_str[v2] = ((0xFFFFFFC2 * (stack_str[v2] - 0x33)) % 0x7F + 0x7F) % 0x7F;
  ++v2;
}
while ( v2 != 0xD );
kernel32_dll_handle = ::GetModuleHandleA(stack_str);// "Kernel32.dll"
output->kernel32_handle = kernel32_dll_handle;
if ( kernel32_dll_handle )
{
  *&v19[8] = 0x4C0C7878;
  *v19 = 0x925110C6F334C7Ei64;
  qmemcpy(&v19[0xC], "__K", 3);
  qmemcpy(stack_str, v19, 0xFui64);
  v4 = 0i64;
  stack_str[0xF] = 0;
  do
  {
    stack_str[v4] = ((0xFFFFFFE6 * (stack_str[v4] - 0x4B)) % 0x7F + 0x7F) % 0x7F;
    ++v4;
  }
  while ( v4 != 0xF );
  GetProcAddress = ::GetProcAddress(kernel32_dll_handle, stack_str);// "GetProcAddress"
  *&v18[8] = 0x42680E7A;
  v6 = 0i64;
  output->mw_GetProcAddress = GetProcAddress;
```

Using the structure's **GetProcAddress** API field, BAZARLOADER retrieves the rest of the required APIs to populate other fields in the structure. This **API_IMPORT_STRUCT** structure will later be used to import other libraries' APIs.

```
GetModuleHandleW = get_proc_addr(output, output->kernel32_handle, stack_str);// "GetModuleHandleW"
v21 = 6;
v14 = 0i64;
output->mw_GetModuleHandleW = GetModuleHandleW;
*&v20 = 0x5E3B6F28114D5D7Di64;
*(&v20 + 1) = 0x6A5D5E6F3A266B5Di64;
v26 = 0;
*stack_str = v20;
v25 = 6;
do
{
  stack_str[v14] = (7 * (stack_str[v14] - 6) % 0x7F + 0x7F) % 0x7F;
  ++v14;
}
while ( v14 != 0x11 );
GetModuleHandleA = get_proc_addr(output, output->kernel32_handle, stack_str);// "GetModuleHandleA"
output->mw_GetModuleHandleA = GetModuleHandleA;
```

Next, for each library to be imported, BAZARLOADER populates the following **LIBRARY_STRUCT** structure that contains a set of functions to interact with the library and the library handle.

```
struct LIB_FUNCS
{
    FARPROC free_lib;
    FARPROC w_free_lib;
    __int64 (__fastcall *get_API_addr)(API_IMPORT_STRUCT*, HANDLE, char*);
};

struct LIBRARY_STRUCT
{
    LIB_FUNCS *lib_funcs;
    HANDLE lib_handle;
};
```

The first 2 functions in the **LIB_FUNCS** structure just call the **FreeLibrary** API from the global **API_IMPORT_STRUCT** to free the library module.

The third function calls the **GetProcAddress** from the **API_IMPORT_STRUCT**'s field to retrieve the address of an API exported from that specific library.

```
int __fastcall free_lib(LIBRARY_STRUCT *lib_struct)
{
  char *v1; // rax
  HANDLE lib_handle; // rcx

  v1 = &unk_20415F090 + 0x10;
  lib_struct->lib_funcs = (&unk_20415F090 + 0x10);
  lib_handle = lib_struct->lib_handle;
  if ( lib_handle )
    LODWORD(v1) = (API_IMPORT_STRUCT->mw_FreeLibrary)(lib_handle);
  return v1;
}
```

```
__int64 __fastcall get_proc_addr(API_IMPORT_STRUCT *API_IMPORT_STRUCT, __int64 library_handle, __int64 API_to_find)
{
  return (API_IMPORT_STRUCT->mw_GetProcAddress)(library_handle, API_to_find);
}
```

To begin populating each **LIBRARY_STRUCT** structure, BAZARLOADER decodes the library name from a stack string and populates it with the corresponding set of functions and the library handle retrieved from calling **LoadLibraryA**.

```
int __fastcall set_up_lib_struct(LIBRARY_STRUCT *output, __int64 library_name)
{
  bool v2; // zf
  API_IMPORT_STRUCT *v5; // rsi
  void *library_handle; // rax

  v2 = API_IMPORT_STRUCT == 0i64;
  output->lib_funcs = (&unk_20415F090 + 0x10);
  output->lib_handle = 0i64;
  if ( v2 )
  {
    v5 = w_HeapAlloc(0x48ui64);
    populate_kernel32_funcs_maybe(v5);
    API_IMPORT_STRUCT = v5;
  }
  library_handle = (API_IMPORT_STRUCT->mw_LoadLibraryA2)(library_name);
  output->lib_handle = library_handle;
  return test_exporting_library(library_handle);
}
```

Below is the list of all libraries used by the malware.

```
kernel32.dll, wininet.dll, advapi32.dll, ole32.dll, rpcrt4.dll, shell32.dll,
bcrypt.dll, crypt32.dll, dnsapi.dll, netapi32.dll, shlwapi.dll, user32.dll,
ktmw32.dll
```

The **LIBRARY_STRUCT** structures corresponding to these are pushed into a global list in the order below.

```
struct LIBRARY_STRUCT_LIST
{
    LIBRARY_STRUCT *lib_struct_kernel32;
    LIBRARY_STRUCT *lib_struct_wininet;
    LIBRARY_STRUCT *lib_struct_advapi32;
    LIBRARY_STRUCT *lib_struct_ole32;
    LIBRARY_STRUCT *lib_struct_rpcrt4;
    LIBRARY_STRUCT *lib_struct_shell32;
    LIBRARY_STRUCT *lib_struct_bcrypt;
    LIBRARY_STRUCT *lib_struct_crypt32;
    LIBRARY_STRUCT *lib_struct_dnsapi;
    LIBRARY_STRUCT *lib_struct_netapi32;
    LIBRARY_STRUCT *lib_struct_shlwapi;
    LIBRARY_STRUCT *lib_struct_user32;
    LIBRARY_STRUCT *lib_struct_ktmw32;
};
```

```
v2 = w_HeapAlloc_16_bytes();
set_up_lib_struct_kernel32(v2);
LIBRARY_STRUCT_LIST->lib_struct_kernel32 = v2;
v3 = w_HeapAlloc_16_bytes();
set_up_lib_struct_wininet(v3);
LIBRARY_STRUCT_LIST->lib_struct_wininet = v3;
v4 = w_HeapAlloc_16_bytes();
set_up_lib_struct_advapi32(v4);
LIBRARY_STRUCT_LIST->lib_struct_advapi32 = v4;
v5 = w_HeapAlloc_16_bytes();
set_up_lib_struct_ole32(v5);
LIBRARY_STRUCT_LIST->lib_struct_ole32 = v5;
v6 = w_HeapAlloc_16_bytes();
set_up_lib_struct_rpcrt4(v6);
LIBRARY_STRUCT_LIST->lib_struct_rpcrt4 = v6;
v7 = w_HeapAlloc_16_bytes();
set_up_lib_struct_shell32(v7);
LIBRARY_STRUCT_LIST->lib_struct_shell32 = v7;
v8 = w_HeapAlloc_16_bytes();
set_up_lib_struct_bcrypt(v8);
LIBRARY_STRUCT_LIST->lib_struct_bcrypt = v8;
v9 = w_HeapAlloc_16_bytes();
set_up_lib_struct_crypt32(v9);
LIBRARY_STRUCT_LIST->lib_struct_crypt32 = v9;
v10 = w_HeapAlloc_16_bytes();
set_up_lib_struct_dnsapi(v10);
LIBRARY_STRUCT_LIST->lib_struct_dnsapi = v10;
v11 = w_HeapAlloc_16_bytes();
set_up_lib_struct_netapi32(v11);
LIBRARY_STRUCT_LIST->lib_struct_netapi32 = v11;
v12 = w_HeapAlloc_16_bytes();
set_up_lib_struct_shlwapi(v12);
```

After this global list of **LIBRARY_STRUCT** is populated, an API can be called from a function taking in its corresponding library's **LIBRARY_STRUCT** structure and its parameters.

This function resolves the API name from a stack string, retrieves the API's address using the **get_API_addr** function from the library structure, and calls the API with its parameters.

```
__int64 __fastcall w_Sleep(LIBRARY_STRUCT *kernel32_lib_struct, unsigned int dwMilliseconds)
{
  __int64 i; // rcx
  __int64 (__fastcall *Sleep)(_QWORD); // rax
  char Sleep_str[16]; // [rsp+28h] [rbp-10h] BYREF

  strcpy(Sleep_str, "r-,,dT");
  for ( i = 0i64; i != 6; ++i )
    Sleep_str[i] = (7 * (Sleep_str[i] - 0x54) % 0x7F + 0x7F) % 0x7F;// "Sleep"
  Sleep = w_get_API_addr(kernel32_lib_struct, Sleep_str);
  return Sleep(dwMilliseconds);
}
```

```
v96 = 0i64;
do
{
  v97 = lpString[v96++];
  lpMemc = v97;
  v98 = GetProcessHeap();
  HeapFree(v98, 0, lpMemc);
}
while ( v96 ≠ 0x11 );
w_Sleep(LIB_STRUCT_ARR→lib_struct_kernel32, 3000u);
```

The way the wrapper function is setup to call the actual API is really intuitive, making the code simple to understand through static analysis. However, it’s a bit more difficult to automate the process since there is no API hashing involved.

For my analysis, I just manually decode the stack strings in my debugger and rename the wrapper function accordingly.

At this point, we have fully unpacked BAZARLOADER and understood how the malware obfuscates its strings and APIs to make analysis harder.

In the next blog post, we will fully analyze how the loader downloads and launches a Cobalt Strike beacon from its C2 servers!

1 Comment

Comments are closed.