基于符号执行的高危 unlink 漏洞判定

黄　宁[1]，黄曙光[1]，黄　晖[1]，常　超[1]

[1]（国防科技大学，安徽 合肥　230037）

通讯作者：黄宁，E-mail：809848161@qq.com

摘要：unlink 是一种常见的堆释放操作。目前，针对 unlink 操作发起的攻击愈加频繁。Linux 系统已经发展出了多种保护机制防止 unlink 操作被利用，但这仍然不能避免 unlink 攻击的发生。本文将基于堆溢出漏洞发起的 unlink 攻击定义为 unlink 漏洞，并将其中可能导致任意内存读写的 unlink 漏洞定义为高危 unlink 漏洞。为了提高程序的安全性，实现对高危 unlink 漏洞的检测，本文提出了一个高危 unlink 漏洞模型，并根据该模型设计了高危 unlink 漏洞判定系统。该系统通过污点传播实现了对程序输入数据以及敏感操作的监控；通过符号执行构建程序污点变量传播的路径约束以及高危 unlink 漏洞的触发约束；通过对约束条件的求解，检测程序 unlink 漏洞是否能实现任意内存读写。实验证明，该系统能有效地对 unlink 漏洞进行分析，判断该漏洞是否属于高危 unlink 漏洞。

关键词：高危 unlink 漏洞；符号执行；污点分析；漏洞分析

中图法分类号：TP311

Identification of high - risk unlink vulnerability based on symbolic execution

Abstract： The unlink vulnerability is a common heap overflow vulnerability. At present, the Linux has developed a variety of protection mechanisms to prevent unlink vulnerabilities for exploitation, but they still cannot avoid unlink attacks. This paper defined the unlink vulnerabilities which can lead to any memory read and write as high-risk unlink vulnerabilities. In order to improve the security of the program, to detect high-risk unlink vulnerabilities, this paper presents a high-risk unlink vulnerability model, and proposed a high-risk unlink vulnerability detection and analysis method according to the model. This method realizes the monitoring of program input data and sensitive operation through taint analysis; using the symbolic execution to build path constraints of tainted variable and trigger constraint of high-risk unlink vulnerability; through the solution of the constraints, analyze that if the unlink vulnerability can achieve any memory read and write. The experiment prove that this method can effectively judge that if the unlink vulnerability is a high-risk unlink vulnerability.

Key words： high-risk unlink vulnerabilities, symbolic execution, taint analysis, vulnerability analysis

1 引言

随着信息技术的发展，软件的安全性也越来越受到人们的重视。近年来，各种软件保护机制层出不穷。与此同时，针对这些保护机制的漏洞利用技术也在不断发展。在二进制软件漏洞中，控制流劫持类漏洞无疑是危害性最大的漏洞种类之一。当程序控制流被劫持之后，极有可能导致任意代码攻击。针对二进制软件漏洞，特别是可能导致控制流劫持漏洞的检测分析，是一款合格软件在被推出之前必须完成的工作。

堆溢出是一种常见的导致控制流劫持的漏洞成因。在 Linux 环境下，典型的堆溢出攻击技术包括 fast bin，unsafe unlink 等。这些技术往往是利用堆块自身的结构，以及系统对堆块进行分配、释放与合并等操作，对存在堆溢出漏洞的程序进行攻击。本文将基于堆溢出漏洞发起的 unlink 攻击定义为 unlink 漏洞，并将其中可能导致任意内存读写的 unlink 漏洞定义为高危 unlink 漏洞。针对常见的 unlink 攻击，glibc 在堆块释放过程中设置了双链表检测，Double free 检测，堆块大小检测等机制。通过近年来的实践可知，虽然这些机制对防御 unlink 攻击发挥了重要作用，但仍无法完全避免攻击的发生。因此，对程序中存在的 unlink 漏洞进行分析，检测出可能导致任意内存读写等严重后果的 unlink 漏洞就显得十分必要。

另一方面，对程序漏洞检测的技术也日臻完善。许多以污点分析，符号执行等技术为基础的漏洞检测工具已经能有效地检测出传统的缓冲区溢出类漏洞，较具代表性的工具有 S2E[1][2]等。以此为基础，　T. Avgerinos 等人提出的 AEG[3]以及 S.K. Huang 等人提出 CRAX[4]等程序崩溃分析方法，为程序漏洞检测提供了一个新的思路。崩溃分析方法使用符号执行对目标程序

进行检测，并构造出可能造成程序崩溃的路径约束及漏洞触发条件。通过求解路径约束，构造可触发漏洞的输入数据，造成程序崩溃，从而实现对控制流劫持类漏洞的分析。但是，上述方法也都存在着不同的局限性。以 CRAX 为例，该工具通过对进程 EIP 寄存器的监视，实现了对控制流劫持类漏洞的检测。但针对一些具体的场景，例如攻击者通过精心构造的数据绕过保护机制，或篡改程序指针变量等，则无法进行检测。而文献[5]则提出了一种可以实时检测漏洞攻击的方法，为防御控制流劫持类漏洞攻击提供了一个新的思路。但该方法主要考虑的是数据执行保护和地址空间分部随机化的条件下进行的漏洞攻击检测，实验条件也多是针对 Windows 系统中的应用程序，缺乏对 Linux 系统和 glibc 特有保护机制的考虑。

为了更全面地对存在 unlink 漏洞的程序安全性进行分析，本文构建了高危 unlink 漏洞模型。该模型充分考虑了目前 unlink 攻击时可能遇到的保护机制以及可能绕过这些机制的攻击方法。在高危 unlink 漏洞模型的基础上，本文还设计了一个高危 unlink 漏洞判定系统。该系统使用的高危 unlink 漏洞判定算法，首先通过污点分析，实现了对程序的可控变量及相关敏感操作的监控；然后利用符号执行技术，设定漏洞触发条件并构建程序的路径约束；求解约束条件，生成测试用例；通过测试用例，分析 unlink 漏洞是否能导致任意内存读写，从而判定该漏洞是否属于高危 unlink 漏洞。

2 相关原理

2.1 Glibc堆块管理机制

Linux 系统会给程序分配一块大小为 132KB 的内存空间。该内存空间被称为 main arena。当程序申请堆内存时，系统会从 main arena 中将相应大小的内存块分配给程序[6]。

对于堆内存管理而言，堆块（chunk）就是最小操作单位。Glibc malloc 将整个堆内存空间分成了连续的、大小不一的 chunk。根据程序申请的堆块大小的不同，glibc 将堆块分为三种类型：堆块数据部分大小为 16 至 64 字节的被称为 fast chunk；堆块数据部分大小小于等于 512 字节的被称为 small chunk；大于 512 字节的称为 large chunk。

堆块在被释放后，其信息将会存入一个被称为 bin 的数据结构中，构成一个已释放堆块的链表。相应的，每种堆块在释放后，其信息存储在不同的链表中。Glibc 定义了四种链表：fast bin、small bin、large bin 和 unsorted bin。glibc 一共维护了 136 条链表，每条链表都存储着相同大小的已释放堆块（free chunk）信息。以 small bin 为例，其链表结构如图 1 所示。

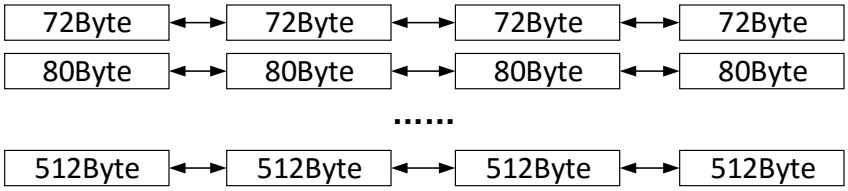

Fig.1 Structure of small bin

图 1 small bin 链表结构示意图

从堆块的使用状态方面看，可被分成四种状态：已分配堆块（allocated chunk）、已释放堆块（free chunk）、进程堆块（top chunk）和保留堆块（Last remainder chunk）。所有类型的 chunk 都是内存中一块连续的区域，通过该区域中特定位置的标识符可以区分不同状态的堆块[7]。

Fig.2 Structure of large chunk header

图 2 large chunk 头部结构示意图

图 2 显示的是 small chunk 的头部结构。当前一个堆块为已释放（free）状态时，第 1 到 4 字节表示的是在内存空间中，

与当前堆块相邻的前一个堆块的大小。第 4 到 8 字节为当前堆块的大小。由于每个堆块的大小一定是 8 字节的整数倍，因此，最后一个比特位被用作表示前一个堆块的状态。前一个堆块为 free 状态，则 F 标志位为 0；若是已分配（allocated）状态，则 F 为 1。当该堆块为 free 状态时，第 8 到 12 字节显示的是 bin 链表中前一个堆块的地址 FD；第 12 到 16 字节是 bin 链表中后一个堆块的地址 BK。当堆块为大于 512 字节的 large chunk 时，第 13 到 16 字节为指向前一个尺寸的堆块的地址；第 17 到 20 字节为指向后一个尺寸的堆块的地址。若当前堆块处于已分配状态，则第 8 到 20 字节存储数据信息。

2.2 unlink漏洞原理及防御机制

当一个堆块被释放后，为了避免内存碎片的产生，glibc 加入了针对内存中相邻堆块的合并机制。这种机制包括向前合并和向后合并两种情况。简而言之，向前合并就是当前被释放堆块与相邻的低地址内存中的已释放堆块进行合并的操作；向后合并则是与相邻的高地址内存中的已释放堆块进行合并。当两个已释放堆块的合并过程中，glibc 会对新形成的已释放堆块进行 unlink 操作[8]。

以图 3 所示堆块结构为例。A 图中的 chunk2 已释放，现准备释放 chunk1，则 glibc 对 chunk2 进行 unlink 操作，使 chunk1 发生向后合并；B 图中的 chunk1 已释放，现准备释放 chunk2，则 glibc 对 chunk1 进行 unlink 操作，使 chunk2 发生向前合并。

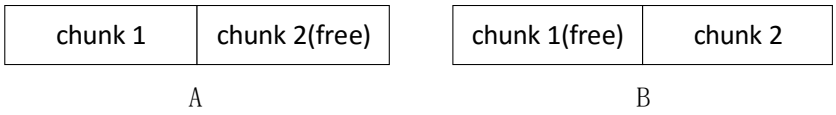

Fig.3 Distribution of chunk1 and chunk2

图 3 chunk 1 与 chunk 2 结构示意图

代码 1 显示的是 unlink 的过程代码。其中，P 指的是当前堆块释放过程中，与准备释放堆块相邻的已释放堆块地址。

代码 1

```
1  #define unlink ( P, BK, FD ){
2       FD = P->fd;
3       BK = P->bk;
4       FD->bk = BK;
5       BK->fd = FD;
6  }
```

以图 3 中的 B 图为例，传统的 unlink 攻击方法如图 4 所示。假设 chunk1 为溢出堆块，chunk2 为准备释放堆块。现通过溢出数据使系统将 chunk1 认定为已释放堆块。在 chunk1 中伪造堆块数据，使 P->fd 指向 free 函数地址-12，使 P->指向 shellcode 地址。当执行代码 1 中第 4 行代码时，FD->bk 指针指向 BK 指针，从而将 free 函数地址替换为 shellcode 地址。当程序下次调用 free 函数时，进程将开始执行 shellcode。

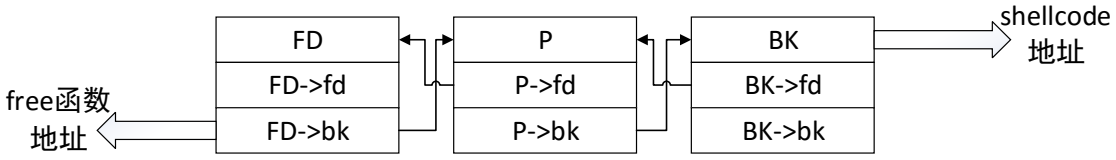

Fig.4 Procedure of traditional unlink

图 4 传统 unlink 利用示意图

针对传统的 unlink 漏洞利用方法，glibc 目前增添了三种保护机制进行防御：Double free 检测，Next size 检测以及双链表冲突检测[8]。

在 glibc 对堆块进行释放操作前，Double free 检测首先检测当前堆块是否已处于 free 状态。若当前堆块处于已释放状态，则无法再进行释放操作。

Next size 机制检测当前被释放堆块的 next chunk 大小（即 next chunk size）是否处于 8 至系统内存大小之间。当 next size 被更改为小于 0 的数值时，系统会报出 invalid next size 错误。

双链表冲突检测会在执行 unlink 操作时，检查链表中前一个堆块的 fd 与后一个堆块的 bk 是否都指向当前需要 unlink 的堆块。其代码如代码 2 所示。该机制保证了 unlink 对象的正确性。

代码 2	
1	if(FD->bk != P || BK->fd != P)
2	malloc_printerr(“corrupted double-linked list”);

3 高危险性 unlink 漏洞模型

3.1 高危险性unlink漏洞属性

判断控制流劫持漏洞能否导致任意内存读写，需要从以下四方面进行分析：是否存在产生漏洞的敏感操作；是否输入了导致漏洞触发的不合法数据；是否进行了对不合法数据的安全检查[9]；输入的不合法数据是否能绕过相应的保护机制[10]。

基于上述四个条件，本文对控制流劫持漏洞的可利用性 control_hijack 定义了四种相关属性：敏感操作属性 sensitive_operation；外部数据触发漏洞属性 trigger_bug；安全检查属性 safety_check；外部数据绕过保护机制属性 bypass_protection。只有在满足上述四个条件的情况下，控制流劫持漏洞才可能导致任意代码执行。因此，高危控制流劫持漏洞与上述四种属性的关系式如式（1）所示：

$$\begin{aligned} control\_hijack = {} & sensitive\_operation \wedge \\ & safety\_check \wedge \\ & trigger\_bug \wedge \\ & bypass\_protection \end{aligned} \quad (1)$$

Unlink 漏洞的本质是 Linux 环境下的堆溢出漏洞，而堆溢出漏洞也属于控制流劫持漏洞中的一种，因此，对高危 unlink 漏洞的分析也需要在控制流劫持类漏洞的四种属性基础上进行。

为实现通过高危 unlink 漏洞达到任意内存读写的目的，需要在程序执行过程中控制 EIP 寄存器，实现控制流劫持。由于 glibc 对 small chunk 和 large chunk 的 unlink 操作并不完全相同，因此，常见的 unlink 攻击方法也分为两种。

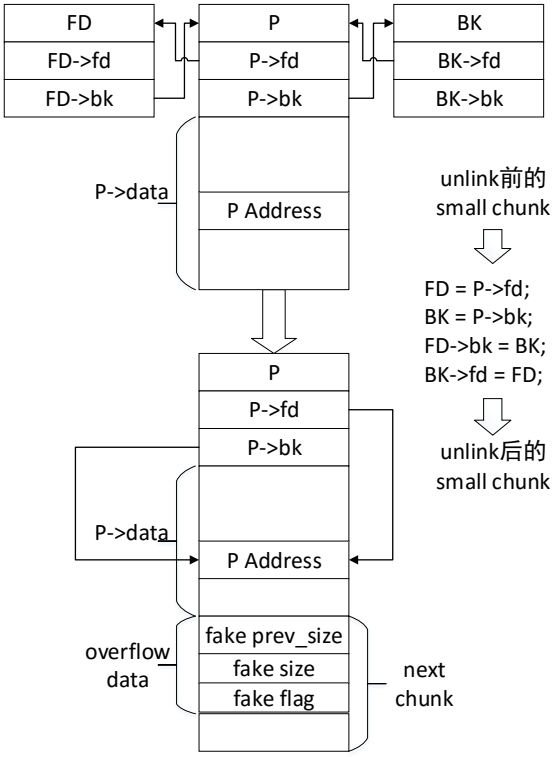

Fig.5 Unlink attack with small chunk

图 5 针对 small chunk 的 unlink 利用示意图

针对 small chunk 的 unlink 过程中需要面对的 Double free 和双链表检测机制，常用伪造堆块数据的方法实现绕过。通过溢出数据，修改 next chunk 的头部信息，使处于已分配状态的 chunk P 被系统误认为已释放；通过在内存中伪造堆块，使

chunk P 在 unlink 的过程中，其 fd 与 bk 指针指向伪堆块的数据。其过程如图 5 所示。

在绕过 Double free 和双链表检测的基础上，可通过控制程序中的指针变量实现控制流劫持。指针变量中存储的是程序的某块内存地址，通过修改指针变量可实现程序的任意内存读写。所以指针变量的可控性决定了是否能实现控制流劫持[11]。

在 glibc-2.21 之前的系统中，针对 large chunk 的 unlink 操作缺少对 fd_nextsize 和 bk_nextsize 指针的检查，因此，可以直接通过修改堆块的 fd_nextsize 和 bk_nextsize 指针，实现任意内存操作。其过程如图 6 所示。

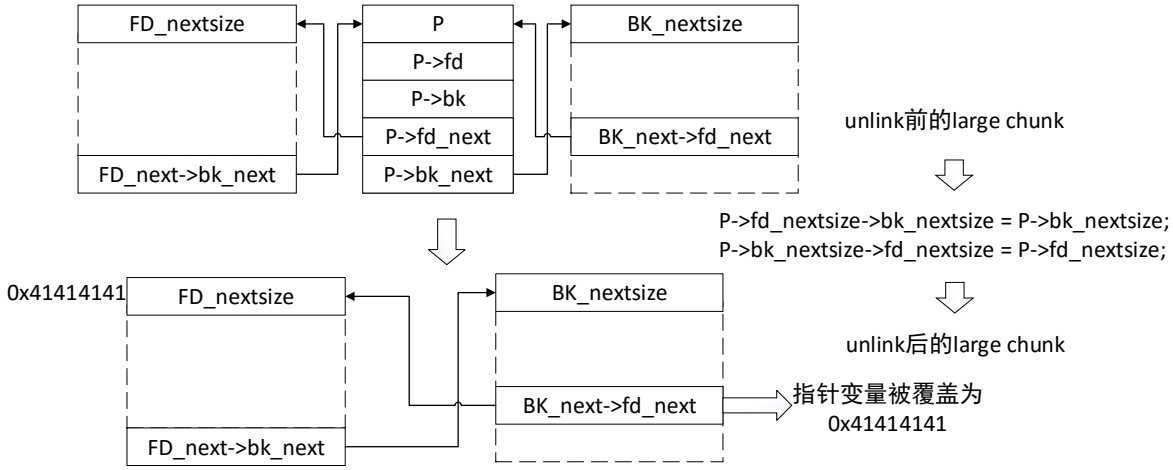

Fig.6 Unlink attack with large chunk

图 6 针对 large chunk 的 unlink 利用示意图

基于对高危 unlink 漏洞利用方法的分析，本文定义了以下四个特征对 unlink 漏洞进行描述：堆溢出触发特征 isOverflow；unlink 操作触发特征 isUnlink；溢出堆块可控性特征 isExploitable；指针变量可控性特征 isPtrChangable。

本文针对高危 unlink 漏洞及其特征有如下定义：

堆溢出触发特征 isOverflow 描述了程序堆块是否会导致溢出错误。

unlink 操作触发特征 isUnlink 描述了程序是否会发生 unlink 操作。

溢出堆块可控性特征 isExploitable 描述了溢出堆块是否处于可被外部数据控制的状态。

指针变量可控性特征 isPtrChangable 描述了指针变量是否处于可被外部数据控制的状态。

unlink 漏洞的结果状态 isUnlinkExe 描述了该漏洞是否处于可能导致任意内存读写的状态。

由于堆溢出漏洞发生的根本原因在于程序敏感操作过程错误以及未对进行敏感操作的不合法数据进行检查[12][13]，因此，堆溢出触发特征 isOverflow 又可表示为式（2）：

$$isOverflow = sensitive\_operation \wedge safety\_check \quad (2)$$

溢出堆块可控性特征 isExploitable 对溢出堆块是否能被外部数据控制进行描述。由堆溢出的原理可知，当不合法数据通过敏感操作输入溢出堆块时，程序发生堆溢出错误。因此，isExploitable 可表示为式（3）：

$$isExploitable = trigger\_bug \quad (3)$$

基于上述过程，对控制流劫持的外部数据绕过保护机制属性 bypass_protection 的描述如式（4）所示。

$$bypass\_protection \leftarrow isPtrChangable \wedge isExploitable \wedge isUnlink \quad (4)$$

综合式（1）（2）（3）（4），本文提出了高危 unlink 漏洞与四种特征的关系式如式（5）所示。

$$isUnlinkExe \leftarrow isOverflow \wedge isExploitable \wedge isPtrChangable \wedge isUnlink \quad (5)$$

3.2 高危险性unlink漏洞判定系统

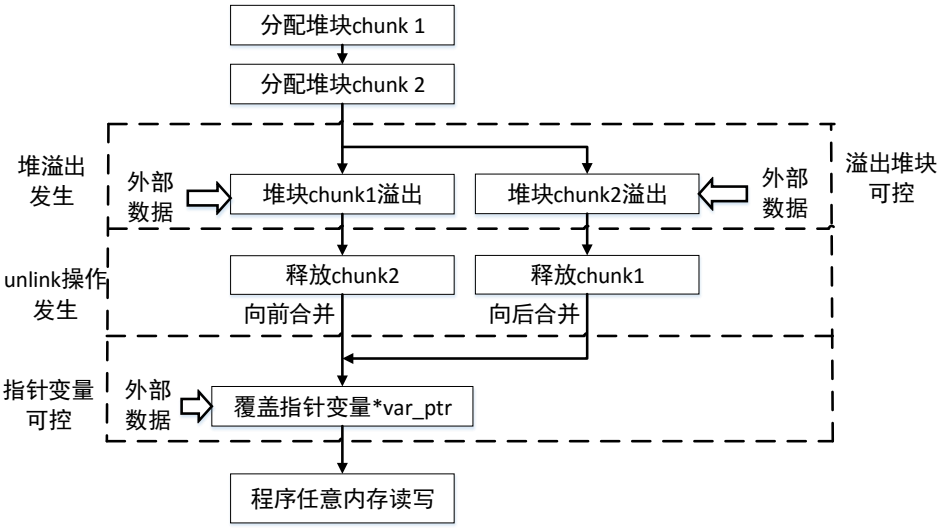

Fig.7 High-risk unlink vulnerability model

图 7 高危 unlink 漏洞发生模型示意图

根据高危 unlink 漏洞的发生原理及其特征，可将高危 unlink 漏洞分为两种情况（堆块 chunk1 相对堆块 chunk2 总处于内存的低地址处且两个堆块相邻）：

1、 chunk1 发生溢出后，释放 chunk2，向前合并，对 chunk1 进行 unlink 操作；

2、 chunk2 发生溢出后，释放 chunk1，向后合并，对 chunk2 进行 unlink 操作。

基于上述两种高危 unlink 漏洞发生场景，本文提出了高危 unlink 漏洞发生模型。该模型架构如图 7 所示。

针对高危 unlink 漏洞发生模型，本文在 S2E 的基础上，使用符号执行和污点跟踪技术，设计了一套高危 unlink 漏洞判定系统。该系统对虚拟机中动态运行的被测程序进行监控，获取程序运行数据；通过高危 unlink 漏洞判定算法，跟踪外部数据在程序中的传播路径，并以此为基础，构建可到达漏洞发生点的约束条件，生成测试用例。系统架构如图 8 所示。

该判定系统主要分为两个模块：第一个模块是堆块创建监控模块；第二个模块是堆块操作监控模块。

堆块创建监视模块的作用是，通过监视堆块创建过程，获取新建堆块的相关数据。这些数据包括：堆块数据区的起始地址、堆块数据区的长度以及堆块地址的指针值等。

堆块操作模块又分为三个子模块，分别是：堆块数据操作监视模块、堆块释放监视模块以及指针变量监视模块。

堆块数据操作监视模块对指向已申请堆块的数据流进行监视。该模块的工作包括：通过符号化内存搜索算法，检查流入已申请堆块的数据是否为符号化数据，从而判断该堆块是否能被外部数据控制。若被检查堆块范围内含有符号化数据，则通过堆溢出触发检测算法，检查符号化数据的长度是否超过相应堆块的边界，从而判断是否发生堆溢出。

堆块释放监视模块对欲释放堆块的状态进行检查。该模块的工作流程为：通过 unlink 操作触发特征检测算法，检查欲释放堆块是否与溢出堆块相邻；若相邻，则判断溢出堆块与欲释放堆块的相对位置，确定 unlink 操作模式是向前还是向后合并；在确定 unlink 操作模式的基础上，根据堆块创建监视模块所获取的数据，构造以下两项数据：溢出堆块的伪头部数据和溢出堆块的溢出数据。为了减少系统开销并确保 unlink 操作顺利进行，溢出堆块的溢出数据仅包括被溢出数据覆盖的下个堆块的头部数据。

具体数据构造完成后，系统对溢出堆块进行可控性判断。建立溢出堆块的伪头部数据约束 fake_header_constraint 和溢出数据约束 cover_data_constraint，并与此前 S2E 符号执行引擎收集的路径约束 state_constraint 进行合取，所得约束即为测试用例 1 的约束条件 testcase1_constraint。其关系式如式 6 所示。

$$testcase1\_constraint = fake\_header\_constraint \wedge cover\_data\_constraint \wedge state\_constraint \quad (6)$$

使用约束求解器 STP 对测试用例 1 的约束条件求解，生成测试用例 1。将生成的测试用例 1 做为外部数据输入程序，检查程序中 unlink 漏洞是否成功触发。若漏洞触发成功，则进入下一步；否则退出判定系统。

指针变量监视模块使用指针变量可控性检测算法，对指定的指针变量进行符号化检查。若指针变量为符号化内存，则判断该指针可被外部数据控制，表明针对该指针的操作可能导致对北侧程序的任意内存操作，从而判定该 unlink 漏洞为高危漏洞。

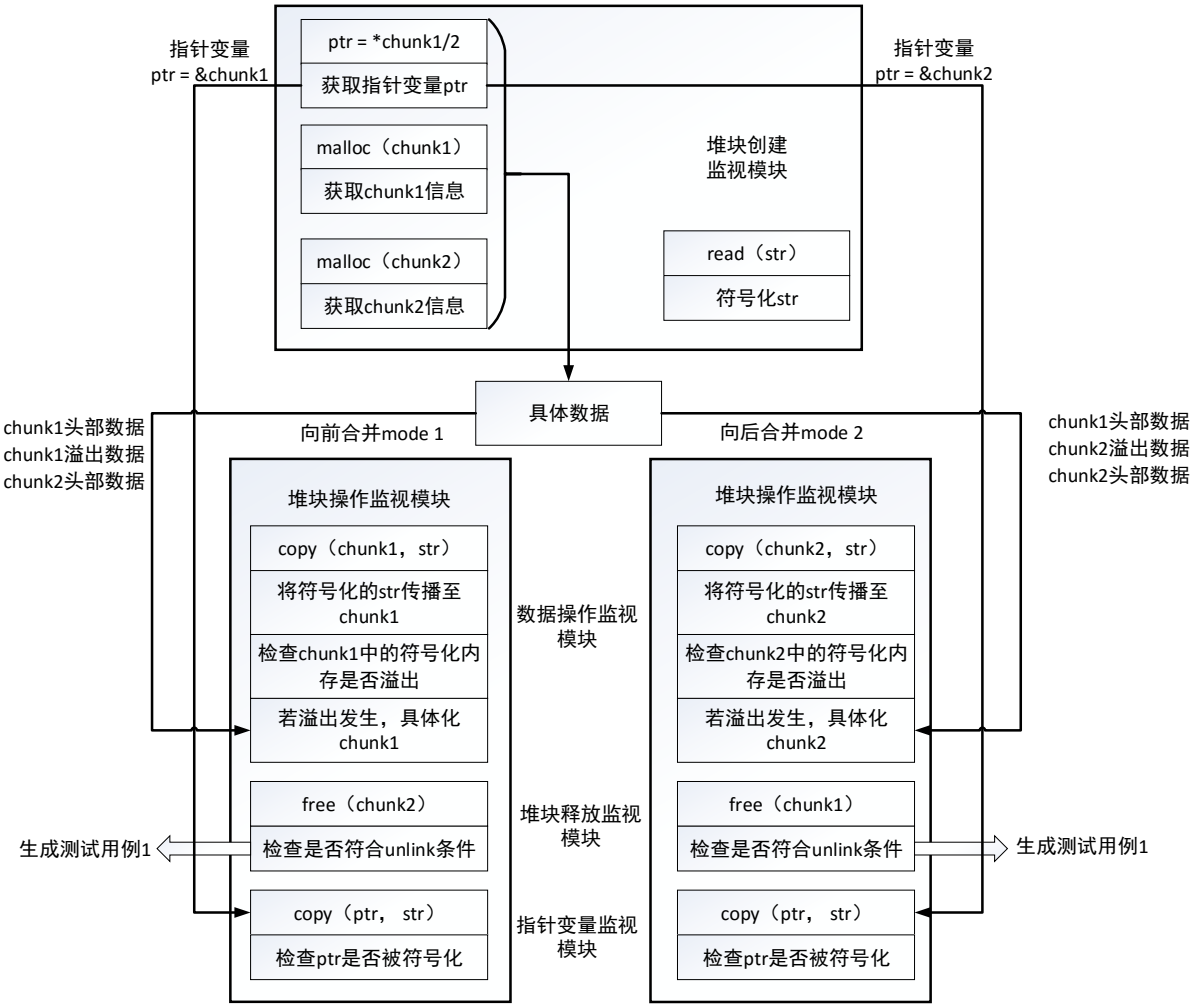

Fig.8　High-risk unlink vulnerability judgement system

图 8　高危 unlink 漏洞判定系统架构示意图

4 高危 unlink 漏洞判定算法

根据高危 unlink 漏洞检测模型，本文提出了高危 unlink 漏洞判定算法。该算法通过对 unlink 漏洞四种特征的检测，分析漏洞是否属于能导致任意内存读写的高危 unlink 漏洞。算法过程如算法 1 所示。

算法 1：高危 unlink 漏洞判定算法

输入：var_ptr（指针变量*var_ptr 地址）

got_add（系统函数的 GOT 表地址）

chunk1_inf（溢出堆块 chunk 1 信息）
chunk2_inf（被释放堆块 chunk 2 信息）
shell（shellcode 字符串）
state（符号执行的路径约束）
输出：isUnlinkExe（高危漏洞检测结果）

```
1  isOverflow = chunk_overflow( chunk1_inf , chunk2_inf ); //堆溢出触发检测
2  isUnlink = unsafe_unlink(chunk1_inf , chunk2_inf );     //unlink 操作触发检测
3  isExploitable = exploitConstraint(chunk1_inf, chunk2_inf, shell);   //溢出堆块可控性检测
4  if(isUnlink && isExploitable)
5      testcase1 = generateTestcase(state, chunk1_inf);    //生成可成功触发 unlink 漏洞的测试用例 1
6  isPtrChagable = ptrConstraint (&var_ptr, chunk1_inf );  //全局变量可控性检测
7  isUnlinkExe = isOverflow ⋀ isUnlink ⋀ isExploitable ⋀ isPtrChagable ;
8  return isUnlinkExe;
```

算法 1 具体实现在污点分析与符号执行的基础之上。通过对程序敏感操作以及外部输入的监控，实现对污点数据的跟踪；通过符号执行，实现对被测程序的路径约束及危险性约束条件的求解，并生成可触发高危 unlink 漏洞的测试用例 1。

4.1 符号化内存搜索算法

程序对外部数据进行不安全操作是触发控制流劫持漏洞的必要条件之一，因此，本文使用了基于控制流污点分析的符号执行技术，对被测程序进行动态插桩，实现对被污染变量的跟踪与传播，以及对敏感操作的监控[14]。

在动态插桩的过程中，监控器会在被测程序接收到外部输入后，对输入的数据所占内存空间进行符号化处理。当符号化数据进行复制，赋值或其他传播操作时，相应的被传播数据同样会被符号化。由此实现对污点数据传播路径的监控。符号内存的搜索过程如算法 2 所示。

算法 2：符号内存搜索算法

输入：state（被测程序当前状态）
minAddr（内存搜索起点地址）
maxAddr（内存搜索终点地址）
输出：symbolicArray（被符号化的内存块集合）

```
1  block = state->findObject;
2  for( i = minAddr ; i <= maxAddr ; i += page_size )
3      for ( j = i; j < i + block_size; j ++ )
4          if( isSymbolic( j ) )
5              if ( isPreviousSymbolic( j ) )
6                  block.width++;
7              else  block.width = 1;
8          else
9              if ( isPreviousSymbolic( j ) )
10                 symbolicArray.insert( block );
11 return  symbolicArray;
```

符号内存搜索算法需要的三个输入分别为：被测程序当前状态变量，内存搜索起点地址，内存搜索终点地址。根据 32 位 Linux 系统中应用程序内存分布特点，内存空间的搜索范围一般为 0x00000000 至 0xC0000000。在此范围内，先以内存页为单位进行搜索。算法 2 中每个内存页大小定义为 page_size。每个内存页当中，又以大小为 block_size 的内存块为单位对符号内存进行搜索。

若在内存中搜索发现某内存块为符号内存，则需做出如下判断：

（1）若前一内存块也为符号内存，则符号内存块的长度加 1；

（2）若前一内存块不为符号内存，则设符号内存块的长度为 1。

当被搜索的内存块不为符号内存时，则判断前一内存块的状态。若前一内存块为符号内存，则将前一符号内存片段的信息存入 symbolicArray 集合。

4.2 堆溢出触发判断

算法 1 中的 chunk_overflow 模块实现了对 unlink 漏洞溢出发生条件的判断。该过程详见算法 3。

算法 3：堆溢出触发特征判断算法

输入：chunk1（溢出堆块 chunk 1 的信息）
　　chunk2（被溢出堆块 chunk 2 的信息）
输出：isOverflow（unlink 漏洞溢出发生判断）

```
1   isSymbolic = symbolic_check( chunk1 ) ;
2   if( isSymbolic ){
3      isOverflow = (chunk1.addr + chunk1.size >= chunk2.addr  -  8) && (chunk1.size > 0x40) &&
(chunk2.size > 0x40) ;
4   }
5   return isOverflow;
```

算法 3 首先对溢出堆块 chunk1 进行符号化判断。通过符号化内存信息与堆块地址进行的匹配，可确定该堆块是否属于污点数据。仅当确定 chunk1 为符号化数据的情况下，算法 3 才会认定 chunk1 中的数据是可受外部影响的，并进行堆块的溢出条件判断。

算法 3 的堆块溢出条件包括以下两点：

（1）chunk1.addr + chunk1.size >= chunk2.addr - 8。由于 chunk1 和 chunk2 必须在内存空间中相邻才能造成可控的堆溢出，因此，判断 chunk1 是否溢出的条件就是判断 chunk1 的数据是否覆盖了 chunk2 的内存空间。

判断条件中 chunk1.size 指的是，在对 chunk1 进行赋值操作时（如 memcpy，memmove 等），chunk1 所接收数据的大小，而非对 chunk1 进行分配操作（如 malloc 等）时所确定的 chunk1 的大小。

由于对堆块溢出判断是在对 chunk1 进行赋值操作的同时进行的，此时的 chunk1 必须处于 allocated 状态。在 chunk2 数据部分的起始地址还有 8 字节的头部信息。因此，在进行溢出条件判断时，需要 chunk2.addr - 8。

（2）根据 glibc 内存管理机制，在对程序进行 unlink 漏洞判断的时候，必须确保有溢出的堆块大小大于 0x40。否则，堆块在被释放的过程中不会进行 unlink 操作。

4.3 unlink操作触发特征判断

在对程序堆内存是否发生溢出进行判断的基础上，还需对溢出堆块是否发生 unlink 操作进行检测，才能确认 unlink 漏洞发生。算法 4 实现了对程序 unlink 操作触发的检测。

算法 4：unlink 操作触发检测算法

输入：chunk1（溢出堆块 chunk1 的信息）
　　chunk2（被释放堆块 chunk2 的信息）
输出：isUnlink（程序是否发生 unlink 操作）

```
1  if( chunk1.add < chunk2.add )//程序发生向前合并
2      c1 = forward;
3  if( chunk1.add > chunk2.add )//程序发生向后合并
4      c1 = backward;
5  if( c1 == forward )
6      c2 = ( chunk1.add + chunk1.len == chunk2.add ) ∧ chunk1.free ∧ chunk2.allocated ;
7      fake_header = chunk1.prevsize + chunk1.size + chunk1.fd + chunk1.bk +chunk1.fd_next +
                     chunk1.bk_next;
8      over_data = chunk1.size + chunk2.size;
9  if( c1 == backward )
```

```
10      c3 = ( chunk2.add + chunk2.len == chunk1.add ) ∧ chunk1.free ∧ chunk2.allocated ;
11      fake_header = chunk1.fd + chunk1.bk +chunk1.fd_next + chunk1.bk_next;
12      over_data = chunk1.size + chunk1->nextchunk.size;
13 isUnlink = c2 ∨ c3;
14 return isUnlink ;
```

在进行 unlink 操作过程中，堆块会发生合并。合并操作触发的前提是两个堆块在内存中相邻，且同处于被释放状态。需要注意的是，算法 4 中对溢出堆块的使用状态进行判断时，溢出堆块 chunk1 不一定处于实际上的释放状态。通过在未释放堆块 chunk1 中构造伪堆块 fake chunk 的数据，可使被释放堆块 chunk2 误认为与其相邻 fake chunk 是一个处于释放状态的堆块。在此情况下，程序同样会发生 unlink 操作。

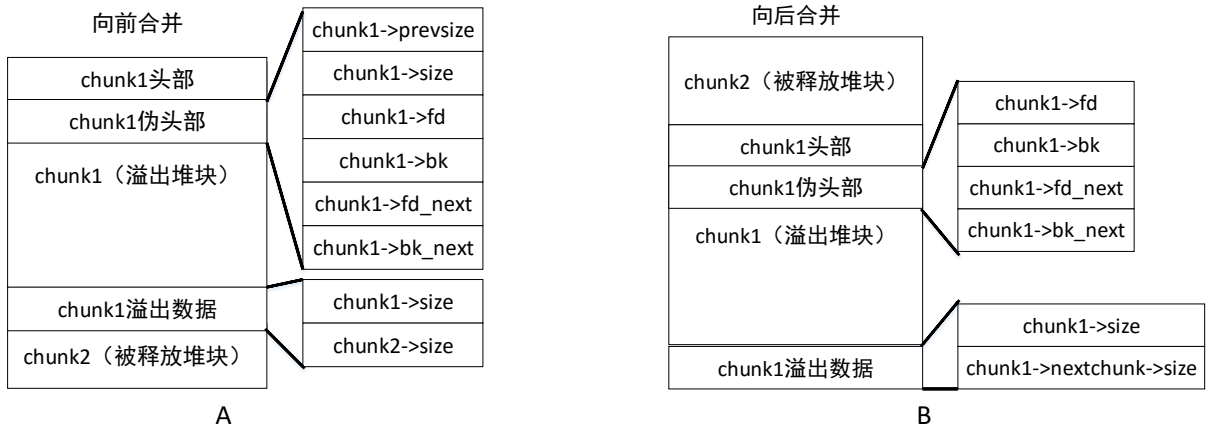

Fig.9　Structure of overflow-chunk and freed-chunk

图 9　溢出堆块与被释放堆块结构示意图

判定系统对被测程序的 free 函数进行监视。当程序调用 free 函数时，对被释放堆块 chunk2 和溢出堆块 chunk1 的信息进行匹配与判断，确定程序是否发生 unlink 操作，以及在 unlink 操作中将进行向前还是向后合并。

图 9 显示的是向前合并及向后合并时，溢出堆块 chunk1 与被释放堆块 chunk2 的结构示意图。根据两种不同的合并形式，unlink 操作触发检测算法在检测得到相应类型的合并操作后，系统将自动构造溢出堆块 chunk1 的伪头部数据 fake_header 与溢出数据 over_data。

4.4 溢出堆块可控性判断

在对溢出堆块 chunk1 进行可控性判断之前，首先需要确定溢出堆块 chunk 状态。只有在确认 chunk1 处于 allocated 状态时，chunk1 中的数据才是可写入的。

算法 1 中的 exploitConstraint 模块实现了对溢出堆块是否符合利用条件的判断。其过程如算法 5 所示。

算法 5：溢出堆块可控性检测算法

```
输入：chunk1（溢出堆块 chunk1 的信息）
      chunk2（被释放堆块 chunk2 的信息）
      shell（shellcode 字符串）
      state（程序的符号执行路径约束）
输出：isExploitable（堆块 chunk1 是否符合利用条件）
1  c1 = symbolic_check( chunk1 );
2  c2 = (chunk1.allocated == 1);
3  c3 = shell.size < chunk1.size  -  16;
4  chunk_constraint = 1;
5  if(! c1 ∨ c2 ∨ c3)
6      return false;
7  foreach addr ∈ ( chunk1.add , chunk1.add + chunk1.size + 8)
```

```
8       while  addr < (chunk1.add + 16)  do    //构建 chunk1 伪头部数据约束
9          fake_header_constraint = buildFakeHeader( fake_header );
10         chunk_constraint = chunk_constraint ∧ fake_header_constraint;
11      while  addr < ( chunk1.add + chunk1.size  -  shell.size )  do  //构建 chunk1 空指令约束
12         nop_constraint = buildNopSled( 0x90 );
13         chunk_constraint = chunk_constraint ∧ nop_constraint;
14      while  addr < (chunk1.add + chunk1.size)  do    //构建 chunk1 中的 shellcode 约束
15         shell_constraint = injectShellcode( shell );
16         chunk_constraint = chunk_constraint ∧ shell_constraint;
17      while addr < (chunk1.add + chunk1.size + 8) do    //构建 chunk1 溢出数据约束
18          over_data_constraint = buildOverData( over_data );
19          chunk_constraint = chunk_constraint ∧ over_data_constraint;
20  isExploitable = STP_solver->maybetrue( chunk_constraint );    //对堆块可控性约束进行求解
20  if(isExploitable)
21      testcase1 = generateTestcase(state, chunk_constraint);    //生成测试例 1
22  return isExploitable;
```

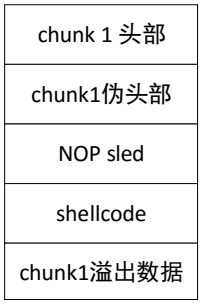

Fig.10　Structure of overflow chunk which can be controlled by input data

图 10　溢出堆块在输入可控数据后的结构示意图

对溢出堆块进行利用的基本形式是，向内容可控的溢出堆块写入数据，伪造一个已释放堆块。伪造堆块的头部信息包含在溢出堆块的数据部分中。另一方面，为了尽可能的降低利用难度，还可将准备注入内存的 shellcode 也写入溢出堆块 chunk1 中。为了在后面的步骤中，尽可能地使程序跳转至可控内存空间中，本文在设计溢出堆块的写入信息时，在伪堆块头部和 shellcode 之间加入了空指令段（NOP sled）。因此，溢出堆块 chunk1 在输入可控数据后的结构如图 10 所示。

为了确保溢出堆块 chunk1 中数据部分的内容可控，首先需要对 chunk1 的数据部分进行符号化检查。在保证 chunk1 的数据是可控的情况下，再检查 chunk1 的大小是否足够容纳准备注入的 shellcode。根据高危 unlink 漏洞模型，由于 chunk1 的数据部分必须留出 16 字节的空间构造伪堆块头部，因此 shellcode 的长度不能超过 chunk1 长度减 16 字节。

在上述两项检查结果的析取式为 1 的情况下，进行对 chunk1 内存空间的可利用约束构造。内存可控性约束的构造过程参照图 9 的结构进行。系统对 chunk1 的堆块可控性约束（chunk_constraint）构造包括四部分：伪堆块头部的约束条件（fake_header_constraints），nop_sled 的约束条件（nop_constraints），shellcode 的约束条件（shell_onstraints）以及溢出数据的约束条件（over_data_constraint）。其关系如式（7）所示。

$$\begin{aligned} chunk\_constraint = {} & fake\_header\_constraint \wedge \\ & nop\_constraint \wedge \\ & shell\_constraint \wedge \\ & over\_data\_constraint \end{aligned} \qquad (7)$$

对 chunk1 数据部分的内存空间以字节为单位进行遍历。遍历过程中，对每个字节的内存构建一个约束表达式。以构建 NOP sled 的约束过程为例，每个字节的约束表达式创建如式（8）所示：

$$klee::create::Eq(0x90,uint8) \quad (8)$$

若输入的 NOP 指令存储于一个名为 buf 的数组中，则上式构造的约束条件如下：

$$(Eq\ 0x90\ (Read\ \ w8\ \ 1\ \ buf))$$
$$(Eq\ 0x90\ (Read\ \ w8\ \ 2\ \ buf))$$
$$\cdots\cdots$$

在对 chunk1 的每个字节迭代构建约束条件的同时，将新构建的约束条件与已有的约束条件 result 进行合取，形成新的内存可控性约束条件。

遍历完 chunk1 的所有内存后，将最终的可控性约束与程序的路径约束一起，交由约束求解器 STP 求解，生成测试用例 1，并得出 chunk1 是否可被特定外部数据控制的结果。

4.5 指针变量可控性判断

根据高危 unlink 漏洞模型，为实现程序控制流劫持，要求外部数据可以实现任意内存写入，这需要对程序中可控的指针变量进行修改。为此，本文提出了指针变量可控性分析算法，判断指定的指针变量* var_ptr 的可控性。过程如算法 6 所示。

算法 6：指针变量可控性分析算法

```
输入：&var_ptr（指针变量*var_ptr 地址）
      chunk1_inf（溢出堆块 chunk1 信息）
输出：isPtrChagable（指针变量*var_ptr 的可控性结果）
1  c1 = symbolic_check( chunk1_inf );
2  if( ! c1 )
2        return false;
4        isPtrChagable = &var_ptr ∈( chunk1_inf.add , chunk1_inf.add + chunk1_inf.size) ∧
                         symbolic_check( &var_ptr );
5  return isPtrChagable;
```

堆块 chunk2 在释放时执行 unlink 操作，使 chunk2 与溢出堆块 chunk1 合并的同时，溢出发生堆块 chunk1 的地址被更改。

由于指针变量* var_ptr 的值为进程中某块内存地址，为使进程内存可控，需对变量* var_ptr 进行可控性判断。

算法 6 首先对堆块 chunk1 进行符号化判断，确定堆块 chunk1 是否可控；若 chunk1 可控，再对变量* var_ptr 进行符号化检查和地址范围的条件判断。当 result 结果为真时，得出结论，可通过外部数据控制堆块 chunk1 更改指针变量* var_ptr 的值，从而实现进程任意内存读写。

5 实验分析

本文选取了 Juliet Test Suite v1.2 测试集中的 CWE122_Heap_Based_Buffer_Overflow__CWE131_memcpy_52a.c 和 CWE122_Heap_Based_Buffer_Overflow__c_CWE805_wchar_t_memcpy_01.c，Shellphish/how2heap 测试集中的 unsafe_unlink.c 以及 Goahead3.0 中的 Goahead.c 程序做为高危 unlink 漏洞判定系统的验证程序。验证程序及实验环境等内容如表 1 所示。

表 1：验证程序及实验环境

程序名称（漏洞代号）	操作系统	Glibc 版本
CWE122_Heap_Based_Buffer_Overflow__CWE131_memmove_31.c	Ubuntu14.04-i386	Glibc-2.19
CWE122_Heap_Based_Buffer_Overflow__c_CWE805_wchar_t_memcpy_01.c	Ubuntu14.04-i386	Glibc-2.19
Shellphish/how2heap/unsafe_unlink.c	Ubuntu14.04-i386	Glibc-2.19
Goahead3.0/ Goahead.c（CVE-2014-9707）	Ubuntu14.04-i386	Glibc-2.19

本文分别对上述四个验证程序进行了四次实验。本文在实验中使用高危 unlink 漏洞判定系统对运行于 Ubuntu14.04 中的四个验证程序进行判定实验，目的是验证该判定系统对不同程序进行判定的正确性。

本文对比高危 unlink 漏洞模型，对验证程序的以下四项特征进行了检测：堆溢出漏洞发生特征，溢出堆块可控性特征，unlink 操作触发特征，指针变量可控性特征。各程序的检测结果如表 2 所示。

表 2：验证程序的高危 unlink 漏洞判定结果

程序名称	堆溢出触发	溢出堆块可控	Unlink 操作触发	指针变量可控	高危 unlink 漏洞
CWE131_memmove_31.c	是	否	否	否	否
CWE805_wchar_t_memcpy_01.c	是	是	否	否	否
unsafe_unlink.c	是	是	是	是	是
Goahead.c	是	是	是	是	是

本文在实验过程中构造了长度为 26 字节，内容为 0x90 的空指令 shellcode，做为输入程序的外部数据的一部分。

5.1 unsafe_unlink.c程序验证实验

通过对 unsafe_unlink.c 程序的检测实验，检验了判定系统对堆块向前合并过程中，触发的高危 unlink 漏洞的检测效果。验证程序 unsafe_unlink.c 的漏洞关键代码如代码 3 所示。

代码 3：unsafe_unlink.c 程序漏洞关键代码

```
1   char *src1, *src2 , *src3;
2   uint *chunk1;
3   char *chunk2;
4   read(src1, 0x88);
5   chunk1 = malloc(0x80);
6   chunk2 = malloc(0x80);
      /*检测 chunk1 堆块溢出与堆块可控性*/
7   memcpy(chunk1, src1, 0x88);
      /*检测 unlink 操作*/
8   free(chunk2);
      /*检测指针变量&chunk1*/
9   strcpy(chunk1, src2);
10   strcpy(chunk1, src3);
```

在实验过程中，判定系统将首先对下述三类 API 函数进行动态挂钩：堆块分配函数、堆块释放函数以及内存数据操作函数。

表 3：unsafe_unlink.c 程序检测信息

分配的堆块地址/长度	0x8f47018/0x80、0x8f470a0/0x80
复制数据后符号化内存地址/长度	0x804a0e0/0x88、0x8f47018/0x88
溢出堆块的堆块地址	0x8f47018
释放堆块的地址	0x8f470a0
释放过程中的 unlink 操作类型	Mode 1（向前合并）
生成测试用例 1 的长度	136 字节
第二次复制数据后符号化内存的地址/长度	0x804a060/0x108、0x804a1f4/0x80
指针变量地址	0x804a200

判定系统的堆块创建监视模块检测到了被测程序的堆块创建过程。通过监视堆块分配函数（如 malloc）的参数及返回值，获取分配堆块的长度及地址等信息。

通过对内存数据操作函数的挂钩，得到复制数据的目的地址、源地址及复制的数据长度等信息；判定系统的数据操作监视模块对复制数据后的内存空间进行搜索，可获得符号化内存的地址及长度。观察表 3 中的分配的堆块地址/长度与复制数据后符号化内存地址/长度，可发现，堆块 0x8f47018 处的符号化长度大于原分配长度，因此可判断堆块 0x8f47018 发生了溢出对堆块 0x8f47018 进行可控性判断，所得结果如表 2 中 unsafe_unlink.c 程序的溢出堆块可控性判断结果所示。

通过对释放函数挂钩可获得被释放堆块的地址。对比被释放函数地址 0x8f470a0 与可控溢出堆块地址 0x8f47018 在内存中的相对位置，发现溢出堆块 0x8f47018 处于被释放堆块 0x8f470a0 的低地址处。由表 2 中 unsafe_unlink.c 程序的溢出堆块可

控性判断可知，溢出堆块 0x8f47018 可控，因此可判断堆块 0x8f470a0 释放时触发向前合并的 unlink 操作。

为使程序成功执行 unlink 操作，判断系统对溢出堆块 0x8f47018 进行具体化处理。系统根据收集的被测程序的路径约束与堆块数据约束，构造了长度为 136 字节的测试用例，并使用例数据覆盖溢出堆块。

在监测 unsafe_unlink.c 程序发生第二次复制数据操作后，再次对符号化内存进行搜索。此时，被测程序的符号化内存地址/长度信息如表 3 所示。其中一块符号化内存的地址/长度为 0x804a1f4/0x80，而通过配置文件所指定的指针变量地址为 0x804a200，处于符号化内存块中。由于被测程序的指针变量 0x804a200 可被外部数据控制，可能导致任意内存操作，因此，系统判定，在 Ubuntu14.04-i386 及 Glibc-2.19 环境下运行的 unsafe_unlink.c 程序中存在高危 unlink 漏洞。

5.2 Goahead.c程序验证实验

验证程序 Goahead.c 的漏洞关键代码如代码 4 所示。

代码 4：Goahead.c 程序漏洞关键代码

```
1    char * arg;
2    read(*arg, 0x210);
3    int len = strlen(arg);
4    char * dupPath = malloc(len);
5    char **segments = malloc(len);
     /*将 dupPath 中的字符串经过处理后分为若干不同的字符串，并将各字符串的地址存储于 segments 中*/
6    int nesg = function(dupPath, segments);
7    char *path = malloc(len);
8    for i:0->nesg
     /*检测 path 堆块溢出与堆块可控性*/
9        strcpy(path, segments[i]);
10   free(dupPath);
     /*检测 unlink 操作与指针变量可控性*/
11   free(segments);
```

本文向该程序输入的外部数据长度为 0x210 字节。根据 glibc 对堆块管理的分类，dupPath，segments 和 path 堆块被标记为 large chunk，glibc-2.19 对 large chunk 的 unlink 操作代码如代码 5 所示。

代码 5：glibc-2.19 对 large chunk 的 unlink 操作过程

```
1    P->fd_nextsize->bk_nextsize = P->bk_nextsize;
2    P->bk_nextsize->fd_nextsize = P->fd_nextsize;
```

表 4：Goahead.c 程序检测信息

分配的堆块地址/长度	0x91742d8/0x210、0x91741e0/0x1e、0x91744f0/0x210
复制数据后符号化内存地址/长度	0xbfc8c36c/0x210、0x91744f0/0x230
溢出堆块的堆块地址	0x91744f0
释放堆块的地址	0x91742d8
释放过程中的 unlink 操作类型	Mode 2（向后合并）
生成测试用例 1 的长度	560 字节
0x91744f0 堆块 P->fd_nextsize 指针地址	0x91744fc
0x91744f0 堆块 P->bk_nextsize 指针地址	0x9174500
堆块释放后符号化内存的地址/长度	0x804a04c/0x4
指针变量地址	0x804a04c

判定系统对 Goahead.c 程序的检测获取信息如表 4 所示。系统首先发现程序分配的三个堆块地址及长度，并在数据复制操作后，搜索内存中的符号区域。通过对比堆块 0x91744f0 的申请长度与相应内存中的符号长度发现，该堆块发生溢出。

当 Goahead.c 程序释放堆块 0x91742d8 后，通过对比溢出堆块 0x91744f0 与释放堆块 0x91742d8 的相对位置，系统判断此处发生 Mode 2 类型（向后合并）的 unlink 操作。此时，判定系统根据配置文件中的相应规则构造堆块数据约束，并与程序路径约束进行合取，得到测试用例约束。通过求解用例约束，生成长度为 560 字节的测试用例 1。

由表 4 可知，堆块 0x91744f0 的 P->fd_nextsize 与 P->bk_nextsize 指针均位于符号区域内，因此，根据代码 7 中的 glibc-2.19 对 large chunk 的 unlink 操作代码以及系统污点传播规则，堆块 0x91744f0 的 P->fd_nextsize->bk_nextsize 与 P->bk_nextsize->fd_nextsize 指针也被设置为符号值。由于配置文件已将 P->bk_nextsize 指针设定为具体值 0x804a03c，因此指针变量 0x804a04c 受到符号值 P->fd_nextsize 的影响，也被标记为符号内存。由此，系统判定指针变量 0x804a04c 可被外部数据控制，该漏洞属于可能导致任意内存操作的高危 unlink 漏洞[15]。

6 总结

本文针对可能导致任意代码执行的 unlink 漏洞提出了高危 unlink 漏洞模型，并在符号执行和污点分析技术的基础上，设计了针对高危 unlink 漏洞的判定系统。该系统分别对程序的堆溢出触发特征，unlink 操作特征，溢出堆块可控性特征以及指针变量可控性特征进行检测分析，判断 unlink 漏洞是否符合高危 unlink 漏洞模型，从而为防护高危 unlink 漏洞提供借鉴。实验选取四个堆溢出漏洞程序对高危 unlink 漏洞判定系统进行了验证。实验证明了该系统对相关程序判定结果的正确性。

另一方面，本文介绍的高危 unlink 漏洞判定系统未考虑数据执行保护，地址随机化等机制对漏洞程序的影响，因此，系统对高危 unlink 漏洞的分析验证仍存在不足。如何实现在地址随机化等机制的环境下实现高危 unlink 漏洞检测及测试例生成，是本文的下一步工作。

References:

[1] Chipounov V, Kuznetsov V, Candea G. S2E:a platform for in-vivo multi-path analysis of software systems[J]. Acm Sigplan Notices, 2011, 47(4):265-278.

[2] Chipounov V, Kuznetsov V, Candea G. The S2E Platform: Design, Implementation, and Applications[M]. ACM, 2012.

[3] Avgerinos T, Sang K C, Hao B L T, et al. AEG: Automatic Exploit Generation[C]// Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, Usa, February -, February. DBLP, 2011.

[4] Huang S K, Huang M H, Huang P Y, et al. CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations[C]// IEEE Sixth International Conference on Software Security and Reliability. IEEE Computer Society, 2012:78-87.

[5]Wang MH, Ying LY, Feng DG. Exploit detection based on illegal control flow transfers identification[J]. Journal on Communications, 2014, 35(9):20-31.

[6]Ali Mobile Security. Analysis of Linux heap memory management（1）. http:// www.cnblogs.com/alisecurity/p/5486458.html

[7] Ali Mobile Security. Analysis of Linux heap memory management（2）. http:// www.cnblogs.com/alisecurity/p/5520847.html

[8] Ali Mobile Security. Linux heap overflow exploits：unlink. http:// www.cnblogs.com/alisecurity/p/5563819.html

[9] Shao Z, Cao J, Chan K C C, et al. Hardware/software optimization for array & pointer boundary checking against buffer overflow attacks[J]. Journal of Parallel & Distributed Computing, 2015, 66(9):1129-1136.

[10] Zhang C. Research on Software Security Defense against Control-Flow Hijacking Attacks[D]. Peking University, 2013.

[11] Ali Mobile Security. Learning of unlink in Android. http:// manyface.github.io/2016/05/19/AndroidHeapUnlinkExploitPractice/.

[12] Shi YF, Fu DS. Research of Buffer Overflow Vulnerability Discovering Analysis and Exploiting [J]. Computer Science, 2013, 40(11):143-146.

[13] Xiao QX, Chen Y, Qi LL,etc. Detection and analysis of size controlled heap allocation [J]. Journal of Tsinghua University (Science and Technology), 2015(5):572-578.

[14] Huang H, Lu YL, Liu LT,etc. A research on Control-flow taint information directed symbolic execution [J]. Journal of University of Science and Technology of China, 2016(1):21-27.

[15]CVE. CVE-2014-9707. http://cve.scap.org.cn/cve-2014-9707. html

附中文参考文献:

[5]王明华, 应凌云, 冯登国. 基于异常控制流识别的漏洞利用攻击检测方法[J]. 通信学报, 2014, 35(9):20-31.

[6]阿里移动安全. Linux 堆内存管理深入分析（一）. http:// www.cnblogs.com/alisecurity/p/5486458.html

[7]阿里移动安全. Linux 堆内存管理深入分析（二）. http:// www.cnblogs.com/alisecurity/p/5520847.html

[8]阿里移动安全. Linux 堆溢出漏洞利用之 unlink. http:// www.cnblogs.com/alisecurity/p/5563819.html

[10]张超. 针对控制流劫持攻击的软件安全防护技术研究[D]. 北京大学, 2013.

[11]阿里移动安全. Android 中堆 unlink 利用学习. http:// manyface.github.io/2016/05/19/AndroidHeapUnlinkExploitPractice/.

[12]史飞悦, 傅德胜. 缓冲区溢出漏洞挖掘分析及利用的研究[J]. 计算机科学, 2013, 40(11):143-146.

[13]肖奇学, 陈渝, 戚兰兰,等. 堆分配大小可控的检测与分析[J]. 清华大学学报(自然科学版), 2015(5):572-578.

[14]黄晖, 陆余良, 刘林涛,等. 控制流污点信息导向的符号执行技术研究[J]. 中国科学技术大学学报, 2016(1):21-27.