俞楚凡 24302010004 ICS 4.27 Lab

大致梳理一下我的实验思路。

1. 使用ldd观察demo可执行文件：需要确定demo链接了哪些动态库以及存在哪些undefined symbol

在文件目录中使用ldd指令：

```
ainfinity@AInfinity:~/ics/lab8$ ldd demo
        linux-vdso.so.1 (0x00007fff7e8d7000)
        libseries.so => not found
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe08faf8000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fe08fd17000)
ainfinity@AInfinity:~/ics/lab8$ 
```

第一行是虚拟内存共享对象，与动态链接库无关。

第二行表示`demo`依赖于名为`libseries.so`的共享库，但是这个共享库未找到。这说明我们可能需要自己编写这个库。

第三行为c核心动态链接库，这个库能够在内核中被找到。

第四行为动态链接器本身。

ldd指令确定了demo的动态链接库依赖关系和缺失的依赖链接库。

2. 使用readelf读取ELF格式文件信息：分析demo和libseries.so的动态链接信息、headers和sections

先来看`demo`。

使用`readelf -d demo`指定-d参数输出动态链接信息。

```
ainfinity@AInfinity:~/ics/lab8$ readelf -d demo

Dynamic section at offset 0x2dd0 contains 27 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libseries.so]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x1000
 0x000000000000000d (FINI)               0x128c
 0x0000000000000019 (INIT_ARRAY)         0x3dc0
 0x000000000000001b (INIT_ARRAYSZ)       8 (bytes)
 0x000000000000001a (FINI_ARRAY)         0x3dc8
 0x000000000000001c (FINI_ARRAYSZ)       8 (bytes)
 0x000000006ffffef5 (GNU_HASH)           0x3b0
 0x0000000000000005 (STRTAB)             0x510
 0x0000000000000006 (SYMTAB)             0x3d8
 0x000000000000000a (STRSZ)              198 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000015 (DEBUG)              0x0
 0x0000000000000003 (PLTGOT)             0x3fe8
 0x0000000000000002 (PLTRELSZ)           144 (bytes)
 0x0000000000000014 (PLTREL)             RELA
 0x0000000000000017 (JMPREL)             0x6f8
 0x0000000000000007 (RELA)               0x620
 0x0000000000000008 (RELASZ)             216 (bytes)
 0x0000000000000009 (RELAENT)            24 (bytes)
 0x000000006ffffffb (FLAGS_1)            Flags: PIE
 0x000000006ffffffe (VERNEED)            0x5f0
 0x000000006fffffff (VERNEEDNUM)         1
 0x000000006ffffff0 (VERSYM)             0x5d6
 0x000000006ffffff9 (RELACOUNT)          3
 0x0000000000000000 (NULL)               0x0
```

这显示需要动态链接库libseries.so和libc.so.6的支持。

使用`readelf -h demo`指定-h参数，使`readelf`显示ELF文件头信息。

```
ainfinity@AInfinity:~/ics/lab8$ readelf -h demo
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x10a0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14232 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         14
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
ainfinity@AInfinity:~/ics/lab8$
```

文件头标识了很多信息。Magic是一串固定字节的序列，标识这是一个ELF文件。

class标识64位，data标识端序，version标识版本，Entry point address代表程序入口点位置，随后两项标识了程序头和节头表地址的偏移量。随后标识了程序头与节头表的大小，条目信息和条目大小。

使用`readelf -S demo`查看程序节头表信息。

```
ainfinity@AInfinity:~/ics/lab8$ readelf -S demo
There are 31 section headers, starting at offset 0x3798:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .note.gnu.pr[...] NOTE             0000000000000350  00000350
       0000000000000020  0000000000000000   A       0     0     8
  [ 2] .note.gnu.bu[...] NOTE             0000000000000370  00000370
       0000000000000024  0000000000000000   A       0     0     4
  [ 3] .interp           PROGBITS         0000000000000394  00000394
       000000000000001c  0000000000000000   A       0     0     1
  [ 4] .gnu.hash         GNU_HASH         00000000000003b0  000003b0
       0000000000000028  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM           00000000000003d8  000003d8
       0000000000000138  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           0000000000000510  00000510
       00000000000000c6  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM           00000000000005d6  000005d6
       000000000000001a  0000000000000002   A       5     0     2
  [ 8] .gnu.version_r    VERNEED          00000000000005f0  000005f0
       0000000000000030  0000000000000000   A       6     1     8
  [ 9] .rela.dyn         RELA             0000000000000620  00000620
       00000000000000d8  0000000000000018   A       5     0     8
  [10] .rela.plt         RELA             00000000000006f8  000006f8
       0000000000000090  0000000000000018  AI       5    24     8
  [11] .init             PROGBITS         0000000000001000  00001000
       0000000000000017  0000000000000000  AX       0     0     4
  [12] .plt              PROGBITS         0000000000001020  00001020
       0000000000000070  0000000000000010  AX       0     0     16
```

方便查看程序中包含了哪些条目。当然，一个条目一定对应一个节头表中的项，而节头表中存在某一项不意味着程序包含对应的条目。

我们也可以对libseries.so执行相同的操作。

`readelf -d libseries.so`:

```
ainfinity@AInfinity:~/ics/lab8$ readelf -d libseries.so

Dynamic section at offset 0x2e78 contains 17 entries:
  Tag        Type                         Name/Value
 0x000000000000000c (INIT)               0x1000
 0x000000000000000d (FINI)               0x1164
 0x0000000000000019 (INIT_ARRAY)         0x3e68
 0x000000000000001b (INIT_ARRAYSZ)       8 (bytes)
 0x000000000000001a (FINI_ARRAY)         0x3e70
 0x000000000000001c (FINI_ARRAYSZ)       8 (bytes)
 0x000000006ffffef5 (GNU_HASH)           0x260
 0x0000000000000005 (STRTAB)             0x330
 0x0000000000000006 (SYMTAB)             0x288
 0x000000000000000a (STRSZ)              107 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000003 (PLTGOT)             0x3fe8
 0x0000000000000007 (RELA)               0x3a0
 0x0000000000000008 (RELASZ)             168 (bytes)
 0x0000000000000009 (RELAENT)            24 (bytes)
 0x000000006ffffff9 (RELACOUNT)          3
 0x0000000000000000 (NULL)               0x0
```

`readelf -h libseries.so`:

```
ainfinity@AInfinity:~/ics/lab8$ readelf -h libseries.so
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          13512 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         24
  Section header string table index: 23
```

`readelf -S libseries.so`:

```
ainfinity@AInfinity:~/ics/lab8$ readelf -S libseries.so
There are 24 section headers, starting at offset 0x34c8:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .note.gnu.bu[...] NOTE             0000000000000238  00000238
       0000000000000024  0000000000000000   A       0     0     4
  [ 2] .gnu.hash         GNU_HASH         0000000000000260  00000260
       0000000000000028  0000000000000000   A       3     0     8
  [ 3] .dynsym           DYNSYM           0000000000000288  00000288
       00000000000000a8  0000000000000018   A       4     1     8
  [ 4] .dynstr           STRTAB           0000000000000330  00000330
       000000000000006b  0000000000000000   A       0     0     1
  [ 5] .rela.dyn         RELA             00000000000003a0  000003a0
       00000000000000a8  0000000000000018   A       3     0     8
  [ 6] .init             PROGBITS         0000000000001000  00001000
       0000000000000017  0000000000000000  AX       0     0     4
  [ 7] .plt              PROGBITS         0000000000001020  00001020
       0000000000000010  0000000000000010  AX       0     0     16
  [ 8] .plt.got          PROGBITS         0000000000001030  00001030
       0000000000000008  0000000000000008  AX       0     0     8
  [ 9] .text             PROGBITS         0000000000001040  00001040
       0000000000000122  0000000000000000  AX       0     0     16
  [10] .fini             PROGBITS         0000000000001164  00001164
       0000000000000009  0000000000000000  AX       0     0     4
  [11] .eh_frame_hdr     PROGBITS         0000000000002000  00002000
       000000000000002c  0000000000000000   A       0     0     4
  [12] .eh_frame         PROGBITS         0000000000002030  00002030
       000000000000009c  0000000000000000   A       0     0     8
  [13] .init_array       INIT_ARRAY       0000000000003e68  00002e68
       0000000000000008  0000000000000008  WA       0     0     8
  [14] .fini_array       FINI_ARRAY       0000000000003e70  00002e70
       0000000000000008  0000000000000008  WA       0     0     8
```

3. 使用objdump反汇编文件：反汇编demo和libseries.so，配合readelf读取动态链接函数的GOT

GOT是**Global offset table**，也就是全局偏移表的简称。GOT被用来存储全局变量和外部函数的实际地址。当程序运行时，动态链接器会解析程序符号，并将他们的真实地址填入GOT中。这样做的好处是允许共享库中的代码在内存的任何位置加载并执行，同时还能访问到正确的外部符号地址。

我们首先反汇编`demo`和`libseries.so`：

```
objdump -d demo > demo.asm
objdump -d libseries.so > libseries.asm
```

我们可以使用`readelf -r demo | grep R_X86_64_JUMP_SLO`来列出所有重定位条目。在输出中查找`Type = R_X86_64_JUMP_SLO`的项，这将列出所有被重定位的函数名称。可以看到，在四个gcc标准库函数之外，有两个自定义的square_sum和linear_sum函数。

```
ainfinity@AInfinity:~/ics/lab8$ readelf -r demo | grep R_X86_64_JUMP_SLO
000000004000  000300000007 R_X86_64_JUMP_SLO 0000000000000000 square_sum + 0
000000004008  000400000007 R_X86_64_JUMP_SLO 0000000000000000 linear_sum + 0
000000004010  000500000007 R_X86_64_JUMP_SLO 0000000000000000 printf@GLIBC_2.2.5 + 0
000000004018  000600000007 R_X86_64_JUMP_SLO 0000000000000000 fprintf@GLIBC_2.2.5 + 0
000000004020  000800000007 R_X86_64_JUMP_SLO 0000000000000000 strtol@GLIBC_2.2.5 + 0
000000004028  000900000007 R_X86_64_JUMP_SLO 0000000000000000 exit@GLIBC_2.2.5 + 0
ainfinity@AInfinity:~/ics/lab8$
```

使用`objdump -s -j .got.plt demo`可以获取原始的GOT表值。

```
ainfinity@AInfinity:~/ics/lab8$ objdump -s -j .got.plt demo

demo:     file format elf64-x86-64

Contents of section .got.plt:
 3fe8 d03d0000 00000000 00000000 00000000  .=..............
 3ff8 00000000 00000000 36100000 00000000  ........6.......
 4008 46100000 00000000 56100000 00000000  F.......V.......
 4018 66100000 00000000 76100000 00000000  f.......v.......
 4028 86100000 00000000                    ........
ainfinity@AInfinity:~/ics/lab8$
```

但是有较大阅读难度。

注意到`libseries.so`没有重定位条目。

4. 分析函数原型和功能：通过阅读汇编代码，生成两个函数的原型并逆向分析其功能

`vim demo.asm`：

可以看到，有两个相关的`square_sum`函数定义：

```
0000000000001020 <square_sum@plt-0x10>:
    1020:       ff 35 ca 2f 00 00       push   0x2fca(%rip)        # 3ff0 <_GLOBAL_OFFSET_TABLE_+0x8>
    1026:       ff 25 cc 2f 00 00       jmp    *0x2fcc(%rip)        # 3ff8 <_GLOBAL_OFFSET_TABLE_+0x10>
    102c:       0f 1f 40 00             nopl   0x0(%rax)

0000000000001030 <square_sum@plt>:
    1030:       ff 25 ca 2f 00 00       jmp    *0x2fca(%rip)        # 4000 <square_sum@Base>
    1036:       68 00 00 00 00          push   $0x0
    103b:       e9 e0 ff ff ff          jmp    1020 <_init+0x20>
```

第一个是处理plt首次访问逻辑的模板函数。第二个是实际的plt条目，用于后续的直接调用。`linear_sum`同理。这两个函数在`libseries.so`中有定义。检查`libseries.asm`：

```
00000000000010f9 <linear_sum>:
    10f9:	55                   	push   %rbp
    10fa:	48 89 e5             	mov    %rsp,%rbp
    10fd:	89 7d ec             	mov    %edi,-0x14(%rbp)
    1100:	48 c7 45 f8 00 00 00 	movq   $0x0,-0x8(%rbp)
    1107:	00 
    1108:	c7 45 f4 01 00 00 00 	movl   $0x1,-0xc(%rbp)
    110f:	eb 0d                	jmp    111e <linear_sum+0x25>
    1111:	8b 45 f4             	mov    -0xc(%rbp),%eax
    1114:	48 98                	cltq   
    1116:	48 01 45 f8          	add    %rax,-0x8(%rbp)
    111a:	83 45 f4 01          	addl   $0x1,-0xc(%rbp)
    111e:	8b 45 f4             	mov    -0xc(%rbp),%eax
    1121:	3b 45 ec             	cmp    -0x14(%rbp),%eax
    1124:	7e eb                	jle    1111 <linear_sum+0x18>
    1126:	48 8b 45 f8          	mov    -0x8(%rbp),%rax
    112a:	5d                   	pop    %rbp
    112b:	c3                   	ret    
```

```
000000000000112c <square_sum>:
    112c:	55                   	push   %rbp
    112d:	48 89 e5             	mov    %rsp,%rbp
    1130:	89 7d ec             	mov    %edi,-0x14(%rbp)
    1133:	48 c7 45 f8 00 00 00 	movq   $0x0,-0x8(%rbp)
    113a:	00 
    113b:	c7 45 f4 01 00 00 00 	movl   $0x1,-0xc(%rbp)
    1142:	eb 10                	jmp    1154 <square_sum+0x28>
    1144:	8b 45 f4             	mov    -0xc(%rbp),%eax
    1147:	0f af c0             	imul   %eax,%eax
    114a:	48 98                	cltq   
    114c:	48 01 45 f8          	add    %rax,-0x8(%rbp)
    1150:	83 45 f4 01          	addl   $0x1,-0xc(%rbp)
    1154:	8b 45 f4             	mov    -0xc(%rbp),%eax
    1157:	3b 45 ec             	cmp    -0x14(%rbp),%eax
    115a:	7e e8                	jle    1144 <square_sum+0x18>
    115c:	48 8b 45 f8          	mov    -0x8(%rbp),%rax
    1160:	5d                   	pop    %rbp
    1161:	c3                   	ret    
```

分析这两段代码。可以看到，`linear_sum`求的是1-n的和，也即，答案可以由一个公式简单的求出：`ans = n * (n - 1) / 2`.

而square_sum求的是1-n的平方和。也即，答案可以由一个公式简单地求出：`ans = n * (n + 1) * (2n + 1) / 6`.

编写高效动态链接库

1. 分析现有函数：通过逆向工程理解libseries.so中实现的函数功能和算法
2. 设计高效算法：根据数学知识，设计更加高效的计算方法替代原有实现

由上面给出的公式可以O(1)的算出答案。

3. 编写动态链接库：实现优化后的函数并编译生成新的动态链接库

我们编写`libseries.h`和`libseries.c`，用于生成更高性能的链接库：

```
// libseries.c
#include <stdio.h>
#include <stdint.h>

int64_t linear_sum(int n) {
        return n * (n + 1) / 2;
}

int64_t square_sum(int n) {
        return n * (n + 1) * (2 * n + 1) / 6;
}
"libseries.c" 11L, 180B                                    9,16          All
```

在demo链接`libseries.so`之前，首先需要添加全局变量，让操作系统找到你的`libseries.so`共享库。我们可以通过设定临时全局变量的方式将当前目录添加到环境变量中：

```
ainfinity@AInfinity:~/ics/lab8$ export LD_LIBRARY_PATH=/home/ainfinity/ics/lab8:$LD_LIBRARY_PATH
```

这样系统能够正确调用共享库。

随后，我们编译新的`libseries.so`共享库：

```
gcc -fPIC -c libseries.c -o libseries.o
gcc -shared -o libseries.so libseries.o
```

最后`./demo`运行。

```
ainfinity@AInfinity:~/ics/lab8$ ./demo 3
The linear sum from 1 to 3 is 6
The square sum from 1 to 3 is 14
```

程序运行成功。

库打桩技术

库打桩技术允许我们劫持可执行文件对于共享库函数的访问。

我们首先编写自己的`mylibseries.c`文件。这个文件包括了两个包装函数，在原本的函数之外套了一层计时器，可以输出运行函数的时间。

```c
#ifdef RUNTIME
#define _GNU_SOURCE

#include <stdio.h>
#include <time.h>
#include <stdint.h>
#include <dlfcn.h>
#include <stdlib.h>

// linear_sum wrapper function
int64_t linear_sum(int n) {
        int64_t (*default_linear_sum)(int) = NULL;
        void *handle;
        char *error;

        // open .so
        handle = dlopen("libseries.so", RTLD_LAZY);
        if (!handle) {
                fprintf(stderr, "%s\n", dlerror());
                exit(EXIT_FAILURE);
        }

        dlerror();

        default_linear_sum = dlsym(handle, "linear_sum");

        if ((error = dlerror()) != NULL) {
                fprintf(stderr, "%s\n", error);
                exit(EXIT_FAILURE);
        }

        // clock
        struct timespec start, end;
        clock_gettime(CLOCK_MONOTONIC, &start);

        int64_t res = default_linear_sum(n);

        clock_gettime(CLOCK_MONOTONIC, &end);

        long long elapsed_ns = (end.tv_sec - start.tv_sec) * 1000000000LL + (end.tv_nsec - start.tv_nsec);

        printf("linear_sum used time: %lld nanoseconds\n", elapsed_ns);

        return res;
}

// square_sum wrapper function
int64_t square_sum(int n) {
        int64_t (*default_square_sum)(int) = NULL;
        void *handle;
        char *error;

        // open .so
"mylibseries.c" 83L, 1689B                                        42,20-27      Top
```

```c
        // clock
        struct timespec start, end;
        clock_gettime(CLOCK_MONOTONIC, &start);

        int64_t res = default_linear_sum(n);

        clock_gettime(CLOCK_MONOTONIC, &end);

        long long elapsed_ns = (end.tv_sec - start.tv_sec) * 1000000000LL + (end.tv_nsec - start.tv_nsec);

        printf("linear_sum used time: %lld nanoseconds\n", elapsed_ns);

        return res;
}

// square_sum wrapper function
int64_t square_sum(int n) {
        int64_t (*default_square_sum)(int) = NULL;
        void *handle;
        char *error;

        // open .so
        handle = dlopen("libseries.so", RTLD_LAZY);
        if (!handle) {
                fprintf(stderr, "%s\n", dlerror());
                exit(EXIT_FAILURE);
        }

        dlerror();

        default_square_sum = dlsym(handle, "square_sum");

        if ((error = dlerror()) != NULL) {
                fprintf(stderr, "%s\n", error);
                exit(EXIT_FAILURE);
        }

        // clock
        struct timespec start, end;
        clock_gettime(CLOCK_MONOTONIC, &start);

        int64_t res = default_square_sum(n);

        clock_gettime(CLOCK_MONOTONIC, &end);

        long long elapsed_ns = (end.tv_sec - start.tv_sec) * 1000000000LL + (end.tv_nsec - start.tv_nsec);

        printf("square_sum used time: %lld nanoseconds\n", elapsed_ns);

        return res;
}
#endif
                                                                  83,1          Bot
```

随后，构建自定义共享库：

```
linux> gcc -DRUNTIME -shared -fpic -o mylibseries.so mylibseries.c
-ldl
```

随后，通过将`LD_PRELOAD`变量设定为我们的共享库来劫持共享库函数的访问。

```
linux> LD_PRELOAD="./mylibseries.so" ./demo 3
```

得到了带有计时的程序输出。

我们首先对我们实现的优化的O(1)的函数进行计时：

```
ainfinity@AInfinity:~/ics/lab8$ LD_PRELOAD="./mylibseries.so" ./demo 1000
linear_sum used time: 57 nanoseconds
square_sum used time: 44 nanoseconds
The linear sum from 1 to 1000 is 500500
The square sum from 1 to 1000 is 333833500
ainfinity@AInfinity:~/ics/lab8$ LD_PRELOAD="./mylibseries.so" ./demo 1000
linear_sum used time: 122 nanoseconds
square_sum used time: 94 nanoseconds
The linear sum from 1 to 1000 is 500500
The square sum from 1 to 1000 is 333833500
ainfinity@AInfinity:~/ics/lab8$ LD_PRELOAD="./mylibseries.so" ./demo 1000
linear_sum used time: 116 nanoseconds
square_sum used time: 94 nanoseconds
The linear sum from 1 to 1000 is 500500
The square sum from 1 to 1000 is 333833500
ainfinity@AInfinity:~/ics/lab8$ LD_PRELOAD="./mylibseries.so" ./demo 1000
linear_sum used time: 74 nanoseconds
square_sum used time: 60 nanoseconds
The linear sum from 1 to 1000 is 500500
The square sum from 1 to 1000 is 333833500
ainfinity@AInfinity:~/ics/lab8$ LD_PRELOAD="./mylibseries.so" ./demo 1000
linear_sum used time: 112 nanoseconds
square_sum used time: 81 nanoseconds
The linear sum from 1 to 1000 is 500500
The square sum from 1 to 1000 is 333833500
ainfinity@AInfinity:~/ics/lab8$ LD_PRELOAD="./mylibseries.so" ./demo 1000
linear_sum used time: 126 nanoseconds
square_sum used time: 103 nanoseconds
The linear sum from 1 to 1000 is 500500
The square sum from 1 to 1000 is 333833500
ainfinity@AInfinity:~/ics/lab8$ LD_PRELOAD="./mylibseries.so" ./demo 1000
linear_sum used time: 125 nanoseconds
square_sum used time: 98 nanoseconds
The linear sum from 1 to 1000 is 500500
The square sum from 1 to 1000 is 333833500
ainfinity@AInfinity:~/ics/lab8$ LD_PRELOAD="./mylibseries.so" ./demo 1000
linear_sum used time: 150 nanoseconds
square_sum used time: 99 nanoseconds
The linear sum from 1 to 1000 is 500500
The square sum from 1 to 1000 is 333833500
ainfinity@AInfinity:~/ics/lab8$ LD_PRELOAD="./mylibseries.so" ./demo 1000
linear_sum used time: 59 nanoseconds
square_sum used time: 45 nanoseconds
The linear sum from 1 to 1000 is 500500
The square sum from 1 to 1000 is 333833500
ainfinity@AInfinity:~/ics/lab8$ LD_PRELOAD="./mylibseries.so" ./demo 1000
linear_sum used time: 121 nanoseconds
square_sum used time: 98 nanoseconds
The linear sum from 1 to 1000 is 500500
The square sum from 1 to 1000 is 333833500
```

运行了十次的结果。

再对原本的O(n)的函数进行计时：

```
ainfinity@AInfinity:~/ics/lab8$ LD_PRELOAD="./mylibseries.so" ./demo 1000
linear_sum used time: 3340 nanoseconds
square_sum used time: 3949 nanoseconds
The linear sum from 1 to 1000 is 500500
The square sum from 1 to 1000 is 333833500
ainfinity@AInfinity:~/ics/lab8$ LD_PRELOAD="./mylibseries.so" ./demo 1000
linear_sum used time: 4167 nanoseconds
square_sum used time: 3806 nanoseconds
The linear sum from 1 to 1000 is 500500
The square sum from 1 to 1000 is 333833500
ainfinity@AInfinity:~/ics/lab8$ LD_PRELOAD="./mylibseries.so" ./demo 1000
linear_sum used time: 1574 nanoseconds
square_sum used time: 1621 nanoseconds
The linear sum from 1 to 1000 is 500500
The square sum from 1 to 1000 is 333833500
ainfinity@AInfinity:~/ics/lab8$ LD_PRELOAD="./mylibseries.so" ./demo 1000
linear_sum used time: 1284 nanoseconds
square_sum used time: 1501 nanoseconds
The linear sum from 1 to 1000 is 500500
The square sum from 1 to 1000 is 333833500
ainfinity@AInfinity:~/ics/lab8$ LD_PRELOAD="./mylibseries.so" ./demo 1000
linear_sum used time: 3399 nanoseconds
square_sum used time: 3434 nanoseconds
The linear sum from 1 to 1000 is 500500
The square sum from 1 to 1000 is 333833500
ainfinity@AInfinity:~/ics/lab8$ LD_PRELOAD="./mylibseries.so" ./demo 1000
linear_sum used time: 2791 nanoseconds
square_sum used time: 2260 nanoseconds
The linear sum from 1 to 1000 is 500500
The square sum from 1 to 1000 is 333833500
ainfinity@AInfinity:~/ics/lab8$ LD_PRELOAD="./mylibseries.so" ./demo 1000
linear_sum used time: 3428 nanoseconds
square_sum used time: 3757 nanoseconds
The linear sum from 1 to 1000 is 500500
The square sum from 1 to 1000 is 333833500
ainfinity@AInfinity:~/ics/lab8$ LD_PRELOAD="./mylibseries.so" ./demo 1000
linear_sum used time: 2488 nanoseconds
square_sum used time: 2376 nanoseconds
The linear sum from 1 to 1000 is 500500
The square sum from 1 to 1000 is 333833500
ainfinity@AInfinity:~/ics/lab8$ LD_PRELOAD="./mylibseries.so" ./demo 1000
linear_sum used time: 3769 nanoseconds
square_sum used time: 3716 nanoseconds
The linear sum from 1 to 1000 is 500500
The square sum from 1 to 1000 is 333833500
ainfinity@AInfinity:~/ics/lab8$ LD_PRELOAD="./mylibseries.so" ./demo 1000
linear_sum used time: 1867 nanoseconds
square_sum used time: 1683 nanoseconds
The linear sum from 1 to 1000 is 500500
The square sum from 1 to 1000 is 333833500
ainfinity@AInfinity:~/ics/lab8$
```

可以看到，n仅仅到达了1000，差距是非常显著的。