江苏大学学报(自然科学版)
JOURNAL OF JIANGSU UNIVERSITY(Natural Science Edition)
2016 年 7 月
July 2016
第 37 卷 第 4 期
Vol. 37 No. 4

doi: 10.3969/j.issn.1671 -7775.2016.04.013

基于源码分析的缓冲区溢出漏洞检测方法

尹 茗, 张功萱
(南京理工大学 计算机科学与工程学院, 江苏 南京 210094)

摘要: 根据缓冲区溢出原因提出一种基于源码分析的缓冲区溢出漏洞检测方法,该方法对源码预处理后进行静态分析并依次构造相应的抽象语法树、控制流图、函数调用图和变量表,最后建立有限状态自动机检测模型. 以容易出现溢出的 C/C++源码为例,构造相应的检测模型,结果表明:该检测模型相比已有检测方案,可以更加有效地检测出缓冲区溢出漏洞;同时,该方法对程序代码中的危险函数调用和溢出过滤机制也能进行有效识别从而降低误报率,该检测方法也适用于其他语言的源码检测.

关键词: 缓冲区溢出检测; 软件开发; 有限状态自动机; 静态源码分析; 蠕虫
中图分类号: TP311 **文献标志码**: A **文章编号**: 1671 -7775(2016)04 -0450 -06

引文格式: 尹 茗, 张功萱. 基于源码分析的缓冲区溢出漏洞检测方法[J]. 江苏大学学报(自然科学版),2016,37(4):450 -455.

Buffer overflow detection method based on source code analysis

YIN Ming, ZHANG Gongxuan
(School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, Jiangsu 210094, China)

Abstract: According to the causes of buffer overflows, a novel detection method was proposed based on source code analysis. The sources were pre-processed and analyzed statically to construct relevant abstract syntax tree, control flow graph, function call graph and variable table in sequence. A finite automata based on the developed detection model was created to detect overflows. The C/C++ program with common buffer overflows was used to demonstrate the proposed method. The extensive experimental results show that compared to existing methods, the proposed detection model can detect all buffer overflow vulnerabilities efficiently. The dangerous function calls and the overflow filtering mechanism in the code can be recognized to reduce false positive rate. The proposed method can also be easily extended to detect the buffer overflows in the codes of other language source.

Key words: buffer overflow detection; software development; finite automata; static code analysis; worm

随着信息化的不断发展,信息技术广泛应用于社会各个领域,随之而来的信息安全问题也越来越多. 从 1988 年第一个 Morris 蠕虫病毒出现到现在,利用缓冲区溢出漏洞进行攻击的案例层出不穷,如 2001 年名为“Code Red”的蠕虫病毒最终导致了全球运行微软的 IIS Web Server 的 30 多万台计算机受到攻击;2003 年“Slammer”(也称为“Sapphire”)蠕虫使得南韩和日本的部分 Internet 崩溃、中断美国航空订票系统[1];2015 年 4 月阿里安全研究实验室发现了安卓系统的一个重大漏洞,并命名为“WiFi 杀手”,该漏洞可导致具有 WiFi 功能且开启 WLAN 直连的安卓设备可被远程执行代码,这些攻击都是

收稿日期: 2015 -10 -09
基金项目: 国家自然科学基金资助项目(61472189)
作者简介: 尹 茗(1991—),女,黑龙江肇东人,硕士研究生(767475027@qq.com),主要从事信息安全研究.
张功萱(1961—),男,江西景德镇人,教授,博士生导师(gongxuan@njust.edu.cn),主要从事分布式系统与服务计算、智能服务与云计算、应用密码学与信息安全研究.

利用了缓冲区溢出这一程序缺陷；此外，还有很多利用 Shellcode 进行远程溢出攻击的案例[2].

2010 年 CWE 组织列举了作者们认为最严重的 25 种代码错误，也是软件最容易受到的攻击点，经典的缓冲区溢出排在了第 3 位，即在没有检测输入大小的情况下对缓冲区进行复制. 从 CWE 提供的 21 个和缓冲区相关的条目来看，造成缓冲区溢出的原因主要有 3 点：粗心编程、调用不安全的库函数和语言自身的某些缺陷. 根据 CWE 提供的缓冲区相关条目，缓冲区溢出大致分类如图 1 所示.

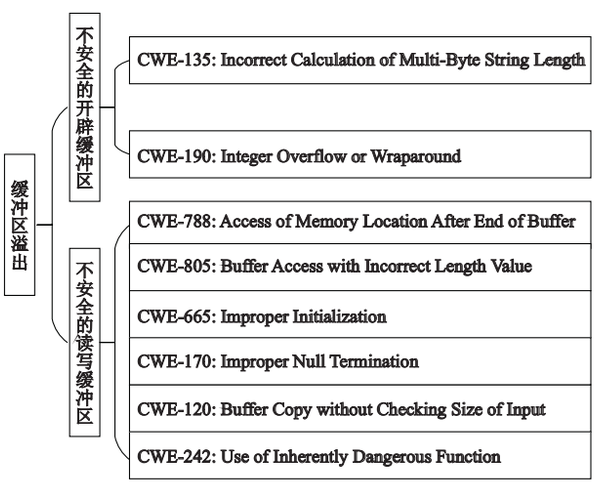


图 1　缓冲区溢出分类

现有的编程语言中，高级语言的溢出较少，最容易出现缓冲区溢出的是 C/C ++ 语言. 在 C 和 C ++ 语言中，对缓冲区的操作通常是诸如 malloc( ) 和 new( ) 这样的内存分配以及一些拷贝函数. 此外，常见的外部输入函数 gets，read 等在操作时由于对输入的长度没有限定，也容易产生溢出. 表 1 列举了部分比较容易引起溢出的危险函数.

表 1　容易引起溢出的危险函数(部分)

类型	缺陷	函数
字符串	无限制	strpy( )，strcat( )，sscanf( )，vsscanf( )，sprint( )，vsprintf( )
	伪长度	snprintf( )，vsnprintf( )，strncpy( )，strncat，memcpy( )，memmove( )，bcopy( )，memccpy( )，memset( )，bzerp( )
文件	无限制	gets( )，fscanf( )，scanf( )，vscanf( )，vfscanf( )
	伪长度	fgets( )，read( )，fread( )
网络		recv( )，recvfrom( )

当前的静态检测方法存在很多误报和漏报，原因是不完备的区间计算和不完备的上下文信息[1]. 文中提出了一种基于源码分析的静态检测方法，该方法首先对源码进行静态分析，生成相应的抽象语法树、控制流图、函数调用图、变量表、函数摘要等，在静态分析过程中提取所有变量的分配空间和使用空间的区间范围，并根据控制流图不断更新区间数据，根据模型检测理论，构造相应的有限状态自动机. 为更精准地检测，本方法对上下文信息进行有效的处理，对程序中出现的缓冲区溢出过滤机制进行检测，从而减少误报. 最后以 C/C ++ 源码为例，建立相应的检测模型，并针对不同危险函数制定不同的比较方法，提高了检测精度.

1　相关研究

目前缓冲区溢出漏洞检测方法大致分成动态检测[1,3]、静态检测[4-9]和动静结合检测[10-12]3 种.

动态检测顾名思义就是在程序执行的同时进行检测，当前的动态检测方法有：① 基于二进制平台，通过代码插桩技术和动态污点传播技术实现对缓冲区溢出的自动化检测，如俞许[1]基于 Pin 的动态插桩平台设计了缓冲区溢出检测的原型系统 bptrace；② 通过修改编译代码，动态检测缓冲区是否发生溢出，如董鹏程等[3]提出基于 DynamoRIO 平台，利用插桩技术，针对不同溢出覆盖类型，通过异常捕获、控制流分析和内存状态检查实现了对缓冲区的自动化检测. 动态检测相对来讲很少有误报，但对二进制代码的识别难度比较大，会有定位不准确、漏报率高的缺点.

静态检测是通过直接扫描程序代码，提取程序关键语法，解释其语义，理解程序行为，根据预先设定的漏洞特征、安全规则等检测溢出漏洞. 静态代码分析工具包括 Fority，BOON，FlawFinder 等[1]，王雅文等[5]是对静态代码进行分析检测，提取变量信息进行比较，漏报较少，但没有对程序中已有的溢出过滤机制进行检测，存在误报情况. 此外，模型检测也可以用于静态检测中，模型检测是一种很重要的自动验证技术，目前各种检测工具也会用到模型检测. 胡定文等[4]针对缓冲区溢出条件，提出了相应的有限状态自动机的漏洞检测模型，检测过程中仅仅对源缓冲区和目的缓冲区大小进行比较，对于空间大小不确定的变量，输出结果为可疑，这样无疑又增大了人工分析的工作量.

正是考虑到动态检测和静态检测的缺点，文献[11]提出了一种动静结合的检测方法. 该方法先将二进制文件转换成汇编语言，对汇编代码进行静态分析，测出潜在的溢出点，再使用 OllyDbg 调试器进

行动态测试，输出溢出点位置. 这种方法适用于在没有源码的情况下挖掘溢出漏洞，降低了误报率，但使用 OllyDbg 对检测人员有一定的限制，需要检测人员对汇编代码有很深的了解.

2 缓冲区溢出漏洞检测方法及模型

图 2 为缓冲区溢出检测的整体流程. 首先对源程序进行预处理，包括生成抽象语法树、控制流图、函数调用图以及建立函数摘要和变量表，再根据缓冲区溢出原因和自动机原理建立缓冲区溢出漏洞检测模型，对控制流图基本块进行扫描，得到溢出点位置.

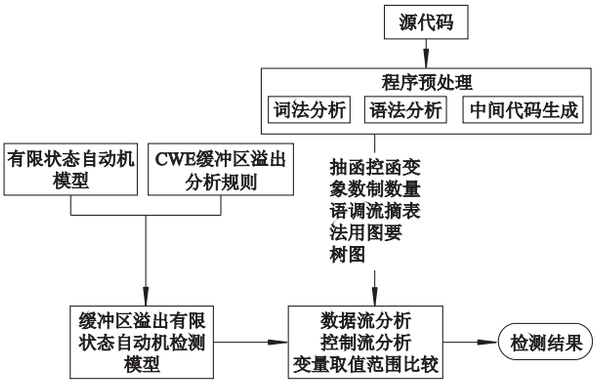


图 2 缓冲区溢出检测整体流程

2.1 源程序预处理

2.1.1 抽象语法树

抽象语法树(abstract syntax tree，AST)是程序的一种中间表现形式，能够比较直观地表示源程序的语法结构，含有源程序结构显示的所有静态信息，具有较高的存储效率. GCC(the gnu compiler collection)是由美国自由软件基金会开发的编译组件，它能够支持 C，C ++，Java 等程序设计语言. GCC 以源程序为单位生成 AST，而且包含整个编译单元的完整表示，文中第一步就是通过 GCC 编译器生成 AST，对代码进行预处理，并进行静态分析.

由于 GCC 产生的是文本抽象语法树，包含很多编译信息，如由#include 命令产生的内部函数、类型声明、出错信息、常量等，不利于直接进行代码分析，所以先进行优化抽象语法树，先消除抽象语法树中的冗余信息，如与分析数据流和控制流无关的节点，得到规范化的抽象语法树，再消除冗余字段(节点中与定位无关的字段信息).

2.1.2 控制流图

由于抽象语法树无法有效地表示程序的控制结构，因此需要生成控制流图(control flow graph，CFG). 控制流图是以基本块为结点的有向图 $G=(N,E)$，其中 N 是表示程序中基本块结点集合，E 是边的集合. 如果从块 u 的出口转向块 v，则从 u 到 v 有一条有向边 $u\rightarrow v$，表示 u 到 v 存在一条可执行路径，也就代表在执行完结点 u 中的代码语句后，有可能顺序执行结点 v 中的代码语句. 基本块是一个最大化的指令序列，程序执行只能从这个序列的第一条指令进入，从这个序列的最后一条指令退出.

控制流图是后续工作的基础，它模拟程序的执行路径，根据操作类型提取相关的变量信息，对于控制流图中的不可达路径可以忽略，减少误报.

2.1.3 函数调用图与函数摘要

为了避免反复提取信息，提高函数分析效率，这里需要用到函数调用图与函数摘要.

函数调用图是对程序进行整体分析而生成的有向图，它是对函数间关系的静态描述，提供了一种函数之间调用关系的形象化表示方法. 可以形式化表示为 $G=(V,E)$. 其中 V 代表某函数的调用图结点集合；E 为有向边集合，代表函数之间的调用关系. $v_1\rightarrow v_2$ 表示 v_1 到 v_2 存在一条有向边，v_1 结点中的函数调用了 v_2 结点的函数. 文中利用函数调用图明确函数间的调用关系，便于分析控制流图.

函数摘要[6]是对函数信息的抽象概括，提取出针对具体问题的信息，而忽略相对冗余的部分. 包含 3 个部分：① 前置条件，如上下文环境、输入参数等；② 函数操作，文中主要涉及到对缓冲区的操作；③ 后置条件，如函数执行后对变量及返回值等带来的影响.

2.1.4 变量表

对于全局变量，由于作用域相同，实质是一个包含所有全局变量的链表. 对于局部变量，每个局部代码块对应一个链表，包含变量指针信息(包括指针名、指向的内存空间 id、类型信息、当前指针在所指内存空间的索引值和下一个元素节点)，以及指针指向的内存信息(包括内存空间 id、空间大小 size、分配方式、被引用次数以及链表中指向的下一个元素节点).

2.2 缓冲区溢出漏洞检测模型

2.2.1 模型检测

模型检测过程是给定一个系统模型，自动验证这个模型是否满足指定的条件或属性. 本质上它是用严密的数学方法来验证一个系统的设计是否满足预先设定的需求，从而自动地发现设计中的错误. 由

于模型检测有着完备的理论基础和实现的自动化，使得模型检测成为目前各种检测工具都会采用的方法. 程序也可看成是一个系统，程序潜在的缓冲区溢出漏洞也可以理解为一个程序的一个错误状态，因此可以将模型检测技术用在程序的漏洞检测上.

2.2.2　有限状态自动机

以自动机的理论来分析一个软件系统，其实就是设计程序从输入开始，之后通过在不同的中间状态之间转移，最后执行到输出结果的一种自动机[12]. 有限状态自动机是为了研究计算机内存执行过程和某些高级语言类而抽象得出的一种计算机模型，其特点是很好地抽象了软件程序在计算机中的执行过程. 通过对软件系统进行静态分析、对功能和流程进行抽象解析、然后通过有限状态自动机进行模拟软件系统的执行流程，静态检测可以获得一定的动态漏洞检测的部分模型推导功能. 定义 1 给出了一个有限状态自动机的形式化定义.

定义 1　有限状态自动机(finite automaton，FA)是一个五元组 $M=(Q,\Sigma,\delta,q0,F)$，其中：Q 为状态的非空有穷集合；$\forall q\in Q$，q 称为 M 的一个状态；Σ 为输入字母表；δ 为状态转移函数，有时又叫作状态转换函数或者移动函数，$\delta:Q\times\Sigma\to Q$，$\delta(q,a)=p$；$q0$ 为 M 的开始状态，也可叫作初始状态或启动状态，$q0\in Q$；F 为 M 的终止状态集合，F 被 Q 包含，任给 $q\in F$，q 称为 M 的终止状态.

2.3　缓冲区溢出漏洞检测步骤

检测之前，对代码进行预处理的过程中提取变量的信息，存在于不同字段中，如分配的缓冲区空间 allocate 和拷贝数据使用空间 used，在遇到拷贝和复制操作时比较字段的大小，来判断是否存在溢出，对于不同的拷贝函数有不同的计算方式.

对于 Strncpy(char *dst, char *src, size n)函数：allocate(dst)=$[x_1,x_2]$，used(src)=$[x_1',x_2']$，若 $n\leqslant x_2-x_1$，则安全，若 $n>x_2-x_1$，且 $(x_2'-x_1')\leqslant(x_2-x_1)$ 则安全，否则，溢出.

对于 Memcpy(char *dst, char *src, size n)函数：allocate(dst)=$[x_1,x_2]$，used(src)=$[x_1',x_2']$，若 $n\leqslant x_2'-x_1'$ 且 $n\leqslant x_2-x_1$ 且 $(x_2'-x_1')\leqslant(x_2-x_1)$ 则安全，否则，溢出.

部分函数在拷贝字符串时会对字符串末尾的\0也同时进行拷贝，这样的函数在计算时就要考虑到“加一”问题，其他函数判断方法如表 2 所示(allocate(*str*)与 used(*str*)分别表示变量 *str* 的分配空间和使用空间).

表 2　部分函数溢出判断

函数	判断方法
strcpy(*dst*, *src*)	allocate(*dst*) > used(*src*)
sscanf(*dst*, “%s”, *str*)	allocate(*dst*) > used(*src*)
memcpy(*dst*, *src*, *count*)	allocate(*dst*) > min(used(*src*), *count*)
gets(*buf*)	allocate(*buf*) = [0, ∞)(本身存在漏洞)
fread(*buff*, *size*, *count*, *fp*)	allocate(*buff*) > *size* * *count*
recv(*sock*, *buff*, *size*, *flag*)	allocate(*buff*) > *size*

现在研究人员也已经注意到这些溢出问题，所以在写代码时加入了溢出过滤机制. 如进行比较后再拷贝，这段代码不会产生溢出. 这种具有溢出过滤机制的代码就不能按照传统的检测方式进行. 因此新增一个字段指针 new_interval，初始值为 *null*，检测到比较缓冲区大小语句时，如 strlen 函数和 if 语句，指针指向一个新的区间，此区间内是新的使用空间大小，最后进行溢出判断是将新区间的空间大小和分配空间做比较. 对于像 gets 这样的外部输入函数，没有自己的安全保护机制，输入时数组长度是 0 至无穷大，而分配的缓冲区空间是一定的，那么它本身就存在漏洞. 在图 3 所示的代码段中，第 2 行使用 gets 外部输入函数是存在溢出漏洞的，因为 gets 本身没有对输入的字符串 *str* 的长度进行检测，但是不能断定以后对 *str* 的所有操作都是有漏洞的，若代码中存在拷贝前判断(如第 3 行)，那么就不一定存在漏洞(这里不考虑外部输入的 *str* 足够大而造成的整型溢出情况).

```
1. char str = new char[30]; char str1 = new char[20];
2. gets(str);
3. if(strlen(str) <20)
4.        strcpy(str1, str);
5. else ……
```

图 3　一段具有 gets 危险函数的代码段

下面以图 3 所示代码为例分析其是否存在漏洞(new_interval 指向新申请空间)，若程序中先把获取的字符串长度先赋给某个变量，new_interval 指向的空间用这个变量来表明，便于以后更新数据.

第 1 行：allocate(*str*) = [0, 30], used(*str*) = [0, 0], new_interval(*str*) = *null*;

allocate(str1) = [0, 20], used(str1) = [0, 0], new_interval(*str1*) = *null*.

第 2 行：allocate(*str*) = [0, 30], used(*str*) = [0, ∞), new_interval(*str*) = *null*, 存在溢出漏洞.

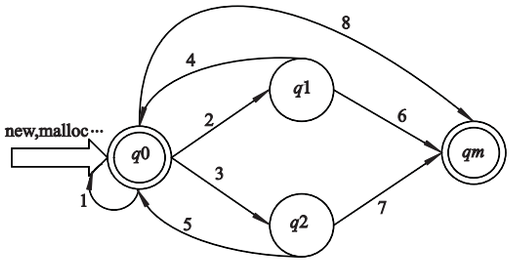
第 4 行：allocate(*str*) = [0, 30], used(*str*) =

$[0,\infty)$, new_interval(str) $\geqslant [0,20]$.

比较 allocate($str1$) 和 new_interval(str) 指向区间的大小，不溢出.

将模型检测和自动机理论应用于缓冲区溢出检测中，在一定条件下变量的状态会发生转换，在变量处于某个不安全的状态时，则认为程序存在缓冲区溢出漏洞. 结合以上分析建立了有限状态自动机 $M=(Q,\Sigma,\delta,q0,F)$，其中：$Q=\{q0,q1,q2,qm\}$ 为有限状态集合；Σ 为有限字符表，包括待分析源代码、缓冲区分配空间 allocate(a)，拷贝数据使用空间 used(u)，对输入无穷大设置的新字段区间长度 new_interval(n)；δ 为状态转移，$f(x,y)$ 为缓冲区溢出的判断表达式，$f(x,y)=1$ 代表 $x>y$，缓冲区不溢出，$f(x,y)=0$ 则表示缓冲区溢出；$q0$ 为初始状态；$F=\{q0,qm\}$ 为终止状态集合，$q0$ 为安全状态，qm 为不安全状态.

其自动机模型如图 4 所示.



1：无拷贝或复制操作；2：无边界检测的拷贝操作，如 strcpy，memcpy，strcat 等；3：有边界检测的拷贝操作；4：$f(a,u)=1$；5：$f(a,n)=1$；6：$f(a,u)=0$；7：$f(a,n)=0$；8：外部输入范围无穷大，且 n 为 $null$

图 4　缓冲区溢出漏洞的自动机检测模型

根据自动状态机模型显示，当声明分配空间时（如 new 或 malloc 函数等）触发状态机，进入状态 $q0$，无拷贝或复制操作，状态仍为 $q0$；若有拷贝操作，且 new_interval 字段为 $null$，则表示外部输入无限定，状态转到 qm；若有拷贝操作且没有边界检测（如 strcpy，memcpy 函数等），则状态转为 $q1$；若有边界检测的拷贝操作，则状态转为 $q2$；若 $f(a,u)=1$ 或 $f(a,n)=1$，状态转为 $q0$，$q0$ 为安全接受态；若 $f(a,u)=0$ 或 $f(a,n)=0$，则状态转为 qm，qm 为不安全接受态.

3　试验与分析

选取 2 段测试代码，第 1 段测试代码（图 5）是由于不安全的调用危险函数产生了缓冲区溢出，并且在检测过程中容易因为区间运算的不完备而产生漏报；第 2 段测试代码（图 6）具有溢出过滤机制，即便调用危险函数也没有发生溢出，但在检测过程中容易因为上下文信息的不完备而产生误报.

```
1.  #include < string. h >
2.  int main( )
3.  {
4.      char * str1 = new char[10];
5.      char * str2 = new char[10];
6.      int * num1 = new int[8];
7.      int * num2 = new int[6];
8.      str2 = "123456789";
9.      char * str3;
10.     char * str4 = new char[10];
11.     str3 = str1 + 4;
12.     strcpy(str3,str2);    //溢出
13.     str4 = "12345";
14.     strcat(str4,str2);    //溢出
15.     memcpy(num2,num1,3);                    //无溢出
16.     return 0;
17.   }
```

图 5　由调用危险函数引起的缓冲区溢出代码

```
1.  #include < string. h >
2.  int main( )
3.  {
4.      char * str1 = new char[4];
5.      str1 = "abcd";
6.      char * str2 = new char[10];
7.      str2 = "123456789";
8.      char * str3 = new char[6];
9.      if( strlen(str1) < strlen(str3) )
10.     {
11.       strcpy(str3,str1);                    //无溢出
12.     }
13.     if( strlen(str2) < strlen(str3) )
14.     {
15.       strcpy(str3,str2);                    //无溢出
16.     }
17.     return 0;
18.   }
```

图 6　容易产生溢出误报的代码

试验结果如表 3 所示，表中列出了被测代码存在的缓冲区溢出漏洞数量和存在溢出的位置.

表 3　试验结果

试验	溢出数量	溢出位置
1	2	12,14
2	0	0

对于第 1 段代码，文中提出的方法成功检测出其 12，14 行存在溢出；而对于第 2 段代码，可以成功

检测出含过滤机制的代码，没有产生误报. 而相比本方法，已有方法包括 DTS，K8 等都无法处理存在过滤机制导致类似第 2 段代码的情况下产生误报[4-5].

现有方法产生误报和漏报的原因包括：① 不完备的区间运算导致很多方法对溢出进行判断时仅仅是用源缓冲区大小和目的缓冲区大小进行比较，没有考虑到缓冲区已经被使用过这种情况，这样就会产生漏报；② 不完备的上下文信息导致在计算过程中没有考虑路径信息，即带比较的数据拷贝，也是文中说的溢出过滤机制，这样就会产生误报. 文中提出的方法中储存的变量信息不是缓冲区大小，而是缓冲区的可用区间，这样就能够明确判断是否存在溢出；同时，通过对带有溢出过滤机制的代码进行有效检测，结果表明本方法提高了准确率.

4　结　论

提出了一种缓冲区溢出漏洞检测方法. 该方法首先对源代码进行预处理，提取信息，消除冗余，并建立缓冲区溢出漏洞分析规则和有限状态自动机模型，根据模型检测原理进行检测. 试验证明，该检测方法有效地解决区间运算和上下文信息的不完备问题，对源程序中的危险函数和溢出过滤机制进行了有效的识别，降低了误报率并提高检测的准确性. 目前检测中没有考虑由于格式化字符串产生的溢出，以后将会做进一步研究.

参考文献(References)

[1] 俞许. 二进制代码缓冲区溢出检测技术研究[D]. 南京：南京大学，2012.

[2] 张之刚，周宁，牛霜霞，等. 远程缓冲区溢出攻击及防护[J]. 重庆理工大学学报(自然科学)，2010，24(11)：80－84.
ZHANG Z G，ZHOU N，NIU S X，et al. Remote buffer overflow attack and prevention [J]. Journal of Chongqing University of Technology(Natural Science)，2010，24(11)：80－84. (in Chinese)

[3] 董鹏程，舒辉，康绯，等. 基于动态二进制平台的缓冲区溢出过程分析[J]. 计算机工程，2012，38(6)：66－68.
DONG P C，SHU H，KANG F，et al. Process analysis of buffer overflow based on dynamic binary platform[J]. Computer Engineering，2012，38(6)：66－68. (in Chinese)

[4] 胡定文，朱俊虎，吴灏. 基于有限状态自动机的漏洞检测模型[J]. 计算机工程与设计，2007，28(8)：1804－1806.
HU D W，ZHU J H，WU H. Vulnerability detection model based on finite automata[J]. Computer Engineering and Design，2007，28(8)：1804－1806. (in Chinese)

[5] 王雅文，姚欣洪，宫云战，等. 一种基于代码静态分析的缓冲区溢出检测算法[J]. 计算机研究与发展，2012，49(4)：839－845.
WANG Y W，YAO X H，GONG Y Z，et al. A method of buffer overflow detection based on static code analysis [J]. Journal of Computer Research and Development，2012，49(4)：839－845. (in Chinese)

[6] DING S，TAN H B K，LIU K P，et al. Detection of buffer overflow vulnerabilities in C/C + + with pattern based limited symbolic evaluation[C]//Proc of the 36th Computer Software and Applications Conference. [S. l.]：IEEE，2012：559－564.

[7] 徐有福，文伟平，万正苏. 基于漏洞模型检测的安全漏洞挖掘方法研究[J]. 信息网络安全，2011(8)：72－75.
XU Y F，WEN W P，WAN Z S. Vulnerability-based model checking of security vulnerabilities mining method [J]. Netinfo Security，2011(8)：72－75. (in Chinese)

[8] LIU X，CAI W D. A program vulnerabilities detection frame by static code analysis and model checking[C]// 2011 IEEE 3rd International Conference on Communication Software and Networks. Xi′an：IEEE，2011：130－134.

[9] KANG F，DONG P C，SHU H，et al. Process analysis of buffer overflow based on dynamic binary platform[C]// The 2nd International Conference on Computer Application and System Modeling. Paris：Atlantis Press，2012：1056－1059.

[10] HAUGH E，BISHOP M. Testing C programs for buffer overflow vulnerabilities[J]. Proceedings of the Network & Distributed System Security Symposium，2002，17(3)：411－423.

[11] YUAN J B，DING S L. A method for detecting buffer overflow vulnerabilities[C]//2011 IEEE 3rd International Conference on Communication Software and Networks. [S. l.]：IEEE，2011：188－192.

[12] 陈文宇. 形式语言与自动机理论若干问题研究[D]. 成都：电子科技大学，2009.

(责任编辑　祝贞学)