缓冲区溢出脆弱性检测和预防技术综述*)

徐良华[1,2] 陈左宁[1,2]

(信息工程大学信息工程学院 郑州 450002)[1] (江南计算技术研究所 无锡 214083)[2]

摘 要 利用缓冲区溢出脆弱性进行攻击,是网络攻击中最常见和最危险的攻击方法。为解决缓冲区溢出脆弱性问题,在研究和商业领域提出了各种各样的方案。本文首先将缓冲区溢出脆弱性检测和预防技术划分成9大类;然后研究了每一类技术的原理、特性、适用范围和优缺点等;最后分析讨论了整个缓冲区溢出脆弱性检测和预防技术。

关键词 缓冲区溢出,脆弱性,缓冲区溢出脆弱性,检测,预防

A Survey on Detection and Prevention Technologies of Buffer Overflow Vulnerability

XU Liang-Hua[1,2] CHEN Zuo-Ning[2]

(Institute of Information Engineering, Information Engineering University, Zhengzhou 450002)[1]

(Jiangnan Institute of Computing Technology, Wuxi 214083)[2]

Abstract Buffer overflow attack is the most common and the most dangerous attack method used in network attacks. Various solutions have been developed to address the buffer overflow vulnerability problem in both research and commercial communities. First, detection and prevention technologies of buffer overflow vulnerability are classified as nine groups. Then, each group's rationale, characteristics, applicable field, merits and limits are researched. Finally, the whole detection and prevention technologies of buffer overflow vulnerability are analyzed and discussed.

Keywords Buffer overflow, Vulnerability, Buffer overflow vulnerability, Detection, Prevention

1 引言

攻击者能够利用程序中的脆弱性获得该程序的控制权,对运行该程序的系统形成安全威胁。1988年的莫里斯蠕虫是第一次利用缓冲区溢出脆弱性的攻击。从那以后,利用缓冲区溢出脆弱性的攻击愈来愈多,现在已经成为最常见也是最危险的网络攻击。

缓冲区溢出是由于程序或编译器没有对缓冲区进行边界检查导致的。发展C语言的初衷是为了编写操作系统,后来被广泛使用,其运行高效、精确控制资源等优点成为系统程序员最喜欢选择的编程语言,很多商业软件都是使用C语言编写的。但C语言缺少安全机制,对数组和指针不做边界检查,由此带来很多缓冲区溢出隐患。最直接的预防方法是编程时严格按照规则对边界进行检查,但实际效果不尽如人意,即使是经过严格培训的程序员也会产生错误。为了防止缓冲区溢出,专家学者研究了很多的检测和预防技术。本文试图对这些技术的原理、特性、适用范围和优缺点等进行一次综述。

2 检测和预防技术

2.1 基于源码的静态分析

基于源码的词法分析、模型化、标记驱动、符号分析等静态分析技术能在程序发布前检测和修正缓冲区溢出脆弱性。

2.1.1 词法分析

词法分析工具ITS4使用潜在危险结构数据库来检测安全问题[1]。很多程序员在进行安全审计时使用grep命令查找危险的函数调用。ITS4的设计者的目的是提供比grep更智能的工具,在找到危险的函数调用时,ITS4还能提供有用的安全建议。类似ITS4的词法分析工具还有Cqual,Flex-eLint,Flawfinder,RATS和Pscan等。词法分析的好处是快捷简单,但没有考虑程序的语义,忽略了过程之间的交互、变量值和控制流。

2.1.2 模型化

可以将缓冲区溢出检测形式化为整数限制求解问题[2]。C串被当作库函数要访问的一个抽象数据类型,缓冲区被模型化为一对表示大小和当前长度的整数,库函数被模型化成如何修改串的大小和长度。使用图论技术构造有效的算法,求解每个程序变量允许值的范围。BOON就是该技术的一个原型系统。缺点是无法检测非字符串的缓冲区溢出。

2.1.3 标记驱动

SPLint[3]要求程序员在C源码中按照固定的注释格式/*@……@*/加入标记,这些标记被当作规格属性,然后利用词法分析和过程内流敏感程序分析技术。根据标记在每个函数中增加事先条件和事后条件(事先条件是指函数参数必须满足指定的限制,事后条件保证函数结果满足一些限制要求),并验证标记的限制条件与程序是否一致。Lclint[4]在程序中加入语义注释,对循环使用启发方式。标记驱动技术的缺点是不能很好地处理指针的运算。

2.1.4 符号分析

ARCHER数组检查器利用路径敏感、上下文有关和过程间的数据流分析技术来限制变量和内存的边界[5]。ARCHER首先使用修改过的C编译器将源码解析成抽象语法树

*)本文研究得到国家"八六三"高技术研究发展计划项目资助(2003AA146010)。**徐良华** 高工,博士研究生,研究方向为网络安全;**陈左宁** 院士,教授,博士生导师,主要研究方向为高性能计算机、操作系统和网络攻防。

AST,然后将 AST 转换为范式表示,并据此构造控制流图 CFG 和函数调用关系图。最后,根据 CFG 和函数调用关系图,自底向上,对每个函数的 CFG 使用随机的、深度优先的遍历,尽可能全面地进行检查。优点是速度快,缺点是不能对字符串操作进行分析。

2.2 基于执行码的分析转换

在不能得到源码的情况下,最直接的办法就是对可执行码进行分析、转换,来检测和预防缓冲区溢出。

2.2.1 定位函数边界

通过反汇编器线性扫描和递归遍历可以对可执行码定位函数边界[6]。线性扫描从代码段的第一个字节开始逐个字节分析,直到遇到非法指令为止;递归遍历从 main 函数开始,分析每个控制流,按深度优先或广度优先遍历所有的分支。为了不破坏函数内部的引用结构,插入的安全检查代码存放在原函数代码后面一个新的独立的只读代码段中。在原函数代码段中,头几个字节被移到安全检查代码中,代之以指到安全检查代码的 JMP 指令。在函数调用时,保留一份返回地址的副本,在函数返回前将堆栈中的返回地址与保留的副本进行比较。缺点是函数范围难以精确判定,且不能保护动态加入的代码。

2.2.2 保护返回地址

RAD 是最早不需要源码、直接利用静态二进制码转换来保护程序不受缓冲区溢出攻击的技术[7]。在函数调用产生新堆栈时,将函数返回地址副本保存到返回地址仓库 RAR 中,在函数返回时检查副本和返回地址的一致性。RAD 对 RAR 提供了两种保护方法:一种是将 RAR 前后的段设置成只读的;另一种是在函数返回地址副本存放到 RAR 后,立即用系统调用来保证该副本的只读属性。前者的缺点是攻击者能够通过破坏指针来改写 RAR 中的副本;后者的缺点是对性能影响很大。RAD 的优点是不改变堆栈的布局,保证了程序二进制兼容性;缺点是不能预防基于破坏内存指针的缓冲区溢出攻击。

2.2.3 检测系统调用

利用缓冲区溢出脆弱性的攻击代码往往会使用系统调用访问文件、套节字等系统资源[8]。首先对可执行码进行预处理,定位所有系统调用;然后将系统调用后面一条指令的地址存放到堆栈中作为返回地址,这些地址副本被保存到一个列表中;最后在在程序运行时对程序进行监控,在系统调用结束时检查堆栈中的返回地址与保存在列表中的副本是否一致。如果不一致,就将有关状态记入日志并中止程序。该方法不适用于攻击代码没有使用系统调用或者系统调用被另一个函数包装起来的情况。

2.2.4 反汇编

sprintf()等隐含安全风险的函数在堆栈上建立缓冲区时,往往有特定的汇编指令操作序列[9]。对可执行码反汇编,在堆栈上建立缓冲区时进行检查,如果操作序列不正常,就预示着可能存在缓冲区溢出。缺点是对在堆栈上建立缓冲区时没有上述特征操作序列时不适用。

2.3 扩展编译器功能

编译器是源码变成可执行码的桥梁。有很多缓冲区溢出脆弱性检测和预防技术解决方案是通过扩展原编译器,增加缓冲区边界信息并插入边界检查代码来实现的。

2.3.1 增加返回地址保护功能

Stackguard[10]对编译器进行扩展,调用函数时在堆栈的局部变量和返回地址之间存放一个程序运行时随机产生的 4 个字节的"canary"字,在函数返回前检查"canary"的完整性。如果"canary"被破坏,则表明有缓冲区溢出产生,程序被中止。缺点是需要源码;不能保护函数参数、局部变量、栈指针;通过破坏堆栈中旧帧指针或局部指针变量可以绕过安全机制。

Pointguard[11]对 Stackguard 进行了扩展,将"canary"置于所有的代码指针后面和所有受保护变量的前后,在函数调用和对受保护变量访问时都会检查"canary"值。PointGuard 可以预防写缓冲区攻击,却不能预防读缓冲区攻击。通过对函数指针在内存中存放时加密、在引用时解密,可以解决这个问题[12]。

ProPolice[13],核心技术源于 StackGuard,但重新布局了堆栈中的局部变量,让字符型缓冲区分配到挨着旧基指针的栈底,使得该缓冲区溢出时不会破坏到其它局部变量。与 Stackguard 在编译时将汇编码加入到可执行文件不同,ProPolice 修改寄存器转换语言 RTL 来加入安全机制,从而具有与平台无关的特性。

2.3.2 扩展指针表示

对原指针进行扩展,增加"对象属性",从而使指针变成安全指针[14]。该思路来源于 Lisp 语言的标记指针。对象属性包括该指针引用的位置、大小和生存期。首先将所有定义和说明的指针转换为具有对象属性的安全指针;然后插入生存期和边界检查。有两种扩展指针的方法:一种是将指针变成记录[14],其缺点是与标准 C 不兼容;一种是将扩展属性存放在一张专门的标准,而指针指向表中的表项[15]。

2.3.3 插入内存修改日志

DIRA[16]是 GCC 编译器的扩展。在编译时插入适当的内存修改日志代码,然后在所有对敏感数据引用的地方插入检查代码,最后在程序中插入识别攻击代码和修复程序的函数。如果在日志中发现敏感数据结构被破坏,就说明发生了攻击。这时就报警,并调用修复程序函数,对程序进行修复。

2.3.4 增加安全类型

Cyclone 在保留 C 语言的句法和语义的前提下对 C 进行改造,增加了安全类型,并将静态检测和运行时插入检查很好地结合在一起[17]。为避免悬空指针错误,Cyclone 对代码进行区域分析。一个区域是指在一次释放中被释放的内存段,例如一个函数退出时所释放的局部变量在同一个区域。Cyclone 实施监测程序的在用区域,不允许指针指向已释放的区域。Cyclong 使用可增长区域技术对内存进行更严格的安全管理。Cyclone 改变了指针表示,从而与 C 不兼容。

CCured[18]首先将 C 源码转换成类型安全的 C 中间语言 CIL,然后插入运行时检查代码。在 CIL 中增加了 SAFE、SEQ 和 DYNAMIC 3 种指针类型:SAFE 是标准 C 类型,但不能进行指针算术运算,所以不能用来访问数组;SEQ 类型可用来访问数组,包含数组边界,在运行时可做边界检查;DYNAMIC 类型指向堆。CCured 首先通过指针类型限制符来静态验证是否存在缓冲区溢出,然后在运行时任何对 SEQ 和 DYNAMIC 类指针的访问都要进行边界检查。缺点是通过无用内存收集器来释放内存而不支持人工释放,导致某些软件不能运行。

2.4 运行时拦截并检查

运行时拦截危险函数并进行安全检查是对原系统影响较小的软件实现方法。

2.4.1 拦截脆弱函数

Libsafe[19]重新编写C库中不安全的函数，先于标准C库安装。在运行时首先拦截所有对具有缓冲区溢出风险的库函数的调用，然后调用具有边界检查功能的安全函数来完成原功能。如果边界检查不通过，就报警并终止进程。Libverify是Libsafe的增强版，对所有函数都做安全检查。优点是性能高；对系统影响小；不需要重新编译源码。缺点是不能保护局部指针变量；对静态连接库无效；不能预防基于堆或BSS数据段的攻击；Libsafe依赖编译器产生嵌入式的帧指针定位函数返回地址，因此编译时不能带有-fomit-frame-pointer选项。

2.4.2 运行时预防堆溢出

通过修改C库、内存块结构和内存管理子程序来检测和预防堆溢出[20]。在内存块数据结构中增加一个canary标志，存放某些敏感数据与密钥种子的检查和。在内存分配和回收子程序中增加检测代码，在分配和回收内存块时对canary值进行检查。程序不需要重新编译，只要程序使用的是动态链接的C库，就能检测和预防堆缓冲区溢出。

2.5 使堆栈和堆不可执行

大部分基于堆栈缓冲区溢出脆弱性的攻击依赖于堆栈可执行。如果不允许堆栈执行程序，就能防御这类攻击。Linux内核补丁PaX，就可以使堆栈和堆不可执行[21]。对支持标注页为不可执行的体系结构可以直接应用，而对IA32结构则需要软件来模拟。在IA32体系结构中，利用页表入口PTE、数据翻译后备缓冲区DTLB和指令翻译后备缓冲区ITLB的内存页面管理机制，在DTLB和ITLB中增加执行、读和写权限，在内存页面更新时进行权限检查，实现不让堆栈和堆中数据执行的目的。优点是对性能影响小。缺点是需要重新编译内核；用已经加载到内存中的另一个函数的地址覆盖下一条指令地址的return-into-libc技巧可以旁路安全保护机制；递归函数需要利用可执行堆栈，需要利用堆栈或堆执行程序的应用软件不能正常运行；需要修改操作系统核心，以对原利用可执行堆栈属性进行信号处理和运行时代码生成功能进行支持。

2.6 抽象执行网络数据

所谓抽象执行是指在检查网络数据时分析它是否表示了有效的机器指令，通过抽象执行可以检测网络数据中是否包含带有缓冲区溢出脆弱性的程序[22]。数据请求包如果包含了缓冲区溢出脆弱性，那它的有效指令链会比正常的数据请求包中的有效指令链长很多，对可能包含缓冲区溢出脆弱性的数据包再在虚拟机上模拟执行，以进一步验证。

2.7 软件测试

使用一些软件测试技术也能检测缓冲区溢出脆弱性，其主要缺点是需要测试者提供触发脆弱性发生的测试数据。

2.7.1 基于错误注入的测试

错误注入检测有些类似变异测试。对被测试程序进行语法改变，然后观察这样的改变是否导致了与系统的安全政策的冲突。Ghosh等使用该技术进行了检测缓冲区溢出脆弱性的研究[23]。基于错误注入技术的环境扰动法可以测试软件的脆弱性[24]。测试时，向系统注入错误的环境，然后观测系统的行为。如果系统表现不正常，就说明可能存在安全缺陷。

2.7.2 基于属性的测试

软件测试分析的目标就是验证软件执行是否满足规格说明书[25]。分析者使用低级规格书描述语言TASPEC说明目标属性，然后对程序进行切片[26]，切去与目标属性无关的程序。测试系统自动地将TASPEC规格书转化为一个测试精灵(Oracle)来验证程序在指定属性下能否正确执行。在规格书中说明“没有缓冲区溢出脆弱性”的属性就可以检查程序中的缓冲区溢出脆弱性。

2.7.3 渗透测试

许多安全测试工具使用渗透测试法，利用众所周知的脆弱性对安装的系统进行入侵来评估安全性。Pfleeger[27]描述了渗透测试的过程。渗透测试不容易选择合适的渗透实例；没有一个很好的标准能用来决定什么时候可以停止渗透测试；很多系统的脆弱性与该系统的配置环境有关系。

2.8 打乱和加密

该技术使得攻击代码即使注入，也不能运行。

2.8.1 打乱内存地址

很多攻击都基于内存分配的规律，一般都需要知道攻击代码在内存的什么地方或者要覆盖内存的什么地方。改变传统的内存分配算法，在程序加载时，将内存布局打乱，从而有效地防御利用缓冲区溢出等脆弱性的攻击[28]。主要的变化有：随机化堆栈、堆、动态链接库、函数等存储区基址；随机化变量和函数的顺序；在堆栈、变量、静态数据结构、代码段等后面附加随机长度的空隙等。优点是防御范围广，对基于堆栈、堆、整数溢出、格式串溢出都有效；转换快，负载小。缺点是减少了可用内存空间。

2.8.2 打乱系统调用等入口

基于堆栈的缓冲区溢出攻击一般需要了解系统调用映射或库的入口点，通过随机化堆栈起点、随机化系统调用映射和改变库的入口点可以防御缓冲区溢出攻击[29]。随机化系统调用的方法是使用不同的系统调用映射来重编译操作系统内核，然后重写系统调用以静态方式与合法的源码链接。为了进一步增加攻击者的难度，还可以将系统调用表的空间增加一倍。缺点是攻击者只要花时间，就能分析清楚随机化后的系统调用映射关系。

2.8.3 对段设置保护属性

将指针和控制流信息与缓冲区等数据分区存放，在不同的区之间设置一个不可写的页，使缓冲区溢出不会影响到相邻的区[30]。对IA32体系结构，对内存中的进程布局做如下修改：修改堆栈布局，将原来的一个堆栈扩展到三个堆栈，第一个堆栈存放返回地址，第二个堆栈存放帧指针、局部指针、数组指针等控制信息，第三个堆栈存放通常的数据；内存动态分配时，在堆的开始处增加存放管理信息的哈希表；数据段重新布局，ctors和dtors段在最前，紧跟全局偏移表GOT、例外处理帧、规则数据、指针数组，最后是通常的数组。缺点是对利用格式串溢出写任意地址的攻击无效。

2.8.4 加密可执行代码

将可执行文件加密，在程序运行前解密[31,32]。如果运行到攻击代码，因为攻击代码事先没有加密，故对其解密后必然不能正确运行。文[31]将密钥存放在文件头里，如果攻击者能获得该密钥，就能破解该安全措施。文[32]是在可执行程序加载到内存时加密。与文[31]比较起来，它的好处是每次密钥不同；密钥不存放在文件里。两者共同的缺点是需要硬件和操作系统的支持，处理器需要一个寄存器来存放加解密的密钥；因为动态链接库在内存中只驻留一个副本，所以需要专门研究如何支持动态链接。

2.9 基于硬件的支持

硬件技术是最底层的技术，如果能用于检测和预防缓冲

区溢出脆弱性，将是一个性能最高、解决最彻底的方案。

2.9.1 增加硬件堆栈

将原堆栈中的返回地址另外保留一份到硬件管理的堆栈。当代码返回时，检查原堆栈中的返回地址与保留在硬件管理堆栈中的副本是否一致，通过对硬件和操作系统的增强可以检测和预防缓冲区溢出[33]。该技术要保证原堆栈中的返回地址与保留在硬件管理堆栈中的返回地址同步，对可以弹出堆栈的 setjmp/longjmp 等情况要特别处理。为克服这个困难，另一个技术是不改变堆栈结构，而是将返回地址加密存储[12]。基于硬件的技术具有安全、对性能影响小、无需源程序重编译等优点，但操作系统需要做相应的修改以支持额外的硬件堆栈。

2.9.2 使用两个堆栈

基于堆栈的缓冲区溢出依靠在局部变量中故意存放过多数据来覆盖返回地址，而将返回地址存储在不同的堆栈里，使得即使缓冲区溢出了，也不会覆盖返回地址。为实现两个堆栈，可以扩展编译器来模拟，也可以通过修改硬件体系结构来实现。前者增加的负载大，后者负载小。后者在原堆栈的基础上专门再增加一个能预测目标返回指令的安全返回地址堆栈 SRAS，通过比较 SRAS 和原堆栈中的返回地址来检测和预防基于堆栈的缓冲区溢出。SRAS 需要修改处理器的分支预测结构，还需要操作系统和编译器进行相应修改，但对应用软件是透明的[34]。

2.9.3 地址保护

硬件、软件地址保护 HSAP 由“堆栈保护”和“函数指针异或”两部分组成。前者防止基于堆栈的缓冲区溢出，后者防止基于函数指针的缓冲区溢出[35]。“堆栈保护”使用当前帧指针的值进行硬件边界检查，来防止基于堆栈的缓冲区溢出。在写操作前增加“地址检查”，当目标地址大于等于帧指针值时拒绝写操作。优点是保护了帧指针、返回地址和参数；对性能影响小；不需要源码，也不增加保护代码。“函数指针异或”在进程生成时随机产生一个密钥，并保存到一个特别的寄存器 R；增加一个新的安全跳转指令 SJMP，SJMP 的功能是将输入地址与 R 异或后跳转到该地址。为了实现函数指针保护，程序员需要按照 SJMP 的功能修改源码。因为增加了额外的寄存器、SJMP 和 XOR 加密等，导致不兼容原来的程序。

2.9.4 地址完整性检查

如果能检测和保护地址的完整性，就能检测和预防缓冲区溢出[36]。内存的每个字都对应一个安全比特位，该安全比特位起到了完整性签名的作用。该方案既能保护函数的返回地址，也能保护函数指针。使用硬件的方法增加安全比特位，在对原系统性能影响很小的同时，又能强制进行完整性保护，安全性更高。

3 分析讨论

缓冲区溢出脆弱性检测和预防技术可以分成静态和动态分析技术两大类。

静态分析技术通过分析代码而不执行代码来发现程序中的脆弱性。静态分析技术通常采用预先定义的样式检测可能的缓冲区溢出，一般只能检测已知的脆弱性。常用的静态分析方法有词法分析、模型化、标记驱动、符号分析以及在编译器中增加安全检查功能等。静态分析的好处是检测速度快，容易建立突出安全属性的程序模型，有一定的数学基础，可以在程序发布前去除脆弱性。有些程序（如操作系统的设备驱动程序），必须在特定的配置中运行，如果系统中增加了动态分析工具，它就不能运行了，这时只能使用静态分析技术。但由于静态分析没有运行时信息，因此会遗漏脆弱性，同时也会产生假警告，需要仔细地检查和验证警告信息。

动态分析技术在程序运行的过程中发现和预防脆弱性。优点是不需要源码，源码有时很难获取，如第三方不愿意提供；程序是库的形式；程序是以汇编码甚至机器码直接编写的。缺点是速度慢，有的脆弱性可能很难检测到，因为动态分析很难模拟变化几乎是无限多的数据和环境。

本文在第 2 节中介绍和分析了 9 种主流的缓冲区溢出脆弱性检测和预防技术，每种技术都解决了一定的问题，也存在相应的限制。如不可执行内存区域的技术不能预防攻击代码已经事先注入内存和利用共享库的攻击，也不能预防基于函数指针的缓冲区溢出攻击。使用编译器生成或事先产生密钥打乱地址或程序的技术不能支持共享库，保护密钥和密钥生成算法也是一个难题，利用信息泄漏攻击、暴力攻击也许能获得密钥；利用逆向工程技术也许能获得密钥生成算法。StackGuard 等保护堆栈类技术和安全库技术的共同缺点是不能检测和预防堆/BSS/data 和 longjmp 缓冲区溢出；对函数的递归深度有限制。保留返回地址副本和使用分离堆栈的技术不兼容跳板（trampolines）技术，没有后进先出流控制，和一些动态内存管理也不兼容，也不能保护基于函数指针的攻击。

对数组索引进行边界检查，能有效地检测和预防缓冲区溢出。Java 等语言正是因为进行了数组索引边界检查，从而能有效地抵御缓冲区溢出攻击。但在 C 语言中，因为数组和指针是分离的，使得无法进行全面的边界检查。在 C 语言中对编译器进行扩展进行的边界检查，只能检查 buffer[i]类的索引操作，而不能检查 buffer+i 类的表达式。因为边界信息的管理复杂，每次对数组访问时都进行边界检查，所以对性能有很大的影响。性能有时会有十几倍甚至几十倍的下降，因此在商用化的软件中很少采用边界检测，多在调试程序时使用。边界检查最大的优点是不会被攻击者旁路，其它的许多技术都存在被绕过的可能。对数组和指针的边界检查技术分两种，一种采用扩展数组和指针结构，该技术存在程序兼容性问题；另外一种不改变结构，只插入边界检查代码，避免了兼容性问题。

很多技术和工具在检测到缓冲区溢出时都中止进程，使得系统不能正常运行，不能防止拒绝服务攻击。为避免这样的缺点，文[37]提出了类似数据库管理系统中的事务的概念，将一个函数当作一个事务，在事务失败时可以撤销事务，而不影响系统的正常运行。

为了对抗缓冲区溢出脆弱性，专家学者研究了很多的检测和预防技术。检测技术的目标是检查是否存在缓冲区溢出。基于源码的检测能预防缓冲区溢出的发生；基于运行时的检测在发现缓冲区溢出后，通常是报警和/或终止进程，因此不能预防拒绝服务攻击，但能验证软件中是否包含缓冲区溢出脆弱性，从而有利于程序员改进软件。预防技术的目标是隐藏缓冲区溢出脆弱性或使缓冲区溢出只有部分危害甚至全部无害。预防虽然不能定位缓冲区溢出的起因、位置等信息，却能避免拒绝服务或系统被渗透。

结束语 完成基于缓冲区溢出脆弱性的攻击需要注入攻击代码、将控制转到攻击代码和执行攻击代码三个步骤。检测和预防技术就是检测或预防其中任何一个步骤的发生或可能发生。每种技术都有其不足之处，根本解决问题的方案是

开发出全系列的能预防缓冲区溢出的操作系统、库、编译器和编程语言。

参 考 文 献

1 Viega J, Bloch J T, Kohno T, et al. ITS4: A Static Vulnerability Scanner for C and C++ Code. ACM Transactions on Information and System Security, 2002, 5(2)

2 Wagner D, Foster J S, Brewer E A, et al. A first step towards automated detection of buffer overrun vulnerabilities. In: Proc. of Network and Distributed System Security Symposium, San Diego, California, 2000

3 Larochelle D, Evans D. Statically Detecting Likely Buffer Overflow Vulnerabilities. USENIX Security Symposium, Washington D C, 2001

4 Evans D, Guttag J, Horning J, et al. LCLint: A Tool for Using Specifications to Check Code. Software Engineering Notes, 1994, 19(5):87～96

5 Xie Yichen, Chou Andy, Engler D. ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors. Software Engineering Notes, 2003, 28(5):327～336

6 Prasad M, Chiueh Tzi-cker. A Binary Rewriting Defense against Stack based Buffer Overflow Attacks. In: Proc. of the General Track: 2003 USENIX Annual Technical Conference, San Antonio, Texas, 2003

7 Chiueh Tzi-cker, Hsu Fu-hau. RAD: A compile time solution for buffer overflow attacks. In: 21st IEEE Intl. Conf. on Distributed Computing Systems(ICDCS), Phoenix, AZ, 2001

8 Shaneck M. An Overview of Buffer Overflow Vulnerabilities and Internet Worms. CSCI, 2003

9 Gillette T B. A Unique Examination of the Buffer Overflow Condition:[thesis for the degree of Master of Science]. Melbourne, Florida: College of Engineering at Florida Institute of Technology, 2002

10 Cowan C, et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: 7th USENIX Security Symposium, San Antonio, TX, 1998

11 Cowan C, Beattie S, Johansen J, et al. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In: 12th USENIX Security Symposium, Washington DC, 2003

12 Tuck N, Calder B, Varghese G. Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow. In: Proc. of the 37th International Symposium on Microarchitecture, Portland, Oregon, 2004

13 Etoh H. GCC extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/, June 2000

14 Austin T, Breach S, Sohi G. Efficient detection of all pointer and array access errors. ACM Conference on Programming Language Design and Implementation, Orlando, FL, 1994

15 Jones R, Kelly P. Backwards-compatible bounds checking for arrays and pointers in C programs. In: Proc. of the Third International Workshop on Automatic Debugging, Sweden, 1997

16 Smirnov A, Chiueh Tzi-cker. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In: Proc. of NDSS'05, San Diego, CA, 2005

17 Jim T, et al. Cyclone: A safe dialect of C. USENIX Annual Technical Conference, Monterey, 2002

18 Condit J, et al. CCured in the Real World. Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, San Diego, 2003

19 Baratloo A, Singh N, Tsai T. Libsafe: Protecting critical elements of stacks:[Technical Report]. New Jersey: Avaya Labs, Avaya Inc, 1999

20 Robertson W, Kruegel C, Darren Mutz D, et al. Run-time Detection of Heap-based Overflows. In: Proc. of the 17th Large Installation Systems Administrators Conference, San Diego, 2003

21 Pierre-Alain F, Vincent G. A Buffer Overflow Study Attacks & Defenses. http://www.wntrmute.com/docs/bufferoverflow/report.html

22 Toth T, Kruegel C. Accurate Buffer Overflow Detection via Abstract Payload Execution. In: Proc. of the 5th International Symposium on Recent Advances in Intrusion Detection, Zurich, Switzerland, 2002

23 Ghosh A K, O'Connor T. Analyzing Programs for Vulnerability to Buffer Overrun Attacks. In: Proc. of the National Information System Security Conf. Crystal City, VA, 1998

24 Du Wenliang. Testing for Software Vulnerability Using Environment Perturbation. In: Proc. in Intl. Conf. on Dependable Systems and Networks, New York, 2000

25 Fink G, Bishop M. Property-Based Testing: A New Approach to Testing for Assurance. ACM SIGSOFT Software Engineering Notes, 1997, 22(4):74～80

26 Weiser M. Program slicing. IEEE Transactions on Software Engineering, 1984, SE-10(4):352～375

27 Pfleeger C, Pfleeger S, Theofanos M. A methodology for penetration testing. Computers and Security, 1989, 8(7):613～620

28 Bhatkar S, DuVarney D, Sekar R. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In:Proc. of the 12th USENIX Security Symposium, Washington, DC, 2003

29 Chew M, Song S. Mitigating Buffer Overflows by Operating System Randomization: [Technical Report]. Pittsburgh: Carnegie Mellon University,2002

30 Younan Y, Joosen W, Piessens F. A Methodology for Designing Countermeasures Against Current and Future Code Injection Attacks. In:Proc. of the Third IEEE International Information Assurance Workshop, University of Maryland, 2005

31 Kc G, Keromytis A, Prevelakis V. Countering Code-Injection Attacks with Instruction-Set Randomization. In:Proc. of the 10th ACM Conf. on Computer and Communication Security, Washington, DC, 2003

32 Barrantes E, Ackley D, Forrest S, et al. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In:Proc. of the 10th ACM Conf. on Computer and Communication Security, Washington, DC, 2003

33 Lee R B, Karig D K, McGregor, et al. Enlisting hardware architecture to thwart malicious code injection. In: Proc. of the Intl. Conf. Security in Pervasive Computing, Boppard, Germany, 2003

34 Xu J, Kalbarczyk Z, Patel S, et al. Architecture support for defending against buffer overflow attacks. In: 2nd Workshop on Evaluating and Architecting Systems for Dependability, San Jose, California, 2002

35 Shao Zili, Zhuge Qingfeng, He Yi, et al. Defending Embedded Systems Against Buffer Overflow via Hardware/Software. In: Proc. of IEEE 19th Annual Computer Security Applications Conference, Las Vegas, 2003

36 Piromsopa K, Enbody R. Secure Bit2: Transparent, Hardware Buffer-Overflow Protection: [Technical Report]. Michigan: Michigan State University, 2004

37 Sidiroglou S, Keromytis A D. Using Execution Transactions To Recover From Buffer Overflow Attacks:[Technical Report]. Columbia: Columbia University, 2004