

针对恶意代码的连续内存镜像分析方法

李伟明¹, 邹德清¹, 孙国忠²

(1. 华中科技大学计算机学院, 湖北 武汉 430074;

2. 曙光信息产业有限公司, 北京 100080)

摘要: 为了更加全面地检测恶意代码的行为, 提出连续内存镜像分析技术。核心是在 QEMU 虚拟机中运行恶意代码样本, 获取样本运行时期连续增量的内存镜像, 然后按照时序解析为多个完整的内存镜像。在单个内存镜像分析的基础上, 对不同时刻内存镜像做对比分析。同时设计运用可视化工具 D3.js, 以图表的形式直观动态地展示系统运行过程中内存状态的变化。最后实现原型系统, 通过对 40 种恶意代码样本进行测试, 检测出的恶意代码行为数量在传统单镜像内存分析的基础上增加了 19.7%。

关键词: 恶意代码; 内存镜像; 对比分析; 数据可视化

中图分类号: TP309

文献标识码: A

doi: 10.11959/j.issn.2096-109x.2017.00144

Successive memory image analysis method for malicious codes

LI Wei-ming¹, ZOU De-qing¹, SUN Guo-zhong²

(1. School of Computer Science, Huazhong University of Science and Technology, Wuhan 430074, China;

2. Dawning Information Industry Co., Ltd., Beijing 100080, China)

Abstract: In order to detect the behavior of malicious code more comprehensively, the technology of continuous memory image analysis was proposed. The core idea was to run malicious code in QEMU virtual machine, to obtain the memory image of the continuous increment in the running period, and then to analyze the memory image of the base and increment as the memory image. On the basis of the analysis of a single memory image, different memory images were analysed comparatively. At the same time, the visualization tool D3.js was used to visually display the change of the memory state in the process of system operation. Finally, the prototype system was tested by 40 kinds of malicious code samples, and the number of malicious code behavior was increased by 19.7% than traditional single memory image.

Key words: malware, memory image, comparative analysis, data visualization

1 引言

《2014-2015 中国互联网络安全研究报告》^[1]指出, 2014 年猎豹移动安全实验室共截获计算机

病毒样本 3 587 万个, 平均每天有近 10 万个新的计算机病毒被捕获, 虽然这一数字较 2013 年已有所下降, 但病毒总数仍相当庞大。因此, 恶意代码的检测和防御至关重要。当前主要的恶意代码

收稿日期: 2016-08-28; 修回日期: 2016-10-23。通信作者: 李伟明, lwm@hust.edu.cn

基金项目: 国家自然科学基金资助项目 (No.61272072); 国家重点基础研究与发展计划基金资助项目 (“973” 计划) (No.2016YFB0200300)

Foundation Items: The National Natural Science Foundation of China (No.61272072), The National Basic Research Program of China (973 Program) (No.2016YFB0200300)

分析方法分为静态和动态 2 种。静态分析指的是在不运行恶意代码的前提下, 对恶意代码使用二进制分析、反汇编等一系列的分析方法; 动态分析则是通过运行恶意代码, 分析其在执行过程中的函数调用过程, 以及所产生的恶意行为来判断其是否为恶意程序。一些传统的恶意代码分析工具只能粗粒度地分析恶意代码的部分行为, 并不能深层次地揭示恶意代码对于系统的更改。

内存镜像是指计算机物理内存的完整复制, 包含系统运行状态的关键信息。恶意代码在系统中运行时, 会改变系统内存, 捕获内存完整副本并在另一台计算机上分析, 就有可能重构出原先系统的状态, 从而分析恶意代码的行为。所以, 在检测恶意代码行为的技术中, 内存分析技术受到了越来越多的关注, 用于计算机内存分析的技术工具也逐渐发展成为信息安全产业的一个特殊种类。由于传统的针对单个内存镜像的分析多是分析系统某一时刻“静止”的状态, 因此并不能够实时追踪系统的变化, 为了动态反映恶意代码生命周期中的活动及其对系统的影响, 本文提出一种连续内存镜像分析技术。通过连续获取恶意代码运行时期的内存镜像, 并在单个时刻内存镜像分析的基础上做镜像对比分析, 更好地揭示恶意代码的行为。本文主要完成 3 个方面的工作: 1) 使用虚拟化技术为恶意代码提供一个真实且隔离的运行环境, 在其中运行恶意代码, 实现连续的初始增量内存的获取; 2) 将初始及增量内存解析为完整的内存镜像并做对比分析; 3) 基于内存数据的可视化, 以图表形式动态地展示内存及系统运行过程中内存状态的变化。

2 相关工作

在内存镜像的获取方面, 主要有硬件、软件 2 种方法。基于硬件的方法中, 文献[2]提出一种用名为 Tribble 的硬件扩展卡来获取系统物理内存的方法, 可以用 Tribble 将系统的物理内存复制到外接存储设备中。这样调查人员就可以在既不引入任何新代码又不使用系统代码的情况下完成系统物理内存的获取工作, 缺点是硬件必须事先安装到系统中, 所以 Tribble 设备还不能被广泛使用。基于软件的物理内存获取方法主要有: 1) 使

用 Microsoft 崩溃转储技术, 在生成崩溃转储时, 系统会被冻结, 物理内存中的数据会被写入磁盘, 这样就完整地保存了系统状态; 2) 使用虚拟机技术, 当运行 VMWare 会话时, 可以挂起(suspend)这个会话, VMWare 会将系统的“物理内存”镜像以 DD 格式存放到一个扩展名为.vmem 的文件中, 使用这种方法的优点是挂起一个 VMWare 会话的操作非常简单快捷, 而且对系统内存的影响最小, 但是使用虚拟机方式, 每次获得的都是完整的系统内存镜像, 因此存储和传输的开销比较大。

在内存镜像的分析方面, 技术核心主要是对内存中关键数据结构实例的定位。时下最流行的在内存镜像中定位数据结构实例的办法主要是基于链表和特征的扫描。Walters 等^[3]提出了一个概念性的工具 Volatools, 其实现主要依赖于遍历链表和各种与进程相关的表来进行分析, 目前根据该工具可以找到的信息主要有运行进程、端口、进程与端口的映射关系等, 然后将其研究过程整合到数字分析进程的相关阶段。殷联甫^[4]提出了基于进程的检测方案, 主要是通过进程结构中的 PEB(环境控制块)来定位进程信息。内存取证分析工具 Volatility, 则结合链表扫描和标签扫描 2 种方法来提取内存对象, 其中在标签扫描方面, 提出一种针对内存对象定位的池标签扫描方法, 通过在系统内存中全局扫描与关键系统对象所对应的标签, 来定位诸如进程、DLL、服务、网络连接等一系列内存对象, 同时利用系统的一些属性, 来确定并提取出可疑的信息。

内存分析也存在一些其他研究方法。Dolan-Gavitt^[5]主要描述了 Windows 注册表在物理内存中的存储结构, 提出技术用来直接从内存转储中提取出这些数据。Okolica 等^[6]提出利用内存转储文件中的调试系统信息以及微软的程序数据库来提取出内存转储中的进程、配置以及网络活动信息。Kornblum^[7]则通过恶意代码的固有属性来进行分析。Stevens 等^[8]从寻找命令行历史出发, 但没有采取跟踪进程的方法, 而是直接根据命令程序的特点来直接寻找命令行执行历史, 这也和之前的定位进程、线程、网络连接的思路略有不同。

2005年, Bergeron等^[9]在基于语义的恶意代码分析方面进行了研究, 指出了实现对一个二进制可执行文件的静态分析可分为3个主要步骤: 构筑中间表示; 基于流的分析, 捕获程序的面向安全的行为; 对安全策略的关键行为进行静态验证(模型检测)。在2007年, YIN等^[10]提出了首个基于动态污点跟踪方法来检测和分析恶意代码的系统 Panorama, 通过使用全系统的细粒度动态污点跟踪方法, 来捕获恶意代码的基础特征, 同时通过对污点传播过程的分析, 来确定恶意代码是否操作了用户的敏感数据。

近年来, 国外的一些研究机构开始把沙箱技术引入到恶意代码的检测中, 德国曼海姆大学在2006年开发了一个名为 CW Sandbox 的系统^[11]。此系统利用 VMware 软件, 并且在其中运行沙箱软件, 利用沙箱软件隔离恶意代码的运行环境和真实的计算机环境, 所以对操作系统进行了有效的保护。同时使用用户级的 API Hook 方法对恶意代码的行为进行捕获, 可以自动地生成分析结果。然而, CW Sandbox 也存在不足的地方, 它在恶意程序分析结束后, 并没有对分析结果做进一步处理, 往往还需要进行人工分析才能确定目标程序所具有的破坏性。

综上所述, 目前在对于内存镜像的获取中, 只能获得当前情况下的内存镜像, 不能实现镜像自动地连续获取。同时在内存镜像的分析研究中, 多是针对单一时刻内存镜像, 因此分析的多是某一个时刻的内存状态, 不能很好地揭示内存状态与恶意代码之间的关系。本文使用操作系统虚拟化技术为恶意代码提供一个真实且隔离的运行环境, 在其中运行恶意代码, 并对获取的内存镜像做连续分析。

3 针对恶意代码的连续内存镜像分析设计和实现

当系统处于活动状态时, RAM 中包含了关于系统运行状态的关键信息。获取 RAM 的完整副本并在另一台计算机上进行分析, 就有可能重构出原先系统的状态, 包括系统运行的进程和网络连接等信息。通过连续获取系统的内存镜像, 进行前后镜像之间的对比分析, 可以获得恶意程序

的具体行为。

3.1 系统整体架构

运用虚拟机的脏页机制, 通过捕获并分析不同时刻内存镜像的对象集的差异, 推断出在镜像捕获的不同时期系统的变化。同时运用可视化工具, 将内存对象的变化动态直观地展示出来。其中在获取连续内存镜像的时候, 并不是连续获取各个时段的完整内存, 而是在获取系统初始时刻完整内存镜像的基础上, 连续获取一段时间内系统被修改部分的内存镜像, 并与前一时刻完整内存镜像进行覆盖和融合, 形成当前时刻的完整内存镜像, 此举可以有效地节省存储开销。整个连续内存镜像获取的架构如图1所示, 内存镜像的分析对比如图2所示。

3.2 连续内存镜像获取模块的设计

由于之前的内存镜像分析多是针对单一时刻的内存镜像, 因此传统的内存镜像获取方法多是直接获取整个系统的内存, 在做连续内存镜像分析的时候, 如果继续使用以前的方法, 产生的内存镜像文件数据将会很大, 无形中增加了存储和传输内存镜像的开销。因此, 提出一种新的内存镜像获取思路, 即在获取初始时刻完整内存镜像的基础上, 连续获取一段时间间隔内被修改部分的“增量”内存, 最后解析时将当前时刻的增量内存与前一时刻完整内存镜像进行覆盖融合, 从而形成当前时刻的完整内存镜像, 依次按照时间顺序进行覆盖融合, 就能够以付出一个完整内存镜像文件的代价, 获取连续的多个时刻的内存镜像文件。

利用 QUME 虚拟机制作 Windows XP 镜像, 挂载镜像到系统/mnt/winxp 目录下, 成功挂载之后, 将恶意代码复制到挂载目录下, 即镜像的 C 盘根目录。将恶意代码启动程序复制到 /mnt/winxp/Documents and settings Administrator/[开始]菜单/程序/启动目录下。与此同时, 针对 QUME 设计一个 mdump 命令来获取连续增量的虚拟机内存。系统启动之后, 从客户操作系统向虚拟机外运行的 QUMU 发送 mdump 命令, 开始获取初始及增量内存, 此时, 通过程序启动虚拟机内部的恶意样本。这样就可以获取恶意代码运行前、中、后的所有内存镜像。整个过程的时间线如图3所示。

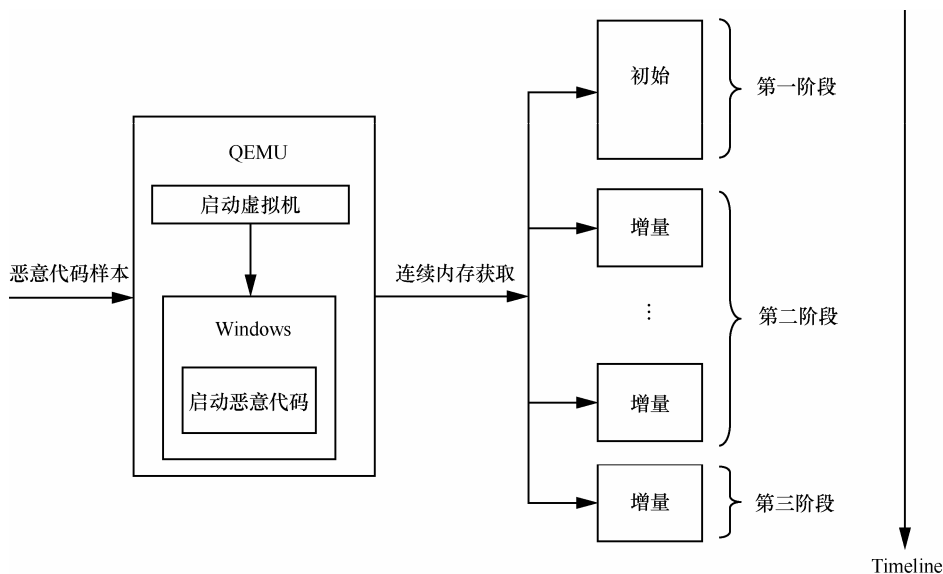


图1 连续内存镜像获取的架构

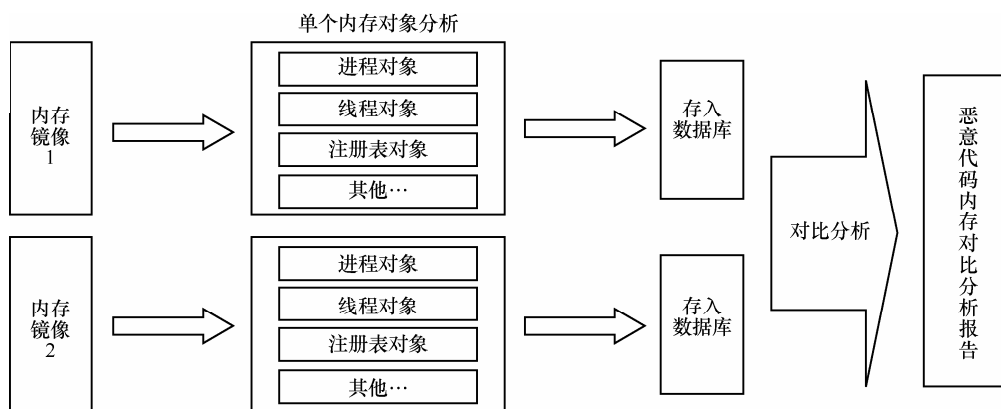


图2 内存镜像的分析对比

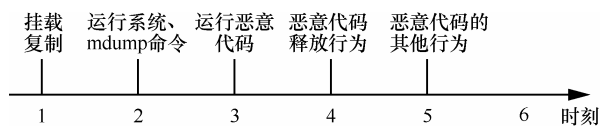


图3 连续内存获取时间线

在QEMU的monitor增加mdump命令用来连续地获取在一段时间内被修改的虚拟机内存镜像，此命令的实现借助了QEMU虚拟机的脏页机制。如果内存页在一段时间内被修改，就说该内存页变“脏”，并称之为脏页。将整个虚拟机内存的保存分为3个阶段。

- 1) 第一阶段：将内存中所有页面设置为脏页。
- 2) 第二阶段：将所有前一时刻中标记为脏页的内存页保存下来。
- 3) 第三阶段：将剩余所有脏页一次性保存，并结束整个存储过程。

获取过程的时间线如图4所示。

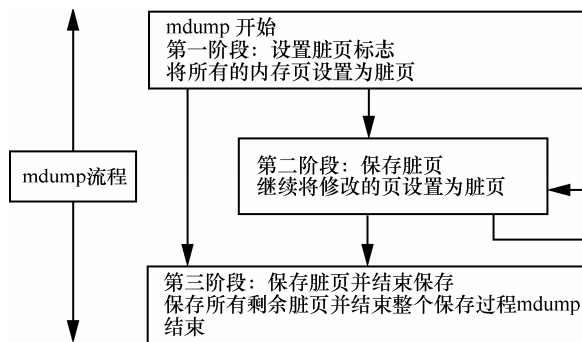


图4 mdump的连续内存获取时间线

具体到每一轮迭代中，对于内存页的设计如图5所示。

图5中的虚拟机内存，即所需要保存的内存页分为2种状态：每轮迭代期间内存页如果被改写，则页面状态标记为W，表示页面为脏页；其

他情况包括被读取以及没有操作均被标记为 R。那么在下轮迭代中只需传输被标记为 W 的内存页。为了更方便地查询内存页是否被改写，为每个页面设置一个标签来随时记录内存页的读写情况，在下一轮迭代中扫描该状态表，便可找出被改写的内存页并保存。

依次按照 QEMU 中内存块的顺序扫描，在每一个块中，对每一个大小为 4 096 B 的页面进行标签设置，如果页面的状态为 W，具体的做法是将其页面地址右移 12 bit 的值作为数组索引，并让这一索引指向的值为 0xff，对于状态为 R 的页面，则不做处理。通过这种方法，将状态为 W 和 R 的不同内存页区分开来。

由于在第一阶段中，所有的内存页都被设置为脏页，因此得到的只是一个描述性的文件，真正开始保存内存页是从第二阶段开始，第二阶段的第一个文件保存系统的完整内存页，从第二个文件开始，保存上一时钟中断时间内所有新产生的脏页。当进入第三阶段时，一次性保存前一时钟被设置为脏的内存页，并结束连续内存镜像获取的过程，至此，便得到了经过处理压缩的、初始的和过程中被修改的虚拟机内存镜像文件。

3.3 单内存镜像的解析

解析过程是保存的逆过程。第二阶段生成的第一个完整文件为整个系统的内存复制，结构与

第一阶段文件描述相同，即 pc.ram 为第一个块，大小 128 M。由保存的逻辑可以看到，前 8 个字节为“OFFSET| CONT| COMPRESS”。其中，OFFSET 为页在内存块中的偏移，CONT 用来判断页面是否在一个块内，COMPRESS 为 2，表示一整个页面的内容为重复的一个字节，保存时用这一个字节的内容来替代整页 4 096 B 的内容。COMPRESS 为 8，表示页面在存储时未被压缩，直接保存整页 4 096 B 的内容。

镜像内容如图 6 所示，前 8 个字节值为 0x8，由于 $8=0|0|8$ ，表示页偏移为 0，CONT=0 表示进入一个新的块，标志 COMPRESS 表示保存的时候未压缩，紧跟的一个字节 6，表示块名长度为 6，后面 6 个字节 pc.ram 表示块名，pc.ram 后面的部分即为所保存的页面内容，由于页面未压缩，故 pc.ram 后面 4 096 B 的内容即为页偏移为 0 这个页面的全部内容。

第一个页面的内容写完，继续读取 8 个字节。如图 7 所示，图中 8 字节大小为 0x1028，由于 $0x1028=1000|20|8$ ，表示页偏移为 0x1000，即第二个页，cont=0x20 表示还在之前的块中，压缩标志 COMPRES 表示保存的时候未压缩，故这 8 个字节后面 4 096 B 的内容属于 pc.ram 块中第二个页面的内容。依次类推，直到再次读取到 cont=0 时块结束，表示 pc.ram 块的内容已经全部解析完毕。这样，第一个完整的文件就解析完毕。

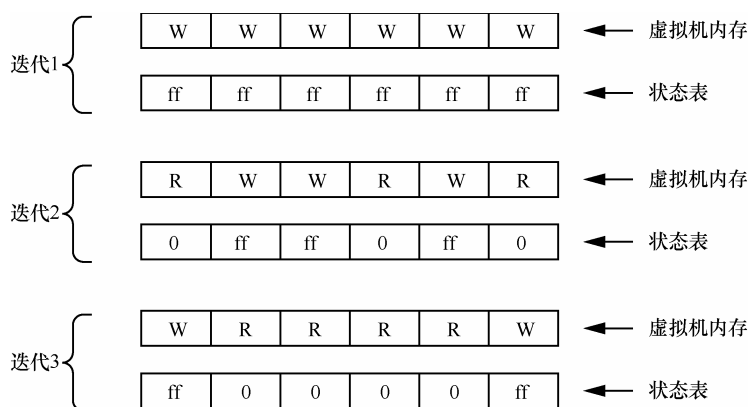


图5 连续内存镜像保存中每轮迭代示意

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	00	00	00	00	00	00	00	08	06	70	63	2E	72	61	6D	53	pc.ramS
00000010	FF	00	F0	53	FF	00	F0	C3	E2	00	F0	53	FF	00	F0	53	ÿ ãSÿ ãÃã ãSÿ ãS

图6 第二阶段文件片段

00000FF0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	
00001000	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00
00001010	00 00 00 00 00 10 28 45	E4 13 00 00 00 83 4D FC	(Eä Mü
00001020	FF 8B 75 E0 83 7D E4 00	74 0A 85 F6 74 06 56 E8	ÿ!uä!}ä t !öt Vè

图7 第二阶段文件片段

第二阶段剩下的文件为增量内存文件，只记录修改部分的内存。结构一般为带标志位的页面地址+页面内容。值得注意的是，不同于第二阶段第一个文件每个块只出现一次的情况，第二阶段剩余文件中，可能多次出现 pc.ram 块。造成这种情况的原因是整个时钟中断时间内不止扫描了一遍内存。因此需要先在文件中定位 pc.ram 块，再对每一个 pc.ram 块做解析。

第三阶段文件格式与第二阶段文件格式相同。第三阶段作为收尾阶段，会将此前所有被设置为脏页的内存页一次保存。至此可以得到各个时期的完整物理内存镜像，对各个时刻的完整物理内存镜像做解析，就可以得到各个时刻具体内存对象信息，之后将解析结果存入数据库。

3.4 连续内存镜像对比分析模块设计

在连续内存镜像对比分析方面，对比不同镜像

之间具体内存对象的差异，获得系统状态的更改。连续内存镜像对比分析方法的核心流程如图8所示。

要想获取恶意代码运行中的具体活动信息，就需要将不同时刻的内存镜像做对比，提取内存镜像之间的差异，从而获取恶意代码对系统的影响。将之前获取的不同时刻内存镜像分析结果的数据库进行对比，从而获得内存对象状态的更改。

对每一个具体的对象设置3种属性，分别是未改变、改变、新增。3种属性对应的情况如下。

1) 前后2个时刻的内存镜像中都存在该对象，且对象所有相关信息全部相同，则认为该对象的状态为未改变。

2) 前后2个时刻的内存镜像中都存在该对象，但对象的相关信息有所改变，则认为该对象的状态为改变。

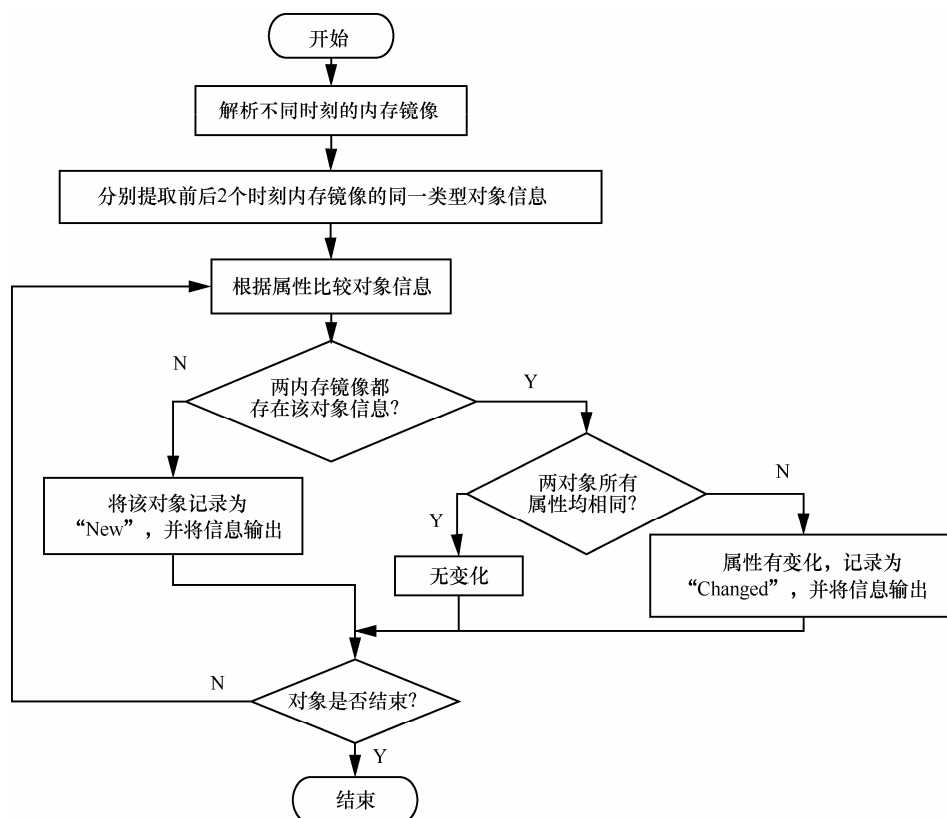


图8 连续内存镜像对比分析方法的核心流程

3) 某一对象出现在后一时刻的内存镜像中但未出现在前一时刻的内存镜像中, 则认为该对象的状态为新增, 这也是最应该引起重视的一种状态。

在对一些关键的内存对象进行定位以及信息提取时, 主要以进程对象为中心, 针对对象提取有2种方法: 链表扫描和特征字符扫描。

1) 特征字符扫描

具体的内存对象以及它们对应的标签如表1所示。

表1 Windows 内存对象对应的标签

对象	标签	标签(保护)	最小大小/B (Win7 X64)
Process	Proc	Pro\xe3	1 304
Thread	Thrd	Thr\xe4	1 248
Mutant	Mute	Mut\xe5	128
Driver	Driv	Dri\xf6	408
File	File	Fil\xe5	288

例如, 要对进程对象进行定位, 则在内存镜像文件中扫描标签“Proc”, 定位进程对象结构体, 获取进程对象的信息。

2) 链表扫描

链表扫描时, 则参照文献[3]中进程定位的方法, 对与对象相关的数据结构进行定位, 再通过关键元素定位对象相关的链表、扫描链表、定位对象, 实现内存镜像的对象对比, 一共设立3个列表, 分别表示未改变、改变和新增的内存对象。针对获取的3种列表结果进行过滤。

3.5 内存可视化模块的设计

内存中蕴含着丰富而又数量庞大的信息, 在针对恶意代码的连续内存镜像分析时, 整个系统中对象和对象之间的联系、单个对象内部之间的联系都是比较复杂和繁琐的。设计一个针对内存数据的可视化, 以图表的形式动态地展示内存及系统运行过程中内存状态的变化, 从而明确、有效地传递信息。

选择可视化工具 D3.js 进行可视化, 它是一个数据可视化的 JavaScript 函数库。设计使用点来代表各个进程, 相关联的父子进程用连线联系到一起, 同时对于具体进程对象的各种相关信息(如释放的文件、获取的权限、创建的网络连接等)均建立对应的点, 并将它们与进程

对象连接起来。

将每一时刻的完整内存镜像的分析结果的数据库文件转换成 JSON 数据文件, 然后依次对比当前时刻的 JSON 文件与前一时刻 JSON 文件的差异, 生成2个文件的差异 JSON 文件。用初始完整内存镜像的 JSON 数据文件画图, 然后在此基础上再读取差异 JSON 文件, 从而在图表上动态地反映系统的变化。整体的 JSON 文件格式如图9所示。

```
links:{"source":"pid:source_process","source_group":id,
"target":"pid:target_process","target_group": id }

source :起点节点 source_group: 起点的属性(进程, 连接, 权限等)
target:终点节点 target_group: 终点的属性(进程, 连接, 权限等)
```

图9 数据文件 JSON 格式

D3 会自动解析 JSON 数据文件, 将其中 source 和 target 表示部分设为节点, 并在这两点间创建连线, target_group 和 source_group 分别表示对点设置属性。开始时解析进程表, 若其中一个的 ppid(父进程号)与另一个的 pid(进程号)相等, 就认为这2个进程是父子进程, 将所有的父子进程连接在一起, 然后开始扫描其他的插件表。以进程作为图像的中心, 对于每个插件, 首先设置形如“进程 id_插件名称”的中间节点, 将具体进程与其对应的中间节点连接在一起, 然后将每条具体的插件项与中间节点连接起来, 形成进程的插件网。

将每一时刻的完整 JSON 文件进行对比, 生成差异 JSON 文件, 具体的文件格式如表2所示。

表2 增量 JSON 文件格式

域名	含义	取值范围
links_add	新增连线	当前时刻较上一时刻新增的节点
nodes_remove	移除节点	当前时刻较上一时刻减少的节点
nodes_add	新增节点	当前时刻较上一时刻新增的连线
links_remove	移除连线	当前时刻较上一时刻减少的连线

利用初始完整的 JSON 文件形成初始的完整内存对象 D3 图之后, 利用差异 JSON 文件, 就在原图像的基础上加载下一时刻、上一时刻的差异 JSON 文件, 实现节点和连线的动态扩展。通

过这种方式连续、动态地反映系统状态的改变。

以恶意样本 Explorer.exe 为例，图 10 显示了初始系统的所有进程对象，以及和进程相关的插件，如权限、网络连接、代码注入等，此时恶意样本还没有启动。

当单击图 11 中具体某个点时，会显示点所代表的具体含义。设置 2 个 prev 和 next 按钮，单击之后分别显示系统上一时刻和下一时刻的状态。切

到下一时刻状态，通过图 11 可以看到，run_malware.exe 进程启动了样本进程 Explorer.exe，同时恶意样本出现了代码注入的情况，同时还创建了新的网络连接，提升了自己的一部分权限。

4 实验及分析

测试的样本集包括搜集到的 Sentry_MBA、stub 等 40 多种典型恶意代码，包括木马、蠕虫、

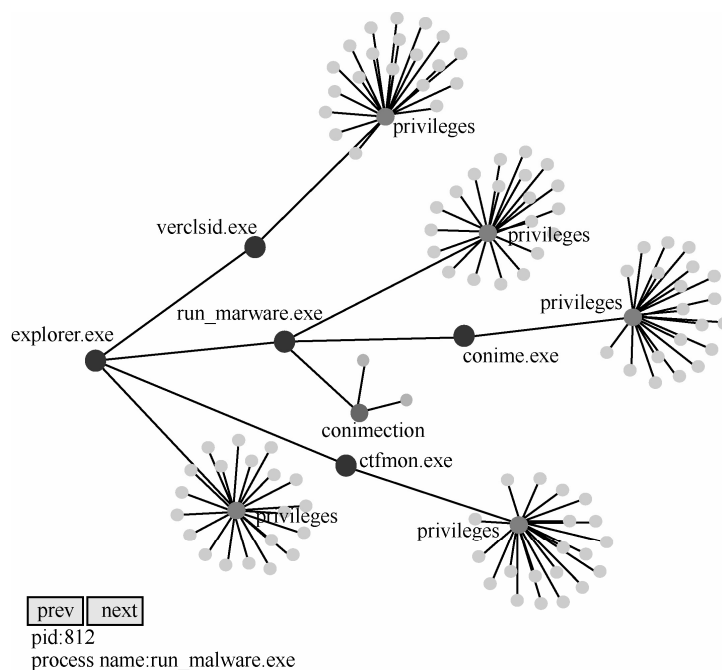


图 10 系统初始时恶意代码的可视化状态

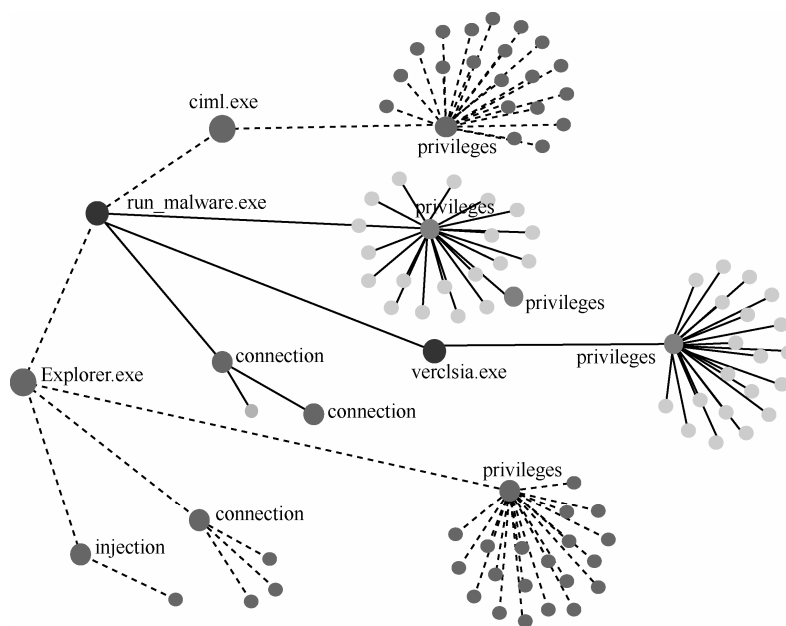


图 11 恶意代码的行为可视化

后门、嗅探器等各种类型。

4.1 典型样本测试

本实验选取 Sentry_MBA.exe 恶意代码样本，启动原型系统，在虚拟机中运行样本，并获取连续的虚拟机内存，解析为连续的内存镜像之后做分析，生成恶意程序的行为报告。选取检测结果中的进程，网络连接进行分析，如图 12 所示。

恶意代码程序在运行过程中创建了 sentry_mba.exe 和 systemtray.exe 这 2 个进程（由 pid 及 ppid 可以看出 Sentry_MBA 是这 2 个进程的父进程）。在网络连接方面 systemtray.exe 进程创建了网络连接，绑定了本地的 1031 端口，并访问了远程的 80 端口，可以推测是访问了远程的网站。

对比系统和文件 B 超对于样本的整体检测结果，如表 3 所示。

表 3 Sentry_MBA 样本内存镜像分析与文件 B 超结果对比

恶意代码行为	检测到恶意代码行为	
	文件 B 超	连续内存镜像分析技术
进程检测	否	是
代码注入	否	是
注册表	是	是
设置启动项	是	是
互斥量	是	是
尝试访问主机	否	是
释放 PE 文件	是	是
提升调试权限	是	是
枚举进程树	是	否

连续内存镜像分析方法在进程检测、代码注入、尝试访问主机方面都能检测到文件 B 超未能检测到的行为，对于注册表的修改、设置开机启动项、创建互斥、释放 PE 文件、提升调试权限

等两者都可以检测到。

4.2 分析结果与文件 B 超对比

对 40 种恶意代码样本进行统计分析，如表 4 所示，分别从进程检测、代码注入、注册表相关、互斥量、网络相关、文件相关、提升权限等方面进行统计。将分析结果与文件 B 超对恶意代码样本的分析结果进行对比。

表 4 参与测试和统计分析的 40 种恶意代码样本

编号	样本名称	编号	样本名称
1	aimware	21	wj
2	_v_exe	22	YY_611j_dk
3	5	23	Fallen
4	8de13aa98	24	Trojan.Spy
5	56be19bc00	25	Angeldos
6	b12b4b580c	26	KingBomb
7	BP	27	Winp
8	c380ab61	28	AIM
9	ebe3ae8749	29	XScript.03
10	Explorer	30	KillCerb
11	GenericMalware	31	Acidoor.11
12	ghostface	32	parite
13	He	33	AF
14	InstaGet	34	MiSpace.a
15	juschk	35	Lovesan.a
16	Mega_Joiner	36	keylogger
17	Office	37	lastgift
18	Sentry_MBA	38	Inor
19	setup_20006	39	agobot
20	stub	40	Emaner

总体来看，检测到的行为分布主要如图 13 所示。

接下来，针对文件 B 超检测到的结果与本系统检测到的结果进行对比，总体情况如图 14 所示。

在针对注册表、注入代码以及文件相关的行为检测方面，系统分别比文件 B 超的检测数量多

processes						
Status	offset	name	pid	ppid	image_path_name	
New	0x6ef8ab8	Sentry_MBA.exe	872	656	C:\Sentry_MBA.exe	
New	0x6235da0	sentry_mba.exe	1332	872	C:\WINDOWS\sentry_mba.exe	
New	0x62358a8	systemtray.exe	1248	872	C:\WINDOWS\systemtray.exe	
connections						
Status	pid	local_ip	local_port	remote_ip	remote_port	proto protocol
New	1332	10.0.2.15	1031	216.146.38.70	80	
New	1332	0.0.0.0	1031			TCP 6

图 12 进程和网络连接检测结果

出6、4、4个，数量超过了文件B超。注册表方面主要在于本系统并非单纯地对单个对象的分析，而是基于相关联多对象的分析，从而可以发现更多的可疑对象，注入代码方面则由于本系统是直接针对虚拟地址描述器的属性检测，行为方面更加具体，释放文件方面由于和文件B超检测项并不是一一对应关系，所以这里仅作为参考。

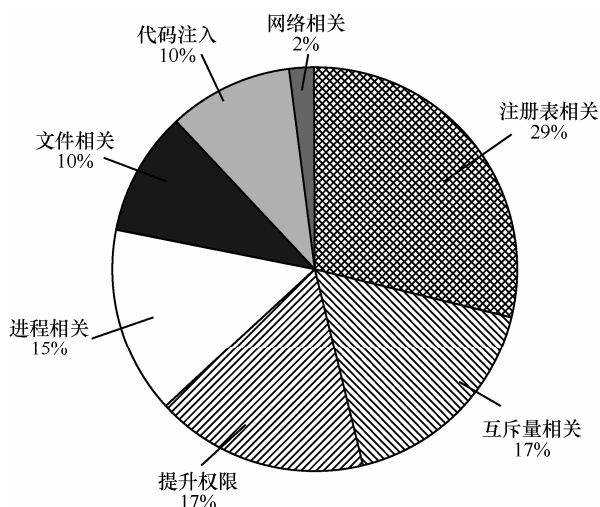


图13 恶意代码内存分析方法检测行为分布

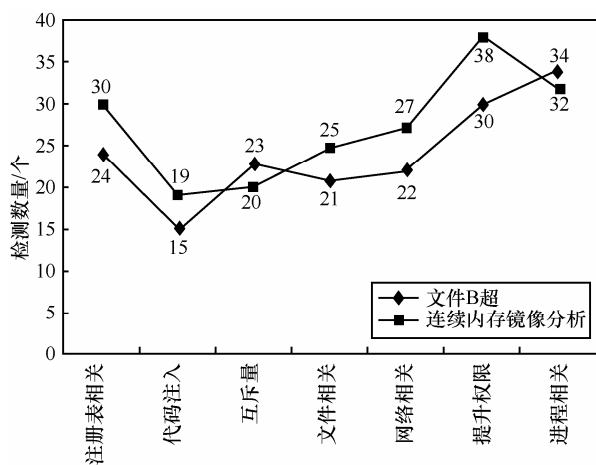


图14 恶意代码内存分析结果与文件B超结果对比

进程相关、网络连接和提升权限方面，系统分别比文件B超的检测数量多出2、5、8个。由于在做进程相关的检测时，运用了包括链表扫描以及特征码扫描等多种办法，因此可以发现部分被隐藏的进程。网络相关方面，虽然能发现更多由恶意程序发起的网络连接，但对部分恶意代码所访问的网址并不能很好地解析。

在互斥量方面，文件B超检测到的恶意代码

行为数量超过本系统3个，由于本系统互斥量检测采取正则匹配原则，是基于目前已有的特征库的搜索，而文件B超是实时监控恶意代码的所有行为，能够准确地确定有恶意代码产生的互斥量。

4.3 连续内存镜像与单内存镜像分析对比

针对4.2节中的恶意代码样本进行统计分析，先获取恶意代码运行一段时间之后保存的单内存镜像做分析，再利用连续内存镜像分析技术，对恶意代码进行分析。2种方法在恶意代码行为检测方面的对比结果如图15所示。

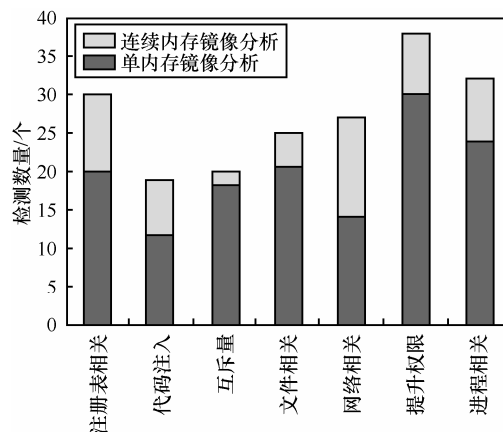


图15 连续内存镜像对比分析较单内存镜像分析的结果对比

总体来讲，连续内存镜像对比分析较单内存镜像分析检测结果有明显提升，针对单个内存镜像的分析多是分析系统某一时刻“静止”的状态，而连续分析模块在单内存分析模块的基础上增加了多个时刻的内存状态对比的分析，可以更加全面地捕获恶意代码整个生命周期的行为。根据实验数据得出，恶意代码样本行为数量在单内存镜像分析的基础上增加了19.7%。

5 结束语

本文提出了针对恶意代码的连续内存镜像分析技术，通过在虚拟机中运行恶意代码，并获取恶意代码运行前、中、后不同时期的连续虚拟机内存镜像，解析为各个时刻完整内存镜像之后，做镜像对比分析，从而提取出恶意代码的行为以及其对系统的影响，并且通过可视化工具对检测结果进行展示，通过动态的演进展示恶意代码运行的生命周期。实验中对40种典型的恶意代码样本利用系统的方法进行了检测。结果表明，这种方法可以有效且直观地发

现被恶意代码感染的系统内存中存在的恶意行为。

本文提出的针对恶意代码的连续内存镜像分析技术虽然能够比较准确地发现目前典型的恶意代码行为,但是仍然存在需要改进的方面:1) 由于是在 QEMU 虚拟环境中运行恶意代码样本并分析,因此当遇到一些反沙箱的恶意代码,并且这些恶意代码意识到自己处在沙箱环境中,就会自己休眠,从而不能启动恶意样本并进行分析;2) 在做针对内存数据的可视化时,各种对象之间的关系比较复杂,如何突出显示真正有效的信息,对最后的结果进行过滤,也是需要解决的问题。

参考文献:

- [1] 2014-2015 中国互联网安全研究报告[R].猎豹移动安全实验室, 2015.
2014-2015 Report of China Internet security study[R].Cheetah Mobile Security Laboratory, 2015.
- [2] ROLLES R. Unpacking virtualization obfuscators[C]//The 3rd USENIX Conference on Offensive Technologies. 2009: 1.
- [3] WALTERS A A, PETRONI N L. Volatools: integrating volatile memory into the digital investigation process[C]//Black Hat DC. 2007:9-20.
- [4] 殷联甫. 计算机取证中的物理内存取证分析方法研究[J]. 计算机应用与软件, 2010, 27(12): 295-298.
YIN L F. Study of physical memory forensic analysis method in computer forensics[J]. Computer Application and Software, 2010, 27(12): 295-298.
- [5] DOLAN-GAVITT B. Forensic analysis of the Windows registry in memory[J]. Digital Investigation, 2008,5(1): S26-S32.
- [6] OKOLICA J, PETERSON G L. Windows operating systems agnostic memory analysis[J]. Digital Investigation, 2010(7): S48-S56.
- [7] KORNBLUM J. Exploiting the Rootkit paradox with Windows memory analysis[J]. International Journal of Digital Evidence Fall, 2006,5(1):4.
- [8] STEVENS R M, CASEY E. Extracting Windows command line details from physical memory[J]. Digital investigation, 2010(7): S57-S63.
- [9] BERGERON J, DEBBABI M, DESHARNAIS J, et al. Static detection of malicious code in executable programs[C]//BIBUV. 2005: 1-5.
- [10] YIN H, SONG D, EGELE M, et al. Panorama: capturing system-wide information flow for malware detection and analysis [C]//The 14th ACM Conference on Computer and Communications Security. 2007: 116-127.
- [11] WILLEMS C. CWSandbox: automatic behaviour analysis of malware[J]. Digital Investigation, 2013, 3(4): 97-12.

作者简介:



李伟明 (1975-), 男, 湖南株洲人, 博士, 华中科技大学副教授, 主要研究方向为信息安全。

邹德清 (1973-), 男, 湖南湘潭人, 博士, 华中科技大学教授、博士生导师, 主要研究方向为云计算、信息安全。

孙国忠 (1973-), 男, 吉林大安人, 博士, 曙光信息产业(北京)有限公司高级工程师, 主要研究方向为高性能计算、云计算。