逆向汪的自我修养

Trinity 李骁
xiao.li@smail.nju.edu.cn

内容一览

- 逆向工程简介
- 前置知识
- 逆向工具IDA基本教程
- 推荐列表
- 作业

内容一览

- 逆向工程简介
- 前置知识
- 逆向工具IDA基本教程
- 推荐列表
- 作业

什么是逆向工程

- Reverse engineering, also called back engineering, is the process by which a man-made object is deconstructed to reveal its designs, architecture, or to extract knowledge from the object; ----- from wikipedia

- 软件代码逆向主要指对软件的结构，流程，算法，代码等进行**逆向拆解和分析**。

- CTF中的逆向题涉及 Windows、Linux、Android 平台的多种编程技术，要求利用常用工具对源代码及二进制文件进行逆向分析，掌握 Android 移动应用APK文件的逆向分析，掌握加解密、内核编程、算法、反调试和代码混淆技术。 ------《全国大学生信息安全竞赛参赛指南》

学逆向，学什么？

- 逆向的目标：ELF文件，PE文件，APK文件，固件，IR.....
- 逆向的原理：编译原理
- 正向的知识：编程语言，算法与数据结构，编译工具链，社会工程学.....
- 逆向的工具：IDA，ollydbg，gdb，Ghidra，mcsema，retdec.....

学逆向，学什么？

- 逆向的目标：ELF文件，PE文件，APK文件，固件，IR.....
- 逆向的原理：编译原理
- 正向的知识：编程语言，算法与数据结构，编译工具链，社会工程学.....
- 逆向的工具：IDA，ollydbg，gdb，Ghidra，mcsema，retdec.....

不要沉迷于使用和追逐工具，懂原理才是硬道理。
工具总有歇菜的时候，到时候还得你手动分析。

内容一览

- 逆向工程简介
- **前置知识**
- 逆向工具IDA基本教程
- 推荐列表
- 作业

编译型语言 vs 解释型语言

编译型语言，源码和汇编代码是1对N的对应关系，逆向的难度较大。

编译型语言 vs 解释型语言

解释型语言，源码和字节码基本是1对1的对应关系，逆向的难度基本等于直接读源码的难度。

ELF文件格式详解

- Unix和类Unix操作系统的标准二进制格式
- 可用于可执行文件，共享库，目标文件，Coredump文件
- 由文件头(File Header)，程序头(Program Header)等定义
- Linux逆向工程的重要基础

ELF文件头

- man elf

The ELF header is described by the type Elf32_Ehdr or Elf64_Ehdr:

```
#define EI_NIDENT 16

typedef struct {
    unsigned char e_ident[EI_NIDENT];
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    ElfN_Addr     e_entry;
    ElfN_Off      e_phoff;
    ElfN_Off      e_shoff;
    uint32_t      e_flags;
    uint16_t      e_ehsize;
    uint16_t      e_phentsize;
    uint16_t      e_phnum;
    uint16_t      e_shentsize;
    uint16_t      e_shnum;
    uint16_t      e_shstrndx;
} ElfN_Ehdr;
```

ELF文件类型(e_type)

- ET_NONE：未知类型，表明文件类型不确定或未定义
- ET_REL: 重定位文件，是还没有被链接到可执行程序的一段位置独立代码(通常为.o后缀)
- ET_EXEC: 可执行文件。
- ET_DYN：共享目标文件，也称共享库(.so)，这类文件会在程序运行时被装载并链接到程序的进程镜像中
- ET_CORE : 核心文件。在程序崩溃或进程传递了一个SIGSEGV信号时，会把整个进程的镜像信息记录在核心文件中。这类文件可以用GDB来调试

例子

```
➜  test readelf -h ./test
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:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4005f0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6848 (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:         29
  Section header string table index: 26
```

ELF程序头

- ELF程序头时对二进制文件中段(segment)的描述，是程序装载的必需部分。它描述了可执行文件的内存布局以及如何被映射到内存中。

```
typedef struct {
                uint32_t   p_type;
                uint32_t   p_flags;
                Elf64_Off  p_offset;
                Elf64_Addr p_vaddr;
                Elf64_Addr p_paddr;
                uint64_t   p_filesz;
                uint64_t   p_memsz;
                uint64_t   p_align;
            } Elf64_Phdr;
```

p_type

- 这个成员描述了段的类型。常见的有以下的几种类型
  - *PT_LOAD：一个可执行文件至少有一个PT_LOAD类型的段。这类程序头描述的是可装载的段。这就是说，这种类型的段将被装载或映射到内存中。例如，一个需要动态链接的ELF可执行文件通常包含以下两个装载的段(类型为PT_LOAD):*
    - 存放程序代码的text段
    - 存放全局变量和动态链接信息的data段
  - *PT_DYNAMIC: 动态段。动态段是动态链接可执行文件所特有的，包含了动态链接器所必需的一些信息，例如*
    - 运行时所需要链接的共享库列表
    - 全局偏移表(GOT)的地址
    - 重定位条目的相关信息

p_type

- 这个成员描述了段的类型。常见的有以下的几种类型
  - *PT_NOTE：保存了操作系统的规范信息，执行时是不需要这个段的，因此这个段常常成为病毒感染的一个目标*
  - *PT_INTERP: 对程序解释器位置的描述，如/lib/ld-linux.so.2*
  - *PT_PHDR: 保存了程序头表本身的位置和大小。*

例子

- readelf -l ./test

```
→  test readelf -l ./test

Elf file type is EXEC (Executable file)
Entry point 0x4005f0
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E    8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x000000000000099c 0x000000000000099c  R E    200000
  LOAD           0x0000000000000de0 0x0000000000600de0 0x0000000000600de0
                 0x0000000000000230 0x0000000000000270  RW     200000
  DYNAMIC        0x0000000000000df8 0x0000000000600df8 0x0000000000600df8
                 0x00000000000001c0 0x00000000000001c0  RW     8
  NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_EH_FRAME   0x0000000000000848 0x0000000000400848 0x0000000000400848
                 0x000000000000003c 0x000000000000003c  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x0000000000000de0 0x0000000000600de0 0x0000000000600de0
                 0x0000000000000220 0x0000000000000220  R      1
```

ELF节头(Section Header)

- 节头(section header)和段头(segment header)是不一样的。段是程序执行的必要组成部分，在每个段中，会有代码或数据被划分成不同的节（即节是在段上的更细划分）。
- 节头表是对这些节的位置和大小的描述，主要用于链接和调试。但不是程序执行所必须的。它只是对程度头的补充。

```
typedef struct {
    uint32_t   sh_name;
    uint32_t   sh_type;
    uint64_t   sh_flags;
    Elf64_Addr sh_addr;
    Elf64_Off  sh_offset;
    uint64_t   sh_size;
    uint32_t   sh_link;
    uint32_t   sh_info;
    uint64_t   sh_addralign;
    uint64_t   sh_entsize;
} Elf64_Shdr;
```

常见节

- .text: 保存了程序代码指令的代码节
- .rodata: 保存了只读数据，如C语言代码中的常量字符串。
- .plt节: 包含了动态链接器调用从共享库导入的函数所必须的相关代码。
- .data: 保存了初始化了的全局变量等数据。
- .bss : 保存了未进行初始化的全局数据，在文件中占用的空间不超过4个字节，仅表示这个节本身的空间。程序加载时数据被初始化为0.

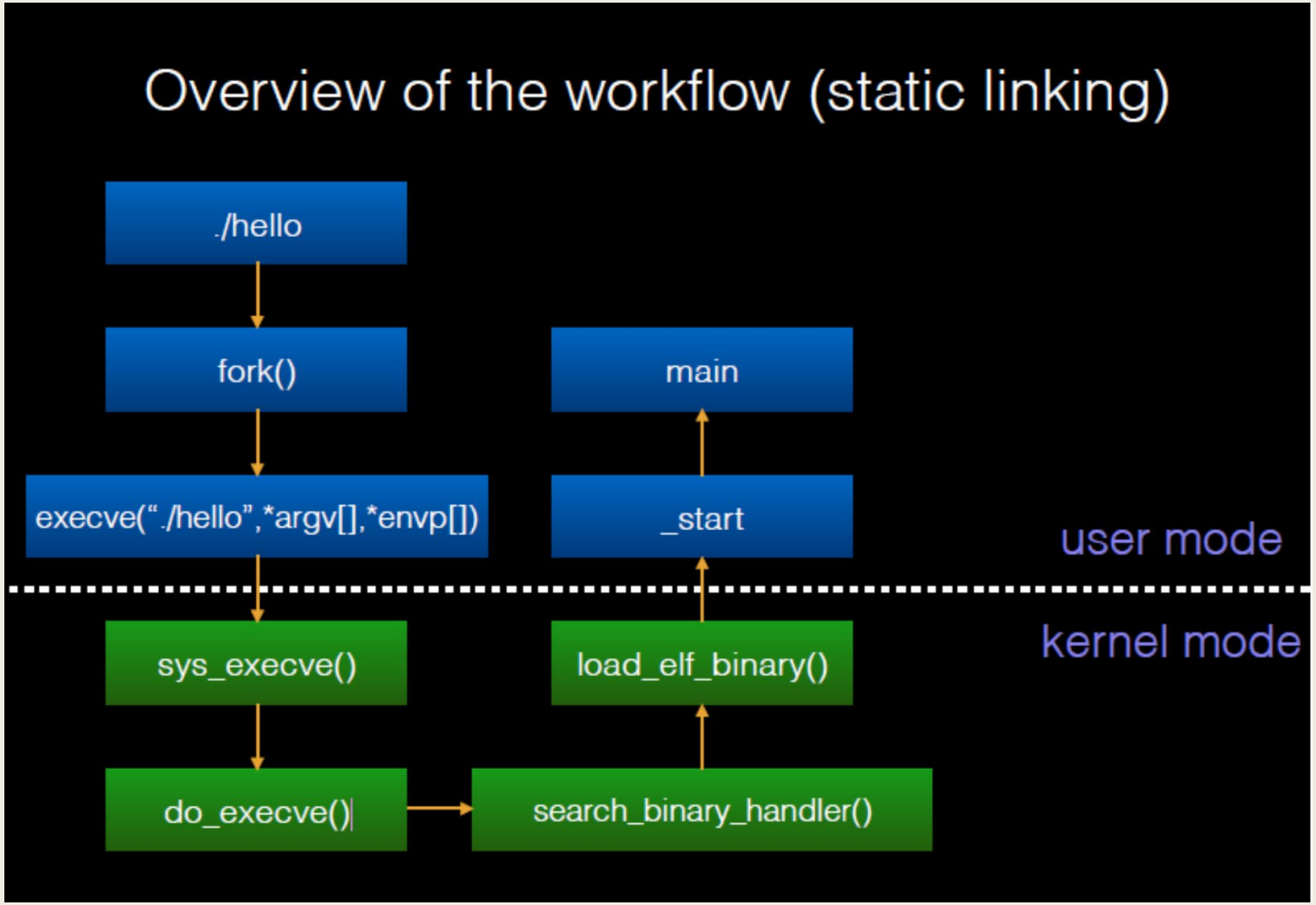
常见节

- .got.plt节: 保存了全局偏移表。.got和.plt节一起提供了对导入的共享库函数的访问入口，由动态链接器在运行时进行修改。
- .dynsym节: 保存了从共享库导入的动态符号表信息。
- .dynstr节: 保存了动态符号字符串表，表中存放了一系列字符串，这些字符串代表了符号的名称，以空字符作为终止符。
- .init_array和. fini_array节：. init_array（构造器）和.fini_array (析构器)这两个节保存了指向构造函数和析构函数的函数指针。构造函数时在main函数执行之前需要执行的代码，析构函数时在main函数之后需要执行的代码。

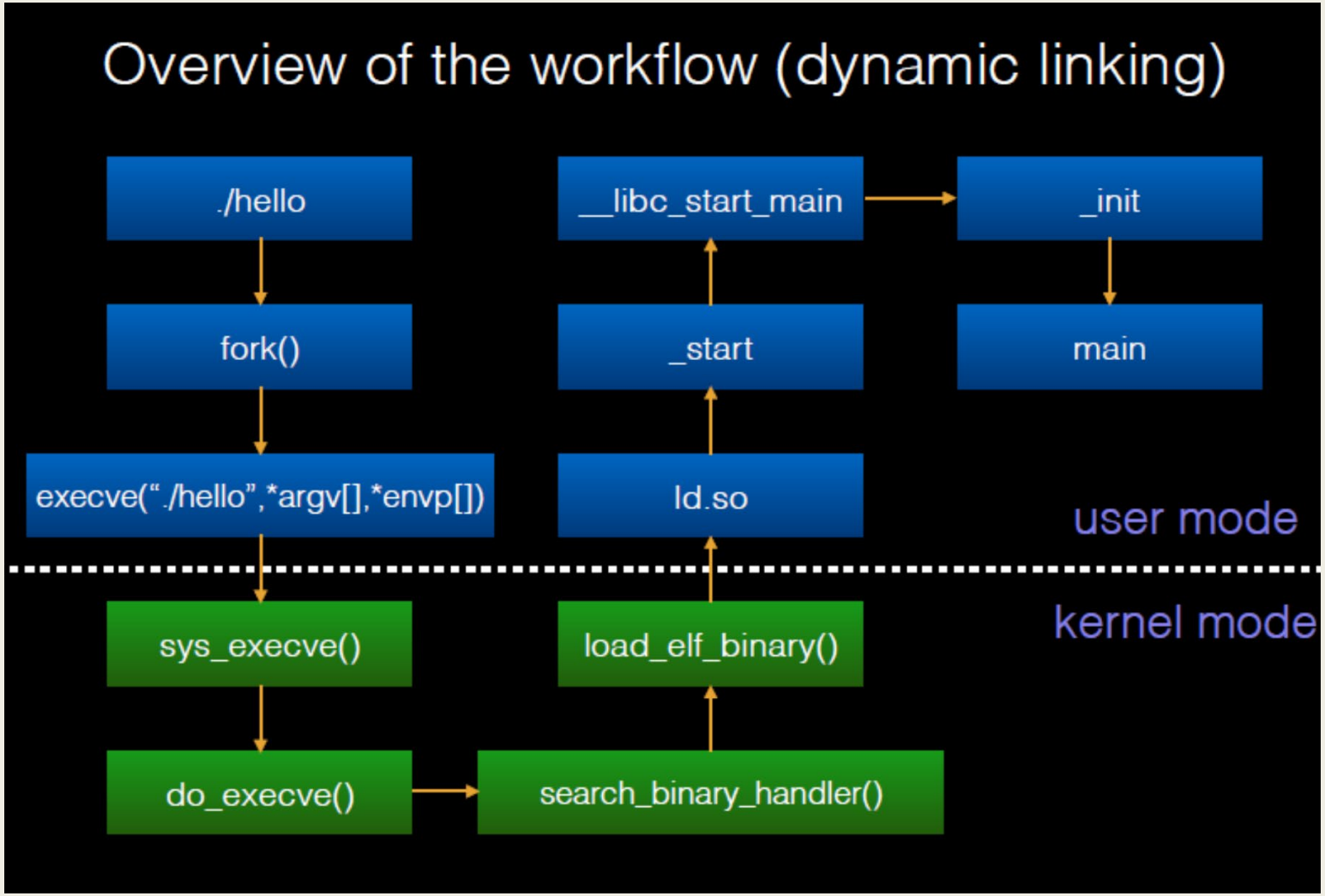
ELF文件的加载执行过程

- 我们知道，常见的不同的ELF格式映像。
  - 一种是静态链接的，在装入/启动其运行时无需装入函数库映像、也无需进行动态连接。
  - 另一种是动态连接，需要在装入/启动其运行时同时装入函数库映像并进行动态链接
- 因此，GNU把对于动态链接ELF映像的支持作了分工：把ELF映像的装入/启动入在Linux内核中；而把动态链接的实现放在用户空间（glibc），并为此提供一个称为”解释器”(ld-linux.so.2)的工具软件，而解释器的装入/启动也由内核负责

静态链接


Overview of the workflow (static linking)
./hello
fork()
execve(“./hello”,*argv[],*envp[])
sys_execve()
do_execve()
search_binary_handler()
load_elf_binary()
_start
main
user mode
kernel mode

动态链接


Overview of the workflow (dynamic linking)
./hello
fork()
execve(“./hello”,*argv[],*envp[])
sys_execve()
do_execve()
search_binary_handler()
load_elf_binary()
ld.so
_start
__libc_start_main
_init
main
user mode
kernel mode

汇编基础

- 常见的汇编指令集有:
  - *X86/x64*
  - *ARM/ARM64(Arch64)*
  - *Mips*
  - *其它*
- 每一条汇编指令的功能都非常简单，我们只需要掌握常用的汇编指令即可
- 需要详细的用法解释可以查手册

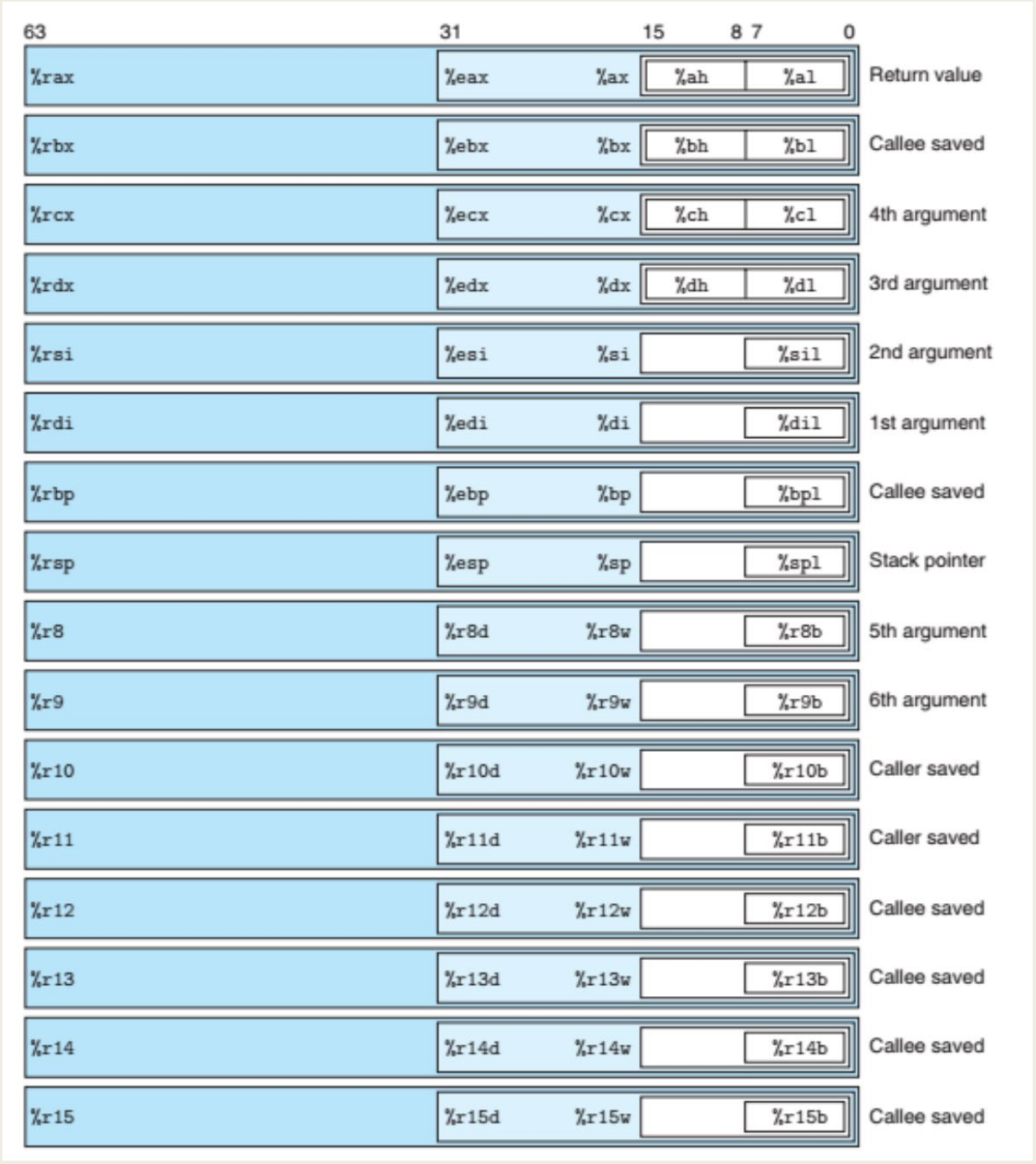
常用指令

- 数学运算
  - 加减乘除(add / sub /mul /div)
  - 与或非(and / or / not)
  - 异或(xor)
- 数据转移指令
  - 读写内存
  - 读写寄存器
- 控制流转移指令
  - 无/有条件跳转(jmp / jge/jle/jne/je)
- 栈操作
  - 入栈/出栈(push / pop)

常用指令(x86为例)

- Call addr→ push eip ; jmp addr
- Leave → mov esp ebp, ret
- 注意不同风格的汇编操作数的位置变化
  - *AT&T： movl $8, %eax*
  - *Intel： mov eax, 8*

X86/64寄存器


63
31
15
8 7
0
%rax
%eax
%ax
%ah
%al
Return value
%rbx
%ebx
%bx
%bh
%bl
Callee saved
%rcx
%ecx
%cx
%ch
%cl
4th argument
%rdx
%edx
%dx
%dh
%dl
3rd argument
%rsi
%esi
%si
%sil
2nd argument
%rdi
%edi
%di
%dil
1st argument
%rbp
%ebp
%bp
%bpl
Callee saved
%rsp
%esp
%sp
%spl
Stack pointer
%r8
%r8d
%r8w
%r8b
5th argument
%r9
%r9d
%r9w
%r9b
6th argument
%r10
%r10d
%r10w
%r10b
Caller saved
%r11
%r11d
%r11w
%r11b
Caller saved
%r12
%r12d
%r12w
%r12b
Callee saved
%r13
%r13d
%r13w
%r13b
Callee saved
%r14
%r14d
%r14w
%r14b
Callee saved
%r15
%r15d
%r15w
%r15b
Callee saved

X86/x64函数调用传参规则(Linux)

- X86：栈传参
  - 如 *func(1, 2, 3, 4)*
    - push 4
    - push 3
    - push 2
    - push 1
    - call func
- X64: 寄存器加栈
  - *前6个参数分别在rdi, rsi, rdx, rcx, r8, r9*
  - *第七个或以上: 栈传参*

内容一览

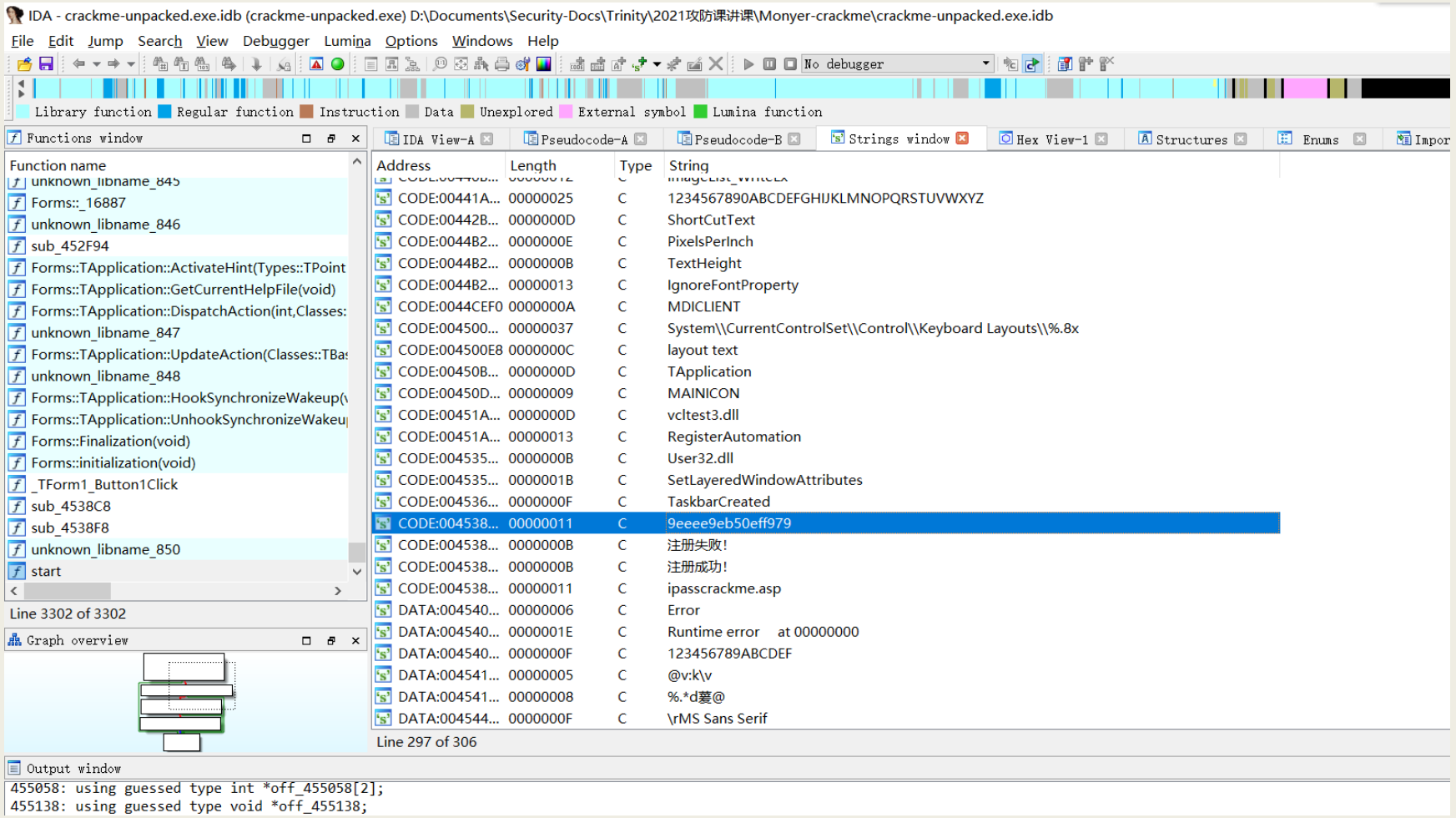
- 逆向工程简介
- 前置知识
- **逆向工具IDA基本教程**
- 推荐列表
- 作业

IDA的使用

- 常用快捷键
  - *n: 修改名字*
  - *y: 修改定义*
  - */:输入注释*
  - *F5：反编译为源代码*
- 常用技巧:
  - *编写结构体*
  - *静态函数匹配*
- 实例讲解

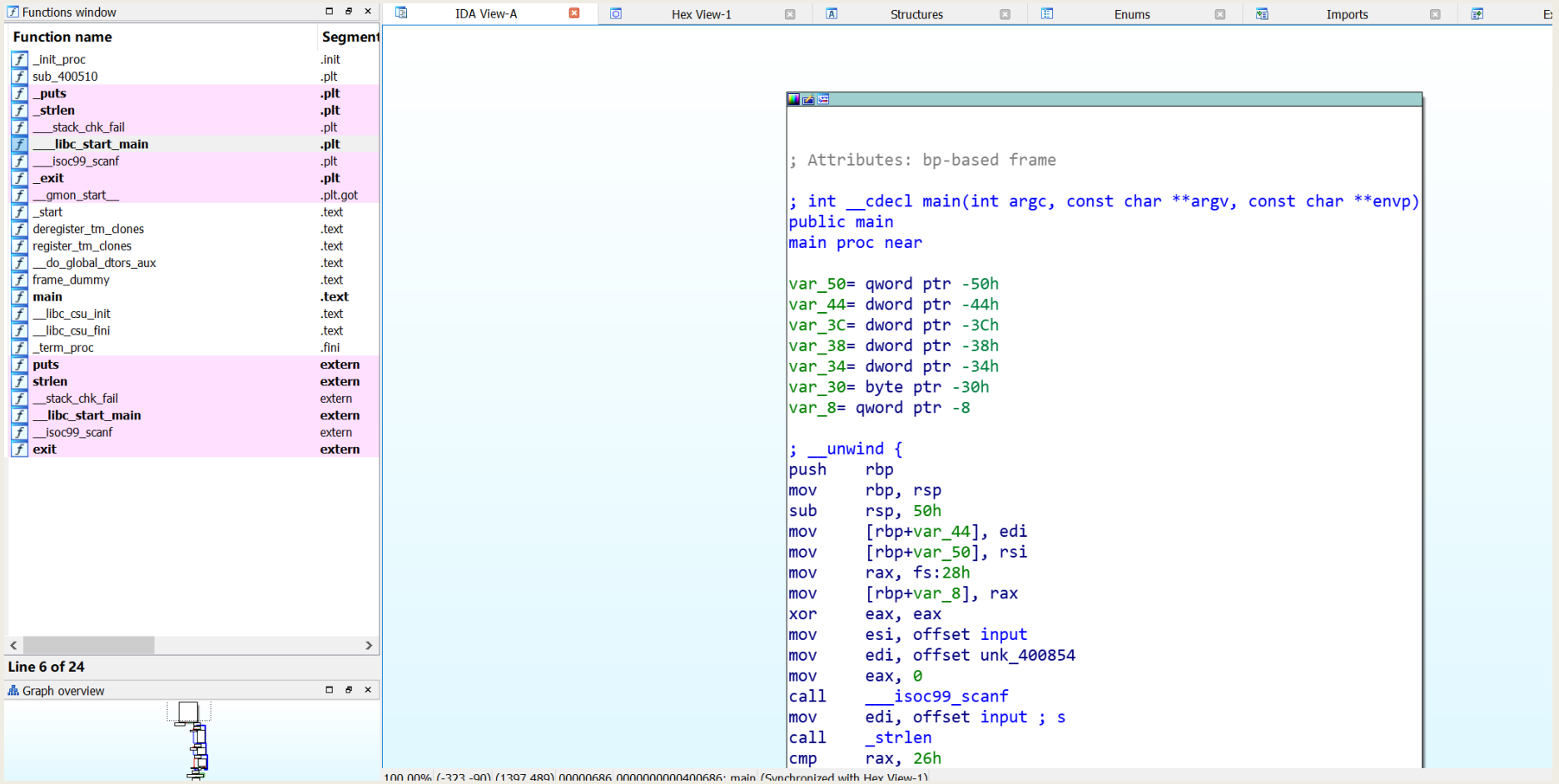
案例1： Monyer’s game - crackme

- PEID/Exeinfo查壳
- Upx脱壳
- IDA string



案例2：SimpleXor

- IDA64打开



- F5反编译

unk_400854即”%s”

程序scanf一个字符串，长度为38

如果长度不对则退出程序

接着进入19次的一个循环

可以看到对v7数组进行了赋值

v7数组即为input数组奇偶位交换

v7[0]=input[1];

v7[1]=input[0];

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    signed int i; // [rsp+14h] [rbp-3Ch]
4    signed int j; // [rsp+18h] [rbp-38h]
5    signed int k; // [rsp+1Ch] [rbp-34h]
6    char v7[40]; // [rsp+20h] [rbp-30h]
7    unsigned __int64 v8; // [rsp+48h] [rbp-8h]
8
9    v8 = __readfsqword(0x28u);
10   __isoc99_scanf(&unk_400854, input, envp);
11   if ( strlen(input) != 38 )
12   {
13     puts("You lost!");
14     exit(0);
15   }
16   for ( i = 0; i <= 18; ++i )
17   {
18     v7[2 * i] = input[2 * i + 1];
19     v7[2 * i + 1] = input[2 * i];
20   }
21   for ( j = 0; j <= 37; ++j )
22     v7[j] ^= j;
23   for ( k = 0; k <= 37 && v7[k] == (unsigned __int8)toCmp[k]; ++k )
24     ;
25   if ( k == 38 )
26   {
27     puts("You win!Your flag is");
28     puts(input);
29   }
30   return 0;
31 }
```

■ 分析完简单的交换顺序逻辑

下一个38次的循环将

v7[i]异或当前循环计数i

也就是

v7[0]^=(unsigned char)0;

v7[1]^=(unsigned char)1;

.....

获得38长度的密文，最终的循环

是和toCmp数组进行比较

如果每一个密文都和预期结果相等

则输出flag

由于异或是可逆的，只需要将密文再

异或一次i即可还原

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    signed int i; // [rsp+14h] [rbp-3Ch]
4    signed int j; // [rsp+18h] [rbp-38h]
5    signed int k; // [rsp+1Ch] [rbp-34h]
6    char v7[40]; // [rsp+20h] [rbp-30h]
7    unsigned __int64 v8; // [rsp+48h] [rbp-8h]
8
9    v8 = __readfsqword(0x28u);
10   __isoc99_scanf(&unk_400854, input, envp);
11   if ( strlen(input) != 38 )
12   {
13     puts("You lost!");
14     exit(0);
15   }
16   for ( i = 0; i <= 18; ++i )
17   {
18     v7[2 * i] = input[2 * i + 1];
19     v7[2 * i + 1] = input[2 * i];
20   }
21   for ( j = 0; j <= 37; ++j )
22     v7[j] ^= j;
23   for ( k = 0; k <= 37 && v7[k] == (unsigned __int8)toCmp[k]; ++k )
24     ;
25   if ( k == 38 )
26   {
27     puts("You win!Your flag is");
28     puts(input);
29   }
30   return 0;
31 }
```

exp：

- 先找到toCmp数组

然后依次对异或、下标变换

进行逆操作即可

```
.data:0000000000601080 toCmp            db 72h
.data:0000000000601081                  db  55h ; U
.data:0000000000601082                  db  6Ch ; l
.data:0000000000601083                  db  6Ah ; j
.data:0000000000601084                  db  70h ; p
.data:0000000000601085                  db  6Ch ; l
.data:0000000000601086                  db  7Dh ; }
.data:0000000000601087                  db  7Eh ; ~
.data:0000000000601088                  db  69h ; i
.data:0000000000601089                  db  64h ; d
.data:000000000060108A                  db  62h ; b
.data:000000000060108B                  db  68h ; h
.data:000000000060108C                  db  62h ; b
.data:000000000060108D                  db  64h ; d
.data:000000000060108E                  db  51h ; Q
.data:000000000060108F                  db  6Ah ; j
.data:0000000000601090                  db  7Fh ;
.data:0000000000601091                  db  68h ; h
.data:0000000000601092                  db  4Dh ; M
.data:0000000000601093                  db  66h ; f
.data:0000000000601094                  db  66h ; f
.data:0000000000601095                  db  74h ; t
.data:0000000000601096                  db  49h ; I
.data:0000000000601097                  db  72h ; r
.data:0000000000601098                  db  77h ; w
.data:0000000000601099                  db  6Ah ; j
.data:000000000060109A                  db  78h ; x
.data:000000000060109B                  db  44h ; D
.data:000000000060109C                  db  7Dh ; }
.data:000000000060109D                  db  78h ; x
.data:000000000060109E                  db  6Ah ; j
.data:000000000060109F                  db  6Ah ; j
```

exp：

```c
#include <stdio.h>

char toCmp[38] =
{
    0x100-0x8e, 0x55, 0x6c, 0x6a, 0x70, 0x6c, 0x7d,
    0x7e, 0x69, 0x64, 0x62, 0x68, 0x62, 0x64, 0x51,
    0x6a, 0x7f, 0x68, 0x4d, 0x66, 0x66, 0x74, 0x49,
    0x72, 0x77, 0x6a, 0x78, 0x44, 0x7d, 0x78, 0x6a,
    0x6a, 0x46, 0x48, 0x4e, 0x56, 0x59, 4
};

int main()
{
    char v[38];
    for (int i = 0; i < 38; i++)
    {
        toCmp[i] ^= i;
    }
    for (int i = 0; i < 19; i++)
    {
        v[2*i] = toCmp[2*i+1];
        v[2*i+1] = toCmp[2*i];
    }
    for (int i = 0; i < 38; i++)
    {
        printf("%c", v[i]);
    }
    printf("\n");
    return 0;
}
```

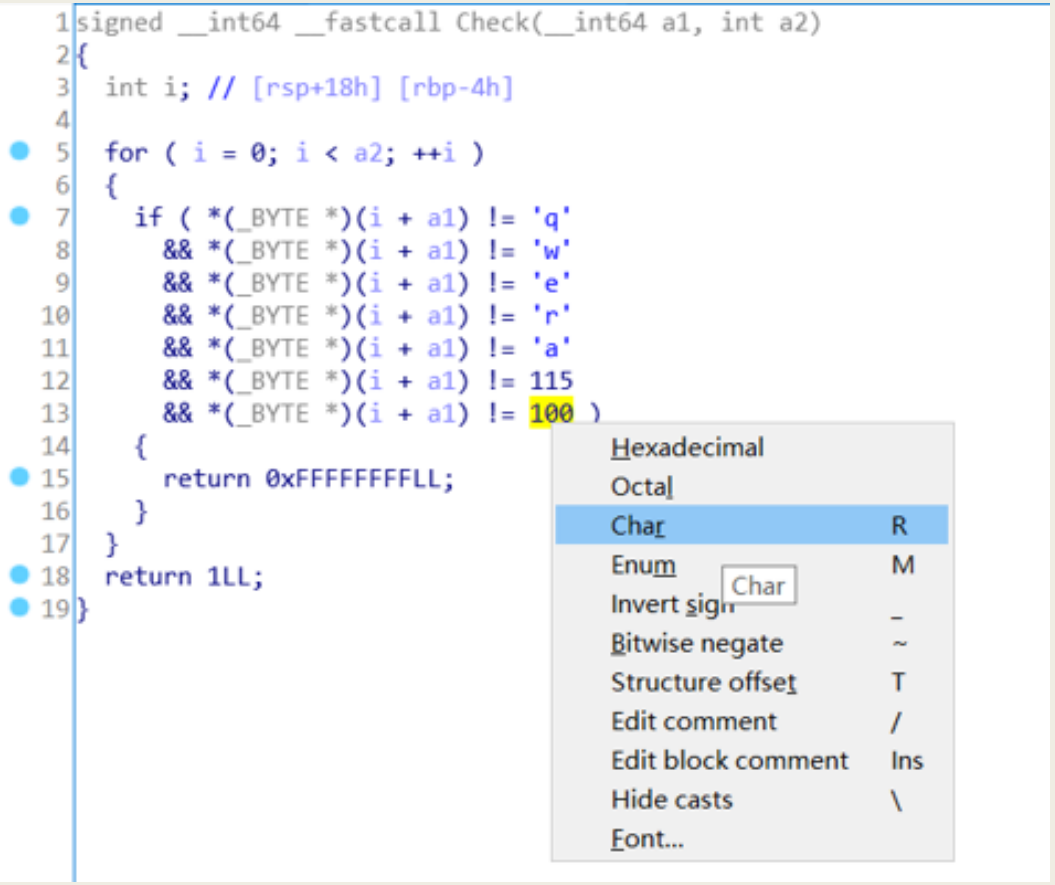
Trinity{machine_you_are_so_beautiful!}

案例3：Maze

- 先把Maze丢进IDA，找main函数，F5大法
- main中输入字符串后，先经历Check函数

```
13
14  v13 = __readfsqword(0x28u);
15  printf("Please input your solution: ", argv, envp);
16  __isoc99_scanf("%50s", s);
17  v3 = (unsigned int)strlen(s);
18  if ( (unsigned int)Check(s, v3) != 1 )
19  {
20    puts("Input is valid");
21    exit(0);
22  }
```

- 进入Check函数，发现是限制输入为qwerasd
- 可以用下图的技巧转数字为ascii字符，省的自己查表



- 回到main，接下来是对一个全局数组paMcnE进行处理
- 对其中的每一个元素依次调simple_funcN进行处理

```
23  for ( i = 0; i <= 9; ++i )
24  {
25    for ( j = 0; j <= 9; ++j )
26    {
27      paMcnE[j + 10LL * i] = simple_func1(paMcnE[j + 10LL * i], (unsigned int)j);
28      paMcnE[j + 10LL * i] = simple_func2(paMcnE[j + 10LL * i], (unsigned int)i);
29      paMcnE[j + 10LL * i] = simple_func3(paMcnE[j + 10LL * i], (unsigned int)(2 * j));
30      paMcnE[j + 10LL * i] = simple_func4(paMcnE[j + 10LL * i], (unsigned int)i);
31    }
32  }
```

- 查看simple_funcN函数
- simple_func1(a,b) = a + 4b
- simple_func2(a,b) = a - 3b
- simple_func3(a,b) = ~((a|~b)&(~a|b))
  - *a = 0, b = 0, f3 = 0*
  - *a = 0, b = 1, f3 = 1*
  - *a = 1, b = 0, f3 = 1*
  - *a = 1, b = 1, f3 = 0*
  - *推广到32bit，发现是异或*
- simple_func4(a,b) = a^b

```
__int64 __fastcall simple_func3(int a1, int a2)
{
  return ~((a1 | ~a2) & (a2 | (unsigned int)~a1));
}
```

- 查看全局数组paMcnE的初值，理解为10*10的二维数组，处理函数等价于图1
- 结合初值，然后根据上面的结果推理paMcnE被处理后的值为图2

```
for (int i = 0; i < 10; i++)
{
    for (int j = 0; j < 10; j++)
    {
        map[i][j] ^= i;
        map[i][j] ^= 2*j;
        map[i][j] += 3*i;
        map[i][j] -= 4*j;
    }
}
```

图1

```
{5, 5, 5, 5, 5, 5, 5, 5, 5, 5, },
{5, 1, 0, 1, 0, 0, 0, 0, 0, 5, },
{5, 0, 0, 1, 0, 1, 0, 1, 0, 5, },
{5, 1, 0, 1, 0, 1, 0, 1, 1, 5, },
{1, 1, 0, 0, 0, 1, 1, 1, 0, 5, },
{5, 0, 0, 1, 1, 0, 1, 0, 1, 5, },
{5, 0, 1, 0, 0, 0, 0, 0, 0, 7, },
{5, 0, 1, 0, 1, 1, 0, 1, 1, 5, },
{5, 0, 0, 0, 1, 0, 0, 1, 0, 5, },
{5, 5, 5, 5, 5, 5, 5, 5, 5, 5, },
```

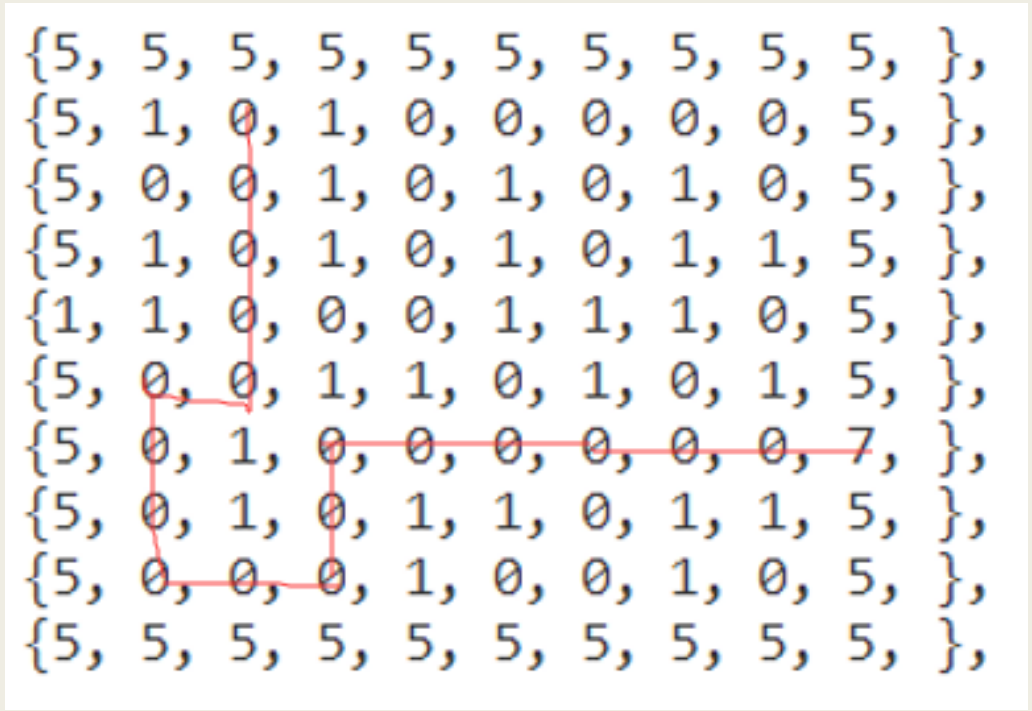
图2

- 回到main继续
- 发现根据输入内容，决定v7和v8的值，累积v9的值
- 然后判断paMcnE[v7][v8]的值来决定是否是正确路径
- 结合simple_funcN的结果
  - *对于wasd，即在数组中上左下右移动*
  - *qe原地不动*
  - *r会归零*

```
v7 = 2;
v8 = 1;
v9 = 0;
for ( k = 0; k < strlen(s); ++k )
{
  ++v9;
  switch ( s[k] )
  {
    case 'q':
      v7 += simple_func4(s[k], s[k]);
      v8 += simple_func3(s[k], s[k]);
      break;
    case 'w':
      v7 -= simple_func1(1, 0);
      v8 -= simple_func4(7, 7u);
      break;
    case 'e':
      v7 -= simple_func3(s[k], s[k]);
      v8 -= simple_func4(s[k], s[k]);
      break;
    case 'r':
      v7 *= (unsigned int)simple_func4(s[k], s[k]);
      v8 *= (unsigned int)simple_func4(3, 3u);
      break;
    case 'a':
      v7 -= simple_func3(5, 5);
      v8 -= simple_func2(4, 1);
      break;
    case 's':
      v7 += simple_func1(-11, 3);
      v8 += simple_func4(3, 3u);
      break;
    case 'd':
      v7 += simple_func1(-8, 2);
```

- 下面又是合法判断，paMcnE[v7][v8] = 5/4/3/1都是错误路径，而7是正确的
- 同时要求步长v9<=19步
- 所以目的是从（1,2）的位置，走到值为7的地方，且不碰到5431，其实只有5和1
- 所以沿着0一直走就完事了，dsssasssddwwdddddd
- flag就是Trinity{dsssasssddwwdddddd}

```
  if ( v7 < 0 || v8 < 0 || v7 > 9 || v8 > 9 )
  {
    puts("Wrong path");
    exit(0);
  }
  if ( paMcnE[v8 + 10LL * v7] == 5
    || paMcnE[v8 + 10LL * v7] == 4
    || paMcnE[v8 + 10LL * v7] == 3
    || paMcnE[v8 + 10LL * v7] == 1 )
  {
    puts("Try again");
    exit(0);
  }
  if ( paMcnE[v8 + 10LL * v7] == 7 && v9 <= 19 && v9 == strlen(s) )
  {
    puts("You win, the flag is");
    printf("Trinity{");
    for ( l = 0; l < strlen(s); ++l )
      putchar(s[l]);
    putchar(125);
    putchar(10);
    exit(0);
  }
}
```



内容一览

- 逆向工程简介
- 前置知识
- 逆向工具IDA基本教程
- **推荐列表**
- 作业

LLVM IR

- 编译路线：源码 => clang前端 => LLVM IR => llc后端 => 可执行文件
- LLVM IR的出现使得编译优化和反向优化都变得比以前容易了
- LLVM IR入门教程：https://zhuanlan.zhihu.com/c_1267851596689457152
- 基于LLVM的逆向工具：
  - mcsema：https://github.com/lifting-bits/mcsema
  - retdec：https://github.com/avast/retdec
  - llvm-mctoll：https://github.com/Microsoft/llvm-mctoll

逆向学习资源

- XCTF攻防世界：https://adworld.xctf.org.cn/
- BUUCTF平台：https://buuoj.cn/
- CTF Time：https://ctftime.org/
- CTF Wiki：https://wiki.x10sec.org/reverse/introduction/
- 看雪论坛：https://bbs.pediy.com/
- 吾爱破解：https://www.52pojie.cn/

内容一览

- 逆向工程简介
- 前置知识
- 逆向工具IDA基本教程
- 推荐列表
- **作业**

作业

- 作业内容：完成2道逆向题：card和maze
- 题目地址：https://box.nju.edu.cn/d/6301648ab22b4db7be7a/

- 提交内容：提交一份pdf格式的解题报告，其中包含你的解题思路和得到的flag，没有字数要求，能体现你的思考即可。
- 提交文件名：<学号>-<姓名>-RE.pdf，如171860000-张三-RE.pdf
- 提交邮箱：xiao.li@smail.nju.edu.cn，截止日期：2021.12.3

Q & A