文章编号:1001 - 9081(2005)12 - 2808 - 03

对超粒度混杂技术的改进:基于瘦虚拟机的指令集交替技术

于 森,孙 强
(华东师范大学 计算机科学技术系,上海 200062)
(platinicslug@ sohu. com)

摘 要:超粒度混杂技术对于小型解密程序效率低下,为此提出了基于瘦虚拟机的指令集交替技术。该技术使用一个自动机来记录程序加密和解密的方法,并且使用瘦虚拟机来完成对加密过的程序解释执行。测试结果表时,该技术在保证加密强度的条件下,对效率有较大的提高。

关键词:瘦虚拟机;指令集交替技术;加密;加壳技术;超粒度混杂
中图分类号: TP311.56 **文献标识码**:A

Improving hyper-granularity immingling technique: instruction set alternation technology based on thin virtual machine

YU Miao, SUN Qiang
(*Department of Computer Science and Technology, East China Normal University, Shanghai* 200062, *China*)

Abstract: The hyper-granularity immingling technology is inefficent with small decrypted programs, then instruction set alternation technology based on a thin virtual machine was proposed. This technology employed an automata to record encrypt and decrypt ways. Furthermore, the thin virtual machine was responsible for interpretation and execution of these encrypted programs. Test results show that the technology can improve encrypted programs' efficiency without weaken the security.

Key words: thin virtual machine; instruction set alternation technology; encrypt; encryption shell protection; hyper-granularity immingling

0 引言

目前,对软件的攻击主要通过反向工程的方法进行,即通过静态分析、动态分析或两者的结合,分析目标软件的结构和意图,找出关键代码,对其进行修改。其中静态分析是指通过各种反汇编软件,如 IDA Pro,把目标代码反编译成汇编代码指令,然后通过人力阅读汇编代码指令,理解目标程序的意图。而动态分析是利用各种调试器,如 softICE 或 ollyDbg,对目标软件进行调试,利用调试器提供的各种信息(如当前执行的指令,寄存器信息,局部变量信息,线程变量信息等)来辅助分析目标代码。

现在对于反向工程,主要采用代码模糊技术,水印/指纹技术和加壳技术来防范[6]。其中代码模糊技术是对代码进行等价变换使之难以理解,主要用于防止静态分析;水印/指纹技术是在程序中隐藏某些信息,主要作为在法庭起诉的证据;在软件保护中使用最广泛也最有效的是加壳技术。

1 加壳技术

1.1 传统的加壳技术和攻击方法

加壳技术是指:将要保护的程序以某种算法(如 RSA)加密,并且和"壳"一起放入到一个新的可执行程序中。"壳"是一段专门负责目标软件不被非法修改或反编译,并且负责在运行时对被保护的程序进行解密的程序。在程序运行时,首先执行的是壳,它会检测是否有威胁存在(如:有调试器正在调试自己,自己的代码已经被修改等),如果没有威胁,壳就会对被保护的程序进行解密,并且把执行权转移。现在加壳软件种类很多,如著名的 ASPack, UPX, PECompect。

但是在传统的加壳软件保护下,安全程度并不是很高。这是由于壳仅仅是一段程序,它不可能识别出所有的调试器,并且在壳执行完毕后(这个位置一般称为 OEP,Original Entry Point),壳就会把被保护的程序解密,此时只要把内存中的代码提取出来就行了(常称为 dump)。下面利用动态分析的方法简述攻击的基本过程:首先根据特征,判断壳的种类(可利用 PEiD 等工具),根据不同的壳找出隐藏调试器的不同方法;然后运行调试器跟踪目标程序查找 OEP,暂停目标软件的运行;接下来 dump 代码,修正 OEP 使之指向真正的入口点、修复块表,最后重建输入表。当然实际的过程比较复杂,但原理和步骤还是一样的。

1.2 超粒度混杂技术[7]和它的不足

可以看到,使用传统的加壳软件生成的壳在 OEP 处和目标软件有一条明显的界限,这条界线使它们无论在内存空间还是在逻辑功能上都明显地分隔开了。针对这个缺点,我们使用超粒度混杂技术来弥补。超粒度混杂技术的基本思想是把目标代码分解成多个充分小的模块,分别保护。模块之间交替执行,保证在内存中的解密后的目标模块只有一个。通过使用超粒度混杂技术可以大大提高代码的安全性。

但是超粒度混杂技术也有自己的不足,即对目标程序的性能有比较大的影响。因为各个模块使用成熟的高强度加密算法,而且为了保障程序的安全性,划分的模块比较多,并且每个模块必须在执行完毕后重新加密,下次运行时还要解密,最后每个模块还都有自己的校验代码。综合以上各个因素可以看出,超粒度混杂技术的主要缺点就是:使目标程序无论从

收稿日期:2005 - 06 - 13;**修订日期**:2005 - 08 - 16

作者简介:于淼(1978 -),男,河南开封人,硕士研究生,主要研究方向:软件理论; 孙强(1963 -),男,安徽合肥人,副教授,主要研究方向:软件理论.

空间消耗还是时间消耗上都有很大的增加。

我们做过试验,使用超粒度混杂技术,目标模块为 10 个,采用 CRC 校验,使用 DES 加密方法,对于大型程序性能下降为 15% 左右,对于中型程序性能下降为 20% ~40%,对于小型程序性能下降要超过 50%。可以看出超粒度混杂技术比较适合于大型程序。

为了解决性能问题,提出了基于瘦虚拟机的指令集交替技术。本方法不但使性能有大幅的提高,并且使被保护的软件的安全性也略有提高。

2　基于瘦虚拟机的指令集交替技术

2.1　原理

基本思想是在目标程序中加入一个小型虚拟机(称为瘦虚拟机),然后使用瘦虚拟机的指令集替换原来的指令集。当目标程序运行时,首先运行虚拟机,它在进行一些反调试和完整性校验操作后,负责对替换后的指令集进行解释执行。当然仅替换指令并不能带来多少安全性的提高,所以利用一个自动机来记录语境,然后根据语境来决定翻译成的代码。这就造成新的指令集和原来的指令集并不是一一对应的,目标程序中的一条指令可能翻译成多条指令,同时目标程序中的不同指令也可能翻译成同一条指令。本处的虚拟机就是要起到壳的作用,即对被加密的指令解释执行。

这就相当于把加密的模块缩小到一条指令,安全性有所上升。同时由于不需要对模块重复加密,解密,所以主要的消耗是虚拟机的消耗。由于这里的虚拟机仅仅进行指令的解释和一次反调试和完整性校验,所以开销不是很大。

2.2　总体结构

如图 1 所示,本文的加壳软件在加壳时,随机选择一个自动机 M_x(包含一种私有的指令集,一个状态集,一个输出函数集,一个转换函数集和一些其他信息),把目标程序的代码根据 M_x 影射(加密)成虚拟机的私有指令集代码,同时根据 M_x 自动生成其逆自动机 M_x^{-1},然后把 M_x^{-1} 隐藏到目标程序中,最后把虚拟机放到目标程序中。运行时虚拟机需要使用 M_x^{-1} 对受保护的程序进行解密,M_x^{-1} 也是解密的钥匙,使用水印技术把它隐藏在目标程序的资源(比如一张图片,一段音乐)中。

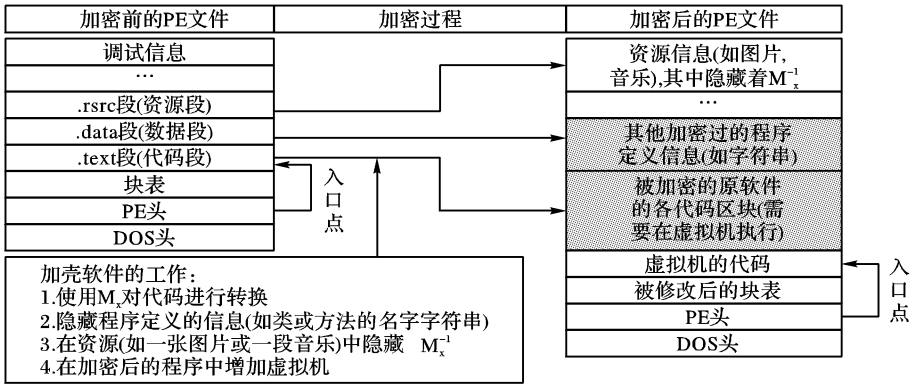

图 1　系统整体结构

2.3　瘦虚拟机技术

目前虚拟机技术已经十分成熟了,这里的瘦虚拟机是指在本文中使用的虚拟机在功能上比较简单:仅仅进行指令的解释执行。鉴于篇幅原因,本文对这种成熟的技术不再介绍。

2.4　指令集交替技术

上面已经提到,仅仅使用虚拟机,对程序的指令进行一次替换,并不能带来多少安全性的提高,所以我们使用自动机 M_x 根据语境来控制和记录指令间的映射关系。而 M_x 就像常用加密算法中的密钥。利用自动机可以使每次加密都采用不同的映射,这样如果不知道自动机根本无法知道原指令序列是什么。使用自动机进行指令替换的技术在这里称为指令集交替技术。

2.4.1　加密使用的自动机 M_x 和解密使用的自动机 M_x^{-1}

把一些事先定义好的 M_x 放在加壳软件中,供加密使用,用户也可以根据规则自己定义 M_x。在加密完毕后加壳软件会根据使用的 M_x,自动生成相应的解密使用的自动机 M_x^{-1}。

1) 加密使用的自动机 M_x 的定义

对于加密使用的 M_x,可以用一个确定性有限状态自动机(DFA)表示,但是为了对指令进行翻译,必须对 DFA 进行一些修改,首先必须有一个根据当前的状态和输入指令,决定生成指令的输出函数。因此本文中的 DFA 可以用一个 7 元组表示:$M_x = (K, \Sigma_i, \Sigma_o, f_t, f_o, S, Z)$,其中:

K 是一个有穷集,它的每个元素称为一个状态;

Σ_i 是一个有穷子母集,它的每个元素称为一个输入字符,所以也称 Σ_i 为输入符号字母表;在这里,Σ_i 就是目标程序原来使用的指令集;

Σ_o 是一个有穷子母集,它的每个元素称为一个输出字符,所以也称 Σ_o 为输出符号字母表;在这里,Σ_i 就是加密后的指令集;

f_t 是转换函数,是在 $K \times \Sigma_i \to K$ 上的映像,即 $f_t(k_i, a) = k_j (k_i \in K, k_j \in K, a \in \Sigma_i)$ 就意味着:当前状态是 k_i,输入的指令为 a 时,将转换到下一个状态 k_j,把 k_j 称为 k_i 的一个后继状态;

f_o 是输出函数,是在 $K \times \Sigma_i \to \Sigma_o$ 上的映像,即 $f_o(k_i, a) = b (k_i \in K, a \in \Sigma_i, b \in \Sigma_o)$ 就意味着:当前状态是 k_i,输入的指令为 a 时,就输出指令 b;

$S \in K$ 是唯一的一个初态;

$Z \in K$ 是一个终态集,终态也就是可接受状态或结束状态。

2) 加密使用的自动机 M_x 的一个简单例子

下面以一个最简单的例子来说明自动机的使用,如图 2 所示(为了方便读者阅读,这里仅使用指令 Push 和 Pop)。

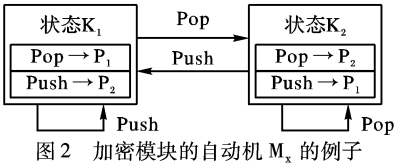

图 2　加密模块的自动机 M_x 的例子

其中:状态集 $K = \{k_1, k_2\}$;输入字母集 $\Sigma_i = \{Pop,$

$Push\}$;输出字母集 $\Sigma_o = \{P_1, P_2\}$;转换函数集是 $\{f_t(k_1, Pop) = k_2, f_t(k_1, Push) = k_1, f_t(k_2, Pop) = k_2, f_t(k_2, Push) = k_1\}$;输出函数集是 $\{f_o(k_1, Pop) = P_1, f_o(k_1, Push) = P_2, f_o(k_2, Pop) = P_2, f_o(k_2, Push) = P_1\}$;开始符号集 $S = \{k_1\}$;终结符号集 $Z = \{k_1, k_2\}$。

用此自动机对表 1 中 1 ~ 3 列所示的代码进行加密,加密后的代码如第 4,5 列。从加密后的代码可以看出:第 2,4 行同样的 push 指令被翻译成了不同的代码;第 5,6 行同样的 pop 指令也被翻译成了不同的指令;第 3,4 行不同的指令 push 和 pop 都被翻译成了 P_1。

表 1 加密前与加密后的代码对比

加密前的代码			加密后的代码	
行号	状态	指令	行号	指令
1	k_1	Push a	1	P_2 a
2	k_1	Push a	2	P_2 a
3	k_1	Pop	3	P_1
4	k_2	Push a	4	P_1 a
5	k_1	Pop	5	P_1
6	k_2	Pop	6	P_2

3)根据加密模块使用的自动机 M_x 生成解密模块使用的自动机 M_x^{-1}

下面给出由 M_x 生成 M_x^{-1} 的算法:①$M^{-1}x$ 的状态集 K^{-1} 与 M_x 的状态集 K 相同;②M_x^{-1} 的输入符号字母表 Σ_i^{-1} 是 M_x 的输出符号字母表 Σ_o;③M_x^{-1} 的输出符号字母 Σ_o^{-1} 是 M_x 的输入符号字母表 Σ_i;④对于 M_x 的 K 中的每个状态 k_i,把 k_i 中所有从本地出发的转换函数 $f_t(k_i, a) = k_j (k_i \in K, k_j \in K, a \in \Sigma_i)$,用 k_i 中相应的输出函数 $f_o(k_i, a) = b (k_i \in K, a \in \Sigma_i, b \in \Sigma_o)$ 替换转换函数中的输入字母 a,就得到 M_x^{-1} 关于状态 k_i 的转换函数:$f_t^{-1}(k_i, b) = k_j (k_i \in K^{-1}, k_j \in K^{-1}, b \in \Sigma_i^{-1})$;⑤对于属于 M_x 的 K 中的每个状态 k_i 的输出函数:$f_o(k_i, a) = b (k_i \in K, a \in \Sigma_i, b \in \Sigma_o)$,只要交换其中的输入 a 与输出字母 b 的位置,就可以得到 M_x^{-1} 的输出函数:$f_o^{-1}(k_i, b) = a (k_i \in K^{-1}, a \in \Sigma_o^{-1}, b \in \Sigma_i^{-1})$;⑥$M_x^{-1}$ 的初态 S^{-1} 和终态集 Z^{-1} 与 M_x 的相同。

2.4.2 哑指令

上面的方法对于顺序结构的程序是正确的,但是对于分支结构和循环结构的程序却存在着问题。因为我们在加密代码的时候忽略代码是什么结构,都依照代码出现的位置、顺序加密。但是在解密的时候,是按程序的逻辑顺序进行的,也就说解密是按照程序的执行顺序而不是文件中的空间顺序进行的。这就造成可能出现加密与解密的顺序不同,影响指令的正常解密,这就是为什么要加入哑指令。

哑指令是一条仅仅使自动机的状态发生变化的指令,也就是说每条哑指令都有一个空状态转换函数与之对应:$f_{null}(k_i) = k_j$,它不必有任何输入就可以使自动机的状态发生变化。哑指令应放在程序块的入口处和出口处。

2.5 隐藏解密使用的自动机 M_x^{-1}

对于本文提出的方法,M_x^{-1} 是关键部分,是解密的钥匙,攻击者会设法获取程序所使用的自动机来进行破译,所以保存 M_x^{-1} 的方法必须安全。

由于自动机 M_x^{-1} 是一些静态数据,并且对它仅仅进行读操作。所以这就为我们的隐藏带来了很好的便利,因为程序本身也要频繁的读取自己的资源,这些资源的性质和我们的自动机 M_x^{-1} 的性质十分相似,所以把 M_x^{-1} 隐藏到资源中是很好的选择。

对于在图片和音频中隐藏信息早已成为成熟的技术,这里不再叙述,可以参考文献[10]。

3 性能分析

分别编写了采用本文方法的加壳软件和采用超粒度混杂技术的加壳软件,并且分别使用这两种加壳软件对不同种类程序进行加壳,然后进行性能比较。结果发现采用本文方法加密过的程序的性能降低约在 10% 左右,并且基本不随目标程序的规模的改变而改变。显然效果远远好于超粒度混杂技术,如图 3 所示。

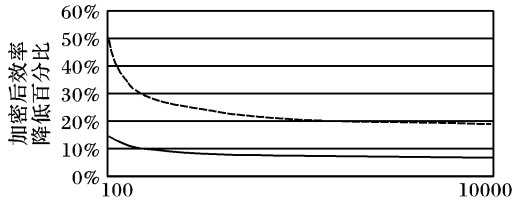

图 3 两种方法的性能比较

4 结语

本文提出了一种新的软件保护技术:基于瘦虚拟机的指令集交替技术。由于篇幅原因这里不能对所有方面进行详细介绍,其他未能涵盖的方面包括:瘦虚拟机中的反调试代码的组成,指令的解释执行过程,用户自定义 M_x 的规则,哑指令的具体位置与内容,隐藏 M_x^{-1} 使用的具体算法等。

本文的方法是对超粒度混杂技术的改进,从性能分析可以看出本文的方法确实在性能上有很大的提高,并且由于解密模块的减小,安全性也有小幅的提高。

参考文献:

[1] KASPERSKY K. Hacker Disassembling Uncovered[M]. A-LIST Publishing, 2003.

[2] 段钢. 加密与解密[M]. 北京:电子工业出版社, 2003.

[3] 薛亮, 刘路放, 冯博琴. 使用注册表和网卡实现软件保护[J]. 计算机工程, 2001, 27(12): 149 - 150.

[4] 李涛, 欧宗瑛. 利用加密技术和网卡进行软件保护[J]. 计算机应用, 2000, 20(1).

[5] 佟晓筠, 王蓁, 杜宇, 等. 基于软件安全混合加密技术的研究[J]. 计算机工程, 2004, 30(23): 98 - 100.

[6] COLLBERG CS, THOMBORSON C. Watermarking, tamper-proffing, and obfuscation: tools for software protection[J]. IEEE Transactions on Software Engineering, 2002, 28(8): 735 - 746.

[7] 于淼, 孙强. 对加壳技术的改进:超粒度混杂技术[J]. 计算机应用. 2004, 24(8): 137 - 139.

[8] BLUNDEN B. Virtual Machine Design and Implementation in C / C++[M]. Wordware Publkshed, Inc, 2002.

[9] VENNERS B. Inside the Java Virtual Machine[M]. Second Edition. McGraw-Hill, 2002.

[10] KATZENBEISSER S, PETITCOLAS FAP. Information Hiding Techniques for Steganography and Digital Wattermarking[M]. Artech House, Inc, 2000.