Статья BAZARLOADER: распаковка зараженного файла ISO

xss.is/threads/68903

BAZARLOADER (также известный как BAZARBACKDOOR) — это загрузчик для Windows, который распространяется через вложения в фишинговых письмах. Во время заражения конечная полезная нагрузка загрузчика обычно загружает и запускает маячок кобальта, чтобы предоставить злоумышленникам удаленный доступ, что во многих случаях приводит к развертыванию программы-вымогателя на компьютере жертвы.

В этом начальном посте мы раскроем различные этапы заражения BAZARLOADER, которое приходит в виде файла образа оптического диска (ISO). Мы также углубимся в методы обфускации, используемые основной полезной нагрузкой BAZARLOADER.

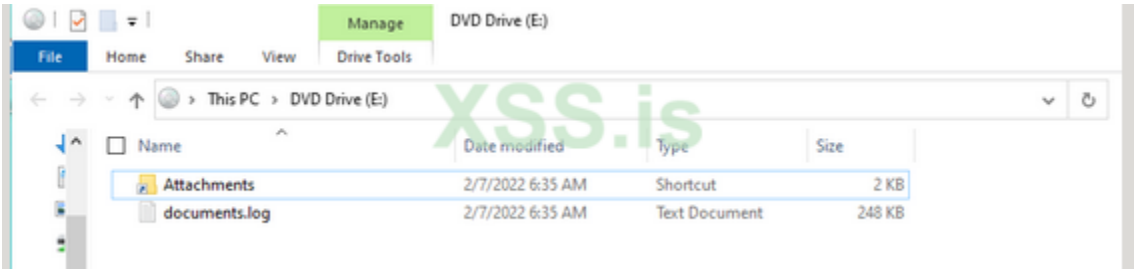
Чтобы продолжить, вы можете получить образец, а также файлы PCAP для него на Malware-Traffic-Analysis.net. (https://www.malware-traffic-analysis.net/2022/02/07/index.html)

SHA256:
0900b4eb02bdcaefd21df169d21794c8c70bfbc68b2f0612861fcabc82f28149

Шаг 1: Монтирование ISO-файла и извлечение исполняемого файла этапа № 1

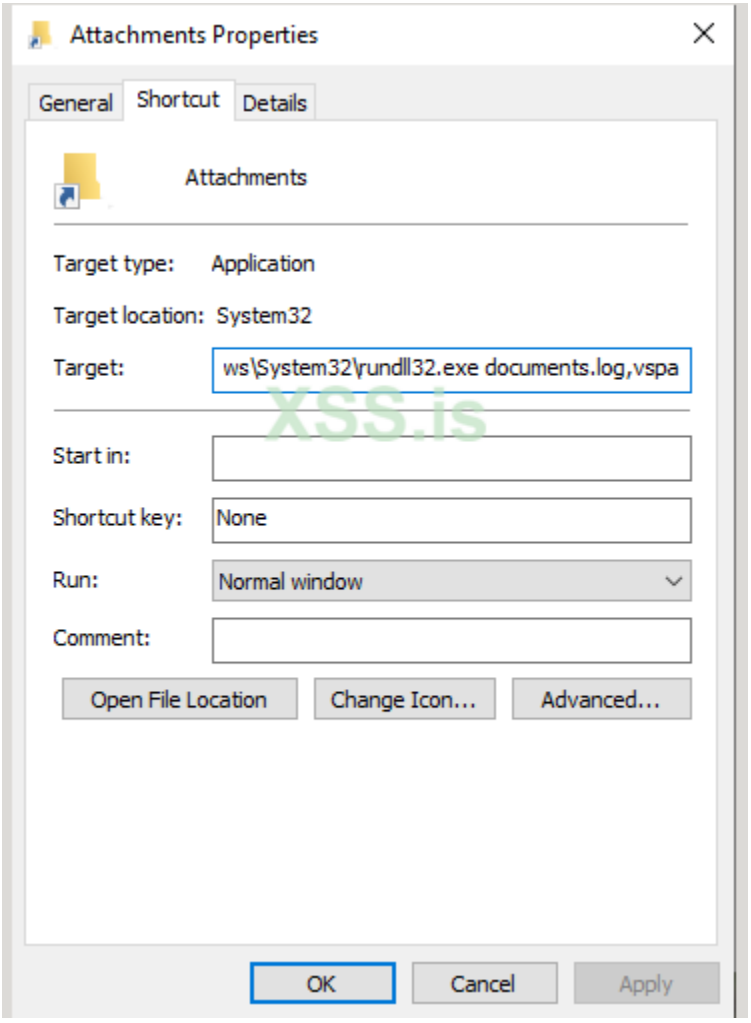
Недавние образцы BAZARLOADER приходят по электронной почте, содержащей ссылки OneDrive для загрузки файла ISO, чтобы избежать обнаружения, поскольку большинство AV, как правило, игнорируют этот конкретный тип файла. В Windows 7 и более поздних версиях функции монтирования интегрированы в проводник Windows, и мы можем монтировать любой файл ISO как виртуальный диск, дважды щелкнув его.

Когда мы монтируем вредоносный файл ISO, мы видим, что в системе подключен диск, который содержит файл ярлыка с именем "**Attachments.lnk**" и скрытый файл с именем "**documents.log**".



Файл ярлыка должен быть запущен жертвой, чтобы начать цепочку заражения. Мы можем быстро извлечь фактическую команду, выполняемую этим ярлыком, из окна его **свойств**.

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



Как только жертва дважды щелкает файл ярлыка, команда запускает программу Windows **rundll32.exe** для запуска файла "**documents.log**". Это дает нам понять, что запускаемый файл является DLL-файлом, а точкой входа является его функция экспорта **vspa**.

Шаг 2: Извлечение шелл-кода второго этапа

Быстро взглянув в IDA, мы можем сказать, что извлеченная DLL упакована, поскольку она имеет только несколько функций и очень подозрительно выглядящий буфер символов ASCII в пользовательском разделе .odata.

```
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 'b0fcc2434aeae404958ac30725dde2d46c622acf094f3cf91d1749b1ec6650c'
odata:000000018001D000                 db '3f2483e0cf8d5e9be621dce79c5485fae5284e1dc89d11a92bbada73f85452fe7e'
odata:000000018001D000                 db '0f80b1949ce5b5a77bae11b9714c7eda9a4289c4f41a0e75ff7688f201ac670e5'
odata:000000018001D000                 db '822582d189e64d9ab0e1a4306fecfcea8f4197751d09e9a3c7ae2a5a5777d785'
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'
```

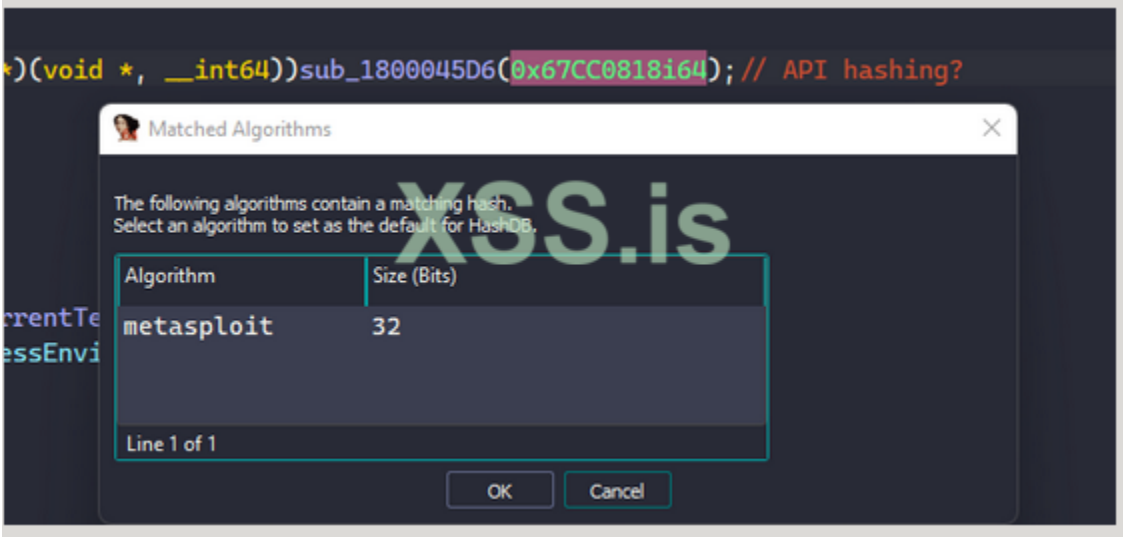
Имея это в виду, мы просто проведем небольшой статический анализ, чтобы определить, где мы можем сдампить следующий стейдж.

В первой функции экспорта vspa мы видим, что **sub_1800045D6** принимает DWORD в качестве параметра. Эта функция возвращает переменную, содержащую адрес функции, которая позже вызывается в коде.

```
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;
  }
```

На данный момент мы можем с уверенностью предположить, что **sub_1800045D6** — это функция разрешения API, а параметр, который она принимает, — это хэш имени API. Поскольку это все еще фаза распаковки, мы не будем слишком углубляться в анализ этой функции.

Вместо этого я просто воспользуюсь подключаемым модулем **OALabs HashDB** IDA для быстрого обратного поиска используемого алгоритма хеширования из хеша. Результат показывает, что хэш соответствует имени API, хэшированному с помощью алгоритма хэширования Metasploit (https://github.com/OALabs/hashdb/bl...16663ac13f02bf1aaf37/algorithms/metasploit.py) **ROR13.**



После определения алгоритма хэширования мы можем использовать **HashDB** для быстрого поиска API, которые разрешаются этой функцией. Становится ясно, что эта функция разрешает API **RtlAllocateHeap**, вызывает его для выделения буфера кучи и записывает в него закодированные данные ASCII.

```
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;
  }
}
```

С этого момента мы можем предположить, что упаковщик будет декодировать этот буфер и запускать его где-то позже в коде. Если мы перейдем к концу экспорта **vspa**, мы увидим инструкцию **вызова** переменной, которая не возвращается из

разрешающей функции API, поэтому потенциально это может быть наш главный прыжок.

Последняя функция для изменения этой переменной **v19 — sub_180003FE6**, поэтому мы можем быстро взглянуть на нее.

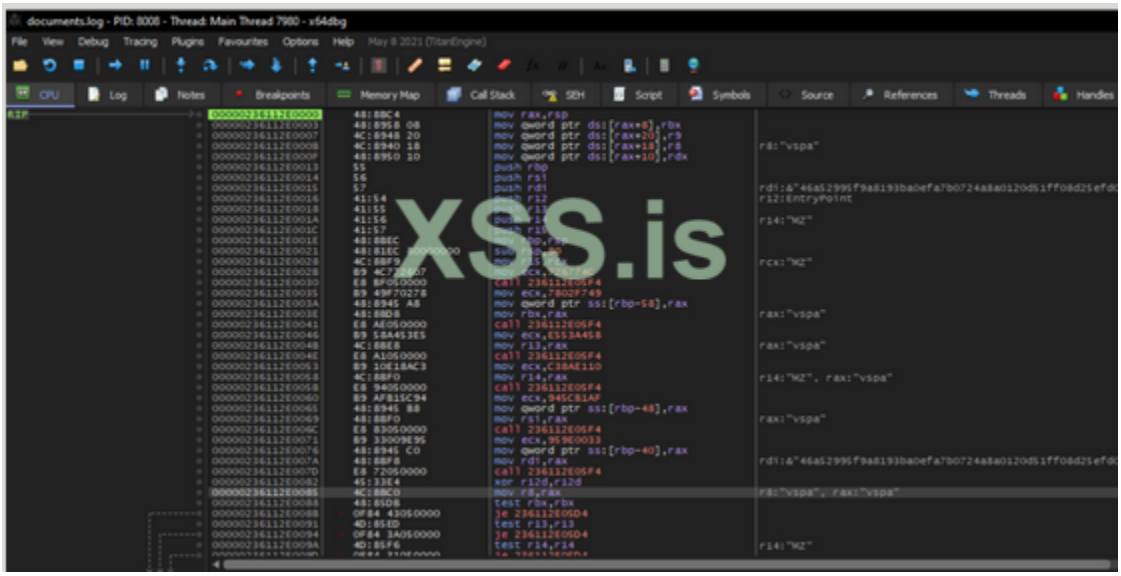
Оказывается, функция **sub_180003FE6** просто разрешает и вызывает **NtMapViewOfSection** для сопоставления представления раздела с виртуальным адресным пространством и записывает базовый адрес представления в переменную **v19**. Затем он просто выполняет **qmemcpy** для копирования данных из второй переменной в возвращенный виртуальный базовый адрес.

```
__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;
}
```

Это говорит нам о двух вещах. Во-первых, наше предположение, что переменная **v19** будет содержать адрес исполняемого кода, верно. Во-вторых, мы знаем, что исполняемый код является шеллкодом, поскольку данные отображаются и выполняются непосредственно по смещению 0 от того места, где они были записаны.

Отсюда мы можем настроить **x64dbg**, выполнить файл DLL при экспорте **vspa** и прервать выполнение инструкции call. После входа в функцию мы окажемся во главе шеллкода.



Теперь мы можем сдампить этот буфер виртуальной памяти, чтобы получить шелл-код второй стадии для следующего шага распаковки.

Шаг 3. Извлечение последней полезной нагрузки BAZARLOADER

Когда мы исследуем шелл-код в IDA, мы можем быстро использовать тот же прием с **HashDB**, описанный выше, чтобы увидеть, что шелл-код также выполняет хеширование API с **ROR13** Metasploit.

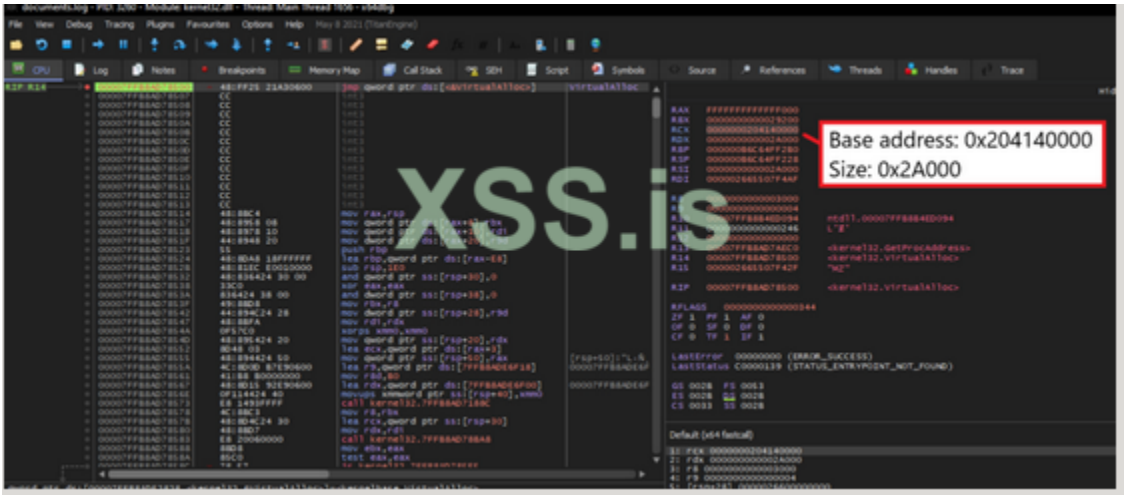
```
__int64 __fastcall sub_0(__int64 a1, __int64 a2, char *a3, _QWORD *a4)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  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;
```

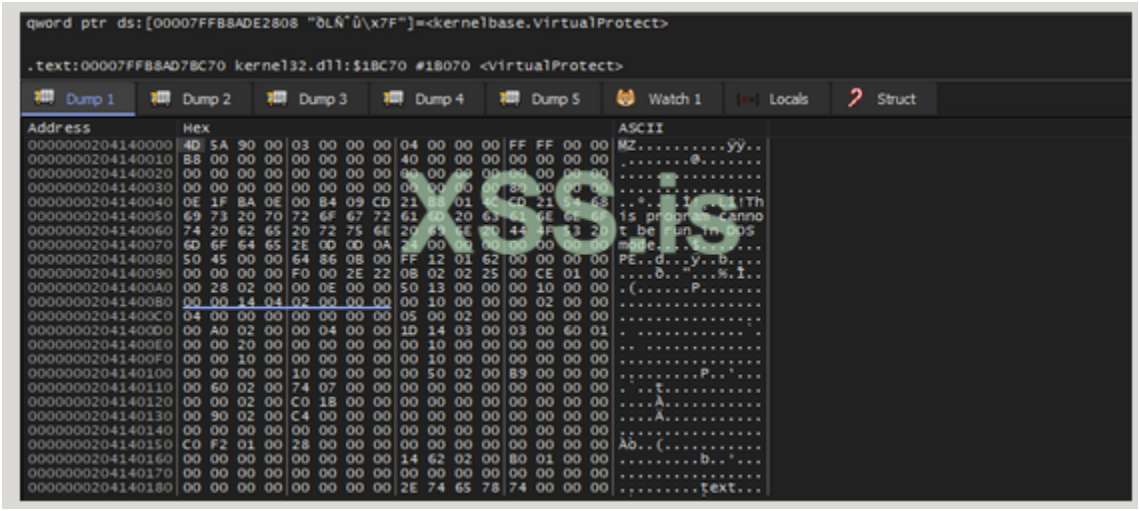

В указанной выше точке входа шелл-код разрешает набор функций, которые он будет вызывать, в первую очередь **VirtualAlloc** и **VirtualProtect**. Эти две функции обычно используются упаковщиками для выделения виртуальной памяти для декодирования и записи исполняемого файла следующего этапа перед его запуском.

Имея это в виду, нашим следующим шагом должна быть отладка шелл-кода и установка точек останова на этих двух вызовах API. Мы можем выбрать, где мы находимся после дампа в **x64dbg** на **шаге 2**, или мы можем запустить шелл-код непосредственно в нашем отладчике, используя BlobRunner (https://github.com/OALabs/BlobRunner) OALabs или аналогичный пусковой механизм шелл-кода.

Наше первое обращение к **VirtualAlloc** — это вызов выделения буфера виртуальной памяти по виртуальному адресу 0x204140000 размером 0x2A000 байт.



Мы можем запускать до тех пор, пока **VirtualAlloc** не вернется и не начнет мониторинг памяти по адресу 0x204140000. После запуска до следующего вызова **VirtualProtect** мы видим, что в эту область памяти был записан допустимый исполняемый файл PE.



Наконец, мы можем выгрузить эту область памяти в файл, чтобы извлечь полезную нагрузку BAZARLOADER.

Шаг 4: Обфускация строк BAZARLOADER

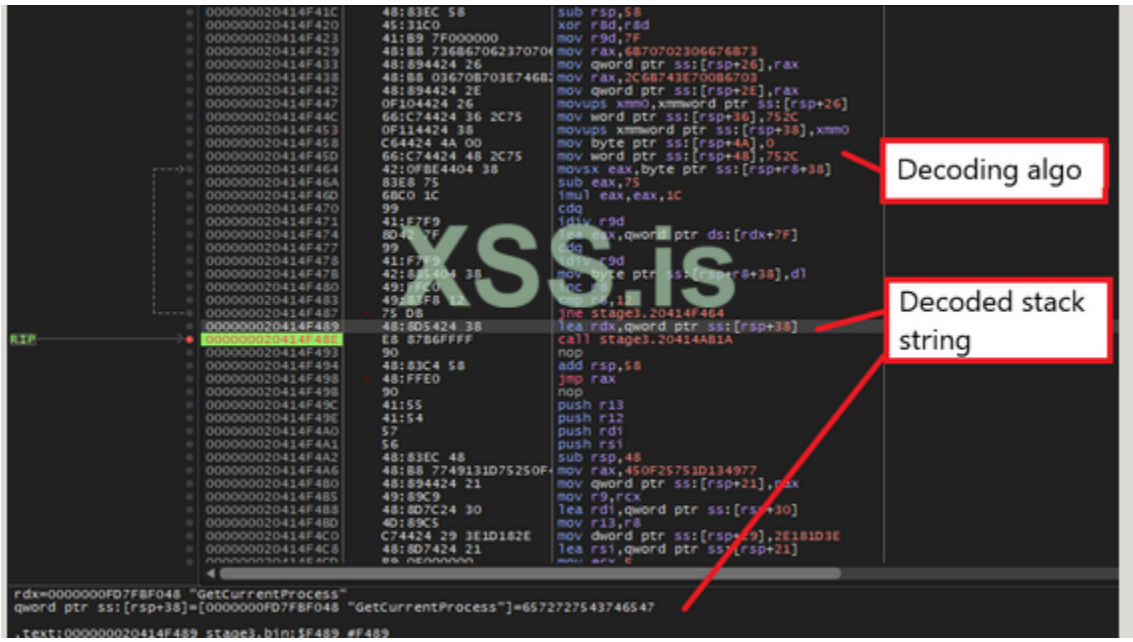
Поскольку мы начинаем выполнять статический анализ BAZARLOADER, крайне важно определить методы запутывания, которые использует вредоносное ПО.

Одним из таких методов является обфускация строк, когда вредоносное ПО использует закодированные строки стека, чтобы скрыть их от статического анализа.

```
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;
}
```

Как показано, типичная закодированная строка помещается в стек и динамически декодируется с использованием некоторых операций умножения, вычитания и модуля.

Существуют разные способы разрешения этих строк стека, например, написание сценариев IDAPython, эмуляция (https://github.com/mandiant/flare-emu) или просто запуск программы в отладчике и сброс строк стека после их разрешения.



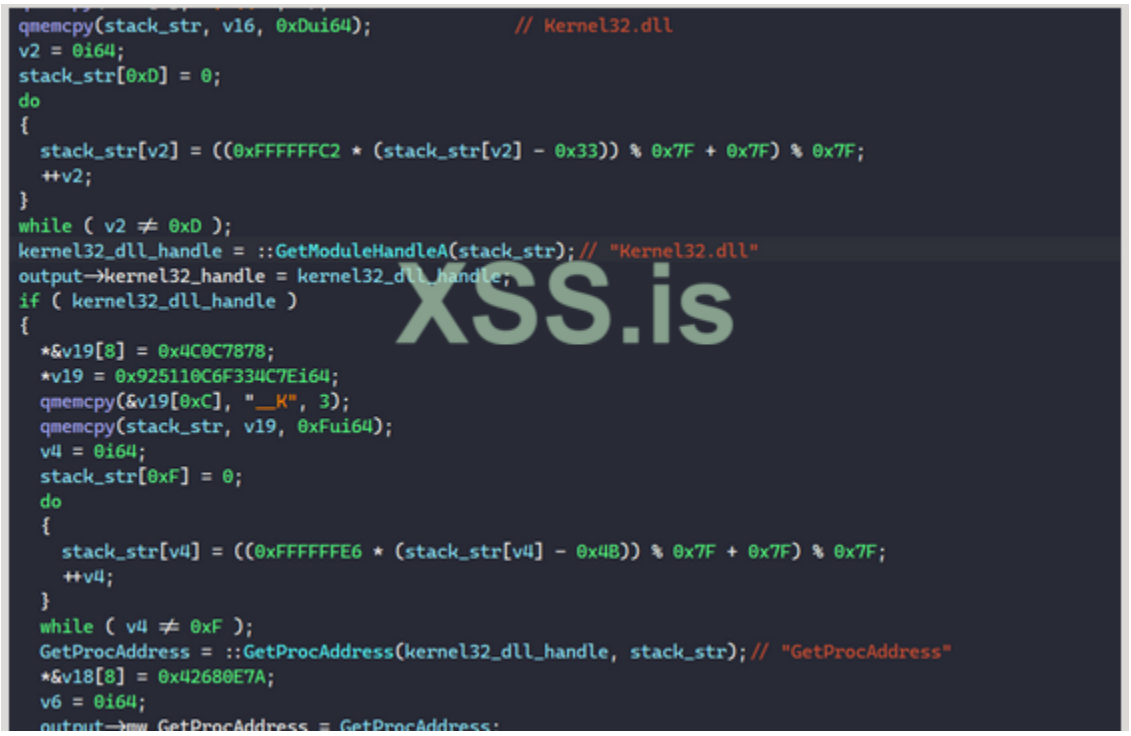
Шаг 5: Оббфускация API BAZARLOADER

BAZARLOADER запутывает большинство своих вызовов API с помощью нескольких структур, которые он создает в функции **DllEntryPoint**.

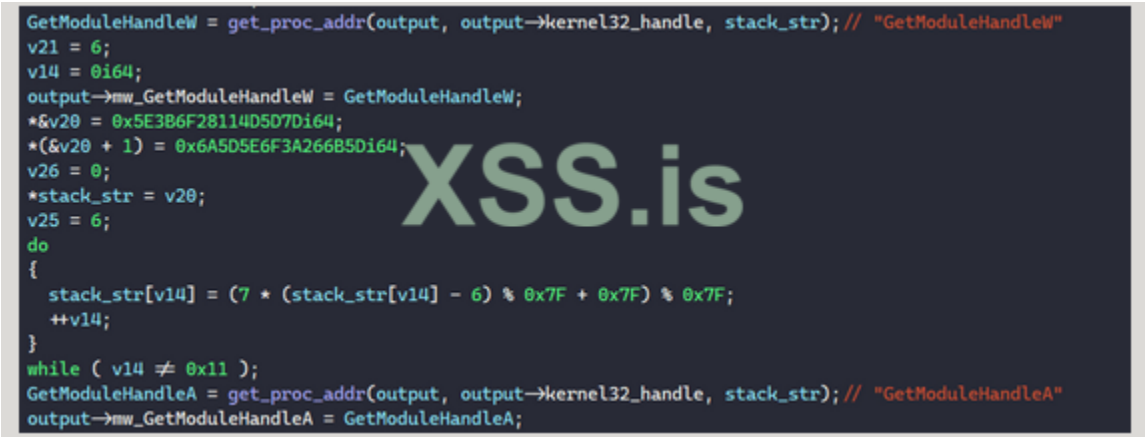
Сначала вредоносная программа заполняет следующую структуру, содержащую дескриптор **Kernel32.dll** и адреса API, необходимые для загрузки библиотек и получения их API-адресов.

Он вызывает **GetModuleHandle** для получения дескриптора **Kernel32.dll**, вызывает **GetProcAddress** для получения адреса API **GetProcAddress** и записывает их в структуру.

```
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;
};
```



Используя поле **GetProcAddress** API структуры, BAZARLOADER извлекает остальные необходимые API для заполнения других полей в структуре. Эта структура **API_IMPORT_STRUCT** позже будет использоваться для импорта API других библиотек.



Затем для каждой импортируемой библиотеки BAZARLOADER заполняет следующую структуру **LIBRARY_STRUCT**, содержащую набор функций для взаимодействия с библиотекой и дескриптором библиотеки.

```
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;
};
```

Первые 2 функции в структуре **LIB_FUNCS** просто вызывают **FreeLibrary API** из глобального **API_IMPORT_STRUCT**, чтобы освободить библиотечный модуль.

Третья функция вызывает **GetProcAddress** из поля **API_IMPORT_STRUCT**, чтобы получить адрес API, экспортированного из этой конкретной библиотеки.

```
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);
}
```

Чтобы начать заполнение каждой структуры **LIBRARY_STRUCT**, BAZARLOADER декодирует имя библиотеки из строки стека и заполняет ее соответствующим набором функций и дескриптором библиотеки, полученным при вызове **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);
}
```

Ниже приведен список всех библиотек, используемых вредоносной программой.

```
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
```

Соответствующие им структуры **LIBRARY_STRUCT** помещаются в глобальный список в порядке, указанном ниже.

После того, как этот глобальный список **LIBRARY_STRUCT** заполнен, API может быть вызван из функции, принимающей структуру **LIBRARY_STRUCT** соответствующей библиотеки и ее параметры.

Эта функция разрешает имя API из строки стека, извлекает адрес API с помощью функции **get_API_addr** из структуры библиотеки и вызывает API с его параметрами.

```
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);
```



```
__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);
```

Способ настройки функции-оболочки для вызова фактического API действительно интуитивно понятен, что упрощает понимание кода с помощью статического анализа. Однако автоматизировать этот процесс немного сложнее, поскольку не используется хеширование API.

Для моего анализа я просто вручную декодирую строки стека в своем отладчике и соответствующим образом переименовываю функцию-оболочку.

На данный момент мы полностью распаковали BAZARLOADER и поняли, как вредоносное ПО запутывает свои строки и API, чтобы затруднить анализ.

В следующем посте мы полностью проанализируем, как загрузчик загружает и запускает маячок Cobalt Strike со своих серверов C2!

Переведено специально для XSS.IS
Автор перевода: yashechka
Источник: https://www.offset.net/reverse-engineering/bazarloader-iso-file-infection/