Hack 虚拟内存系列（五）：栈，寄存器和汇编

lm0963 / 2018-08-10 17:31:18 / 浏览数 2565 技术文章 技术文章 顶(0) 踩(0)

本文翻译自：Hack the virtual memory: the stack, registers and assembly code

Hack the virtual memory: the stack, registers and assembly code

栈

正如我们在第2章中看到的，栈位于内存的高端并向下增长。但它如何正常工作？它如何转换为汇编代码？使用的寄存器是什么？在本章中，我们将详细介绍栈的工作原理，

一旦我们理解了这一点，我们就可以尝试劫持程序的执行流程。

*注意：我们将仅讨论用户栈，而不是内核栈

前提

为了完全理解本文，你需要知道：

- C语言的基础知识（特别是指针部分）

环境

所有脚本和程序都已经在以下系统上进行过测试：

- Ubuntu
  - Linux ubuntu 4.4.0-31-generic #50~14.04.1-Ubuntu SMP Wed Jul 13 01:07:32 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
- 使用的工具:
  - gcc
    - gcc (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4
  - objdump
    - GNU objdump (GNU Binutils for Ubuntu) 2.2

下文讨论的所有内容都适用于此系统/环境，但在其他系统上可能会有所不同

自动分配

首先让我们看一个非常简单的程序，只有一个函数，并在函数中使用了单个变量（0-main.c）：

```
#include <stdio.h>

int main(void)
{
   int a;

   a = 972;
   printf("a = %d\n", a);
   return (0);
}
```

让我们编译这个程序，并使用objdump反汇编它：

```
holberton$ gcc 0-main.c
holberton$ objdump -d -j .text -M intel
```

main函数反汇编生成的代码如下：

```
000000000040052d <main>:
 40052d:       55                      push   rbp
 40052e:       48 89 e5                mov    rbp,rsp
 400531:       48 83 ec 10             sub    rsp,0x10
 400535:       c7 45 fc cc 03 00 00    mov    DWORD PTR [rbp-0x4],0x3cc
 40053c:       8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
 40053f:       89 c6                   mov    esi,eax
 400541:       bf e4 05 40 00          mov    edi,0x4005e4
```

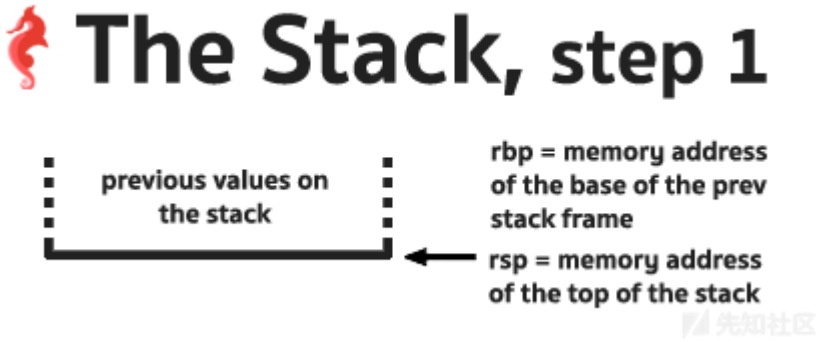
```
 400546:       b8 00 00 00 00          mov    eax,0x0
 40054b:       e8 c0 fe ff ff          call   400410 <printf@plt>
 400550:       b8 00 00 00 00          mov    eax,0x0
 400555:       c9                      leave
 400556:       c3                      ret
 400557:       66 0f 1f 84 00 00 00    nop    WORD PTR [rax+rax*1+0x0]
 40055e:       00 00
```

现在先关注前三行：

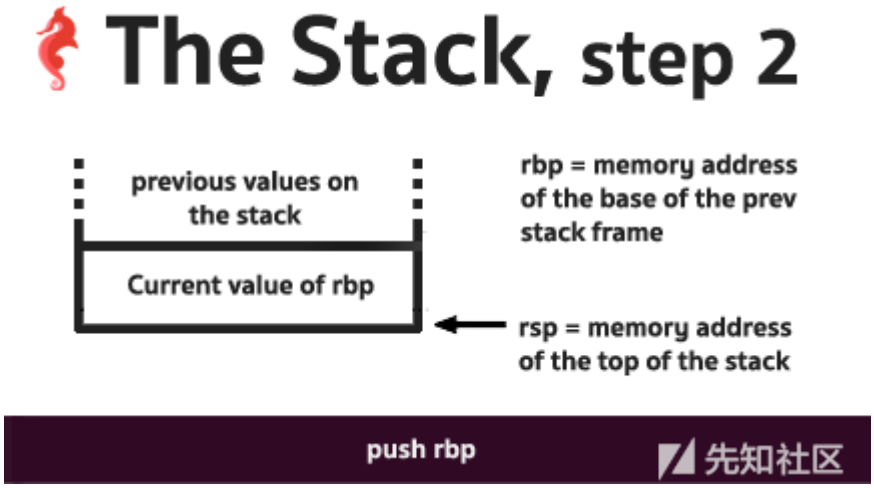
```
000000000040052d <main>:
 40052d:       55                      push   rbp
 40052e:       48 89 e5                mov    rbp,rsp
 400531:       48 83 ec 10             sub    rsp,0x10
```

main函数的第一行引用了rbp和rsp;这些是有特殊用途的寄存器。rbp是基址指针寄存器，指向当前栈帧的底部，rsp是栈指针寄存器，指向当前栈帧的顶部。

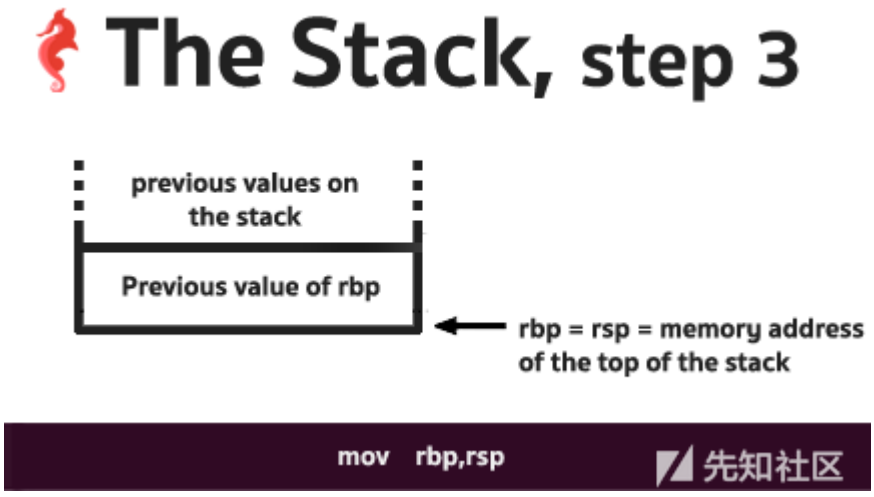
让我们逐步分析这句汇编语句。这是main函数第一条指令运行之前栈的状态：



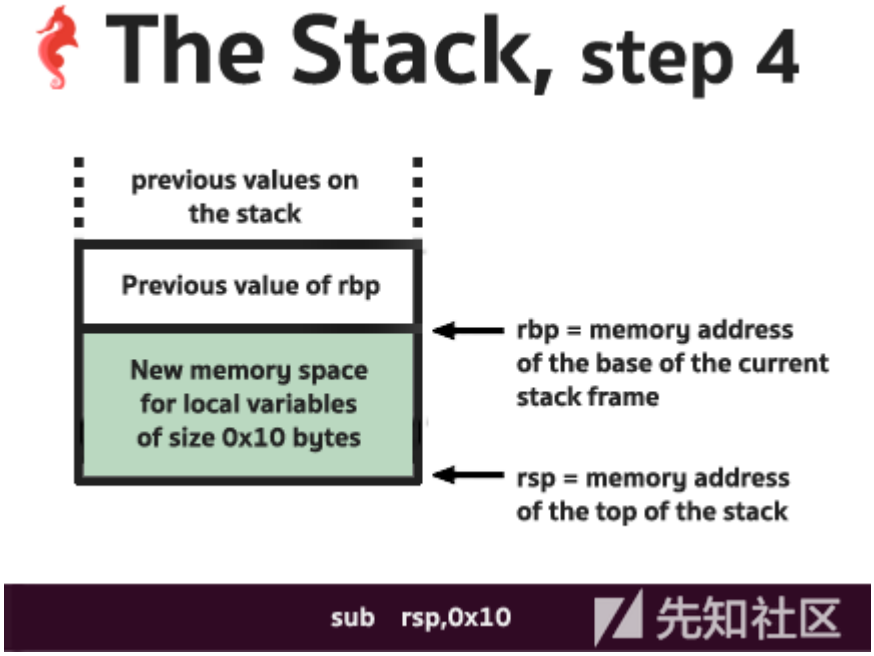
push rbp指令将寄存器rbp的值压入栈中。因为它"压入"到栈上，所以现在rsp的值是栈新的顶部的内存地址。栈和寄存器看起来像这样：



mov rbp，rsp将rsp的值复制到rbp中 -> rpb和rsp现在都指向栈的顶部



sub rsp，0x10预留一些空间（rbp和rsp之间的空间）来存储局部变量。请注意，此空间足以存储我们的整型变量



我们刚刚在内存中为局部变量创建了一个空间 —— 在栈中。这个空间被称为栈帧。每个具有局部变量的函数都将使用栈帧来存储自己的局部变量。

使用局部变量

main函数的第四行汇编代码如下：

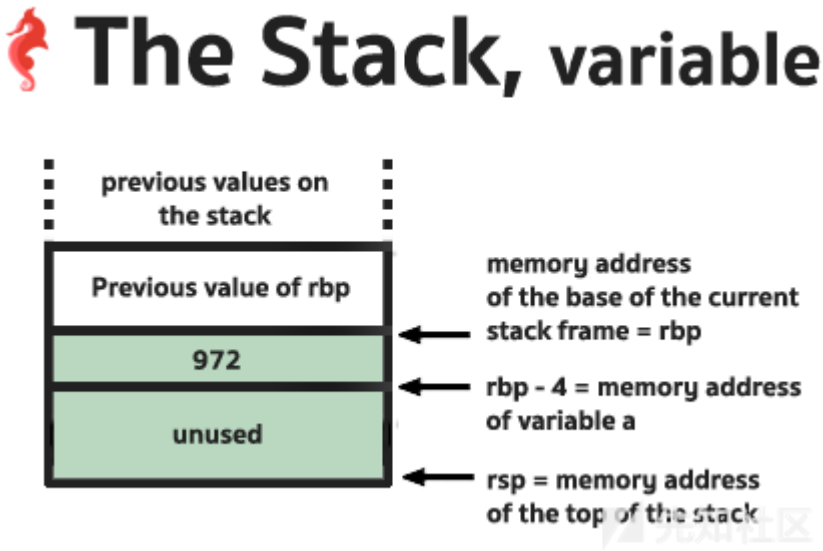
```
400535:       c7 45 fc cc 03 00 00    mov    DWORD PTR [rbp-0x4],0x3cc
```

0x3cc的十进制值为972。这一行对应于我们的C代码：

```
a = 972;
```

mov DWORD PTR [rbp-0x4]，0x3cc 将rbp-4地址处的内存赋值为972。[rbp -
4]是我们的局部变量a。计算机实际上并不知道我们在代码中使用的变量名称，它只是引用栈上的内存地址。

执行此指令后的栈和寄存器状态：

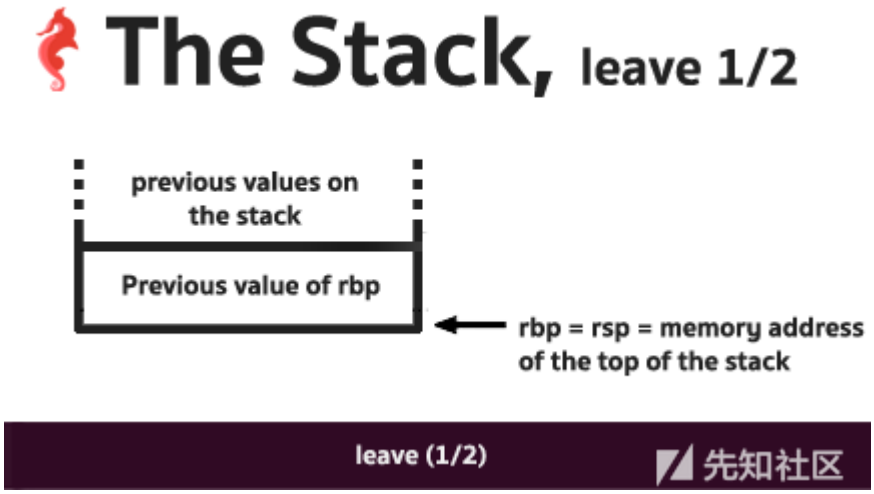


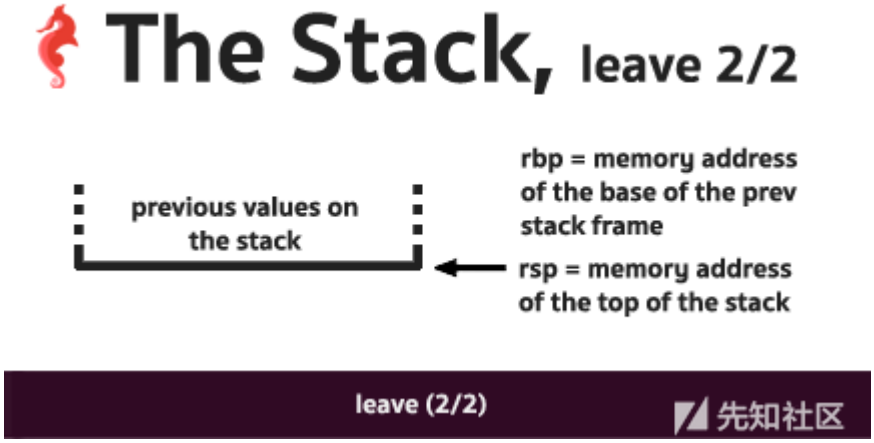
函数返回，自动释放

如果查看一下函数的末尾代码，我们会发现：

```
400555:       c9                      leave
```

此指令将rsp设置为rbp，然后将栈顶弹出赋值给rbp。





因为我们在函数的开头处将rbp的先前值压入栈，所以rbp现在设置为rbp的先前值。这就是：

- 局部变量被“释放”
- 在离开当前函数之前，先恢复上一个函数的栈帧。

栈和寄存器rbp和rsp恢复到刚进入main函数时的状态。

Playing with the stack

当变量从栈中自动释放时，它们不会被完全“销毁”。它们的值仍然存在内存中，并且这个空间可能会被其他函数使用。
所以在编写代码时要初始化变量，否则，它们将在程序运行时获取栈中的任何值。
让我们考虑以下C代码（1-main.c）：

```
#include <stdio.h>

void func1(void)
{
    int a;
    int b;
    int c;

    a = 98;
    b = 972;
    c = a + b;
    printf("a = %d, b = %d, c = %d\n", a, b, c);
}

void func2(void)
{
    int a;
    int b;
    int c;

    printf("a = %d, b = %d, c = %d\n", a, b, c);
```

```
}

int main(void)
{
   func1();
   func2();
   return (0);
}
```

正如所见，func2函数并没有初始化局部变量a，b和c，如果我们编译并运行此程序，它将打印...

```
holberton$ gcc 1-main.c && ./a.out
a = 98, b = 972, c = 1070
a = 98, b = 972, c = 1070
holberton$
```

...与func1函数输出的变量值相同！这是因为栈的工作原理。这两个函数以相同的顺序声明了相同数量的变量（具有相同的类型）。他们的栈帧完全相同。当func1结束时，其
—— 只有rsp递增。
因此，当我们调用func2时，它的栈帧位于与前一个func1栈帧完全相同的位置，当我们离开func1时，func2的局部变量具有与func1的局部变量相同的值。

让我们检查反汇编代码来证明上述：

```
holberton$ objdump -d -j .text -M intel

000000000040052d <func1>:
 40052d:       55                      push   rbp
 40052e:       48 89 e5                mov    rbp,rsp
 400531:       48 83 ec 10             sub    rsp,0x10
 400535:       c7 45 f4 62 00 00 00    mov    DWORD PTR [rbp-0xc],0x62
 40053c:       c7 45 f8 cc 03 00 00    mov    DWORD PTR [rbp-0x8],0x3cc
 400543:       8b 45 f8                mov    eax,DWORD PTR [rbp-0x8]
 400546:       8b 55 f4                mov    edx,DWORD PTR [rbp-0xc]
 400549:       01 d0                   add    eax,edx
 40054b:       89 45 fc                mov    DWORD PTR [rbp-0x4],eax
 40054e:       8b 4d fc                mov    ecx,DWORD PTR [rbp-0x4]
 400551:       8b 55 f8                mov    edx,DWORD PTR [rbp-0x8]
 400554:       8b 45 f4                mov    eax,DWORD PTR [rbp-0xc]
 400557:       89 c6                   mov    esi,eax
 400559:       bf 34 06 40 00          mov    edi,0x400634
 40055e:       b8 00 00 00 00          mov    eax,0x0
 400563:       e8 a8 fe ff ff          call   400410 <printf@plt>
 400568:       c9                      leave
 400569:       c3                      ret

000000000040056a <func2>:
 40056a:       55                      push   rbp
 40056b:       48 89 e5                mov    rbp,rsp
 40056e:       48 83 ec 10             sub    rsp,0x10
 400572:       8b 4d fc                mov    ecx,DWORD PTR [rbp-0x4]
 400575:       8b 55 f8                mov    edx,DWORD PTR [rbp-0x8]
 400578:       8b 45 f4                mov    eax,DWORD PTR [rbp-0xc]
 40057b:       89 c6                   mov    esi,eax
 40057d:       bf 34 06 40 00          mov    edi,0x400634
 400582:       b8 00 00 00 00          mov    eax,0x0
 400587:       e8 84 fe ff ff          call   400410 <printf@plt>
 40058c:       c9                      leave
 40058d:       c3                      ret

000000000040058e <main>:
 40058e:       55                      push   rbp
 40058f:       48 89 e5                mov    rbp,rsp
 400592:       e8 96 ff ff ff          call   40052d <func1>
 400597:       e8 ce ff ff ff          call   40056a <func2>
 40059c:       b8 00 00 00 00          mov    eax,0x0
 4005a1:       5d                      pop    rbp
 4005a2:       c3                      ret
 4005a3:       66 2e 0f 1f 84 00 00    nop    WORD PTR cs:[rax+rax*1+0x0]
 4005aa:       00 00 00
 4005ad:       0f 1f 00                nop    DWORD PTR [rax]
```

正如所见，栈帧的形成方式始终是一致的。在两个函数中，栈帧的大小是相同的，因为他们的局部变量是一样的。

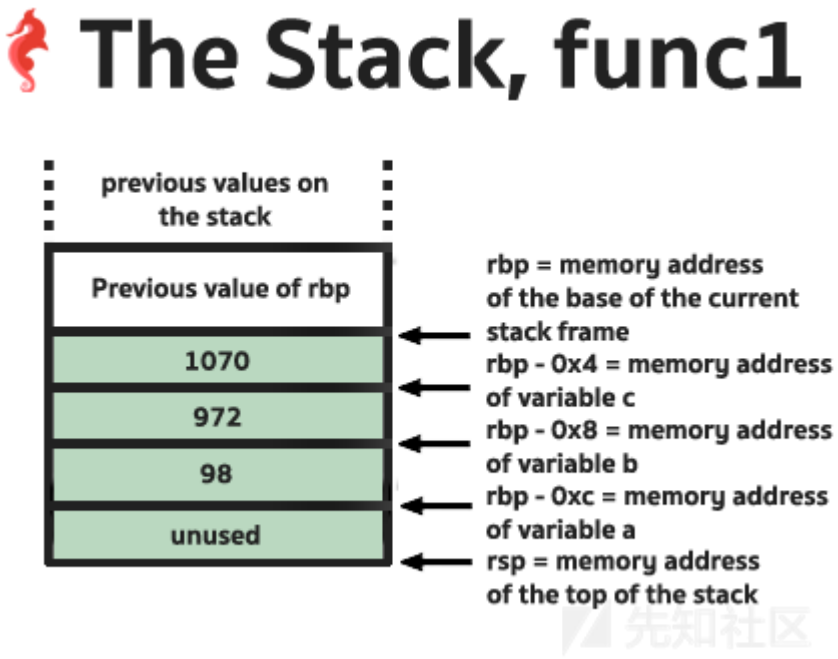
```
push   rbp
mov    rbp,rsp
sub    rsp,0x10
```

两个函数都以leave语句结束。
变量a，b和c在两个函数中都以相同的方式引用：

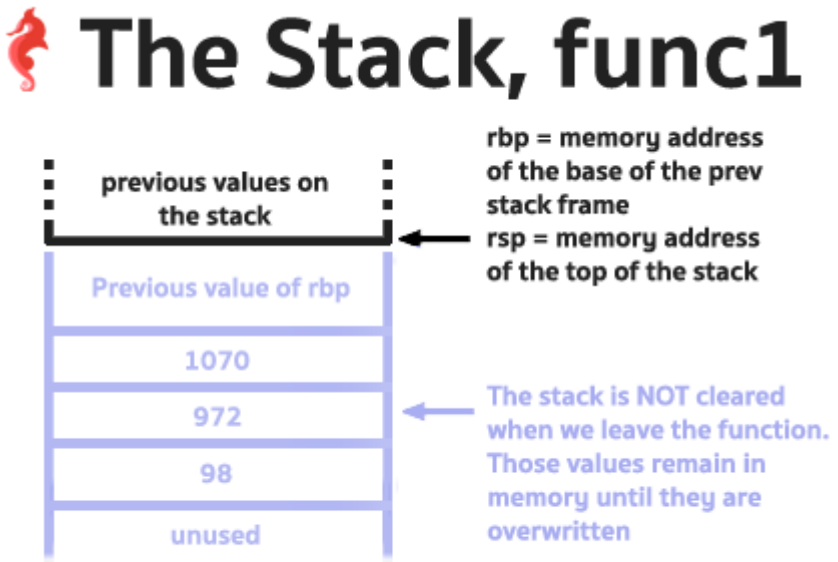
- a 位于内存地址rbp - 0xc
- b 位于内存地址rbp - 0x8
- c 位于内存地址rbp - 0x4

请注意，栈中这些变量的顺序与我们代码中的顺序不同。编译器按需要对它们进行排序，因此永远不要假设栈中的局部变量的顺序。

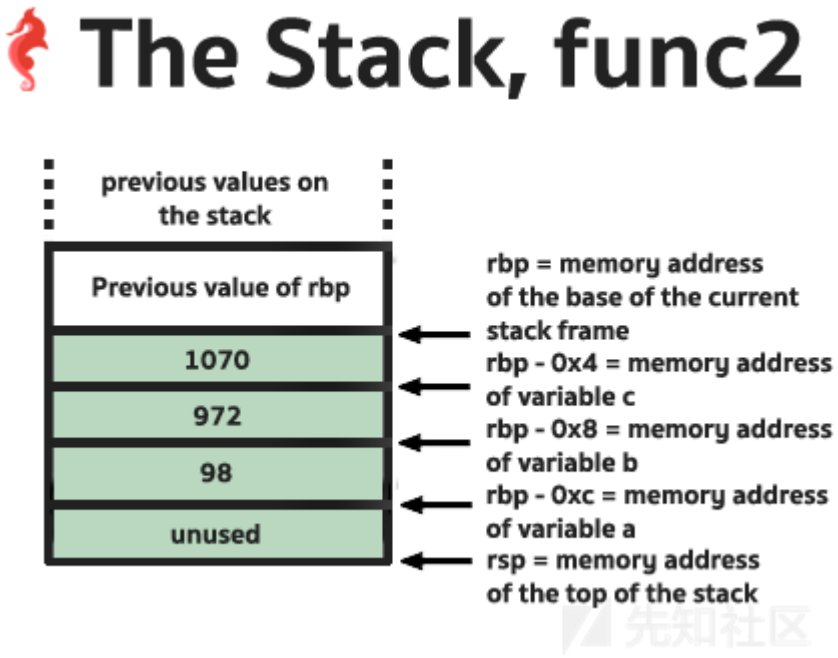
所以，这是我们离开func1之前的栈和寄存器rbp和rsp的状态：



当我们离开func1函数时，leave已经执行; 如前所述，这是栈，rbp和rsp在返回到main函数之前的状态：



因此，当进入func2时，局部变量被设置为栈中的任何内容，这就是它们的值与func1函数的局部变量相同的原因。



ret

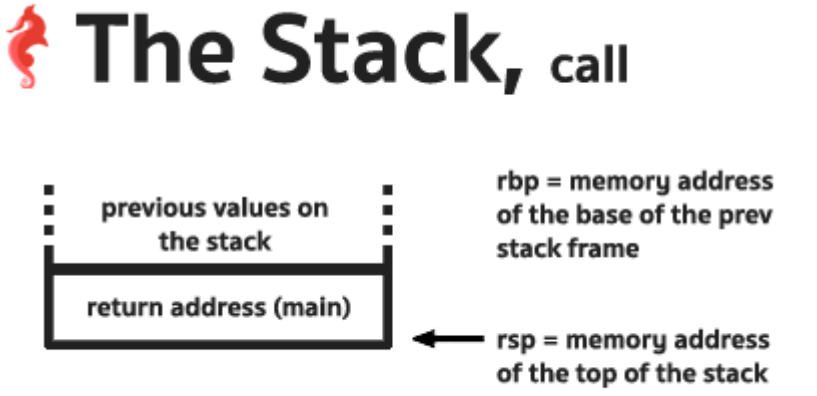
你可能已经注意到我们所有的示例函数都以ret指令结束。ret从栈中弹出返回地址并跳转到该处。当函数被调用时，程序使用call指令来压入返回地址，然后跳转到被调用函数
这就是程序如何调用函数并从被调用函数返回到调用函数接着执行其下一条指令的原理。

所以这意味着栈上不仅有变量，还有指令的内存地址。让我们重新审视我们的1-main.c代码。

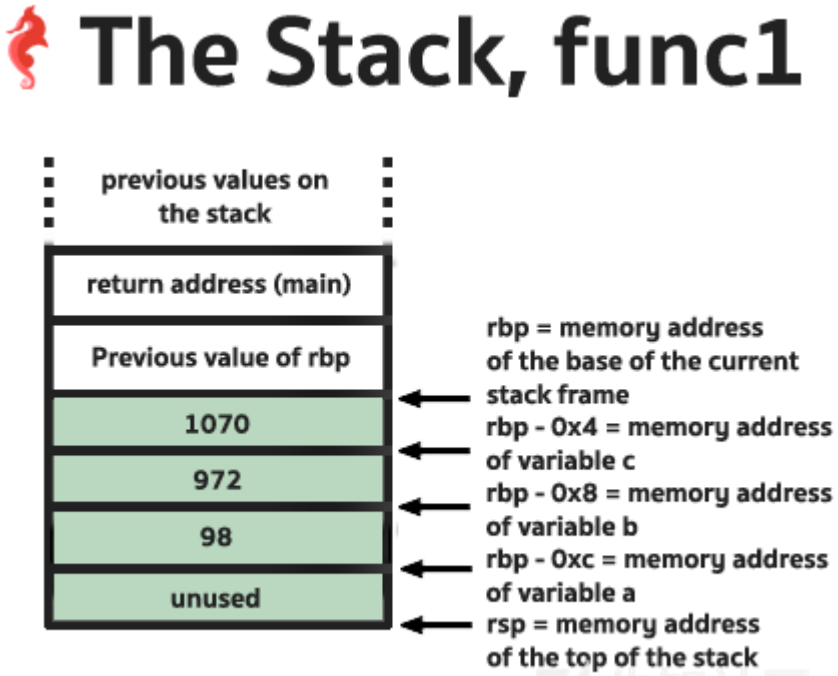
当main函数调用func1时，

```
400592:       e8 96 ff ff ff          call   40052d <func1>
```

它将下一条指令的内存地址压入栈中，然后跳转到func1。
因此，在执行func1中的指令之前，栈顶部包含此地址，既rsp指向此值。



在形成func1的栈帧之后，栈看起来像这样：



综上所述

根据我们刚刚学到的东西，我们可以直接使用rbp访问所有局部变量（不需要使用C变量！），以及栈中保存的rbp值和函数的返回地址。

在C语言中，可以这样写：

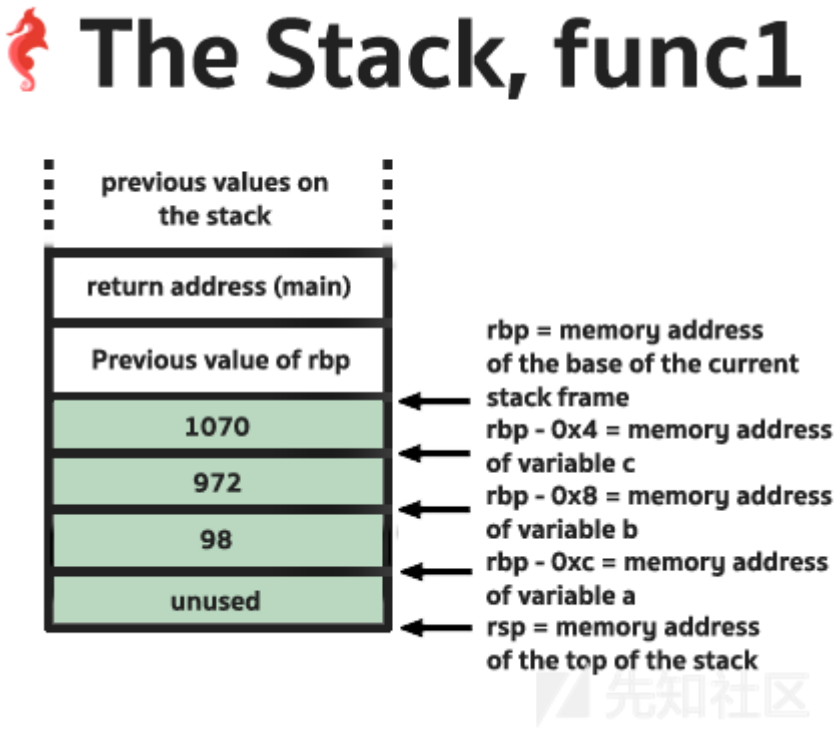
```
register long rsp asm ("rsp");
   register long rbp asm ("rbp");
```

程序2-main.c代码：

```
#include <stdio.h>

void func1(void)
{
   int a;
   int b;
   int c;
   register long rsp asm ("rsp");
   register long rbp asm ("rbp");

   a = 98;
   b = 972;
   c = a + b;
   printf("a = %d, b = %d, c = %d\n", a, b, c);
   printf("func1, rpb = %lx\n", rbp);
   printf("func1, rsp = %lx\n", rsp);
   printf("func1, a = %d\n", *(int *)(((char *)rbp) - 0xc) );
   printf("func1, b = %d\n", *(int *)(((char *)rbp) - 0x8) );
   printf("func1, c = %d\n", *(int *)(((char *)rbp) - 0x4) );
   printf("func1, previous rbp value = %lx\n", *(unsigned long int *)rbp );
   printf("func1, return address value = %lx\n", *(unsigned long int *)((char *)rbp + 8) );
}

void func2(void)
{
   int a;
   int b;
   int c;
   register long rsp asm ("rsp");
   register long rbp asm ("rbp");
```

```
   printf("func2, a = %d, b = %d, c = %d\n", a, b, c);
   printf("func2, rpb = %lx\n", rbp);
   printf("func2, rsp = %lx\n", rsp);
}

int main(void)
{
   register long rsp asm ("rsp");
   register long rbp asm ("rbp");

   printf("main, rpb = %lx\n", rbp);
   printf("main, rsp = %lx\n", rsp);
   func1();
   func2();
   return (0);
}
```

获取变量的值



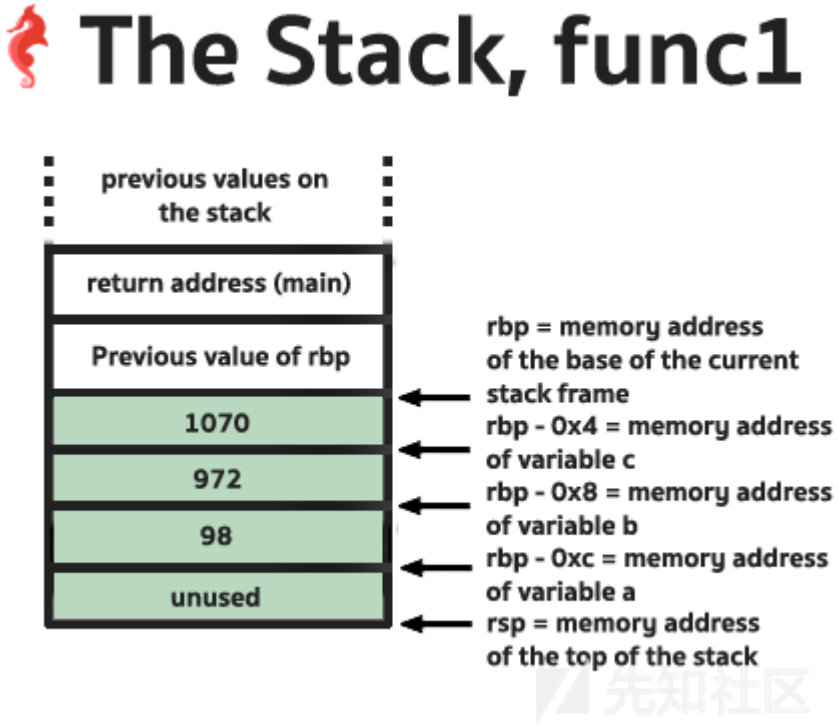
在之前的发现中，我们知道变量是通过rbp - 0xX的方式引用的：

- a is at rbp - 0xc
- b is at rbp - 0x8
- c is at rbp - 0x4

因此，为了获得这些变量的值，我们需要解引用rbp。对于变量a：

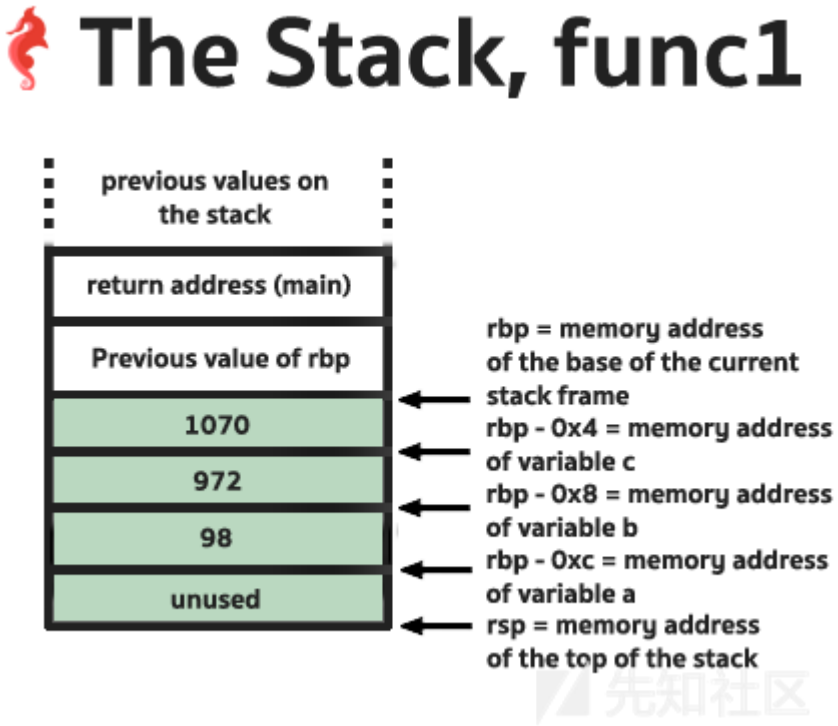
- 将变量rbp转换为char *：（char *）rbp
- 减去正确的字节数以获取变量在内存中的地址：（char *）rbp - 0xc
- 再次将它转换为指向int的指针，因为a是int类型变量：（int *）（（（char *）rbp） - 0xc）
- 解引用以获取此地址内存中的值：*（int *）（（（char *）rbp） - 0xc）

保存的rbp值



查看上图，当前rbp直接指向保存的rbp，因此我们只需将变量rbp转换为指向unsigned long int的指针并解引用它：*（unsigned long int *）rbp。

返回地址值



返回地址值就在栈上保存的先前rbp之前。rbp是8个字节长，所以我们只需要在rbp的当前值上加8，就获得该返回值在栈上的地址。这是我们的做法：

- 将变量rbp转换为char *：（char *）rbp
- 将8加到此值上：（（char *）rbp + 8）
- 将它转换为指向unsigned long int的指针:(unsigned long int *）（（char *）rbp + 8）
- 解引用以获取此地址内存中的值：*（unsigned long int *）（（char *）rbp + 8）

程序输出

```
holberton$ gcc 2-main.c && ./a.out
main, rpb = 7ffc78e71b70
main, rsp = 7ffc78e71b70
a = 98, b = 972, c = 1070
```

```
func1, rpb = 7ffc78e71b60
func1, rsp = 7ffc78e71b50
func1, a = 98
func1, b = 972
func1, c = 1070
func1, previous rbp value = 7ffc78e71b70
func1, return address value = 400697
func2, a = 98, b = 972, c = 1070
func2, rpb = 7ffc78e71b60
func2, rsp = 7ffc78e71b50
holberton$
```

可以看到：

- 从func1我们可以通过rbp正确访问所有变量
- 从func1我们可以得到main函数的rbp
- 证实了func1和func2确实具有相同的rbp和rsp值
- rsp和rbp之间的差异是0x10，如汇编代码中所示（sub rsp，0x10）
- 在main函数中，rsp == rbp，因为main函数中没有局部变量

func1的返回地址是0x400697。让我们通过反汇编程序来验证一下。如果我们是正确的，那么这应该是在main函数中调用func1之后的指令的地址。

```
holberton$ objdump -d -j .text -M intel | less

0000000000400664 <main>:
 400664:       55                      push   rbp
 400665:       48 89 e5                mov    rbp,rsp
 400668:       48 89 e8                mov    rax,rbp
 40066b:       48 89 c6                mov    rsi,rax
 40066e:       bf 3b 08 40 00          mov    edi,0x40083b
 400673:       b8 00 00 00 00          mov    eax,0x0
 400678:       e8 93 fd ff ff          call   400410 <printf@plt>
 40067d:       48 89 e0                mov    rax,rsp
 400680:       48 89 c6                mov    rsi,rax
 400683:       bf 4c 08 40 00          mov    edi,0x40084c
 400688:       b8 00 00 00 00          mov    eax,0x0
 40068d:       e8 7e fd ff ff          call   400410 <printf@plt>
 400692:       e8 96 fe ff ff          call   40052d <func1>
 400697:       e8 7a ff ff ff          call   400616 <func2>
 40069c:       b8 00 00 00 00          mov    eax,0x0
 4006a1:       5d                      pop    rbp
 4006a2:       c3                      ret
 4006a3:       66 2e 0f 1f 84 00 00    nop    WORD PTR cs:[rax+rax*1+0x0]
 4006aa:       00 00 00
 4006ad:       0f 1f 00                nop    DWORD PTR [rax]
```

yes! \o/

Hack the stack!

既然我们知道了返回地址在栈上的位置，那么我们修改这个值会怎么样？
我们是否可以改变程序的流程并使func1返回到其他地方？在程序中添加一个名为bye的新函数（3-main.c）：

```
#include <stdio.h>
#include <stdlib.h>

void bye(void)
{
   printf("[x] I am in the function bye!\n");
   exit(98);
}

void func1(void)
{
   int a;
   int b;
   int c;
   register long rsp asm ("rsp");
   register long rbp asm ("rbp");
```

```
   a = 98;
   b = 972;
   c = a + b;
   printf("a = %d, b = %d, c = %d\n", a, b, c);
   printf("func1, rpb = %lx\n", rbp);
   printf("func1, rsp = %lx\n", rsp);
   printf("func1, a = %d\n", *(int *)(((char *)rbp) - 0xc) );
   printf("func1, b = %d\n", *(int *)(((char *)rbp) - 0x8) );
   printf("func1, c = %d\n", *(int *)(((char *)rbp) - 0x4) );
   printf("func1, previous rbp value = %lx\n", *(unsigned long int *)rbp );
   printf("func1, return address value = %lx\n", *(unsigned long int *)((char *)rbp + 8) );
}

void func2(void)
{
   int a;
   int b;
   int c;
   register long rsp asm ("rsp");
   register long rbp asm ("rbp");

   printf("func2, a = %d, b = %d, c = %d\n", a, b, c);
   printf("func2, rpb = %lx\n", rbp);
   printf("func2, rsp = %lx\n", rsp);
}

int main(void)
{
   register long rsp asm ("rsp");
   register long rbp asm ("rbp");

   printf("main, rpb = %lx\n", rbp);
   printf("main, rsp = %lx\n", rsp);
   func1();
   func2();
   return (0);
}
```

让我们看看这个函数的代码位于哪里：

```
holberton$ gcc 3-main.c && objdump -d -j .text -M intel | less

00000000004005bd <bye>:
 4005bd:       55                      push   rbp
 4005be:       48 89 e5                mov    rbp,rsp
 4005c1:       bf d8 07 40 00          mov    edi,0x4007d8
 4005c6:       e8 b5 fe ff ff          call   400480 <puts@plt>
 4005cb:       bf 62 00 00 00          mov    edi,0x62
 4005d0:       e8 eb fe ff ff          call   4004c0 <exit@plt>
```

现在让我们在func1函数中替换栈上的返回地址为bye函数的起始地址，4005bd（4-main.c）：

```
#include <stdio.h>
#include <stdlib.h>

void bye(void)
{
   printf("[x] I am in the function bye!\n");
   exit(98);
}

void func1(void)
{
   int a;
   int b;
   int c;
   register long rsp asm ("rsp");
   register long rbp asm ("rbp");
```

```
   a = 98;
   b = 972;
   c = a + b;
   printf("a = %d, b = %d, c = %d\n", a, b, c);
   printf("func1, rpb = %lx\n", rbp);
   printf("func1, rsp = %lx\n", rsp);
   printf("func1, a = %d\n", *(int *)(((char *)rbp) - 0xc) );
   printf("func1, b = %d\n", *(int *)(((char *)rbp) - 0x8) );
   printf("func1, c = %d\n", *(int *)(((char *)rbp) - 0x4) );
   printf("func1, previous rbp value = %lx\n", *(unsigned long int *)rbp );
   printf("func1, return address value = %lx\n", *(unsigned long int *)((char *)rbp + 8) );
   /* hack the stack! */
   *(unsigned long int *)((char *)rbp + 8) = 0x4005bd;
}

void func2(void)
{
   int a;
   int b;
   int c;
   register long rsp asm ("rsp");
   register long rbp asm ("rbp");

   printf("func2, a = %d, b = %d, c = %d\n", a, b, c);
   printf("func2, rpb = %lx\n", rbp);
   printf("func2, rsp = %lx\n", rsp);
}

int main(void)
{
   register long rsp asm ("rsp");
   register long rbp asm ("rbp");

   printf("main, rpb = %lx\n", rbp);
   printf("main, rsp = %lx\n", rsp);
   func1();
   func2();
   return (0);
}
```

```
holberton$ gcc 4-main.c && ./a.out
main, rpb = 7fff62ef1b60
main, rsp = 7fff62ef1b60
a = 98, b = 972, c = 1070
func1, rpb = 7fff62ef1b50
func1, rsp = 7fff62ef1b40
func1, a = 98
func1, b = 972
func1, c = 1070
func1, previous rbp value = 7fff62ef1b60
func1, return address value = 40074d
[x] I am in the function bye!
holberton$ echo $?
98
holberton$
```

bye函数成功执行了，并没有显示调用它:)

上一篇：Flask debug pin安全问题 下一篇：检测并实现绕过DBMS_ASSERT

1. 0 条回复

- 动动手指，沙发就是你的了！

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板