原文地址:http://drops.wooyun.org/papers/8814

http://int0xcc.svbtle.com/a-guide-to-malware-binary-reconstruction

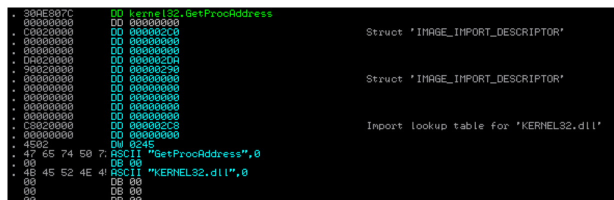
在分析恶意软件或对恶意软件进行脱壳的时候，我们经常会遇到重建PE文件的需求。现在大多数自动化的PE重建工具虽然很棒，但并不能针对每一种情况，有时候需要我们自己手动重建PE文件。在这篇博客中，我们将介绍一些重建PE文件的方法。

0x00 重建“stolen API code”的IAT表

“stolen API code”技术常被恶意软件用于阻挠逆向人员脱壳之后重建IAT表，从而达到反脱壳的效果。具体修复“stolen API code”后IAT表的方法在后面会介绍到，我们先了解一下IAT在PE（Portable Executable）文件里面的具体实现。

0x01 IAT基础知识

IAT(Import Address Table)是PE文件里面的一种结构，它包含了Windows loader加载动态链接库和导入API函数地址的信息。查看PE文件的时候你应该注意到IMAGE_OPTIONAL_HEADER结构里面的两个指针：一个指向IMAGE_IMPORT_DESCRIPTOR，另一个指向导入函数地址的数组。



函数可以通过函数名称或序号（API号）导入。

FirstThunk成员指向导入的API函数数组（也称为导入地址表）。

上图显示了一个kernel32.dll的导入函数GetProcAddress()的例子。

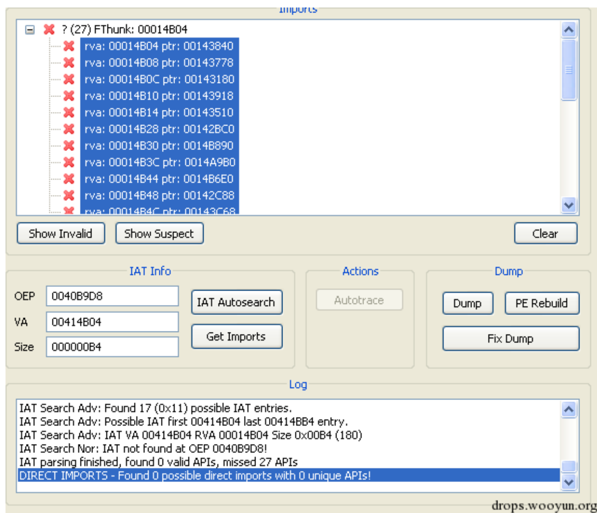
```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;            // 0 for terminating null import
        DWORD   OriginalFirstThunk;         // RVA to original unbound IAT (P
    } DUMMYUNIONNAME;
    DWORD   TimeDateStamp;                  // 0 if not bound,
                                            // -1 if bound, and real date\tim
                                            //     in IMAGE_DIRECTORY_ENTRY_B
                                            // O.W. date/time stamp of DLL bo

    DWORD   ForwarderChain;                 // -1 if no forwarders
    DWORD   Name;
    DWORD   FirstThunk;                     // RVA to IAT (if bound this IAT
} IMAGE_IMPORT_DESCRIPTOR;
typedef IMAGE_IMPORT_DESCRIPTOR UNALIGNED *PIMAGE_IMPORT_DESCRIPTOR;
```

drops.wooyun.org

加壳程序通常会破坏IAT表的原始形式，由壳代码自己解决函数导入而不是依靠Windows loader。因此脱壳后需要重建程序的IAT表，下面我们将使用Scylla v0.9.6b这个工具来重建IAT。

0x02 Stolen API code

一些加壳程序会使用“stolen code”技术防止逆向人员重建IAT表。“stolen code”重新把跳转到API函数的指令在某个内存区域被重新仿真。所以使用扫描器扫描这些导入函数的时候得到的是一些无效的API指针。



Disassembler

Address	Size	Opcodes	Instructions	Comment
00143840	02	8BFF	MOV EDI, EDI	
00143842	01	55	PUSH EBP	
00143843	02	8BEC	MOV EBP, ESP	
00143845	05	E9B5926D7C	JMP 0x7C81CAFF	-> 7C81CAFF = kernel32.dll
0014384A	02	0000	ADD [EAX], AL	

drops.wooyun.org

使用“stolen API code”技术的情况下，Scylla是无法自动重建IAT表的。我们需要写一个Scylla插件方便获得正确的偏移然后重建IAT表。

0x03 编写一个Scylla插件

Scylla插件的基本上是以DLL文件形式注入到目标进程。为此它提供构建插件所需的API接口，还有让Scylla插件方便嵌入到目标程序的接口。此外Scylla还提供了一个命名内存映射文件用于Scylla插件在目标程序中可以获取一些信息。

Scylla提供命名的文件映射用于目标DLL指向特定的内存区域。

这个内存映射文件名为“ScyllaPluginExchange”。

通过ScyllaPluginExchange可以获取到以下的信息：

```
17 typedef struct _UNRESOLVED_IMPORT {       // Scylla Plugin exchange format
18   DWORD_PTR ImportTableAddressPointer;  //in VA, address in IAT which points to an invalid api address
19   DWORD_PTR InvalidApiAddress;          //in VA, invalid api address that needs to be resolved
20 } UNRESOLVED_IMPORT, *PUNRESOLVED_IMPORT;
21
22 typedef struct _SCYLLA_EXCHANGE {
23   BYTE status; //return a status, default 0xFF == SCYLLA_STATUS_MAPPING_FAILED
24   DWORD_PTR imageBase; //image base
25   DWORD_PTR imageSize; //size of the image
26   DWORD_PTR numberOfUnresolvedImports; //number of unresolved imports in this structure
27   BYTE offsetUnresolvedImportsArray;
28 } SCYLLA_EXCHANGE, *PSCYLLA_EXCHANGE;
29
```

drops.wooyun.org

UNRESOLVED_IMPORT结构体通过SCYLLA_EXCHANGE.offsetUnresolvedImportsArray成员取得。

编写插件的第一步是利用Scylla提供的命名内存映射文件拿到SCYLLA_EXCHANGE结构体的基地址：

```
#!c++
BOOL getMappedView()
{
    hMapFile = OpenFileMappingA(FILE_MAP_ALL_ACCESS, 0, FILE_MAPPING_NAME); //open named file mapping object

    if (hMapFile == 0)
    {
        writeToLogFile("OpenFileMappingA failed\r\n");
        return FALSE;
    }

    // lpViewOfFile就是SCYLLA_EXCHANGE结构体的基地址
    lpViewOfFile = MapViewOfFile(hMapFile, FILE_MAP_ALL_ACCESS, 0, 0, 0); //map the view with full access
    if (lpViewOfFile == 0)
    {
        CloseHandle(hMapFile); //close mapping handle
        hMapFile = 0;
        writeToLogFile("MapViewOfFile failed\r\n");
        return FALSE;
    }
    return TRUE;
}
```

UNRESOLVED_IMPORT包含了一个未解决的导入函数列表。

```
#!c++
typedef struct _UNRESOLVED_IMPORT {       // Scylla Plugin exchange format
    DWORD_PTR ImportTableAddressPointer;  //in VA, address in IAT which points to an invalid api address
    DWORD_PTR InvalidApiAddress;          //in VA, invalid api address that needs to be resolved
} UNRESOLVED_IMPORT, *PUNRESOLVED_IMPORT;
```

ImportTableAddressPointer指针指向有效的API地址。InvalidApiAddress指针指向未决断的API函数地址，在本文例子中，这是一块动态分配的内存区域，那些被偷的代码（stolen code）就是在这里进行仿真。

Disassembler

Address	Size	Opcodes	Instructions	Comment
00143840	02	8BFF	MOV EDI, EDI	
00143842	01	55	PUSH EBP	
00143843	02	8BEC	MOV EBP, ESP	
00143845	05	E9B5926D7C	JMP 0x7C81CAFF	-> 7C81CAFF = kernel32.dll
0014384A	02	0000	ADD [EAX], AL	

drops.wooyun.org

可以看到我们需要计算每个ImportTableAddressPointer到jmp指令有多少个字节，然后取出JMP指令所跳转的目标地址减去这个字节数得到原来的API基地址：

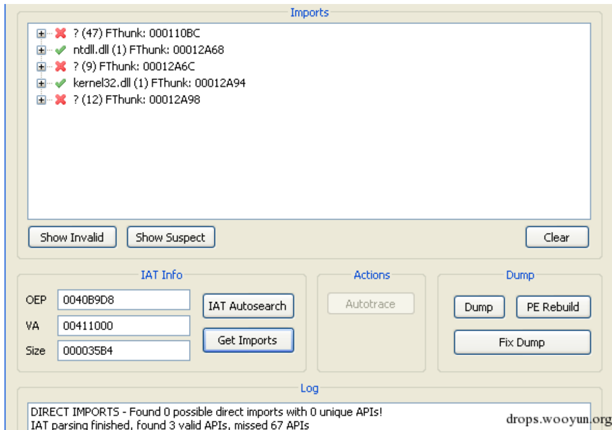
```
#!c++
    while (unresolvedImport->ImportTableAddressPointer != 0) //last element is a nulled struct
    {
        insDelta = 0;
        invalidApiAddress = unresolvedImport->InvalidApiAddress;
        sprintf(buffer, "API Address = 0x%p\t IAT Address = 0x%p\n",  invalidApiAddress, unresolvedImport->ImportTableAddressPointer);

        writeToLogFile(buffer);

        IATbase = unresolvedImport->InvalidApiAddress;
        for (j = 0; j <  COUNT_INS; j++)
        {
            memset(&inst, 0x00, sizeof(INSTRUCTION));

            i = get_instruction(&inst, IATbase, MODE_32);
            memset(buffer, 0x00, sizeof(buffer));
            get_instruction_string(&inst, FORMAT_ATT, 0, buffer, sizeof(buffer));
            if (strstr(buffer, "jmp"))
            {

                printf(" JUMP Dest = %d" , ( (unsigned int)strtol(strstr(buffer, "jmp") + 4 + 2, NULL, 16)));
                *(DWORD*)(unresolvedImport->ImportTableAddressPointer) =  ( (unsigned int)strtol(strstr(buffer, "jmp") + 4 + 2, NULL, 16) + IATbase ) - insDelta;
                unresolvedImport->InvalidApiAddress = ( (unsigned int)strtol(strstr(buffer, "jmp") + 4 + 2, NULL, 16) + IATbase ) - insDelta;
                break;
            }
            else
            {
                insDelta = insDelta + i;
            }

            IATbase = IATbase + i;
        }
        unresolvedImport++; //next pointer to struct
    }
```

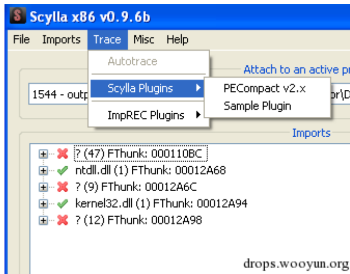
这段代码将遍历所有未决断的导入函数，并尝试定位到正确的API地址。

JMP指令的目标地址减去insDelta就可以得到最终的InvalidApiAddress地址：

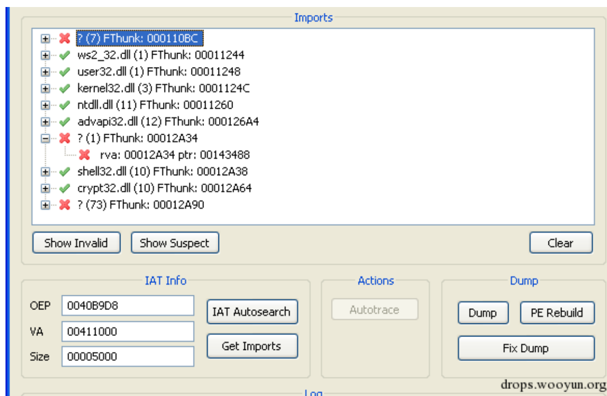
```
#!c++
unresolvedImport->InvalidApiAddress = ((unsigned int)strtol(strstr(buffer, "jmp”) + 4 + 2, NULL, 16) + IATbase) - insDelta;
```

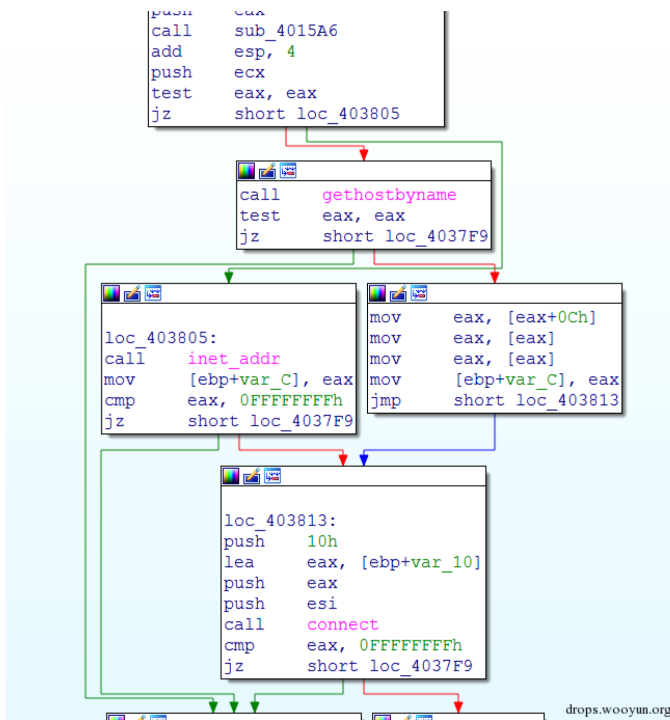
在修复整个IAT表之后可能还会有一些无效的导入地址，这些无效的导入地址需要手动把它们删除掉。





运行上面编写的插件之后还有一些残留的无效地址，现在手动把它们删掉：





0x04 导出RunPE加壳后的程序

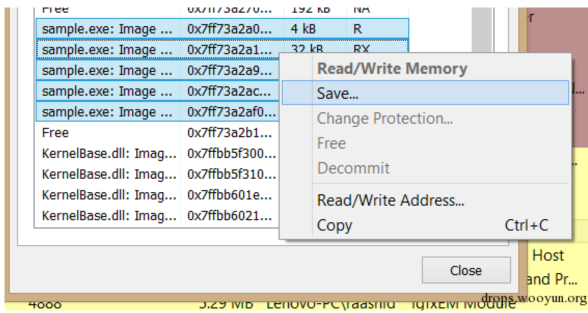
RunPE的工作原理是创建一个暂停状态的dummy进程，然后挖空并注入恶意代码。这种技术常用于隐藏恶意代码。RunPE注入的代码可以导出为一个有效的PE文件。对于PE+文件头需要修改一下，因为64位架构的PE文件一些字段使用了QWORD类型。

Windows loader加载程序到内存后根据IMAGE_SECTION_HEADER.VirtualSize进行对齐。但是Section表的RawSize可能会小于VirtualSize，这个时候会操作系统需要填充这块区域。

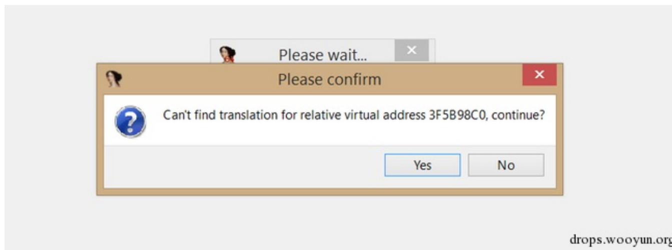
磁盘上的PE文件是根据IMAGE_OPTIONAL_HEADER64.FileAlignment进行对齐的，因此从内存中导出PE文件之后还需要根据IMAGE_OPTIONAL_HEADER64.FileAlignment对PE文件进行对齐。

```
typedef struct _IMAGE_OPTIONAL_HEADER64 {
    WORD        Magic;
    BYTE        MajorLinkerVersion;
    BYTE        MinorLinkerVersion;
    DWORD       SizeOfCode;
    DWORD       SizeOfInitializedData;
    DWORD       SizeOfUninitializedData;
    DWORD       AddressOfEntryPoint;
    DWORD       BaseOfCode;
    ULONGLONG   ImageBase;
    DWORD       SectionAlignment;
    DWORD       FileAlignment;
    WORD        MajorOperatingSystemVersion;
    WORD        MinorOperatingSystemVersion;
```

drops.wooyun.org



用IDA加载导出的PE文件时提示无法找到正确的虚拟地址，因为它的PE文件不对齐。



这个问题很好解决，我们先取出PE+文件结构：

```
#!c++
IMAGE_DOS_HEADER DosHdr = {0};
IMAGE_FILE_HEADER FileHdr = {0};
IMAGE_OPTIONAL_HEADER64 OptHdr = {0};

// Read All Structure as per offset

fread(&DosHdr, sizeof(IMAGE_DOS_HEADER), 0x01, fp);

fseek(fp, (unsigned int)DosHdr.e_lfanew + 4,SEEK_SET);

fread(&FileHdr, sizeof(IMAGE_FILE_HEADER), 1, fp);
fread(&OptHdr, sizeof(IMAGE_OPTIONAL_HEADER64), 1, fp);
```

遍历读取所有section header：

```
#!c++
    while (iNumSec < FileHdr.NumberOfSections)
    {
        fread(&pTail[iNumSec], sizeof( IMAGE_SECTION_HEADER), 1, fp);
        iNumSec++;
    }
```

然后读取第一个section的PointerToRawData：

```
#!c++
    i = ftell(fp);

    buffer = (unsigned char*) malloc(sizeof(char) * pTail[0].PointerToRawData + 1);

    fseek(fp, 0, SEEK_SET);

    fread(buffer, pTail[0].PointerToRawData, 1, fp); // Read/Write Everything Till the beginning of first section

    fwrite(buffer, pTail[0].PointerToRawData, 1, out);
```

最后，将数据以一个对齐的形式重写：

```
#!c++
    while ( i < iNumSec)
    {

        buffer = (unsigned char*) malloc(sizeof(char) * pTail[i].SizeOfRawData + 1);

        fseek(fp, pTail[i].VirtualAddress, SEEK_SET);
        fread(buffer, pTail[i].SizeOfRawData, 1, fp);

        fwrite(buffer, pTail[i].SizeOfRawData, 1, out);
        i++;
    }
```

全部修复完成之后就可以得到一个正确的PE文件，下面是IDA加载那个修复好的PE文件：

```
.text:0000000140004FBE    call    cs:GetProcAddress
.text:0000000140004FC4    mov     rcx, rax        ; Ptr
.text:0000000140004FC7    call    cs:EncodePointer
.text:0000000140004FCD    lea     rdx, aGetlastactivep ; "GetLastActivePopup"
.text:0000000140004FD4    mov     rcx, rsi        ; hModule
.text:0000000140004FD7    mov     cs:qword_14000D970, rax
.text:0000000140004FDE    call    cs:GetProcAddress
.text:0000000140004FE4    mov     rcx, rax        ; Ptr
.text:0000000140004FE7    call    cs:EncodePointer
.text:0000000140004FED    lea     rdx, aGetuserobjecti ; "GetUserObjectInformationW"
.text:0000000140004FF4    mov     rcx, rsi        ; hModule
.text:0000000140004FF7    mov     cs:qword_14000D978, rax
.text:0000000140004FFE    call    cs:GetProcAddress
.text:0000000140005004    mov     rcx, rax        ; Ptr
.text:0000000140005007    call    cs:EncodePointer
.text:000000014000500D    mov     r11, rax
.text:0000000140005010    mov     cs:qword_14000D988, rax
.text:0000000140005017    test    rax, rax
.text:000000014000501A    jz      short loc_14000503E
.text:000000014000501C    lea     rdx, aGetprocesswind ; "GetProcessWin
```

drops.wooyun.org

附： sample_plugin.rar