מבנה מחשב - תרגיל 4

עידן טורקניץ 32668515
31.12.2021

הערה - הקוד הוא שילוב של פונקציות מתגבורים שונים בקורס מבוא למדמח

חלק א

Constant	Start Address
x	0x0000000000004010
c	0x0000000000004014
d	0x0000000000004018
"The new string is %s”	0x2015
"the result is %d"	0x2004

על מנת למצוא את הכתובות של המשתנים הגלובלים את הקוד בעזרת gdb, ורשמתי info variables

```
pwndbg> info variables
All defined variables:

Non-debugging symbols:
0x0000000000002000  _IO_stdin_used
0x000000000000202c  __GNU_EH_FRAME_HDR
0x00000000000021f4  __FRAME_END__
0x0000000000003db0  __frame_dummy_init_array_entry
0x0000000000003db0  __init_array_start
0x0000000000003db8  __do_global_dtors_aux_fini_array_entry
0x0000000000003db8  __init_array_end
0x0000000000003dc0  _DYNAMIC
0x0000000000003fb0  _GLOBAL_OFFSET_TABLE_
0x0000000000004000  __data_start
0x0000000000004000  data_start
0x0000000000004008  __dso_handle
0x0000000000004010  x
0x0000000000004014  c
0x0000000000004018  d
0x0000000000004020  __TMC_END__
0x0000000000004020  __bss_start
0x0000000000004020  _edata
0x0000000000004020  completed
0x0000000000004028  _end
```

לפי הקוד C רואים שהמשתנים הגלובליים נקראים x,c,d. אפשר גם למצוא את הערך שלהם ע"י :

```
pwndbg> x/d 0x0000000000004010
0x4010 <x>:     100
```

```
pwndbg> x/c 0x0000000000004014
0x4014 <c>:     97 'a'
```

```
pwndbg> x/f 0x0000000000004018
0x4018 <d>:     15.23
```

נשאר למצוא עוד 2 כתובות של מחרוזות.

נמצא את הכתובת של המחרוזת הנמצאת בסוף הפונקציה lower.

```
void lower(char *s){
    for (int i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');

    printf("The new string is %s", s);
}
```

נעשה disassembly לפונקציית lower ונקבל כי בסופה מופיע

```
0x0000000000001243 <+136>:   lea    0xdcb(%rip),%rdi        # 0x2015
0x000000000000124a <+143>:   mov    $0x0,%eax
0x000000000000124f <+148>:   callq  0x1070 <printf@plt>
```

כלומר קוראים לprintf עם rdi שבתוכו יש את הכתובת 0x2015, לכן המחרוזת
The new string is %s

נמצאת בכתובת 0x2015.

נוכל לבדוק זאת ע"י בדיקה מה יש בכתובת 0x2015

```
pwndbg> x/s 0x2015
0x2015: "The new string is %s"
```

באותו אופן, נמצא את המחרוזת

"the result is %d"

שנמצאת בסוף הפונקציה arith
נעשה disassembly לפונקציה

```
Dump of assembler code for function arith:
   0x0000000000001189 <+0>:     endbr64
   0x000000000000118d <+4>:     push   %rbp
   0x000000000000118e <+5>:     mov    %rsp,%rbp
   0x0000000000001191 <+8>:     sub    $0x20,%rsp
   0x0000000000001195 <+12>:    mov    %edi,-0x14(%rbp)
   0x0000000000001198 <+15>:    mov    %esi,-0x18(%rbp)
   0x000000000000119b <+18>:    mov    %edx,-0x1c(%rbp)
   0x000000000000119e <+21>:    mov    -0x14(%rbp),%edx
   0x00000000000011a1 <+24>:    mov    -0x18(%rbp),%eax
   0x00000000000011a4 <+27>:    add    %edx,%eax
   0x00000000000011a6 <+29>:    mov    %eax,-0xc(%rbp)
   0x00000000000011a9 <+32>:    mov    -0x1c(%rbp),%edx
   0x00000000000011ac <+35>:    mov    %edx,%eax
   0x00000000000011ae <+37>:    add    %eax,%eax
   0x00000000000011b0 <+39>:    add    %edx,%eax
   0x00000000000011b2 <+41>:    shl    $0x4,%eax
   0x00000000000011b5 <+44>:    mov    %eax,-0x8(%rbp)
   0x00000000000011b8 <+47>:    mov    -0xc(%rbp),%eax
   0x00000000000011bb <+50>:    imul   -0x8(%rbp),%eax
   0x00000000000011bf <+54>:    mov    %eax,-0x4(%rbp)
   0x00000000000011c2 <+57>:    mov    -0x4(%rbp),%eax
   0x00000000000011c5 <+60>:    mov    %eax,%esi
   0x00000000000011c7 <+62>:    lea    0xe36(%rip),%rdi        # 0x2004
   0x00000000000011ce <+69>:    mov    $0x0,%eax
   0x00000000000011d3 <+74>:    callq  0x1090 <printf@plt>
   0x00000000000011d8 <+79>:    nop
   0x00000000000011d9 <+80>:    leaveq
   0x00000000000011da <+81>:    retq
```

נוכל לראות כי לפי הפונקציה printf שומרים בrdi את הכתובת 0x2004, שהיא צריכה להיות הכתובת של המחרוזת שלנו.

```
pwndbg> x/s 0x2004
0x2004: "the result is %d"
```

ואכן כן היא.

לסיכום מצאנו כתובות של 3 משתנים גלובליים ו2 מחרוזות.

חלק ב

פונקציית *arith*

C code	Assembly	Optimization

Int t2 = z * 48	movl %edx, %eax
	addl %eax, %eax
	addl %edx, %eax
	sall $4, %eax
		במקום להכפיל ב48, הקומפיילר עשה חיבור של המשתנה לעצמו 3 פעמים ואז עשה shift 4 מקומות שמאלה, התוצאה של זה זהה כמו לעשות $z \cdot 3 \cdot 2^4 = z \cdot 3 \cdot 16 = 48z$
		
		האופטימזציה הזאת היא מסוג strength reduction

Int t1 = (x+y) / 2	shr $0x1f,%edx	במקום לחלק ב2, הקומפיילר עשה שיפט ימינה ב1. האופטימזציה הזאת היא מסוג strength reduction

פונקציית *returnSumWithBias*

C code	Assembly	Optimization

for(int i = 0; i < len; i++){
int bias = a * b;
sum += arr[i] + bias;
}
	imul %ecx,%edx
	(לפני הלולאה)
		הקומפיילר ביצע אופטימיזציה מסוג code motion, הוא שם לב שמחשבים את int bias = a * b כל פעם שנכנסים ללולאה, ולכן הוציא את החישוב שיתקיים פעם אחת לפני הלולאה

אלו אופטימזציות היחידות שהקומפיילר עשה כאשר קימפלתי ללא דגלי האופטימזיות (כלומר $-Oo$), על מנת למצוא עוד אופטימזציות בפונקציות אחרות, קיבלתי עם אופטימזציות בדרגה 2:

```
gcc  -O2 code.c -o MyEx
```

פונקציית *multArrBy2*

C code	Assembly	Optimization

int mult = 2;
for(int i = 0; i < len; i++){
arr[i] = arr[i] * mult;
}
	shll (%rdi)	הקומפיילר לא שמר את mult2, במקום זה במקום להכפיל כל איבר במערך ב2, הוא עשה שיפט ימינה ב1.

int i = 0;	xor %ebx,%ebx	במקום לאפס את i, הקומפיילר עשה xor עם עצמו strength reduction

פונקציית *returnSumWithBias*

C code	Assembly	Optimization
int i = 0;	xor %eax,%eax	במקום לאפס את i, הקומפיילר עשה xor עם עצמו strength reduction

חלק ג

- בפונקציות אשר איפסו חלק מהמשתנים, הגרסה עם הפחות אופטימזציות $O1$, השתמשה ב mov $0x0,%esi, אך בגרסאות האחרות $(O2, O3)$ השתמשו ב xor %eax,%eax

- בפונקציית arith אפשר לראות שמספר הפקודות בגרסת $O2$ קטן ממספר הפקודות בגרסת $O1$. ונשים לב גם כי אין הבדל בגרסאות $O2$ ו-$O3$ בפונקציה זו. כנראה מכיוון שזאת פונקציה קטנה, עם רק 3 חישובים, ואין איך לשפר אותה הרבה.

- בפונקציית multBy2, עם גרסת המקסימום אופטימזציות, ראיתי כי הקומפיילר דווקא האריך את אורך הקוד (הגדיל את מספר השורות), אך השתמש ברגיסטרים מסוג xmm. אני מאמין שזאת בשביל להשתמש בפעולות SIMD, ובכך לבצע את הפעולה של ההכפלה ב2 של האיברים במערך, בו זמנית.

תמונות:

פונקציית arith בגרסת $O1$

```
pwndbg> disassemble arith
Dump of assembler code for function arith:
   0x0000000000001169 <+0>:     endbr64
   0x000000000000116d <+4>:     sub    $0x8,%rsp
   0x0000000000001171 <+8>:     mov    %edx,%ecx
   0x0000000000001173 <+10>:    lea    (%rdi,%rsi,1),%edx
   0x0000000000001176 <+13>:    mov    %edx,%eax
   0x0000000000001178 <+15>:    shr    $0x1f,%eax
   0x000000000000117b <+18>:    add    %eax,%edx
   0x000000000000117d <+20>:    sar    %edx
   0x000000000000117f <+22>:    imul   %ecx,%edx
   0x0000000000001182 <+25>:    lea    (%rdx,%rdx,2),%edx
   0x0000000000001185 <+28>:    shl    $0x4,%edx
   0x0000000000001188 <+31>:    lea    0xe75(%rip),%rsi        # 0x2004
   0x000000000000118f <+38>:    mov    $0x1,%edi
   0x0000000000001194 <+43>:    mov    $0x0,%eax
   0x0000000000001199 <+48>:    callq  0x1070 <__printf_chk@plt>
   0x000000000000119e <+53>:    add    $0x8,%rsp
   0x00000000000011a2 <+57>:    retq
```

פונקציית arith בגרסאות $O2, O3$

```
pwndbg> disassemble arith
Dump of assembler code for function arith:
   0x00000000000011e0 <+0>:     endbr64
   0x00000000000011e4 <+4>:     mov    %edx,%r8d
   0x00000000000011e7 <+7>:     lea    (%rdi,%rsi,1),%edx
   0x00000000000011ea <+10>:    lea    0xe13(%rip),%rsi        # 0x2004
   0x00000000000011f1 <+17>:    mov    $0x1,%edi
   0x00000000000011f6 <+22>:    mov    %edx,%eax
   0x00000000000011f8 <+24>:    shr    $0x1f,%eax
   0x00000000000011fb <+27>:    add    %eax,%edx
   0x00000000000011fd <+29>:    xor    %eax,%eax
   0x00000000000011ff <+31>:    sar    %edx
   0x0000000000001201 <+33>:    imul   %r8d,%edx
   0x0000000000001205 <+37>:    lea    (%rdx,%rdx,2),%edx
   0x0000000000001208 <+40>:    shl    $0x4,%edx
   0x000000000000120b <+43>:    jmpq   0x1090 <__printf_chk@plt>
```

פונקציית multArrBy2 בגרסת $O3$

```
pwndbg> disassemble multArrBy2
Dump of assembler code for function multArrBy2:
   0x00000000000012c0 <+0>:     endbr64
   0x00000000000012c4 <+4>:     test   %esi,%esi
   0x00000000000012c6 <+6>:     jle    0x1320 <multArrBy2+96>
   0x00000000000012c8 <+8>:     lea    -0x1(%rsi),%eax
   0x00000000000012cb <+11>:    cmp    $0x2,%eax
   0x00000000000012ce <+14>:    jbe    0x1329 <multArrBy2+105>
   0x00000000000012d0 <+16>:    mov    %esi,%edx
   0x00000000000012d2 <+18>:    mov    %rdi,%rax
   0x00000000000012d5 <+21>:    shr    $0x2,%edx
   0x00000000000012d8 <+24>:    shl    $0x4,%rdx
   0x00000000000012dc <+28>:    add    %rdi,%rdx
   0x00000000000012df <+31>:    nop
   0x00000000000012e0 <+32>:    movdqu (%rax),%xmm0
   0x00000000000012e4 <+36>:    add    $0x10,%rax
   0x00000000000012e8 <+40>:    pslld  $0x1,%xmm0
   0x00000000000012ed <+45>:    movups %xmm0,-0x10(%rax)
   0x00000000000012f1 <+49>:    cmp    %rdx,%rax
   0x00000000000012f4 <+52>:    jne    0x12e0 <multArrBy2+32>
   0x00000000000012f6 <+54>:    mov    %esi,%eax
   0x00000000000012f8 <+56>:    and    $0xfffffffc,%eax
   0x00000000000012fb <+59>:    test   $0x3,%sil
   0x00000000000012ff <+63>:    je     0x1328 <multArrBy2+104>
   0x0000000000001301 <+65>:    movslq %eax,%rdx
   0x0000000000001304 <+68>:    shll   (%rdi,%rdx,4)
   0x0000000000001307 <+71>:    lea    0x1(%rax),%edx
   0x000000000000130a <+74>:    cmp    %edx,%esi
   0x000000000000130c <+76>:    jle    0x1320 <multArrBy2+96>
   0x000000000000130e <+78>:    movslq %edx,%rdx
   0x0000000000001311 <+81>:    add    $0x2,%eax
   0x0000000000001314 <+84>:    shll   (%rdi,%rdx,4)
   0x0000000000001317 <+87>:    cmp    %eax,%esi
   0x0000000000001319 <+89>:    jle    0x1320 <multArrBy2+96>
   0x000000000000131b <+91>:    cltq
   0x000000000000131d <+93>:    shll   (%rdi,%rax,4)
   0x0000000000001320 <+96>:    retq
   0x0000000000001321 <+97>:    nopl   0x0(%rax)
   0x0000000000001328 <+104>:   retq
   0x0000000000001329 <+105>:   xor    %eax,%eax
   0x000000000000132b <+107>:   jmp    0x1301 <multArrBy2+65>
```