一种基于模型检验的缓冲区溢出检测方法

张 媛[1,3] 于冠龙[2] 李仁见[3]
(65042 部队 沈阳 110001)[1] (海军驻沈阳地区航空军事代表室 沈阳 110031)[2]
(国防科学技术大学计算机学院 长沙 410073)[3]

摘 要 缓冲区溢出已经成为程序漏洞的主要根源之一。目前存在的缓冲区溢出检测方法或多或少地都存在着不足,从而限制了这些方法的实际应用。深入研究了当前缓冲区溢出检测方法的优缺点,对程序中的缓冲区及其相关操作进行建模,设计了一种基于模型检验的缓冲区溢出检测方法,并开发了一个原型工具来对该方法进行初步验证。在剖析缓冲区溢出基本原理的基础上,对程序中的缓冲区及其相关操作建立了理论模型,设计了一种基于模型检验的缓冲区溢出检测方法。最后,实现了一个原型工具来对该方法进行初步验证。

关键词 缓冲区溢出,模型检验,人工分析
中图法分类号 TP309 **文献标识码** A

Buffer Overflow Detection Method Based on Model Checking

ZHANG Yuan[1,3] YU Guan-long[2] LI Ren-jian[3]
(The 65042nd Unit of PLA, Shenyang 110001, China)[1]
(Naval Aeronautic Military Representatives Office in Shenyang, Shenyang 110031, China)[2]
(School of Computer Science, National University of Defense Technology, Changsha 410073, China)[3]

Abstract Buffer overflow has become one of the major sources of program flaws. All of the existing solutions for detecting buffer overflow have serious drawbacks that hinder their wider adoption by practitioners. In this paper, we researched those solutions, analyzed their advantages and disadvantages. We presented a model for buffers in the program and related operations. At last we designed a buffer overflow detecting approach based on model checking. A prototype tool was also developed to verify this method.

Keywords Buffer overflow, Model checking, Artificial analysis

1 引言

缓冲区溢出是软件安全中的一个主要漏洞根源。据统计,缓冲区溢出导致的安全问题占所有安全问题的 50%以上[1]。已有的缓冲区溢出检测方法包括堆栈不可执行、对源代码的静态分析、改进编译器、使用安全的编程语言、使用更安全的函数库等,但它们或多或少的都存在着限制其实际应用的缺陷。本文试图用模型检验的方法来检测缓冲区溢出。

2 缓冲区溢出检测模型

为证明程序不存在缓冲区溢出,只需要证明每次缓冲区访问都不超过缓冲区边界,而不必关心访问缓冲区时具体的内容。为此,给出如下定义。

定义 1 缓冲区状态 $buffer\_state$ 是一个三元组 $buffer\_state=\langle buff, length, size\rangle$,表示缓冲区 $buff$ 运行时的一个状态;$buff$ 表示该缓冲区,$length\in N$ 为自然数,表示变量预分配大小;$size\in N$ 为非负整数,表示缓冲区的动态位置。

本文采用 SB_buff 表示输入空间中缓冲区 $buff$ 的状态集合。那么程序运行时缓冲区在其生存期的中任一时刻可以用其状态集合中的一个元素来表示。

定义 2(缓冲区操作) $OP: SB\_buff \rightarrow SB\_buff$ 为 SB_buff 到 SB_buff 的映射,表示缓冲区 $buff$ 状态间的一个迁移。

注意这样定义缓冲区操作之后,操作缓冲区的程序语句以及相关函数都属于缓冲区操作的范畴。

定义 3 检测空间上任意缓冲区 $buff$ 状态集上的偏序关系 $RB(SB\_buff, <)$ 表示对于 SB_buff 中的任一元素 $sb=\langle buff, length, size\rangle$ 如果满足 $size<length$,就有 $sb\in RB$。

经过这样的定义之后,可以得到如下的结论。

结论 1 输入空间 P 中源程序不存在缓冲区溢出当且仅当 P 中所有缓冲区的任意状态 $sb\in RB$。

对于任意缓冲区,若有某个状态 $sb\notin SB$,则该状态下 $size\geqslant length$,此时访问的缓冲区位置已超出缓冲区的预分配大小,造成缓冲区溢出。否则,若所有缓冲区状态都有 $sb\in RB$,则其访问的缓冲区位置始终位于缓冲区的容量之内,不会造成缓冲区溢出。

下面通过一个简单的例子来看一下程序中缓冲区模型中状态的变化。

张 媛(1982—),女,硕士,工程师,主要研究方向为网络体系结构、分布式计算,E-mail:ruby820111@126.com;于冠龙(1983—),男,学士,工程师;李仁见(1980—),男,博士生,主要研究方向为程序正确性分析与验证。

例 1　假设在文件 file. c 中有如下的代码：

```
void func() {
    char str[15];
str[10]='a';
strcpy(str,"buffer overflow!");
}
```

在例 1 中，首先定义了一个缓冲区，然后进行了数组基本赋值操作，此时不会发生缓冲区溢出。但是最后调用了 strcpy 函数拷贝字符串。由于目标缓冲区的大小小于源字符串，因此会发生缓冲区溢出。整个过程中缓冲区的状态变化可以表示如下：

$\langle str,15,0\rangle \rightarrow \langle str,15,10\rangle \rightarrow \langle str,15,16\rangle$

可以看出在第 3 个状态，由于(size=16)>(length=15)，因此发生了缓冲区溢出。

3　缓冲区相关操作建模

由于缓冲区相关操作直接影响到缓冲区状态的变化，因此，必须准确地针对缓冲区相关操作进行建模。但是由于 C 语言操作缓冲区时很灵活，因此实现过程中，必须在可操作性和准确性以及效率等因素之间做一个折中。实际程序中涉及缓冲区的操作很多，对一些情况的特殊处理会显著提高系统的性能和运行效率，下面将分情况进行讨论。

针对基本的数组操作，把操作看成一个简单的状态引用和状态改变。由于缓冲区溢出检测不关心具体的数据流，因此，不必关心操作的具体内容，可以简单地进行缓冲区建模，例如上面例 1 中第二处引用字符串 str 时访问的状态是$\langle str,15,10\rangle$，并没有出现缓冲区溢出。

C 语言中缓冲区通常通过指针进行访问和操作，因此，必须对指针进行建模。如果忽略了指针分析，则整个模型将是非常不精确的。

在程序的不同执行点，指针变量可以指向不同的缓冲区。为此，提出指针指向集的概念，用来保存一个指针变量可能指向的缓冲区的结合。

定义 4(指针变量的指向集)　指针变量 p 的指向集 $PointTo\_p$ 是一个指针可能指向的缓冲区的集合。

在进行指针指向集合的计算中，需要指针分析的知识。指针分析是目前的研究热点，出现了很多经典的指针分析算法[2]，在此不再专门讨论这个问题。下面讨论对普通指针建模的处理。

结论 2　一个指针的指向集中所有元素均满足结论 1，则该指针不会造成缓冲区溢出。

在针对程序的建模过程中，如果通过指针访问了缓冲区，那么需要针对指针分析算法对指针的指向集进行相关操作，并根据相关建模语义对指针指向集中的元素进行对应的操作。

例 2　假设有如下的代码：

```
void func() {
    char str[15];
    char * p;
    p=str;
    strcpy(str,"buffer overflow!");
}
```

针对该例中的代码，本文所得到的建模操作如表 1 所列。该例定义了一个字符指针 p，然后让其指向缓冲区 str，最后通过指针 p 对缓冲区进行拷贝操作。从上面的建模过程可以看出对指针 p 的操作发生了缓冲区溢出。

表 1　例 2 中代码对应的建模操作

代码	对应的建模操作
char str[15];	_str_. length=15;_str_. size=0;
char * p;	PointTo_p={ };
p=str;	PointTo_p={_str_};
strcpy(p,"buffer overflow!");	_str_. size=16;

对于形如 $p=f+i$ 的情况，本文不考虑，而是简单地按照 $p=f$ 的情况进行建模，这种简化看似不合理，其实可以处理大多数的情况，提高了实现效率，对于模型的精确程度又没有太大的影响。

对于标准字符串操作函数，由于标准函数库中的函数的操作语义是具体不变的，这有助于建模，因此只需要针对相应的函数操作语义进行建模即可。C 语言提供的库函数很多，在此不能全部列出。表 2 中仅仅列出了常见的几个字符串操作函数的建模过程。

表 2　C 语言常见字符串操作函数建模

C 库函数	模型操作
p=(char *) malloc(n);	_p_. length=n;_p_. size=0;
strlen(buf)	_buf_. size=0;
strcpy(str,buf);	_str_. size=strlen(buf);
strcat(str,p);	_str_. size= strlen(str)+strlen(p);
gets(buf);	_buf_. size=65535;//本文可以认为是∞
p=strdup(str);	_p_. length=strlen(str);_p_. size=strlen(str);

4　基于模型检验的缓冲区溢出检测原型工具——MCBuffer

基于模型检验的基本思路，本文设计了一个检测 C 程序中缓冲区溢出问题的原型工具 MCBuffer(Model Checking Buffer Overflow)。

MCBuffer 基于模型检验来检测 C 程序中的缓冲区溢出问题。本文把输入的 C 语言源程序通过 Modex 抽取 C 语言模型，在此过程中针对缓冲区及其相关操作进行建模，最后以 Promela 表示程序，输入模型检验工具 SPIN，进行模型检验，检查缓冲区溢出。如果检查到缓冲区溢出，则通过反例验证器，确定这个反例是否为缓冲区溢出。然后将确定的缓冲区溢出错误报错，并将怀疑的缓冲区溢出反例重新进行建模并再次进行检测。图 1 描述了整个过程，其中阴影部分是本文进行研究和实现的重点。下面详述各个重要步骤的设计和实现思路。

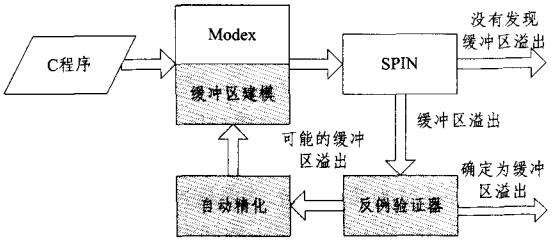

图 1　MCBuffer 结构框图

4.1　源程序模型抽取

模型检验的工具语言往往具有一定的特点，在描述某些类型的系统时有着很好的适应性，比如 Promela 特别适合于

描述并行系统，而 SMV 则比较适合于描述硬件系统的性质。如何针对 C 语言的源程序进行模型检验，一直以来都是一个重要的前沿课题。

Modex[3]是 Bell 实验室开发的一个 C 语言模型抽取工具，它主要是依靠 Flex 进行词法分析，然后利用 Bison 建立语法分析器，进行语法分析，在这个过程后获取建立的语法树，并进行内部表示，然后针对语法树进行操作，在屏蔽尽量少的信息的条件下，把语法树翻译成 Promela 表示。具体的实现可以参考 Modex 的相关手册以及源代码，在此不再进行详细描述。

Modex 在实现时可以利用默认配置，自动地进行建模，也可以结合配置文件，手工辅助建模。系统在默认的抽象层次下能够满足大部分系统的建模验证需求，但是操作人员也可以增加抽象层次，从而更加精确地对系统进行建模分析验证。

4.2 缓冲区建模

缓冲区是系统分配的一片内存区域，在进行缓冲区操作时，如果写入或者读取的内容大小超过了缓冲区预分配的大小，就会发生缓冲区溢出，这就是缓冲区溢出的基本原理。通过上面的建模分析，本文使用一个抽象数据结构 buf_model 来记录系统内部的缓冲区。系统中分配的每个缓冲区 buff 都使用一个变量_buff_记录其所在文件、函数、缓冲区名字、预分配长度、当前操作大小。

```
struct buf_model{
    char file[100];/*所在文件*/
    char func[100];/*所在函数*/
    char name[100];/*缓冲区名字*/
    int length;/*预分配长度*/
    int size;/*当前操作大小*/
}
```

本文对缓冲区相关操作进行建模，在缓冲区操作的基础之上加入模型操作语句。将所有的缓冲区保都存在一个表内，跟踪缓冲区相关操作并对其进行建模，在可能产生缓冲区溢出的操作之后，判断是否仍然满足结论 1，如果不满足，那么可以断定发生了缓冲区溢出，则会在 SPIN 中报错。具体实现就是在缓冲区操作之后，判断是否仍然有 assert(_buff_.size<_buff_.length)成立。

4.3 验证引擎 SPIN

本文方法中，处于工具 MCBuffer 核心的是 Bell 实验室的模型验证工具 SPIN[4]。SPIN 以 Promela 作为模型的描述语言，通常，操作人员要对系统采用 Promela 语言手工进行建模，系统的性质利用时序逻辑来表示，然后 SPIN 就可以穷举地或在既定时间、既定内存限制内进行模型验证。如果验证的系统和模型满足性质，就输出 OK，否则可以打印反例路径。

直接从源代码中抽取模型，而不对源语言施加某些语言或者语法应用上的限制，使 SPIN 真正地应用到软件开发过程中。在 C 语言到 Promela 的转换过程中，两种语言在很多的情况下并不完全等价，直接转换会存在一定的困难，效率也不会很高。为此，本文充分利用最新版本的 Promela 语言中提供的 c_code 与 c_expr 等机制，把许多在 Promela 中比较难以表示的模型操作以嵌入式 C 代码的形式进行操作，试验表明这种方法能够起到很好的作用。

4.4 反例验证器

如果 SPIN 认为程序中没有缓冲区溢出漏洞，那么整个检测过程结束；否则，需要对检测到的缓冲区溢出操作进行确认，查看是否是真正的缓冲区溢出。这项工作由反例验证器负责。如果确认是真正的缓冲区溢出，则将错误所在的位置报告给用户；如果确认是误报，则需要重新精化模型，再次进行检测。在这个过程中需要用到程序解释以及反例解释的知识和相关工具。

4.5 自动精化

利用模型检验的方法来进行缓冲区溢出检测时，模型的自动精化是关键的一个部分。误报率高、所有的可疑缓冲区溢出漏洞都需要人工分析来进行检查，这是一般静态方法难以克服的问题。而模型检验通过模型自动精化过程处理这些问题。当然这个过程中，自动程度的高低是一个非常重要的指标，如果需要大量的人工参与，那么该方法的优势就不能有效体现。同反例验证器一样，模型的自动精化需要有效的反例解释算法和工具加以支持，这也是目前的一个研究热点。

4.6 缓冲区溢出检测

为了充分说明 MCBuffer 的工作过程，下面以一个简单的缓冲区溢出检测的例子来进行说明。

例 3　example3.c

```
void main(){
    char buf[10];
    char *p;
    p="Buf Overflow Checking!";
    strcpy(buf,p);
}
```

表 3　例 3 转化之后的 Promela 代码

```
/*全局变量*/
c_state "buf_model _p_" "Local main"
c_state "buf_model _buf_" "Local main"
c_state "char * p" "Local main"
c_state "char buf[10]" "Local main"
/*数据结构声明*/
c_decl {
  typedef struct buf_model{
    char file[100];          /*所在文件*/
    char func[100];          /*所在函数*/
    char name[100];          /*缓冲区名字*/
    int length;              /*预分配长度*/
    int size;                /*当前操作大小*/
  }buf_model;
}
active proctype main()
{ c_code { strcpy(Pmain->_buf_.file, "example.c");};
c_code {strcpy(Pmain->_buf_.func,"main");};
c_code {strcpy(Pmain->_buf_.name,"buf");};
c_code {Pmain->_buf_.length=10;};
c_code {Pmain->_buf_.size=0;};
c_code {strcpy(Pmain->_p_.file,"example.c");};
c_code {strcpy(Pmain->_p_.func,"main");};
c_code {strcpy(Pmain->_p_.name,"p");};
c_code {Pmain->_p_.length=0;};
c_code {Pmain->_p_.size=0;};
c_code {Pmain->p="Buff overflow checking"; };
c_code {Pmain->_p_.length=strlen("Buff overflow checking");};
c_code {Pmain->_p_.size=strlen("Buff overflow checking");};
c_code [(Pmain->_p_.length>=Pmain->_p_.size)]{ ; };
c_code {strcpy(Pmain->buf,Pmain->p);};
c_code {Pmain->_buf_.size=Pmain->_p_.size;};
c_code [(Pmain->_buf_.length>=Pmain->_buf_.size2)]{ ; };
{;};
}
```

这个例子中由于 p 的大小超过了 buf 的限制，在调用 strcpy 函数之后，肯定会发生缓冲区溢出。表 3 中列出了 MCBuffer 转化生成的 Promela 代码，可以直接利用 SPIN 进行验证。表中黑斜体部分表示在 SPIN 中需要进行确认和验证的属性。如果这些性质不满足，则说明源程序中相关的缓冲区操作存在缓冲区溢出。

图 2 是例 3 通过 MCBuffer 检测的执行结果。可以看出，MCBuffer 成功地检测到了源代码中存在的缓冲区溢出问题。

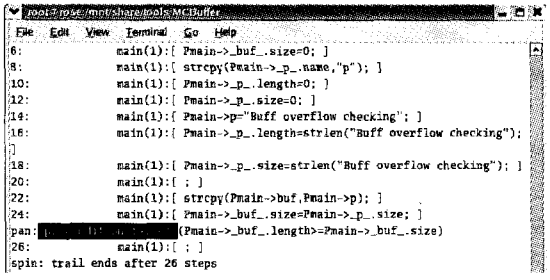

图 2　MCBuffer 检测结果

结束语　传统的静态方法误报较多，所有的可能漏洞都需要人工进行分析[5]；基于模型检验的缓冲区溢出检测方法属于静态方法的范畴，但若能够充分利用模型检验产生的反例，反例验证技术和模型自动精化技术则会有较低的误报率，可减少人工的参与。相对于动态检测方法而言，此方法又具有静态方法的特点，能够在系统运行之前进行检测，从而不会引入性能损失。因此基于模型检验进行缓冲区溢出检测应该是一个非常值得研究的方法。

但是本方法目前仍然处于研究的初级阶段，仅仅实现了一个简单的原型工具，整个工具中作用十分重要的反例验证器和模型自动精化技术的研究还没有深入开展，这也是后续工作的重点。

参 考 文 献

[1] Wagner D, Foster J S. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities[EB/OL]. http://www.cs.berkeley.edu/~daw/papers/overruns-ndss00.ps, 2000

[2] 张明军. 缓冲区溢出分析中的指针分析技术研究[D]. 长沙：国防科技大学，2004

[3] Holzmann G J, Smith M H. Software Model Checking: Extracting verification models from source code[J]. Software Testing Verification and Reliability, 2001, 11(6): 65-79

[4] Holzmann G J. The Model Checker Spin [J]. IEEE Trans on Software Engineering, 1997, 23(5): 279-295

[5] Larochelle D, Evans D. Statically Detecting Likely Buffer Overflow Vulnerabilities[C]// Proceedings of the USENIX Security Symposium. 2001

(上接第 30 页)

机攻击，3 个网络的全局效率随删除节点比例的变化如图 8、图 9 所示。由图 8 易知随机攻击策略对 3 个网络的影响较蓄意攻击更小，因 3 条曲线达到崩溃的速度明显减缓；但 3 条曲线的整体趋势类似，这说明当采用随机攻击策略，对 3 个网络的影响相当。由图 9 可知，在攻击初期，随机攻击对真实网络的影响略小于无标度网络和随机网络，这说明真实网络有一定的容错能力。

由以上两部分的结果分析可知，随机攻击对不同网络的影响不大，但蓄意攻击却能破坏网络的非同质结构，直接导致网络崩溃，对 BBS 用户回复网络的影响尤其明显，这说明在随机攻击下，无标度网络和随机网络有更强的稳定性。在蓄意攻击下，BBS 用户回复网络崩溃得更早，只要少数“核心节点”被移除，整个网络就会陷入瘫痪。这表明 BBS 用户回复网络面对蓄意攻击显得异常脆弱。

结束语　结合复杂网络理论，首先基于节点度和介数概念提出了 5 种攻击策略，并阐述了随机网络、无标度网络以及 BBS 用户回复网络的构建方法；其次描述了网络抗毁性的定义及其测度；最后采用不同的攻击方法对不同的网络进行攻击，对攻击结果作了详细的分析，结果表明蓄意攻击能在短时间内破坏网络结构，使网络崩溃，效果显著，其中 BBS 用户回复网络最不堪一击，抗毁能力远不及随机网络和无标度网络；但进行随机攻击时，BBS 用户回复网络体现了较好的容错能力。为了使 BBS 用户回复网络遭受蓄意攻击时受到的损害程度最低，必须对网络中的重要节点施以保护，使其体现一定的现实意义。

本文对网络的抗毁性作了初步的研究，但仍存在不足：采用的抗毁性测度即全局效率不能很好地区分 4 种蓄意攻击策略的攻击效果，这也将是下一步的研究方向，即优化抗毁性测度，使之更精确地描述每次攻击的结果。

参 考 文 献

[1] 章忠志，周水庚，方锦清. 复杂网络确定性模型研究的最新进展[J]. 复杂系统与复杂性科学，2008，5(4)：29-46

[2] 丁琳，谭敏生，肖炜. 复杂网络抗毁性研究综述 [J]. 电脑知识与技术，2009，5(1)：51-61

[3] Barabási A L, Albert R. Emergence of Scaling in Random Networks [J]. Science, 1999, 286(5439): 286-309

[4] Barabási A L, Albert R. Statistical Mechanics of Complex Networks [J]. Reviews of Modern Physics, 2002, 74(1): 47-97

[5] 章忠志. 复杂网络的演化模型研究 [D]. 大连：大连理工大学，2006

[6] 孟微. 复杂网络分析方法在情报学科研合著网络分析中得应用[D]. 北京：中国科学技术信息研究所，2008

[7] 吴渝，肖开洲，刘洪涛，唐红. BBS 虚拟社区的演化规律探索及仿真[J]. 系统工程理论与实践，2010，30(10)：1883-1890

[8] Wu Min, Li Hui, Zhang Ke, et al. An Evolutionary Model of Reply Network on Bulletin Board System [C]// International Conference of Information Technology, Computer Engineering and Management Science: ICM 2011. Nanjing, 2011

[9] Freeman L. Sociometry. 1977, 40: 35-41

[10] Holme P, Kim B J, Yoon C N, et al. Attack vulnerability of complex networks[J]. Phys. Rev. E, 2002, 65(5): 056109

[11] 丁琳，谭敏生，肖炜. 复杂网络抗毁性研究综述[J]. 电脑知识与技术，2009，5(1)：51-53

[12] Crucitti P, Latora V, Marchiori M, et al. Error and Attack Tolerance of Complex Networks[J]. Physica, 2004, 340: 388-394