一种针对 C 程序缓冲区溢出的检测方法 *

徐 超[1a,2], 何炎祥[1a,1b], 胡明昊[1a], 吴 伟[1a], 陈 勇[1a], 刘健博[1a]
(1. 武汉大学 a. 计算机学院; b. 软件工程国家重点实验室, 武汉 430072; 2. 徐州工业职业技术学院, 江苏 徐州 221000)

摘 要: 为了增强对程序缓冲区溢出漏洞的检测,提出一种利用 CCured 和 BLAST 对 C 程序进行分析的检测方法。首先利用 CCured 对 C 语言源程序进行运行时检测的代码插桩;然后用 BLAST 提供的自定义安全属性语言对这些插桩代码进行相关约束描述;最后让 BLAST 根据约束描述文件对代码插桩后的程序进行模型检测,就可以尽可能地找出 C 语言程序中潜在的缓冲区溢出漏洞。

关键词: CCured; BLAST; 模型检测; 缓冲区溢出; 安全属性
中图分类号: TP301.2 **文献标志码**: A **文章编号**: 1001-3695(2012)02-0617-04
doi:10.3969/j.issn.1001-3695.2012.02.057

Method to detect buffer overflow in C programs

XU Chao[1a,2], HE Yan-xiang[1a,1b], HU Ming-hao[1a], WU Wei[1a], CHEN Yong[1a], LIU Jian-bo[1a]
(1. *a. School of Computer, b. State Key Laboratory of Software Engineering, Wuhan University, Wuhan* 430072, *China*; 2. *Xuzhou College of Industrial Technology, Xuzhou Jiangsu* 221000, *China*)

Abstract: To enhance the buffer overflow detection for programs, this paper put forward a testing method based the analysis for C programs using CCured and BLAST. Firstly, the method used CCured to insert runtime detections into C program, and then described the constraints of the detections with the customized security-attribute language provided by BLAST. At last, according to the descriptions, BLAST could do model checking on the programs to find out the potential buffer overflow vulnerabilities in the programs as far as possible.

Key words: CCured; BLAST(Barkeley lazy abstraction software verification tool); model checking; buffer overflow; security attributes

0 引言

缓冲区溢出漏洞是软件开发中最常见的安全漏洞之一,同时也被公认为是目前最具破坏性的安全漏洞。利用缓冲区溢出攻击,可以导致程序运行失败、系统宕机、重新启动等后果,因此很多病毒都利用各种软件或系统潜在的缓冲区溢出漏洞来攻击主机。更为严重的是,攻击者可以利用它来执行非授权指令,甚至还可以取得系统特权,进而进行各种非法操作,最后还可能造成严重的经济损失。防止缓冲区溢出的一种方法就是在内存访问代码前先进行漏洞检测,可以利用模型检测来完成这个工作。

模型检测指的是这样一个过程,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(即用户预期),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。程序也可以看成是一个系统,程序潜在的缓冲区溢出漏洞也都可以理解为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[1,2]。而大多数程序分析和检测工具也正是利用模型检测技术作为其底层实现的。

但是这些检测工具都只侧重某一方面的错误检测,而且也都是站在工具设计者的角度对程序进行分析处理,这样就不可避免地会导致以下两个问题:

a)经过检测的程序只是被找出了特定类型的错误,因此不能保证其完备性。

b)检测工具没有与用户交互,所以它不一定能满足用户的检测需求。

1 CCured 和 BLAST 简介

为了解决上面的两个问题,本文借助 CCured 和 BLAST 两种工具分两步对 C 语言源程序进行分析和检测。

1.1 CCured

CCured[3,4] 可以认为是一个源到源的 C 语言翻译器,它可以保证你的 C 语言程序在运行时不会产生缓冲区溢出,即使有潜在的漏洞,它也会在对应语句执行前给出错误提示。而缓冲区溢出都是由于程序(更具体的是指针操作)对内存的不安全访问,因此 CCured 的原理就是对各种指针(包括常量指针和变量指针)进行运行时检测代码的插桩。根据指针作用的不同,CCured 将指针划分为以下几类:空指针(null pointer)、转

收稿日期: 2011-07-04; **修回日期**: 2011-08-09 **基金项目**: 国家自然科学基金可信软件重大研究计划资助项目(90818018,91018009)
作者简介: 徐超(1980-),男,博士研究生,主要研究方向为可信软件、嵌入式系统等(xuch@whu.edu.cn);何炎祥(1952-),男,教授,博导,主要研究方向为可信软件等.

换指针(pointer cast)、函数指针(function pointer)、返回指针(returning pointer)、全局引用指针(writing pointer)、序列指针(sequence pointer)、前序列指针(forward sequence pointer)和野指针(wild pointer)。其中序列指针是指能通过加减运算改变的指针变量,前序列指针是特指序列指针中通常只向前增加的指针。做这种划分的目的是为了使 CCured 根据不同的指针类型对相关代码进行相应的变换,并相应地插桩不同的运行时检测代码,以防止程序对指针的不当使用,确保程序的执行不会进入不应该进入的内存区域。

在插桩过程中,CCured 会根据不同指针插桩不同名称的运行时检测,这些检测都是以"CHECK_"为起始的字符串。比如当释放一个空指针时可能会使程序崩溃,对于空指针的检测,CCured 就会在指针释放语句之前插桩代码 CHECK_NULL(pointer)来检测指针变量 pointer 是不是空指针。

CCured 通过分析程序代码来确定需要插桩的运行时检测代码片段的个数(必须添加 CCured 的优化选项),这些代码片段被插入到源程序代码中,用于防止程序对内存的不安全访问,最终输出带有运行时检测的程序代码。经过这种源到源的转换后,对于存在不安全内存访问的程序,在运行时就会提前给出错误提示,而不至于在非法访问之后出现实际的缓冲区溢出错误;也就是说 CCured 的输出是能保证内存访问安全的 C 语言代码。所有源程序经过 CCured 的分析和处理后都会被转换成内存安全的 C 语言程序代码。翻译流程及其输出如图 1 所示,其中双线框所表示的就是转换后带有运行时检测的 C 语言程序代码。

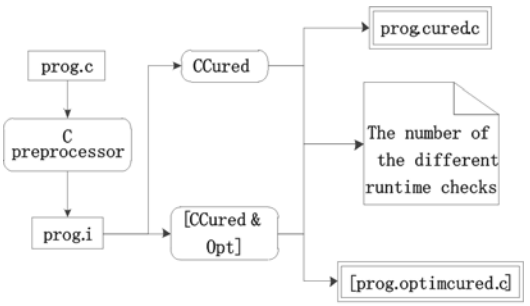

图1　CCured转换过程

但运行时检测会消耗程序额外的运行时间,而且程序错误也只能在程序运行时才能暴露,显然这不是人们所期望的。由于 CCured 的保守设计,它只是在内存访问代码前插桩运行时检测指令,即使是加入 CCured 的优化选项(可以删除重复插桩的检测),也不能精确地证明所有的内存访问代码是否安全,从而使得程序中可能插桩了一些不必要的运行时检测。为了减少额外消耗的时间,对于这样的运行时检测,就希望它不被执行,那么最好的方法就是删除它,这就需要下面介绍的 BLAST 工具。

1.2　BLAST

BLAST[5] 是由加州大学 Berkeley 分校的 Henzinger 等人开发的,是基于路径敏感的 C 语言程序模型检测工具[6,7]。

BLAST 可以进行代码的可达性分析,也就是判断程序是否可以从入口处开始执行,并到达某个指定的位置(即程序中的 error 标签语句);也可以由用户输入一个 C 语言源程序(prog.c)及自定义的安全属性约束文件(prog.spc),然后 BLAST 可以根据该描述文件对程序进行静态分析,在与安全属性约束文件所匹配的对应位置添加判断,并在违背约束的分支下添加 error 标签语句;最后进行可达性分析。若源程序满足所有的属性约束,即程序不会执行到任何添加 error 标签的语句,则返回 safe;只要源程序不满足一条属性约束,则返回一个程序执行的反例路径,亦即程序的一个错误状态。

但是 BLAST 自身提供的安全属性描述语言对于 C 语言程序属性的表达还不够强大,比如数组和指针用全属性约束文件来定义不容易,甚至有些复杂语句是无法表示的。利用 CCured 则可以将 C 程序的安全属性用可被 BLAST 表示的插桩代码来表示,这样,利用 BLAST 不仅可以在经过 CCured 处理后的程序中找出部分不必要的无效插桩,还能够发现程序中的错误,既可以降低运行的时间消耗,又能使得部分程序错误在开发时就暴露出来,而不必等到程序运行时暴露,做到防范于未然。

在安全属性约束文件(在本文中安全属性约束指的是 CCured 在 C 语言源程序中插桩的运行时检测代码片段)中,可以定义全局变量、结构、(事件)匹配模式及处理。形式化定义[8] 如下(其中粗体表示类 C 语言语法):

```
Observer: DeclSeq
DeclSeq: Declaration
| DeclSeq Declaration
Declaration: "#include" CFile
| "global" CVarDef
| "shadow" CTypeName '{' CFieldSeq '}'
| "initial" '{' CExpression '}'
| "final" '{' CExpression '}'
| "event" '{'
Temporal "pattern" '{' Pattern '}'
[Assertion]
[Action] '}'
Temporal: "before" | "after" | empty
Pattern: ParamCStmt | ParamCStmt "at" LocDesc
Assertion: "assert" '{' CExpression '}'
| "guard" '{' CExpression '}'
| empty
Action: "action" '{' CStatementSeq '}' | empty
LocDesc: CLable | CFilename_CNumberofLine
```

下面给出一个查询描述文件的具体例子:

```
#include 〈lock.h〉
global int locked = 0;
event {
pattern { $? = unlock( $ 1); }
guard { locked == 1 }
action{ locked = 0; }
}
```

上面定义了一个全局变量 locked 用来辅助程序的分析,pattern{ $? =unlock($ 1); }表示匹配程序中的函数调用语句"$? =unlock($ 1);",其中 $ 1 和 $? 分别匹配函数 unlock 的参数和返回值,guard{locked == 1}表示正确匹配时程序应满足的状态,即安全属性约束,action{locked = 0;}则表示程序在满足安全属性约束后程序可以额外执行的指令。通过这种方式,BLAST 就可以分析程序在某一匹配模式下是否满足对应的安全属性约束。

利用上面所述原理,就可以自定义安全属性约束文件,然后将它和 C 语言源程序一起输入到 BLAST 中,BLAST 就会自动完成属性约束的验证。其执行流程如图 2 所示。

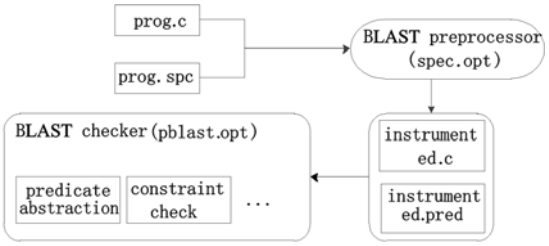

图2　Blast的执行流程

在 pblast. opt 的执行过程中,BLAST 是对实际的程序进行模型抽象,在抽象的基础上作模型检测。这个方法的优点是明显的,它可以大幅度地减小状态空间。但正是由于这种抽象,导致检测过程不是在原来的数据域上执行,而是基于谓词描述(BLAST 利用的是 Craig 谓词插值和描述)的符号化执行[9],从而会使模型丢失原型的一些细节,在进行模型检测时发现的反例路径就可能是假反例。所以 BLAST 在找出反例路径后会利用后向搜索技术在实际程序中找到对应的执行路径,如果找得到,那么就找到了一个程序的错误状态;否则 BLAST 就会根据发现的反例路径细化抽象模型,然后再进行循环检测。这种检测方法通常也称为 CEGAR(反例引导的抽象细化)引擎,其处理流程如图 3 所示。另外,微软的 SLAM 和卡内基梅隆大学的 MAGIC 等也都是用 CEGAR 引擎对程序进行模型检测,其区别在于抽象和细化的实现细节上。

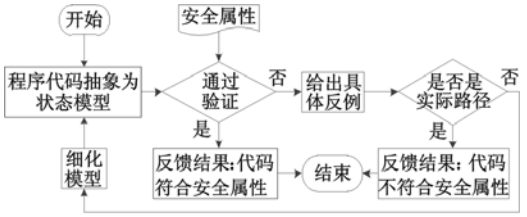

图3　基于CEGAR引擎的程序代码模型检测流程

2　缓冲区溢出检测

通常所说的内存安全访问是指程序只能在相关对象所申请的空间内(即所允许访问的局部空间)执行,而不能越界访问,它是软件正确性保证的基本要求。本章以“数组限界”这一特定属性约束为例来介绍如何利用 CCured 和 BLAST 进行程序检测。

对于程序中可能出现越界的数组操作,CCured 会插入一个运行时检测语句 CHECK_GEU(num, para)。其中:参数 num 表示参数 para 所能变化的范围。当程序运行时,就会先调用 CHECK_GEU(num, para)。如果 para 超出了 num 所限定的范围,CHECK_GEU(bum, para)就会终止程序,防止后续不安全的内存访问代码的执行。但是正如引言所说,CCured 插桩的代码并不都是有效的,对于那些经过程序各条执行路径都不会发生数组越界的操作,也就不希望执行这个运行时检测了,而应该把它删除以节省程序运行的时空代价。因此利用 BLAST 对以插入运行时检测的程序进行进一步的检测分析,找出那些不必要的运行时检测,同时对于那些能够被证明是不安全的内存访问代码,则提前了错误的暴露时间。

首先,利用 BLAST 的安全属性约束对 CHECK_GEU(num, para)调用添加了匹配模式,BLAST 就会在 CHECK_GEU(num, para)函数调用前添加一条条件语句:

```
if (num≥para)
{
}
else
{
  __error_();
}
```

__error_()是 BLAST 的自定义函数,定义如下:

```
void __error_(void)
{
  __BLAST_error = 0;
  error: goto error;
}
```

这样 BLAST 就在程序不安全的执行分支(num < para)下加入了 error 标签,然后 BLAST 再自动检测 error 标签在程序执行的所有路径中是否存在实际可达路径。由于 BLAST 是使用基于路径流敏感的模型检测方法,所以这是可以做到的。若存在,就置标记变量__BLAST_error = 0,并报错;否则继续检测直至结束。这样就达到了检测该程序片段是不是安全的内存访问代码和对应的运行时检测是否必要的目的。

正如前面所说,BLAST 的检测会有三种可能的输出,也就对应三种不同的处理方式:

a)如果经过检测,error 标签是不可达的,则会返回 safe。这说明 CCured 在对应语句前插桩的运行时检测一定不会失败,即该检测是无效的,可以删除。

b)如果经过检测,error 标签是实际可达的,则会返回达到 error 标签的程序执行路径(也称为反例路径)。这就说明对应语句是不安全的内存访问,是一个错误,就可以根据这条反例路径检查程序的错误并修改,然后再进行 BLAST 检测。

c)如果由于某些原因,使得 BLAST 不能证明对应的内存访问代码是不是安全的,那就只能选择保留 CCured 插桩的运行时检测,让它进入程序执行阶段进行检测保护。

3　实验结果及分析

为了观察本文方法的效果,本文选择了表 1 所示的 ACM 常用参考算法程序进行了测试。这些程序中包含了大量的访存指令,以使实验结果比较明显。本文的所有测试均在 2.60 GHz Pentium Dual-Core CPU 与 2 GB 内存的环境和 GNU/Linux 系统下完成。

表 1　运行时检测分析

测试程序	代码行数			插桩的运行时检测			无效插桩	错误数
	源程序	CCured 未优化	CCured 优化	CCured 未优化	CCured 优化	BLAST		
1)大整数加法	127	509	488	34	23	15	8	15
2)单链表处理	399	1 627	1 570	98	66	53	13	49
3)二叉树实例	237	806	777	41	26	12	14	12
4)各种排序法	382	1 719	1 654	150	120	40	80	37
5)哈夫曼算法	413	3 415	3 375	339	277	146	131	75
6)货郎担算法	407	1 571	1 515	87	59	47	12	39
7)迷宫问题	117	545	529	29	21	16	5	16
8)逆矩阵	98	797	748	80	57	51	6	37
9)神经元模型	150	1 227	1 213	39	31	24	7	13
10)图处理	561	3 514	3 340	333	230	104	126	71

对每个测试用例,首先使用 CCured 不带优化进行插桩;然后 CCured 带优化进行插值;最后再用 BLAST 对经 CCured 带优化插值的程序进行自定义属性模型检测。

代码行数下的三列依次表示源程序代码行数、经过 CCured(不加优化选项)插桩代码后的代码行数、经过 CCured(加入优化选项)插桩代码后的代码行数;插桩的运行时检测下的三列依次表示 CCured(不加优化选项)在程序中插桩的运行时检测个数、CCured(加入优化选项)在程序中插桩的运行时检测个数、经过 CCured(加入优化选项)插桩后的程序在用 BLAST 进行一遍检测后剩下的运行时检测个数;CCured(加入优化选项)在程序中插桩的运行时检测个数与 BLAST 检测后剩下的运行时检测个数的差就是经 BLAST 证明的无效插桩的个数;程序中存在的错误表示的是进行 BLAST 第一次检测时直接暴露的程序错误(或者说是反例路径数),它与 BLAST 检测后剩下的运行时检测个数的差就是 BLAST 检测时出现超时等情况而无法对匹配对象给出明确证明的失败数量。

将表 1 中最后四列数据用折线图来表示,得到对应的图示,如图 4 所示。

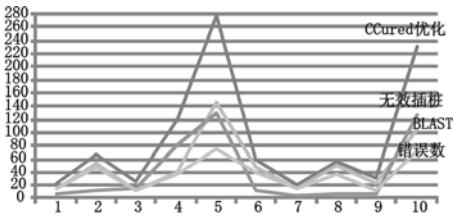

图4　BLAST检测前后数据对比

通过对表 1 不同列数据的分析以及折线图的表示,可以知道经过 BLAST 的处理后,不仅能够在程序静态检测阶段就能发现程序的部分潜在错误,而且基本上随着 CCured 优化程序后插桩代码片段的增加,BLAST 检测到的不安全插桩片段也在增加,这表示 BLAST 的处理确实减少了 CCured 的插桩片段的数量,节省了程序运行时检测消耗的额外时间。

4　结束语

本文首先通过 CCured 把 C 语言源程序转换成插桩了运行时检测的程序,然后再利用 BLAST 在程序中添加运行时检测的模式匹配以插入 error 标签,从而将缓冲区溢出漏洞的检测问题转换成为 error 标签的可达性判断问题,解决了 BLAST 不能直接用于缓冲区溢出漏洞检测的问题。与传统模型检测工具相比,本文利用 CCured 与 BLAST 结合的方法不仅能够对 C 语言程序进行安全性检测,找出程序的潜在错误,保证程序的内存访问安全,同时又能够减少运行时进行额外检测而消耗的时间。

参考文献:

[1] PAN Wen, JIANG Zhan-jun, DU Zheng-feng, *et al*. Analysis of a reduced-ML algorithm in BLAST[J]. Science in China Series F: Information Sciences,2008,51(12):2094-2100.

[2] BEYER D, HENZINGER T, THEODULOZ G. Lazy shape analysis [C]//Proc of Computer Aided Verification. Berlin: Springer,2006: 532-546.

[3] CCured manual[EB/OL]. http://hal.cs.berkeley.edu/ccured.

[4] NECULA G C, McPEAK S, WEIMER W. CCured:type-safe retrofitting of legacy code[C]//Proc of ACM SIGPLAN-SIGACT Conference on the Principles of Programming Languages. Portland: ACM Press, 2002:128-139.

[5] BLAST manual[EB/OL]. http://www.sosy-lab.org/dbeyer/blast_doc.

[6] ŠERY O. Enhanced property specification and verification in BLAST [C]//Proc of Fundamental Approaches to Software Engineering. [S.l.]: IEEE Computer Science Press,2009:456-469.

[7] BEYER D, HENZINGER T, THEODULOZ G. Configurable software verification: concretizing the convergence of model checking and program analysis[C]//Proc of Computer Aided Verification. [S.l.]: IEEE Computer Science Press, 2007:504-518.

[8] BEYER D, JHALA R, HENZINGER T, *et al*. The BLAST query language for software verification[C]//Proc of SAS. Heidelberg: Springer,2004:2-18.

[9] BEYER D, HENZINGER T, THEODULOZ G. Program analysis with dynamic precision adjustment[C]//Proc of Automated Software Engineering. [S.l.]: IEEE Computer Society Press, 2008:29-38.

[10] 何恺铎,顾明,宋晓宇,等.面向源代码的软件模型检测及其实现[J].计算机科学,2009,36(1):267-272.

[11] 王雷,李吉,李博洋.缓冲区溢出漏洞精确检测方法研究[J].电子学报,2008,36(11):2200-2204.

(上接第 616 页)

参考文献:

[1] MYERS D. On the use of NAND flash memory in high-performance relational databases[D]. Massachusetts: MIT, 2008.

[2] SHAH M A, HARIZOPOULOS S, WIENER J L, *et al*. Fast scans and joins using flash drives[C]//Proc of the 4th Workshop on Data Management on New Hardware. New York: ACM Press, 2008: 17-24.

[3] LI Yu, ON S T, XU Jian-liang, *et al*. DigestJoin: exploiting fast random reads for flash-based joins[C]//Proc of the 10th International Conference on Mobile Data Management. Washington DC: IEEE Computer Society,2009: 152-161.

[4] 梁智超,周大,孟小峰. Sub-Join:面向闪存数据库的查询优化算法[J].计算机科学与探索,2010,4(5):401-409.

[5] MERRETT T H, KAMBAYASHI Y, YASUURA H. Scheduling of page fetches in join operations[C]//Proc of the 7th International Conference on Very Large Data Bases. [S.l.]: IEEE Computer Society,1981: 488-498.

[6] LI Zhe, ROSS K A. Fast joins using join indices[J]. The VLDB Journal,1999,8(1):1-24.

[7] MISHRA P, EICH M H. Join processing in relational databases[J]. ACM Computing Surveys,1992,24(1):63-113.

[8] CHENG J M, HADERLE D J, HEDGES R, *et al*. An efficient hybrid join algorithm: a DB2 prototype[C]//Proc of the 7th International Conference on Data Engineering. Washington DC: IEEE Computer Society,1991:171-180.

[9] CHAN C Y, OOI B C. Efficient scheduling of page access in index-based join processing[J]. IEEE Trans on Knowledge and Data Enginnering,1997,9(6):1005-1011.

[10] 孙文隽,李建中.排序合并 Join 算法的新结果[J].软件学报,1999,19(3):264-269.