数组初始化 –memset or {0}

秦新良

2015 年 1 月 26 日

目　录

1 缘起

平时在编码或代码检视的时候，有时候会听到这样的说法：数组初始化直接用{0}，不要调用memset，这样执行效率高，如代码1所示。

```
1 void func()
2 {
3     char array[1024] = {0};
4 }
```

代码 1: {0}初始化

但有时候又会听到截然不同的另外一种说法：数组初始化用memset，这样比直接用{0}初始化效率高很多，如代码2所示。

```
1 void func()
2 {
3     char array[1024];
4
5     memset(array, '\0', sizeof(array));
6 }
```

代码 2: memset初始化

当只有一种观点的时候，一般会相信其的正确性[1]。但当两种完全相反的观点同时存在时，不得不上人对孰是孰非产生一个大大的疑问，所以就有了把两种代码编译后的结果分析一下的想法。

2 从汇编说起

2.1 环境信息

本文实验所用的环境为Linux + GCC，如图1所示。

2.2 代码分析

代码1和代码2编译后的结果如下所示：

```
1 push   %ebp
```

```
1 push   %ebp
```

[1]因为自己不清楚，既然有人清楚是怎么回事，也就相信是这么回事了。

```
vampire@vampire:~$ uname -a
Linux vampire 3.2.0-29-generic #46-Ubuntu SMP Fri Jul 27 17:04:05 UTC 2012 i686
i686 i386 GNU/Linux
vampire@vampire:~$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/i686-linux-gnu/4.6/lto-wrapper
Target: i686-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu/Linaro 4.6.3-1ubu
ntu5' --with-bugurl=file:///usr/share/doc/gcc-4.6/README.Bugs --enable-languages
=c,c++,fortran,objc,obj-c++ --prefix=/usr --program-suffix=-4.6 --enable-shared
--enable-linker-build-id --with-system-zlib --libexecdir=/usr/lib --without-incl
uded-gettext --enable-threads=posix --with-gxx-include-dir=/usr/include/c++/4.6
--libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-li
bstdcxx-debug --enable-libstdcxx-time=yes --enable-gnu-unique-object --enable-pl
ugin --enable-objc-gc --enable-targets=all --disable-werror --with-arch-32=i686
--with-tune=generic --enable-checking=release --build=i686-linux-gnu --host=i686
-linux-gnu --target=i686-linux-gnu
Thread model: posix
gcc version 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5)
```

图 1: 环境信息

```
2  mov     %esp,%ebp
3  push    %edi
4  push    %ebx
5  sub     $0x410,%esp
6  mov     %gs:0x14,%eax
7  mov     %eax,-0xc(%ebp)
8  xor     %eax,%eax
9  lea     -0x40c(%ebp),%ebx
10
11 mov     $0x0,%eax
12 mov     $0x100,%edx
13 mov     %ebx,%edi
14 mov     %edx,%ecx
15 rep stos %eax,%es:(%edi)
16 mov     -0xc(%ebp),%eax
17 xor     %gs:0x14,%eax
18 je      8048441 <func+0x3d>
19 call    8048320 <
       __stack_chk_fail@plt>
20 add     $0x410,%esp
21 pop     %ebx
22 pop     %edi
23 pop     %ebp
24 ret
```

```
2  mov     %esp,%ebp
3  push    %edi
4  push    %ebx
5  sub     $0x410,%esp
6  mov     %gs:0x14,%eax
7  mov     %eax,-0xc(%ebp)
8  xor     %eax,%eax
9  lea     -0x40c(%ebp),%eax
10 mov     %eax,%ebx
11 mov     $0x0,%eax
12 mov     $0x100,%edx
13 mov     %ebx,%edi
14 mov     %edx,%ecx
15 rep stos %eax,%es:(%edi)
16 mov     -0xc(%ebp),%eax
17 xor     %gs:0x14,%eax
18 je      8048443 <func+0x3f>
19 call    8048320 <
       __stack_chk_fail@plt>
20 add     $0x410,%esp
21 pop     %ebx
22 pop     %edi
23 pop     %ebp
24 ret
```

可以看出，编译后的两份代码只在第 9、10 行有一点差异[2]，其余部分完全相同。而且还有一个明显的特点是：代码2虽然调用了 memset 来初始化数组，但在编译后的代码

[2] 其实功能完全相同。

里根本就没有调用 memset 来初始化[3]。

既然代码一样，那我们只拿一份来分析说明，如下所示：

```
1  push   %ebp                    ; save previous stack frame address
2  mov    %esp,%ebp               ; save new stack frame address to
       register ebp
3  push   %edi
4  push   %ebx                    ; saving register ebx, edi on stack
5  sub    $0x410,%esp             ; allocate 0x410 bytes space on
       stack
6  mov    %gs:0x14,%eax
7  mov    %eax,-0xc(%ebp)         ; protection for stack overflow
8  xor    %eax,%eax               ; reset register eax to 0
9  lea    -0x40c(%ebp),%ebx       ; beginning address of array
10
11 mov    $0x0,%eax               ; set register eax to 0
12 mov    $0x100,%edx             ; size of the array(1024)
13 mov    %ebx,%edi               ; beginning address to register edi
14 mov    %edx,%ecx               ; ecx's value is 0x100
15 rep stos %eax,%es:(%edi)       ; loop to initialize the array
16 mov    -0xc(%ebp),%eax
17 xor    %gs:0x14,%eax           ; stack overflow detect
18 je     8048441 <func+0x3d>
19 call   8048320 <__stack_chk_fail@plt>
20 add    $0x410,%esp             ; restore esp
21 pop    %ebx                    ; restore ebx
22 pop    %edi                    ; restore edi
23 pop    %ebp                    ; pop the previous stack frame's
       address to ebp
24 ret                            ; return
```

[3]如果有调用的话，汇编代码里应该有一行 call memset 的指令。

第 1 和 2 行是每个函数开始执行的例行公事：保存上一个函数[4]调用栈的栈基址，然后将本函数的栈基址保存到%ebp寄存器，如图??所示。与之对应的是第 23 和 24 行，首先将调用者的栈基址重新载入%ebp寄存器，然后返回。

因为栈的操作基本都是基于栈基址[5]偏移来实现的，所以调用者在调用一个函数的前后，其基址是不能改变的，但被调用者同样也会使用寄存器%ebp来操作其堆栈，所以每个函数在使用%ebp前，必需先保存寄存器的当前值，在返回前将其 restore。

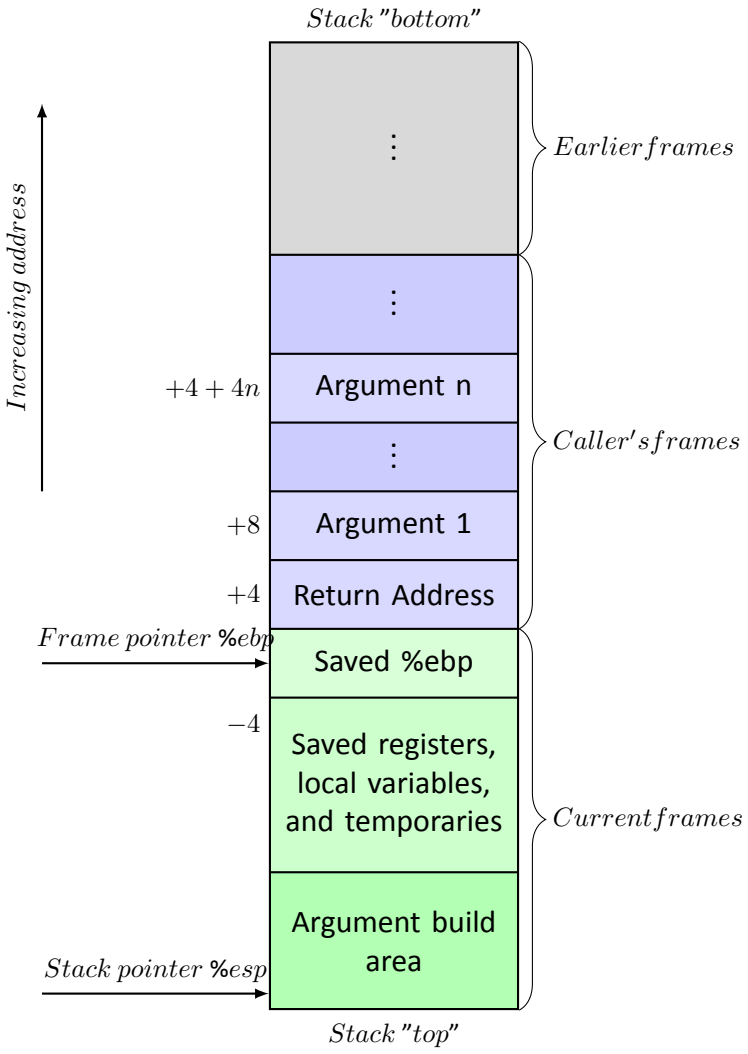

图 2: 函数调用栈

寄存器%edi、%ebx和%esi是被调用者保存寄存器[6]，即如果当前函数需要使用这些寄存器，必须先将寄存器当前的内容压栈，然后在返回前 restore。第 3、4 和第 21、22

[4]调用该函数的函数，即调用者（caller）。

[5]%ebp 寄存器的值。

[6]与之对应的是调用者保存寄存器：%eax、%edx 和%ecx。

行就是完成这个工作的。

第 5 行是为当前调用栈申请栈空间。紧拉着第 6 和 7 行是编译器为缓冲区保护自动插入的代码，其基本原理是：在栈的某个地址[7]保存一个随机值，然后在函数返回前，查看该地址的值是否被修改，如果被修改，则说明栈被破坏了，直接抛出异常（17–19 行）。

紧接着 8 到 15 行就是对数组的初始化。第 9 行先将数组的首地址存到寄存器%ebx 中；接着第 11 行重置寄存器%eax；第 12 行将数组的元素个数存入寄存器%edx；第 13、14 行是将数组的元素个数存入寄存器%ecx 中，作为循环初始化数组的循环变量；最后第 15 行循环初始化数组，每次初始化 4 字节，一共循环 256 次直到寄存器%ecx 的值变为 0。

2.3 循环还是 memset

从上一小节对编译后的代码分析可以知道，无论是 memset 还是 {0}，最终都是通过循环每次初始化 4 字节完成的。那是不是无论 memset 还是 {0}，最终都被编译成了每次 4 字节的循环初始化呢？让我们把代码1和2中的数组大小调整到 10K 看一下编译后的结果。如下所示：

```
1  push   %ebp
2  mov    %esp,%ebp
3  sub    $0x2828,%esp
4  mov    %gs:0x14,%eax
5  mov    %eax,-0xc(%ebp)
6  xor    %eax,%eax
7  lea    -0x280c(%ebp),%eax
8  mov    $0x2800,%edx
9  mov    %edx,0x8(%esp)
10 movl   $0x0,0x4(%esp)
11 mov    %eax,(%esp)
12 call   8048370 <memset@plt>
13 mov    -0xc(%ebp),%eax
14 xor    %gs:0x14,%eax
15 je     8048478 <func+0x44>
16 call   8048340 <
       __stack_chk_fail@plt>
17 leave
18 ret
```

```
1  push   %ebp
2  mov    %esp,%ebp
3  sub    $0x2828,%esp
4  mov    %gs:0x14,%eax
5  mov    %eax,-0xc(%ebp)
6  xor    %eax,%eax
7  lea    -0x280c(%ebp),%eax
8  mov    $0x2800,%edx
9  mov    %edx,0x8(%esp)
10 movl   $0x0,0x4(%esp)
11 mov    %eax,(%esp)
12 call   8048370 <memset@plt>
13 mov    -0xc(%ebp),%eax
14 xor    %gs:0x14,%eax
15 je     8048478 <func+0x44>
16 call   8048340 <
       __stack_chk_fail@plt>
17 leave
18 ret
```

[7]该地址的值在函数正常执行的情况下不会被修改。

可以看出，调整大小后的数组，无论是memset还是{0}初始化，最终都是通过调用memset完成的。网上查了一些资料，原来这是编译器的优化策略，对于小数组，直接循环初始化，大数组使用memset初始化。那数组达到多大的时候编译器会自动使用memset呢？在我的机子上测试的结果是大于8K，但在公司的单板服务器上测的结果是大于64个字节。这可能跟编译器版本及平台有关，这里不再继续深入分析。

3 结论

数组初始化，memset和{0}完全等价。