ISSN 1000-0054 清华大学学报（自然科学版） 2016 年 第 56 卷 第 1 期 5/17
CN 11-2223/N J Tsinghua Univ (Sci & Technol), 2016, Vol. 56, No. 1 28-34

基于污点分析和符号执行的漏洞签名生成方法

辛 伟， 时志伟， 郝永乐， 董国伟

（中国信息安全测评中心，北京 100085）

摘 要：漏洞签名是指触发程序漏洞的输入的集合，利用漏洞签名对程序输入进行过滤是一种有效的保护漏洞程序的方法。该文主要研究漏洞签名的生成技术，提出了一种有效的基于污点分析和符号执行的漏洞签名生成方法，它通过污点信息传播定位输入中的与触发漏洞相关的字节，然后，通过符号执行得到路径约束，并通过约束求解得到最终的漏洞签名。基于开源项目 Pin 和 Z3，该文构建了基于污点分析和符号执行的漏洞签名生成原型系统 TASEVS，并对漏洞程序进行了验证。实验结果表明，TASEVS 能有效地生成漏洞签名。

关键词：二进制程序；漏洞签名；污点分析；符号执行；约束求解

中图分类号：TP 309 **文献标志码**：A

文章编号：1000-0054(2016)01-0028-07

DOI：10.16511/j.cnki.qhdxxb.2016.23.006

Approach ofgenerating vulnerability signature based on taint analysis and symbolic execution

XIN Wei, SHI Zhiwei, HAO Yongle, DONG Guowei

(China Information Technology Security Evaluation Center, Beijing 100085, China)

Abstract: A vulnerability signature matches a set of inputs which trigger software vulnerability. Application of vulnerability signature to input filtering is one of the most popular and effective defense mechanisms for protecting vulnerable programs against exploits. A method for generating vulnerability signature was developed using taint analysis and symbolic execution. The method locates bytes in input that direct execution to vulnerable points using taint analysis. Path constraints are generated via dynamic symbolic execution with the final vulnerability signature obtained through constraint solving. A proof-of-concept system, TASEVS, was implemented based on instrumentation tool Pin and constraint solver Z3. Experimental results show that the TASEVS can effectively generate vulnerability signature.

Key words: binary-executable-oriented software; vulnerability signature; taint analysis; symbolic execution; constraint solving

漏洞是软件系统或产品在设计、实现、配置、运行等过程中，由操作实体有意或无意产生的缺陷、瑕疵或错误，它们以不同形式存在于信息系统的各个层次和环节之中，而且随着信息系统的变化而改变。漏洞一旦被恶意主体所利用，就会造成对信息系统的安全损害，从而影响构建于信息系统之上正常服务的运行，危害信息系统及信息的安全属性[1]。近年来，云计算、物联网的大规模部署，以及社交网络、微博、移动互联网的蓬勃发展，带给用户方便的同时，也增加了对这些系统被漏洞攻击的可能性；与此同时，各类信息安全事件如“红色代码”事件、“震网病毒”事件、“火焰病毒”事件、“CSDN 拖库”事件等层出不穷，这些信息安全事件都存在一个共同点，那就是信息系统或软件自身存在可被利用的漏洞。微软宣布停止 Windows XP 补丁更新服务，如何应对 XP 出现新漏洞后快速地对各种攻击进行拦截的问题也成为了热点。从利用漏洞进行病毒传播的速度上来讲，Slammer 病毒在 10 min 之内就感染了系统 90 %的主机，这个时间数值未来可能还会降低[2]，安全人员很难及时做出响应。面对如此严峻的漏洞带来的安全挑战，如何快速而准确地预防漏洞出现之后、补丁发布之前的漏洞攻击，成为漏洞分析领域新的研究热点。

一般来讲，可以将软件漏洞分析划分为源代码漏洞分析技术和二进制漏洞分析技术两大类。源代码漏洞分析需借助程序的源代码，具有一定的局限性；二进制漏洞分析是针对程序的二进制代码进行分析以检测其安全性、发现软件漏洞的方法[3]。传统的二进制漏洞分析主要包括模糊测试、动态污点分析、二进制代码比对、智能灰盒测试、符号执行等。

收稿日期：2014-10-28

基金项目：国家“八六三”高技术项目(2012AA012903)；国家自然科学基金资助项目(61272493)

作者简介：辛伟(1981—)，男(汉)，山东，助理研究员。E-mail：xinw@itsec.gov.cn

国内外学者已经对二进制代码分析技术进行了广泛而深入的探讨，出现了很多覆盖多种语言、集多种分析方法于一体、分析精度和准确度较高的工具，如 BitBlaze[4]、Peach[5]、Sulley[6]等。

漏洞签名借用了数字签名的思想，对触发漏洞的程序输入产生一个签名，如果一个程序的输入与该签名相匹配，那么表明程序的输入是一个不安全的输入。Wang 等[7]在 SIGCOMM 2004 上第一次提出了漏洞签名的概念。Borisov 等[8]对其进行了扩展，将漏洞签名的思想用在协议分析中。Brumley 等[9]对漏洞签名进行了形式化的定义，并将漏洞签名表示为 3 种形式：图灵机签名、符号约束签名、正则表达式签名。漏洞签名技术主要包含签名的生成和漏洞签名的匹配两个方面的研究。关于漏洞签名的生成技术方面，Costa 和 Crowcroft 等研究者[10]提出了一项漏洞签名生成方法——Vigilante，该签名本质上是一种符号约束签名，但是仅仅只创建了一个针对到达漏洞点的一条执行路径的签名，在覆盖率方面有所欠缺；Brumley 等[11]采用了静态方法，Costa 等[12]和 Cui 等[13]采用动态方法试图增加漏洞签名的路径覆盖率；Newsome 等[14]研究出一种具体执行过滤器 VSEF，该过滤器可以看作图灵机签名的一种类型。在漏洞签名的匹配方面，Paxson[15]提出了一个正则表达式匹配的引擎 Bro，系统假设存在多个正则表达式的签名，对输入进行过滤就是同时与这些签名进行匹配的过程。Schear 等[16]提出了一种符号约束漏洞签名的匹配框架，采用模式匹配和控制逻辑相结合的方法。Li 等[17]提出了候选人选择算法解决大量符号约束漏洞签名的匹配问题。

本文首先对漏洞签名的研究框架进行了描述；然后，对污点分析技术和符号执行技术分别进行了介绍；随后，对基于这两种技术的漏洞签名的生成方法及步骤进行了详细描述；最后，基于二进制插桩工具 Pin 和约束求解工具 Z3 实现了漏洞签名生成原型系统 TASEVS，对并对漏洞程序进行了验证。

1　漏洞签名框架

为了更好地形式化地描述漏洞签名，首先需要对漏洞签名相关的几个概念进行数学描述。用(P, i_p, c)表示漏洞。其中：P 表示程序，由一系列指令组成；c 表示漏洞触发条件；i_p 表示导致程序执行异常的指令，称为漏洞点。程序 P 对于输入 x 的执行轨迹可以用 $T(P,x)$表示，如果 $T(P,x)$满足漏洞条件 c，用 $T(P,x)|=c$ 表示，$L_{P,c}$ 表示满足 $T(P,x)|=c$ 的所有输入组成的语言。漏洞签名是一个匹配函数 Match，它不需要运行程序 P；对于输入要么返回 1，表示输入 x 是一个不安全输入，否则，返回 0，表示输入 x 是安全的。一个完美的漏洞签名满足以下属性：

$$\mathrm{Match}(x)=\begin{cases}1, & x\in L_{P,c};\\ 0, & x\notin L_{P,c}.\end{cases}$$

一个漏洞签名具备完整性，则满足

$$\forall x\in L_{P,c}\Rightarrow \mathrm{Match}(x)=1.$$

一个漏洞签名具备可靠性，则满足

$$\forall x\notin L_{P,c}\Rightarrow \mathrm{Match}(x)=0.$$

漏洞签名的具体形式可以是图灵机，即可以识别漏洞的一段程序；或者是符号约束，例如 ver==1 && method=="put" && len(buf)>300；或者是正则表达式，例如. * Abc. * \x90+de[^\r\n]{30}。

漏洞签名主要用在对用户的不安全输入进行过滤，用户的输入在到达漏洞程序之前先经过输入过滤器，过滤器可以采用基于漏洞签名的过滤方法。漏洞签名通过漏洞信息和程序的二进制分析产生。如果用户的输入经过漏洞签名判断后为一个不安全输入，那么就将该输入丢弃；如果经过漏洞签名判断后为一个安全输入，那么，将该安全输入发送给有漏洞的程序。整个过程如图 1 所示。本文主要研究漏洞签名的生成方法，采用污点分析和符号执行相结合的技术。下面分别对这两种技术进行介绍。

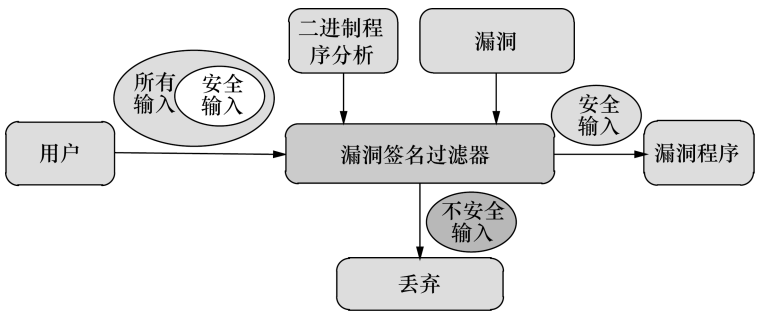

图 1　基于漏洞签名的输入过滤过程示意图

2 理论基础

2.1 污点分析

污点分析是一种跟踪并分析污点信息在程序中流动的技术[18]。它的分析对象是污点信息流。污点或者污点信息在字面上的意思是受到污染的信息或者“脏”的信息。在程序分析中，常常将来自程序之外的并且进入程序的信息当作污点信息，这时污点信息可以指程序接收的外部输入数据，也可以指程序捕捉到的来自外界的信号，如鼠标点击等。此外，根据分析的需要，程序内部使用数据也可作为污点信息，并分析其对应的信息的流向。根据污点分析时是否运行程序，可以将其分为静态污点分析和动态污点分析。

污点分析的过程常常包括以下几个部分：识别污点信息在程序中的产生点并对污点信息进行标记；利用特定的规则跟踪分析污点信息在程序中的传播过程；在一些关键的程序点检查关键的操作是否会受到污点信息的影响。一般情况下，将污点信息的产生点称为 Source 点，污点信息的检查点称为 Sink 点。相应的识别程序中 Source 点和 Sink 点的分析规则分别称为 Source 点规则和 Sink 点规则。Source 点规则、Sink 点规则以及污点信息的传播规则称为污点分析规则[19-20]。目前有代表性的工具有 TaintCheck[21]和 Flayer[22]等。

图 2 是一个污点分析过程的示例。在这个示例中，将 scanf 所在的程序点作为 Source 点，将通过 scanf 接收的用户输入数据标记为污点信息，并且认为存放它的变量 x 是被污染的。如果在污点传播规则中规定“如果二元操作的操作数是污染的，那么二元操作的结果也是污染的”，那么对于语句“$y=x+k$”，由于 x 是污染的，因此 y 也被认为是污染的。一个被污染的变量如果被赋值为一个常数，它将被认为是未污染的。例如图 2 中的赋值语句“$x=0$；”，将 x 从污染状态转变为未污染。循环语句 whlie 所在的程序点在这里被认为是一个 Sink 点，如果污点分析规则规定“循环的次数不能受程序输入的控制”，那么在这里就需要检查变量 y 是否是被污染的。本文主要通过污点分析定位程序输入中的与触发漏洞相关的字节。

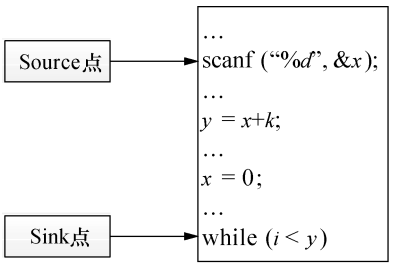

图 2 污点分析过程示意图

2.2 符号执行

符号执行首先由 King[23]在 1976 年提出，是一种使用符号值代替数字值执行程序的技术。符号是表示一个取值集合的记号。使用符号执行分析程序时，对于某个表示程序输入的变量，通常使用一个符号表示它的取值，这个符号可以表示程序在此处接受的所有可能的输入。此外，在符号执行的分析过程中那些不易或者无法确定取值的变量也常常使用符号表示的方式进行分析。

符号执行的分析过程大致如下：首先将程序中的一些需要关注但是又不能直接确定其取值的变量用符号表示其取值，然后通过逐步分析程序可能的执行流程，将程序中变量的取值表示为符号和常量的计算表达式。程序的正常执行和符号执行的主要区别是：正常执行时，程序中的变量可以看作被赋予了具体的值，而符号执行时，变量的值既可以是具体的值，也可以是符号和常量的运算表达式。图 3 是一个符号执行的例子，其中函数中的参数 x、y 分别用符号 a 和 b 表示。

```
1   int foo(int a, int b)
2   {
3       int x=a;
4       int y=b;
5       if (x>60){
6           x=y*2;
7           y=0;
8           if(x==128)
9               return True;
10      }
11      else{
12          x=0;
13          y=0;
14      }
15      /*...*/
16      return False;
17  }
```

图 3 符号执行源代码

基于图 3 的代码，可以得到图 4 所示的程序流程图。该程序一共有 3 条执行路径，每条路径都对应着一个路径约束(path constraint，PC)。其中返回 True 的有一条路径，经符号带入后，对应的路径约束为 $a>60$ & $(b*2)==128$；返回 False 的有两条路径，对应的路径约束为 $a<=60|(a>60\&(b*2)!=128)$。

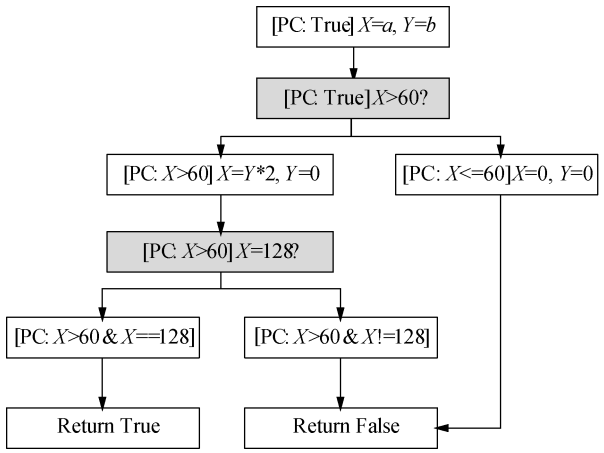

图 4　符号执行生成路径约束示意图

通过上面简单的例子，可以发现：使用符号执行技术分析程序，对于分析过程所遇到的程序中带有条件的控制转移语句(通常为条件分支语句和循环语句)，可以利用变量的符号表达式将控制转移语句中的条件转化为对符号取值的约束，通过分析约束是否可以满足，判断程序的哪条路径是可行的。

判断路径条件的可满足性是符号执行分析的关键部分。在符号执行的分析过程中，由于变量的取值被表示为符号和常量组成的表达式，路径条件被表示为对于符号的取值约束，因此，判断路径条件是否可满足的问题也就转化为判断对于符号取值的约束是否可满足的问题。对于约束是否可满足的判断，通常使用约束求解的方法[24]，约束求解的过程由约束求解器完成。约束求解器是对特定形式的约束表示进行求解的工具。在符号执行的分析过程中，常常使用可满足性模理论(satisfiability modulo theories，SMT)求解器对约束进行求解[25]。为了使用这样的约束求解器，需要将符号取值约束的求解问题转化为 SMT 问题，即一阶逻辑的可满足性判断问题。STP[26]、Z3[27]等是常用的 SMT 求解器。本文通过程序的动态符号执行技术得到与程序输入相关的路径约束，并利用约束求解工具进行约束求解。

3　方法实现

漏洞签名一般可分为两个部分，漏洞触发条件和路径约束条件。漏洞签名生成方法一般默认漏洞触发条件是已知的(也可以通过漏洞利用代码分析得到)，因此，解决漏洞签名的生成问题主要是求解路径约束条件。而路径约束条件需要满足两个要求：其一，路径约束的变量必须跟输入有关，本文将输入标定为污染源，并通过污染源的传播算法保存每一个与污染源相关的约束，通过对约束进行逆向求解便可确定与输入相关字节；其二，路径约束条件必须能够到达漏洞点，本文采用程序切片技术得到到达漏洞点的程序分片。

本文基于 Pin 和 Z3 实现了漏洞签名生成的原型系统 TASEVS，该系统分为预处理模块、污点分析模块和漏洞签名生成模块，如图 5 所示。

1) 预处理。给定需要分析的二进制目标软件，在接收到任何输入之前对程序 P 进行预处理，反汇编程序 P，对汇编程序进行动态插桩。为了降低程序分析的复杂度，可以利用切片技术，得到一个更小的程序分片，该分片包括到达漏洞点的所有路径。Valgrind 和 Pin 为目前主流插桩工具，但它们的机制不同，前者将基本块翻译成中间语言再进行插桩，插桩后的代码再翻译为 x86 基本块。Pin 直接进行代码级插桩，因此速度更快，本文基于 Pin 进行二进制程序的动态插桩。

2) 污点分析。由污点分析引擎对程序进行动态的污点分析。首先，将程序输入标定为污染源，利用污染传播算法，将分析结果输出到污点结果分析器，它利用依赖性分析定位输入中的哪些字节导致程序执行流到达漏洞点。在污点传播过程中主要记录两种类型的指令。一类称为传播类，它们会传播污染，如 mov 指令、push/pop 指令。另一类称为消除类，它们会将污染的数据删除，比如，一个被污染的变量如果被赋值为一个常数。通过双向链表跟踪污点传播，反向遍历链表便可确定路径约束与输入相关字节。

3) 计算漏洞签名。采用二进制程序遍历技术和动态符号执行技术，选择程序 P 中的到达漏洞点的路径集来计算漏洞签名，多条路径的计算采用叠加法。具体步骤如下。

步骤 1　运行二进制程序，得到第一个路径约束，保存该约束；

步骤 2 对该约束进行求解，并根据求解结果对输入数据进行更新；

步骤 3 重新运行二进制程序，得到新的约束，执行步骤 2；

步骤 4 对到达漏洞点的约束求解结果进行合并。

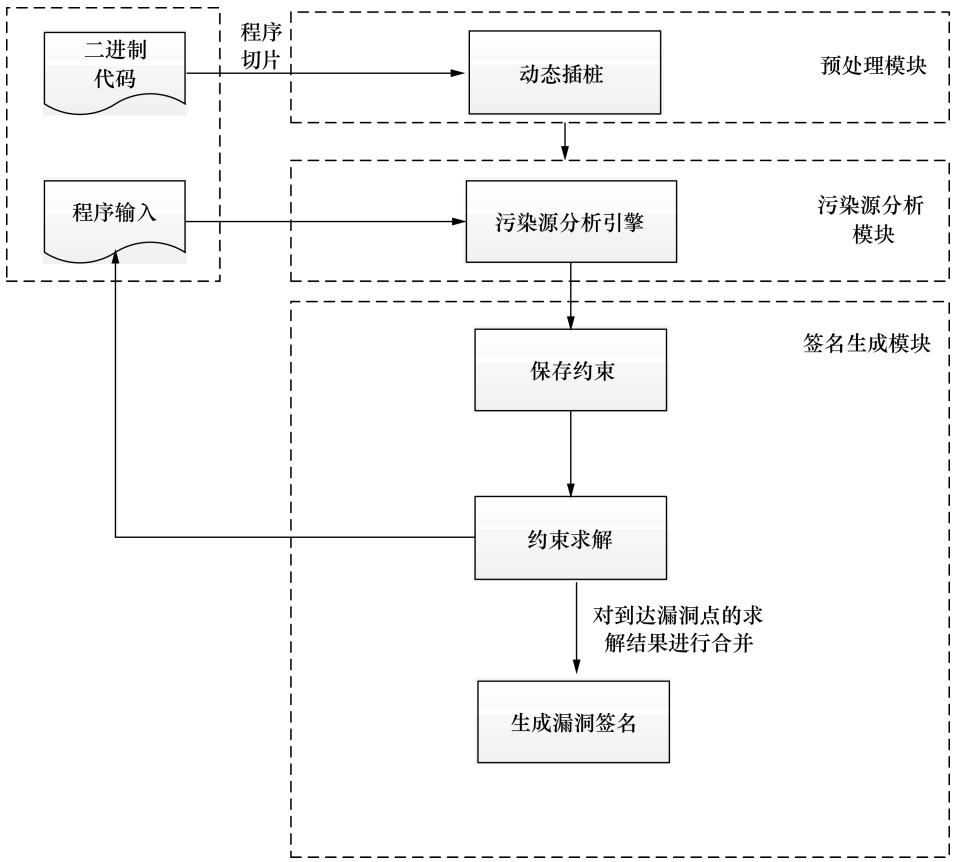

图 5 实现方法原理图

4 实 验

4.1 实验环境

本文的实验环境是运行在 Intel Core2 Q9550 2.83 GHz 和 4G 内存等硬件平台上的 Ubuntu 12.04 操作系统，Pin 采用 linux 环境下的 2.13 版本，Z3 采用的是 4.3.0 版本。

4.2 漏洞程序

实验采用图 6 所示的漏洞程序。程序读取文件 taint.txt，内容存入 buf，进行一系列判断后，将 buf 拷贝到 vulBuf 中去。由于程序没有对 strcpy 函数中的变量 vulBuf 进行长度的检查，导致如果 buf 的长度大于 10，将产生缓冲区溢出，也就是说程序的漏洞触发条件为 length (buf)＞10。同时，程序要想到达该漏洞点，必须满足如下的路径条件：(buf[0]＝＝‘*a*’‖ buf[1]＝＝‘*b*’)&& buf[2]＝＝‘*c*’。综合路径条件和漏洞触发条件，得到漏洞签名为：(buf[0]＝＝‘*a*’‖ buf[1]＝＝‘*b*’)&& buf[2]＝＝‘*c*’&& length (buf)＞10。

4.3 实验结果

实验通过程序是否输出 Success 作为触发漏洞的标志(返回 True)。程序每运行到一个路径约束，就利用 Z3 进行约束求解，并将结果更新至 taint.txt 中，用字符数组 Buff 来表示 taint.txt 中的内容，表 1 是约束求解的结果。对返回 True 的约束进行合并即可得到漏洞的路径约束条件为：(Buff[0]＝＝‘*a*’‖ Buff[1]＝＝‘*b*’)&&Buff[2]＝＝‘*c*’，加上漏洞点的条件，得到最终的漏洞签名为：(Buff[0]＝＝‘*a*’‖ Buff[1]＝＝‘*b*’)&& Buff[2]＝＝‘*c*’&& length (Buff)＞10，与前面的分析结果一致。

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <stdlib.h>
4  #include <fcntl.h>
5
6  int main(void)
7  {
8    int fd,i=0;
9    char buf[20]={0};
10    char vulBuf[10];
11
12    fd=open("taint.txt", O_RDONLY);
13    read(fd, buf, 256);
14    close(fd);
15
16    if (buf[0]=='a' ||
17    buf[1]=='b')
18    {
19    if (buf[2] != 'c')
20    return False;
21    strcpy(vulBuf, buf);
22    printf("Success\n");
23
24    return True;
25    }
26    return False;
27 }
```

图 6　漏洞程序

5　结束语

本文提出了一种有效的基于污点分析和符号执行的漏洞签名生成方法，它通过污点信息传播定位输入中的与触发漏洞相关的字节，然后，通过符号执行得到路径约束，通过约束求解得到最终的漏洞签名。基于开源项目 Pin 和 Z3，本文构建了一个漏洞签名生成原型系统 TASEVS，并对漏洞程序进行了实验，实验结果表明 TASEVS 能有效地生成漏洞签名。TASEVS 系统还有很多不完备的地方，未来将加强以下几方面工作：

1）加强汇编指令分析的精度；

2）在 TASEVS 中加入正则表达式匹配模块；

3）将系统应用于真实的网络型软件的漏洞签名生成。

漏洞签名技术，对保证程序的安全性具有重要的现实和理论意义。目前这个研究方向还在起步阶段，相信随着漏洞技术的进步和人们关注度的提高，该领域的研究将会得到进一步的发展。

表 1　实验结果

序号	路径约束条件	返回值
1	Buff[0] ! ='a' && Buff[1] ! ='b'	False
2	Buff[0] ! ='a' && Buff[1]=='b' && Buff[2] ! ='c'	False
3	Buff[0] ! ='a' && Buff[1]=='b' && Buff[2] ! ='c'	False
4	Buff[0]=='a' && Buff[1]=='b' && Buff[2] ! ='c'	False
5	Buff[0] ! ='a' && Buff[1]=='b'&& Buff[2]=='c'	True
6	Buff[0]=='a' && Buff[1] ! ='b' && Buff[2]=='c'	True
7	Buff[0]=='a' && Buff[1]=='b' && Buff[2]=='c'	True

参考文献　(References)

[1] 吴世忠，刘晖，郭涛，等. 信息安全漏洞分析基础 [M]. 北京：科学出版社，2013.
WU Shizhong, LIU Hui, GUO Tao, et al. Fundamentals of information security vulnerability analysis [M]. Beijing: Science Press, 2013. (in Chinese)

[2] Moore D, Paxson V, Savage S, et al. Inside the slammer worm [C]// Proceedings of IEEE Security and Privacy. New York, USA: IEEE Press, 2003: 33 - 39.

[3] 严俊，郭涛，阮辉，等. JUTA：一个 Java 自动化单元测试工具 [J]. 计算机研究与发展，2010，**47**(10)：1840 - 1848.
YAN Jun, GUO Tao, RUAN Hui, et al. JUTA: An automated unit testing framework for Java [J]. *Journal of Computer Research and Development*, 2010, **47**(10): 1840 - 1848. (in Chinese)

[4] Song D, Brumley D, Yin M, et al. BitBlaze: A new approach to computer security via binary analysis [C]// Proceedings of the 4th International Conference on Information Systems Security. New York, USA: ACM Press, 2008: 147 - 162.

[5] Déjà vu Security. Peach [Z/OL]. (2014-10-10). http://peachfuzzer.com/.

[6] Pedram A. Sulley [Z/OL]. (2014-10-10). http://code.google.com/p/sulley/.

[7] Wang H, Guo C, Simon D. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits [C]// Proceedings of the 2004 ACM SIGCOMM Conference. Chicago, USA: ACM, 2004: 193-204.

[8] Borisov N, Brumley D. Ageneric application-level protocol parser analyzer and its language [C]// Proceedings of the 14th Annual Network and Distributed System Security Symposium. San Diego, USA: The Internet Society, 2007: 89-95.

[9] Song D, Brumley D, Yin M, et al. BitBlaze: A new approach to computer security via binary analysis [C]// Proceedings of the 4th International Conference on Information Systems Security. New York, USA: ACM Press, 2008: 147-162.

[10] Costa M, Crowcroft J, Castro M. Vigilante: End-to-end containment of internet worms [C]// Proceedings of the 20th ACM Symposium on Operating System Principles. Chicago, USA: ACM, 2005: 133-147.

[11] Brumley D, Wang H, Song D. Creating vulnerability signatures using weakest pre-conditions [C]// Proceedings of IEEE Computer Security Foundations. Venice, Italy: IEEE Press, 2007: 311-325.

[12] Costa M, Castro M, Zhou L. Bouncer: Securing software by blocking bad input [C]// Proceedings of ACM Symposium on Operating Systems Principles. Chicago, USA: ACM, 2007: 117-130.

[13] Cui W, Peinado M, Wang H. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing [C]// Proceedings of IEEE Symposium on Security and Privacy. Berkeley, USA: IEEE Press, 2007: 252-266.

[14] Newsome J, Dawn S. Vulnerability-specific execution filtering for exploit prevention on commodity software [C]// Proceedings of the 13th Annual Network and Distributed System Security Symposium. San Diego, USA: The Internet Society, 2006: 1-14.

[15] Paxson V. Bro: A system for detecting network intruders in real-time [C]// Proceedings of the 7th USENIX Security Symposium. San Antonio, Texas, 1998.

[16] Schear N, Albrecht D, Borisov N. High-speed matching of vulnerability signatures [C]// Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection. Berlin, Germany: Springer, 2008: 155-174.

[17] Li Z, Xia G, Gao H, et al. NetShield: Massive semantics-based vulnerability signature matching for high-speed networks [J]. *ACM Sigcomm Computer Communication Review*, 2010, **40**(4): 279-290.

[18] Denning D. Alattice model of secure information flow [C]// Proceedings of Communications of the ACM. Chicago, USA: ACM, 1976: 236-243.

[19] Schwartz E, Avgerinos T, Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution [C]// Proceedings of IEEE Symposium on Security and Privacy. New York, USA: IEEE Press, 2010: 317-331.

[20] Lam M, Martin M, Livshits B. Securing web applications with static and dynamic information flow tracking [C]// Proceedings of the 2008 ACM SIGPLANSymposium on Partial Evaluation and Semantics-based Program Manipulation. Chicago, USA: ACM, 2008: 3-12.

[21] Newsome J, Song D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software [C]// Proceedings of the 2007 International Symposium on Software Testing and Analysis. New York, USA: ACM, 2005: 104-123.

[22] Drewry W, Ormandy T. Flayer: Exposing application internals [C]// Proceedings of USENIX Workshop on Offensive Technologies. Berkeley, USA: ACM, 2007: 1-9.

[23] King J. Symbolic execution and program testing [J]. *Communications of the ACM*, 1976, **19**(7): 385-394.

[24] Gallaire H. Logic programming: Future developments [C]// IEEE Symposium on Logic Programming. Boston, USA: IEEE Press, 1985: 88-96.

[25] Barrett C, Sebastiani R, Seshia S, et al. Handbook of Satisfiability [M]. Amsterdam: IOS Press, 2009.
Vijay G. STP [EB/OL]. (2014-10-10). http://people.csail.mit.edu/Vganesh/STP_files/stp.html.

[26] Vijay G. STP [EB/OL]. (2014-10-10). http://people.csail.mit.edu/Vganesh/STP_files/stp.html.

[27] Moura L, Bjorner N. Z3: An efficient SMT solver [M]// Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Germany: Springer, 2008: 337-340.