维普资讯 http://www.cqvip.com

栈溢出的动态检测技术

刘通平
（上海宇梦通信科技有限公司　上海 200121）

摘　要　缓冲区溢出是计算机界的一个古老话题，计算机界和学术界为检测和预防缓冲区溢出投入了很多的精力。但根据 CERT(www. cert. org)的数据显示，最近几年中，缓冲区溢出大约占程序错误的 50%。另外，根据 CERT Advisory 数据显示，目前仍然有 50%左右的安全威胁系来自缓冲区溢出。因此可以说，缓冲区溢出的问题并没有得到根本的解决，而栈溢出是一种最基本的缓冲区溢出。和堆溢出相比，栈溢出更难于监控和危害性更大，因此研究栈溢出具有实际意义。本文对各种栈溢出的检测技术进行了分类和总结，希望能够对栈设计溢出的检测工具提供一些思路。同时，本文介绍了实现栈溢出的动态检测技术中涉及到的各种插装技术，并对各种各样的插装技术进行了总结。

关键词　缓冲区溢出，栈溢出，插装技术

Dynamic Detection of Stack Overflow

LIU Tong-Ping
(Shanghai Yumeng Communication Technology CO. LTD, Shanghai 200121)

Abstract　Buffer overflow has been studied carefully and sophisticatedly in these years. Computer community has spent a lot of efforts in this field. But according to the datum in www. cert. org, buffer overflow error is still about 50% of all program error in recent years. And according to CERT Advisory, 50% security threat comes from the bug relating with buffer overflow too. From this point, we can see that buffer overflow has not been resolved completely. The research on buffer overflow still has actual meaning. Stack overflow is a basic form of buffer overflow and it is more difficult to be detected and protected comparing to heap overflow. In this article, different dynamic techniques about the detection of stack overflow have been described. It is useful for the design of software analyzing tools. In the same time, the instrumentations used by different technique are described and concluded too.

Keywords　Buffer overflow, Stack overflow, Instrument technique

1　引言

缓冲区溢出通常包括栈溢出和堆溢出。堆溢出主要指堆空间的内存溢出。由于堆空间的内存管理都是通过几个固定的库函数进行的，因此其溢出问题可以通过对这几个函数进行监控得到解决。目前已经有比较完善的工具检测堆溢出问题，比如在 Linux 系统下有一个开源的工具 Valgrind，已经几乎能够检测各种类型的堆溢出。栈溢出是指栈的使用错误造成的栈的内容的破坏，比较难以检测，因为栈是动态生长的而栈溢出的类型多种多样。John 和 Mariam[1] 根据栈溢出所破坏的内容把栈溢出分成以下 6 种类型：

1)返回地址；

2)基指针；

3)作为变量的函数指针；

4)作为函数参数的函数指针；

5)作为变量的 longjmp 缓冲区；

6)作为函数参数的 longjmp 缓冲区。

实际上，这仅仅是给出了栈溢出的一种分类方法。栈溢出还有其它的分类，比如根据栈溢出发生的位置可以分为本地栈溢出和远程栈溢出等。

本文第 2 节首先对栈溢出的类型进行分类，然后简单地阐述目前存在的栈溢出检测技术(包括静态检测和动态检测技术)和栈的保护技术。第 3 节对不同的栈溢出检测技术进行比较详细的阐述，特别是动态检测技术。动态检测技术也是目前比较流行的栈溢出检测技术，本文将对每种动态检测技术都以一个具体的实现进行详细阐述，并且分析此类技术的优缺点和实现难度。最后对本文描述的各种动态检测技术进行了总结。

2　栈溢出原理和栈溢出的检测技术

2.1　栈溢出原理

栈溢出主要指栈的关键内容被外界所改变。实际上，在 i386 体系结构中，一个典型的函数调用栈如图 1 所示。

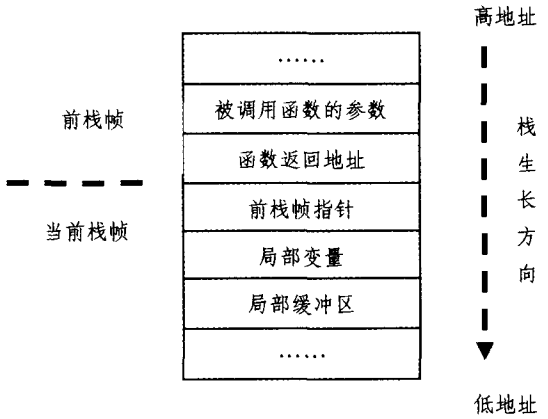


图 1　栈帧结构

如果局部缓冲区发生溢出(超过了缓冲区的上边界)，即往栈生长方向的相反方向生长，那么就有可能覆盖一些关键指针，比如前栈帧指针或者被函数返回地址等。此时，程序的运行就会发生异常，比如返回地址的修改将使程序运行跳转不可预料的地址，例如跳转到恶意程序等。

维普资讯 http://www.cqvip.com

栈溢出的分类多种多样。根据栈溢出的方向可以分为“栈上溢”和“栈下溢”，比如固定大小的缓冲区的正负越界操作就会发生栈上溢和栈下溢。根据栈溢出时对数据的改写方式是否连续，可以分为连续的栈溢出和不连续的栈溢出。目前的大多数检测工具都只能检测连续的栈溢出。

2.2 栈溢出的检测和保护技术

根据检测发生的阶段，栈溢出的检测可以分为静态检测技术和动态检测技术。

·静态检测技术指的是不依赖于程序的运行就能检测出栈溢出的错误，静态检测一般发生在编译前和编译后。

·动态检测技术指的是检测过程必须依赖于程序的运行，一般需要在程序中插入或者修改一些指令来检测栈的溢出。根据插入方式的不同，动态检测技术可以进一步划分为静态插装和动态插装。静态插装技术主要是指对程序的修改发生在程序运行之前。目前静态插装技术主要有两类：基于源码的静态插装技术和基于二进制代码的静态插装技术。而动态插装指的是在程序运行过程中对二进制代码进行修改，在函数的入口和出口插入一些指令来完成对栈使用情况的监控。

2.2.1 静态检测技术

当前静态检测工具主要集中于利用函数流程图来分析堆内存的溢出，但非常少的工具静态分析栈溢出。不过，还是有些工具尝试着静态分析栈溢出，这些工具有的是在二进制层面或者汇编层面进行源码分析，以获得比较精确的栈的使用过程，或者通过扩展编译器让编译器输出栈的使用信息。由于在静态环境下很难获得精确的栈的大小，特别是在函数的递归调用而递归的次数却是一个静态下不可确定的函数参数时，因此静态工具只能部分检测栈溢出的情况。另外，静态分析工具通常都会产生很多错误的警告信息，这会实际上加重编程者的负担。因此，对栈溢出的检测更多的是通过动态检测的方式进行。

2.2.2 动态检测技术

·基于源码的静态插装技术

根据插装的方式不同，基于源码的静态插装技术还可以进一步划分成基于编译器和直接修改源码的插装技术。基于编译器的插装技术将对现有的编译器进行扩展，通过编译器收集栈的信息，然后在合适的地方插入一些指令。根据栈溢出检测方式的不同，目前常用的两种方法是基于参考对象的边界检查和利用“canary”值检查栈溢出。

实际应用中，直接修改源代码并不是一种非常可取的行为。因为这可能导致大量的人力浪费，对于每一个新的检测对象都必须进行源代码的替换。因此，直接修改源码的方式来检测栈溢出的工具不多，但它们往往能够提供一种非常好的思想，比如“以堆替栈”的思想。通过把栈的使用转换成易于检测的“堆的使用”，因此可以比较方便地进行检测。

·基于二进制代码的静态插装技术

目前，这类技术可以分成以下两类：一类是通过静态分析程序生成的二进制代码，找出相关函数的入口点和退出点，然后修改二进制代码。在相关函数的入口点和退出点插入一些语句来检测栈的运行情况。另一类工具是通过对带调试信息的程序提取出全局和局部缓冲区的大小信息，并把这些信息重新组织后，重新写回到二进制代码文件中，然后通过监控内存操作库函数对这些全局和局部缓冲区的操作行为来检测是否发生了栈溢出。因为其主要信息并不是来自二进制代码而是来自编译生成的调试信息，因此从根本意义上来说，该类工具并不能算是一种纯粹的基于二进制代码的工具，而应该属于基于调试信息的二进制代码插装技术。

·动态插装技术

这类技术是在运行过程中对二进制代码进行修改，在函数的出入口插入一些指令来获取栈的运行情况。因为能够摆脱对源代码的依赖，比较适合分析那些无法获得源码的遗产程序。

2.3 其它栈检测和保护技术

因为目前的栈溢出检测技术并不完美，没法检测出所有类型的栈溢出等。由于无法做到避免所有栈溢出错误，学术界就提出其它一些技术用来保护栈的使用。比较突出的技术有：

库函数的重写。因为目前的内存操作库函数很多都是危险性的函数，比如 strcpy()，只有检测到字符串结束标志才会停止拷贝行为。如果 strcpy()的目标地址是栈地址的话，那么很有可能会导致栈的内容都被破坏。比如 libsafe[3] 对 strcpy()进行了改装，在改装后的 strcpy()中，将会计算目标缓冲区的上限并进行边界检查。只有边界检查通过后，才会调用安全的库函数 memcpy()进行字符串的拷贝。

扩展 C 语言规范。这类工具往往对现存的计算机语言进行扩展(比如语言)，对语言进行了更严格的限制。因为造成栈溢出的一个最根本原因是 C 语言本身缺乏类型安全，过分追求性能而忽视正确性。数组和通用指针的混用造成数组边界很难确定。这类工具包含 CCured[9]，CyClone 等。CCured 根据指针的使用对指针类型进行了区分，目的是防止对指针的不正确使用(比如访问不该访问的区域)而导致栈溢出。不过，为了让 CCured 能够正常，开发者必须对代码进行标注，以便 CCured 能够确定指针类型。CCured 是一种源到源的 C 语言翻译器。它可以通过分析 C 程序代码来确定最少需要插入的运行时检查程序的个数。这些检查被嵌入程序代码，用于防止所有的内存安全冲突。最终生成的程序是(内存安全)具有内存保护功能的，也就是说这时的程序不会出现缓冲区溢出或者对本不能涉及的内存区域胡乱做修改的错误。

栈不可执行限制。由于实际上大多数程序不会在栈中放置可执行代码，Solar Designer 提出了一种栈的可执行限制来进行一定的栈栈保护[13]。比如原来 Linux 系统中用户代码段的界限为 0 到 3G 空间。新的实现通过重新调整操作系统的地址空间，用户代码段的界限从 3Gb 减小到 2.75GB，而 2.75G以上为栈空间。当用户企图执行处于新的栈空间 0xB0000000-0xBFFFFFFFF 范围内的指令时，就会产生段越界的一般保护故障(GFP)，相应的处理程序就可以对栈溢出进行处理。但是这种办法并不完美，只能用于保护可执行代码在栈中的情况。实际上，当恶意代码位于代码段或其它可执行的数据段时，就可绕过该方法的保护。

3 动态栈溢出检测技术

3.1 基于源码的静态插装技术

3.1.1 基于编译器的静态插装技术

基于“canary”值的边界检测

基于“canary”值的边界检测主要是 Cowan 提出来的一种检测策略[9]，目前其实现 StackGuard 可能是最早的且目前被引用率最高的检测技术。后来，又有很多工具在 StackGuard 的基础上进行了扩展，比如 PointGuard，Stackshield，Propo-

维普资讯 http://www.cqvip.com

lice 等。但其基本原理都是一样的，只不过在局部进行了优化而已。这类工具的说明很多，因此此处就不再赘述了，仅仅说明一下其最主要的特点：

StackGuard[8]是一种编译器的扩展，编译时修改后的编译器将在返回地址和局部变量之间插入一个“canary”字，并在函数的退出前插入一段指令，用来校验该“canary”字是否被修改。如果当函数返回时，发现这个 canary 的值被改变了，就证明可能有缓冲区溢出攻击发生，工具会发送一则入侵警告消息给 syslogd，然后停止工作。StackGuard 的基本假设是缓冲区溢出的写操作是连续的，不能用于检测非连续的栈溢出。

PointGuard[4]是对 StackGuard 的提高和改进，原来的 StackGuard 仅仅用于检测返回地址是否溢出。PointGuard 对此进行了扩展，在函数指针、longjmp 缓冲区之后也增加“canary”字来进行保护，防止被缓冲溢出操作所改写。由于原来的 StackGuard 可以通过某种办法被绕过[8]，为此，PointGuard 提出了一种新的机制来保护“canary”的安全，对保存在内存中的指针进行了加密，不过也引入了一定的性能开销。

StackShield[6]也是对 StackGuard 进行的改进，StackShield 在另一块不能缓冲溢出的内存空间储存函数返回地址的一份拷贝。它在受保护的函数的入口处和出口处分别增加一段代码，入口处的代码用来将函数返回地址拷贝到一个新的内存空间中，而出口处的代码将返回地址从表中拷贝回堆栈。因此函数执行流程不会被改变，很好地预防了栈溢出对函数返回地址的破坏。

Propolice[14]是 IBM 日本研究院实现的一种栈溢出检测技术，其基本思想也是来源于 StackGuard，但是 Propolice 还对栈上的缓冲区、局部变量和函数指针的位置进行了重新排列，使得缓冲区的位置比局部变量和函数指针更靠近函数的返回地址，这样使得局部变量和函数指针不会遭受栈的缓冲区溢出的影响。不过，其实这种办法并不能完全防止栈溢出。

基于参考对象的边界检测

Jones 和 Kelly[11]基本假设是从一个界内指针计算的地址必须和其原始指针具有相同的参考对象。利用一个全局的对象表收集所有参考对象的基地址和大小信息。为了确定一个从界内指针计算的地址仍然在界内，检查器必须先通过和存储在对象表中对象的基地址和大小进行比较以定位参考对象是否落在参考对象的地址范围内。该工具通过修改 GCC 的前端捕获所有对象的创建，地址使用和解除参照的操作，并插入一些指令调用校验库中的相关函数进行分析和处理。因此，所有对象的操作都会被截获，而这些操作将由校验库中相关的函数对对象的操作进行校验，检查所有创建的地址和解除参照的地址都在既定范围内。

但由于 Jones 和 Kelly 的机制并没有很好地处理界外指针，当处理界外指针时会出现程序的中断，因此 CRED[10]（C 范围错误检测器）采用了一个新的方法来支持程序使用界外内存。Jones 和 Kelly 的机制不允许出现界外指针，当出现界外指针时就被认为已经发生了栈溢出而停止程序的执行。但实际程序中，出现了界外指针而又不造成栈溢出的例子很多，如果采取这种标准，那么就会造成大量的程序中断。比如以下案例：

```
void exam()
{
  1: char *p, *q, *r, *s;
  2:
  3: p = malloc(4);
  4: q = p + 1;
  5: s = p + 5;
  6: r = s - 3;
  ..............
}
```

由于第 5 行实际上出现了界外指针，因为实际分配了 4 个字节的空间，而 s 指针却出现了(p+5)的操作。这类程序在 Jones 和 Kelly 的机制中就会由于出现了界外指针而被意外中断。因此，CRED 提出了一种宽松的标准，只要界外指针的使用不会造成栈溢出，就允许使用界外指针。当然，这类越界指针的使用可能会造成实际的问题。因此，CRED 同样也对这类问题进行了记录，重新分配一个特殊的“越界对象”数据结构，用来保存这种越界指针。这种越界指针可以和普通指针一样进行数据拷贝的操作。另外，CRED 只检查可能发生安全问题的函数，有效地减少了运行时的负荷。目前，CRED 已经在 20 个开源程序共计 120 行代码上进行了测试。目前所有的程序都能够编译通过，而 Jones 和 Kelly 的工具却有 60%的程序会由于越界指针的问题而编译失败，而且速度提高了 12 倍。

由于 CRED 是对现有的编译器进行修改，提取出所关注对象的操作行为并进行检测，因此其检测的效果取决于对关注对象的定义。实际上，这种技术的致命缺点是把结构和数组当作一个单一的内存块进行处理，但这实际上也是 C 标准所决定的。CRED 只能对在编译时就已经明确的操作指令进行检测，比如检测 s.buffer[i]或者 s->buffer[i]这类缓冲区操作指令。但对结构中的缓冲区通过一个假名或者一个类型转换后的指针进行访问时，这类栈错误就没法被测到。不过，这实际上也是所有居于编译器的插装技术的一种固有的弱点。

3.1.2 直接修改源代码的静态插装技术

实际应用中，直接修改源代码并不是一种非常好的行为，因为涉及到大量的人力劳动。因此，这类工具并不是很多，大多数栈溢出检测工具都是从编译器的角度进行静态插装，但并不是所有的思想都能通过修改编译器实现的，因此下面这两个工具通过修改源代码体现出了非常有意义的栈溢出的检测思想。

“以堆替栈”技术：相对栈溢出而言，堆内存操作比较容易监测，因为堆空间的分配和释放都是通过几个固定的库函数进行的。常用的检测方法是设法获取堆内存的分配函数和释放函数的控制权（比如通过编译重定向或者预加载技术等），通过在预定内存的两侧都额外分配一些的多余的内存并填充一些特殊字节人为地制造两块“内存夹板”。当堆内存释放时，通过检测这两块夹板的内容是否发生改变来确定是否发生了堆溢出。但栈溢出比较难于检测，因为栈是动态生长的而且栈溢出的类型多种多样。因此，美国哥伦比亚大学的 Sidiroglou，Giovanidis 和 Keromytis 提出了“以堆替栈”的技术来检测栈溢出[12]，即 DYBOC 技术。它是把栈搬移到堆上，由此所有对栈的操作就会转换成对堆的操作而便于检测。其基本原理如图所示：

```
Original Code                    Modified Code
int func()                       int func()
{                                {
  char buf[10];                    char buf[10];
  ...                              ...
  sub_func()2  sub  func();
  ...                              ...
  return 0;                        Free(buf);return 0;
}                                }
```

维普资讯 http://www.cqvip.com

DYBOC 通过自己定义的 pmalloc()函数就可把栈上的内存搬移到堆上，然后在函数的结尾处再通过 pfree()来释放掉这段内存。由于局部变量 buf 所指的内存是进入函数时分配而退出函数时释放，因此和栈内存的声明周期保持一致。通过对 pmalloc()和 pfree()函数进行包装就可实现检测堆内存是否溢出。和一般检测堆内存溢出的基本原理一样，pmalloc()函数也是在预定内存的两侧进行处理来检测溢出情况。不过，为了不用分配实际的内存而节省物理内存，pmalloc()函数巧妙地利用了类 Unix 系统的 mmap 机制，在预定内存的两侧映射了两块全 0 的写保护的页。在类 Unix 系统中，/dev/zero 是只读属性的设备，因此把预定内存两侧的页映射到/dev/zero 设备上。当对 pmalloc()分配的预定内存以外的区域（上溢或者下溢）进行写操作时，就会引发系统的中断，通过预先注册的中断处理函数，就可以实现对“栈溢出”的捕获和恢复。

Eric Haugh 和 Matt Bishop 提出了一种新的修改源代码的方法（STOBO）来检测由于危险的库函数引起的栈的溢出[17]。现在使用的很多字符串操作或者内存操作的库函数都是不安全的，比如 memcpy，strcpy 等。比如 memcpy(dst, src, n)，库函数并不会判断 dst 所对应的缓冲区的大小是否大于拷贝的字节数 n。而 strcpy(dst, src)不清楚在遇到结束符‘\0’之前会有多少字节，经常导致溢出。

因此，Haugh 和 Bishop 想出一种基于源码的插装技术。首先通过在代码中插入一个函数用来向测试系统报告缓冲区的大小，然后通过修改危险的库函数对缓冲区的操作进行监控。其监控栈溢出的基本原理如下所示：

```
原始代码                    修改代码
int func(char * src)        int func(char * src)
{                           {
  char buf[10];               char buf[10];
  …                           __STOBO_stack_buf(buf,sizeof(buf));
  strcpy(buf,src);            …
  …                           __STOBO_strcpy(buf,src);
  return 0;                   …
}                             return 0;
                            }
```

首先通过__STOBO_stack_buf()把栈缓冲区的大小记录到一个全局的缓冲区中，然后通过修改过的__STOBO_strcpy()函数进行字符串拷贝操作。在__STOBO_strcpy()函数中，可以通过比较预先记录的栈的大小和源字符串的长度来检查字符串拷贝是否会造成栈溢出。如果会造成栈溢出，那就拒绝进行字符串拷贝，并且报告给用户。不过，利用这种机制只能检测由危险库函数引起的栈溢出现象，但实际应用中很多的栈溢出并不是由于危险库函数引起的，因此可见这种方法只能检测特定类型的栈溢出，不具有普遍适用性。

3.2 基于二进制代码的静态插装技术

根据第 2 节所述，基于二进制代码的静态插装技术主要分成两类：一类是完全依赖二进制代码进行代码的静态插装，比如 Prasad 和 Chiueh 提出来的 RAD（Retrun Address Defense）工具[15]。由于以前的一些栈溢出检测工具和技术都必须依赖源码，这在遗产程序和合作开发的程序中经常是不可能的。因此 Prasad 和 Chiueh 提出来一种完全抛弃了源码的解决方案，对 Win32/Intel 下面的可移植的可执行文件（Portable Executable，内含 32 位程序代码和数据，是 UNIX Common Object File Format（COFF）的演化）进行静态插装。RAD 的大体思路是通过静态分析二进制代码，找出“感兴趣”函数的入口点和退出点，并插入一些语句来检测栈的运行情况，然后重新生成新的二进制代码文件。RAD 所定义的“感兴趣”函数为包含一系列的关于局部变量的栈的分配和释放的指令的函数，其检测方法是在函数的开头把栈上的返回地址保存到一个返回地址仓库（RAR-Return Address Repository）中，而函数的结束处将会检查栈上的返回地址和返回地址仓库中所保存的是否一致。如果不一致，则证明发生了栈溢出。RAD 实现的主要难点在于源码的反汇编和“感兴趣”函数的确定。对于 i386 体系结构的可执行程序而言，其指令的二进制码长度并不固定，从 1 字节到 8 个字节都有可能，因此反汇编时往往容易出错，一个字节差距可能会导致反汇编后的汇编代码完全不一致，而且目前还有一种技术可以通过在函数的代码段中插入一些额外的字节来“欺骗”反汇编器。如果没有源码或者符号信息的帮助，如何确定函数的开头和结尾也是一项非常艰巨的任务。虽然 Prasad 和 Chiueh 也提出来通过从头信息中提取 main 函数的信息，以此为突破口继续往下分析，但实际上由于反汇编技术的局限性，这类方法并不能作为一种实用的栈溢出检测技术。

第二类技术是利用程序所带的调试信息来获取栈相关的缓冲区信息，从而完成二进制代码的静态插装。如第 2 节所述，其实这类工具应该属于基于调试信息的二进制代码插装技术。这种方法的典型是印度理工的 Kumar Avijit、Prateek Gupta 和 Deepak Guptaa 提出来的两个工具 TIED 和 Libsafe-Plus[5]。TIED 利用 libdwarf 提供的一个消费者接口去读取可执行文件中的 DWARF 信息，包括缓冲区到栈起始地址的偏移和大小。然后构建一个能够提供运行时快速查询的数据结构来保存缓冲区信息，并且把这个结构作为一个可读的、可加载的段重新写回到可执行文件中。通过从程序所带有的调试信息中提取出全局和局部缓冲区的大小信息，并把这些信息重新组织后重新写回到二进制代码文件中，然后通过监控内存操作库函数对这些全局和局部缓冲区的操作行为来检测是否发生了栈溢出。然后，利用另一个工具 Libsafe-Plus 对缓冲区的内存操作进行监控，其实现原理和 Libsafe 基本一致，因此又称为 Libsafe-Plus。传统的 Libsafe 仅仅对栈溢出进行了保护，而且对于栈变量而言，Libsafe 也只是对缓冲区的上限做了一个假设，实际上并没有确定其实际的大小，因此攻击者完全可以通过修改紧挨着缓冲区的变量而实施攻击。而 Libsafe-Plus 则可检测出更多的栈溢出。这类技术实际上仍然需要依靠源代码提供某种特殊的调试信息，这必然需要源代码重新按这种特殊的命令编译才能产生这种调试信息。实际上，如果按照“-g”这种方式编译出来的二进制文件，利用“objdump -S”进行反汇编时是可以看见源码的。因此，这实际上并不能摆脱对源代码的依赖。

3.3 动态插装技术

上面阐述的技术除了基于二进制代码的静态插装技术 RAD 以外，其它技术都需要源码的支持。而 RAD 技术本身又会由于反汇编技术的限制，容易产生太多的 false negatives 和 false positives。因此，很多人就想到了动态插装技术。由于动态插装技术可以不依赖源码的支持，因此对此技术的研究比较热。下面提出其中两种有代表性的技术。

第一个技术是 Arash Baratloo、Timothy Tsai 和 Navjot Singh 开发的 libverify[3]。Libverify 利用特定平台提供的预加载技术，比如 Linux 下的 PRELOAD 环境变量的设定，通过预加载_init()函数对二进制代码进行插装。_init 和_fini 是函数库中用来初始化函数库和关闭的时候做一些必要的处理的函数，libverify 通过预加载_init()函数。_init()函数首先确定监控的函数代码的大小和位置，然后把这部分代码搬移

维普资讯 http://www.cqvip.com

到一个新的堆中，并对代码进行插装，起始地址处将放置一个跳转到 wrapper_entry()的跳转语句，而结束代码处也放置一个跳转语句(跳转到 wrapper_exit())。Wrapper_entry()和 wrapper_exit()的处理大致和 StackGuard 相仿，也是采用“canary”的技术来检测栈溢出，即 wrapper_entry()保存返回地址而 wrapper_exit()将检查是否栈上的返回地址已经被修改。该技术不能用于监控静态链接的程序。

第二个技术是印度理工的 Suhas Gupta、Pranay Pratap、Huzur Saran 和 S. Arun-Kumar 提出来的[2]，其原理和 libverify 极为相似。但是，没有做函数代码的拷贝，只是修改了函数的入口和出口。其主要的原理如图 2 所示。

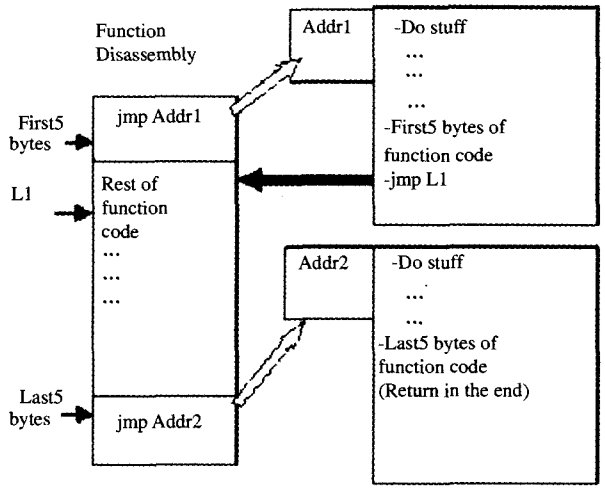


图 2　出入口替换的动态插装技术

实际上，不一定能够找出 5 个字节的指令来进行安全的跳转。比如，函数的出口处是一种这样的指令：

```
8048340:e8 03 00 00 00   call 8048348 <hello>
8048345:c9               leave
8048346:c3               ret
8048347:90               nop
08048348:<hello>:
8048348:55               push %ebp
```

实际上，按文[2]的意思，必须从函数的出口处找出 5 个字节的代码放置跳转语句。因此，从上面的二进制代码分析，很难找出 5 字节的代码用于放置跳转指令。如果选择从 0x8048345 处开始放置的话，那么它将会覆盖新的函数(此处为 hello)的代码。同时，因为替换时不可能从一条完整代码的中间开始替换，这样会打断代码的执行顺序而造成错误，因此根据分析只能从 0x8048340 处进行替换。这样，在跳转到如上图所示的 Addr2 时没有问题，但执行函数代码的最后 5 字节会出现问题。因为此处是一个短跳转语句，不可能跳转到我们需要的 hello 函数，这样就会出错。当然，研究了作者的另外一篇文章[16]才知道，原来这种方法只能适用于特殊的编译格式，比如 Win32/Intel 平台下以调试模式的编译，其函数的入口和出口如下。

函数入口：

```
00416FE0: 55             push ebp
00416FE1: 8B EC          mov ebp,esp
00416FE3: 83 EC 60       sub esp,60h
00416FE6: 53             push ebx
00416FE7: 56             push esi
00416FE8: 57             push edi
```

函数出口：

```
00416FF9: 5F       pop edi
00416FFA: 5E       pop esi
00416FFB: 5B       pop ebx
00416FFC: 8B E5    mov esp,ebp
00416FFE: 5D       pop ebp
00416FFF: C2 04    ret 4
```

该工具是通过把上面语句中标灰的语句进行替换的。但实际上，并不是所有的编译器在不同的编译模式下都会产生如上所示的代码的，比如 GCC 在-fomit-frame-pointer 编译选项的控制下产生的代码就和以上的代码大相径庭，因此该方法不具有普遍适用性。

结论　目前，栈溢出的检测方式多种多样，但是都没有一种工具能够完全检测出所有类型的栈溢出，因此学术界才会提出一些技术对栈的使用进行保护。当然栈保护也容易被绕过。

使用什么样的栈溢出检测技术，取决于实际的环境，但各种栈溢出技术都有其各自不同的局限性。此处，对这些技术进行一个小结，期望对学术研究和工程应用提供一个很好的借鉴作用。

静态检测技术：由于在静态环境下很难获得精确的栈使用情况，特别是当发生函数的递归调用时，因此静态工具只能检测部分的栈溢出情况。另外，静态分析工具通常都会由于产生很多错误的警告信息而带来大量的人工检查工作。相对动态检测技术而言，静态检测技术有一个优势，即可以不需要实际发生栈溢出。而动态检测技术只有在栈溢出发生时才能检测到。由于测试代码不太可能对所有的程序进行完全的覆盖测试，因此动态情况下有些栈溢出的代码实际上不能被触发，而静态分析工具实际上能够检测出这样的一些栈溢出情况。

动态检测技术：当前比较著名的动态检测栈溢出的技术如图 3 所示。

顾名思义，基于源码的静态插装技术都必须依赖源代码才能进行代码的动态插装。目前主要有两种实现方法：基于编译器的方法，实现难度比较高，但是能够提供比较完整的栈操作的信息和函数的信息；直接修改源码的方法，实现难度低，但可能会带来一些误操作，而且需要大量的人力。而基于编译器的两种方法相比，基于“canary”值的边界检测技术比较容易被绕过，检测效果可能会差点。而基于参考对象的技术实际上可以检测出“栈上溢”和“栈下溢”的错误，其主要的缺点是负荷太大，而且可能有些内存操作不能被编译器识别，但结合其优点而言，不失为一种比较不错的技术。

对于基于二进制码的静态插装技术而言，基于调试信息的插装技术实际上也不能摆脱对源码的依赖。而反汇编的插装技术容易出现反汇编的错误，导致检测效果大打折扣，甚至可能造成代码的运行不正常。

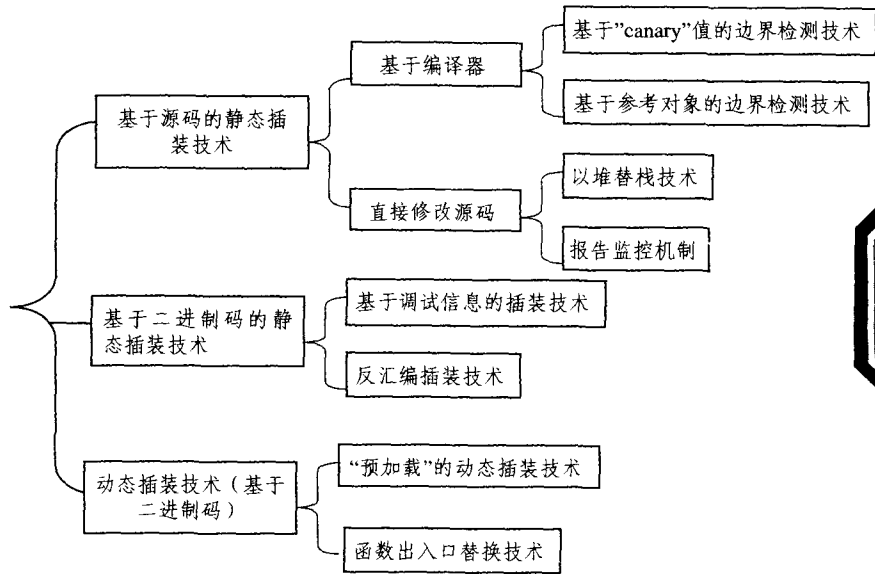
关于动态插装技术，“预加载”的技术不能检测出静态链接代码的栈溢出错误，而函数出入口替换技术目前由于只能用于特殊编译模式下的代码，因此大大限制了应用范围。

总的而言，现存的工具都还有不少的缺点。有些工具只是通过监控 strcpy 或者 memcpy 等几个易造成缓冲区溢出的函数(比如 libverify，TIED 等)，这在实际的应用中是非常不够的。有些工具在建立参考对象时也仅仅监控指针或者缓冲区数组，但实际的溢出中也有可能是通过对函数参数取地址再进行溢出的。但动态检测技术的最主要缺点在于只能检测出真正发生动态溢出的代码。实际上只有极少量的代码能够触发栈溢出，因此动态检测技术并不能检测出所有可能的栈溢出代码。

(下转封四)

维普资讯 http://www.cqvip.com

（上接第 286 页）



P-281
N.361P1

图 3　当前的栈溢出动态检测技术总结

因此，很多人提出是否能够结合静态技术和动态技术共同检测栈溢出。但到目前为止，仍然没有一个完美的解决方案来检测出所有的栈溢出。

参 考 文 献

1 Wilander J, Kamkar M. A comparison of publicly available tools for dynamic buffer overflow prevention. In: Proceedings of the 10th Network and Distributed. System Security Symposium, Feb. 2003. 149～162

2 Gupta S, Pratap P, Saran H, et al. Dynamic Code Instrumentation to Detect and Recover from Return Address Corruption. In: Workshop on Dynamic Analysis WODA 2006. May 2006

3 Baratloo A, Tsai T, Singh N. Transparent run-time defense against stack smashing attacks. In: Proceedings of the USENIX Annual Technical Conference, June 2000

4 Cowan C, Beattie S, Johansen J, et al. Pointguard : Protecting pointers from buffer overflow vulnerabilities. In: Proc. USENIX Security Symposium, 2003

5 Avijit K, Gupta P, Gupta D. TIED, LibsafePlus: Tools for runtime buffer overflow protection. In: Proceedings of the 13th USENIX Security Symposium, August 2004. 45～46

6 Vendicator. Stack Shield technical info file v0. 7. http://www.angelfire.com/sk/stackshield/, January 2001

7 Bulba, Kil3r. Bypassing StackGuard and StackShield. http://www.phrack.org/show.php? p=56&a=5, 2000.

8 Cowan C, Pu C, Maier D et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, January 1998. 63～78

9 Necula G, McPeak S, Weimer W. CCured: Typesafe retrofitting of legacy code. In: Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages, Portland, OR, January 2002

10 *Ruwase O, Lam M. A practical dynamic buffer overflow detector.* In Proceedings of Network and Distributed System Security Symposium, 2004. 159～169

11 Jones R W M, Kelly P H J. Backwards-compatible bounds checking for arrays and pointers in C programs. In: Automated and Algorithmic Debugging. 1997. 13～25

12 *Sidiroglou S, Giovanidis G, Keromytis A D. A Dynamic Mechanism for Recovering from Buffer Overflow Attacks.* In: 8th Information Security Conference, September 2005

13 Solar Designer. NonExecutable User Stack. http://www.false.com security/linux-stack/

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

15 Prasad M, Chiueh T. A binary rewriting defense against stack based buffer overflow attacks. In: Proceedings of the 2003 Usenix Annual Technical Conference, Jun. 2003

16 Pratap P, Gupta S. Dynamic Code Instrumentation to Prevent and Recover from Stack Smashing Attacks. Dec. 2005. http://genie.iitd.ernet.in/projects/pranay/FinalReport.pdf

17 Haugh E, Bishop M. Testing C Programs for Buffer Overflow Vulnerabilities. In: Proceedings of the 2003 Network and Distributed System Security Symposium. Feb. 2003

计算机科学

（1974 年 1 月创刊）
第 34 卷第 9 期（月刊）
2007 年 9 月 25 日出版

国际标准连续出版物号 ISSN 1002－137X
国内统一连续出版物号 CN50－1075/TP

定价：30.00 元　国外定价：5 美元
邮发代号：78－68
发行范围：国内外公开

主管单位：国家科学技术部
主办单位：国家科技部西南信息中心
编辑出版：《计算机科学》杂志社
重庆市渝北区北部新区洪湖西路 18 号　邮政编码：401121
电话：（023）63500828　E-mail：jsjkx@swic.ac.cn
网址：www.jsjkx.com
社　　长：牟炳林
总　　编：彭　丹
主　　编：朱宗元
主编助理：徐书令
印 刷 者：重庆科情印务有限公司
总发行处：重庆市邮政局
订 购 处：全国各地邮政局
国外总发行：中国国际图书贸易总公司（北京 399 信箱）
国外代号：6210－M0