第34卷 第22期 计 算 机 工 程 2008年11月
Vol.34 No.22 Computer Engineering November 2008
·安全技术· 文章编号：1000—3428(2008)22—0187—02 文献标识码：A 中图分类号：TP311.13

二进制环境下的缓冲区溢出漏洞动态检测

夏 超，邱卫东
(上海交通大学密码与信息安全实验室，上海 200240)

摘 要：提出一种在二进制环境下挖掘缓冲区溢出漏洞的方法。结合动态与静态挖掘技术对二进制环境下的程序作进一步的漏洞查找。静态方法主要对二进制程序中函数栈帧的特征和汇编语句的内在语义关系进行分析，动态模拟方法为程序和函数提供了一个虚拟的运行环境，使程序在运行过程中结合一些静态特性得到该函数缓冲区变量的内存读写语义，最终判定程序中是否有缓冲区溢出。
关键词：动态检测；虚拟运行环境；语义

Dynamic Detection of Buffer-overflow Vulnerabilities in Binary Environment

XIA Chao, QIU Wei-dong
(Cryprography and Information Security Lab, Shanghai Jiaotong Univerisity, Shanghai 200240)

【Abstract】This paper proposes a method to detect buffer-overflow vulnerabilities for executables. Combining dynamic analysis and static analysis, it makes further detection of buffer-overflow vulnerabilities. Static methods mainly deal with the internal semantic relationship of assembly instructions and the properties of a function's stack frame for executables. Dynamic emulation provides a virtual run-time environment, which enables the program to combine its static properties while virtually executed, and then it can get the function's semantic results on buffer manipulation, and determine whether there is a buffer-overflow vulnerability.
【Key words】dynamic detection; virtual run-time environment; semantic

目前针对缓冲区溢出漏洞攻击的防范技术分析主要分为静态分析和动态分析。静态分析主要是对程序进行扫描，判别缓冲区的位置和大小，对一些可疑点(如strcpy/strcat/ sprintf函数调用处)进行一定的上下文相关分析，然后进行溢出判断。特点是速度快，缺点是分析不够精准，容易造成误报。动态分析实际上是一种黑盒测试，通过构造一些参数，然后施以一定的测试模式(暴力测试、边界测试等)。这种做法的盲目性比较大，需要有经验的分析员对程序漏洞进行预测[2]。然而动态分析的优势在于提供了一个真实的环境，直接针对二进制程序，使漏洞的挖掘和定位很准确。

1 模型与方案

本文主要借助于Wagner静态分析[1]的思想，设计了一种基于二进制代码的、动静态分析相结合的缓冲区溢出漏洞挖掘模型。Wagner方法的基本思想是：对所有的缓冲区建模，将其看作由2个整型数构成的一个“范围”。不跟踪字符串变量的真实内容，而是给每个字符串赋予2个权值：为它分配了多少字节(ALLOC)以及它实际使用了多少字节(LENS)，即字符串实际的长度。所有对字符串s的操作都可以定义为对ALLOC(s)和LENS(s)的操作。这样就可以定义“安全”缓冲区，其原型定义为：$LENS(s) \subseteq ALLOC(s)$。

本文的漏洞挖掘模型在分析二进制反汇编代码的基础上，对Wanger方法的缺陷作了改进：(1)更加精确的变量分析。在动态运行的环境中可以精确地进行指针运算。(2)增加了对程序流程的简单分析。Wanger方法不能对if/while语句的判断条件进行有效的分析，而引入了动态环境后，可以减少误报。具体方案如图1所示，主要分为3部分：中间表示转换，函数扫描引擎和动态运行环境。中间表示转换主要是将汇编指令转换成一棵中间表示树，着重刻画每条汇编中内在操作数之间的关系。函数的扫描引擎是从汇编指令中识别出函数的局部变量和参数的个数、大小(变量的ALLOC属性)、位置。这2部分主要对二进制程序进行静态分析。最后，动态运行环境基于前面2个部分的分析结果，动态运行函数，得到函数的内存读写行为，即读或写了哪些内存单元。如果写入的内存单元范围大小(变量的LENS属性)超过了变量的大小(ALLOC属性)，就可以判定溢出。

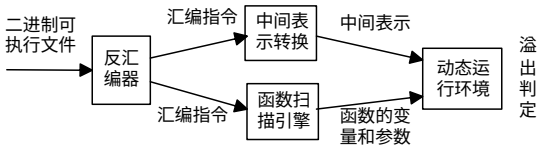


图1 模型的总体方案

2 函数栈帧结构特征

函数的栈帧结构是函数运行时的活动记录，包括函数的参数、函数的局部变量以及一些受保护的寄存器的值(如返回地址)。变量的类型分为值和指针2类。每个变量的位置为离开返回地址的偏移量，偏移量为正时，表示一个参数，偏移量为负时，表示一个局部变量。

变量类型的匹配采用逐步待定的方法。比如，

作者简介：夏 超(1982－)，男，硕士研究生，主研方向：缓冲区溢出漏洞挖掘；邱卫东，副教授
收稿日期：2008-01-04 **E-mail：**rollestar@163.com

op1=i_lea(op0)这句中间操作中有 2 个操作数 op0, op1，由于 lea 指令一般是用于取得地址的，因此 op1 的类型为指针，而 op0 的类型为值。在算术运算中，2 个操作数相加减的类型规则如下：(1)值 ± 值=值；(2)值 ± 指针=指针；(3)指针 ± 指针=非法。通过以上规则，可以对一些不确定的操作数类型进行匹配搜索，当所有的类型都确定并且所有的规则都符合时，类型匹配就完成了。

根据相邻 2 个变量的地址间隔中可以计算出变量的大小。以下面一段代码为例：

```
mov     eax, [ebp + 0x08]
imul    eax, [ebp + 0x0c]
mov     [ebp - 0x04], eax
sub     esp, 8
lea     eax, [ebp - 0x14]
push    eax
```

识别出的栈帧结构如下：

0xbffff0e8	local0	局部变量(值) 缓冲区大小 16 Byte
0xbffff0ec		
0xbffff0f0		
0xbffff0f4		
0xbffff0f8	local1	局部变量(值)
0xbffff0fc	ebp	受保护的寄存器
0xbffff100	eip	受保护的寄存器
0xbffff104	arg0	参数
0xbffff108	arg1	参数

由于逆向工程没有成熟的算法，因此本文的分析无法十分精准[2]。

3 中间表示转换

在一般的编译过程中，要将程序的源代码转换成中间代码，然后根据目标平台的体系结构转换成相应的机器代码。一个编译器开发到中间代码后加上一个中间代码的模拟器，就可以仿真程序的运行。然而，一个良好的中间代码不仅要考虑跨平台的通用性，也要考虑源语言语法和语义对中间代码的支持。

中间表示的设计有很多，典型的如 Halvar Flake 设计。这种设计的好处在于它能够形成一个跨体系结构的虚拟层，而本文主要针对 x86 平台进行研究，更注重分析程序的一般性语义部分。事实上，程序的语义对漏洞的分析更有价值。

如图 2 所示，3 条汇编语句对应一棵完整的中间表示树，事实上，每条汇编语句都对应某一条或几条中间操作。以 mov [eax], 0x41 指令为例，它能够翻译成 2 条中间操作：[eax] = i_mem(eax)和 i_mov([eax], 0x41)，这样的翻译是完整且正确的。

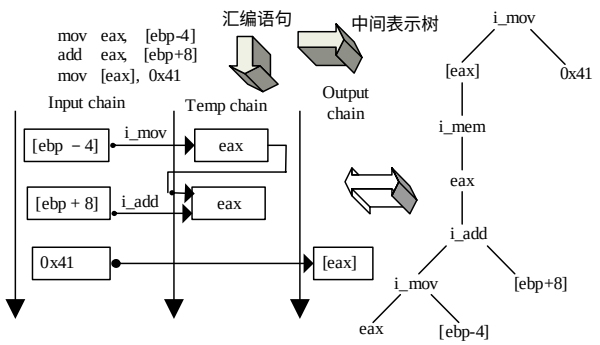


图 2 中间表示转换

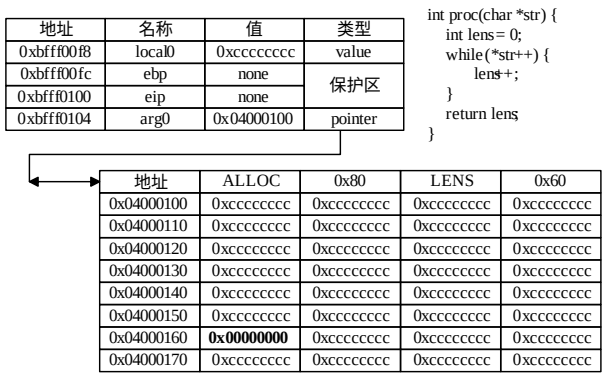
为了体现对内存的读写操作，在实际设计中将中间表示树转化为 3 条链：输入链，临时链和输出链。所有涉及内存读取的操作数及立即操作数均放入输入链，内存写入的操作数均放入输出链，临时链中存放寄存器的临时值。这样既可以与中间表示树作等价转换，也可以在动态模拟运行时为操作数提供存放数据的位置。图 2 显示，这 3 条汇编语句读了 2 个内存单元[ebp-4], [ebp+8]，写了一个内存单元[eax]。

4 动态模拟

在定义程序的中间表示后，便可以定义虚拟执行环境了。本文实现了一个小型的虚拟执行环境，用以仿真 x86 平台程序的运行。但这里只能监控用户空间，系统内核空间还无法监控。这里有一个问题：如果要运行一个函数，就要为它的参数虚拟化，即赋予这个参数一个初始值，让它运行。一个参数有 2 种类型：值和指针，一个值的初始化值为 0x10，而一个指针的初始化需要虚拟段的支持。

一般程序的地址空间按属性划分成段，如代码段、数据段、堆栈段。为了支持虚拟参数的指针运算，在程序运行时增加了一个虚拟段，地址空间从 0x04000100 到 0x04020000。第 1 个虚拟指针参数的值为 0x04000100，以后每个参数的地址增加 0x200，即第 2 个参数的值为 0x04000300。

每个虚拟指针参数都有 2 个属性：ALLOC 和 LENS，ALLOC 属性记录这个虚拟指针所指数据结构的实际空间大小，类似于 sizeof 运算符；LENS 属性指字符串的实际长度。虚拟指针参数指向的地址空间的初始化状态如图 3 中的 arg0，整块缓冲区的大小为 0x80，缓冲区长度为 0x60。这样做既能支持动态执行环境，又能帮助归纳。如在图 3 中，读取的地址空间是[0x04000100-0x04000160]，从而归纳出读取的地址空间是[arg0, arg0+ LENS(arg0)]。这样就可以通过动态运行函数归纳出这个函数的内存读写特征。当处理函数调用时，要将实际参数与虚拟参数绑定。如果其他函数调用函数 proc(string)，就可以知道实际是要读取[string, string+ LENS(string)]这段地址空间。



地址	名称	值	类型
0xbfff00f8	local0	0xccccccccc	value
0xbfff00fc	ebp	none	保护区
0xbfff0100	eip	none	
0xbfff0104	arg0	0x04000100	pointer

```
int proc(char *str) {
    int lens= 0;
    while(*str++) {
        lens++;
    }
    return lens;
}
```

地址	ALLOC	0x80	LENS	0x60
0x04000100	0xccccccccc	0xccccccccc	0xccccccccc	0xccccccccc
0x04000110	0xccccccccc	0xccccccccc	0xccccccccc	0xccccccccc
0x04000120	0xccccccccc	0xccccccccc	0xccccccccc	0xccccccccc
0x04000130	0xccccccccc	0xccccccccc	0xccccccccc	0xccccccccc
0x04000140	0xccccccccc	0xccccccccc	0xccccccccc	0xccccccccc
0x04000150	0xccccccccc	0xccccccccc	0xccccccccc	0xccccccccc
0x04000160	**0x00000000**	0xccccccccc	0xccccccccc	0xccccccccc
0x04000170	0xccccccccc	0xccccccccc	0xccccccccc	0xccccccccc

图 3 虚拟参数示例

5 测试结果

(1)下面是一个程序的测试例子(这里显示的是 C 语言的源代码，实际是对二进制的反汇编代码进行分析)：

```
void test() {
  char str[12];
  view(str, 40); }
void view(char *str, int LENS)
{ int i = 0;
  for(; i < LENS; i++) { str[i] = 'a'; } }
```

其中，归纳出 view 函数的读写内存空间为

mem_write [0x06000100-0x06000110]=[arg0,arg0+arg1]

(下转第 191 页)