# 基于反编译的循环脆弱点检测

李永伟, 尹 青, 舒 辉, 李继中
(解放军信息工程大学, 郑州 450002)

**摘 要**: 循环拷贝出错是缓冲区溢出漏洞产生的主要原因之一。为了提高此类漏洞的检测效率,提出一种基于反编译的循环脆弱点检测方法。该方法首先对目标文件进行反编译,在反编译的基础上构建函数的 AST(抽象语法树),设计算法提取函数内部的循环信息;然后根据循环脆弱点存在的特性,构建有限状态自动机,对循环脆弱点进行检测。该方法在无源码漏洞检测方面有明显优势,能有效发掘软件中存在的循环脆弱点,提高漏洞挖掘的效率和自动化程度。

**关键词**: 反编译; 漏洞; 缓冲区溢出; 循环检测; 自动机
**中图分类号**: TP393.08 **文献标志码**: A **文章编号**: 1001-3695(2013)05-1508-03
doi:10.3969/j.issn.1001-3695.2013.05.058

## Loop vulnerability detection based on decompile

LI Yong-wei, YIN Qing, SHU Hui, LI Ji-zhong
(*PLA Information Engineering University*, *Zhengzhou* 450002, *China*)

**Abstract**: Improper circulatory buffer copy is one of the main reasons of buffer overflow, in order to improve the efficiency of such vulnerability detection, this paper presented a method which based on decompile to detect loop vulnerability. Firstly, the method decompiled target binary files and constructed the abstract syntax tree (AST) of functions, designed algorithm to extract circulation information of functions. Then, according to the characteristics of loop vulnerability, it built finite state automata to detect the loop vulnerability. This method had obvious advantages in non-source code vulnerability detection, could discover loop vulnerability in the software effectively, and improve the efficiency and automation of vulnerability mining.
**Key words**: decompile; vulnerability; buffer overflow; loop detection; automata

随着信息技术的发展,计算机在经济、军事、能源等各个方面的应用越来越广泛,其安全性变得尤为重要,应用软件安全漏洞的存在使信息安全面临严重威胁。根据国家互联网应急中心发布的报告,2011 年,CNVD 共收集整理并公开发布信息安全漏洞 5 547 个,较 2010 年增加 60.9%。在各类漏洞中,涉及应用软件的最多,占 62.6%[1]。漏洞是应用软件产生安全问题的一个主要原因,在众多应用软件漏洞中,缓冲区溢出[2]漏洞是最常见的安全漏洞之一,同时也是公认的最具破坏性的安全漏洞。因此,如何快速有效地检测出应用软件中存在的缓冲区溢出漏洞,对提高软件安全性有着重要的意义。

通过对大量漏洞样本进行分析发现,缓冲区溢出漏洞成因大致可以分为以下两类:不安全函数调用和循环拷贝出错。针对循环拷贝出错引发的漏洞,本文提出了基于反编译的循环脆弱点检测方法,利用 Hex-Rays 提供的 SDK 设计并实现了循环脆弱点检测插件。该插件能够提取应用软件各模块内函数的循环类型、结束条件等信息,通过对这类信息进行分析,以判断函数的安全性。

## 1 相关研究

现有的漏洞挖掘技术大致可以分为面向源码的挖掘技术、面向二进制代码的挖掘以及代码无关的黑盒测试技术[3]。面向二进制代码的挖掘技术又可分为逆向分析技术[4]和 Fuzz 测试技术[5]。主要采用的技术有词/语法分析、控制流分析、数据流分析、抽象解释、符号执行、模式匹配等。

在基于循环的缓冲区溢出检测方面,文献[6]针对缓冲区溢出漏洞,提出静态分析与动态测试相结合的检测方法,在反汇编层面上对循环造成的缓冲区溢出进行了讨论;Rawat 等人[7]提出 BOIL(buffer overflow inducing loops)的概念,并在反汇编结果的基础上实现轻量级的静态分析用于检测 BOIL,其主要采用模式匹配的思想;胡定文等人[8,9]针对源码构建缓冲区溢出漏洞检测模型,并采用自动机对循环拷贝错误进行检测;此外,文献[10]介绍了基于支配树的循环识别算法,在此基础上,Flake[11]提出面向二进制代码的函数体内循环检测方法,并判别循环内部内存读写操作的安全性;Silberman[12]基于二进制代码的反汇编结果,以函数为基本粒度提取循环结构。

现有的漏洞挖掘技术都取得了一定效果,但也存在一些不足。例如,对于面向源码的漏洞挖掘技术来说,大部分软件厂商不会将源码对外公开,同时其不能发现编译、链接阶段引入的漏洞。对于面向二进制的漏洞挖掘,程序信息过于匮乏,缺少数据类型、函数参数、函数结构等信息,给漏洞挖掘带来很大的挑战。对于黑盒测试,数据构造难度较大,并且很难覆盖程

**收稿日期**: 2012-09-12; **修回日期**: 2012-10-22
**作者简介**:李永伟(1988-),男,河南周口人,硕士研究生,主要研究方向为软件逆向工程(huacm6171@hotmail.com);尹青(1968-),女,副教授,硕导,主要研究方向为计算机应用技术;舒辉(1974-),男,副教授,硕导,主要研究方向为软件逆向工程、信息安全;李继中(1983-),男,博士研究生,主要研究方向为计算机应用技术、软件逆向工程.

序的所有执行路径。

## 2　基于反编译的循环脆弱点检测方法

反编译技术的出现,为软件漏洞挖掘提供了新的思路,以 Hex-Rays、IDC 等为代表的反编译工具逐渐发展成熟,这些工具可以将目标二进制代码或汇编代码反编译为类 C 语言形式,恢复其函数结构、参数信息以及部分数据类型信息,因此在反编译基础上进行漏洞挖掘有明显优势。

### 2.1　Hex-Rays SDK 简介

Hex-Rays Decompiler 是 IDA Pro 下的一款功能强大的插件,它可以将应用软件的二进制代码反编译为类 C 语言的代码[13]。与反汇编相比,反编译的结果更加简洁,可读性更强,易于理解,为分析人员节省了大量分析时间。目前,Hex-Rays 支持 x86 和 ARM 编译生成的 32 位代码,并提供有 Hex-Rays Decompiler SDK,允许开发者实现自己的分析方法,比如漏洞挖掘、软件确认、覆盖分析等。

### 2.2　系统框架

为了对循环拷贝出错引发的漏洞进行检测,提出了基于反编译的循环脆弱点检测方法,该方法系统框架如图 1 所示。首先将目标文件输入 IDA Pro 中,对其进行反编译生成对应的类 C 代码。基于生成的 C 代码构建函数的 AST。循环信息提取检测 AST 中节点类型,提取循环信息。循环脆弱点检测模块根据构建的自动机来判断循环的安全性,最后生成分析结果报表。


**图1　基于反编译的循环脆弱点检测系统框架**

### 2.3　AST 生成

AST 生成模块是循环脆弱点检测的基础,Hex-Rays 反编译插件将函数的反编译结果存储在 ctree 结构中。Ctree 是一种类似于树的数据结构,顶层有 cfunc_t 类,该类对函数进行描述,提供对属性的访问,包括类型、局部变量、函数体等。但其无法满足后续开发需求,因此在此基础上构建了自己的 AST 结构。该 AST 借助于德国 Kasper Peeters 实现的 tree 进行构建,提供了 *n*-叉树的类 STL 容器,并将存储于节点内的数据实现了模块化;另外,还提供了多种迭代器类型(前序、后序等),其访问方法尽可能与 STL 保持一致,并有多种算法供选择。AST 节点类型使用 Hex-Rays Decompiler SDK 中定义的节点类型。在 ctree 构建过程中,遍历其每个节点,将其添加到自定义的 AST 结构中。具体生成算法描述如下:

```
Algorithm Generate AST 生成 AST
input: ea // ea 表示函数起始地址
output: T// T 表示地址 ea 处函数的 AST
begin
  T := ∅
  pfunc := get_func(ea)//获取地址 ea 处函数 pfunc
  hf := NULL //hf 用于存储反编译出错信息
  pcfunc := decompile(pfunc, hf)
  //对 pfunc 进行反编译,结果存储在 pcfunc 中
  for all node in pcfunc→body do
  //深度优先遍历 pcfunc→body 内每个节点 node
   if exist_node(T,node) then
   //判断节点 node 是否存在于 T 中
   return T
   end if
   if exist_parent( node ) then
   //判断 node 有无父节点
   insert_root(T, node) //插入根节点
   else
   current_node := get_parent (pcfunc→body,node)
   //获取 pcfunc→body 中 node 的父节点
   loc := get_loc( T,current_node )
   //获取当前节点在 T 中的位置 loc
   append_child(T,current_node,node )
   //将 node 作为 current_node 的儿子插入 T 中
   end if
  end for
end
```

使用本算法生成的语法树结构如图 2 所示。函数 func 的反编译结果如下:

```
int_cdecl func( int a)
{
  int result;[sp+0h][bp-4h]@2
  if(a==1)
    result=5;
  else
  result=6;
  return result;
}
```

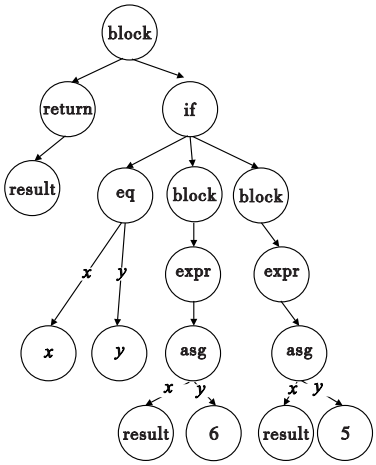


**图2　函数func的语法树**

图 2 中每个圆表示一个节点,节点内部包含节点类型、地址等信息。

### 2.4　循环信息提取

现有的循环检测大多是针对源码,主要用于编译优化。面向二进制代码的循环检测也有比较成熟的方法,大体思路是首先对二进制代码进行反汇编,然后在反汇编的基础上进行循环检测。这类方法能很容易地检测出简单的循环,但在对复杂循环的检测上容易出错。相比之下,在反编译结果的基础上进行循环信息提取,具有明显的优势。当前提取的循环结构主要包括 for、while 和 do-while。使用深度优先算法对语法树的各个节点进行遍历,判断其节点类型,若属于上述循环类型,提取其相关信息,供后续检测使用。

### 2.5　循环脆弱点检测模型

经过前面两步,已经得到函数内所包含的循环信息,但此时提取出的循环结构大多是安全的,需要对其作进一步过滤,否则将存在大量的误报。

#### 2.5.1　检测循环内部缓冲区写操作

缓冲区是一段连续的内存空间,它可能是数组,也可能以指针的形式出现。所谓写操作也就是赋值操作,对其进行检

测,就是判断循环体内是否存在赋值操作。假设 $A_1$ 表示存在指针或数组操作,$A_2$ 表示存在赋值语句,$A_3$ 表示赋值语句左操作数是变量,$A_4$ 表示赋值语句左操作数是指针与常量的和,$A_5$ 表示赋值语句左操作数为数组某一固定元素,$A_6$ 表示存在递增或递减操作。根据以上假设给出式(1)。

$$A_1 \wedge A_2 \wedge (\neg (A_3 \vee A_4 \vee A_5)) \wedge A_6 \tag{1}$$

对于每个循环结构,提取 $A_1 \sim A_6$ 的一组真值,为式(1)的一组赋值。判断循环体内是否有缓冲区写操作,也就是判断式(1)在每组赋值下的真假。

### 2.5.2 获取循环结束条件

循环结束条件受外部输入或中间结果影响是造成循环拷贝出错的原因之一,通过大量分析可以发现,若循环结束条件为常量,则几乎不可能出错,若循环结束条件受外部输入影响,则出错的可能性比较大。循环结束条件主要包括两类:循环语句中的结束条件和循环体内的 break 语句。对于循环语句中的结束条件,可对语法树进行遍历,找到循环类型的节点,获取该节点的 expr 属性即可。对于循环体内的结束条件,判断循环节点的直接孩子节点类型,若为 break 语句,获取其条件即可。

### 2.5.3 建立有限状态自动机

有限状态自动机是一个五元组:

$$M = (Q, \Sigma, \delta, q_0, F)$$

其中:$Q = \{q_0, q_1, q_2, q_3, q_4\}$,表示有限状态集;$\Sigma$ 表示字符表,包括循环和循环结束条件等;$\delta$ 为状态转移函数;$q_0 \in Q$ 为初始状态;$F = \{q_3, q_4\} \subset Q$ 为终结状态。其状态转移图如图 3 所示。

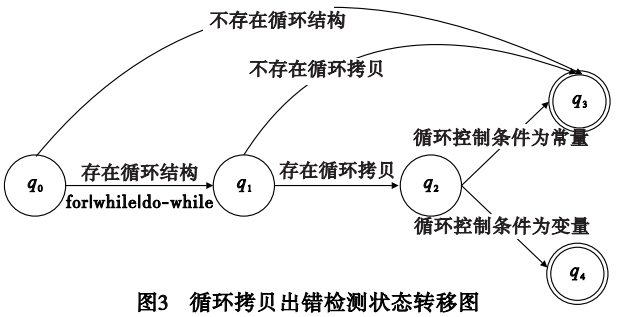


**图3 循环拷贝出错检测状态转移图**

其中:$q_0$ 为初始状态;$q_3$、$q_4$ 为终止状态,$q_3$ 表示正常状态,不存在溢出可能,$q_4$ 表示可能存在溢出;$q_1$、$q_2$ 为中间状态,$q_1$ 表示存在循环,$q_2$ 表示存在缓冲区拷贝操作。

检测对象为反编译得到的函数代码,检测初始状态为 $q_0$;若存在循环结构,转状态 $q_1$,否则,转状态 $q_3$;检测是否存在循环拷贝,若有,转状态 $q_2$,否则,转状态 $q_3$;提取循环结构中的结束条件,判断结束条件是否为常量,若是,转状态 $q_3$,若不是,转状态 $q_4$。至此,检测结束,生成结果报表。

## 3 结果分析

本文设计并实现了一个用于循环脆弱点检测的 IDA Pro 插件。为了验证本插件的运行效果,本文选用了几款常用的软件进行测试,分别统计使用本文方法前后所提取的循环个数。表 1 给出了测试结果。测试使用的硬件环境为 3.00 GHz Pentium Dual-Core CPU、6 GB 内存、Windows 7 SP1。

从表中可以看出,使用本文提出的方法后,提取的循环数量大大减少。对于漏洞挖掘人员来说,减少了代码分析范围,使得分析目标更加明确,提高了漏洞挖掘的效率。

表 1 插件运行结果

| 软件名 | 版本 | 函数个数 | 使用前 | 使用后 |
|---|---|---|---|---|
| EXCEL. exe | 11.0.8342.0 | 29 205 | 14 414 | 2 231 |
| chrome. exe | 21.0.1180.79 | 3 683 | 1 849 | 403 |
| calc. exe | 6.1.7601.17514 | 1 887 | 448 | 92 |
| WinRAR. exe | 3.90.0.0 | 2 804 | 2 462 | 593 |
| Feiq. exe | 2.4.0.0 | 13 531 | 2 410 | 495 |
| libpng. dll | 1.4.9.0 | 349 | 420 | 123 |

以下为使用 Hex-Rays 对函数 func 反编译生成的类 C 代码,该函数的功能为将字符串 str 中的内容拷贝到字符数组 name 中,在字符拷贝的过程中可能存在缓冲区溢出。

```
int __cdecl func(const char *str)
{
  unsigned int i; // [sp+10h] [bp-10h]@1
  char name[10]; // [sp+14h] [bp-Ch]@3
  for ( i = 0; i < strlen(str); ++i )
    name[i] = str[i];
  return 0;
}
```

使用本插件对其进行检测,检测结果如图 4 所示。

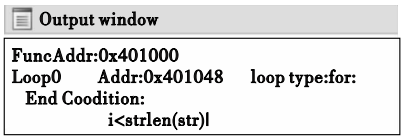


**图4 函数func检测结果**

检测开始时状态为 $q_0$,当检测到 for 循环时,状态转 $q_1$;继续检测到循环体内的赋值语句 name[$i$] = str[$i$],其中索引 $i$ 的值循环递增,因此循环体内存在循环拷贝操作,状态转 $q_2$;获取循环结束条件 $i <$ strlen(str),strlen(str)非常量,其值与传入参数 str 有关,状态转 $q_4$,说明 for 循环可能引起缓冲区溢出漏洞。进一步分析,若传入函数 func 的字符串 str 的长度不大于 10,则局部缓冲区 name 足以容纳字符串 str,不会发生溢出;若字符串 str 长度大于 10,则在将字符向 name 写入的过程中,会超出 name 边界进行写操作,从而造成溢出。图 4 检测结果显示,插件能有效地检测出潜在的缓冲区溢出漏洞,并能给出漏洞的具体信息。

## 4 结束语

本文介绍了基于反编译的循环脆弱点检测方法,并开发出了相应的插件。该插件以目标二进制文件为输入,借助 Hex-Rays 反编译插件生成类 C 代码,在此基础上作漏洞检测。相比针对反汇编结果的检测方法,该方法有明显优势。使用该插件可以大大提高漏洞挖掘人员对循环拷贝出错类漏洞的挖掘效率。本方法虽然有一定效果,但也有不少需要改进的地方。目前,笔者所做的检测主要是针对过程内部,不关心过程间数据传递及控制流转移,以后需要对这些方面作进一步研究。此外,插件检测存在误报,需要对漏洞模型进一步细化,以降低误报率。

**参考文献:**

[1] CNCERT. 2011 年我国互联网网络安全态势综述[EB/OL]. (2012-03-19)[2012-08-01]. http://www.cert.org.cn/UserFiles/File/201203192011annualreport(1).pdf.

[2] Aleph One. Smashing the stack for fun and profit[EB/OL]. (1996-08-11)[2012-08-01]. http://www.phrack.com/issues.html?issue=49&id=14.

(下转第 1514 页)

表 6 系统所产生的密钥流的游程检测结果(1 bit)

| LR | min | max | mean | 置信区间 $\alpha = 0.01$ |
|---|---|---|---|---|
| 1 | 2371 | 2629 | 2501 | [2496 2506] |
| 2 | 1141 | 1352 | 1248 | [1245 1251] |
| 3 | 557 | 696 | 625 | [622 627] |
| 4 | 256 | 362 | 313 | [311 315] |
| 5 | 123 | 194 | 156 | [155 157] |
| 6 + | 111 | 200 | 157 | [156 158] |

表 7 系统所产生的密钥流的游程检测结果(0 bit)

| LR | min | max | mean | 置信区间 $\alpha = 0.01$ |
|---|---|---|---|---|
| 1 | 2333 | 2642 | 2500 | [2495 2505] |
| 2 | 1165 | 1344 | 1250 | [1246 1253] |
| 3 | 551 | 702 | 625 | [623 628] |
| 4 | 259 | 363 | 313 | [311 315] |
| 5 | 119 | 202 | 156 | [155 157] |
| 6 + | 122 | 191 | 156 | [155 157] |

结果表明新系统所产生的二进制序列都通过了 FIPS 140-2 测试。此外,定义检验的显著水平 $\alpha = 0.01$,经检验 CPNG,产生的二进制序列满足正态分布,因此可利用下面的公式求得置信区间。

$$[\bar{x} - t_{n-1,1-\alpha/2} S/\sqrt{n}, \bar{x} + t_{n-1,1-\alpha/2} S/\sqrt{n}] \tag{13}$$

其中:$t_{n-1,1-\alpha/2}$ 是学生分布,$\bar{x}$ 为二进制序列的均值,$S$ 是对应的标准差,$\alpha = 0.01$。由于游程中要同时对"0"和"1"进行统计分析,因此 $n = 200$。置信区间分析表明,伪随机序列具有良好的随机性。

## 4 结束语

本文提出了一类能达到严格同步的传递函数 $H$。基于离散广义混沌同步定理和 3D-Lorenz 系统构造了一个新的 6 维广义同步混沌系统。通过引入变换 $T$ 设计了一个产生二进制序列的 CPNG。计算机仿真表明该序列通过了 FIPS 140-2 标准的所有检测。置信区间分析、不同密钥流的相关性和不同比特百分比的数值分析也说明该 CPNG 可应用于信息安全领域。

**参考文献:**


[1] 方锦清. 驾驭混沌与发展高新技术[M]. 北京:原子能出版社, 2002:31-32.

[2] LI T Y, YORKE J A. Period three implies chaos[J]. American Mathematical Monthly, 1975, 82(10):481-485.

[3] DVORÁKOVÁ J. Chaos in non-autonomous discrete dynamical systems[J]. Communications in Nonlinear Science and Numerical Simulation, 2012, 17(12):4649-5704.

[4] KARIMI A, PAUL M R. Extensive chaos in the Lorenz-96 model [J]. Chaos, 2010, 20(4):043105.

[5] CHEN Guan-rong, MAO Yao-bin, CHUI C K. A symmetric image encryption scheme based on 3D chaotic cat maps[J]. Chaos, Solitons and Fractals, 2004, 21(3):749-761.

[6] NI Xuan, LAI Ying-cheng. Transient chaos in optical metamaterials [J]. Chaos, 2011, 21(3):1054-1500.

[7] DENG Xiao-ping, ZHAO Dao-mu. Color component 3D Arnold transform for polychromatic pattern recognition[J]. Optics Communications, 2011, 284(24):5623-5629.

[8] WANG Xing-yuan, WANG Ya-qin. Adaptive generalized synchronization of hyper-chaos system[J]. International Journal of Modern Physics B, 2011, 25(32):4563-4571.

[9] YANG Tao, CHUA L O. Generalized synchronization of chaos via linear transformations[J]. International Journal of Bifurcation Chaos in Applied Sciences and Engineering, 1999, 9(1):215-219.

[10] 臧鸿雁,闵乐泉,吴春雪,等. 基于离散混沌系统广义同步定理的数字图像加密方案[J]. 北京科技大学学报, 2007, 29(1):97-101.

[11] XIAO Wen-xian, FU Jun-hui, LIU Zhen, *et al*. Generalized synchronization of typical fractional order chaos system[J]. Journal of Computers, 2012, 7(6):1519-1526.

[12] LI Pei, MIN Le-quan, ZANG Hong-yan, *et al*. A generalized chaos synchronization-based pseudo-random generator number and performance analysis[C]//Proc of International Conference on Communications, Circuits and Systems. [S. l. ]:IEEE Press, 2010:781-785.

[13] FERNADEZ B. Discontinuous generalized synchronization of chaos [J]. Dynamical Systems, 2012, 27(1):105-116.

[14] SPROTT J C. Chaos and time-series analysis[M]. Oxford: Oxford University Press, 2003:513.

[15] GOLOMB S W. Shift register sequence [M]. CA: Aegean Park Press, 1982:24-59.


(上接第 1510 页)


[3] BALAKRISHNAN G. WYSINWYX: What you see is not what you exe-cute[D]. Wisconsin: University of Wisconsin, 2007.

[4] 张晓锋. 软件逆向工程相关技术研究与实现[D]. 成都:电子科技大学, 2007.

[5] TAKANEN J D A, MILLER C. Fuzzing for software security testing and quality assurance [M]. London: Artech House, 2008.

[6] YUAN Jing-bo, DING Shun-li. A method for detecting buffer overflow vulnerabilities[C]//Proc of the 3rd IEEE International Conference on Communication Software and Networks. 2011:188-192.

[7] RAWAT S, MOUNIER L. Finding buffer overflow inducing loops in binary executables[C]//Proc of IEEE International Conference on Software Security and Reliability. 2012:177-186.

[8] 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.

[9] 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.

[10] SREEDHAR V C, GAO G R, LEE Y F. Identifying loops using DJ graphs[J]. ACM Trans on Programming Languages and Systems, 1996, 18(6):649-658.

[11] FLAKE H. More fun with graphs[K]. [S. l. ]: Black Hat Federal, 2003.

[12] SILBERMAN P. Loop detection[EB/OL]. [2012-08-10]. http://old. idapalace. net/papers/loopdetection. pdf.

[13] Hex-Rays. Hex-Rays decompiler v1.0[EB/OL]. [2012-08-10]. http://www. hex-rays. com/files/hexrays_info. pdf.