基于 Wine 的 Windows 安全机制模拟及沙箱系统实现

刁铭智[1] 周 渊[2] 李舟军[1] 赵宇飞[1]
(北京航空航天大学计算机学院 北京 100191)[1] (国家计算机网络应急技术处理协调中心 北京 100029)[2]

摘 要 基于开源软件 Wine 模拟了 ASLR 和 UAC Virtualization 两种 Windows 安全机制,使得 Wine 环境更接近真实的操作系统且更加安全,并利用 wineserver 机制初步实现了动态行为检测功能,同时借助 Wine 自身的.wine 目录作为样本的运行环境,最终形成了一个较真实的沙箱系统。实验结果表明,该沙箱系统具备了 ASLR 和 UAC Virtualization两种安全机制的基本特征。与其他沙箱系统相比,该系统不仅可以对未知样本进行有效的行为检测,而且具有占用资源少、隔离性强、回滚速度快等突出特点,因此能够很好地满足批量部署和运行的需要。

关键词 Wine,地址空间布局随机化,用户账户控制虚拟化,动态行为检测,沙箱
中图法分类号 TP311.56 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.11.037

Windows Security Mechanisms Simulation and Sandbox System Implementation Based on Wine

DIAO Ming-zhi[1] ZHOU Yuan[2] LI Zhou-jun[1] ZHAO Yu-fei[1]
(School of Computer Science and Engineering, Beihang University, Beijing 100191, China)[1]
(National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing 100029, China)[2]

Abstract We simulated two Windows security mechanisms, adress space layout randomization (ASLR) and user account control (UAC) Virtualization, based on open source software Wine. The two mechanisms make the Wine's environment closer to the real operating system and safer. Based on the two security mechanisms, we further presented a relatively real sandbox system, which employs the wineserver mechanism and utilizes the .wine directory of Wine as the running environment for samples to detect the dynamic behavior. The experimental results show that the proposed sandbox system presents the basic characteristics of ASLR and UAC Virtualization. Compared with other sandboxes, our proposed sandbox system can not only effectively detect behaviors of unknown samples, but also has features like low resource utilization, good isolation and fast status rollback, which make it meet the requirements of batch deployment and operation.

Keywords Wine, ASLR, UAC Virtualization, Dynamic behavior detection, Sandbox

1 引言

虚拟机是一种分析与检测恶意代码的有效工具,可以基于其构建沙箱系统以运行未知代码但不损害真实系统。当前针对 Windows 病毒检测的成熟在线沙箱系统主要包括 Cuckoo[1]、Comodo[2]、JOESandbox[3]、魔盾[4]以及金山火眼[5]等,此外还存在以 TxBox[6]为代表的加入特定规则的沙箱、多安全机制 Linux 应用沙箱[7]、LSM 沙箱[8]以及以 Sandboxie[9]为代表的入侵防御沙箱系统。多安全机制 Linux 应用沙箱虽然拥有良好的隔离性和安全性,但部署和回滚速度慢;LSM 沙箱虽然消耗资源少,但需要修改内核,导致易用性差;成熟的在线沙箱系统的检测效果较好,但由于其底层是基于 VMware, VirtualBox 或 KVM 等虚拟机来构建的,因此部署和运行这些沙箱系统均会占用大量资源,并且虚拟机并非真实的物理环境,恶意代码通常会探测这些差异特征[10-11]来隐藏其自身的真实行为[12];以 Sand-boxie 为代表的轻量级沙箱由于未开源,因此不能根据业务需求实现相应的动态行为检测功能。

为对抗恶意代码的反调试反虚拟化技术[13-15],并减少沙箱系统的资源消耗,需研究轻量级的恶意代码运行检测环境,并使其尽可能接近真实的操作系统环境。本文基于开源软件 Wine 模拟了 ASLR 和 UAC Virtualization 两种 Windows 安全机制,使 Wine 的内部更接近当前主流的 Windows 7 操作系统且运行环境更加安全;并借助 wineserver 服务进程,设计并实现了具备动态行为检测功能的沙箱系统。相对于现有沙箱系统,该系统具有占用资源少、隔离性强、回滚速度快等突

到稿日期:2016-10-27 返修日期:2017-01-05 本文受国家自然科学基金(61170189,61370126,61202239),国家"八六三"高技术研究发展计划项目(2015AA016004),教育部博士点基金(20111102130003)资助。

刁铭智(1990-),男,硕士生,主要研究方向为信息安全,E-mail:dmz0907@163.com;**周 渊**(1972-),博士,高级工程师,主要研究方向为网络与信息安全;**李舟军**(1963-),男,教授,博士生导师,主要研究方向为网络与信息安全、数据挖掘与智能信息处理技术;**赵宇飞**(1990-),男,博士生,主要研究方向为网络与信息安全。

出特点，能够很好地适应批量部署和运行的需求。最终测试结果表明，该系统达到了预期的效果。

2　Wine 的体系结构

Wine 是 Windows 应用软件与 Linux 内核之间的适配层，体现为一个 wineserver 服务进程和一组动态链接库的集合[16]。Wine 的用户图形界面依赖于 X11，由动态链接库 x11drv 和 X11 服务进程构成。这样，对于每个运行的 Windows 应用程序，系统中都至少存在 3 个进程与之相关：应用进程本身、Wine server 服务进程和 X11 服务进程。

Wine 的映像装入和启动过程十分复杂。实际上，Windows应用并非由 Linux 的 Shell 直接启动，而是通过一条类似于“wine 应用程序名”的命令，由 shell 启动 Wine 的装入工具 wine-preloader，再由 wine 间接地装入并启动具体应用程序。整个过程要依次经由 wine，wine-preloader，然后再次回到 wine 的接力才能完成。在整个过程中，wine-preloader，wine 及目标映像均在同一进程中活动。

图 1 给出了 Wine 的体系结构。

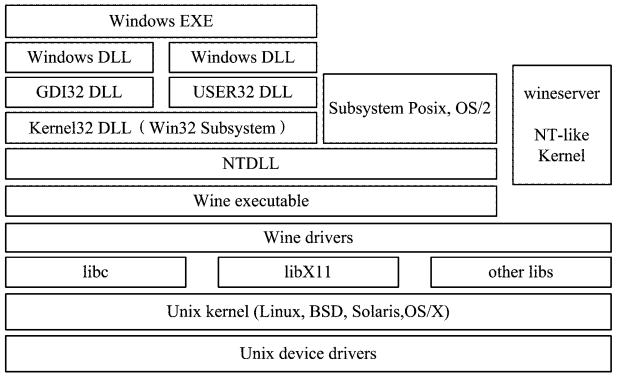

图 1　Wine 的体系结构

所有对动态链接库的调用均在应用进程本身的上下文中进行。当需要 wineserver 进程的服务或者通过 Wine 间接提供的其他服务时，应用进程经由 Wine 所提供的各种动态链接库逐层向下调用。在 Wine 的内部，每个应用进程既可通过 socket 和 pipe 与 wineserver 进程进行通信，以接受服务进程的管理与协调，也可经由另一个动态链接库 x11drv 通过别的 socket 与 X11 服务进程通信，向其发送图形操作请求并接收键盘和鼠标的输入。Wineserver 进程的主要作用包括：提供 Windows 进程间的通信与同步手段、Windows 进程与线程的管理、注册表服务等。它在 Linux 内核之外基本填平了 Linux 与 Windows 内核间的差异，使得应用程序可以顺利地运行在 Wine 上。X11 服务进程主要负责图形的显示以及键盘、鼠标的输入等。

3　ASLR 机制的分析及模拟

3.1　ASLR 机制及其特征

地址空间布局随机化（ASLR）是一种针对缓冲区溢出的安全保护技术，其通过对堆、栈、共享库等线性区域布局的随机化来增加攻击者预测目的地址的难度，以防止攻击者直接定位攻击代码的位置，达到阻止缓冲区溢出攻击的目的[17-18]。ASLR 机制可以有效地降低缓冲区溢出攻击的成功率，如今 Linux，FreeBSD，Windows 等主流操作系统都采用了该技术。

下文对 ASLR 机制进行简单演示，该段程序的目的是输出 Kernel32. dll 和 MSVCR120. dll 的加载地址、LoadLibrary() 和 system()两个函数的入口地址以及应用程序本身的函数 func()的入口地址。若需在 Visual Studio 中开启 ASLR 机制，可以通过项目属性→编译属性→链接器→高级来找到随机化基地址并更改选项。

本文在 Visual Studio 2013 环境中，使用 Win32 命令行模式来编译链接演示程序。当演示程序未开启 ASLR 功能时在 Windows 7 下运行，重启系统前后的输出结果如图 2 所示。

```
Kernel32 loaded at 753B0000
Address of LoadLibrary is 753C48FB
MSVCR120.dll loaded at 70540000
Address of system func is 705C08C2
Address of func is 00401000
```

```
Kernel32 loaded at 75D70000
Address of LoadLibrary is 75D848FB
MSVCR120.dll loaded at 724A0000
Address of system func is 725208C2
Address of func is 00401000
```

图 2　未开启 ASLR 机制的运行效果

若演示程序开启了 ASLR 功能，则在 Windows 7 下运行，重启系统前后的输出结果如图 3 所示。

```
Kernel32 loaded at 753B0000
Address of LoadLibrary is 753C48FB
MSVCR120.dll loaded at 70540000
Address of system func is 705C08C2
Address of func is 000F1000
```

```
Kernel32 loaded at 75D70000
Address of LoadLibrary is 75D848FB
MSVCR120.dll loaded at 703B0000
Address of system func is 704308C2
Address of func is 009F1000
```

图 3　开启 ASLR 机制的运行效果

由演示程序的运行结果可知，只有开启 ASLR 安全机制，func()函数的入口地址才会发生变化。但 Kernel32. dll 和 MSVCR120. dll 这两个动态链接库因默认开启 ASLR 安全机制而使加载地址发生了变化。

3.2　ASLR 机制的原理分析

本文从 LoadLibrary 函数开始对 ASLR 安全机制进行分析，跟踪 dll 的加载过程[19]，并尝试找出选择模块加载地址的方法。本文通过使用 Windbg 和 IDA pro 来对 LoadLibrary 函数进行跟踪，得到的调用序列依次为：LoadLibrary，LoadLibrary Ex，LdrLoad Dll，LdrpLoadDll，LdrpFindOrMapDll，NtCreate Section，MmCreateSection，MiReLocateImageAg ain 以及 MiSelectImageBase。最后调用的 MiSelect ImageBase 函数是实现 ASLR 安全机制的关键函数，操作系统在初始化时确定了一个随机变量（记作 ImageBias），并使用该变量在虚拟地址空间中占据 ImageBias * 64kB 大小的区域。当模块被加载时，函数会在剩余的区域中按照模块加载顺序由高地址开始依次寻找可加载的空闲空间，以此来决定模块的加载地址。

关键部分的反汇编代码如图 4 所示。

```
.text:0047FC89                                  ; MiSelectImageBase(x,x)+59↑j
.text:0047FC89                rdtsc
.text:0047FC8B                push    0
.text:0047FC8D                shrd    eax, edx, 4
.text:0047FC91                push    0FEh
.text:0047FC96                shr     edx, 4
.text:0047FC99                push    edx
.text:0047FC9A                push    eax
.text:0047FC9B                call    __aullrem
```

图 4 ASLR 机制的关键汇编代码

ImageBias 的值由 rdtsc 指令决定，该指令的作用是将处理器的时间标签计数器的当前值加载到 edx:eax 寄存器中。时间标签计数器包含在 64 位 MSR 中，MSR 的高 32 位加载到 edx 寄存器中，低 32 位加载到 eax 寄存器中。处理器周期性地递增时间标签计数器 MSR，并在复位时将 MSR 重新设置为 0。ImageBias 值的随机性以及模块加载顺序的不确定性，保证了每次重启系统后 DLL 文件的加载地址的随机变化。应用程序的随机加载机制与 dll 加载类似。

3.3 ASLR 机制在 Wine 中的模拟

在 Windows 的 4G 虚拟地址空间中，对于应用程序而言仅低 2G 地址是可用的，在 Windows NT 中剩余的高 2G 地址由操作系统和内核使用(从 0x80000000 开始)。而在 Windows 9x 中，事实上仅最高的 1G 地址空间(从 0xc0000000 开始)被内核使用，2G 到 3G 的地址空间是一个共享区域，该区域被用来加载系统 DLL 和映射文件。此外，Windows NT 和 Windows 9x 的低 2G 地址空间都是可以被应用程序使用的。Windows 和 Wine 的地址空间布局情况如表 1 所列。

表 1 Windows 和 Wine 的地址空间布局情况

Address	Windows 9x	Windows NT	Wine
00000000-7fffffff	User	User	User
80000000-bfffffff	Shared	User	User
c0000000-ffffffff	Kernel	Kernel	Kernel

真实的 Windows 操作系统会根据 PE 结构中的相关标志位来判断是否启用 ASLR 机制，若 PE 结构 Optionnal Header 中偏移量 70 位置的对应数值为 0x0040，则代表该模块在加载时可以被重新定位。本文编写了一个 python 脚本程序来对即将加载的 Windows 应用程序进行检测，一旦发现该程序中的 OPTIONAL_HEADER. DllCharacteristics 标志位以 40 结尾，即设置一个新的环境变量 ASLR，地址随机化操作只在检测到该环境变量后才会进行，这与真实操作系统的行为基本一致。

基于 Wine 近似实现 ASLR 机制的伪代码如下：

```
begin
  用编写的 python 脚本检测样本指定标志位
  if 标志位以 0x0040 结尾 then
    设置新的环境变量
    生成随机数
    if wine 已经初始化并已将各种 dll 的目录路径设置完毕 then
      在装入第一个动态链接库 ntdll 之前先行占据一定地址空间
    end if
    if 在线程初始化部分即将分配首个 PEB 和 TEB then
      抢先占据一定地址空间
    end if
    if 在线程初始化部分即将分配 process_heap then
      先行占据一定地址空间并使其淹没 0x400000 位置
    end if
  else
    忽略 ASLR 实现部分并按原有的代码执行
  end if
end
```

为测试对 ASLR 机制的模拟情况，本文编写了一个带有图形界面的测试程序，该程序可以展示应用程序自身、Kernel32. dll 以及 ntdll. dll 的加载地址。利用该程序，在 Wine 中模拟 ASLR 机制之前的模块加载基址情况如图 5 所示。

图 5 原始 Wine 中的模块加载地址

在 Wine 中近似实现 ASLR 机制之后的模块加载基址情况如图 6 所示。

图 6 实现 ASLR 后 Wine 中的模块加载地址

从测试程序的运行结果可知，应用程序本身以及两个主要的动态链接库在原始 Wine 中的加载地址固定不变，但在模拟 ASLR 之后其加载地址发生了显著变化，表明 ASLR 在 Wine 中的模拟是比较成功的。

4 UAC Virtualization 机制的分析及模拟

4.1 UAC Virtualization 机制及其特征

在早期的 Windows 操作系统中，例如 XP，NT，95 等，以管理员身份安装应用程序后，这些应用程序可能会任意读或写系统文件和注册表键值；然而，如果以普通用户身份运行相同的应用程序则可能导致运行时错误，其主要原因在于应用程序有写入受保护系统文件、目录或注册表的行为，这些行为需要用户以管理员身份运行或将该用户添加到管理组。显然，将用户账户添加到管理员所在的组来正常运行应用程序并非一个好的解决方法[20]。

由于许多应用程序需要对操作系统中的关键文件进行操作且用户必须运行它们，因此从 Vista 开始，Windows 操作系统采取了一种不同的方式来修复该问题。UAC 将应用程序的部分操作虚拟化，从而使得用户能够以标准用户身份正常运行原本需要管理员权限的应用程序。文件系统和注册表的虚拟化并非是系统范围的，而是一个有限区域的虚拟化，这些区域对于操作系统至关重要。受 UAC 虚拟化机制影响的区域[21]有：1)/Program Files 目录和其子目录；2)位于 64 位系统上的/Program Files(x86)目录；3)/Windows 目录和其子目录，包括 system32 目录；4)/User/%AllUsersProfile%/ProgramData 目录；5)/Documents and Settings 目录；6)HKLM\

Software 注册表路径。

4.2 UAC Virtualization 机制的原理分析

Windows 7 通过将用户对关键文件或注册表的写操作重定向到其他路径，来实现以普通用户身份运行的应用程序的兼容性。例如，如果一个普通用户运行了一个应用程序，该程序试图写 C:/Program Files/National Instruments/Settings.ini 文件，由于 UAC 虚拟化的原因，该操作将会被重定向到 C:/User/Username/AppData/Local/Virtual Store/Program-Files/National Instruments/Settings.ini 中。如果应用程序试图写注册表路径 HKEY_LOCAL_MACHINE\Software\National Instruments\，该操作最终将被重定向到 HKEY_CURRENT _USER\Software\Classes\VirtualStore\MACHINE\Software\National Instruments 或 HKEY_USERS\UserSID_Classes\VirtualStore\Machine\Software\National Instruments 路径下。

图 7 给出了 UAC 虚拟化的两个组成部分（文件虚拟化和注册表虚拟化）的原理。

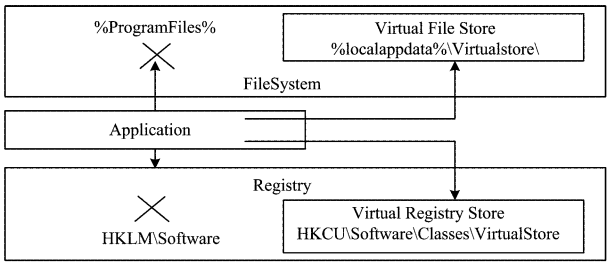

图 7　UAC 虚拟化的原理

4.3 UAC Virtualization 机制在 Wine 中的模拟

本文以 UAC 虚拟化原理为基础模拟了系统盘路径下的文件虚拟化以及 HKLM\Software 路径下的注册表虚拟化。由于在 Windows 程序正式运行之前，wine 也会进行一些文件及注册表等的操作，此时进行重定向将导致段错误，因此重定向的工作需要在应用程序正式启动之后进行。本文首先检测加载模块位置是否在 0x400000 之上的不远处，若是则确定即将加载的模块为 Windows 应用程序，并设置一个 UAC 环境变量。UAC 虚拟化操作在 UAC 环境变量被定义后才开始进行。

UAC Virtualization 机制的文件部分在 Wine 中模拟的伪代码如下：

```
begin
  if .wine 环境需要更新 then
    拷贝新添加的系统盘目录以及用户目录到.wine
  end if
  do while 进入模块加载环节
      if 模块加载地址在 0x400000 以上的不远处 then
          设置新的环境变量
          break
      end if
  end while
  if 文件操作路径以 C 开头 and 指定环境变量存在 then
      将路径参数转换成 Linux 路径下的统一格式
      if 路径以 C://Windows 开头 then
        将路径除 C://以外的部分拼接在虚拟化路径之后
      end if
      if 路径以 C://Program Files 开头 then
        将路径除 C://以外的部分拼接在虚拟化路径之后
      end if
      if 路径以 C://开头且不再包含路径分隔符 then
        将路径除 C://以外的部分拼接在虚拟化路径之后
      end if
      if 参数是打开已存在文件 then
      复制当前文件至拼接后的新路径
      end if
      将处理后的路径转换为 Windows 下的形式并赋值给原路径
  else
      在原有路径下正常执行
  end if
end
```

其中，文件虚拟化路径为 C:/User/Username/AppData/Local/ VirtualStore/。

UAC Virtualization 机制的注册表部分在 Wine 中模拟的伪代码如下：

```
begin
  if .wine 环境需要更新 then
      执行 regedit 读取配置文件以添加注册表虚拟路径
  end if
  do while 进入模块加载环节
      if 模块加载地址在 0x400000 以上的不远处 then
        设置新的环境变量
        break
      end if
  end while
  if 注册表操作路径以 HKLM\Software 开头 and 环境变量存在 then
      将除 HKLM\以外的部分拼接在注册表虚拟化路径之后
  else
      以原有路径继续执行
  end if
end
```

其中，注册表虚拟化路径为 HKEY_CURRENT_USER \Software\Classes\VirtualStore\MACHINE\。

本文针对全部路径均编写了对应的测试用例，测试结果表明所有对系统敏感路径的操作都被重定向到相应的虚拟路径下。因此，当前的 Wine 已经具备 UAC 虚拟化机制的主要特征。

5 沙箱系统实现

本文沙箱后端采用 python 和 C 语言编写，实现了包括动态行为记录、动态行为检测、文件 PE 结构检测、字符串检测、反调试反虚拟机特征检测、子进程 dump 以及反病毒引擎扫描等功能，前端采用 bootstrap 框架，最终构成了一个完整的沙箱系统。下文详细介绍主要功能的实现方法。

5.1 wineserver 机制的介绍及动态行为记录的功能实现

wineserver 是 Wine 中最为复杂的组件，它在 Linux 内核之外消除了 Linux 和 Windows 内核间的差异，使得 Wine 上的 Windows 应用程序尽可能顺利地运行。客户进程向服务进程传递信息是通过 Wine 内部的 wine_server_add_data 和 get_req_data 函数来实现的。Wine 中的每一个线程都拥有自己的 request 结构体，该结构体与 wineserver 共享。当一个线程需要同步或与其他线程通信时，其先在 request 中填写相应的信息，然后通过 socket 套接字发送一条命令代码，wineserver 酌情处理收到的命令，处理的同时客户线程阻塞等待 reply 结构体的到来。

在某种意义上可认为 wineserver 是在为 Windows 应用程序提供远程过程调用(RPC)，即跨进程的系统调用。从如下的过程调用表(函数指针数组 req_handlers[])可以看出 wineserver 的作用。

```
static const req_handler
req_handlers[REQ_NB_REQUESTS]=
{
    (req_handler)req_new_process,
    (req_handler)req_new_thread,
    (req_handler)req_terminate_process,
      ……
};
```

wineserver 服务进程对于进程、线程间的通信等的管理与服务与原本 Linux 内核提供的管理并不冲突，这些功能是对内核的补充。wineserver 存在的意义在于在 Linux 内核提供的功能的基础上建立对 Windows 对象的管理。本文利用 wineserver 服务进程的 request/reply 机制来添加 req_handlers 处理函数，以记录样本运行时调用的重要 API 信息[22-24]并输出到指定目录下，所记录的 API 情况如表 2 所列。

表 2 记录的 API 类别及个数

类别	API 个数	类别	API 个数
OLE	5	System	12
Filesystem	31	Misc	5
Resource	5	Iexplore	8
Mutex	2	Network	17
Registry	11	Process	16
Services	12	Time	5
Systeminfo	7	Thread	11
Crypto	8	VirtualMemory	7
Window	6	Socket	24
Cert	9	Hooks	2

在 Wine 中创建新的 request/reply 时主要需进行下列操作：声明并定义新的结构体、添加新的请求服务处理函数类型并为新的 request/reply 添加请求服务时的处理函数定义。其中，前两步可以由 Wine 内的工具脚本 make_request 自动生成，用户只需要添加请求服务时的处理函数定义即可。此外，在利用脚本生成 request/reply 结构体的声明与定义时，需要对检测 wine 版本部分进行代码注释，否则会导致版本检测处出错，进而使 Wine 终止运行。

本文新增了 4 个 DECL_HANDLER，其名称及作用如下。

1)report_init：在样本正式运行前创建相应的记录文件；

2)write_config：在样本正式运行前创建配置文件并填写公共配置信息；

3)hook_reg_call：在样本运行时记录其注册表行为；

4)hook_api_call：在样本运行时记录其文件、网络和进程线程等其他行为。

5.2 动态行为检测及沙箱系统的其他功能实现

应用程序在沙箱中运行一定时间后，会在指定目录下生成最初的行为报告，本文使用该报告中的 API 信息分别与构建的恶意行为特征库和主要行为的配置文件进行对比，从而得到样本的恶意行为及主要行为。

本文构建的特征库可以有效地识别主流的恶意行为，例如针对运行后的删除自身行为，特征库设置过滤条件，只关心“DeleteFileW”和“MoveFileWithProgressW”两条 API 的调用信息。若调用“MoveFileWithProgressW”，source 参数为样本自身的路径并且 dest 参数为空，则匹配样本运行后删除自身的特征；若调用“DeleteFileW”且 path 参数为样本自身的路径，则也匹配该特征。表 3 列出了特征库可以识别的部分恶意行为。

表 3 特征库可以识别的部分恶意行为

序号	恶意行为名称
1	拷贝自身到其他目录
2	运行后删除自身
3	从资源中释放文件到系统目录 system32 下
4	尝试延迟执行以拖延检测
5	提升系统权限
6	尝试修改代理设置
7	伪装系统进程
8	查找杀毒软件，对安全软件采取反制措施

除上述功能外，本沙箱系统还使用静态与动态特征相结合的方式来实现基本的反调试反虚拟机特征检测功能，结合 Virustotal[25]的公共 API 实现了反病毒引擎扫描功能。

6 系统测试

基于开源软件 Wine，本文首先模拟了 ASLR 和 UAC Virtualization 两种 Windows 安全机制，然后利用 wineserver 机制实现了动态行为检测功能；最终借助 Wine 自身的. wine 目录作为应用程序的运行环境，构建了一个较真实的沙箱系统 WSandbox，WSandbox 的界面如图 8 所示。下文将给出 WSandbox 与其他沙箱系统的对比测试结果。

图 8 WSandbox 沙箱界面

6.1 沙箱独立运行测试

本节对 WSandbox 与针对 Windows 系统的成熟在线沙

箱系统 JOESandbox、VirusTotal、金山火眼进行了对比测试，在 Ubuntu 14.04 LTS 版本系统中部署了 WSandbox。对比测试分为两部分：1）选取带有安全机制探测功能的恶意代码样本，以测试不同沙箱系统对该样本检测的有效性；2）选取一定数量的恶意代码样本，并分别在不同沙箱系统中运行，对这些样本运行过程中的文件行为、模块加载行为、进程线程行为、注册表行为以及网络行为进行统计与分析，以检验不同沙箱系统的检测粒度与效果。

第一部分测试主要针对近似实现的安全机制。本文选取的样本行为包括：释放文件到系统目录下、从自身的二进制镜像中读取数据以及打开自身等。但是在样本的起始处有一段查看自身加载地址的代码，若该地址与 0x400000 相同，将不执行后续行为而直接返回。由于样本 PE 结构 IMAGE_DLL_CHARACTERISTICS_DYNAMIC_B ASE标志存在，因此其在主流 Windows 操作系统中的加载位置不固定。该样本在 WSandbox 与金山火眼中的检测结果如图 9 所示。

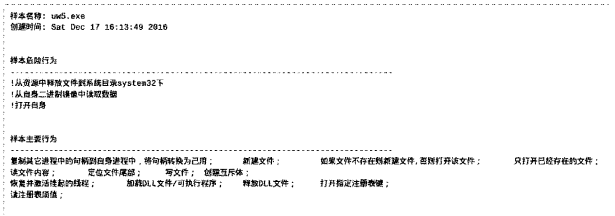

图 9　探测 ASLR 机制样本的运行结果

从图 9 可以看出，WSandbox 捕捉到了该样本的主要行为。但在金山火眼上运行样本时，并未输出任何行为。

根据报告的截图推测金山火眼应该是在 Windows XP 虚拟机中运行恶意代码样本，而当前主流操作系统早已变为 Windows 7，故要求样本运行环境支持 ASLR 等主流安全机制是十分必要的；否则，病毒样本可通过检测自身加载地址等特征，来判断其是否在沙箱等非真实环境或不满足要求的操作系统中运行，从而逃避行为检测。

图 10 给出一个随机样本在 WSandbox 中的运行结果。

图 10　随机恶意代码样本的运行结果

分别在魔盾和金山火眼中运行同样的样本，将动态行为点评给出的报告与 WSandbox 的报告进行对比发现其主要行为及危险行为均基本一致，因此可以初步认为 WSandbox 系统对样本行为的检测是较全面且准确的。

第二部分测试为对比测试，通过不同沙箱系统对恶意代码样本行为的检测情况验证 WSandbox 系统的检测有效性。本文在众多备选恶意代码样本中随机选取了 15 份，并分别在 JOESandbox、魔盾、金山火眼以及 WSandbox 中进行运行以获取行为报告，关注的恶意代码行为包括：文件行为、模块加载行为、进程线程行为、注册表行为以及网络行为。汇总后的样本行为数据如表 4 所列。

表 4　样本行为统计表

	Fireeye	JOESandbox	Maldun	WSandbox
File	105	107	110	110
Module	44	43	49	47
Process	66	62	71	67
Registry	125	123	127	126
Network	105	108	110	102

将实验中的每类行为的最大值作为假定总体行为空间 Max，将每种沙箱系统各个行为的数据作为单个行为空间 Single，定义 Single 与 Max 的比值乘以 100% 为该行为在该沙箱系统中的有效检测指标 Ed，根据此公式及上表中各个行为的统计情况进行计算，从而得到有效检测指标 Ed 的数据。Ed 的对比柱状统计图如图 11 所示。

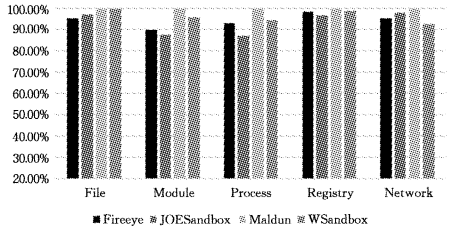

图 11　各沙箱系统有效检测指标的对比图

从图 11 可看出，本文所关注的文件行为、模块加载行为、进程线程行为、注册表行为以及网络行为，在不同的沙箱系统中的统计结果相差不大。WSandbox 的有效检测指标与当今主流成熟沙箱系统基本持平，因此可认为本文实现的沙箱系统 WSandbox 可有效地进行行为检测。

6.2　沙箱批量部署测试

本文运行沙箱批量部署测试的服务器配置如下：硬盘为 1TB；内存为 32GB；CPU 为 Intel(R) Xeon(R) CPU X5650 @ 2.67GHz，服务器共 2 个 CPU，且每个 CPU 拥有 6 个核心和 24 线程。

WSandbox 的批量部署及运行情况如图 12 所示。

图 12　WSandbox 沙箱批量部署截图

本次测试在上述配置的服务器上部署了共计 441 个运行

环境。由于本文根据真实系统改造后的.wine 目录中采取了链接技术,因此每个.wine 环境只有 15MB,大幅减少了运行环境的资源消耗,准备阶段只需要 55s 即可完成。若设置样本超时时间为 10min,则运行及生成行为报告的总时间约为 14min。

对比以 Vmware 或 KVM 等虚拟机为基础的沙箱系统,WSandbox 仅在运行应用程序时占用一小部分内存,而虚拟机需要一直开启并占用宿主机大量资源。若使用虚拟机作为沙箱,则普通的 4G 内存 PC 机仅能够运行 Windows 7 虚拟机 2~3 台,而本文基于 Wine 开发的沙箱 WSandbox 可以在普通服务器中批量部署上百个运行环境。

与当前主流的轻量级沙箱系统(如采用虚拟技术的 Sandboxie、Bufferzone 以及采用策略限制的 Geswall)进行对比,WSandbox 在空载及运行同一测试程序时的内存使用情况如表 5 所列。

表 5 内存使用情况统计表/MB

沙箱名称	Sandboxie	Bufferzone	Geswall	WSandbox
空载	5	12	6	6
运行	10	13	8	10

在空载及运行程序时,WSandbox 的内存使用情况在所比较的轻量级沙箱中处于中等水平,因此以 Wine 为基础构建的 WSandbox 具有资源消耗少的特点。

6.3 沙箱的隔离性及回滚速度

Wine 实现了 Windows 环境仿真。通过 Wine 启动的 Windows 进程可以根据各自的启动命令获得不同的环境变量,其中 WINEPREFIX 设定了每个进程的运行根目录,该目录默认为 $HOME/.wine。所有 Windows 进程使用相同的 wineserver、注册表、共享内存以及配置文件。通过对不同的进程设置不同的 WINEPREFIX,即可运行一定数量并且相互独立的 Windows 进程(见图 11)。本文以普通用户身份进行操作,为每一个测试样本分配了各自的运行根目录,不同的目录间不会相互影响。因此,以 Wine 为基础构建的 WSandbox 具有隔离性强的特点。

由于每个运行在 Wine 上的 Windows 进程均以.wine 作为运行根目录,因此本文利用 git 进行运行环境的批量分配和恢复。运行根目录的批量分配命令如下:

1)在原始.wine 目录下依次执行“git init”“git add.”以及“git commit -m”;

2)在 shell 脚本中循环执行“git clone.wine 目录名 目标目录名”。

批量恢复运行根目录环境时,进入各个目录依次执行“git clean -fd”以及“git checkout -- *”命令即可。另一种恢复方法是直接删除 git clone 命令得到的目录,并在下一次运行时重新分配。在 i5 处理器、内存 1G 的电脑中使用上述方法时,10s 可以分配大约 40 个运行环境,批量恢复这些运行环境需要 6s 左右。因此,以 Wine 为基础构建的 WSandbox 具有回滚速度快的特点。

结束语 本文以开源软件 Wine 为基础,模拟了 ASLR 和 UAC Virtualization 两种 Windows 7 操作系统中的安全机制,并以 wineserver 机制为核心,初步实现了一个具备一定行为检测能力的沙箱系统。该系统结合模拟的安全机制使得其更接近真实 Windows 7 操作系统环境,并且由于使用 Wine 自带的.wine 目录作为应用程序的运行环境,因此具有资源消耗少、隔离性强以及回滚速度快等突出特点。但是该系统仍存在一些与真实操作系统的区别,可能被恶意代码探测识别。因此,如何进一步完善该系统将是我们未来的研究重点。

参 考 文 献

[1] GUARNIERI C,SCHLOESSER M,et al. Cuckoo Sandbox[EB/OL]. https://cuckoosandbox.org.

[2] Comodo Instant Malware Analysis[EB/OL]. http://cam-as.comodo.com.

[3] JOESandbox[EB/OL]. http://www.file-analyzer.net.

[4] Maldun[EB/OL]. https://www.maldun.com.

[5] Fireeye by Kingsoft[EB/OL]. http://fireeye.ijinshan.com.

[6] JANA S,PORTER D,SHMATIKOV V. TxBox:Building Secure,Efficient Sandboxes with System Transactions[C]// IEEE Conference on Security and Privacy. 2011:329-344.

[7] LI C,TU B B,et al. Design and Implementation of of Linux Application Sandbox Based on Multiple Security Mechanisms[J]. Journal of Integration Technology,2014,3(4):31-37. (in Chinese)
李晨,涂碧波,等. 基于多安全机制的 Linux 应用沙箱的设计与实现[J]. 集成技术,2014,3(4):31-37.

[8] CHENG X P,CHEN L J. Design and Implementation of Sandbox Module Based on LSM[J]. Computer & Digital Engineering,2014,42(8):1521-1525. (in Chinese)
程香鹏,陈莉君. 基于 LSM 的沙箱模块设计与实现[J]. 计算机与数字工程,2014,42(8):1521-1525.

[9] Sandboxie[EB/OL]. http://www.sandboxie.com.

[10] CHEN X,ANDERSEN J,MAO Z M,et al. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware[C]// IEEE Conference on Dependable Systems and Networks With FTCS and DCC(DSN). 2008:177-186.

[11] CARPENTER M,LISTON T,SLOUDIS E. Hiding virtualization from attackers and malware[J]. IEEE Security & Privacy,2007,5(3):62-65.

[12] LIU K,LU S,LIU C G. POSTER:Fingerprinting the Publicly Available Sandboxes[C]// Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. 2014:1469-1471.

[13] LINDORFER M,KOLBITSCH C,COMPARETTI P. Detecting Environment-Sensitive Malware[C]// Proceedings of 14th International Symposium,RAID. 2011:338-357.

[14] HSU C W,et al. Divergence Detector:A Fine-Grained Approach to Detecting VM-Awareness Malware[C]// IEEE International Conference on Software Security and Reliability. 2013:80-89.

[15] KATSUNARI Y,YOSHIHIKO H,TATSUNORI O,et al. Vulnerability in Public Malware Sandbox Analysis Systems[C]// Proceedings of 10th IEEE/IPSJ International Symposium. 2010:265-268.

(下转第 267 页)

由表 2 可知，若需攻击 $B_{5,3}$ 中所有的 $B_{3,1}$，则在节点故障模型下需要攻击至少 20 个节点；在边故障模型下，则需要攻击至少 20 条边。因此，$F_{3,1}(B_{5,3})=f_{B_{3,1}}(B_{5,3})=20$。

结束语　随着硬件制造工艺瓶颈的趋近，并行计算机系统特别是超级并行计算机系统的性能越来越依赖于其基础网络。当系统遭受攻击而发生节点故障或线路(边)故障时，被攻击方希望知道保护至少多少节点和线路才能保证还存在既定规模的子系统可以正常运行。另一方面，当攻击对手的系统时，希望能够用尽可能小的攻击代价，来使对手系统中既定规模的子系统损失殆尽。上述问题反映到系统的基础网络上，可以归结为子网排除问题。目前，子网排除问题得到完全解决的网络仅限于超立方网络。超立方网络的子网划分方式相对简洁，较易于构造子网节点排除割和边排除割，从而可计算出任意规模子网的子网节点排除数和子网边排除数。

本文通过构造子网节点排除割和边排除割的方案完全解决了 k 元 n 维冒泡排序网络。对于大多数典型的、具有潜在应用价值的著名网络，诸如 k 元 n 立方网络、Star 网络、Bubble-sort 网络、Arrangement 网络等，它们的子网划分方式不唯一。以不同的方式划分时，相同规模的若干子网往往具有公共节点或公共边，从而导致这些网络的子网节点排除割和边排除割的构造异常困难。这些网络的子网排除问题主要在于具体规模的子网节点排除割和边排除割的构造，以及任意规模子网的节点排除数和边排除数的上界优化问题。上述经典网络中任意规模子网的节点排除割和边排除割的构造是子网排除问题有待攻克的难点之一。

参 考 文 献

[1] LATIFI S. A study of fault tolerance in star graph [J]. Information Processing Letters, 2007, 102(5): 196-200.

[2] LATIFI S, SABERINIA E, WU X. Robustness of star graph network under link failure [J]. Information Sciences, 2008, 178(3): 802-806.

[3] WALKER D, LATIFI S. Improving bounds on link failure tolerance of the star graph [J]. Information Sciences, 2010, 180(13): 2571-2575.

[4] WANG S, YANG Y. Fault tolerance in bubble-sort graph networks [J]. Theoretical Computer Science, 2012, 421: 62-69.

[5] YANG Y X, WANG S Y. Recursive algorithm for k-cycle preclusion problem in k-ary n-cubes [J]. Journal of Computer Applications, 2013, 33(9): 2401-2403. (in Chinese)
杨玉星，王世英. k 元 n 立方网络的 k 圈排除问题的递归算法 [J]. 计算机应用，2013，33(9)：2401-2403.

[6] YANG Y, WANG S, LI J. Subnetwork preclusion for bubble-sort networks [J]. Information Processing Letters, 2015, 115(1): 817-821.

[7] WANG S, ZHANG G, FENG K. Fault tolerance in k-ary n-cube networks [J]. Theoretical Computer Science, 2012, 460: 34-41.

[8] WANG S, FENG K. Fault tolerance in the arrangement graphs [J]. Theoretical Computer Science, 2014, 533: 64-71.

[9] FENG K, WANG S, ZHANG G. Link failure tolerance in the arrangement graphs [J]. International Journal of Foundations of Computer Science, 2015, 26(2): 241-254.

[10] WANG M, YANG W, GUO Y, et al. Conditional fault tolerance in a class of Cayley graphs [J]. International Journal of Computer Mathematics, 2016, 93(1): 678-682.

[11] SHAWASH N. Relationships among popular interconnection networks and their common generalization [D]. Oakland, USA: Oakland University, 2008.

[12] CHENG E, LIPTÁK L, SHERMAN D. Matching preclusion for the (n,k)-bubble-sort graphs [J]. International Journal of Computer Mathematics, 2010, 87(11): 2408-2418.

[13] BONDY J A, MURTY U S R. Graph Theory [M]. New York: Springer Press, 2008: 624-626.

(上接第 252 页)

[16] Wine Develop Guide[EB/OL]. http://www.winehq.org/docs/winedev-guide/index.

[17] SHACHAM H, PAGE M, PFAFF B, et al. On the effectiveness of address-space randomization [C]// Proceedings of the 11th ACM Conference on Computer and Communications Security. 2004: 298-307.

[18] LI L, JUST J E, SEKAR R. Address-Space Randomization for Windows Systems [C]// Proceedings of Computer Security Applications Conference (ACSAC'06). 2006: 329-338.

[19] WHITEHOUSE O. An Analysis of Address Space Layout Randomization on Windows Vista [M]. Symantec Advanced Threat Research, 2007.

[20] Inside Windows Vista User Access Control [EB/OL]. https://technet.microsoft.com/zh-cn/magazine/2007.06.uac(en-us).aspx.

[21] UAC Windows7 Tutorial[EB/OL]. http://sourcedaddy.com/windows-7/users-accounts-and-uac.html.

[22] SAMI A, YADEGARI B, RAHIMI H, et al. Malware detection based on mining API calls [C]// Proceedings of the 2010 ACM Symposium on Applied Computing. 2010: 1020-1025.

[23] QIAO Y, et al. Analyzing Malware by Abstracting the Frequent Itemsets in API Call Sequences [C]// Proceedings of 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications. 2013: 265-270.

[24] FARUKI P, LAXMI V, VINOD P, et al. Behavioural detection with API call-grams to identify malicious PE files [C]// Proceedings of the First International Conference on Security of Internet of Things. 2012: 85-91.

[25] VirusTotal[EB/OL]. https://www.virustotal.com/en.