Статья Разработка вредоносного ПО. Часть 8 - инъекция COFF и выполнение в памяти

Введение

Это восьмой пост из серии, посвященной разработке вредоносного ПО. В этой серии статей мы исследуем и попытаемся реализовать несколько методов, используемых вредоносными приложениями для выполнения кода, укрытия от защиты и персистентности.

На этот раз мы реализуем загрузчик объектного файла COFF, который похож на функцию BOF (объектный файл маячка) в Cobalt Strike https://www.cobaltstrike.com/help-beacon-object-files .

TrustedSec также решила эту проблему https://www.trustedsec.com/blog/coffloader-building-your-own-in-memory-loader-or-how-to-run-bofs/

Код находится в моем репозитории GitHub https://github.com/0xpat/COFFInjector .

Примечание. Здесь мы работаем с 64-битным кодом.

Компиляция кода C

Создание исполняемого файла из исходного кода C/C ++ - это трехэтапный процесс:

1. Предобработка - интерпретация директив прекомпилятора (объединение #include файлов, замена #define идентификаторов). Прекомпилятор в основном заменяет текст в исходном коде для создания единицы перевода.

2. Компиляция (которую мы подробно рассматривали в части 6). Компилятор генерирует ассемблерный код из исходного кода и создает объектный файл.

3. Линковка - объединение объектных файлов и необходимых библиотек в окончательный исполняемый файл (который также может быть DLL).

Исполняемый файл может быть либо изначально выполнен загрузчиком ОС, либо инжектирован в память (например, с помощью процесса или любого другого применимого метода).

Но что, если бы мы могли выполнять объектные файлы? На самом деле это возможно, поскольку эти файлы содержат реальный машинный код, который нам нужен.

Объектные файлы COFF

Общий формат объектного файла - это формат исполняемого кода, созданный в Unix. На основе этого Microsoft создала свой вариант формата COFF и PE. [Документация Microsoft содержит множество информации о форматах файлов COFF и PE.

Объектные файлы, созданные компилятором Visual Studio, используют формат COFF. Такой объектный файл (с расширением .obj) содержит:

- заголовок (с информацией об архитектуре, отметкой времени, количеством секций и символов и др.),

- секции (с ассемблерным кодом, отладочной информацией, директивами компоновщика, информацией об исключениях, статическими данными и т. д.),

- таблица символов (например, функций и переменных) с информацией об их расположении.

Секции могут содержать информацию о перемещении, которая указывает, как данные раздела должны быть изменены компоновщиком, а затем во время загрузки в память. Например, секция .text содержит информацию о том, какие части кода следует заменить и на что они должны ссылаться в памяти. Подробнее об этом позже.

Нам нужно просмотреть содержимое файла COFF и извлечь код вместе с данными перемещения и выполнить перемещения. Окончательный код (с примененными перемещениями) можно выполнить, просто вызвав его как функцию (((void (*) ()) (code)) ()) или с помощью (например, CreateThread).

Пример объектного файла

Рассмотрим очень простое консольное приложение:

C:

```
int main()
{
    MessageBoxA(NULL, "Content", "Title", NULL);
    return 0;
}
```

Функция MessageBoxA находится в user32.dll - нам нужно намекнуть об этом компоновщику.

Обычно файлы .lib представляют собой статические библиотеки, содержащие код (фактически объектные файлы), которые могут быть статически связаны с исполняемым файлом. Однако при динамической компоновке компоновщик использует специальные файлы .lib, которые указывают на соответствующие динамические библиотеки - эта информация используется компоновщиком для построения таблицы адресов импорта исполняемого файла.

Это можно сделать, изменив параметры проекта в Visual Studio или используя следующую директиву:

#pragma comment (lib, "user32.lib")

Я отключил оптимизацию компилятора (/Od) для этого фрагмента кода. Включение оптимизации привело к различному расположению данных в объектном файле и вызвало проблемы с моим загрузчиком COFF. Потребуются дальнейшие испытания.

Компиляция с использованием компилятора MSVC (cl.exe) создает объектный файл (с расширением .obj). Мы можем проанализировать его содержимое с помощью инструмента dumpbin, поставляемого с MSVC. Давайте посмотрим на результат работы инструмента.

Директивы (секции .drectve)

Вот директивы компоновщика, наиболее важная информация о том, какие библиотеки следует просматривать для внешних функций.

```
SECTION HEADER #1
.drectve name
       0 physical address
       0 virtual address
      AB size of raw data
     12C file pointer to raw data (0000012C to 000001D6)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
  100A00 flags
         Info
         Remove
         1 byte align

RAW DATA #1
  00000000: 20 20 20 2F 44 45 46 41 55 4C 54 4C 49 42 3A 22     /DEFAULTLIB:"
  00000010: 75 75 69 64 2E 6C 69 62 22 20 2F 44 45 46 41 55  uuid.lib" /DEFAU
  00000020: 4C 54 4C 49 42 3A 22 75 75 69 64 2E 6C 69 62 22  LTLIB:"uuid.lib"
  00000030: 20 2F 46 41 49 4C 49 46 4D 49 53 4D 41 54 43 48   /FAILIFMISMATCH
  00000040: 3A 22 5F 43 52 54 5F 53 54 44 49 4F 5F 49 53 4F  :"_CRT_STDIO_ISO
  00000050: 5F 57 49 44 45 5F 53 50 45 43 49 46 49 45 52 53  _WIDE_SPECIFIERS
  00000060: 3D 30 22 20 2F 44 45 46 41 55 4C 54 4C 49 42 3A  =0" /DEFAULTLIB:
  00000070: 22 6B 65 72 6E 65 6C 33 32 2E 6C 69 62 22 20 2F  "kernel32.lib" /
  00000080: 44 45 46 41 55 4C 54 4C 49 42 3A 22 4C 49 42 43  DEFAULTLIB:"LIBC
  00000090: 4D 54 22 20 2F 44 45 46 41 55 4C 54 4C 49 42 3A  MT" /DEFAULTLIB:
  000000A0: 22 4F 4C 44 4E 41 4D 45 53 22 20                 "OLDNAMES"

   Linker Directives
   -----------------
   /DEFAULTLIB:uuid.lib
   /DEFAULTLIB:uuid.lib
   /FAILIFMISMATCH:_CRT_STDIO_ISO_WIDE_SPECIFIERS=0
   /DEFAULTLIB:kernel32.lib
   /DEFAULTLIB:LIBCMT
   /DEFAULTLIB:OLDNAMES
```


Данные только для чтения (раздел .rdata)

Это статически инициализированные данные, например строковые литералы.

```
SECTION HEADER #3
  .rdata name
       0 physical address
       0 virtual address
      10 size of raw data
     2C0 file pointer to raw data (000002C0 to 000002CF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
40400040 flags
         Initialized Data
         8 byte align
         Read Only

RAW DATA #3
  00000000: 54 69 74 6C 65 00 00 00 43 6F 6E 74 65 6E 74 00  Title...Content.
```


Исполняемый код (раздел .text)

Это собственно ассемблерный код. В моем примере компилятор MSVC назвал этот раздел .text$mn.

```
SECTION HEADER #4
.text$mn name
       0 physical address
       0 virtual address
      24 size of raw data
     2D0 file pointer to raw data (000002D0 to 000002F3)
     2F4 file pointer to relocation table
       0 file pointer to line numbers
       3 number of relocations
       0 number of line numbers
60501020 flags
         Code
         COMDAT; sym= main
         16 byte align
         Execute Read

RAW DATA #4
  00000000: 48 83 EC 28 45 33 C9 4C 8D 05 00 00 00 00 48 8D  H.ì(E3ÉL......H.
  00000010: 15 00 00 00 00 33 C9 FF 15 00 00 00 00 33 C0 48  .....3Éÿ.....3ÀH
  00000020: 83 C4 28 C3                                      .Ä(Ã

RELOCATIONS #4

                                                 Symbol    Symbol
 Offset    Type              Applied To         Index     Name
 --------  ----------------  -----------------  --------  ------
 0000000A  REL32                     00000000         8  $SG89438
 00000011  REL32                     00000000         9  $SG89439
 00000019  REL32                     00000000         C  __imp_MessageBoxA
```


Здесь все становится интереснее, давайте дизассемблируем этот код:

```
main:
  0000000000000000: 48 83 EC 28        sub         rsp,28h
  0000000000000004: 45 33 C9           xor         r9d,r9d
  0000000000000007: 4C 8D 05 00 00 00  lea         r8,[$SG89438]
                    00
  000000000000000E: 48 8D 15 00 00 00  lea         rdx,[$SG89439]
                    00
  0000000000000015: 33 C9              xor         ecx,ecx
  0000000000000017: FF 15 00 00 00 00  call        qword ptr [__imp_MessageBoxA]
  000000000000001D: 33 C0              xor         eax,eax
  000000000000001F: 48 83 C4 28        add         rsp,28h
  0000000000000023: C3                 ret
```


Здесь мы видим, как аргументы MessageBoxA (int 0, char * "Content", char * "Title", int 0) передаются в соответствии с соглашением о вызовах x64 https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention .

Давайте посмотрим на третью инструкцию (расположенную по смещению 0x07 . При дизассемблировании с использованием, например, ShellNoob (https://github.com/reyammer/shellnoob , инструкция использует смещение 0x00000000

```
kali@kali:~$ shellnoob -i --to-asm --64 --intel
opcode_to_asm selected (type "quit" or ^C to end)
>> 4C8D0500000000
b'4C8D0500000000' -> Disassembly of section .text:

0000000000000000 <.text>:
   0:   4c 8d 05 00 00 00 00    lea    r8,[rip+0x0]        # 0x7
```


Итак, как дизассемблер dumpbin знает, что функция должна ссылаться на символ? Вот где в игру вступает релокация. Под необработанными данными раздела .text мы можем увидеть информацию о перемещении. Например, первая запись таблицы перемещений говорит, что 0x00000000 байтов по смещению 0x0A (который является вторым операндом инструкции lea) следует заменить фактическим (относительно RIP) адресом символа №. 8.

То же самое касается инструкции вызова по смещению 0x17 вместе с соответствующей записью релокации и символом. Однако здесь перемещение касается относительного адреса функции. Это верно - относительный адрес (операнд вызова) разыменовывается и вызывается сохраненное в нем значение (фактический адрес MessageBoxA).

Когда код загружается в системную память, загрузчик анализирует данные о перемещении и помещает функции и адреса данных в нужные места. Однако это происходит во время загрузки исполняемого файла PE. Мы хотим загрузить

объектный файл COFF, поэтому нам нужно проанализировать его и выполнить перемещения в памяти.

Таблица символов

Эта таблица содержит символы, такие как статические переменные или внешние функции.

```
COFF SYMBOL TABLE
000 010571B7 ABS    notype       Static       | @comp.id
001 80000090 ABS    notype       Static       | @feat.00
002 00000000 SECT1  notype       Static       | .drectve
    Section length   F8, #relocs    0, #linenums    0, checksum        0
004 00000000 SECT2  notype       Static       | .debug$S
    Section length   9C, #relocs    0, #linenums    0, checksum        0
006 00000000 SECT3  notype       Static       | .rdata
    Section length   10, #relocs    0, #linenums    0, checksum 1EFD1E42
008 00000000 SECT3  notype       Static       | $SG89438
009 00000008 SECT3  notype       Static       | $SG89439
00A 00000000 SECT4  notype       Static       | .text$mn
    Section length   24, #relocs    3, #linenums    0, checksum  62465EB, selection    1 (pick no duplicates)
00C 00000000 UNDEF  notype       External     | __imp_MessageBoxA
00D 00000000 SECT4  notype ()    External     | main
00E 00000000 SECT4  notype       Label        | $LN3
00F 00000000 SECT5  notype       Static       | .xdata
    Section length    8, #relocs    0, #linenums    0, checksum  FC539D1, selection    5 (pick associative Section 0x4)
011 00000000 SECT5  notype       Static       | $unwind$main
012 00000000 SECT6  notype       Static       | .pdata
    Section length    C, #relocs    3, #linenums    0, checksum 7D3C6CAC, selection    5 (pick associative Section 0x4)
014 00000000 SECT6  notype       Static       | $pdata$main
015 00000000 SECT7  notype       Static       | .chks64
    Section length   38, #relocs    0, #linenums    0, checksum        0
```


Таблицу символов немного сложно читать и понимать. Однако обычно поле "Значение" указывает смещение символа внутри раздела (описывается полем "SectionNumber ").

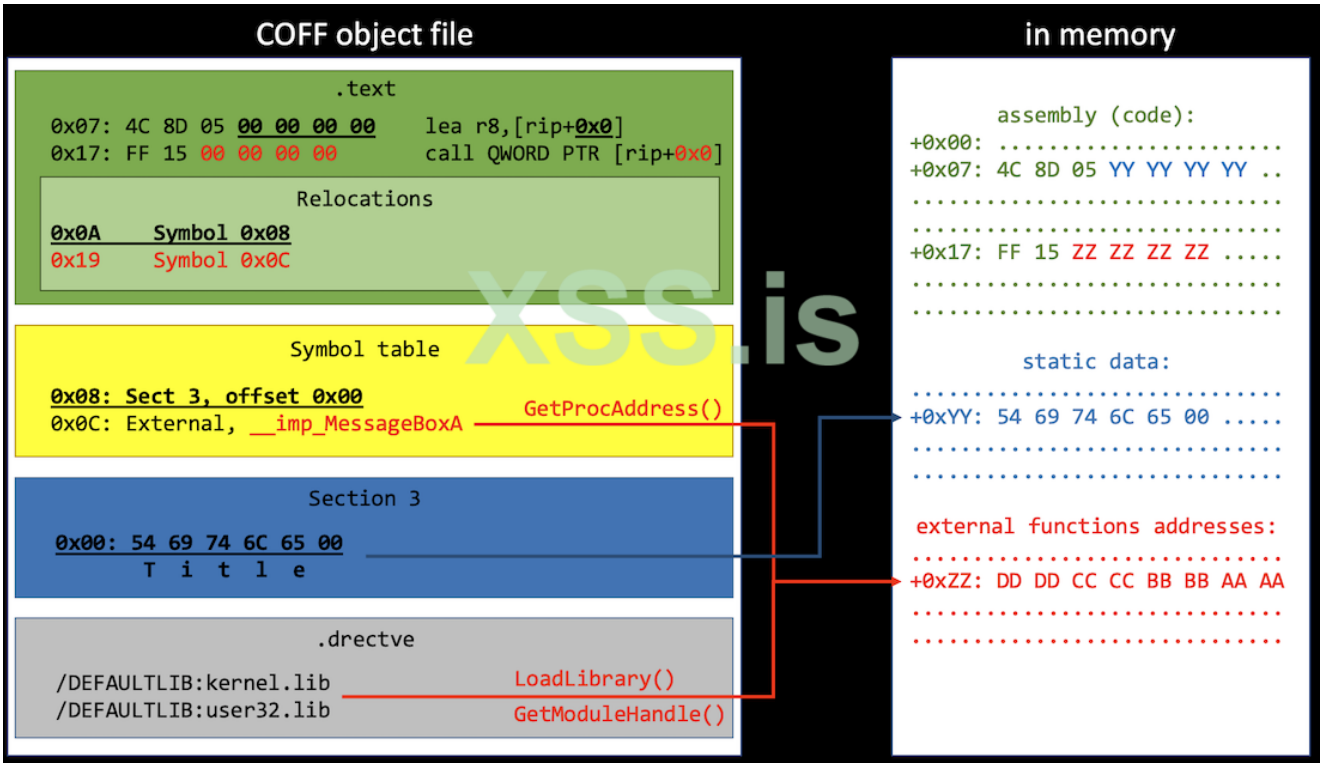
Кроме того, чтобы знать, где заканчиваются данные одного символа, нам нужно проверить смещение следующего символа в той же секции или общий размер секции.

Загрузчик объектных файлов

Чтобы внедрить и выполнить простой файл COFF, нам нужно прочитать раздел .text и заполнить все нули относительными адресами внешних функций и статических данных (т.е. переместить символы, указанные в этом разделе). Конечно, нам также нужно разместить эти символы где-нибудь в памяти, например, после кода.

Чтобы найти внешние функции, нам нужно будет просмотреть библиотеки, указанные в директивах компоновщика. Мы можем использовать функции LoadLibrary/GetModuleHandle/GetProcAddress или, например, просматривать PEB и InMemoryOrderModuleList (смотри Часть 4).

Схема ниже иллюстрирует эту концепцию:



Я использовал библиотеку COFFI https://github.com/serge1/COFFI для парсинга файлов COFF. Это отличная библиотека C++ только для заголовков, в которой есть все функции, необходимые для чтения данных из объектных файлов. COFFI использует некоторые структуры данных стандартной библиотеки C++, такие как строки, векторы и т.д. , как и мой код.

Мой алгоритм выглядит так:

1. Получите указатели на раздел .text и релокации, директивы, статические данные и таблицу символов.

2. Вычислите память, необходимую для кода + статических данных + указателей внешних функций (путем итерации всех перемещений .text).

3. Скопируйте код в память RW (X).

4. Скопируйте статические символы сразу после кода (размер каждого символа рассчитывается путем проверки смещения следующего в данном разделе).

5. При копировании статических символов выполнить перемещения (заменить нули в коде относительными адресами).

6. Разрешите все статические функции, просмотрев библиотеки,

указанные в директивах компоновщика (LoadLibrary, но не файлы библиотеки DLL), поместите адреса в память (сразу после статических данных; используйте GetProcAddress) и выполните перемещения. Имена функций WinAPI имеют префикс __imp_ в таблице символов COFF.

7. Вызовите место где начинается код (убедитесь, что память исполняемая - при необходимости используйте VirtualProtect).

Определение дополнительных API

BOF Cobalt Strike реализует набор функций, которые могут быть вызваны из кода внедренного объектного файла (также известного как Beacon API).Мы тоже можем это сделать.

В загруженном объектном файле может быть определена внутренняя функция, например:

C:

```
void COFF_API_Print(char* string)
{
    printf(string);
}
```

и добавлен как импорт в код объектного файла:

__declspec (dllimport) void COFF_API_Print (char * string);

Затем она должна быть обработана, как импорт WinAPI во время загрузки.

Возврат значения

При вызове внедренной основной функции из объектного файла мы можем получить доступ к возвращаемому значению от вызывающей стороны:

int returnValue = ((int (*) ()) code) ();

Предостережения

- Этот код PoC предполагает, что объектный файл содержит только одну функцию (основную) и не будет работать, если есть другие подпрограммы.

- Объектный файл скомпилирован без среды выполнения C, и в нем нет функций инициализации среды выполнения - точка входа - main. Также отключена оптимизация кода компилятором.

Резюме

Мы получили представление о формате объектного файла COFF, создаваемом компилятором MSVC. Поскольку эти файлы содержат всю информацию, необходимую для выполнения кода, они также могут быть инжектированы и выполнены в памяти, например, по каналу C&C. Это мощный метод, который, несомненно, создает проблемы для обнаружения вредоносного кода.

Код находится в моем репозитории на GitHub https://github.com/0xpat/COFFInjector

Переведено специально для **XSS.is**
Автор перевода: yashechka
Источник: https://0xpat.github.io/Malware_development_part_8/