10
Solaris 破解入门

很长一段时间以来，Solaris 主要支持高端的 Web 和数据库服务。尽管 Solaris 有 Interl 发行版，但绝大多数的 Solaris 还是运行在 SPARC 平台之上。我们在本章将把精力放在 SPARC 上的 Solaris，而它也是影响深远的操作系统之一。Solaris 在以前被称为 SunOS，当然这样的称呼已渐渐被大家遗忘了，现在常见的版本是 2.6，7，8，和 9。

当其它的操作系统倾向于在默认安装情况下只提供基本服务时，Solaris 9 仍提供了大量的网络服务，例如，默认安装的 Solaris 9 启用近 20 个 RPC 服务。在以前，RPC 服务里涌现了大量的漏洞；现在，在网络上可以获得大量的 Solaris 源代码，似乎预示着在 RPC 里可能会发现更多的漏洞。

从历史上说，几乎所有的 Solaris RPC 服务都出现过漏洞（sadmind，cmsd，statd，automount 通过 statd，snmXdmid，dmispd，cachefsd，等等），也发现了通过 inetd 使用的远程错误，如 elnetd，/bin/login（通过 telnetd 和 rshd），dtspcd，lpd 等。Solaris 在缺省状态下有大量带 setuid 位的文件，因此，在正式使用 Solaris 前，应该对它进行仔细地加固。

当然，Solars 也内置了一些安全功能，包括进程记帐、审计、和可选的 non-executable 栈。从管理员的立场来看，启用这个选项是值得的，因为它提供了一定程度上的保护。

10.1 SPARC 体系结构介绍

Scalable Processor Architecture（SPARC）是广泛使用的硬件平台，对 Solaris 的支持非常好。它最初由 Sun Microsystems 开发，后来逐渐演变成一个开放的标准。它最早有两个版本（v7 和 v8），都是 32 位的，当然，最新的版本（v9）是 64 位。SPARC v9 处理器在传统的低效率运行模式中，可以运行 64 位及 32 位程序。

UltraSPARC 处理器源自 Sun Microsystems 的 SPARC v9，具有运行 64 位程序的能力；除此之外的 CPU 实际上都是来自 Sun 的 SPARC v7 或 v8，仅在 32 位模式下运行程序。Solaris 7，8 和 9 都支持 64 位内核，可以运行 64 位用户模式的程序；然而，大多数用户模式的程序是以 32 位运行的。

SPARC 处理器有 32 个通用寄存器，随时可以使用。其中一些有特殊用途，剩下的由编译器分配或由程序员自由处理。我们一般把 32 个寄存器分成四类：全局寄存器，局部寄存器，输入和输出寄存器。

实际上，SPARC 体系结构是 big-endian，意味着首先在内存里用最有效的字节表示整数和指针。它的指令长度是固定的，都是 4 字节长；所有的指令以 4 字节为界进行对齐，任何在不对齐的地址上执行代码，都会导致 BUS 错误；同样，读写任何未对齐的地址，也会导致 BUS 错误并引起程序崩溃。

10.1.1 寄存器和寄存器窗口

SPARC CPU 可使用的寄存器总数可以改变，但它们分成了固定数量的寄存器窗口。一个寄存器窗口是函数使用的一组寄存器。当前寄存器窗口的指针通过 save 和 restore 指令来增加或减少。典型的，函数在开始和结束时会执行这两条指令。

save 指令保存当前的寄存器窗口，使系统分配一组新的寄存器，restore 指令丢弃当前的寄存器窗口，恢复前面保存的数据。我们可以用 save 指令为局部变量保留栈空间，用 restore 函数释放局部栈空间。

寄存器	用途
%g0	Always zero
%g1	Temporary storage
%g2	Global variable 1
%g3	Global variable 2
%g4	Global variable 3
%g5	Reserved
%g6	Reserved
%g7	Reserved

表 10.1. 全局寄存器及其用途

无论是函数调用，还是执行 save 或 restore 指令，都不会影响到全局寄存器（%g0-%g7）。第一个全局寄存器%g0 永远是零。写入它的数据会被丢弃，任何以它为源寄存器的复制操作将把目标操作数设为零。除了%g0 之外，剩下的 7 个全局寄存器也各有用途，表 10.1.里有介绍。

局部寄存器（%l0-%l7）象它们名字暗示的那样，对于具体的函数来说是局部的。它们作为寄存器窗口的一部分被保存和恢复。局部寄存器没有特殊的用途，编译器可以随意使用。每个函数也都可以使用它们。

执行 save 时，输出寄存器（%o0-%o7）改写输入寄存器（%i0-%i7）。执行 restore 时，执行相反的操作，输入寄存器将改写输出寄存器。save 把前一个函数的输入寄存器作为寄存器窗口的一部分加以保存。

开始的 6 个输入寄存器（%i0-%i5）传入函数参数。它们作为%o0 至%o5 传递给函数，当执行 save 时，它们变成%i0 到%i5。当函数需要 6 个以上的参数时，额外的参数通过栈传递。函数的返回值存贮在%i0 里，执行 restore 时转为%o0。

%o6 寄存器和栈指针%sp 是同义词，而%i6 是帧指针%fp。Save 象前一个函数预期那样，把栈指针作为帧指针保存，restore 把保存的栈指针恢复到它原来的地方。

至今没有提及的 2 个通用寄存器—%o7 和%i7，用于保存返回地址。执行 call 后，返回地址保存在%o7。当执行 save 时，这个值毫无疑问会被复制到%i7，它保持直到执行一个返回和 restore。在这个值被复制到输入寄存器之后，%o7 就变成一个普通用途的寄存器了。总结输入/输出寄存器用途的列表见表 10.2.。

为了方便，在表 10.3.里和 10.4.里总结了 save 和 restore 的作用。

寄存器	用途
%i0	First incoming function argument, return value

%i1 - %i5	Second through sixth incoming function arguments
%i6	Frame pointer (saved stack pointer)
%i7	Return address
%o0	First outgoing function argument, return value from called function
%o1 - %o5	Second though sixth outgoing function arguments
%o6	Stack pointer
%o7	Contains return address immediately after call, otherwise general purpose

表 10.2. 寄存器名称和用途

1. Local registers (%l0 - %l7) are saved as part of a register window.
2. Input registers (%i0 - %i7) are saved as part of a register window.
3. Output registers (%o0 - %o7) become the input registers (%i0 - %i7).
4. A specified amount of stack space is reserved.

表 10.3. save 指令的作用

1. Input registers become output registers.
2. Original input registers are restored from a saved register window.
3. Original local registers are restored from a saved register window.
4. As a result of step one, the %sp (%o6) becomes %fp (%i6) releasing local stack space.

表 10.4. restore 指令的效果

对于 leaf 函数（那些不调用其它函数的函数），编译器可以生成不执行 save 或 restroe 的指令，省去这些操作带来的开销；但是系统不能改写输入或局部寄存器，必须在输出寄存器里访问参数。

任何确定的 SPARC CPU 都有固定数量的寄存器窗口。在它们可用的时候，用来保存保存的寄存器。当寄存器窗口用完后，最早的寄存器窗口被刷新，相关的数据压入栈。每条 save 指令至少在栈上保留 64 字节的空间，在必要时也会保存本地寄存器和输入寄存器的内容。当频繁发生上下文切换时，或者发生大量的陷阱或中断时，所有的寄存器窗口都有可能被刷新，从而把寄存器窗口中的数据压入栈。

10.1.2 延迟槽

和其它的体系结构类似，SPARC 在执行 branch，call，或 jump 时使用延迟槽。在程序执行过程中，有两个寄存器用来指定控制流； %pc 是程序计数器，指向当前的指令，%npc 指向将被执行的下一条指令。当执行 branch 或 call 时，目的地址被加载到%npc 而不是%pc。这导致在执行流被重定向到目的地址之前，branch/call 之后的指令被执行。

```
0x10004:    CMP %o0, 0
0x10008:    BE 0x20000
0x1000C:    ADD %o1, 1, %o1
```

```
0x10010:    MOV 0x10, %o1
```

在这个例子里，如果%o0 保存零，在 0x10008 的分支将被采用。然而，在采用这个分支前，0x1000c 处的指令被执行。如果这个分支在 0x10008 没有被采用， 0x1000c 处的指令仍被执行，执行流继续到 0x10010。如果一个分支被取消，例如 BE, A address，那么仅仅在采用这个分支时，才会执行延迟槽的指令。很多因素都会影响 SPARC 上的执行流程；然而，即使是为了写破解，也没有必要全部理解它们。

10.1.3 合成指令

SPARC 里的许多指令是由其它指令合成的，或者是其它指令的别名。因为所有的指令都是 4 个字节，所以，它要用两条指令把 32 位值加载到寄存器。更有趣的是，call 和 ret 都是合成的指令。call 更准确的形式是 jmpl address, %o7。jmpl 是一个连接的（linked）跳转，它把当前指令指针的值保存在目标操作数里。在 call 的例子里，目的操作数是寄存器%o7。ret 是 jmpl %i7+8, %g0，回到保存的返回地址上来。程序计数器的值被丢给%g0 寄存器，而它总是为零。

Leaf 函数用另外的合成指令—retl 返回。因为它们不必执行 save 或 restore，因此，返回地址在%o7 里。retl 是 jmpl %o7+8, %g0 的别名。

10.2 Solaris/SPARC Shellcode 基础

SPARC 上的 Solaris 和其它的 UNIX 类似，都有明确定义的系统调用接口。传统的 Solaris/SPARC 和其它的平台差不多，Shellcode 使用系统调用而不是调用库函数。网上有无数的 Solaris/SPARC Shellcode，大多都流传了 N 年，如果你只想拿来用或进行简单的修改，在网上肯定可以找到合适的；然而，如果你希望自己写 Shellcode，那么必须掌握本章所介绍的基础知识。

系统通过特殊的系统陷阱 8 开始系统调用。然而，SunOS 最初是用陷阱 0 开始系统调用的，只是最近的 Solaris 版本才改成陷阱 8。系统调用号通过全局寄存器%g1 指定。作为正常的函数参数，少于 6 个的系统调用参数都是通过输出寄存器%o0 到%o5 传递。大多数系统调用的参数一般都少于 6 个，但有些函数可能需要 6 个以上的参数，这时，一般是通过栈来传递这些额外的参数。

10.2.1 自定位和 SPARC Shellcode

许多 Shellcode 为了引用自身包含的字符串，需要在内存里定位自己的位置。通过作为代码的一部分——在运行时构造字符串，有可能避免这样做，但这样明显缺乏效率和可靠性。在 x86 上，通过 jump 和 call/pop 指令对可以轻松完成自定位。但在 SPARC 上，由于延迟槽的存在以及为了避免 Shellcode 里出现 Null 字节，所用的指令非常复杂。

下面的指令把 Shellcode 的地址载入寄存器%o7，这个方法工作得很好，在 SPARC

Shellcode 里使用多年了：

```
1. \x20\xbf\xff\xff // bn, a shellcode - 4

2. \x20\xbf\xff\xff// bn, a shellcode

3. \x7f\xff\xff\xff // call shellcode + 4

4. rest of shellcode
```

这个bn，是已经废除的 branch never 指令。换句话说，这些分支指令从来没被采用（branch never）。这意味着延迟槽总是被跳过。call 指令是真正的连接（linked）跳转，把当前指令计数器的值存贮在%o7 里。

上述指令执行的顺序是 1，3，4，2，4。

这段代码导致 call 的地址保存在%o7 里，使 Shellcode 可以定位它在内存里的字节串。

10.2.2 简单的 SPARC exec Shellcode

大部分 Shellcode 的最终目的是执行命令行 Shell，然后从 shell 里完成其它事情。下面介绍一些非常简单的 Shellcode，它们在 Solaris/SPARC 上执行/bin/sh。

在现代 Solaris 机器上，exec 系统调用的编号是 11，它需要两个参数，第一个是指向要执行的文件名的字符指针，第二个是一个 null-terminated 字符指针数组，用于指定文件参数。这两个参数分别保存在%o0 和%o1，系统调用编号保存在%g1。下面的 Shellcode 演示怎么做的。

```
static char scode[]=   "\x20\xbf\xff\xff"      // 1: bn,a scode - 4
                       "\x20\xbf\xff\xff"      // 2: bn,a scode
                       "\x7f\xff\xff\xff"      // 3: call scode + 4
                       "\x90\x03\xe0\x20"      // 4: add %o7, 32, %o0
                       "\x92\x02\x20\x08"      // 5: add %o0, 8, %o1
                       "\xd0\x22\x20\x08"      // 6: st %o0, [%o0 + 8]
                       "\xc0\x22\x60\x04"      // 7: st %g0, [%o1 + 4]
                       "\xc0\x2a\x20\x07"      // 8: stb %g0, [%o0 + 7]
                       "\x82\x10\x20\x0b"      // 9: mov 11, %g1
                       "\x91\xd0\x20\x08"      // 10: ta 8
                       "/bin/sh";             // 11: shell string
```

下面逐行解释这段代码：

1. 这段熟悉的代码把 Shellcode 的地址载入%o7。
2. 定位延续的载入代码。[Location loading code continued..]
3. 重复一次。
4. 把/bin/sh 的地址载入%o0；这是系统调用的第一个参数。
5. 把函数参数数组的地址载入%o1。这个地址位于/bin/sh 后面 8 个字节处，Shellcode

结尾后面 1 个字节处，是系统调用的第二个参数。

6. 用字符串/bin/sh 初始化参数数组（argv[0]）的第一个成员。
7. 把参数数组的第二个成员设为 NULL，终止数组（%g0 总是 NULL）。
8. 在正确位置写一个 NULL 字节，确保/bin/sh 字符串完全 NULL 终止。
9. 把系统调用编号载入%g1（11＝SYS_exec）。
10. 通过陷阱 8（ta＝trap）执行系统调用。
11. Shell 字符串。

10.2.3 Solaris 里面有用的系统调用

除 execv 外，Solaris 里还有几个系统调用可以使用，在/usr/include/sys/syscall.h 里面可以找到完整的列表。表 10.5.提供一个快速预览。

系统调用	编号
SYS_open	5
SYS_exec	11
SYS_dup	41
SYS_setreiud	202
SYS_setregid	203
SYS_so_socket	230
SYS_bind	232
SYS_listen	233
SYS_accept	234
SYS_connect	235

表 10.5. 有用的系统调用和相关编号

10.2.4 NOP 和填充指令

为了增加破解的可靠性和减少对精确地址的依赖性，在破解负载里使用填充指令是个不错的选择。但在大多数情况下，SPARC 的 NOP 指令实际上没什么用处，因为它包含三个 NULL 字节，在大多数基于字符串的溢出里不会被复制。但是，许多指令可以代替它，并且具有同样的效果。表 10.6.是一些例子。

Sparc 填充指令	字节序
sub %g1, %g2, %g0	"\x80\x20\x40\x02"
andcc %l7, %l7, %g0	"\x80\x8d\xc0\x17"
or %g0, 0xfff, %g0	"\x80\x18\x2f\xff"

表 10.6. NOP 替代物

10.3 Solaris/SPARC 栈帧介绍

Solaris/SPARC 对栈帧的组织和其它平台类似。栈象 Intel x86 那样，用于保存局部变量和寄存器中的数据（见表 10.7.），地址也是从大到小，依次减少。系统在栈上为 32 位二进制文件里的函数至少保留 96 个字节的空间，这些空间除了保存 8 个本地寄存器和 8 个输入寄存器外，还剩下 32 个字节；这 32 个字节用于保存返回的结构指针和参数的拷贝，以防止它们被寻址（如果指向它们的指针必须被传递给另外的函数）。对任何函数都这样组织栈帧，所以为局部变量保留的空间比为保存的寄存器保留的空间更靠近栈顶。这预防函数改写它自己保存的寄存器。

Top of stack - Higher memory addresses
Function 1
Space reserved for local variables
Size: Variable
Function 1
Space reserved for return structure
pointer and argument copies.
Size: 32 bytes
Function 1
Space reserved for saved registers
Size: 64 bytes
Bottom of stack - Lower memory addresses

表 10.7. Solaris 的内存管理

Solaris 的栈通常用于保存（populated）结构和数组，而不象 x86 平台那样还保存整数和指针。在大多数情况下，整数和指针保存在通用寄存器里，除非是参数的数量超出可用的寄存器，或者要求它们必须是可寻址的，才会把它们放到栈上。

10.4 栈溢出的方法

让我们看一些流行的 Solaris 栈溢出方法。在某些情况下，他们和 Intel IA32 的稍微有点不一样，但还是有很多共性。

10.4.1 任意的大小溢出

SPARC 下允许改写任意大小的栈溢出和 Intel x86 的有很多相似之处。最后的目标都是改写保存在栈上的指令指针，把执行流重定向到包含 Shellcode 的地址。然而，因为栈的组织形式，它可能只能改写调用函数保存的寄存器。最终的效果是它采用两个函数的最小值来

获取执行控制。

如果你考虑一个假设有栈溢出的函数，这个函数的返回地址保存在%i7 里。SPARC 的 ret 指令是由 jmpl %i7+8, %g0 合成的。延迟槽将典型的被 restore 指令填充。第一个 ret / restore 指令对将产生一个新值，这个值来自从保存的寄存器窗口恢复的%i7。如果这是从栈上恢复而不是从　　寄存器，已经作为溢出的一部分被改写了，那么第二个 ret 将导致代码执行攻击者选择的地址。

表 10.8.显示了栈上保存的 Solaris/SPARC 寄存器窗口信息。这个信息的组织形式和调试器（比如说 GDB）里输出的差不多。输入寄存器比局部寄存器更靠近栈顶。

%l0	%l1	%l2	%l3
%l4	%l5	%l6	%l7
%i0	%i1	%i2	%i3
%i4	%i5	%i6 (saved %fp)	%i7 (saved %pc)

表 10.8. 栈上保存的寄存器窗口布局

10.4.2 寄存器窗口和栈溢出的复杂性

任何 SPARC CPU 都有固定数量的　　寄存器窗口。SPARC v9 的 CPU 可以使用 2 到 32 个寄存器窗口。当 CPU 的寄存器窗口用完后，再执行 save 时，将产生窗口溢出陷阱，CPU 将刷新寄存器窗口的　　寄存器，把相关数据压入栈；在发生上下文切换或暂停线程时，寄存器窗口肯定会被刷新，数据入栈；系统调用通常也会刷新寄存器窗口，数据入栈。

在发生溢出时，如果你试图改写的寄存器窗口不在栈上，而是在 CPU 的寄存器里，你的破解显然不会成功。依据返回，保存的寄存器将不会从你在栈上改写的位置恢复，而是从　　寄存器。这将使试图改写保存的%i7 寄存器的攻击更加困难。

当缓冲区溢出的进程被调试时，它的行为不同于平时，因为调试器的停顿（break）将会刷新所有的寄存器窗口。如果你正在调试程序，并在溢出发生前停顿，可能会刷新寄存器窗口，从而导致其它的不再发生了。最常见的情形是，只有当 GDB 附上目标进程时，破解才能正常工作。这是因为没有调试器停顿时，寄存器窗口不会被刷新入栈，从而使改写没有效果。

10.4.3 其它复杂的因素

当寄存器压入栈时，%i7 是最后压入的。这意味着在典型的字符串溢出里，为了改写%i7，你首先要改写其它的寄存器。在最好的情形下，为了获取程序的执行控制，将需要一个额外的返回。然而，所有的本地和输入寄存器已经被溢出破坏了。最常见的情形是，寄存器包含被破坏的指令，如果这些指令是无效的，那么在关键函数返回之前，它们将引起访问违例或段失效（segmentation fault）。为了在个案的基础上评定这个情形，以及为不同于返回地址的寄存器确定适当的值，可能必需这样。

SPARC 上的帧指针必须以 8 字节为界对齐。如果改写一个帧指针，或者在溢出里改写多个保存的寄存器，在帧指针里对齐是基本的保护措施。在执行 restore 指令时，如果没有对齐帧指针，将引起 BUS 错误，从而导致程序崩溃。

10.4.4 可能的解决方法

即使第一个寄存器窗口没有保存在栈上，但仍有一些方法可以执行保存的%i7 的栈改写。如果攻击者可以多次尝试，将有可能尝试多次溢出，等待在合适的时刻发生上下文切换，从而导致被立刻刷新入栈。然而，这个方法不太可靠，因为并不是所有的溢出都可以重复利用。

对于靠近栈顶的函数，可以改写保存的寄存器。对任何确定的二进制文件，从一个栈帧到另一个栈帧之间的距离，通常是可以预计的和可以计算的。因此，如果第一个调用函数的寄存器窗口没有刷新入栈，或许在调用第二个或第三个函数时才会导致它被刷新入栈。然而，你企图改写的保存的寄存器越往调用树上面，为了得到控制需要更多的函数返回，防止程序由于栈恶化而崩溃将会变得更加困难。

在许多情况下，用两个返回改写第一个保存的寄存器窗口并执行代码是有可能的；然而，对破解来说，知道最坏的情况对我们有好处。

10.4.5 Off-By-One 栈溢出漏洞

在大多数情况下，SPARC 上的 Off-By-One 漏洞是不可破解的。Off-By-One 破解的原理主要是基于指针恶化。对于 Intel x86 上的破解，最明确的方法是改写保存的帧指针的最没意义的位，通常是栈上紧跟着局部变量的第一个地址。如果帧指针不是目标，另外的指针很有可能是。绝大部分的 Off-By-One 漏洞是由于 NULL 终止的原因，当剩余的缓冲区空间不够用时，通常导致一个 NULL 字节写到边界之外。

SPARC 用 big-endian 字节序表示指针。在 Off-By-One 例子里，最有意义的字节将被破坏，而不是改写保存在内存里的指针的最没意义的字节。相反，稍微改动指针高位的值，将导致指针的整个值发生巨大的变化。例如，当标准栈指针 0xFFBF1234 最有意义的字节被改写后，可能指向 0xBF1234。这个地址是无效的，除非堆非常大而扩展到那个地址。只有在可选择的情况下，这才是可行的。

除字节序问题外，Solaris/SPARC 上指针恶化的目标是受限的。它不可能到达帧指针，因为它（帧指针）保存在寄存器数组深处。它很可能是唯一可能被破坏的局部变量，或第一个保存的寄存器%l0。尽管必须对 Off-By-One 漏洞进行评估后才能做出正确判断，但对破解来说，SPARC 的 Off-By-One 栈溢出最多只提供了有限的可能性。

10.4.6 Shellcode 位置

必需寻找一个好的方法把执行流重定向到包含 Shellcode 的地址。Shellcode 可以放在一些地方，每个地方都有它的优缺点。选择把 shellcode 放在哪里，考虑最多的因素应该是可靠性，这通常由你正在破解的目标程序体现出来。

对于破解本地 setuid 程序来说，有可能完全控制目标程序的环境变量和参数。假若这样，把 shellcode 加上大量的填充物后注入环境变量是可能的，这样的话，在可预计的栈地址可

以找到 Shellcode，我们也可以非常可靠的完成破解任务。如果有可能的话，这应该是最好的选择。

在破解守护程序时，特别是远程守护程序时，在栈上寻找 Shellcode 并执行它仍是一个不错的选择。栈缓冲区的地址会因环境变量或程序的改变稍微有点改变，因此通常可以比较准确地预计。对可能只有一次机会的 exploit 来说，栈地址由于好的可预测性和较小的变化，从而成为不错的选择。

当在栈上找不到合适的缓冲区时，或栈被标为不可执行时，堆显然是第二选择。如果我们可以在 shellcode 周围注入大量的填充物，并把执行流程指向堆地址，它可以象栈缓冲区溢出那样可靠。然而，在大多数情况下，在堆上寻找 Shellcode 可能要尝试多次，要想可靠的工作，最好是用暴力猜测的方式重复尝试攻击。不可执行栈的系统不反对在堆上执行代码，所以，对破解加固后的系统来说，这是很好的选择。

在 Solaris/SPARC 上，返回 libc 的方法通常不太可靠，除非可以重复攻击，或者攻击者掌握目标系统函数库的具体知识。Solaris/SPARC 有多种版本的函数库，可能多于其它的商业操作系统，如 Windows；期望把 libc 载入特殊的基地址是不合理的，每个重要的 Solaris 发行版都可能有一打以上的 libc 版本。对本地攻击来说，返回 libc 的攻击因为可以仔细检查函数库，所以能可靠的完成。如果攻击者花时间为不同版本的函数库创建一个完整的函数地址的列表，返回 libc 的方法对于远程破解来说也许是可行的。

对于基于字符串的溢出（复制操作止于 NULL 字节）来说，通常不可能把执行流程重定向到主程序的可执行的数据部分。许多程序被载入基地址 0x00010000，这个地址的高位包含 NULL 字节。在某些情况下，把 Shellcode 注入函数库的数据部分是可能的；如果在栈或堆上存贮 Shellcode 不能可靠地完成破解，可以试一下这个。

10.5 实际的栈溢出破解

适当的演示可以使 Solaris/SPARC 上基于栈的破解更加通俗易懂。下面使用本章提到的方法，介绍在假定的 Solaris 应用程序里怎样破解栈溢出。

10.5.1 脆弱的程序

为了演示怎样破解栈溢出，我们专门写了这个脆弱的程序。它不太复杂，你可能会在真实的应用程序里发现它的身影；然而，它的确是一个好的起点。脆弱的代码如下：

```
int vulnerable_function( char *userinput)
    { char buf[64];
    strcpy( buf, userinput );
    return 1;
}
```

在这个例子里，userinput 是通过命令行传递的第一个参数。注意，这个程序在退出前有两次返回，从而给了我们破解的可能性。