Systemy komputerowe

Lista zadań nr 6

NOTATKI

Zadanie 1

Zadanie 1. Poniżej podano zawartość pliku `main.c`:

```
1  #include "stdio.h"
2
3  static int global = 15210;
4
5  static void set_global(int val) {
6          global = val;
7  }
10 int main(void) {
11         printf("before: %d\n", global);
12         set_global(15213);
13         printf("after: %d\n", global);
14         return 0;
15 }
```

Polecenie `gcc main.c -o main` jest równoważne ciągowi poleceń
`cpp -o main_p.c main.c; gcc -S main_p.c; as -o main.o main_p.s; gcc -o main main.o.`

Jaka jest rola poszczególnych poleceń w tym ciągu?

- **cpp -o main_p.c main.c** - jest to instrukcja preprocesora generująca plik main_p.c
- **gcc -S main_p.c** - sprawia, że kod z main_p.c jest tłumaczony na assembly i zapisany w main_p.s
- **as -o main.o main_p.s -** tu kod tłumaczony jest na kod binarny i zapisany w main.o
- **gcc -o main main.o** - jest to instrukcja powoduje,że tworzony jest plik wykonywalny main

Skąd pochodzi kod, który znalazł się w pliku main_p.c?

Kod, który znalazł się w pliku main_p.c został wygenerowany poleceniem cpp przez preprocesor.

Co zawiera plik main_p.s ? Zauważ etykiety odpowiadające zmiennej *global* i obydwu funkcjom. W jaki sposób przyporządkować etykiecie jej typ?

Plik **main_p.s** zawiera:

```
.type    global,@object
.size    global, 4
```

Global:

```
.long   15210
.text
.type    set_global, @function
```

Plik zawiera kod przetłumaczony na assembly, a typ zmiennej znajduję się tuż nad nią.

Poleceniem objdump –t wyświetl tablicę symboli pliku main.o . Jakie położenie wg. tej tablicy mają symbole global i set_global?

0000000000000000 l O .data 0000000000000004 global

0000000000000000 l F .text 0000000000000013 set_global

global ma adres 0 w sekcji .data

set_global ma adres 0 w sekcji .text

Poleceniem objdump –h wyświetl informacje o sekcjach w pliku main.o. Dlaczego adres sekcji .text i .data to 0? Jakie są adresy tych sekcji w pliku wykonywalnym main?

LMA – Load Memory Address

VMA – Virtual Memory Address

Polecenie: objdump –h main.o

```
main.o:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         0000005a  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000004  0000000000000000  0000000000000000  0000009c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  0000000000000000  0000000000000000  000000a0  2**0
                  ALLOC
  3 .rodata       00000017  0000000000000000  0000000000000000  000000a0  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      0000001d  0000000000000000  0000000000000000  000000b7  2**0
                  CONTENTS, READONLY
  5 .note.GNU-stack 00000000  0000000000000000  0000000000000000  000000d4  2**0
                  CONTENTS, READONLY
  6 .eh_frame     00000058  0000000000000000  0000000000000000  000000d8  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

W main.exe:

```
 12 .plt.got     00000008  0000000000001040  0000000000001040  00001040  2**3
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .text        000001a1  0000000000001050  0000000000001050  00001050  2**4
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
 14 .fini        00000009  00000000000011f4  00000000000011f4  000011f4  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
 15 .rodata      0000001b  0000000000002000  0000000000002000  00002000  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 16 .eh_frame_hdr 00000044  000000000000201c  000000000000201c  0000201c  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 17 .eh_frame    00000128  0000000000002060  0000000000002060  00002060  2**3
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 18 .init_array  00000008  0000000000003da8  0000000000003da8  00002da8  2**3
                 CONTENTS, ALLOC, LOAD, DATA
 19 .fini_array  00000008  0000000000003db0  0000000000003db0  00002db0  2**3
                 CONTENTS, ALLOC, LOAD, DATA
 20 .dynamic     000001e0  0000000000003db8  0000000000003db8  00002db8  2**3
                 CONTENTS, ALLOC, LOAD, DATA
 21 .got         00000028  0000000000003fd8  0000000000003fd8  00002fd8  2**3
                 CONTENTS, ALLOC, LOAD, DATA
 22 .got.plt     00000020  0000000000004000  0000000000004000  00003000  2**3
                 CONTENTS, ALLOC, LOAD, DATA
 23 .data        00000014  0000000000004020  0000000000004020  00003020  2**3
                 CONTENTS, ALLOC, LOAD, DATA
```

Relokacja, kompilatory i asemblery generują kod i sekcje zaczynające się od adresu 0.

Linker przenosi te sekcje poprzez przydzielanie miejsca pamięci dla każdej definicji symbolu, a następnie modyfikowanie wszystkich odniesień do symboli, tak aby wskazywały one na to miejsce w pamięci.

Zadanie 2

Zadanie 2. Poniżej podano zawartość pliku `swap.c`:

```
1  extern int buf[];
2
3  int *bufp0 = &buf[0];
4  static int *bufp1;
5  int intvalue = 0x77;
6
7  static void incr() {
8    static int count = 0;
9    count++;
10 }
```

```
10 void swap() {
11   int temp;
12   incr();
13   bufp1 = &buf[1];
14   temp = *bufp0;
15   *bufp0 = *bufp1;
16   *bufp1 = temp;
17 }
```

Wygeneruj **plik relokowalny** «`swap.o`», po czym na podstawie wydruku polecenia «`readelf -t -s`» dla każdego elementu tablicy symboli podaj:

- adres symbolu względem początku sekcji,
- typ symbolu – tj. «`local`», «`global`», «`extern`»,
- rozmiar danych, na które wskazuje symbol,
- numer i nazwę sekcji – tj. «`.text`», «`.data`», «`.bss`» – do której odnosi się symbol.

Co przechowują sekcje «`.strtab`» i «`.shstrtab`»?

Nazwa	Adres	Typ	Rozmiar	Sekcja
incr	0	local	22	1 .text
count.1798	0x8	local	4	4 .bss
swap	0x16	global	77	1 .text
bufp0	0	global	8	5 .data.rel
bufp1	0	local	8	4 .bss
intvalue	0	global	4	3 .data

<<.strtab>> tablica stringów, przechowuje indexy do stringów używanych do reprezentacji nazw symboli.

<<shstrtab>> tablica stringów, przechowuje nazwy sekcji, każda sekcja ma swój nagłówek zawierający index tablicy shstrtabx.

Zadanie 3

Zadanie 3. Rozważmy program skompilowany z opcją -Og składający się z dwóch plików źródłowych:

```
1 /* oof.c */
2 void p2(void);
3
4 int main() {
5   p2();
6   return 0;
7 }
```

```
1 /* rab.c */
2 #include <stdio.h>
3
4 char main;
5
6 void p2() {
7   printf("0x%x\n", main);
8 }
```

Po wypisaniu 0x48 program kończy działanie bez zgłaszania błędu.
Dzieje się tak, ponieważ w plikach jest tylko jeden silny symbol main (definicja funkcji w oof.c). char ma 1 bajt, więc wypisanie main w rab.c spowoduje wypisanie pierwszego bajtu kodu funkcji main z /*oof.c*/.
Po użyciu objdump -d możemy się przekonać, że Pierwszy bajt kodu funkcji main to faktycznie 0x48, co możemy zobaczyć po użyciu objdump –d.

Przypisanie wartości main w funkcji p2 prowadzi do błędu segmentacji.

Przypisanie wartości main w momencie deklaracji w rab.c doprowadzi do błędu linkera. Linker znajduje dwa silne symbole main, przez co mamy błąd linkera.

Zadanie 5

Zadanie 5. Wpis w tablicy symboli dla zmiennej `buf` w pliku `swap.o` wskazuje, że zmienna ta znajduje się w sekcji `UNDEF`. Zmodyfikuj deklarację zmiennej `buf` w pliku `swap.c` tak, by wpis w tablicy symboli dla `buf` wskazywał na `COMMON`. Jaka jest rola tych wpisów w procesie tworzenia pliku wykonywalnego przez linker?

cpp=

int buf[2];

(2, bo używamy 0, 1 indeksy tej tablicy)

COMMON - nigdzie nie definiujemy jej wartości, ponieważ są to dane, których nie inicjalizujemy.

UNDEF - Zmienne, których deklaracje dostajemy podczas procesu linkowania z zewnętrznych plików, czyli zmienne ze słowem kluczowym extern.

Zadanie 8

Zadanie 8. Używając narzędzi do analizy **plików relokowalnych** w formacie ELF i bibliotek statycznych, tj. `objdump`, `readelf` i `ar` odpowiedz na następujące pytania:

1. Ile plików zawierają biblioteki `libc.a` i `libm.a` (katalog `/usr/lib/x86_64-linux-gnu`)?
2. Czy polecenie «`gcc -Og`» generuje inny kod wykonywalny niż «`gcc -Og -g`»?
3. Z jakich bibliotek współdzielonych korzysta interpreter języka `Python` (plik `/usr/bin/python`)?

Odpowiedź 1:

Żeby odpakować archiwum korzystamy z programu ar z wartością. Wypisuje ono content w postaci listy.
Wpisujemy zatem:

ar t /usr/lib/x86_64-linux-gnu/libc.a | wc -l

wc(word count) z flagą -l program policzy liczbę plików archiwalnych biblioteki (linie pliku).

*Wynik = 1690 *

Przy drugiej bibliotece polecenie podane wyżej nie działa. Otrzymujemy następujący wynik:

/usr/lib/x86_64-linux-gnu/libm.a: File format not recognized

Sprawdzamy zawartość tego pliku poleceniem:

cat /usr/lib/x86_64-linux-gnu/libm.a

Otrzymujemy:

/ GNU ld script*

**/*

OUTPUT_FORMAT(elf64-x86-64)

GROUP (/usr/lib/x86_64-linux-gnu/libm-2.27.a /usr/lib/x86_64-linux-gnu/libmvec.a)

Zatem mamy odniesienie do grupy dwóch archiwów:

/usr/lib/x86_64-linux-gnu/libm-2.27.a

/usr/lib/x86_64-linux-gnu/libmvec.a

Wystarczy więc teraz wywołać:

ar t /usr/lib/x86_64-linux-gnu/libmvec.a | wc -l

ar t /usr/lib/x86_64-linux-gnu/libm-2.27.a | wc -l

Otrzymane wyniki to: *129; 794*

Odpowiedź 2

Czy polecenie <<{gcc -Og}>> generuje inny kod wykonywalny niż «{gcc -Og -g}»?

Tak. Flaga -Og nie generuje symboli o etykiecie ".debug_*". Kompilacja bez flagi -g pozbawia nas symboli, które generowane są do łatwiejszego debugowania aplikacji.

Odpowiedź 3

Z jakich bibliotek współdzielonych korzysta interpreter języka Python (plik /usr/bin/python)?

Należy wykorzystać z programu "readelf" służącego do wizualizacji plików "Executable Linkable Format".

Polecenie: readelf -d /usr/bin/python

Dynamic section at offset 0x301370 contains 32 entries:

Tag	Type	Name/Value
0x0000000000000001	(NEEDED)	Shared library: [libc.so.6]
0x0000000000000001	(NEEDED)	Shared library: [libpthread.so.0]
0x0000000000000001	(NEEDED)	Shared library: [libdl.so.2]
0x0000000000000001	(NEEDED)	Shared library: [libutil.so.1]
0x0000000000000001	(NEEDED)	Shared library: [libz.so.1]
0x0000000000000001	(NEEDED)	Shared library: [libm.so.6]
0x000000000000000c	(INIT)	0x4c728
0x000000000000000d	(FINI)	0x1f3bd0
0x0000000000000019	(INIT_ARRAY)	0x5002b0
0x000000000000001b	(INIT_ARRAYSZ)	8 (bytes)
0x000000000000001a	(FINI_ARRAY)	0x5002b8
0x000000000000001c	(FINI_ARRAYSZ)	8 (bytes)
0x000000006ffffef5	(GNU_HASH)	0x298
0x0000000000000005	(STRTAB)	0xc930

0x0000000000000006 (SYMTAB) 0x2b80

0x000000000000000a (STRSZ) 28107 (bytes)

0x000000000000000b (SYMENT) 24 (bytes)

0x0000000000000015 (DEBUG) 0x0

0x0000000000000003 (PLTGOT) 0x5015b0

0x0000000000000002 (PLTRELSZ) 6960 (bytes)

0x0000000000000014 (PLTREL) RELA

0x0000000000000017 (JMPREL) 0x4abf8

0x0000000000000007 (RELA) 0x14550

0x0000000000000008 (RELASZ) 222888 (bytes)

0x0000000000000009 (RELAENT) 24 (bytes)

0x000000000000001e (FLAGS) BIND_NOW

0x000000006ffffffb (FLAGS_1) Flags: NOW PIE

0x000000006ffffffe (VERNEED) 0x14420

0x000000006fffffff (VERNEEDNUM) 6

0x000000006ffffff0 (VERSYM) 0x136fc

0x000000006ffffff9 (RELACOUNT) 9249

Widać więc 6 bibliotek współdzielonych, z których korzysta interpreter. Flaga -d wyświetla zawartość dynamicznych sekcji pliku.