Лекция 16

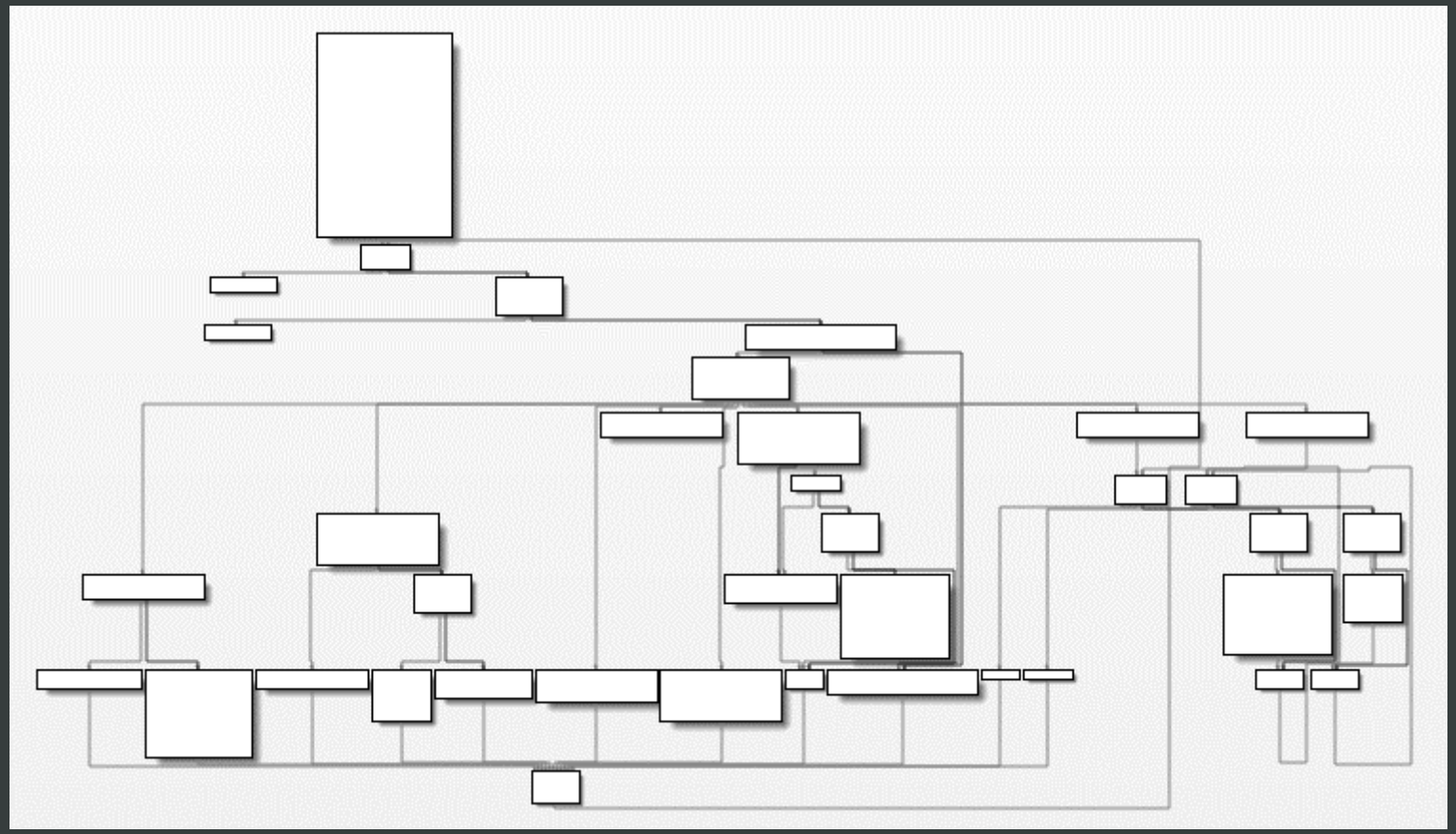
ПРО ДЕКОМПИЛЯТОРЫ, ФОРМАТЫ ИСПОЛНЯЕМЫХ ФАЙЛОВ, БИБЛИОТЕКИ
А ТАКЖЕ НЕМНОГО О ДИНАМИЧЕСКОМ АНАЛИЗЕ

rev. 5

Проблемы традиционной обратной разработки

- Чтение кода на языке ассемблера может быть очень утомительным
  - Особенно в случаях функций с большим числом ветвлений, тут даже отображение в виде графа может не слишком помочь
- Более того, ассемблер может оказаться для аппаратной архитектуры, ассемблер которой вы не знаете
  - А желания изучать новую архитектуру ради одной программы у вас нет
- В целом сложно держать в голове суть происходящего при большом количестве кода и переменных
  - Если только вы не записываете все наблюдения, на что обычно нет времени

Проблемы традиционной обратной разработки



Декомпиляторы

- Декомпиляторы – программы, воссоздающие исходный код (обычно на языке Си) по доступному исполняемому коду
  - Понятно, что на выходе получается скорее псевдокод, так как огромное количество информации теряется в процессе компиляции
- Соответственно, декомпиляторы решают вышеупомянутые проблемы
  - Правда декомпиляция привносит свои трудности: далеко не всегда вывод декомпиляторов корректен, чего обычно нельзя сказать о листинге дизассемблера
- К известным декомпиляторам для машинного кода относятся:
  - Hex-Rays (IDA)
  - Ghidra
  - Binary Ninja
  - Jsdec (Rizin)
  - RetDec

Hex-Rays (IDA)

- Дополнение к дизассемблеру IDA
- Существуют версии для архитектур x86, ARM, PowerPC, MIPS и их 64-битных версий
- Является одним из наиболее старых представителей этого класса продуктов, имеет, скорее всего, самое высокое качество декомпиляции
- Активируется нажатием F5 в IDA
- Очень платный (как и IDA)
- Облачный декомпилятор для x86-64 доступен в составе бесплатной версии IDA, которую можно скачать по ссылке https://www.hex-rays.com/ida-free/

Ghidra

- Декомпилятор Ghidra не имеет отдельного названия и является частью одноименного дизассемблера
- Поддерживает все архитектуры, поддерживаемые дизассемблером
  - Причем, можно даже дописать свою https://habr.com/ru/companies/pt/articles/514292/
- Качество декомпиляции достаточно приемлемое, однако иногда сильно уступает Hex-Rays
- Активируется автоматически в правой части интерфейса дизассемблера
- Доступен абсолютно бесплатно

Binary Ninja

- Менее известный декомпилятор и дизассемблер
- Поддерживает x86 / x86_64, ARM / ARM64, PowerPC и MIPS
- До появления Ghidra можно было считать более дешевой альтернативой IDA, качество декомпиляции неплохое
- Активируется автоматически, вид отображения можно выбрать в меню над кодом
- Имеет сразу две бесплатных версии, облачную и офлайн, платная стоит $300
- Скачать можно по ссылке https://binary.ninja/free/ (на rutracker тоже есть, но скорее всего оно того не стоит)

Jsdec (Rizin)

- Декомпилятор Rizin / Cutter
- Много что поддерживает, но на текущий момент это не особо важно
- Качество декомпиляции так себе, декомпилятор генерирует достаточно низкоуровневый вывод, но появился недавно, так что есть надежды на улучшение
- Активируется вкладкой «декомпилятор» под кодом в Cutter
  - Также Cutter имеет встроенный декомпилятор Ghidra, его можно активировать, выбрав его в этом же окне справа снизу
- Доступен абсолютно бесплатно

RetDec

- Полное название Retargetable Decompiler
- Чистый декомпилятор, разработан Avast
- Поддерживает те же архитектуры, что и Hex-Rays (только зачем-то еще и PIC32)
- Качество декомпиляции – очень так себе
  - Возможно, я просто не умею им пользоваться
- Запускается отдельно командой retdec-decompiler.py <имя_файла>
  - Не работает если в пути файла есть пробелы
- Доступен абсолютно бесплатно на https://github.com/avast/retdec
  - Но не особо развивается, судя по описанию и моим наблюдениям

IDA

```
int __fastcall main(int argc, const char **argv)
{
  int v4; // [rsp+Ch] [rbp-C4h]
  int v5; // [rsp+10h] [rbp-C0h]
  int i; // [rsp+14h] [rbp-BCh]
  int j; // [rsp+18h] [rbp-B8h]
  int v8; // [rsp+1Ch] [rbp-B4h]
  int v9; // [rsp+20h] [rbp-B0h]
  int v10; // [rsp+24h] [rbp-ACh]
  void *v11; // [rsp+28h] [rbp-A8h]
  void *ptr[16]; // [rsp+30h] [rbp-A0h]
  char v13[24]; // [rsp+B0h] [rbp-20h] BYREF
  unsigned __int64 v14; // [rsp+C8h] [rbp-8h]

  v14 = __readfsqword(0x28u);
  v4 = 3;
  v5 = 0;
  setup();
  printf("{?} Enter name: ");
  read_buf(user, 64);
  printf("{?} Enter secret: ");
  read_buf(v13, 16);
LABEL_2:
  while ( v4 > 0 )
  {
    menu();
    v8 = read_int();
    if ( v8 == -559038737 )
      exit(-1);
    switch ( v8 )
    {
      case 1:
        if ( v5 <= 15 )
        {
          v11 = malloc(0x30uLL);
          printf("{?} Enter password: ");
          read_buf(v11, 48);
          ptr[v5++] = v11;
        }
        else
        {
          puts("{-} No available space!");
        }
        goto LABEL_2;
```

Ghidra

```
void main(void)
{
  undefined4 uVar1;
  uint uVar2;
  int iVar3;
  void *pvVar4;
  long in_FS_OFFSET;
  int local_cc;
  int local_c8;
  uint local_c4;
  int local_c0;
  void *apvStack_a8 [16];
  undefined local_28 [24];
  undefined8 local_10;

  local_10 = *(undefined8 *)(in_FS_OFFSET + 0x28);
  local_cc = 3;
  local_c8 = 0;
  setup();
  printf("{?} Enter name: ");
  read_buf(user,0x40);
  printf("{?} Enter secret: ");
  read_buf(local_28,0x10);
  while (0 < local_cc) {
    menu();
    uVar1 = read_int();
    switch(uVar1) {
    case 0:
      local_cc = local_cc + -1;
      break;
    case 1:
      if (local_c8 < 0x10) {
        pvVar4 = malloc(0x30);
        printf("{?} Enter password: ");
        read_buf(pvVar4,0x30);
        apvStack_a8[local_c8] = pvVar4;
        local_c8 = local_c8 + 1;
      }
      else {
        puts("{-} No available space!");
      }
      break;
    case 2:
      printf("{?} Enter password id: ");
      uVar2 = read_int();
```

BinaryNinja

```
int32_t main(int32_t argc, char** argv, char** envp)
{
  void* fsbase;
  int64_t var_10 = *(fsbase + 0x28);
  int32_t var_cc = 3;
  int32_t var_c8 = 0;
  setup();
  printf("{?} Enter name: ");
  read_buf(&user, 0x40);
  printf("{?} Enter secret: ");
  void var_28;
  read_buf(&var_28, 0x10);
  while (true)
  {
    if (var_cc <= 0)
    {
      exit(0xfffffffe);
      /* no return */
    }
    menu();
    void var_a8;
    switch (read_int())
    {
      case 1:
      {
        if (var_c8 <= 0xf)
        {
          void* rax_9 = malloc(0x30);
          printf("{?} Enter password: ");
          read_buf(rax_9, 0x30);
          *(&var_a8 + (var_c8 << 3)) = rax_9;
          var_c8 = (var_c8 + 1);
          continue;
        }
        else
        {
          puts("{-} No available space!");
          continue;
        }
        break;
      }
      case 2:
      {
        printf("{?} Enter password id: ");
        int32_t rax_15 = read_int();
```

Jsdec

```
int32_t main (void) {
    rax = *(fs:0x28);
    *(var_10h) = rax;
    eax = 0;
    *(var_cch) = 3;
    *(var_c8h) = 0;
    setup ();
    eax = 0;
    printf ("{?} Enter name: ");
    esi = 0x40;
    rdi = user;
    read_buf ();
    eax = 0;
    printf ("{?} Enter secret: ");
    rax = var_28h;
    esi = 0x10;
    rdi = rax;
    read_buf ();
label_0:
    if (*(var_cch) <= 0) {
        exit (0xfffffffe);
    }
    menu ();
    eax = read_int ();
    *(var_bch) = eax;
    if (*(var_bch) == 0xdeadbeef) {
        exit (0xffffffff);
    }
    if (*(var_bch) > 8) {
        goto label_1;
    }
    eax = *(var_bch);
    rdx = rax*4;
    rax = data_004020dc;
    eax = *((rdx + rax));
    rax = (int64_t) eax;
    rdx = data_004020dc;
    rax += rdx;
    /* switch table (9 cases) at 0x4020dc */
    void (*rax)() ();
    if (*(var_c8h) > 0xf) {
        puts ("{-} No available space!");
        goto label_2;
    }
```

RetDec

```
// Address range: 0x4011b2 - 0x4015a2
int main(int argc, char ** argv) {
    // 0x4011b2
    __readfsqword(40);
    setup();
    printf("{?} Enter name: ");
    read_buf((int64_t *)&g5, 64);
    printf("{?} Enter secret: ");
    int64_t v1; // bp-40, 0x4011b2
    read_buf(&v1, 16);
    int32_t v2 = 3; // 0x401593
    menu();
    int64_t v3 = read_int(); // 0x401241
    uint32_t v4 = (int32_t)v3; // 0x401246
    while (v4 != -0x21524111) {
        if (v4 < 9) {
            int32_t v5 = *(int32_t *)((4 * v3 & 0x3fffffffc) + (int64_t)&g1); //
0x401284
            return (int64_t)v5 + (int64_t)&g1;
        }
        // 0x401229
        v2--;
        if (v2 == 0) {
            // 0x401232
            exit(-2);
            // UNREACHABLE
        }
        menu();
        v3 = read_int();
        v4 = (int32_t)v3;
    }
    // 0x401258
    exit(-1);
    // UNREACHABLE
}
```

Время задач

Утомительный реверс

Категория: Lesson 16 / ELF + Decompilers + Dynamic analysis

Решивших: 0

Время: 00:00:02

- Доступ к задачам можно получить как всегда на nsuctf.ru
- В этой задаче вам может пригодиться бесплатная IDA / Hex-Rays, скачать ее можно по ссылке https://www.hex-rays.com/ida-free/

Форматы исполняемых файлов

А что мы вообще открываем в дизассемблере?

- Исполняемые файлы, так же как и архивы или картинки, имеют свой формат
- Именно форматом определяется, где в файле лежат данные, где код, а где какая-то служебная информация
- Наиболее популярными форматами на сегодняшний день являются:
  - ELF – исполняемые файлы Linux / Android
  - PE – исполняемые файлы Windows
  - Mach-O – исполняемые файлы macOS / iOS
- Мы разберем только первый формат из этого списка
  - Однако, общие принципы будут применимы и к другим форматам

ELF

- ELF – Executable and Linkable Format
- Формат используется для хранения как исполняемого кода, так и разделяемых библиотек
  - Впрочем ими список не ограничивается, например объектные файлы (.o) также используют этот формат
- Содержит следующие основные компоненты:
  - Заголовок
  - Сегменты
  - Секции

Заголовок ELF

- Парсить его руками вам, скорее всего, не придется
- Содержит следующую информацию:
  - Сигнатуру файла, которая вообще дает понять, что это ELF
  - Информацию о процессоре, для которого собран файл
  - Базовую информацию о файле (исполняемый это файл или библиотека)
  - Где лежит информация о секциях и сегментах
  - Где вообще начинается исполнение файла (точка входа)
- Посмотреть заголовок можно следующей командой:

```
readelf -h <имя_файла>
```

Заголовок ELF

- ei_magic – байты, дающие понять, что это ELF-файл
- ei_class – разрядность файла (1 - 32 бита, 2 - 64 бита)
- ei_data – порядок байтов (1 - обратный, 2 - прямой)
- e_type – тип файла: исполняемый (ET_EXEC), разделяемая библиотека (ET_DYN) или что-то еще
- e_machine – процессор (EM_386, EM_X86_64 и т.д.)
- e_entry – точка входа (начало программы)
- e_phoff, e_shoff – расположение таблиц с описаниями сегментов и секций в файле (или 0, если таблицы нет)
  - Далее также идет информация о размере записей и т.д.

e_ident (16 байт)

ei_magic	ei_class	ei_data	ei_version	...
\x7fELF	0x2	0x1	0x1	...

e_type (2 байта)	e_machine (2 байта)	e_version (4 байта)	e_entry (указатель)
0x2	0x3e	0x1	0x4010d0

e_phoff (указатель)	e_shoff (указатель)	e_flags (4 байта)	...
0x40	0x3bf0	0x0	...

Пример вывода readelf -h

```
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:               0x4010d0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          15344 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         11
  Size of section headers:           64 (bytes)
  Number of section headers:         29
  Section header string table index: 28
```

Сегменты в ELF

- В первом приближении, сегменты содержат информацию о следующем:
  - Каковы разрешения у данного участка в памяти: можно ли его исполнять (т.е. это код), или нет (т.е. это данные)
  - Какое изначально содержимое у данного участка – некоторая часть ELF-файла (для кода или данных) или никакого (для пустой памяти)
- Сегменты обрабатываются непосредственно ядром операционной системы
  - Таким образом, любой исполняемый файл обязан содержать некоторое количество сегментов
  - Также могут использоваться динамическим компоновщиком (сегмент PT_DYNAMIC)
- Посмотреть сегменты можно следующей командой:

readelf --segments <имя_файла>

Сегменты в ELF

- p_type – тип сегмента, говорит о назначении сегмента (только при PT_LOAD загружается в память)
- p_offset – расположение содержимого сегмента в файле
- p_vaddr – виртуальный адрес, куда должен быть загружен сегмент, p_paddr – физический (обычно равен p_vaddr и не используется ядром)
- p_filesz – размер сегмента в файле
- p_memsz – размер сегмента в памяти (если больше p_filesz, оставшаяся память будет заполнена нулями)
- p_flags – содержит информацию о разрешениях сегмента (PF_X, PF_W, PF_R – для исполнения, записи и чтения)
- p_align – выравнивание сегментов в памяти и файле, $\text{p\_offset} = \text{p\_vaddr}\ (mod\ \text{p\_align})$

p_type (4 байта)	p_flags* (4 байта)	p_offset (указатель)	p_vaddr (указатель)
0x1	0x5	0x1000	0x401000

p_paddr (указатель)	p_filesz (указатель)	p_memsz (указатель)	p_align (указатель)
0x401000	0x80d	0x80d	0x1000

* в 32-битной версии ELF p_flags идет предпоследним, а не вторым

Пример вывода readelf --segments

```
...
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x0000000000000268 0x0000000000000268  R      0x8
  INTERP         0x00000000000002a8 0x00000000004002a8 0x00000000004002a8
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000710 0x0000000000000710  R      0x1000
  LOAD           0x0000000000001000 0x0000000000401000 0x0000000000401000
                 0x000000000000080d 0x000000000000080d  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000402000 0x0000000000402000
                 0x00000000000003d0 0x00000000000003d0  R      0x1000
  LOAD           0x0000000000002e10 0x0000000000403e10 0x0000000000403e10
                 0x0000000000000268 0x00000000000002f8  RW     0x1000
  DYNAMIC        0x0000000000002e20 0x0000000000403e20 0x0000000000403e20
                 0x00000000000001d0 0x00000000000001d0  RW     0x8
...
```

Секции в ELF

- Содержат более подробную информацию о содержимом сегментов
  - В первую очередь используются компилятором для связывания с динамической библиотекой, представленной этим ELF-файлом
  - Также могут проверяться динамическим компоновщиком (например на Android)
  - Могут хранить отладочную информацию
- Секции гораздо чаще используются в процессе обратной разработки
  - Хотя и не обязательно должны присутствовать в исполняемом файле, существует утилита sstrip из состава ELFkickers, которая их удаляет
- Посмотреть секции можно следующей командой:

readelf --sections <имя_файла>

Секции в ELF

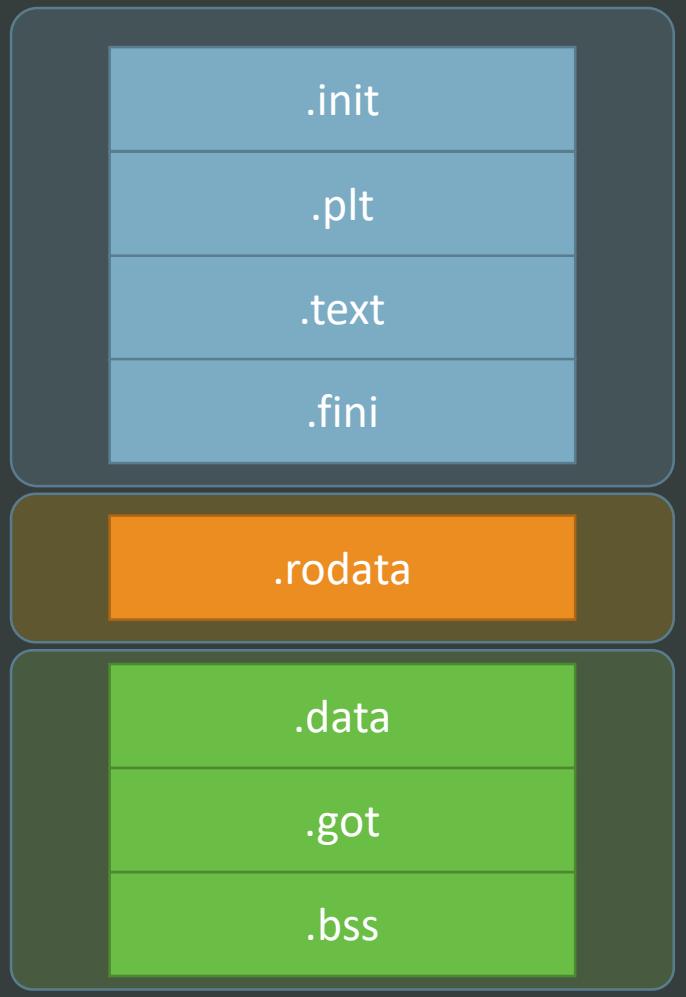
- У секций есть следующие важные параметры:
  - sh_name – имя секции (задается как расположение строки с именем в таблице строк), обычно начинается с точки
  - sh_type – тип секции (например, SHT_PROGBITS – данные, формат которых определен самой программой, или SHT_DYNAMIC – данные для динамического компоновщика)
  - sh_flags – флаги секции (например SHF_WRITE, означающий, что секция доступна для записи или SHF_EXECINSTR, означающий, что в секции лежит исполняемый код)
  - sh_offset – расположение секции в файле (от начала)
  - sh_size – размер секции
- Важно помнить, что секциям нельзя доверять, так как они зачастую не нужны в процессе исполнения программы
- В IDA имя секции по умолчанию выводится слева от кода / данных

Типичные секции в ELF

Название секции	Назначение	Тип. права доступа
.text	Секция с исполняемым кодом, содержит непосредственно машинные инструкции	RX
.plt	Код, обеспечивающий динамическую загрузку функций из библиотек (см. далее)	RX
.init / .fini	Код, выполняющийся в начале и конце программы (см. далее)	RX
.init_array / .fini_array	Адреса функций, выполняющихся в начале и конце программы (см. далее)	RW
.data	Секция с данными, доступными для чтения и записи; содержит, например, различные предварительно инициализированные массивы	RW
.rodata	Как .data, но только для констант; содержит в основном строковые константы	R
.bss	Как .data, только без данных, не занимает места в исполняемом файле; содержит массивы, инициализированные нулями	RW
.got / .got.plt	Таблицы адресов функций, которые импортируются из динамических библиотек; эти поля обычно заполняются динамическим компоновщиком	RW

Секции в ELF

- Можно заметить, что такое разделение на сегменты обеспечивает некоторое подобие Гарвардской архитектуры
- Эта особенность широко используется в обеспечении безопасности ПО, мы вернемся к ней во второй половине семестра



Сегмент 1

Права доступа: чтение + исполнение

Сегмент 2

Права доступа: чтение

Сегмент 3

Права доступа: чтение + запись

Время задач

Секретная секция

Категория: Lesson 16 / ELF + Decompilers + Dynamic analysis

Решивших: 0

Время: 00:00:02

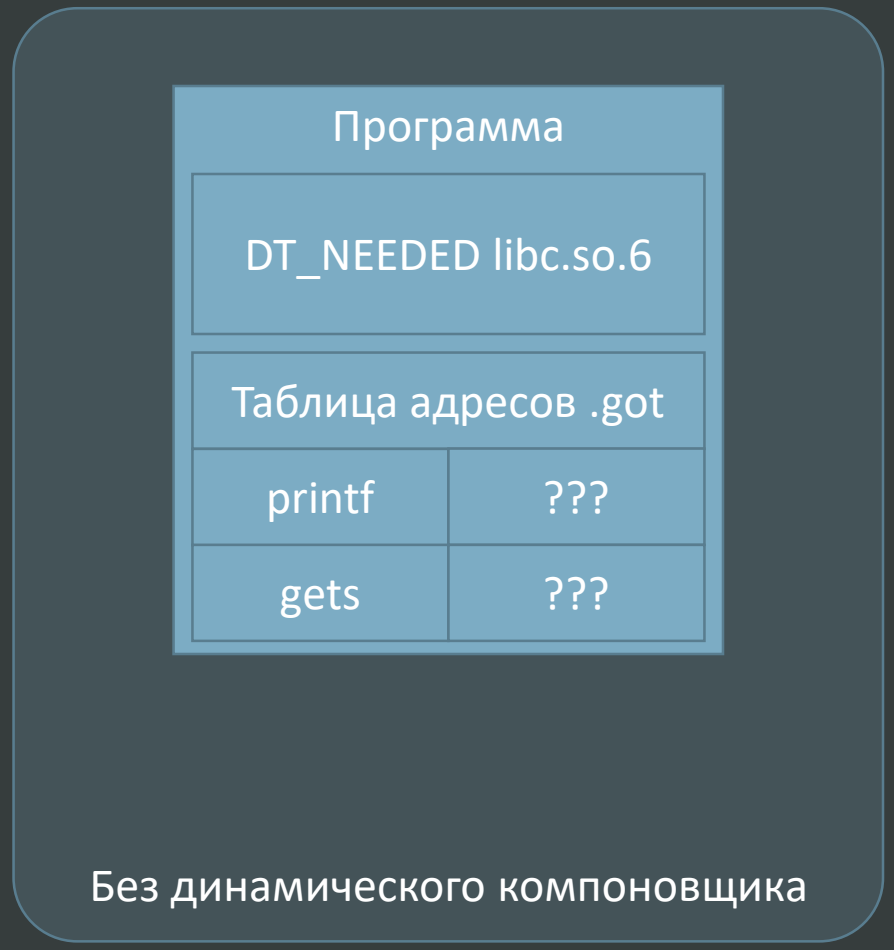
- Доступ к задачам можно получить как всегда на nsuctf.ru
- В этой задаче вам может пригодиться readelf, WinHEX и IDA

Загрузка библиотек

Динамический компоновщик

- Более известен как dynamic linker
- Библиотека или исполняемый файл, помогающий операционной системе загрузить разделяемые библиотеки в исполняемый файл
  - В Linux обычно имеет название "ld.so" или "ld-linux.so"
- Выполняет следующие основные функции:
  - Осуществляет поиск библиотек, от которых зависит запускаемая программа
  - Ищет в этих библиотеках запрашиваемые программой функции
  - Помещает адреса этих функций в память программы, чтобы она могла их вызывать
- В ELF путь к компоновщику указывается в сегменте PT_INTERP
  - Обычно имеет вид вроде /lib64/ld-linux-x86-64.so.2

Динамический компоновщик



.got.plt / .plt

- В самой простой ситуации все указатели на функции просто лежали бы в секции .got с самого начала работы программы
- Но это было бы слишком просто, поэтому в Linux существует такое понятие как Lazy binding библиотек
  - Адреса функций в этом режиме разрешаются в процессе выполнения только когда они становятся нужны
- Для того, чтобы осуществить такой механизм, в программе добавляется специальный код, который обычно лежит в секции .plt

Конструкторы и деструкторы

Конструкторы и деструкторы

- Функции, которые выполняются до и после завершения программы
- Выполняются даже в разделяемых библиотеках при их загрузке (будь то загрузка через dlopen или как зависимости)
- Функция конструктора выполнятся ДО main
- Функция деструктора вызывается даже при вызове exit() (но не в случае получения процессом сигнала SIGKILL, например)
- Эти функции могут сбить начинающего реверсера с толку, потому что большинство людей при чтении кода обращают внимание только на main()

Конструкторы и деструкторы

- Конструкторы и деструкторы могут прятаться в следующих местах:
  - В секциях .init и .fini в качестве кода
  - В секциях .init_array и .fini_array в качестве указателей на функции
- Никакой магии в этих секциях самих по себе нет, код в них отдельными функциями из библиотеки языка Си, например __libc_csu_init для конструкторов
- В свою очередь, эти функции запускаются из точки входа (да, точка входа и функция main это не одно и то же)
- Важно иметь в виду, что для разделяемых библиотек это не так, и их конструкторы и деструкторы вызывает динамический линкер из соответствующих записей DT_INIT, DT_INIT_ARRAY и т.д. сегмента PT_DYNAMIC

Конструкторы и деструкторы

- Ниже приведен пример декомпилированного при помощи IDA кода точки входа
- Обратите внимание, что main – лишь один из указателей, передаваемых в функцию __libc_start_main, наряду с __libc_csu_init

```
void __fastcall __noreturn start(__int64 a1, __int64 a2, void (*a3)(void))
{
  __int64 v3; // rax
  int v4; // esi
  __int64 v5; // [rsp-8h] [rbp-8h] BYREF
  char *retaddr; // [rsp+0h] [rbp+0h] BYREF
  v4 = v5;
  v5 = v3;
  _libc_start_main(
    (int (__fastcall *)(int, char **, char **))main, v4, &retaddr,
_libc_csu_init, _libc_csu_fini, a3, &v5);
  __halt();
}
```

Конструкторы и деструкторы

Код на Си:

```
__attribute__((constructor)) void thiswillrunbeforemain()
{
    printf("Hello world");
}
```

Указатель на функцию в массиве в секции .init_array:

```
.init_array:0000000000200DB0 ; ELF Initialization Function Table
.init_array:0000000000200DB0 ; ===========================================================
.init_array:0000000000200DB0
.init_array:0000000000200DB0 ; Segment type: Pure data
.init_array:0000000000200DB0 ; Segment permissions: Read/Write
.init_array:0000000000200DB0 _init_array     segment qword public 'DATA' use64
.init_array:0000000000200DB0                 assume cs:_init_array
.init_array:0000000000200DB0                 ;org 200DB0h
.init_array:0000000000200DB0 __frame_dummy_init_array_entry dq offset frame_dummy
.init_array:0000000000200DB0                                         ; DATA XREF: LOAD:0000
.init_array:0000000000200DB0                                         ; LOAD:000000000000021
.init_array:0000000000200DB0                                         ; Alternative name is
.init_array:0000000000200DB8                 dq offset thiswillrunbeforemain
.init_array:0000000000200DB8 _init_array     ends
```

Конструкторы и деструкторы

Код на Си:

```
asm(".section \".init\"");
asm("call thiswillrunbeforemain");
asm(".text");

void thiswillrunbeforemain()
{
    printf("Hello world");
}
```

Код в секции .init:

```
.init:00000000000004F0 ; Segment type: Pure code
.init:00000000000004F0 ; Segment permissions: Read/Execute
.init:00000000000004F0 _init           segment dword public 'CODE' use64
.init:00000000000004F0                 assume cs:_init
.init:00000000000004F0                 ;org 4F0h
.init:00000000000004F0                 assume es:nothing, ss:nothing, ds:_
.init:00000000000004F0
.init:00000000000004F0 ; =============== S U B R O U T I N E ==============
.init:00000000000004F0
.init:00000000000004F0
.init:00000000000004F0                 public _init_proc
.init:00000000000004F0 _init_proc      proc near               ; CODE XREF:
.init:00000000000004F0                 sub     rsp, 8          ; _init
.init:00000000000004F4                 mov     rax, cs:__gmon_start___ptr
.init:00000000000004FB                 test    rax, rax
.init:00000000000004FE                 jz      short loc_502
.init:0000000000000500                 call    rax ; __gmon_start__
.init:0000000000000502
.init:0000000000000502 loc_502:                                ; CODE XREF:
.init:0000000000000502                 call    thiswillrunbeforemain
.init:0000000000000507                 add     rsp, 8
.init:000000000000050B                 retn
```

Время задач

Обманчивый код

Категория: Lesson 16 / ELF + Decompilers + Dynamic analysis

Решивших: 0

Время: 00:00:01

- Доступ к задачам можно получить как всегда на nsuctf.ru

Виды исполняемых файлов

Виды исполняемых файлов

- Исполняемые файлы можно разделить на следующие группы по характеру их отношений с динамическими библиотеками:
  - Статические исполняемые файлы (static executable) – не требуют динамических библиотек, все делают сами при помощи системных вызовов, загружаются напрямую ядром
  - Обычные (динамические) исполняемые файлы (dynamic executables) – требуют динамические библиотеки, соответственно загружаются при помощи ядра и динамического линкера
  - Позиционно-независимые исполняемые файлы (position-independent executables) – сами являются динамической библиотекой 0_o (у них тип ET_DYN)
- К разным типам исполняемых файлов применяются разные подходы обратной разработки
  - Последние два типа, правда, в рамках ближайших тем будут очень похожи

Виды исполняемых файлов

	Статический исполняемый файл	Динамический исполняемый файл	Позиционно-независимый исполняемый файл
Размер	Очень большой	Обычный	Обычный
Типичные языки программирования	Go, FreePascal, многие другие (но не по умолчанию)	Любые (кроме Go)	Любые современные
Основная особенность	Возможность запускать ПО без зависимостей		Возможность загружать код исполняемого файла по любому адресу
Устойчивость к реверсу	Высокая (в случае Go – очень высокая)	Обычная	Чуть выше среднего

Введение в динамический анализ

Динамический анализ

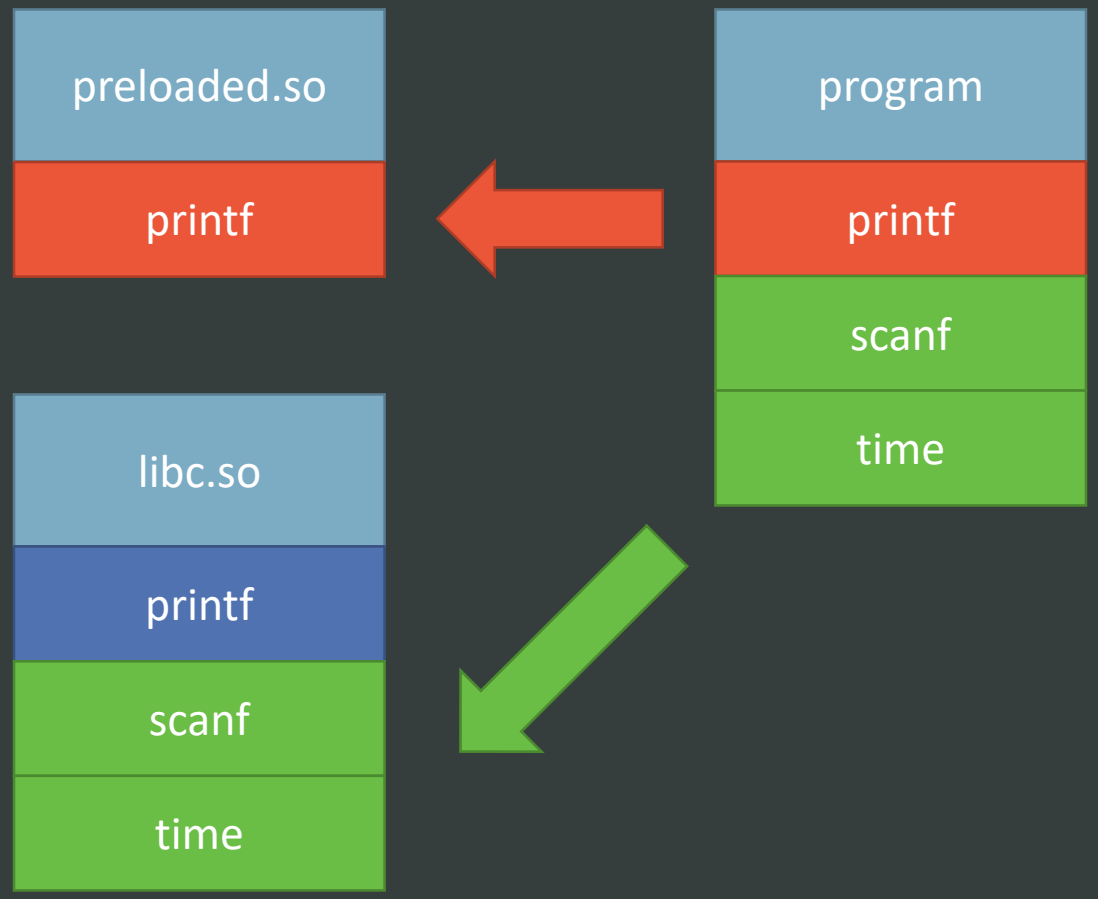
- Способ анализа приложений, основанный на отслеживании процесса их выполнения
- Противоположностью ему является статический анализ, который исполнения кода не требует
  - Большинство ранее рассмотренных методов реверс-инжиниринга относятся сюда
- Классическим примером динамического анализа в процессе разработки программного обеспечения является отладка
- Сегодня мы рассмотрим следующие способы анализа программ:
  - LD_PRELOAD / LD_AUDIT
  - ltrace / latrace
  - strace

LD_PRELOAD

- Специальная переменная окружения, позволяющая загрузить в целевую программу какую-то свою разделяемую библиотеку
  - Достаточно указать в этой переменной путь к вашей .so библиотеке
- Это позволяет сделать следующее:
  - При помощи конструктора в этой библиотеке можно исполнить произвольный код в памяти процесса
  - Заменить функции, которые импортирует программа из других библиотек
- Функциональность LD_PRELOAD реализуется в динамическом компоновщике ld.so
  - Поэтому LD_PRELOAD не будет работать в статических исполняемых файлах
- Кроме LD_PRELOAD существует более низкоуровневый механизм LD_AUDIT, почитать про него больше можно, например, тут: https://www.sentinelone.com/labs/leveraging-ld_audit-to-beat-the-traditional-linux-library-preloading-technique/

LD_PRELOAD и замена функций

- Имена импортируемых функций в Linux не привязаны к какой-то конкретной библиотеке
- Если в нескольких библиотеках есть функция с одинаковым именем, то для использования в программе будет выбрана та, которая была загружена первой
  - LD_PRELOAD, как следует из его названия, загружает библиотеки раньше всех



Пример LD_PRELOAD

main.c:

```c
int main()
{
    printf("Hello World");
}
```

preload.c:

```c
__attribute__((constructor)) int preload() {
    puts("Hello LD_PRELOAD");
}

int printf() {
    puts("Nope");
}
```

```
$ gcc main.c -o main && gcc preload.c -fPIC --shared -o preload.so
$ LD_PRELOAD=./preload.so ./main

Hello LD_PRELOAD
Nope
```

LD_PRELOAD и оригинальные функции

- Как можно было заметить, доступ к оригинальным функциям библиотеки языка Си (или любой другой) теряется
  - Так как в вашей библиотеке уже есть функция, например, printf, попытка вызвать из нее printf приведет лишь к вызову ей самой себя
- Для решения этой проблемы можно использовать dlsym(RTLD_NEXT, "имяфункции")
  - Эта конструкция позволяет получить указатель на «следующий» вариант функции, который обычно будет искомым

preload.c:

```
#define _GNU_SOURCE
#include <dlfcn.h>
static int (*printf_orig)(char* p1) = 0;
int printf()
{
  if (!printf_orig)
    printf_orig=dlsym(RTLD_NEXT, "printf");
  printf_orig("Nope\n");
}
```

LD_PRELOAD

- Зачем можно использовать LD_PRELOAD?
  - В мирных целях – например прокачать свой fopen, чтобы он научился открывать файлы из архивов
  - В целях выключения надоедливых функций во время отладки (каких-нибудь ненужных sleep, например)
  - В целях принципиального изменения работы программ путем замены некоторых функций без модификации кода самой программы (например, замена strcmp своей реализацией)
  - Больше примеров: https://habr.com/ru/post/199090/
- Кстати, в LD_PRELOAD можно загружать и несколько библиотек, указав их через пробел или двоеточие

ltrace

- Программа, позволяющая просмотреть все вызовы к функциям библиотек в некоторой программе
- По этой причине также бесполезна для статических исполняемых файлов
- Работает путем перехвата функций в секции .plt
  - Поэтому не будет работать для исполняемых файлов, собранных, например, при помощи GCC с флагом -fno-plt
  - Еще в Ubuntu 22.04 не работала с Intel CET, но в 24 году починили, ура
- Для работы использует системный вызов ptrace(), поэтому такая трассировка является полноценной отладкой в плане возможности обнаружения
- Используется следующим образом:

ltrace <имя_файла>

Пример ltrace

```
$ ltrace ./main
printf("Hello, enter your name: ")
= 24
gets(0x7ffff237df00, 0x7fffeab84260, 0, 0Hello, enter your name: 123
)                                    = 0x7ffff237df00
printf("Hello, %s\n", "123"Hello, 123
)                                           = 11
+++ exited (status 0) +++
```

ltrace

- Зачем можно использовать ltrace?
  - Чтобы быстро посмотреть, что вообще программа делает (без запуска дизассемблера)
  - Чтобы решать совсем простые задачи на реверс (в духе strcmp)
  - Чтобы решать совсем сложные задачи на реверс, заканчивающие проверку флага тем же strcmp
- На последнем пункте остановимся поподробнее: представим, что вам дана программа на языке Perl, которая множество раз себя модифицирует, распаковывает, да и Perl вы не знаете
- Однако в самых глубинах этой программы введенный флаг сравнивается с настоящим значением
  - А интерпретатор Perl, как и подобает нормальной программе, не реализует сравнение строк сам, а использует функцию strcmp()
  - Тогда ltrace легко и просто решит эту задачу, просто поймав момент сравнения

latrace

- Если вдруг ltrace по каким-то причинам не подходит (или сломался), можно также воспользоваться инструментом latrace
- Использует LD_AUDIT, а не ptrace, поэтому отладкой не считается
  - Однако, подключаться к уже существующим процессам не получится
  - С -fno-plt тоже отваливается, но это проблема LD_AUDIT, а не инструмента (мб исправят)
- Чтобы включить отображение аргументов нужен флаг -A, фильтровать функции можно флагом -s
- По умолчанию генерирует достаточно подробный вывод, включающий вызовы между библиотеками
  - Во вложенных вызовах latrace ставит разные отступы, поэтому можно попробовать отфильтровать это конструкциями вида grep -v '[нужное число пробелов]', встроенной возможности это отключить я не нашел

Пример latrace

```
$ latrace -s gets,printf -A ./main
  442     printf(format = "Hello, enter your name: ")
[/lib/x86_64-linux-gnu/libc.so.6] {
Hello, enter your name:   442     } printf = 24
  442     gets(s = "ШV") [/lib/x86_64-linux-gnu/libc.so.6] {
qweqwe
Hello, qweqwe
  442     } gets = "qweqwe"
  442     printf(format = "Hello, %s
") [/lib/x86_64-linux-gnu/libc.so.6] {
  442     } printf = 14
```

strace

- Программа, позволяющая просмотреть все системные вызовы в некоторой программе
  - В Solaris вы могли встретить ее коллегу под названием truss
- По этой причине крайне полезна для статических исполняемых файлов
- В отличие от ltrace работает практически всегда и очень стабильно
- Для работы использует системный вызов ptrace()
- Используется следующим образом:

strace <имя_файла>

Пример strace

```
$ strace ./main
...
ioctl(1, TCGETS, {B38400 opost isig icanon echo ...}) = 0
brk(NULL)                                       = 0x7fffedbb2000
brk(0x7fffedbd3000)                             = 0x7fffedbd3000
fstat(0, {st_mode=S_IFCHR|0660, st_rdev=makedev(4, 1), ...}) = 0
ioctl(0, TCGETS, {B38400 opost isig icanon echo ...}) = 0
write(1, "Hello, enter your name: ", 24Hello, enter your name: ) = 24
read(0, 123
"123\n", 512)                       = 4
write(1, "Hello, 123\n", 11Hello, 123
)             = 11
exit_group(0)
```

strace

- Зачем можно использовать strace?
  - Для того же, для чего и ltrace, только с гарантированной работоспособностью
  - Чтобы получить информацию о взаимодействии с операционной системой на самом низком уровне, такая трассировка позволяет найти ошибки даже в библиотеке языка Си
- Особенно strace полезен, чтобы установить, с какими файлами взаимодействует программа
  - Если сравнить строки вполне можно без использования системных вызовов (впрочем, иначе и не получится), то открыть файл так не выйдет

Полезные флаги ltrace и strace

- -e <фильтр> – флаг, позволяющий фильтровать по именам функций (ltrace) и системных вызовов (strace)
  - Позволяет использовать восклицательный знак в качестве отрицания (только в bash его нужно брать в апострофы), что позволяет скрыть наиболее надоедливые функции вроде clock_gettime
- -p <pid> – флаг, позволяющий подключиться к процессу прямо во время его выполнения
- -f – флаг позволяющий отслеживать дочерние процессы запущенной программы
  - Будьте внимательны, по умолчанию этого не делается

Ловим временные файлы

Ловим временные файлы

- Некоторые программы создают временные файлы (например, библиотеки или исполняемые файлы), которые впоследствии сразу же могут быть удалены
  - Файлы, в свою очередь, могут представлять значительный интерес, но посмотреть их через strace / ltrace не так просто
- Существует несколько способов поймать такие файлы
- Первый способ – запретить системный вызов удаления
  - Это можно сделать при помощи флага "-e inject=unlink:retval=0" или "-e inject=unlinkat:retval=0" утилиты strace
- Второй способ – попытаться восстановить файл с диска или примонтировать в предполагаемое место расположения файла какую-то свою файловую систему
  - Этот способ мы подробно рассматривать не будем

Ловим временные файлы

- Еще одним, немного странным способом, является создание по месту записи файла именованного канала (named pipe) командой mkfifo
  - Для этого необходимо знать название временного файла
- В этом случае, будет создана сущность, очень близкая по сути обычным пайпам, используемым в командах вроде "ls | wc"
  - В частности, операция записи в файл исследуемой программой будет прервана на этом месте до тех пор, пока какая-то наша программа не считает оттуда файл
- Пример использования этой техники:

Терминал 1:

```
$ mkfifo /tmp/capfile
$ cat /tmp/capfile > ~/capfile
```

Терминал 2:

```
$ ./evilprogram
```

Ловим временные файлы

- Более простым способом при тех же ограничениях (есть имя файла) является создание символьной ссылки
- Тогда при попытке записать по этому пути, файл будет записан по ссылке, а вот при удалении – будет удалена сама символьная ссылка
  - Если программист специально не позаботится об обратном

Пример использования техники:

```
$ ln -s ourcapture secret
$ cat ourcapture
cat: ourcapture: No such file or
directory
$ ./create_secret
$ cat ourcapture
Very secret data
```

Код программы create_secret:

```c
#include <stdio.h>
int main() {
    FILE* f = fopen("secret", "w");
    fprintf(f, "Very secret data\n");
    fclose(f);
    remove("secret");
}
```

Спасибо за внимание!
Задачи доступны на

nsuctf.ru

- Пожалуйста, используйте имя пользователя формата "Фамилия Имя"
  - e-mail можно забить любой, сервером он не проверяется
- Для вопросов по задачам рекомендую присоединиться к @NSUCTF в Telegram
  - Только, пожалуйста, без спойлеров