第5章 软件漏洞挖掘

软件漏洞挖掘是指通过分析软件中存在的安全漏洞来发现并利用这些漏洞，从而实现数据泄露、获取系统权限等攻击目标。软件漏洞挖掘涵盖汇编、C语言、Python编程、GDB调试、操作系统等方面的知识。在阅读本章之前，读者需具备上述基础知识和一定的自学能力，以便完成本章所提供实例的漏洞利用。本章将向读者介绍软件漏洞挖掘的基础知识，根据软件漏洞挖掘的分类，重点介绍栈溢出、格式化字符串、堆溢出原理，并针对每种类型的漏洞，给出实例和利用脚本。

本章学习目标：

- 学习软件漏洞挖掘的原理。
- 学会栈溢出漏洞的利用方式。
- 学会格式化字符串漏洞的利用方式。
- 学会堆溢出漏洞的利用方式。

5.1 软件漏洞挖掘基本概念

5.1.1 程序内存分布及区段介绍

1. 32位系统内存分布

32位操作系统下，进程空间分为用户空间和内核空间。用户空间包含代码段、数据段、BSS段、堆、内存映射段、栈等，如图5-1所示。

内核空间表示运行在处理器最高级别的超级用户模式(Supervisor Mode)下的代码或数据，占用从0xC0000000到0xFFFFFFFF的1GB线性地址空间。内核线性地址空间由所有进程共享，但只有运行在内核态的进程才能访问。用户进程可以通过系统调用切换到内核态，进程运行在内核态时所产生的地址都属于内核空间。

用户空间占用从0x00000000到0xBFFFFFFF共3GB的线性地址空间。每个进程都有一个独立的3GB用户空间，且用户空间由每个进程独有。内核线程没有用户空间，因为它不产生用户空间地址。另外，子进程共享(继承)父进程的用户空间时，只使用与父进程相同的用户线性地址到物理内存地址的映射关系，而不共享父进程用户空间。运行在用户态和内核态的进程都可以访问用户空间。用户空间内存还可以继续细分为以下6个部分。

图 5-1　32 位内存分布

(1) 代码段。

一般始于地址 0x08048000(编译时确定),存放程序编译好的二进制代码,通常为只读。

(2) 数据段。

存放在编译阶段(而非运行时)就能确定的数据,可读可写。数据段即通常所说的静态存储区,存放已赋初值的全局变量、Static 声明的静态变量以及常量。

(3) BSS 段。

存放已定义但未赋初值的全局变量和静态变量,可读可写。

(4) 堆。

存放进程运行中动态分配的内存段。堆大小并不固定,可动态扩张或缩减。当进程调用 malloc 等函数分配内存时,新分配的内存就被动态添加到堆上(堆被扩张);当利用 free 等函数释放内存时,被释放的内存从堆中被剔除(堆被缩减)。堆的生长方向是向上的,即向着内存地址增大的方向。

(5) 内存映射段。

(6) 栈。

存放参数变量和局部变量，由系统进行申请和释放，属于静态内存分配。栈的生长方向是向下的，即向着内存地址减小的方向。

特别注意，栈的地址总是比堆的地址高，栈和堆中间还有一段区域用于文件映射、mmap 堆动态分布等使用。

2. 64 位系统内存分布

对于 Linux 64 位系统，理论上，内存地址可用空间为 0x0000000000000000～0xFFFFFFFFFFFFFFFF(16 位十六进制数)，这是个相当庞大的空间，不过 Linux 64 位操作系统仅使用低 47 位(256T)、高 17 位做扩展(只能是全 0 或全 1)。所以，实际仅用到 0x0000000000000000～0x00007FFFFFFFFFFF 的用户空间和 0xFFFF800000000000～0xFFFFFFFFFFFFFFFF 的内核空间，其余的都是未用空间。同 32 位系统，用户空间也由代码段、数据段、BSS 段、堆、内存映射段、栈组成。

5.1.2　常用寄存器及其作用

1. 32 位寄存器

32 位寄存器如图 5-2 所示。

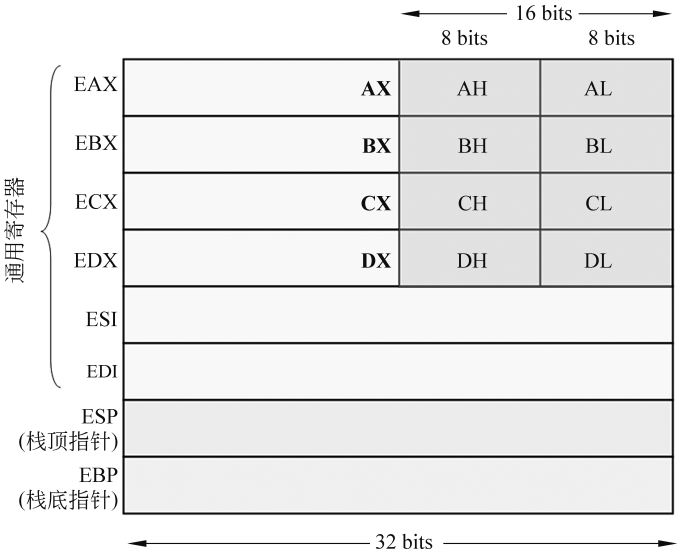

图 5-2　32 位寄存器

(1) 数据寄存器。

数据寄存器主要用来保存操作数和运算结果等信息，从而节省读取操作数占用总线和访问存储器的时间。

32 位 CPU 有 4 个 32 位的通用寄存器 EAX、EBX、ECX 和 EDX。对低 16 位数据的存取，不会影响高 16 位的数据。这些低 16 位寄存器分别命名为 AX、BX、CX 和 DX，它们和先前的 CPU 中的寄存器相一致。

4 个 16 位寄存器又可分割成 8 个独立的 8 位寄存器(AX：AH-AL、BX：BH-BL、CX：CH-CL、DX：DH-DL)，每个寄存器都有自己的名称，可独立存取。程序员可利用数据寄存

器的这种“可分可合”的特性，灵活地处理字/字节的信息。

寄存器 AX 和 AL 通常称为累加器(Accumulator)，用累加器进行的操作可能需要更少时间。累加器可用于乘、除、输入/输出等操作，它们的使用频率很高；

寄存器 BX 称为基地址寄存器(Base Register)，它可作为存储器指针来使用；

寄存器 CX 称为计数寄存器(Count Register)，在循环和字符串操作时，要用它来控制循环次数；在位操作中，当移多位时，要用 CL 来指明移位的位数；

寄存器 DX 称为数据寄存器(Data Register)。在进行乘、除运算时，它可作为默认的操作数参与运算，也可用于存放 I/O 的端口地址。

在 16 位 CPU 中，AX、BX、CX 和 DX 不能作为基址和变址寄存器来存放存储单元的地址，但在 32 位 CPU 中，32 位寄存器 EAX、EBX、ECX 和 EDX 不仅可以传送数据、暂存数据保存算术逻辑运算结果，而且也可以作为指针寄存器，所以，32 位寄存器更具有通用性。

(2) 变址寄存器。

寄存器 ESI、EDI、SI 和 DI 统称为变址寄存器(Index Register)，其中，ESI、EDI 为 32 位系统所有，低 16 位对应先前 CPU 中的 SI 和 DI。对低 16 位数据的存取，不影响高 16 位的数据。变址寄存器不可分割成 8 位寄存器。

变址寄存器主要用于存放存储单元在段内的偏移量，支持多种存储器操作数的寻址方式，为以不同的地址形式访问存储单元提供方便。变址寄存器还可作一般的存储器指针使用。在字符串操作指令的执行过程中，对它们有特定的要求，而且还具有特殊的功能。此外，变址寄存器也作为通用寄存器使用，存储算术逻辑运算的操作数和运算结果。

(3) 指针寄存器。

EBP、ESP、BP 和 SP 统称为指针寄存器(Pointer Register)，其中，EBP 和 ESP 为 32 位系统所有，其低 16 位对应先前 CPU 中的 BP 和 SP。对低 16 位数据的存取，不影响高 16 位的数据。指针寄存器不可分割成 8 位寄存器。

指针寄存器主要用于存放堆栈内存储单元的偏移量，支持多种存储器操作数的寻址方式，为以不同的地址形式访问存储单元提供方便。EBP 和 BP 为栈底指针(Base Pointer)寄存器，用于直接存取栈中的数据；ESP 和 SP 为栈顶指针(Stack Pointer)寄存器，用于访问栈顶数据。此外，指针寄存器也可作为通用寄存器，存储算术逻辑运算的操作数和运算结果。

(4) 指令指针寄存器。

EIP、IP 统称为指令指针寄存器(Instruction Pointer)，其中，EIP 为 32 位系统所有，低 16 位与先前 CPU 中的 IP 作用相同。指令指针寄存器存放下次将要执行的指令在代码段的偏移量。在具有预取指令功能的系统中，下次要执行的指令通常已被预取到指令队列中，除非发生转移情况。所以，在理解它们的功能时，不需要考虑存在指令队列的情况。

在实模式下，每个段的最大范围为 64K，所以 EIP 中的高 16 位为 0，此时，相当于只用其低 16 位的 IP 来反映程序中指令的执行次序。

(5) 标志寄存器。

标志寄存器包含多个状态标志位，以记录计算机运行过程中的不同状态信息，主要的标志位有：零标志位(ZF)，当运算结果为零时，该标志位被置位(设置为 1)，否则被清零(设置

为 0)；进位标志位(CF)，在无符号数加减运算中，当运算结果需要进位或借位时，该标志位被置位，否则被清零；溢出标志位(OF)，在有符号数加减运算中，当运算结果超出了所能表示的范围时，该标志位被置位，否则被清零；符号标志位(SF)，当运算结果为负数时，该标志位被置位，否则被清零；奇偶标志位(PF)，当运算结果中 1 的个数为偶数时，该标志位被置位，否则被清零。

2. 64 位寄存器

相较于 32 位系统，64 位系统寄存器的设计有较大变化。首先，数量不同。64 位系统有 16 个寄存器，32 位系统只有 8 个寄存器。32 位系统前 8 个寄存器都有不同的命名，分别是 EXX；而 64 位系统前 8 个寄存器使用了 R 代替 E，也就是 RXX。E 开头的寄存器命名依然可以直接应用于相应寄存器的低 32 位。剩下的寄存器名则是从 R8～R15，其低位分别用 D、W、B 指定长度，如表 5-1 所示。

表 5-1　64 位寄存器列表

64-bit	32-bit	16-bit	低 8-bit
RAX	EAX	AX	AL
RBX	EBX	BX	BL
RCX	ECX	CX	CL
RDX	EDX	DX	DL
RSI	ESI	SI	SIL
RDI	EDI	DI	DIL
RBP	EBP	BP	BPL
RSP	ESP	SP	SPL
R8	R8D	R8W	R8B
R9	R9D	R9W	R9B
R10	R10D	R10W	R10B
R11	R11D	R11W	R11B
R12	R12D	R12W	R12B
R13	R13D	R13W	R13B
R14	R14D	R14W	R14B
R15	R15D	R15W	R15B

其次，功能用法不同，32 位系统使用栈帧作为函数参数的保存位置，64 位系统使用寄存器 RDI、RSI、RDX、RCX、R8、R9 作为第 1～6 个参数的保持位置，RAX 作为返回值，当超过 6 个参数后，才会使用栈来作为函数参数的保存位置。32 位系统用 EBP 作为栈帧指针，64 位系统取消了这个设定，RBP 作为通用寄存器使用。64 位系统支持一些形式的以 PC 相关的寻址，而 32 位系统只有在 jmp 时才会用到这种寻址方式，如表 5-2 所示。

表 5-2　64 位寄存器功能列表

寄存器	功　能	寄存器	功　能
RDI	第一个参数	RAX	通常用于存储函数调用返回值
RSI	第二个参数	RSP	栈顶指针，指向栈的顶部
RDX	第三个参数	RBX	数据存储，遵循 Callee Save 原则
RCX	第四个参数	RBP	数据存储，遵循 Callee Save 原则
R8	第五个参数	R10～R11	数据存储，遵循 Caller Save 原则
R9	第六个参数	R12～R15	数据存储，遵循 Callee Save 原则

5.1.3　常用工具命令介绍

1. Ubuntu

本书中的环境是 ubuntu18.04 版本，建议在 Intel CPU 架构下运行。部分漏洞利用的其他环境将在代码中备注。

2. pwndbg

linux 调试工具，需要先安装 gdb，再安装 pwndbg 插件。

3. python

本书中的 Python 脚本使用 Python2.7 版本，建议使用 miniconda 或 virtualenv 来控制版本。

4. objdump

反汇编工具，若未预装，通过 apt-get install binutils 安装。

5. pwntools

pwntools 是 Python 的一个库，用来编写利用脚本，本文使用 3.1.1 版本。

6. libcsearcher

根据泄露地址匹配 libc 版本。

7. ghidra

开源静态分析工具。

5.2　栈溢出

5.2.1　栈的概念及特点

栈(Stack)是一种特殊的线性表，其所有的插入和删除均限定在表的一端进行，允许插入和删除的一端称为栈顶(Top)，不允许插入和删除的一端称为栈底(Bottom)。栈结构如图 5-3 所示，若给定一个栈 S=(a1，a2，a3，…，an)，则称 a1 为栈底元素，an 为栈顶元素，元素 ai 位于元素 ai-1 之上。栈中元素按 a1，a2，a3，…，an 的次序进栈，依 an，an－1，…，a1 的

次序出栈，即栈中元素按后进先出的原则进行，这是栈结构的重要特征。因此，栈又称为后进先出(Last In First Out，LIFO)表。

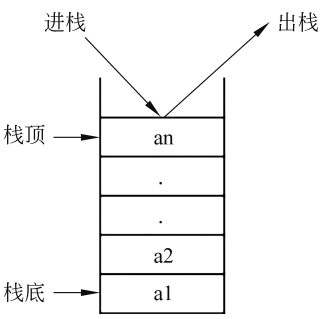

图 5-3　栈结构

通常，栈操作主要有以下 5 种。

(1) 在使用栈之前，需要建立一个空栈，称建栈；

(2) 往栈顶加入一个新元素，称进栈(压栈)；

(3) 删除栈顶元素，称出栈(退栈、弹出)；

(4) 查看当前的栈顶元素，称读栈(注意与出栈的区别 0)；

(5) 在使用栈的过程中，还要不断测试栈是否为空或已满，称为测试栈。

栈是机器系统提供的数据结构，计算机会在底层分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，因此栈有快速高效的特性。栈空间分静态分配和动态分配两种。静态分配是编译器完成的，例如自动变量(auto)的分配。动态分配由 alloca 函数完成。栈动态分配后无须手动释放(自动释放)，也就没有释放函数。为保证程序的可移植性，栈的动态分配操作是不被鼓励的。

因建栈后空间大小固定，不恰当的进栈或出栈操作可能引起栈“溢出”。如果一个栈已经为空，但用户还继续做出栈操作，则会出现栈的“下溢”；如果一个栈已经满了，用户还继续做进栈操作，则会出现栈的“上溢”。

5.2.2　调用者规则

调用者规则包括一系列操作，描述如下所示。

(1) 通常情况下，由于被调用的子程序会修改寄存器 EAX、ECX、EDX，为了在调用子程序完成之后能正确执行，调用者必须在调用子程序之前将这些寄存器的值入栈，如果子程序不会修改寄存器的话，调用者也可省略这一步骤。同理，如果子程序会修改其他寄存器，则也需入栈保存。

(2) 在调用子程序之前，按照从最后一个参数向前的顺序将参数入栈。

(3) 执行 call 指令，将返回地址压栈，并进入子程序的指令执行(子程序的执行将按照被调用者的规则执行)。

(4) 当子程序返回时，调用者期望找到子程序保存在 EAX 中的返回值。为了恢复调用子程序执行之前的状态，调用者需清除栈中的参数，以及将栈中保存的 EAX 值、ECX 值以及 EDX 值出栈，恢复 EAX、ECX、EDX 的值(如果其他寄存器在调用之前需要保存，也需要完成出栈操作)。

接下来，在 pwndbg 中运行一个包含函数调用的程序来展示调用者和被调用者的规则。为避免不同调试环境产生不同的地址，这里不提供运行程序及源码，直接根据调试过程来说明调用者和被调用者的规则。

```
pwndbg> r
Starting program: /root/tmp/5.2.3

Breakpoint 1, 0x0804857d in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
```

```
6.  -----------[ REGISTERS / show-flags off / show-compact-regs off ]-----------
7.   EAX  0x0
8.   * EBX  0x804a000 (_GLOBAL_OFFSET_TABLE_) —▸ 0x8049f08 (_DYNAMIC) ◂— add dword ptr
    [eax], eax
9.   * ECX  0xf7fbe884 ◂— 0
10.  EDX  0x0
11.  EDI  0x0
12.  * ESI  0xf7fbd000 ◂— mov word ptr [ebp + 0x1d], cs /* 0x1d4d8c */
13.  * EBP  0xffffd4f8 ◂— 0x0
14.  * ESP  0xffffd4a8 —▸ 0xffffd4f8 ◂— 0x0
15.  * EIP  0x804857d (main+37) ◂— lea eax, [ebx - 0x19ca]
16.  ---------------[ DISASM / i386 / set emulate on ]--------------
17.   ▶ 0x804857d <main+37>    lea    eax, [ebx - 0x19ca]
18.     0x8048583 <main+43>    push   eax
19.     0x8048584 <main+44>    push   6
20.     0x8048586 <main+46>    call   test                    <test>
21.
22.     0x804858b <main+51>    add    esp, 0x10
23.     0x804858e <main+54>    mov    eax, 0
24.     0x8048593 <main+59>    lea    esp, [ebp - 8]
25.     0x8048596 <main+62>    pop    ecx
26.     0x8048597 <main+63>    pop    ebx
27.     0x8048598 <main+64>    pop    ebp
28.     0x8048599 <main+65>    lea    esp, [ecx - 4]
29.  --------------------[ STACK ]---------------------
30.  00:0000| esp 0xffffd4a8 —▸ 0xffffd4f8 ◂— 0x0
31.  01:0004|     0xffffd4ac —▸ 0x804857a (main+34) ◂— sub esp, 8
32.  02:0008|     0xffffd4b0 ◂— 9 /* '\t' */
33.  03:000c|     0xffffd4b4 —▸ 0xffffd6f7 ◂— '/root/tmp/5.2.3'
34.  04:0010|     0xffffd4b8 —▸ 0xf7e18239 ◂— add ebx, 0x1a4dc7
35.  05:0014|     0xffffd4bc —▸ 0xf7fc0808 ◂— 0
36.  06:0018|     0xffffd4c0 —▸ 0xf7fbd000 ◂— mov word ptr [ebp + 0x1d], cs /* 0x1d4d8c */
37.  07:001c|     0xffffd4c4 —▸ 0xf7fbd000 ◂— mov word ptr [ebp + 0x1d], cs /* 0x1d4d8c */
38.  ------------------[ BACKTRACE ]-------------------
39    ▶ 0 0x804857d main+37
40.   1 0xf7e00fa1 __libc_start_main+241
41.  ----------------------------------------------------
```

在 pwndbg 中运行程序，将断点设置于 main 函数处。程序运行后停止在 0x804857d 这一指令，栈顶是 0xffffd4a8。运行 n 单步执行至指令 0x8048586<main+46>。可以发现，栈顶由 0xffffd4a8 上升到 0xffffd4a0，结合执行的几个命令，发现新入栈的内容从下往上分别为 0xffffd4a4 —▸ hello 字符串地址，0xffffd4a0 ◂— 0x6，即 main 函数将 test(6,"hello")的两个参数从右至左入栈。

```
1.  pwndbg> c
2.  Continuing.
3.
4.  Breakpoint 2, 0x08048586 in main ()
5.  LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
6.  -----------[ REGISTERS / show-flags off / show-compact-regs off ]-----------
7.   * EAX  0x8048636 ◂— push 0x6f6c6c65 /* 'hello' */
8.   EBX  0x804a000 (_GLOBAL_OFFSET_TABLE_) —▸ 0x8049f08 (_DYNAMIC) ◂— add dword ptr [eax], eax
9.   ECX  0xf7fbe884 ◂— 0
10.  EDX  0x0
11.  EDI  0x0
12.  ESI  0xf7fbd000 ◂— mov word ptr [ebp + 0x1d], cs /* 0x1d4d8c */
13.  EBP  0xffffd4f8 ◂— 0x0
14.  * ESP  0xffffd4a0 ◂— 0x6
```

```
15.  * EIP  0x8048586 (main+46) —▸ 0xffff98e8 ◂— 0x0
16.  ----------------------[ DISASM / i386 / set emulate on ]----------------------
17.     0x804857a <main+34>    sub    esp, 8
18.     0x804857d <main+37>    lea    eax, [ebx - 0x19ca]
19.     0x8048583 <main+43>    push   eax
20.     0x8048584 <main+44>    push   6
21.   ▶ 0x8048586 <main+46>    call   test        <test>
22.          arg[0]: 0x6
23.          arg[1]: 0x8048636 ◂— push 0x6f6c6c65 /* 'hello' */
24.          arg[2]: 0xffffd4f8 ◂— 0x0
25.          arg[3]: 0x804857a (main+34) ◂— sub esp, 8
26.
27.     0x804858b <main+51>    add    esp, 0x10
28.     0x804858e <main+54>    mov    eax, 0
29.     0x8048593 <main+59>    lea    esp, [ebp - 8]
30.     0x8048596 <main+62>    pop    ecx
31.     0x8048597 <main+63>    pop    ebx
32.     0x8048598 <main+64>    pop    ebp
33.  ----------------------[ STACK ]-----------------------------------
34.  00:0000| esp 0xffffd4a0 ◂— 0x6
35.  01:0004|     0xffffd4a4 —▸ 0x8048636 ◂— push 0x6f6c6c65 /* 'hello' */
36.  02:0008|     0xffffd4a8 —▸ 0xffffd4f8 ◂— 0x0
37.  03:000c|     0xffffd4ac —▸ 0x804857a (main+34) ◂— sub esp, 8
38.  04:0010|     0xffffd4b0 ◂— 9 /* '\t' */
39.  05:0014|     0xffffd4b4 —▸ 0xffffd6f7 ◂— '/root/tmp/5.2.3'
40.  06:0018|     0xffffd4b8 —▸ 0xf7e18239 ◂— add ebx, 0x1a4dc7
41.  07:001c|     0xffffd4bc —▸ 0xf7fc0808 ◂— 0
42.  ----------------------[ BACKTRACE ]------------------------------
43.   ▶ 0 0x8048586 main+46
44.     1 0xf7e00fa1 __libc_start_main+241
45.  ------------------------------------------------------------------
```

接着，运行 s 单步步入 test 函数内。此时栈顶由 0xffffd4a0 ◂— 0x6 更新为 0xffffd49c —▸0x804858b (main+51)，这是因为调用者自动压栈了子函数的返回地址 0x804858b，即 main 函数执行完 call test 后的下一个地址。

```
1.  pwndbg> s
2.  0x08048523 in test ()
3.  LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
4.  -----------[ REGISTERS / show-flags off / show-compact-regs off ]-----------
5.   EAX  0x8048636 ◂— push 0x6f6c6c65 /* 'hello' */
6.   EBX  0x804a000 (_GLOBAL_OFFSET_TABLE_) —▸ 0x8049f08 (_DYNAMIC) ◂— add dword ptr [eax], eax
7.   ECX  0xf7fbe884 ◂— 0
8.   EDX  0x0
9.   EDI  0x0
10.  ESI  0xf7fbd000 ◂— mov word ptr [ebp + 0x1d], cs /* 0x1d4d8c */
11.  EBP  0xffffd4f8 ◂— 0x0
12.  * ESP  0xffffd49c —▸ 0x804858b (main+51) ◂— add esp, 0x10
13.  * EIP  0x8048523 (test) ◂— push ebp
14.  ----------------------[ DISASM / i386 / set emulate on ]----------------------
15.   ▶ 0x8048523 <test>       push   ebp
16.     0x8048524 <test+1>     mov    ebp, esp
17.     0x8048526 <test+3>     push   ebx
18.     0x8048527 <test+4>     sub    esp, 4
19.     0x804852a <test+7>   call   __x86.get_pc_thunk.ax      <__x86.get_pc_thunk.ax>
20.
21.     0x804852f <test+12>    add    eax, 0x1ad1
22.     0x8048534 <test+17>    sub    esp, 4
23.     0x8048537 <test+20>    push   dword ptr [ebp + 0xc]
```

```
24.    0x804853a <test+23>    push   dword ptr [ebp + 8]
25.    0x804853d <test+26>    lea    edx, [eax - 0x19d0]
26.    0x8048543 <test+32>    push   edx
27. -------------------------------[ STACK ]-------------------------------
28. 00:0000| esp 0xffffd49c —▸ 0x804858b (main+51) ◂— add esp, 0x10
29. 01:0004|     0xffffd4a0 ◂— 0x6
30. 02:0008|     0xffffd4a4 —▸ 0x8048636 ◂— push 0x6f6c6c65 /* 'hello' */
31. 03:000c|     0xffffd4a8 —▸ 0xffffd4f8 ◂— 0x0
32. 04:0010|     0xffffd4ac —▸ 0x804857a (main+34) ◂— sub esp, 8
33. 05:0014|     0xffffd4b0 ◂— 9 /* '\t' */
34. 06:0018|     0xffffd4b4 —▸ 0xffffd6f7 ◂— '/root/tmp/5.2.3'
35. 07:001c|     0xffffd4b8 —▸ 0xf7e18239 ◂— add ebx, 0x1a4dc7
36. -----------------------------[ BACKTRACE ]-----------------------------
37.  ▶ 0 0x8048523 test
38.    1 0x804858b main+51
39.    2 0xf7e00fa1 __libc_start_main+241
40. -----------------------------------------------------------------------
```

5.2.3 被调用者规则

被调用者应该遵循如下规则。

(1) 将 EBP 入栈,并将 ESP 中的值复制到 EBP 中;

(2) 将 callee-saved 寄存器的值入栈,callee-saved 寄存器包括 EBX、EDI 和 ESI;

(3) 在栈上为局部变量分配空间,开始执行子程序;

(4) 当子程序返回时,将返回的执行结果保存在 EAX 中;

(5) 弹出栈中保存的 callee-saved 寄存器值,恢复 callee-saved 寄存器的值(ESI 和 EDI);

(6) 执行指令 mov esp,EBP 收回局部变量的内存空间;

(7) 弹出栈中保存的 EBP 值恢复调用者的基址寄存器值;

(8) 执行 ret 指令返回到调用者程序。

规则(1)的目的是保存调用子程序之前的基址指针,用于寻找栈上的参数和局部变量。当一个子程序开始执行时,基址指针保存栈指针指示子程序的执行。为了在子程序完成之后能正确定位调用者的参数和局部变量,EBP 的值需要返回。

接 5.2.2 节进入 test 函数演示被调用者规则。

```
1.  pwndbg> s
2.  0x08048523 in test ()
3.  LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
4.  -----------[ REGISTERS / show-flags off / show-compact-regs off ]-----------
5.   EAX  0x8048636 ◂— push 0x6f6c6c65 /* 'hello' */
6.   EBX  0x804a000 (_GLOBAL_OFFSET_TABLE_) —▸ 0x8049f08 (_DYNAMIC) ◂— add dword ptr [eax], eax
7.   ECX  0xf7fbe884 ◂— 0
8.   EDX  0x0
9.   EDI  0x0
10.  ESI  0xf7fbd000 ◂— mov word ptr [ebp + 0x1d], cs /* 0x1d4d8c */
11.  EBP  0xffffd4f8 ◂— 0x0
12.  * ESP  0xffffd49c —▸ 0x804858b (main+51) ◂— add esp, 0x10
13.  * EIP  0x8048523 (test) ◂— push ebp
14. ---------------[ DISASM / i386 / set emulate on ]----------------------
15.  ▶ 0x8048523 <test>       push   ebp
16.    0x8048524 <test+1>     mov    ebp, esp
```