X'con2007

AV引擎－虚拟机脱壳技术

大成天下－数据安全实验室
linxer

目　录

- 壳的简要介绍
- 脱壳的必要性
- 常用自动脱壳方法
- 基于虚拟机的自动脱壳技术
  - 简易x86 CPU仿真
  - 仿真API
  - PE loader技术
  - 建立程序执行环境
  - 壳用到的系统特性仿真
  - dump技术
- 虚拟机脱壳的限制
- 其他应用
- 演示

一、壳的简要介绍

- 压缩壳:
  - 减小程序体积，便于保存传输，阻止逆向分析。
- 加密壳:
  - 利用各种Anti技术来防止程序被调试、逆向、破解。

二、脱壳的必要性

- 基于特征码的传统反病毒技术仍然发挥主导作用。
- 越来越多的木马后门使用加壳来逃避特征码检测。
  - 我们统计表明，目前95%以上的病毒加壳。
  - 60%的病毒使用了常见壳来保护自身。
  - 流行壳列表：http://dswlab.com/toppacker.html
- 对病毒细节的了解需求

三、常用自动脱壳方法

- 静态脱壳:
  - 分析加壳算法，写出逆过程来脱壳，但目前很多壳比较复杂，对各类加密壳，这种方法显得苍白。
- 调试器脱壳:
  - 利用调试器，让加壳程序自己解码，暴露”原始”状态来达到脱壳效果，但外壳程序anti能力强悍时，常让这种方法无能为力。
- 虚拟机脱壳:
  - 仿真CPU和一些系统特性，让加壳程序在虚拟机里面运行来脱壳，仿真系统特性的多少决定脱壳能力。

四、基于虚拟机的自动脱壳技术

- 4.1 简易x86 CPU仿真
  - 4.1.1 环境仿真:
    - 普通寄存器，段寄存器，标志位寄存器，EIP寄存器和调试寄存器等。
  - 4.1.2 反汇编器:
    - 负责机器码的识别
  - 4.1.3 寻址系统:
    - 对内存寻址方式进行解析，并负责计算出虚拟地址。
  - 4.1.4 指令解释系统:
    - 完成相应指令的动作行为。

4.2 API仿真

- 仿真具备实际功能的API：
  - 比如ReadFile这类API
- 仿真负责堆栈平衡的API：
  - 比如SetWindowLong，对这类API只要在仿真的函数中把压入的参数弹出，并返回函数调用成功即可。

4.2.1 仿真API的导入

- 三种API导入方式：

  虚拟机里导入的并不是仿真API的地址，而是API的一个代号ID，程序执行过程中根据API ID调用相应的仿真API函数。

  – load的时候导入
  – 程序执行过程中用仿真的GetProcAddress导入
  – 根据具体模块，自行分析模块的导出表，实现API导入

4.2.2 仿真API的调用

- 当eip寄存器存在着API ID的时候，表示要调用仿真API，这可以在执行完每条指令后判定，这虽然完美可行，但效率低。由于用于发生函数调用的指令有限，可以在这些指令解析函数里判定是否要调用API，这些指令主要有:
  - call [xxxxxxxx]/[reg]， 机器码0xff /2
  - jmp [xxxxxxxx]/[reg]， 机器码0xff /4
  - ret 机器码0xc3
  - jmp xxxxxxxx， 机器码0xe9，很少见
  - call xxxxxxxx， 机器码0xe8，很少见

4.3 PE loader技术

- 操作系统的PE加载方法
  - 格式检查，环境创建
- 虚拟机载入PE的方法

4.3.1 PE文件合法性，有效性判定

- 检测MZ标志
- 检测PE标志
- 节的数量 >= 1
- NumberOfRvaAndSizes >= 2
- FileAlignment <= SectionAlignment
- SizeOfHeaders < SizeOfImage
- FileAlignment是否为2的n次幂，在xp上最小为0x200
- 每个节大小的效验，及其节的对齐属性的效验
  - VirtualAddress必须按SectionAlignment对齐
  - VirtualSize的大小仅用来效验是否当前节是否跨越下一个节，或者超出SizeOfImage，如果不存在越界问题的话，值大小无实际意义
  - SizeOfRawData无对齐要求
  - PointerToRawData无对齐要求

4.3.2 PE loader过程

4.3.2.1 加载PE头

memcpy(lpExePEBuff， lppeOriBuf， pINH->OptionalHeader.SizeOfHeaders);

4.3.2.2 加载节

内存中节的大小:并不是由VirtualSize决定的， 对非最后一个节，节在内存中大小是前后两节VirtualAddress之差，对最后一个节，节在内存中大小是SizeOfImage与本节VirtualAddress之差

节在磁盘中偏移:如果PointerToRawData已经按FileAlignment对齐了，偏移就是PointerToRawData;如果PointerToRawData没有按FileAlignment对齐，节在磁盘中偏移为PointerToRawData向下按FileAlignment取整

节在磁盘中大小:如果SizeOfRawData按FileAlignment对齐了，大小就是SizeOfRawData;如果SizeOfRawData没有按FileAlignment对齐，节在磁盘中的大小为SizeOfRawData向上按FileAlignment取整

- 示例:

  假定

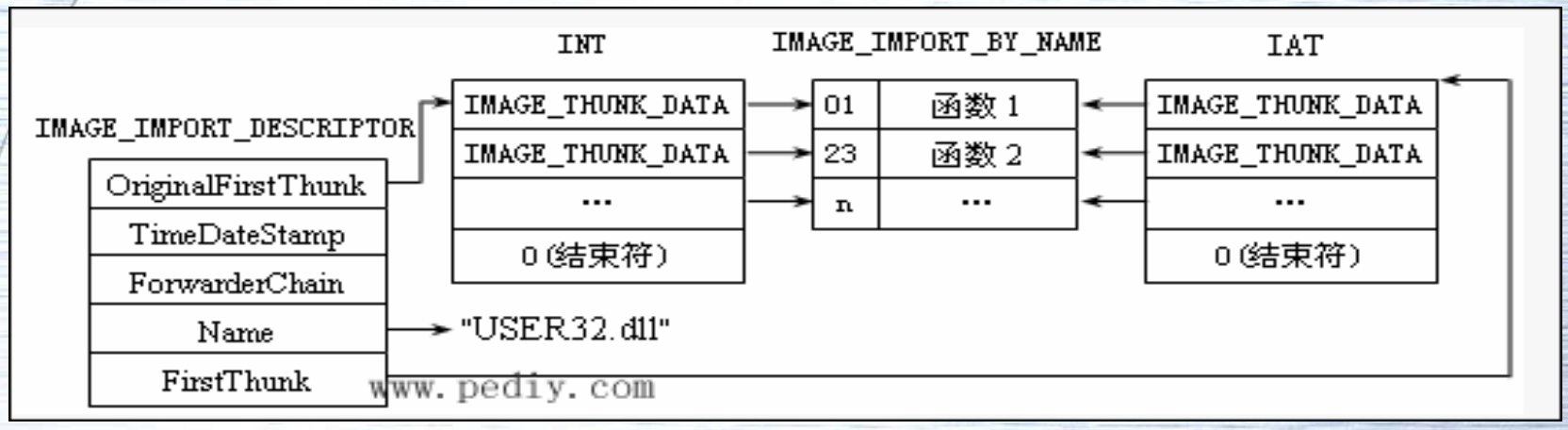
  SectionAlignment = 0x1000

  FileAlignment = 0x200

00000138	2E 55 70 6	ASCII ".Upack"	SECTION
00000140	00800000	DD 00008000	VirtualSize = 8000 (32768.)
00000144	00100000	DD 00001000	VirtualAddress = 1000
00000148	B7000000	DD 000000B7	SizeOfRawData = B7 (183.)
0000014C	11000000	DD 00000011	PointerToRawData = 11
00000150	00000000	DD 00000000	PointerToRelocations = 0
00000154	00000000	DD 00000000	PointerToLineNumbers = 0
00000158	0000	DW 0000	NumberOfRelocations = 0
0000015A	0000	DW 0000	NumberOfLineNumbers = 0

上面这个节并非将磁盘中偏移0x11，大小0xb7的内容拷到内存中偏移0x1000开始处，而是将磁盘中偏移0x00，大小0x200的内容拷到内存中偏移0x1000去了

4.3.2.3 处理IAT，导入仿真API代号



a. x.OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].Size是否一定大于0才说明有导入表呢?

b. IMAGE_IMPORT_DESCRIPTOR数组是否一定需要以0结尾?

c. 图中INT是否一定需要指向IMAGE_IMPORT_BY_NAME数组?

d. 图中IAT是否一定需要指向IMAGE_IMPORT_BY_NAME数组?

e. 不同dll在IAT中是否真的没有0(或者无效地址)隔开就不行?

4.3.2.4 loader过程中其它事宜

- tls处理
- 重定位表:
  - 无论对dll还是exe脱壳，由于虚拟机只加载一个模块，因此不存在重定位问题
- 导入表备份问题:
  - 对有些壳，脱壳完成后可以不重建导入表，可以在load的时候备份导入表，避免dump的时候重建导入表;对有些壳，也可以在load的时候备份导入表，方便dump的时候重建导入表。

4.4 建立程序执行环境

- 初始化CPU寄存器环境
- 判定PE是否合法有效，并获取PE文件中一些数据
- 分配内存，完成load过程
- 仿真dll加载分析环境(很多壳不需要这个)
- 初始化堆、堆栈、PEB、FS段、环境变量等信息、建立”进程”空间。
- 往堆栈中压入一些特殊数据
- 初始化反汇编器
- 设置eip寄存器

4.5 壳用到的系统特性仿真

- 在执行加壳程序过程中，由于没有操作系统的支持，很多系统特性需要仿真，这些系统特性仿真的多少，也直接决定了虚拟机的脱壳能力。

4.5.1 SEH仿真

- 结构化异常处理(Structured Exception Handling)
- 在壳中的应用主要用来反跟踪，也有些壳在异常处理回调函数中修改CONTEXT结构成员，来完成改变程序流程，清除硬件断点等功能.

- 安装SEH异常处理项
  push offset sehHandler
  push fs:[0]
  mov fs:[0]， esp (链表项还可以存在在别处)
- 卸载SEH异常处理项
  pop fs:[0]
  add esp， 4 (链表项还可以存在在别处)
- 当改写fs:[0]内容时，说明在安装或者卸载SEH异常处理项
- 异常的捕获
  - 对改写TF标志的单步异常
  - int3 int1 int n异常

- 内存访问权限导致的异常
- 访问不存在内存页异常
- 特权指令异常
- 无效指令异常
- 除0异常
- 设置Drx寄存器触发的单步异常

– 捕获到异常时，暂停对当前指令的执行，初始化异常处理环境后，从fs:[0]找到异常处理回调函数地址，将这个地址赋值给eip，执行异常处理回调函数

– 在异常处理回调函数执行结束时，对eax值进行判定，eax为0表示异常处理成功，恢复上下文继续执行;eax为1则调用下一个SEH异常处理回调函数;eax为2表示发生异常嵌套;eax为3则需要对异常处理进行堆栈展开操作

4.5.2 仿真API地址空间

- 很多壳在脱壳过程中都要判断自己用到的API是否被下0xcc断点，如果API被下断点，一般都会自己终止进程，使脱壳无法继续。
- 一般都是看API前几个字节是否是0xcc，如果前几个字节存在0xcc，就说明这个API被下断点了，这里可以给每个API分配几个字节，用来供壳验证是否被下0xcc断点;当然这里有比较多的方法可以处理这个问题。

4.5.3 句柄检测

- 很多壳在脱壳过程中要检测调试器，要检测一些监视工具，对这些工具的检测有比较多是利用句柄来检测的。
- 句柄检测的原理是用CreateFile或_lopen函数试图获取一些驱动程序的句柄，如果成功则说明相应工具驻留在内存。
- 当CreateFile或_lopen函数用在句柄检测时，返回不存在这些句柄，否则很多壳在检测到这些句柄后会结束进程，停止脱壳。

4.5.4 PEB仿真

- 进程环境块(Process Environment Block)
- 存放有当前进程是否被调试的信息
- 脱壳过程中更换程序基址。
- 壳使用PEB来anti_dump

4.5.5 仿真dll加载分析

- 有些壳通过自身寻找kernel32.dll位置，来导出API，这里为了支持脱这些壳，对这些壳必须仿真加载kernel32.dll，供外壳程序使用，导出API
- 有些壳不但kernel32.dll中的函数自己分析导出表导出函数，对其它一些dll也自己分析导出表导出函数。

4.6 dump技术

- 寻找IAT：
  - 搜索call [xxxxxxx]，jmp[xxxxxxxx]来发现IAT的位置；
  - 在脱壳过程中从执行GetProcAddress后几条指令回写函数地址发现IAT
- IAT预处理：
  - 对于加密的IAT表进行解密
- 处理PE头：
  - 部分壳破坏PE头来anti-dump

- 重建导入表：
  - 部分壳脱后是不需要重建导入表
- PE文件的后续处理：
  - 修改PE头SizeOfHeaders
  - 修正SizeOfImage
  - 重设AddressOfEntryPoint
  - 修正节

    PointerToRawData = VirtualAddress;

    SizeOfRawData = VirtualSize;
- 额外数据的处理

五、虚拟机脱壳的限制

- 效率低于静态解密
- 仿真工作量大
- 被穿透的可能

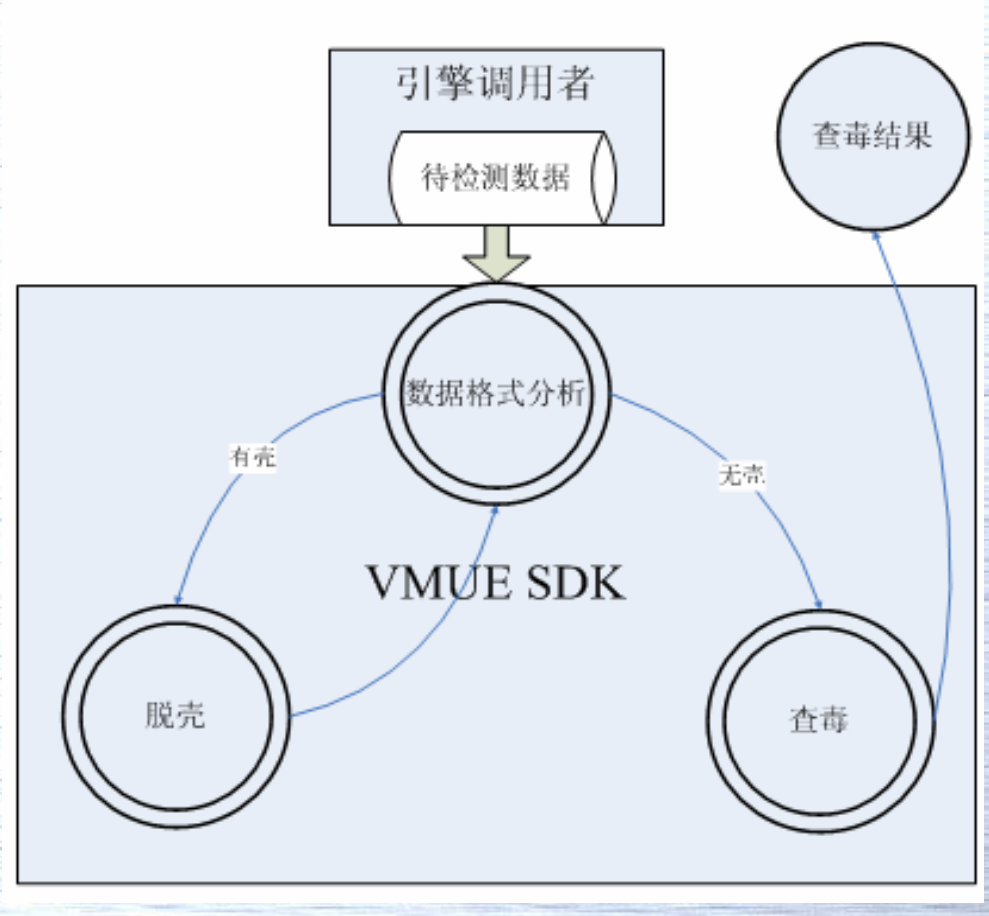
六、其他应用

- 加密变形
- 入口点模糊
  - 双入口点
- 未知壳

七、演示

- 我们的工作
  - 目前的脱壳支持能力
  - 准确解密和模拟跟踪方案
- VMUnpacker Demo

- VMUnpacker 引擎 SDK
  - 无需关心脱壳过程和脱壳方法
  - 支持将壳脱到文件和脱到内存缓冲区,并且直接返回OEP



VMUE SDK工作原理图

谢 谢!