

# 南开大学

## 恶意代码分析与防治技术实验报告

### Lab10



学院：网络空间安全学院

专业：信息安全、法学

学号：2113203

姓名：付政烨

班级：信安法班

## 摘要

此次实验主要针对三个恶意软件样本进行了静态与动态分析,以揭示其内在运行机制。第一组样本 Lab10-01 在执行后会创建一个系统服务,该服务通过调用驱动程序直接修改注册表,将防火墙功能禁用。实验中我们利用 Procmon 监测注册表变化,使用 WinDbg 追踪服务调用的内核行为,发现驱动程序直接篡改了防火墙的启用状态。第二组样本 Lab10-02 也是通过服务创建一个隐藏的驱动程序。我们通过 Resource Hacker 工具提取可执行文件资源段中的驱动程序,逆向分析发现其劫持 SSDT 表针对目录查询函数进行 HOOK,完成目录隐藏的 Rootkit 功能。第三组样本 Lab10-03 同样也是服务驱动方式,不过应用了用户态与内核态结合的方式,通过特殊的 IOCTL 调用驱动内核函数直接修改进程链表,实现进程隐藏的目的。我们分析内核驱动逻辑,发现其直接修改链表节点完成隐藏。本次实验通过不同手段的结合,包括静态分析反汇编、行为监测、内核调试、驱动逆向等方式,对几种典型恶意代码进行了全面的分析检测,Exercise 了动态与静态分析相结合的恶意代码逆向分析思路,增强了对此类攻击技术的理解与应对能力。

**关键字：静态分析与动态分析； Rootkit； YARA 规则编写**

## 目录

<b>一、 实验目的</b>	<b>1</b>
<b>二、 实验原理</b>	<b>1</b>
<b>三、 实验过程</b>	<b>1</b>
(一) Lab10-1 . . . . .	1
(二) Lab10-2 . . . . .	7
(三) Lab10-3 . . . . .	9
<b>四、 Yara 规则编写</b>	<b>13</b>
(一) 脚本编写 . . . . .	13
1. Lab10-1 . . . . .	13
2. Lab10-2 . . . . .	14
3. Lab10-3 . . . . .	15
(二) 运行结果 . . . . .	16
<b>五、 Ida Python 脚本编写</b>	<b>16</b>
(一) 脚本编写 . . . . .	16
(二) 运行结果 . . . . .	17
<b>六、 实验心得与体会</b>	<b>17</b>

## 一、 实验目的

1. 运用静态分析与动态分析相结合的方式对恶意代码进行检测与分析。静态分析可以对程序进行逆向工程,了解其内在机制;动态分析可以观测程序运行时的系统调用与注册表、文件系统变化,揭示程序行为。二者互补使用,可以更全面地分析样本。
2. 运用各种工具辅助分析,如 Strings 提取字符串、PEview 查看导入表、IDA 逆向反汇编等。熟练使用这些工具可以极大地提高分析效率。
3. 学习使用调试器如 OD、WinDbg 跟踪用户态调用内核函数的行为,Hook 相关 API 监测调用参数,分析驱动入口点等内核调试技巧。这些技能对分析基于服务的内核态恶意代码至关重要。
4. 分析恶意代码后编写 YARA 规则与 IDA 脚本的方法,不仅可以检测该样本变种,也可以自动化分析过程中的某些环节,exercise 了此类能力的培养。
5. 通过具体案例的分析,加深对恶意代码的技术原理与手法的理解,如通过服务驱动加载内核组件,Hook 机制,直接操作内核数据结构等方式的应用,积累了针对这类攻击的识别与对策能力。

## 二、 实验原理

此次实验针对三个恶意软件样本,通过静态分析与动态分析相结合的方式进行了全面的检测与分析,以揭示其内在运行机制。在静态分析部分,我们利用导入表分析推断程序功能方向,字符串提取获取关键信息,资源段分析发现隐藏的组件,IDA 逆向工程反汇编分析实现逻辑;动态分析部分则通过 Procmon 等监测程序在运行时对系统的调用与修改,发现异常行为,从而推断功能实现方案。针对涉及内核组件的样本,我们使用 OD、WinDbg 等调试工具跟踪恶意驱动的加载与函数调用情况,Hook 相关 API 监测参数,分析驱动入口点的执行流程。在完整分析样本功能的基础上,我们编写了 YARA 检测规则,并利用 IDA Python 编写脚本自动化部分静态分析过程。第一组样本 Lab10-01 在执行后会创建一个系统服务,该服务通过调用驱动程序直接修改注册表,将防火墙功能禁用。第二组样本 Lab10-02 也是通过服务创建一个隐藏的驱动程序,我们通过 Resource Hacker 工具提取可执行文件资源段中的驱动程序,逆向分析发现其劫持 SSDT 表针对目录查询函数进行 HOOK,完成目录隐藏的 Rootkit 功能。第三组样本 Lab10-03 同样也是服务驱动方式,不过应用了用户态与内核态结合的方式,通过特殊的 IOCTL 调用驱动内核函数直接修改进程链表,实现进程隐藏的目的。我们分析内核驱动逻辑,发现其直接修改链表节点完成隐藏。

## 三、 实验过程

### (一) Lab10-1

#### 1. 这个程序是否直接修改了注册表 (使用 procmon 来检查)?

在进行恶意软件分析的静态阶段,我们首先对 Lab10-01.exe 执行了初步的静态分析。分析结果显示,程序导入了多个与服务管理相关的函数,表明其可能涉及创建并操作服务。此外,使用 Strings 工具进行的静态分析揭示了包含驱动程序路径 C:\Windows\System32\Lab10-01.sys 的关键字符串,以及其他与操作相关的信息。

pFile	Data	Description	Value
00004450	0000450E	Hint/Name RVA	01B2 StartServiceA
00004454	0000451E	Hint/Name RVA	0147 OpenServiceA
00004458	0000452E	Hint/Name RVA	004C CreateServiceA
0000445C	00004540	Hint/Name RVA	0145 OpenSCManagerA
00004460	000044FC	Hint/Name RVA	0035 ControlService
00004464	00000000	End of Imports	ADVAPI32.dll

图 1

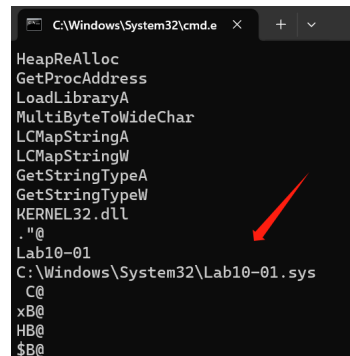


图 2

对 Lab10-01.sys 驱动程序的静态分析发现，其导入了与注册表修改直接相关的函数。尤其值得注意的是，字符串“EnableFirewall”被置为 0 时，Windows XP 自带的防火墙将被禁用。分析注册表键值时发现，\Registry\Machine 前缀的使用暗示了恶意软件可能执行的注册表写入操作。

pFile	Data	Description	Value
00000780	000009BC	Hint/Name RVA	03CB RtlCreateRegistryKey
00000784	000009D4	Hint/Name RVA	0266 KeTickCount
00000788	000009A4	Hint/Name RVA	04B2 RtlWriteRegistryValue
0000078C	00000000	End of Imports	ntoskrnl.exe

图 3

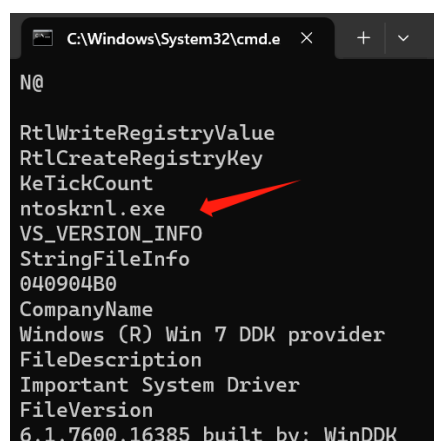


图 4

通过执行 Lab10-01.exe 并使用 procmon 工具监测，观察到了大量注册表操作。所有监测到的注册表写入操作中，唯一显著的是对 HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed 路径的写入。

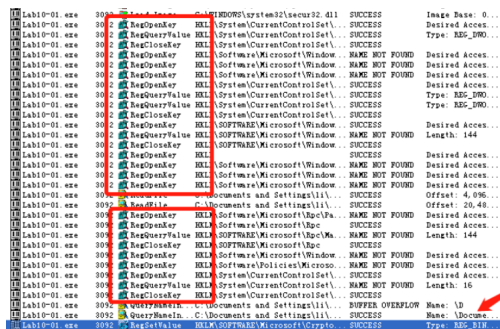


图 5

使用 IDA 工具进行功能概览时，程序首先通过 OpenSCManagerA 函数获取服务管理器句柄，再通过 CreateServiceA 创建名为 Lab10-01 的服务。若服务创建失败，程序将尝试打开已存在的服务。进一步分析表明，程序旨在创建并启动服务，最终通过 ControlService 函数卸载驱动。

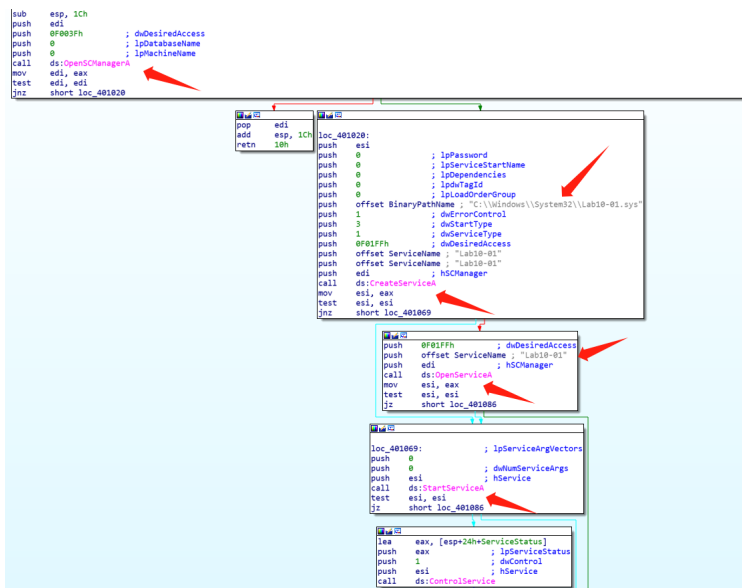


图 6

对 Lab10-01.sys 驱动程序的 DriverEntry 点分析发现，在 00010964 位置有一个无条件跳转指令，跳转至 sub\_10906，表明这里是真正的入口点。尽管 DriverEntry 函数相对较短且没有直接的函数调用，但赋值操作显示了 sub\_10486 函数中注册表操作的细节。

```
INIT:00010959 ; NTSTATUS __stdcall DriverEntry(PDRIVER_OBJECT DriverObject
INIT:00010959 public DriverEntry
INIT:00010959 DriverEntry ; DATA XREF: HEADER:
INIT:00010959 RegistryPath = dword ptr 8
INIT:00010959 RegistryPath = dword ptr 0Ch
INIT:00010959 mov edi, edi
INIT:00010958 push ebp
INIT:0001095C mov ebp, esp
INIT:0001095E call ___security_init_cookie
INIT:00010963 pop ebp
INIT:00010964 jmp _DriverEntry@8 ; DriverEntry(x,x)
INIT:00010964 endp
```

图 7

```

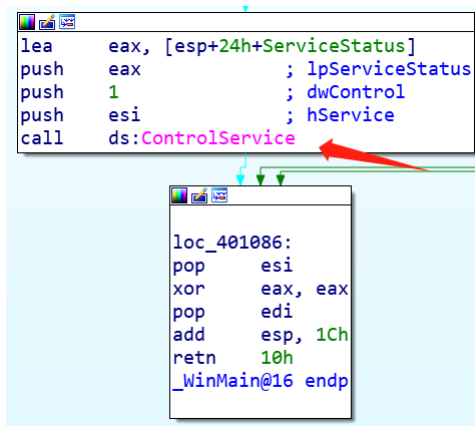
; Attributes: bp-based frame
; __stdcall DriverEntry(x, x)
_DriverEntry@8 proc near
arg_0= dword ptr 8
mov     edi, edi
push    ebp
mov     ebp, esp
mov     eax, [ebp+arg_0]
mov     dword ptr [eax+34h], offset sub_10486
xor     eax, eax
pop     ebp
retn    8
_DriverEntry@8 endp

```

图 8

## 2. 用户态的程序调用了 ControlService 函数，你是否能够使用 WinDbg 设置一个断点，以此来观察由于 controlService 的调用导致内核执行了怎样的操作？

我们首先确定了设置断点的确切位置。为此，我们在虚拟机环境中使用 OllyDbg 打开目标程序，并设置断点。当程序运行并触发断点时，我们便退出虚拟机，并借助宿主机中的 WinDbg 进行内核调试。此过程中，我们运用 !drvobj 命令获取驱动设备的句柄，由此推导出卸载函数的内存地址。接着，在驱动的卸载函数地址上设置新断点，并在虚拟机中运行至该断点被触发，然后返回宿主机中继续使用 WinDbg 进行单步调试，并检查相关代码。



```

lea     eax, [esp+24h+ServiceStatus]
push    eax           ; lpServiceStatus
push    1             ; dwControl
push    esi           ; hService
call    ds:ControlService

loc_401086:
pop     esi
xor     eax, eax
pop     edi
add     esp, 1Ch
retn    10h
_WinMain@16 endp

```

图 9

通过详细分析，我们追踪到恶意代码是如何操作 Lab10-01 服务的：启动服务并通过 ControlService 命令将其关闭。在中断内核调试器时，我们使用 !object \Driver 命令来查看所有已加载的驱动程序。在此过程中，我们发现 SERVICE\_CONTROL\_STOP 命令会调用 DriverUnload 函数。为了确定 DriverUnload 函数的具体地址，我们首先在 Lab10-01 驱动的入口点设置了断点，并使用 bu Lab10\_01!DriverEntry 命令。通过单步执行，我们等待 Lab10\_01.sys 被加载，然后使用 step out 指令跟踪至 nt! IopLoadUnloadDriver + 0x45。进一步地，我们使用 !object \Driver 列出所有已加载的驱动，并通过 dt 命令显示 Lab10-01 驱动的信息。

Hash	Address	Type	Name
00	8a0fd3a8	Driver	Beep
	8a20c3b0	Driver	NDIS
	8a053438	Driver	KSecDD
01	8a0ad7d0	Driver	Mouclass
	8a04ae38	Driver	Raspti
	89e28de8	Driver	es1371
02	89e29bf0	Driver	vax_svga
03	89e57b90	Driver	Fips
	8a1f87b0	Driver	Kbdclass
04	89fe6f38	Driver	VgaSave
	89ee6030	Driver	NDProxy
	8a2a60b8	Driver	Compbat
05	8a292ec8	Driver	Ptilink
	8a31e850	Driver	MountMgr
	8a2717e0	Driver	wdmaud
07	89e0d7a0	Driver	dmload
	8a2b0218	Driver	isapnp
	89e3c2c0	Driver	svaidi
08	8a28b948	Driver	redbook
	8a1f75f8	Driver	vmouse
	8a0ed510	Driver	atapi
09	89e0ea08	Driver	vm SCSI
10	89fe4da0	Driver	IpNat
	8a0e3728	Driver	RasAc

图 10

我们观察到 DriverUnload 函数的地址为 0xbaf7e486，并在该地址上设置断点。当重启 lab10-01.exe 且断点被触发后，我们使用 g 指令进行单步执行。在此过程中，我们注意到了对注册表的操作，特别是 RtlCreateRegistryKey 函数的调用，这表明在注册表中创建了一个新的键。微软官方文档表明，此函数用于向注册表添加键。后续代码中，RtlWriteRegistryValue 的调用成功创建了键值对。

```

kd> dt _DRIVER_OBJECT 89d3ada0
nt!_DRIVER_OBJECT
+0x000 Type : 0n4
+0x002 Size : 0n168
+0x004 DeviceObject : (null)
+0x008 Flags : 0x12
+0x00c DriverStart : 0xbaf7e000 Void
+0x010 DriverSize : 0x00
+0x014 DriverSection : 0x89e3b630 Void
+0x018 DriverExtension : 0x89d3ae48 _DRIVER_EXTENSION
+0x01c DriverName : UNICODE_STRING "\DriverLab10-01"
+0x024 HardwareDatabase : 0x80670ae0 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
+0x028 FastIoDispatch : (null)
+0x02c DriverInit : 0xbaf7e959 long +0
+0x030 DriverStartIo : (null)
+0x034 DriverUnload : 0xbaf7e486 void +0
+0x038 MajorFunction : [28] 0x804f354a long nt!IoInvalidDeviceRequest+0

```

图 11

在对参数列表进行观察后，我们特别关注 Path 参数，其值为 \Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\DomainProfile，并且 ValueName 的值为 EnableFirewall。我们注意到 ValueData 设置为 eax 寄存器中的值，而根据汇编代码，eax 中存放的是 [ebp-4] 的值。幸运的是，WinDbg 能够直接查看寄存器中的值。通过执行 r ebp 命令，我们确定 ebp 中的值为 f78ded58，从而得知 ebp-4 的值为 f78ded54。使用 dc f78ded54 指令检查该地址的值显示为 0，表明 ValueData 参数被设置为 0，即内核中禁用了 Windows 防火墙功能。

```

Lab10_01+0x49d:
baf7e49d 897dfc      mov     dword ptr [ebp-4].edi
kd> t
Lab10_01+0x4a0:
baf7e4a0 ffd6       call    esi
kd> t
nt!RtlCreateRegistryKey:
805ddade 8b1f      mov     edi,edi

```

图 12

继续分析，我们发现 RtlWriteRegistryValue 函数再次被调用，结合前述分析得出其参数结构。重点关注的参数包括 Path、ValueName 及 ValueData，其值分别为 \Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\StandardProfile、EnableFirewall 和 0，其效果与之前的函数相同，即从内核关闭防火墙。



```

EAX 0012FF1C
ECX 7708F86D ADVAPI32.7708F86D
EDX 00000000
EBX 77FD7000
ESP 0012FF08
EBP 0012FFC0
ESI 00144010
EDI 00144F20
EIP 00401080 Lab10-01.00401080
C 0 ES 0023 32bit 0(FFFFFFFF)
P 0 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SERVICE_EXISTS (00000431)
EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)

```

图 13

随后，我们利用 WinDbg 调试来观察卸载 Lab10-01.sys 时内核执行的操作。由于用户态程序会在加载 Lab10-01.sys 后立即卸载它，我们在驱动加载前设置了断点以保证其处于内存中以便分析。在 IDA 中找到 ControlService 的地址 0x401080 之后，我们用 OD 打开 Lab10-01.exe，在该地址设置断点并执行，随后跳出虚拟机，连接内核调试器获取 Lab10-01.sys 的相关信息。通过 !drvobj lab10-01 命令获取驱动对象，我们得知驱动对象地址为 81776978，并确认该驱动没有供用户空间应用程序访问的设备。

```

kd> dt DRIVER_OBJECT 81776978
nt!_DRIVER_OBJECT
+0x000 Type: 0x4
+0x002 Size: 0x168
+0x004 DeviceObject: (null)
+0x008 Flags: 0x1
+0x00c DriverStart: 0x8d44000 Void
+0x010 DriverSize: 0x80
+0x014 DriverSection: 0x8d79900 Void
+0x018 DriverExtension: 0x1776a20 DRIVER_EXTENSION
+0x01c DriverName: UNICODE_STRING "Driver Lab10-01"
+0x024 HardwareDatabase: UNICODE_STRING "REGISTRY-MACHINE-HARDWARE-DESCRIPTION-SYSTEM"
+0x028 FastIoDispatch: (null)
+0x02c DriverInit: 0x8d44959 long +0
+0x030 DriverStartIo: (null)
+0x034 DriverUnload: 0x8d4448c void +0
+0x038 MajorFunction: [2] 0x8d4454e long nt!IoPInvalidDeviceRequest+0

```

图 14

设置断点后，我们使用 g 命令恢复虚拟机运行，回到虚拟机中在 OD 中运行 F9 进行调试，并在断点处停止。在 WinDbg 中使用 p 命令单步调试，我们确认了断点处于函数入口。接下来的代码与 IDA 中的观察一致，包括多次调用 RtlCreateRegistryKey 函数创建注册表键，以及调用 RtlwriteRegistryValue 函数将 EnableFirewall 的值设置为 0，从而禁用 Windows XP 的防火墙。

```

kd> g
Breakpoint 1 hit
Lab10_01+0x408:
f8d44486 8bff          mov     edi,edi
kd> p
Lab10_01+0x408:
f8d44488 55          push    ebp
kd> p
Lab10_01+0x409:
f8d44489 8bec          mov     ebp,esp
kd> p
Lab10_01+0x40b:
f8d4448b 51          push    ecx

```

图 15

### 3. 这个程序做了些什么？

本研究的程序在执行后，将初始化一个服务进程。该进程通过执行特定的驱动程序代码，对注册表中的两个关键路径——\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\StandardProfile 和 \Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\DomainProfile，进行值的创建与修改。此操作的目的是实现对 Windows 操作系统防火墙的关闭。这一过程体现了通过直接干预注册表来控制系统安全配置的方法。

## (二) Lab10-2

### 1. 这个程序创建文件了吗？它创建了什么文件？

在未进行 Lab10-02.exe 文件的基础静态分析前，本研究首先利用 strings 工具对其进行初步检查。通过此方法，观察到程序中包含了一系列关于服务的创建与操作，以及文件写入的功能。随后，借助 PEView 工具进一步审视，揭示了与服务、文件和资源相关的导入函数。这些发现使我们推测该程序可能涉及服务的创建、启动或管理，文件的生成和编辑，以及资源段的某些处理。

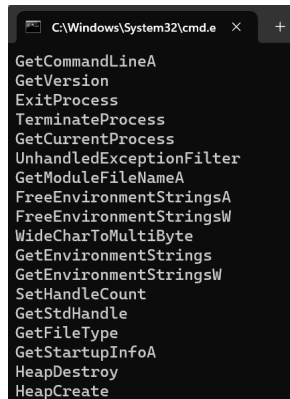


图 16

进一步地，本研究运用 Resource Hacker 工具检视资源段，发现存在一个含有 PE 头的 FILE 项。这一发现暗示了可能的可执行文件嵌入。

Address	Ordinal	Name	Library
00405000		CreateServiceA	ADVAPI32
00405004		StartServiceA	ADVAPI32
00405008		CloseServiceHandle	ADVAPI32
0040500C		OpenSCManagerA	ADVAPI32
00405014		CreateFileA	KERNEL32
00405018		SizeofResource	KERNEL32
0040501C		WriteFile	KERNEL32
00405020		FindResourceA	KERNEL32
00405024		LoadResource	KERNEL32
00405028		CloseHandle	KERNEL32
0040502C		GetCommandLineA	KERNEL32

图 17

之后，本研究转向基础动态分析。执行该程序时，它启动了一个命令行窗口并迅速退出。通过 procmon 工具监测，观察到该程序修改了注册表，并在 C:\Windows\System32 目录下创建了一个文件，同时增添了名为“486 WS Driver”的服务。但在检查上述目录时，并未找到该文件，且在 procmon 记录中也未见其删除活动。基于这些观察，推断该文件可能被隐藏，很可能是一个 Rootkit。

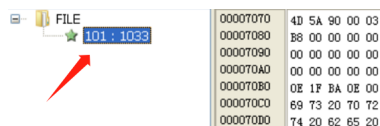


图 18

继续分析，使用命令行工具查询“486 WS Driver”服务状态时，发现服务虽在运行，却并未在硬盘上留有痕迹。为了深入了解，开始使用 WinDbg 工具在主机上进行调试。

```

SERVICE_NAME: 486 US Driver
        TYPE               : 1  KERNEL_DRIVER
        STATE                : 4  RUNNING
                           (STOPPABLE,NOT_PAUSABLE,IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE       : 0  (0x0)
        SERVICE_EXIT_CODE   : 0  (0x0)
        CHECKPOINT           : 0x0
        WAIT_HINT            : 0x0

```

图 19

## 2. 这个程序有内核组件吗？

在进行恶意软件分析的上下文中，Resource Hacker 工具被用于从可执行文件中提取存储在资源节 (Resource Section) 中的数据。具体到 Lab10-02.exe 案例，文件的前两个字节标记为“4D 5A”，这是可执行文件的标准签名，因此数据被提取并以.exe 格式保存。进一步的分析利用了 IDA 工具，一个专业的逆向工程工具，它揭示了文件入口点为 DriverEntry。这一发现指出，该文件实质上是一个驱动程序。由此可以总结，Lab10-02.exe 包含一个内核模块，该模块嵌入在其资源节中。这个驱动程序随后会被安装到硬盘上，并作为一个服务被加载进操作系统的内核，这是典型的驱动程序部署过程。

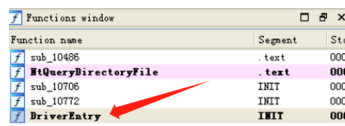


图 20

## 3. 这个程序做了些什么？

在前文的分析中，分析指出存在一种恶意代码，其功能是隐藏文件，属于 RootKit 的一类。该恶意软件通过钩子技术劫持系统服务描述表 SSDT，以篡改 NtQueryDirectoryFile 函数。其机制涉及通过定制特定的操作以达到隐藏文件的目的。根据文献中的相关方法，可以通过访问文件系统目录并使用 cp 命令对文件进行重命名，从而使隐藏文件显现，并将其导出。

```

C:\Windows\System32\cmd.e
Microsoft Windows [版本 10.0.22631.2586]
(c) Microsoft Corporation. 保留所有权利。

C:\Windows\System32>copy Mlux486.sys 1.sys
已复制      1 个文件。

```

图 21

进一步分析,通过使用 IDA 软件打开相关驱动程序,定位到 DriverEntry 例程。在此环节中,利用 RtlInitUnicodeString 函数,以 KeServiceDescriptorTable 和 NtQueryDirectoryFile 作为参数,从而初始化一个 Unicode 字符串。

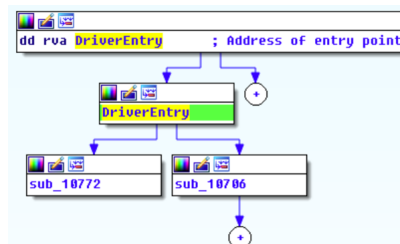


图 22

随后, 通过 `MmGetSystemRoutineAddress` 函数查询这两个参数的内存地址偏移, 以便于进行地址替换操作。

```
sub_10706 proc near
SystemRoutineName= UNICODE_STRING ptr -10h
DestinationString= UNICODE_STRING ptr -8

mov     edi, edi
push    ebp
mov     ebp, esp
sub     esp, 10h
push    esi
mov     esi, ds:RtlInitUnicodeString
push    edi
offset SourceString ; "H"
lea     eax, [ebp+DestinationString]
push    eax
; DestinationString
call    esi ; RtlInitUnicodeString
offset aKerServiceDescr ; "KeServiceDescriptorTable"
lea     eax, [ebp+SystemRoutineName]
push    eax
; DestinationString
call    esi ; RtlInitUnicodeString
mov     esi, ds:MmGetSystemRoutineAddress
lea     eax, [ebp+DestinationString]
push    eax
; SystemRoutineName
call    esi ; MmGetSystemRoutineAddress
mov     edi, eax
lea     eax, [ebp+SystemRoutineName]
push    eax
; SystemRoutineName
call    esi ; MmGetSystemRoutineAddress
```

图 23

实验 Lab10-3 涉及的操作注意事项如下: 本实验包含一个驱动程序文件与一个可执行文件。虽然可执行文件可以在任意位置运行, 为了确保程序能夜正常执行, 必须将驱动程序文件放置在受害者计算机中已存在的 `C:\Windows\System32` 目录下。其中, 可执行文件命名为 `Lab10-3.exe`, 而驱动程序文件则为 `Lab10-03.sys`。进行实验时, 需要特别注意这些文件的存放位置和操作步骤, 以保证实验的有效进行。

### (三) Lab10-3

#### 1. 这个程序做了些什么?

首先通过将驱动程序文件存放于 `C:\Windows\System32` 路径下并尝试运行, 观察到 Windows 任务管理器未显示 `lab10-03.exe` 进程, 推测该程序可能具备进程隐藏的能力。同时, 通过 `promon` 监控工具也未检测到相关的安装注册服务活动。通过 IDA 反汇编工具的分析可以进一步确认这一点。该程序在执行时会频繁地触发网页弹出, 这一现象是通过静态代码分析得出的结论。

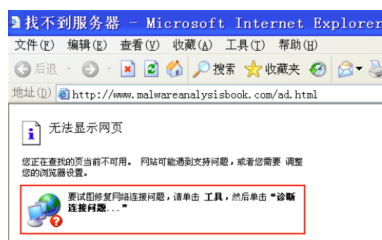


图 24

对可执行文件进行静态分析, 主要关注其导入表, 其中涉及到 `OpenSCManagerA`、`StartServiceA` 及 `CreateServiceA` 等导出函数, 这些函数表明该代码很可能在系统中创建了一个服务。同样地, 系统文件的导入表也展示了相同的函数, 进一步加强了这一推测。

Address	Ordinal	Name	Library
00404000		CloseServiceHandle	ADVAPI32
00404004		OpenSCManagerA	ADVAPI32
00404008		CreateServiceA	ADVAPI32
0040400C		StartServiceA	ADVAPI32
00404014		CreateFileA	KERNEL32
00404018		DeviceIoControl	KERNEL32
0040401C		Sleep	KERNEL32
00404020		GetStringTypeA	KERNEL32
00404024		LCMapStringW	KERNEL32
00404028		LCMapStringA	KERNEL32
0040402C		MultiByteToWideChar	KERNEL32
00404030		LoadLibraryA	KERNEL32
00404034		GetProcAddress	KERNEL32
00404038		GetModuleHandleA	KERNEL32
0040403C		GetStartupInfoA	KERNEL32
00404040		GetCommandLineA	KERNEL32
00404044		GetVersion	KERNEL32
00404048		ExitProcess	KERNEL32
0040404C		TerminateProcess	KERNEL32
00404050		GetCurrentProcess	KERNEL32

图 25

Address	Ordinal	Name	Library
00010480		IoCompleteRequest	ntoskrnl
00010484		IoDeleteDevice	ntoskrnl
00010488		IoDeleteSymbolicLink	ntoskrnl
0001048C		RtlInitUnicodeString	ntoskrnl
00010490		IoGetCurrentProcess	ntoskrnl
00010494		IoCreateSymbolicLink	ntoskrnl
00010498		IoCreateDevice	ntoskrnl
0001049C		KeTickCount	ntoskrnl

图 26

通过深入研究上述函数，我们可以了解到 IoCreateDevice 用于创建设备对象，IoCreateSymbolicLink 用于建立设备名称和用户可见名称之间的符号链接，IoGetCurrentProcess 返回当前进程的指针，IoCompleteRequest 指示已完成给定 I/O 请求的所有处理，KeTickCount 用于检索系统启动以来的毫秒数，而 RtlInitUnicodeString 则用于初始化 Unicode 字符串。这些函数的分析揭示了该驱动可能在修改运行中的进程或需要获取进程相关信息。

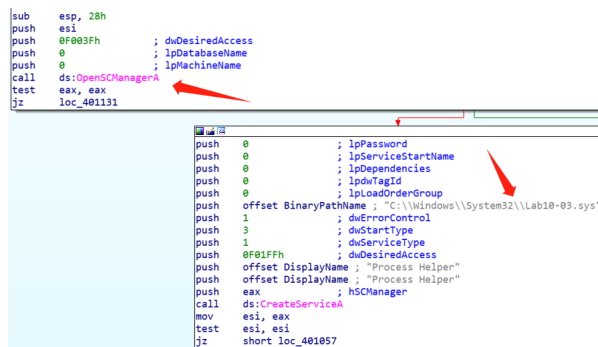


图 27

进一步分析代码，首个调用的函数为 OpenSCManagerA，其用途为打开服务控制管理器，随后的操作涉及到创建服务并指定服务的二进制文件位置等参数。此外，参数 dwStartType 设为 3，意味着用户可以手动启动该服务，而 dwServiceType 值为 1，表示这是一个驱动服务。CreateServiceA 调用成功后，接着会执行 StartService，一旦调用，恶意驱动 Lab10-03.sys 便加载入内核。

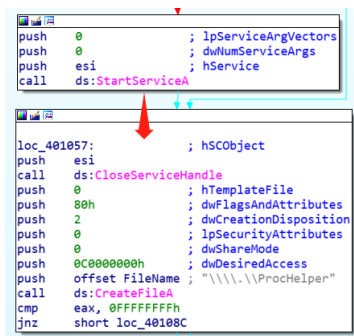


图 28

在代码的后续部分，创建了一个文件并将其作为句柄打开，然后通过 `DeviceIoControl` 将控制代码发送到设备驱动程序，进行特定操作。分析表明，`DeviceIoControl` 的使用在这里显得异常，因为它没有将任何信息发送到内核驱动中，也没有从内核驱动中接收任何反馈。另一个异常点是 `dwIoControlCode` 的值为 `abcdedf01`。

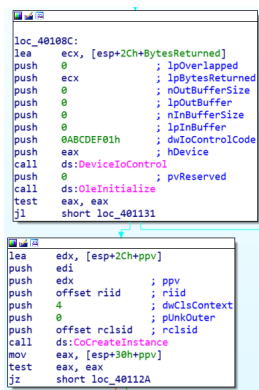


图 29

CoCreateInstance 函数则用于创建与指定 CLSID 关联的类的单个未初始化对象，这通常用于创建 COM 对象。此外，通过 SysAllocString 函数，我们可以确定程序在运行时所打开的特定网页链接。

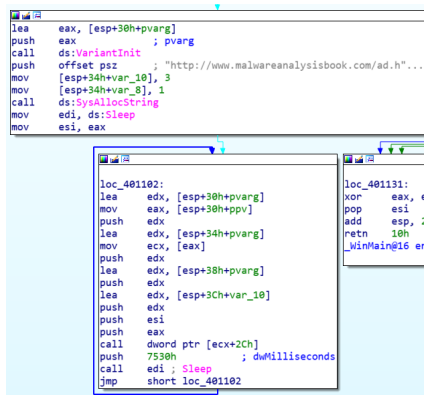
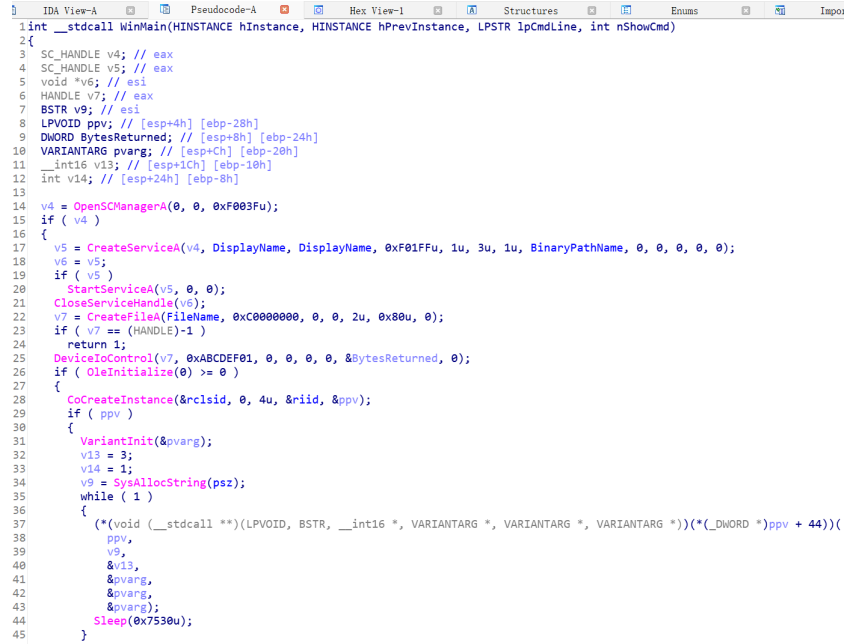


图 30

在代码的最后部分, 恶意程序调用了 Sleep 函数实现了一个时间延迟, 之后进入了一个循环

代码段,直至系统关闭才退出。进一步地,为深入分析恶意代码的具体实现细节,我们对相关的.sys 驱动文件进行了反汇编逆向工程分析。



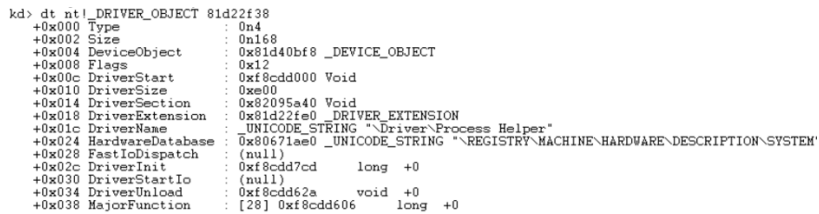
```

1  IDA View-1  Pseudocode-1  Hex View-1  Structures  Enums  Imports
2  int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
3  {
4  SC_HANDLE v4; // eax
5  SC_HANDLE v5; // eax
6  void *v6; // esi
7  HANDLE v7; // eax
8  BSTR v9; // esi
9  LPVOID ppv; // [esp+4h] [ebp-28h]
10 DWORD BytesReturned; // [esp+8h] [ebp-24h]
11 VARIANTARG pvarg; // [esp+Ch] [ebp-20h]
12 __int6 v13; // [esp+1Ch] [ebp-10h]
13 int v14; // [esp+24h] [ebp-8h]
14 v4 = OpenSCManager(0, 0, 0xF003Fu);
15 if ( v4 )
16 {
17 v5 = CreateServiceA(v4, DisplayName, DisplayName, 0xF01FFu, 1u, 3u, 1u, BinaryPathName, 0, 0, 0, 0);
18 v6 = v5;
19 if ( v5 )
20 StartServiceA(v5, 0, 0);
21 CloseServiceHandle(v6);
22 v7 = CreateFileA(fileName, 0xC0000000u, 0, 0, 2u, 0x80u, 0);
23 if ( v7 == (HANDLE)-1 )
24 return 1;
25 DeviceIoControl(v7, 0xABCDEF01, 0, 0, 0, &BytesReturned, 0);
26 if ( OleInitialize(0) >= 0 )
27 {
28 CoCreateInstance(&rcIsid, 0, 4u, &riid, &ppv);
29 if ( ppv )
30 {
31 VariantInit(&pvarg);
32 v13 = 3;
33 v14 = 1;
34 v9 = SysAllocString(psz);
35 while ( 1 )
36 {
37 *(void (__stdcall **)(LPVOID, BSTR, __int6 *, VARIANTARG *, VARIANTARG *, VARIANTARG *))(*(_DWORD *)ppv + 44)(
38 ppv,
39 v9,
40 &v13,
41 &pvarg,
42 &pvarg,
43 &pvarg);
44 Sleep(0x7530u);
45 }

```

图 31

IoCreateSymbolicLink 函数被恶意驱动程序利用,以在设备对象名称和用户可见的设备名称之间创建符号链接。这通常出现在 DriverEntry 函数中,也就是驱动程序的初始化入口点。IoCreateSymbolicLink 可用于将设备对象链接到一个全局的用户可见符号链接名,以便后续可以通过该符号链接名访问驱动对象。在 windbg 调试工具中,也可以观察到 IoCreateSymbolicLink 的调用及其参数,从而看到恶意驱动所建立的符号链接关系。



```

kd> dt nt!_DRIVER_OBJECT 81d22f38
+0x000 Type : 0n4
+0x002 Size : 0n168
+0x004 DeviceObject : 0x81d40bf8 _DEVICE_OBJECT
+0x008 Flags : 0x12
+0x00c DriverStart : 0xf8cdd000 Void
+0x010 DriverSize : 0xe00
+0x014 DriverSection : 0x82095a40 Void
+0x018 DriverExtension : 0x81d22ie0 DRIVER_EXTENSION
+0x01c DriverName : UNICODE_STRING "\Driver\Process Helper"
+0x024 HardwareDatabase : 0x80671ae0 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
+0x028 FastIoDispatch : (null)
+0x02c DriverInit : 0xf8cdd7cd long +0
+0x030 DriverStartIo : (null)
+0x034 DriverUnload : 0xf8cdd62a void +0
+0x038 MajorFunction : [28] 0xf8cdd606 long +0

```

图 32

与前文所述方法类似,本研究案例中的恶意代码同样采用了直接修改操作系统内核链表节点的方式,来实现进程隐藏和根 kit 功能。具体而言,代码修改了保存进程列表的链表结构,通过直接访问链表节点并删改节点内容,将恶意进程从链表中移除,使其避过了普通的进程列举和监控。这一实现过程反映了案例采用了类似的技术手段,即不经由正规的系统调用接口,直接操作系统核心数据结构的方法。这种直接侵入式的核心数据结构修改手法,绕过了操作系统的外围防护,直接攻击了更为脆弱的内核本身。总体来看,本案例的这一代码段再次验证并体现了研究中发现的技术手法和思路,即通过对内核链表的直接编辑来实现进程隐藏的目标,这与前文所述方法类似。



```

F8C8E666
F8C8E666 ; Attributes: bp-based frame
F8C8E666
F8C8E666 ; int __stdcall sub_F8C8E666(int, PIRP Irp)
F8C8E666 sub_F8C8E666 proc near
F8C8E666
F8C8E666 Irp= dword ptr 0Ch
F8C8E666
F8C8E666 mov     edi, edi
F8C8E668 push   ebp
F8C8E669 mov     ebp, esp
F8C8E66B call    ds:IoGetCurrentProcess
F8C8E671 mov     ecx, [eax+8Ch]
F8C8E677 add     eax, 88h
F8C8E67C mov     edx, [eax]
F8C8E67E mov     [ecx], edx
F8C8E680 mov     ecx, [eax]
F8C8E682 mov     eax, [ecx+4]
F8C8E685 mov     [ecx+4], eax
F8C8E688 mov     ecx, [ebp+Irp] ; Irp
F8C8E68B and     dword ptr [ecx+18h], 0
F8C8E68F and     dword ptr [ecx+1Ch], 0
F8C8E693 xor     dl, dl ; PriorityBoost
F8C8E695 call    ds:IoCompleteRequest
F8C8E69B xor     eax, eax
F8C8E69D pop     ebp
F8C8E69E retn    8
F8C8E69E sub_F8C8E666 endp
F8C8E69E

```

图 33

## 2. 一旦程序运行，你怎样停止它？

只能通过重启来停止。

## 3. 它的内核组件做了什么操作？

内核组件利用调用 DeviceIoControl 函数这一方式，去直接操作和修改进程的链表 (LIST\_ENTRY) 结构，从而隐藏自身在链表中的存在。通过直接修改内核的数据结构，内核组件成功地修改了操作系统保留的进程列表，将自身进程从列表中移除，使其不被普通的进程列举和查看到。这种直接绕过 API 接口，直接操作内核内部数据的方式，利用了操作系统数据结构的脆弱性，实现了进程的隐藏和 rootkit 的功能。这种行为修改了操作系统的关键数据，破坏了操作系统的内部完整性，属于对系统的攻击行为。

# 四、 Yara 规则编写

## (一) 脚本编写

### 1. Lab10-1

```

1 rule RukeforLab10_01exe {
2
3     meta:
4         description = "Lab10-01.exe"
5
6     strings:
7         $s1 = "C:\\Windows\\System32\\Lab10-01.sys" fullword ascii
8         $s2 = "Hello_World!" fullword wide
9         $s3 = /RegWriterApp Version 1\\.0/i fullword wide
10        $s7 = "Copyright_(C)_2011" fullword wide
11
12    condition:
13        uint16(0) == 0x5a4d and
14        uint32(uint32(0x3c)) == 0x00004550 and
15        filesize < 80KB and
16        $s1 and $s2 and $s3 and $s7

```



```

17 }

1 rule RuleforLab10_01sys {
2     meta:
3         description = "Lab10-01.sys"
4
5     strings:
6         $s1 = "sioclt.pdb" fullword ascii
7         $s2 = "Lab10-01.sys" fullword wide
8         $s3 = "Important_System_Driver" fullword wide
9         $s4 = /\Registry\Machine\SOFTWARE\Policies\Microsoft(\\
            WindowsFirewall)?/i fullword wide
10        $s6 = "6.1.7600.16385_built_by:_WinDDK" fullword wide
11        $s7 = "_ABC_Corp." fullword wide
12
13    condition:
14        uint16(0) == 0x5a4d and
15        uint32(uint32(0x3c)) == 0x00004550 and
16        filesize < 10KB and
17        $s1 and $s2 and $s3 and $s4 and $s6 and $s7
18 }

```

## 2. Lab10-2

```

1 rule RuleforLab10_02 {
2     meta:
3         description = "Lab10-02.exe"
4
5     strings:
6         $s1 = "Mlwx486.sys" fullword ascii
7         $s2 = "sioclt.pdb" fullword ascii
8         $s3 = "SICTL.sys" fullword wide
9         $s4 = "Failed_to_open_service_manager." fullword ascii
10        $s5 = "Failed_to_start_service." fullword ascii
11        $s6 = "Sample_IOCTL_Driver" fullword wide
12        $s7 = { 22 57 57 53 68 74 54 40 } // Hex representation of "\"WShT@"
            avoiding fullword ascii
13        $s8 = { 56 57 75 42 68 68 54 40 } // Hex representation of "VWuBhhT@"
            avoiding fullword ascii
14        $s9 = "486_WS_Driver" fullword ascii
15        $s10 = "6.1.7600.16385_built_by:_WinDDK" fullword wide
16        $s11 = "KeServiceDescriptorTable" fullword wide
17        $s16 = "Failed_to_create_service." fullword ascii
18
19    condition:
20        uint16(0) == 0x5a4d and
21        uint32(uint32(0x3c)) == 0x00004550 and
22        filesize < 100KB and

```

```
23     3 of ($s4, $s5, $s6, $s10, $s11, $s16) and
24     1 of ($s1, $s2, $s3, $s7, $s8, $s9)
25 }
```

### 3. Lab10-3

```
1 rule RuleforLab10_03exe {
2     meta:
3         description = "Lab10-03.exe"
4     strings:
5         $s1 = "C:\\Windows\\System32\\Lab10-03.sys" fullword ascii
6         $s2 = "http://www.malwareanalysisbook.com/ad.html" fullword ascii
7         $s3 = "Process_Helper" fullword ascii
8         $s4 = "\\\\.\\ProcHelper" fullword ascii
9     condition:
10        uint16(0) == 0x5a4d and
11        uint32(uint32(0x3c)) == 0x00004550 and filesize < 70KB and
12        ($s1 or $s2) and ($s3 or $s4)
13 }
```

```
1 rule RuleforLab10_03sys {
2     meta:
3         description = "Lab10-03.sys"
4     strings:
5         $s1 = "c:\\\\winddk\\\\7600.16385.1\\\\src\\\\general\\\\
6             rootkitprochide\\\\wdm\\\\sys\\\\objfre_wxp_x86\\\\i386\\\\sioclt.
7             pdb" fullword ascii
8         $s2 = "Lab10-03.sys" fullword wide
9         $s3 = "Important_Process_Helper" fullword wide
10        $s4 = /\DosDevices\ProcHelper/i fullword
11        $s5 = /\Device\ProcHelper/i fullword
12        $s7 = "_ABC_Corp." fullword wide
13     condition:
14        uint16(0) == 0x5a4d and
15        uint32(uint32(0x3c)) == 0x00004550 and
16        filesize < 10KB and
17        3 of them
18 }
```

## (二) 运行结果

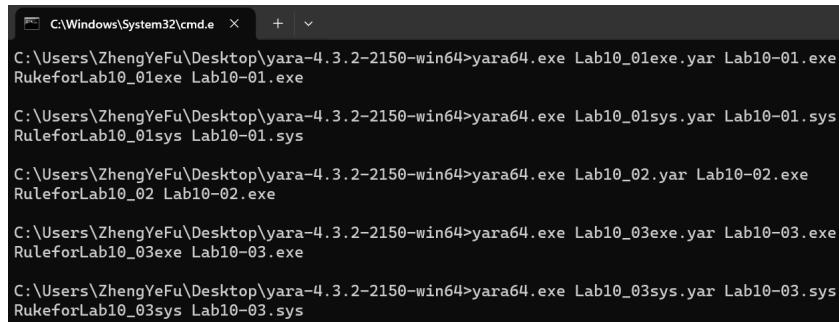


图 34

## 五、 Ida Python 脚本编写

### (一) 脚本编写

```

1 # coding=utf-8
2
3 import idautils
4 import idc
5
6 def find_xor_functions():
7     # 遍历所有函数
8     for func_ea in idautils.Functions():
9         # 为了检查函数是否包含XOR指令和被调用多次，我们需要设置两个计数器
10        xor_count = 0
11        call_count = 0
12
13        # 遍历函数中的每个指令
14        for ins_ea in idautils.FuncItems(func_ea):
15            mnem = idc.print_insn_mnem(ins_ea)
16            if mnem == "xor":
17                xor_count += 1
18
19        # 遍历到该函数的所有交叉引用
20        for xref in idautils.XrefsTo(func_ea):
21            call_count += 1
22
23        # 检查是否满足条件：包含XOR并被调用多次
24        if xor_count > 0 and call_count > 1:
25            print("Function: %s at 0x%x contains XOR and is called %d times"
26                  % (idc.get_func_name(func_ea), func_ea, call_count))
27
28 find_xor_functions()
29
30 }

```

这个 IDA Pro 脚本的主要功能是自动遍历程序中的所有函数, 找出其中包含 XOR 指令并被其他函数多次调用的函数。具体实现方法是:

- 使用 `idautils` 模块的 `Functions()` 函数遍历程序中所有函数的起始地址。
- 对每个函数地址, 利用 `FuncItems()` 获取函数中的所有指令地址, 然后检查每个指令的汇编指令 `mnemonic`, 统计其中 XOR 指令的数量, 作为 `xor_count` 变量。
- 使用 `XrefsTo()` 函数获取所有引用当前函数地址的代码引用, 遍历这些引用, 统计其中调用引用的数量作为 `call_count` 变量。
- 判断 `xor_count` 和 `call_count` 的值, 如果 `xor_count` 大于 0 且 `call_count` 大于 1, 说明该函数同时包含 XOR 指令又被其他函数多次调用, 可能具有某种功能, 如编码解码。
- 最后, 将满足条件的函数名称、函数地址和调用次数打印输出。

通过这种自动化的静态分析方法, 我们可以快速从复杂的程序中检测出含有特定汇编指令并被多次调用的函数, 辅助反编译人员进行进一步的交互式分析与理解。

## (二) 运行结果

```
-----
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:53:40) [MSC v.1500 64 bit (AMD64)]
IDAPython v1.7.0 final (serial 0) (c) The IDAPython Team <idapython@googlegroups.com>
-----

Type library 'vc6win' loaded. Applying types...
Types applied to 0 names.
Using FLIRT signature: Microsoft VisualC 2-14/net runtime
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.
Function: _parse_cmdline at 0x4015ff contains XOR and is called 2 times
Function: __NMSG_WRITE at 0x401cd5 contains XOR and is called 4 times
Function: _CPToLCID at 0x40204d contains XOR and is called 2 times
Function: _strcat at 0x402290 contains XOR and is called 3 times
Function: _strlen at 0x4023f0 contains XOR and is called 5 times
Function: __crtLCMapStringA at 0x40317e contains XOR and is called 2 times
-----
```

图 35

## 六、 实验心得与体会

在这次实验中, 我深入了解了恶意软件的分析过程, 这一过程不仅涉及多方面的技术知识, 也需要耐心和细致的观察力。通过静态和动态分析的结合, 我学会了如何从不同角度解构恶意软件的行为和策略。在静态分析中, 我使用了导入表分析、字符串提取、资源段分析以及 IDA 逆向工程等技术, 这些技术帮助我理解了恶意软件的构建和隐藏机制。动态分析让我看到了恶意软件对系统的实际影响, 如系统调用和修改, 这些都是推断其功能的关键线索。特别是在分析涉及内核组件的恶意软件时, 使用调试工具如 OD 和 WinDbg 进行跟踪, 监测 API 调用和分析执行流程是一项挑战, 但也是一次宝贵的学习经历。此外, YARA 规则的编写和 IDA Python 脚本的自动化应用, 大大提高了分析效率。对于具体的样本分析, 我观察到了多种恶意技术的实现, 例如通过创建系统服务直接修改注册表来禁用防火墙, 或者通过劫持 SSDT 表和修改进程链表来隐藏驱动程序和进程。这些技术不仅体现了恶意软件的隐蔽性, 也展示了攻击者的先进技巧。

## 参考文献

- [1] SM-D.Practical Malware Analysis[J].Network Security, 2012, 2012(12):4-4.DOI:10.1016/S1353-4858(12)70109-5.