



Linux下的Ring0层软件对抗

演讲嘉宾：李文韬(c4droid)

01

Linux下病毒感染类型分析

1

蠕虫病毒：大家都很熟悉的病毒，可以利用系统的漏洞悄无声息的传播和突然爆发。在Linux系统的蠕虫病毒与windows下的蠕虫病毒都差不多，都可以独立的运行，可以在计算机之间传播，在Linux系统下，文件是没有PE结构的，查杀此类病毒都比较容易。但是能彻底的摆脱掉蠕虫病毒是在目前看来是不行的。

2

ELF格式文件病毒：ELF文件格式病毒，最知名的病毒Lindose病毒是一个经典的代表，其运作过程为如果在系统中发现一个ELF文件时，就会查找匹配值，如果匹配的话会立即修改ELF的文件的入口点让其指向病毒代码部分，运行完病毒的代码后，再转向正常程序的代部分。

3

脚本病毒：本身危害性不是很大，一般来说也只是对本机造成破坏。其一般使用的脚本编写语言为shell等脚本语言，此类脚本语言的编写不是很难，但对计算机的破坏也变得比较相对容易。此类病毒目前的最大危害就是可以自动下载安装木马程序。一般的脚本病毒都与其他病毒分开进入用户的系统，传统的杀毒软件只能检测出病毒，并不能检测出脚本，对于这一类的病毒，在Linux系统上必须严格的控制root的权限，不能使用root权限去打开来源不明的脚本程序。



4

后门程序，一般是指可以绕过安全性控制而获取程序或系统访问权的程序。大多数后门程序都不会进行自我复制，它的作用主要是对用户计算机的信息进行收集，或者给hacker提供一个可以进入系统的方法。此类病毒目前在Linux系统下也非常活跃，一般可以利用计算机的服务系统去加载后门，也可以通过很多的方式注入系统的进程和文件中，还有一些甚至加载到内核中去实现。此类病毒非常隐蔽，一般手动查杀对于此类程序很难防御。随着现在加“壳”的技术逐渐发展，一些后门文件加上经过特殊处理的“壳”后，基本可以轻松的通过所有的杀毒软件的检测，在目前，对于这些带有特殊“壳”的软件我们几乎无能为力，只能借助一些行为监控、内存查看等专门工具，对其进行查杀，但效果很有限。

5

其他病毒：linux系统由于其系统进行过开源，在Linux系统下的病毒的种类数量相对windows来说要少得很多，但是需要注意的是，windows系统下的病毒可能会存放在Linux的文件系统中，但它们是不能在Linux系统上直接运行的，它们有可能借助Linux平台传递到网络中的其他windows系统中。

02

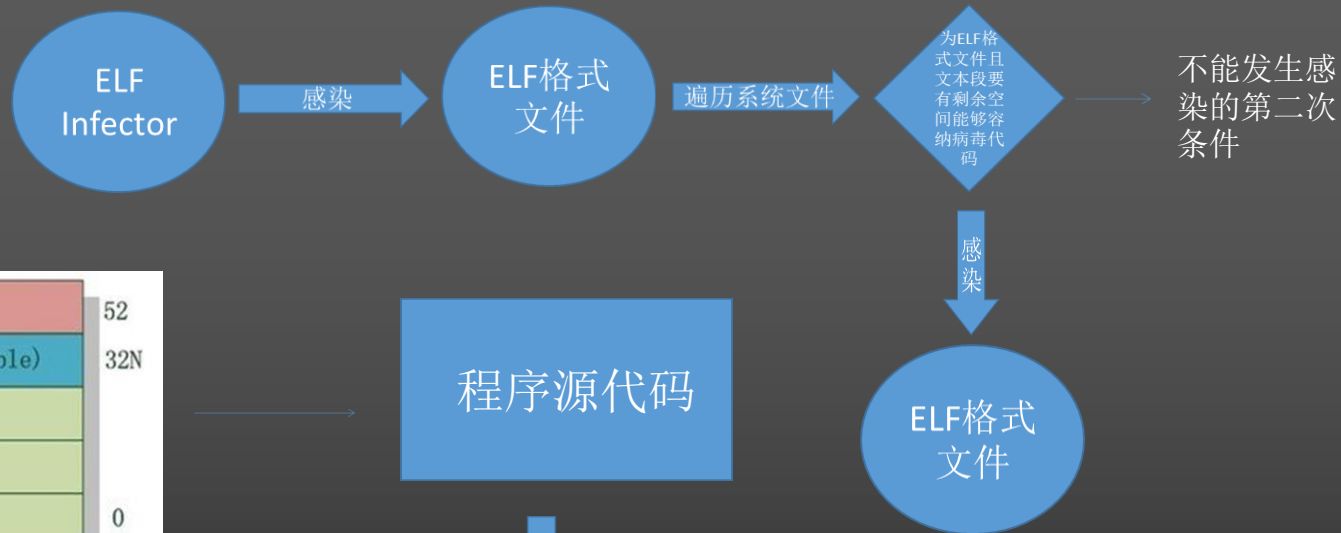
病毒的实现原理

原理：对于ELF文件的感染技术，我们需要一个ELF文件感染器，当然，也是可以在Linux系统中的内核去构造一个感染过程，方法多种多样，对

ELF Infection的过程，可以参考《UNIX ELF PARASITES AND VIRUS》，过程总结如下：

The final algorithm is using this information is.

- * Increase p_shoff by PAGE_SIZE in the ELF header
- * Patch the insertion code (parasite) to jump to the entry point (original)
- * Locate the text segment program header
- * Modify the entry point of the ELF header to point to the new code (p_vaddr + p_filesz)
- * Increase p_filesz by account for the new code (parasite)
- * Increase p_memsz to account for the new code (parasite)
- * For each phdr who's segment is after the insertion (text segment)
- * increase p_offset by PAGE_SIZE
- * For the last shdr in the text segment
- * increase sh_len by the parasite length
- * For each shdr who's section resides after the insertion
- * Increase sh_offset by PAGE_SIZE
- * Physically insert the new code (parasite) and pad to PAGE_SIZE, into the file - text segment p_offset + p_filesz (original)



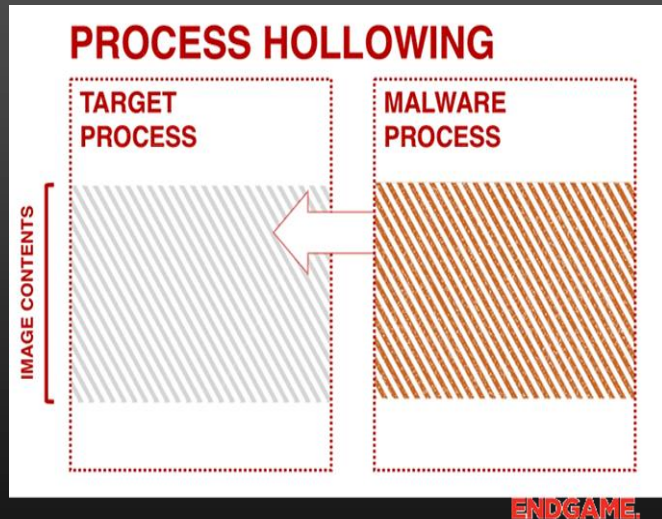
文件头 (ELF Header)	52
程序头表 (Program Header Table)	32N
代码段 (.text)	
数据段 (.data)	
bss段 (.bss)	0
段表字符串表 (.shstrtab)	
段表 (Section Header Table)	40N
符号表 (.symtab)	16N
字符串表 (.strtab)	
重定位表 (.rel.text)	8N
重定位表 (.rel.data)	8N

当然病毒也有很多的方式去植入病毒，但目前来说，在Linux系统下，能绕过安全检测的最直接的方法就是Process Hollowing技术。

那么，什么是Process Hollowing?

Process Hollowing是现代恶意软件经常常用的一种进程创建的技术，一般使用Process Hollowing技术创建出来的进程在使用任务管理器之类的工具进行查看时，进程看起来都是正常的，但事实并不是如此，在这些用肉眼看起来是正常的进程下，其实是包含着恶意代码。这种技术可以对正在运行的进程进行修改，在这个过程中，运行的程序不需要挂起进程，也不需要调用Windows API。

运行机制



03

对于Process Hollowing的内核检测思路

3

对于正常的Process Hollwing的内核检测思路

1、检测父进程与子进程的进程链(父子进程链)

对于父子进程，在程序运行时首先进入的是父进程，其次是子进程。例如，有两个可疑的Lsass.exe进程，但是这两个进程的父进程并不是winlogon.exe(在Vista系统前的系统)或winitit.exe(在Vista系统以及后的系统)。

2、比较PEB结构和VAD结构

PEB结构一般位于该进程的内存中，这结构中保存着进程的磁盘绝对路径，以及内存加载的基质。VAD结构则是位于内核中的内存中，包含着进程连续虚拟内存空间的分配信息，如果进程加载了可执行文件，则节点中会记录有关可执行文件的起始地址、结束地址以及完整的路径信息。

3、检测可疑的内存保护属性

一般的恶意软件都会将要Process Hollwing的程序的内存保护属性设为可读写执行。任何可执行文件正常加载到内存后都会拥有PAG_EXECUTE_WRITECOPY的内存保护属性。

3

对于正常的Process Hollwing的内核检测思路

1、对于申请不同的地址，修改过PEB

有些恶意程序是以挂起创建系统进程，并让其加载到指定的地址，之后再读取系统进程的PEB.ImageBaseAddress得到进程加载基址，之后再进行内存的释放，对系统进程申请读写权限的内存，申请结束后拷贝可执行程序到申请的内存中，拷贝结束修改PEB.ImageBaseAddress的字段值，其值为刚刚申请的内存地址。

2、不会将进程挖空

PEB有些恶意程序以挂起创建系统进程，然后再0x00400000的地址上申请一块RWX的权限内存，这些不会将进程挖空的恶意程序没有执行的操作为:对系统进程基址上的内存0x01000000取消映射，而是直接保存了进程上的基址内存。之后再把PE文件写入到目标系统进程新申请的内存地址中，恶意程序修改了被挂起的系统进程的CONTEXT.EAX的值，让其指向注入到0x00400000的PE入口点，最后对线程进行恢复让其执行。这些样本并没有对Process Hollwing的进程进行挖空，只是修改了PEB，可以达到欺骗的效果。

3

对于正常的Process Hollowing的内核检测思路

3、修改入口点地址(挖空进程)

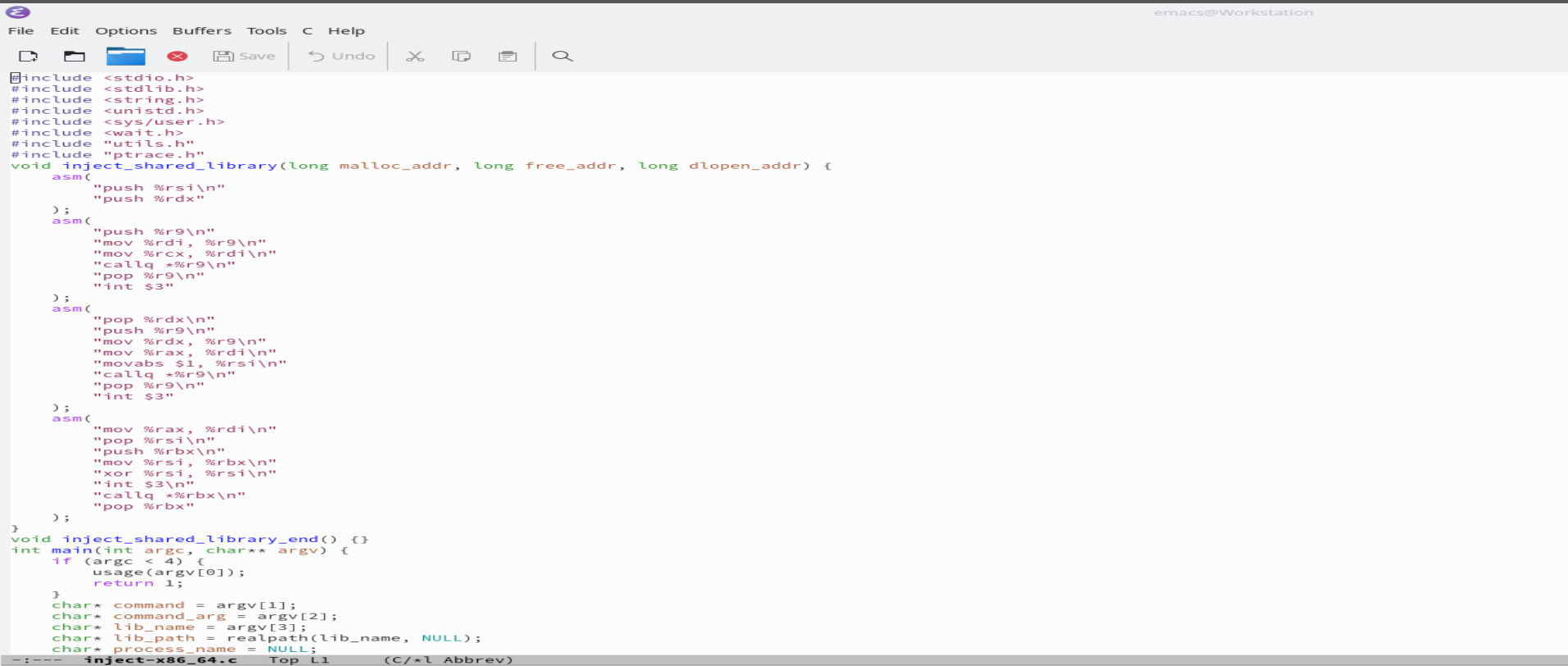
操作与前面的差不多相同，然后与上面相反的是，这类恶意软件并没有使用VirtualAllocEx和WriteProcessMemory这类敏感的函数，而是在自己的进程中创建了一个section，然后再将shellcode写入到创建的section中，使用NtMapViewOfSection函数将RWX的权限将这块内存映射到系统的进程中。此时系统中的程序会申请一块内存来保存映射的数据，当数据被保存结束时，恶意程序在自己的进程中创建第二个section，最后再把系统进程的内容拷贝到第二个section中，修改拷贝后内容的入口处前7个字节，最后取消映射把加载基址上的内存释放掉。

4、修改内存保护为PAGE_EXECUTE_WRITECOPY

一些精心经过处理的恶意程序，挂起创建的系统进程不同。该系统进程在系统启动时，做完自己的事情后，自动退出。这样就没有了父进程。以此同时，将内存属性设为PAGE_EXECUTE_WRITECOPY，操作与挖空进程相同。

代码实操&技术实现

现场展示



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/user.h>
#include <wait.h>
#include "utils.h"
#include "ptrace.h"
void inject_shared_library(long malloc_addr, long free_addr, long dlopen_addr) {
    asm(
        "push %rsi\n"
        "push %rdx\n"
    );
    asm(
        "push %r9\n"
        "mov %rdi, %r9\n"
        "mov %rcx, %rdi\n"
        "callq *%r9\n"
        "pop %r9\n"
        "int $3"
    );
    asm(
        "pop %rdx\n"
        "push %r9\n"
        "mov %rdx, %r9\n"
        "mov %rax, %rdi\n"
        "movabs $1, %rsi\n"
        "callq *%r9\n"
        "pop %r9\n"
        "int $3"
    );
    asm(
        "mov %rax, %rdi\n"
        "pop %rsi\n"
        "push %rbx\n"
        "mov %rsi, %rbx\n"
        "xor %rsi, %rsi\n"
        "int $3\n"
        "callq *%rbx\n"
        "pop %rbx"
    );
}
void inject_shared_library_end() {}
int main(int argc, char** argv) {
    if (argc < 4) {
        usage(argv[0]);
        return 1;
    }
    char* command = argv[1];
    char* command_arg = argv[2];
    char* lib_name = argv[3];
    char* lib_path = realpath(lib_name, NULL);
    char* process_name = NULL;
    inject_x86_64.c Top L1 (C/*1 Abbrev)
```



感谢观看 | THANK YOU