

二进制漏洞挖掘与利用

课时4: 软件逆向工程

- 逆向之前的准备工作
- 软件逆向工程与源代码阅读比较
- 指令与控制结构
- 高级语言中结构的内存布局
- 逆向的两种方法: 自顶向下与自底向上
- 识别常见算法与代码

- 基本工具的使用
 - GNU binutils
 - 静态分析工具
 - IDA Pro: 昂贵, 功能强大, de facto standard
 - radare2: **自由软件**, 上手难度较高, 功能较弱
 - 调试器
 - Windows: WinDbg、x64dbg
 - Linux: gdb
- 基础的体系结构知识
- 英文阅读能力

file工具

```
$ file busybox
busybox: ELF 32-bit LSB executable, MIPS, MIPS32 version 1 (SYSV), dynamically linked, interpreter
/lib/ld-uClibc.so.0, stripped
$ file SendCFTrigger.exe
SendCFTrigger.exe: PE32 executable (console) Intel 80386, for MS Windows
$ file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, not stripped
```

识别待分析的文件格式，目标平台，目标指令集

阅读源代码

- 目的
 - 理解程序的用途与实现方式
- 需要理解对应的编程语言
 - 看懂每一行在做什么
 - 看懂每一个函数在做什么
 - 看懂模块在做什么, 怎么做
- 两种方法
 - 自顶向下
 - 选择感兴趣的应用场景从入口开始看
 - 一行一行, 一个函数一个函数看
 - 自底向上
 - 找到关键函数/变量, 分析引用关系
- 看文档了解设计思路
- 找参考资料, 学习背景知识
- 调试

阅读编译后的目标代码

- 目的
 - 理解程序的用途与实现方式
- 需要理解对应指令集架构的汇编语言
 - 看懂每一条指令的含义
 - 看懂程序中的控制结构
- 了解对应高级语言中结构的内存布局
- 两种方法
 - “自顶向下”
 - 从输入点开始分析
 - “自底向上”
 - 找到关键函数/变量, 分析引用关系
- 没有文档
- 找参考资料, 学习背景知识
- 调试

阅读源代码

- 目的
 - 理解程序的用途与实现方式
- 需要理解对应的编程语言
 - 看懂每一行在做什么
 - 看懂每一个函数在做什么
 - 看懂模块在做什么, 怎么做
- 两种方法
 - 自顶向下
 - 选择感兴趣的应用场景从入口开始看
 - 一行一行, 一个函数一个函数看
 - 自底向上
 - 找到关键函数/变量, 分析引用关系
- 看文档了解设计思路
- 找参考资料, 学习背景知识
- 调试

阅读编译后的目标代码

- 目的
 - 理解程序的用途与实现方式
- 需要理解对应指令集架构的汇编语言
 - 看懂每一条指令的含义
 - 看懂程序中的控制结构
- 了解对应高级语言中结构的内存布局
- 两种方法
 - “自顶向下”
 - 从输入点开始分析
 - “自底向上”
 - 找到关键函数/变量, 分析引用关系
- 没有文档
- 找参考资料, 学习背景知识
- 调试

- 目的
 - 理解程序的用途与实现方式
- 需要理解对应指令集架构的汇编语言
 - 看懂每一条指令的含义
 - 看懂程序中的控制结构
- 了解对应高级语言中结构的内存布局
- 两种方法
 - “自顶向下”
 - 从输入点开始分析
 - “自底向上”
 - 找到关键函数/变量, 分析引用关系
- 没有文档
- 找参考资料, 学习背景知识
- 调试

- x86及x86_64汇编代码的两种文本表示形式
 - ARM, MIPS等其他指令集架构不存在这种混乱
- Intel语法
 - Windows平台上的工具链常用的表示
 - MSVC
 - IDA Pro的反汇编输出是这种形式
 - Intel指令集手册中也采用这种形式
- AT&T语法
 - 基于GNU binutils的工具常用的表示方法
 - GNU as (汇编器)的输入
 - objdump (反汇编器)的输出
 - 可以使用 `-M intel` 改为Intel语法
 - gdb的反汇编输出
 - 可以使用 `set disassembly-flavor intel` 调节为Intel语法

Intel

```
.LC0:
.string "hello, world\n"
main:
    push ebp
    mov ebp, esp ;; insn dest, src
    and esp, 0FFFFFFF0h
    sub esp, 10h
    mov eax, offset LC0
    mov [esp], eax
    call printf
    mov eax, 0
    leave
    ret
```

AT & T

```
.LC0:
.string "hello, world\n"
main:
    pushl %ebp
    movl %esp, %ebp ;; insn src, dest
    andl $-16, %esp
    subl $16, %esp
    movl $.LC0, %eax
    movl %eax, (%esp)
    call printf
    movl $0, %eax
    leave
    ret
```

Intel

```
.LC0:
.string "hello, world\n"
main:
    push ebp
    mov ebp, esp
    and esp, 0FFFFFFF0h
    sub esp, 10h

    mov eax, offset LC0
    mov [esp], eax
    call printf
    mov eax, 0
    leave
    ret
```

AT & T

```
.LC0:
.string "hello, world\n"
main:
    pushl %ebp
    movl %esp, %ebp
    andl $-16, %esp
    subl $16, %esp
    ;; 寄存器名称前加%, 常量前加$
    movl $.LC0, %eax
    movl %eax, (%esp)
    call printf
    movl $0, %eax
    leave
    ret
```

Intel

```
.LC0:
.string "hello, world\n"
main:
    push ebp
    mov ebp, esp
    and esp, 0FFFFFFF0h
    sub esp, 10h
    mov eax, offset LC0
    mov [esp], eax
    call printf
    mov eax, 0
    leave
    ret
```

指令后缀表示操作数宽度:

- q: quad (64位)
- l: long (32位)
- w: word (16位)
- b: byte (8位)

AT & T

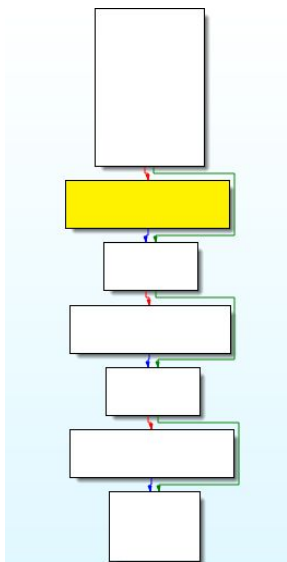
```
.LC0:
.string "hello, world\n"
main:
    pushl %ebp
    movl %esp, %ebp
    andl $-16, %esp
    subl $16, %esp
    movl $.LC0, %eax
    movl %eax, (%esp)
    call printf
    movl $0, %eax
    leave
    ret
```

- 查对应指令集的手册
 - x86, x86_64: 《Intel® 64 and IA-32 Architectures Software Developer Manuals》
 - ARM: 《ARM® and Thumb®-2 Instruction Set Quick Reference Card》

- 目的
 - 理解程序的用途与实现方式
- 需要理解对应指令集架构的汇编语言
 - 看懂每一条指令的含义
 - 看懂程序中的控制结构
- 了解对应高级语言中结构的内存布局
- 两种方法
 - “自顶向下”
 - 从输入点开始分析
 - “自底向上”
 - 找到关键函数/变量, 分析引用关系
- 没有文档
- 找参考资料, 学习背景知识
- 调试

if-then-else

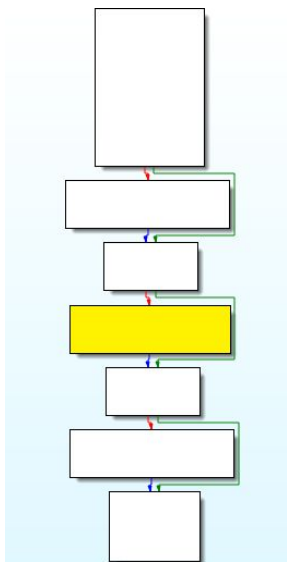
```
void f_signed (int a, int b) {  
    if (a>b) printf ("a>b\n");  
    if (a==b) printf ("a==b\n");  
    if (a<b) printf ("a<b\n");  
}
```



```
f_signed      proc near  
  
; ...<snip>...  
  
    mov     eax, [ebp+arg_0]  
    cmp     eax, [ebp+arg_4]  
    jle     short check_for_equal  
    sub     esp, 0Ch  
    lea     eax, offset aGreater; "a>b"  
    push    eax  
    call    _puts  
    add     esp, 10h  
check_for_equal:  
; ...<snip>...  
  
epilogue:  
    leave  
    retn  
f_signed     endp
```

if-then-else

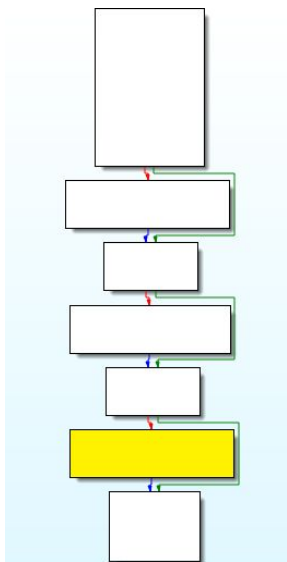
```
void f_signed (int a, int b) {  
    if (a>b) printf ("a>b\n");  
    if (a==b) printf ("a==b\n");  
    if (a<b) printf ("a<b\n");  
}
```



```
f_signed      proc near  
  
; ...<snip>...  
check_for_equal:  
    mov     eax, [ebp+arg_0]  
    cmp     eax, [ebp+arg_4]  
    jnz     short check_for_less_than  
    sub     esp, 0Ch  
    lea     eax, offset aEqual; "a==b"  
    push    eax  
    call    _puts  
    add     esp, 10h  
check_for_less_than:  
; ...<snip>...  
  
epilogue:  
    leave  
    retn  
f_signed      endp
```

if-then-else

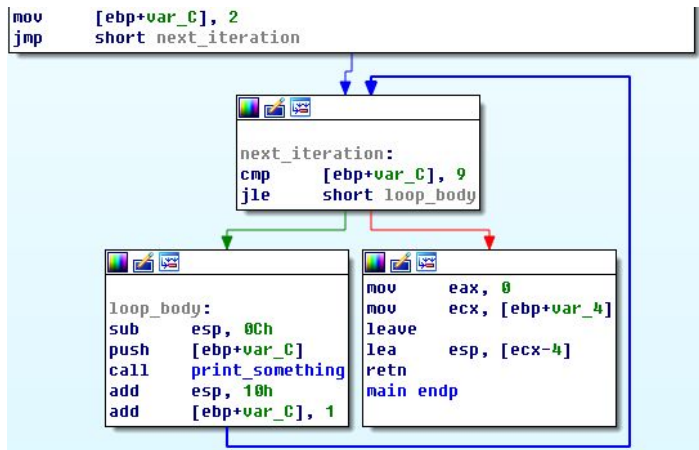
```
void f_signed (int a, int b) {  
    if (a>b) printf ("a>b\n");  
    if (a==b) printf ("a==b\n");  
    if (a<b) printf ("a<b\n");  
}
```



```
f_signed      proc near  
  
; ...<snip>...  
check_for_less_than:  
    mov     eax, [ebp+arg_0]  
    cmp     eax, [ebp+arg_4]  
    jge     short epilogue  
    sub     esp, 0Ch  
    lea     eax, offset aLessThan ; "a<b"  
    push    eax  
    call    _puts  
    add     esp, 10h  
epilogue:  
    leave  
    retn  
f_signed      endp
```


for循环

```
int main()
{
    for (int i = 2; i < 10; i++)
        print_something(i);
    return 0;
}
```



```
main                proc near

var_C                = dword ptr -0Ch
; ... <snip> ...
```

```
    mov     [ebp+var_C], 2
    jmp     short next_iteration
loop_body:
    sub     esp, 0Ch
    push    [ebp+var_C]
    call    print_something
    add     esp, 10h
    add     [ebp+var_C], 1
```

```
next_iteration:
    cmp     [ebp+var_C], 9
    jle     short loop_body
```

```
; ... <snip> ...
main                endp
```

```
void f (int a) {  
    switch (a) {  
        case 0: printf("zero\n"); break;  
        case 1: printf("one\n"); break;  
        case 2: printf("two\n"); break;  
        default: printf("unknown\n");  
                break;  
    }  
}
```

```
mov     edx, [ebp+arg_0]  
cmp     edx, 1  
jz      short caseone  
cmp     edx, 2  
jz      short casetwo  
test    edx, edx  
jnz     short caseunk  
sub     esp, 0Ch  
lea     edx, offset aZero; "zero"  
push    edx  
mov     ebx, eax  
call    _puts  
add     esp, 10h  
jmp     short epilogue  
;  
-----  
caseone:  
    ; ...<snip>...
```

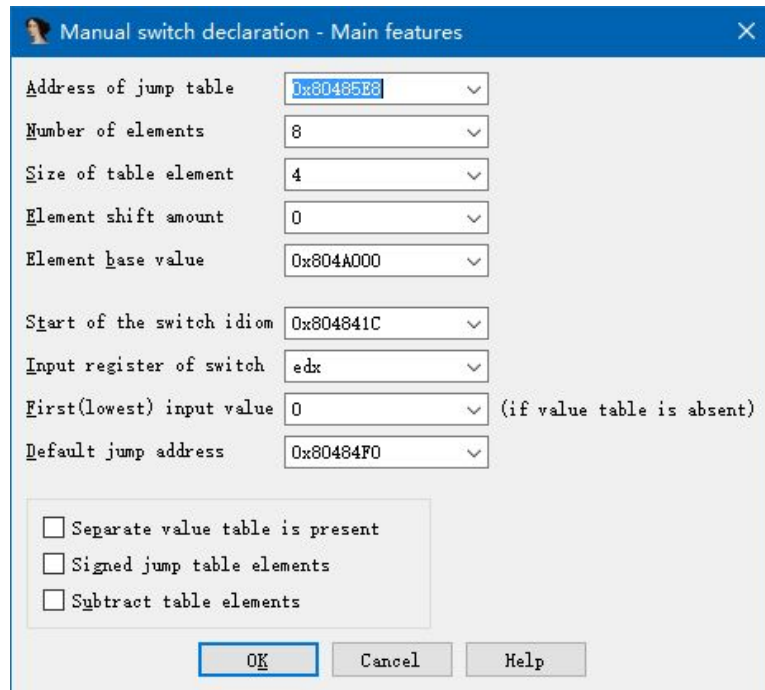
switch-case: 更多case

```
void f (int a) {  
    switch (a) {  
        case 0: printf("zero\n"); break;  
        case 1: printf("one\n"); break;  
        case 2: printf("two\n"); break;  
        case 3: printf("three\n"); break;  
        case 4: printf("four\n"); break;  
        case 5: printf("five\n"); break;  
        case 6: printf("six\n"); break;  
        case 7: printf("seven\n"); break;  
        default: printf("unknown\n");  
                break;  
    }  
}
```

```
cmp [ebp+arg_0], 7 ; switch 8 cases  
ja caseunk ; jumtable default case  
mov edx, [ebp+arg_0]  
shl edx, 2  
mov edx, ds:(jumtable - 804A000h)[edx+eax]  
add edx, eax  
jmp edx ; switch jump  
; -----  
_L3:  
; ... <snip> ...  
  
; jump table for switch statement  
jumtable:  
    dd offset _L3 - 804A000h  
    dd offset _L5 - 804A000h  
; ...<snip>...  
    dd offset _L11 - 804A000h
```

- 转化为多个if-then-else
 - case分支较少
 - case分支不连续
 - 例如 case 10: case 100: case 10000:
- 使用跳转表实现
 - 可以视为一个数组
 - 每一项为对应分支的起始地址
 - `mov eax, [table+idx*4]`
`jmp eax`

- 有时IDA Pro无法自动识别出某些使用跳转表的switch-case情况
- 幸好它是Interactive Disassembler
 - 可以手工辅助
 - Edit -> Other -> Specify switch idiom...
- 人工找出跳转表的位置和索引方式



The dialog box titled "Manual switch declaration - Main features" contains the following fields and options:

Address of jump table	0x80485E8	
Number of elements	8	
Size of table element	4	
Element shift amount	0	
Element base value	0x804A000	
Start of the switch idiom	0x804841C	
Input register of switch	edx	
First(lowest) input value	0	(if value table is absent)
Default jump address	0x80484F0	

Below the fields, there are three unchecked checkboxes:

- ☐ Separate value table is present
- ☐ Signed jump table elements
- ☐ Subtract table elements

At the bottom, there are three buttons: OK, Cancel, and Help.

- 参数与返回值如何传递存在一套约定, 称为调用约定 (calling convention)
- 例如cdecl:

```
int callee(int, int, int);

int caller(void)
{
    int ret;

    ret = callee(1, 2, 3);
    ret += 5;
    return ret;
}
```

```
caller:
    push    ebp
    mov     ebp, esp
    ; 将调用参数从右到左压栈
    push    3
    push    2
    push    1
    ; 进行调用
    call    callee
    ; 清理栈上的参数
    add     esp, 12
    ; 返回值在 eax 中
    add     eax, 5
    pop     ebp
    ret
```

- 目的
 - 理解程序的用途与实现方式
- 需要理解对应指令集架构的汇编语言
 - 看懂每一条指令的含义
 - 看懂程序中的控制结构
- 了解对应高级语言中结构的内存布局
- 两种方法
 - “自顶向下”
 - 从输入点开始分析
 - “自底向上”
 - 找到关键函数/变量, 分析引用关系
- 没有文档
- 找参考资料, 学习背景知识
- 调试

```
struct SomeStruct {  
    int a;  
    char b;  
    short c;  
    double d;  
};
```

struct SomeStruct在内存中长什么样？



C++中的类又是怎样呢？

- 若没有虚函数，只有成员变量
 - 跟C中的struct没有任何不同！

- 继承

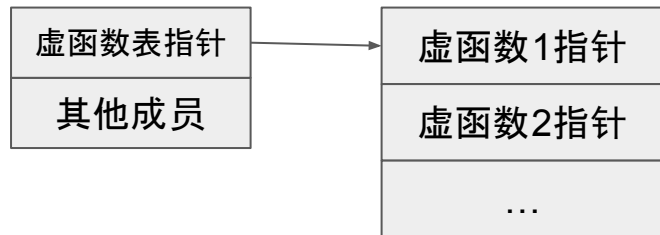
-



- 多继承

- 各编译器实现有区别
 - 基本思路：一张表，存储每个父类的成员的起始偏移

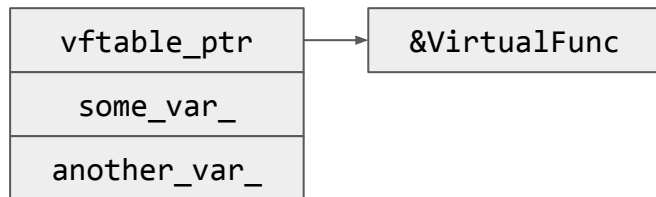
- 存在虚函数的C++类的内存布局:



```
class Example {  
public:  
    int some_var_;  
    int another_var_;  
    virtual void VirtualFunc();  
    void OrdinaryFunc();  
};
```

```
void Example::VirtualFunc() {  
    some_var_ = 1;  
}
```

```
void Example::OrdinaryFunc() {  
    another_var_ = 2;  
}
```



《C++反汇编与逆向分析技术揭秘》钱林松, 赵海旭 著

- 目的
 - 理解程序的用途与实现方式
- 需要理解对应指令集架构的汇编语言
 - 看懂每一条指令的含义
 - 看懂程序中的控制结构
- 了解对应高级语言中结构的内存布局
- 两种方法
 - “自顶向下”
 - 从**输入点**开始分析
 - “自底向上”
 - 找到关键函数/变量, 分析引用关系
- 没有文档
- 找参考资料, 学习背景知识
- 调试

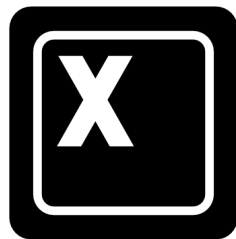
- 找到发起库函数调用、系统调用的代码
 - ltrace
 - strace
- 识别程序中自身包装的一些IO函数
- 从 IO 函数外层的调用栈开始看起
- C++代码？虚函数？
 - 手动去虚表中解析
 - 动态调试

- 目的
 - 理解程序的用途与实现方式
- 需要理解对应指令集架构的汇编语言
 - 看懂每一条指令的含义
 - 看懂程序中的控制结构
- 了解对应高级语言中结构的内存布局
- 两种方法
 - “自顶向下”
 - 从输入点开始分析
 - “自底向上”
 - 找到关键函数/变量, 分析引用关系
- 没有文档
- 找参考资料, 学习背景知识
- 调试

- 黑盒测试
- 关键库函数调用, 系统调用
 - ltrace
 - strace
- 字符串
 - strings
 - Shift + F12 in IDA Pro, 显示字符串
 - 调试信息
 - 可疑的字符串
 - 例如“userRpmNatDebugRpm26525557”
- Magic Number
 - 例如‘qshs’, 搜索该常数来查找解析SquashFS头的代码

- 目的
 - 理解程序的用途与实现方式
- 需要理解对应指令集架构的汇编语言
 - 看懂每一条指令的含义
 - 看懂程序中的控制结构
- 了解对应高级语言中结构的内存布局
- 两种方法
 - “自顶向下”
 - 从输入点开始分析
 - “自底向上”
 - 找到关键函数/变量, 分析引用关系
- 没有文档
- 找参考资料, 学习背景知识
- 调试

- IDA Pro中的 xrefs from 及 xrefs to 功能
- 对.bss段的变量使用
 - 显示所有引用到该变量的代码
- 对函数使用
 - 显示所有直接调用该函数的代码
 - 对于函数是C++中的虚函数等情况, 也会显示函数在哪个虚表中出现
 - 此时考虑追踪虚表的引用情况



- 目的
 - 理解程序的用途与实现方式
- 需要理解对应指令集架构的汇编语言
 - 看懂每一条指令的含义
 - 看懂程序中的控制结构
- 了解对应高级语言中结构的内存布局
- 两种方法
 - “自顶向下”
 - 从输入点开始分析
 - “自底向上”
 - 找到关键函数/变量, 分析引用关系
- 没有文档
- 找参考资料, 学习背景知识
- 调试

- 看关键数据

- 搜索字符串

- 若使用了开源库则通常能找到源码

- 搜索大常数

- 0x67452301 -> MD5/SHA1

- 对密码学算法效果很好

- 看代码的逻辑特征

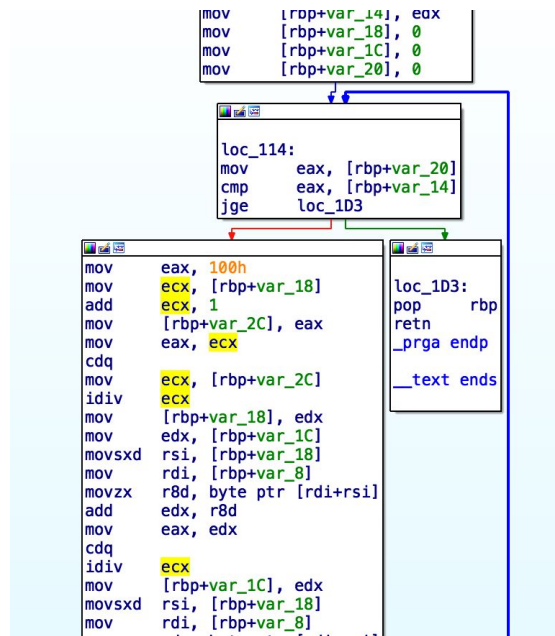
- 除密码学算法以外的算法, 例如 FFT

- 控制流特征

- 密码学算法中也有例外: RC4

- 无法用以上两种方式识别

- 有一个数组、有长度为256的循环、每一步swap了数组[i]和另一个值



- 《Reverse Engineering for Beginners》
 - <https://beginners.re/>
 - 免费且优质的电子书, PDF
 - 强烈推荐
- 《The IDA Pro Book》
 - IDA Pro的使用方法