Exercise Session 7

2024 Autumn

Disclaimer

- **Website**: n.ethz.ch/~falkbe/
- **(Extra) Demos on GitHub**: github.com/falkbe
- My exercise slides have **additional slides** (which are not official part of the course) **having a blue heading**: they are there to complement and go into more depth where I found appropriate
- For the exam **only** the official exercise slides are relevant, if in doubt always check the ones on the official moodle page

Agenda

- Recap Assembly (Code Expert)
- Assignment 4 – Questions
- Lecture Recap: Linking and Loading
- Own Introduction to GDB
- GDB Debugger Introduction
- GDB Demo
  - with `simple_bomb.c`
- Assignment 5 – Introduction & Tips
- Assignment 5: Walkthrough first defuse stage and setup

Assembly Recap

Theory & Code Expert Tasks

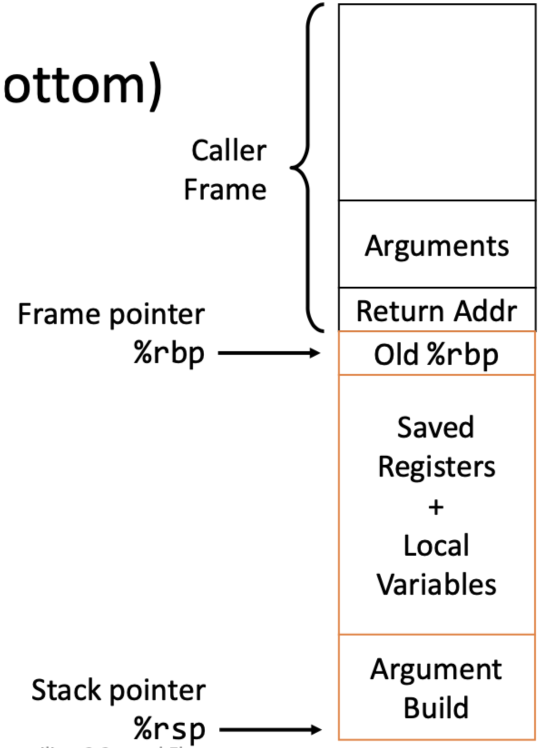
Agenda

- Is **stack frame** clear for everyone or should I go through it at the board **again** quickly, its **really important**?

Recall Stackframe

full x86_64/Linux stack frame

- Current stack frame ("top" to bottom)
  - "Argument build:"
    Parameters for function about to call
  - Local variables
    If can't keep in registers
  - Saved register context
  - Old frame pointer
- Caller stack frame
  - Return address
  - Pushed by `call` instruction
  - Arguments for this call



Recall Basic Operands

Some arithmetic operations

- Two-operand instructions (longword variants):

Mnemonic	*Format*	*Computation*	
`addl`	*Src,Dest*	*Dest ← Dest + Src*	
`subl`	*Src,Dest*	*Dest ← Dest - Src*	
`imull`	*Src,Dest*	*Dest ← Dest * Src*	
`sall`	*Src,Dest*	*Dest ← Dest << Src*	*Also called* `shll`
`sarl`	*Src,Dest*	*Dest ← Dest >> Src*	*Arithmetic*
`shrl`	*Src,Dest*	*Dest ← Dest >> Src*	*Logical*
`xorl`	*Src,Dest*	*Dest ← Dest ^ Src*	
`andl`	*Src,Dest*	*Dest ← Dest & Src*	
`orl`	*Src,Dest*	*Dest ← Dest | Src*	

- No distinction between signed and unsigned int (why?)

Recall Basic Operands

Some arithmetic operations

- One operand instructions

Mnemonic	*Format*	*Computation*
`incl`	*Dest*	*Dest* ← *Dest + 1*
`decl`	*Dest*	*Dest* ← *Dest - 1*
`negl`	*Dest*	*Dest* ← *-Dest*
`notl`	*Dest*	*Dest* ← *~Dest*

- See book for more instructions

Recall Calling Conventions

- We saw, that callee (the called function) always **stores the base pointer** of its parent function (caller), why does he care?
- S.t. there are registers where caller can be assured they are the same as when he called the callee

%rax	Return value, # varargs
%rbx	Callee saved; base ptr
%rcx	Argument #4
%rdx	Argument #3 (& 2nd return)
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack pointer
%rbp	Callee saved; frame ptr

%r8	Argument #5
%r9	Argument #6
%r10	Static chain ptr
%r11	Used for linking
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

Calling Conventions

- I find “callee saved” and ”caller saved” confusing: I remember
- “**callee owned**” (caller saved): callee owns them, so he can do whatever he wants with them
- “**caller owned**” (callee saved): caller owns them, so if the callee wants to do something with it he has to save them

Assembly Recap: Calling Conventions

- How to approach this?

Simple Arithmetic - Student attempt

```
.section .text
.global simple_arithmetic
simple_arithmetic:



# int simple_arithmetic(int a, int b)
# {
   return a + (3 * b) + 2;
  ret
# }


```

Assembly Recap: Calling Conventions

- How to approach this?
- **Idea**: remember 1. arg in %rdi, second one in %rsi
- **Careful**: we are passing INTs (4 bytes, use %edi, %esi)

Simple Arithmetic - Student attempt

```
.section .text
.global simple_arithmetic
simple_arithmetic:



# int simple_arithmetic(int a, int b)
# {
    return a + (3 * b) + 2;
  ret
# }
```

Assembly Recap: Calling Conventions

```
.section .text
.global simple_arithmetic
simple_arithmetic:
  xorl %eax, %eax    # zeros out return register
  movl %edi, %eax    # moves %rdi (a) into return register
  imul $3, %esi      # %esi (b) = 3*b
  addl %esi, %eax    # %eax (a) = a+3*b
  addl $2, %eax      # %eax (a+3*b) = a+3*b+2
  ret


# int simple_arithmetic(int a, int b)
# {
#   return a + (3 * b) + 2;
#  ret
# }
```

Recall Condition Codes: Explicitly via cmpx

Condition Codes (Explicit Setting: Compare)

- Explicit Setting by Compare Instruction

`cmpl/cmpq` *Src2,Src1*

`cmpl b,a` like computing `a - b` without setting destination

CF set if carry out from most significant bit (used for unsigned comparisons)

ZF set if `a == b`

SF set if `(a-b) < 0` (as signed)

OF set if two's complement (signed) overflow:

```
(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)
```

Recall Condition Codes: Explicitly via testx

Condition Codes (Explicit Setting: Test)

- Explicit Setting by Test instruction

  `testl/testq` *Src2,Src1*

  `testl b,a` like computing `a & b` without setting destination

  - Sets condition codes based on value of *Src1* & *Src2*
  - Useful to have one of the operands be a mask

  ZF set when `a & b == 0`
  SF set when `a & b < 0`

Calling Conventions

- **Note**: cmpx is like "subx" that means if we compare **immediate with a register**, we must have the register in the second place i.e. cmpl $1, %edi (subq %edi, $1 wouldn't make any sense either)
- Here we set condition codes **explicitly** i.e. we do the whole instruction BECAUSE we want the codes

- **Cmpx** does SUB, **testx** does logical AND

Recall Condition Codes: Implicitly

Condition codes (implicit setting)

- Single bit registers
  - CF Carry Flag (for unsigned) SF Sign Flag (for signed)
  - ZF Zero Flag OF Overflow Flag (for signed)

- Implicitly set (think of it as *side effect*) by arithmetic operations

  Example: addl/addq Src,Dest ↔ t = a+b
  - **CF set** if carry out from most significant bit (unsigned overflow)
  - **ZF set** if t == 0
  - **SF set** if t < 0 (as signed)
  - **OF set** if two's complement (signed) overflow
    (a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)

- ***Not*** set by lea instruction
- Full documentation link on course website

Calling Conventions

- **Note**: this means that for **ANY** instruction you do (add, sub, etc.) you set condition codes **implicitly** based on the result
- If you do subq %rax, %rdi and this yields 0 as a result => zeroflag is set **implicitly** you don't have to do anything and you cannot prevent it

Recap: Reading Condition Codes

Reading Condition Codes

- SetX Instructions
  - Set single byte based on combinations of condition codes

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF)&~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)|ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

Calling Conventions

- This means based on the condition codes, if we later want to do something like: **set bit** based on result, or **jump** etc. we need to read them
- Intutive understanding: if result is **zero** (ZF set), its "equality" so "sete" or "je"
- If you have a condition like if(a>2) think of what you want to do: you can do a>2 ⇔ a-2>0, so cmpq $2, %rdi ⇔ a-2, then we want to **jump** if its bigger so "jg"

Assembly Recap: Calling Conventions

```
 .section .text
.global branches
branches:


# int branches(int a)
# {
#   if(a < 12)
#   {
#     return -1;
#  }
#   else{
#     return 1;
#   }
# }
  ret

```

Assembly Recap: Calling Conventions

- Careful with **compare** again: compl needs register arg in 2^{nd} pos
- $A<12 \Leftrightarrow A-12<0$ for cmpl $12, %edi = %edi-12

```
 .section .text
.global branches
branches:
 cmpl $12, %edi
 jl IF
 movq $1, %rax
 retq

IF:
 movq $-1, %rax
 retq

# int branches(int a)
# {
#   if(a < 12)
#   {
#     return -1;
#  }
#   else{
#     return 1;
#   }
# }
  ret
```

Calling Conventions

- Also note: **when doing a function call** the rsp has to be 16 byte aligned according to calling conventions: so if rsp is currently only 8 byte aligned, before you call a function you need to substract another 8 byte from rsp and add this after the call to make sure you are aligned

Assembly Recap: Calli

- Why does this work then? It was 16 byte aligned when we were called
- We pushed one value, now its not 16 byte aligned?

Calling Functions - Student attempt

```
.section .text
.global calling_functions
calling_functions:
  pushq %rbp
  movq %rsp, %rbp
  movq %rsi, %rdi
  xorl %esi, %esi
  call call_me
  movq %rbp, %rsp
  popq %rbp
  ret


# int calling_functions(int a, int b)
# {
#    return call_me(b, NULL);
#   ret
# }
```

Test results All succeeded

Test 1 · 1 out of 1; calling_functions()

Assembly Recap: Calling C

- This fails?

Calling Functions - Student attempt

```
.section .text
.global calling_functions
calling_functions:
  pushq %rbp
  movq %rsp, %rbp
  movq %rsi, %rdi
  xorl %esi, %esi
  subq $8, %rsp
  call call_me
  addq $8, %rsp
  movq %rbp, %rsp
  popq %rbp
  ret


# int calling_functions(int a, int b)
# {
#   return call_me(b, NULL);
#  ret
# }
```

Test results All succeeded

Test 1 · 0 out of 1; calling_functions()

Assembly Recap: Calling Conv

- This works again

Calling Functions - Student attempt

```
.section .text
.global calling_functions
calling_functions:
  pushq %rbp
  movq %rsp, %rbp
  movq %rsi, %rdi
  xorl %esi, %esi
  subq $16, %rsp
  call call_me
  addq $16, %rsp
  movq %rbp, %rsp
  popq %rbp
  ret


# int calling_functions(int a, int b)
# {
#   return call_me(b, NULL);
#  ret
# }
```

Test results All succeeded

Test 1 · 1 out of 1; calling_functions()

Calling Conventions

- **Why**? When someone calls us, they have a 16 byte aligned stack pointer
- Then the ”call” function pushes the return address, so now rsp is NOT 16 byte aligned anymore
- **If** the callee doesn’t call another function himself hes fine, but **if** he calls another function he needs a 16 byte aligned RSP: gets this here **implicitly** by setting up the stack frame by pushing **rbp**
- **Makes sense?**

Assembly: local variables

- Generally, if compiler doesn't **explicitly** need a memory address for a local variable it will try to do it in a register
- **Add(long a, long b) = {return a+b;}**
- The compiler would do

```
add:
 movq %rdi, %rax
 addq %rsi, %rax
ret
```

- So it uses registers to do the calculation instead of using a stack relative address to store them

Assembly Recap: Calling Conventions

- Suggestions?

```
# int local_variables()
# {
#   int local = 3;
#   return call_me(local, &local);
#  ret
# }


```

Assembly Recap: Calling Conventions

- Will this work?

```
.section .text
.global local_variables
local_variables:
 pushq %rbp
 movq %rsp, %rbp     # set up stack frame
 subq $8, %rsp       # make space for local variable
 movq $3, (%rsp)     # move the local variable inside it
 movq (%rsp), %rdi   # deref rsp, move value in %rdi, 1st
 leaq (%rsp), %rsi   # put address of rsp into %rsi, 2nd
 call call_me        #call
 movq %rbp, %rsp
 popq %rbp           #deconstruct stack frame
 ret

# int local_variables()
# {
#    int local = 3;
#    return call_me(local, &local);
#  ret
# }
```

Assembly Recap: Callir

- Will this work?
- **ALMOST**, but remember alignment, we push %rbp: now its 16 byte aligned
- **BUT THEN** we subq $8 to make space for our local value: now its not 16 byte aligned anymore

Local Variables - Student attempt

```
.section .text
.global local_variables
local_variables:
 pushq %rbp
 movq %rsp, %rbp        # set up stack frame
 subq $8, %rsp          # make space for local variable
 movq $3, (%rsp)        # move the local variable inside it
 movq (%rsp), %rdi      # deref rsp, move value in %rdi, 1st
 leaq (%rsp), %rsi      # put address of rsp into %rsi, 2nd
 call call_me           #call
 movq %rbp, %rsp
 popq %rbp              #deconstruct stack frame
 ret

# int local_variables()
# {
#    int local = 3;
#    return call_me(local, &local);
#   ret
# }
```

Test results All succeeded

Test 1 · 0 out of 1; local_variables()

Assembly Recap: Calling

- Either substract 16 directly

Local Variables - Student attempt

```
.section .text
.global local_variables
local_variables:
 pushq %rbp
 movq %rsp, %rbp    # set up stack frame
 subq $16, %rsp      # make space for local variable
 movq $3, (%rsp)    # move the local variable inside it
 movq (%rsp), %rdi  # deref rsp, move value in %rdi, 1st
 leaq (%rsp), %rsi  # put address of rsp into %rsi, 2nd
 call call_me       #call
 movq %rbp, %rsp
 popq %rbp          #deconstruct stack frame
 ret

# int local_variables()
# {
#    int local = 3;
#    return call_me(local, &local);
#   ret
# }
```

Test results All succeeded

Test 1 · 1 out of 1; local_variables()

Assembly Recap: Ca

- Or 8 to store it
- Then another 8 for it to be aligned before and after the call

Local Variables - Student attempt

```
.section .text
.global local_variables
local_variables:
 pushq %rbp
 movq %rsp, %rbp     # set up stack frame
 subq $8, %rsp       # make space for local variable
 movq $3, (%rsp)     # move the local variable inside it
 movq (%rsp), %rdi   # deref rsp, move value in %rdi, 1st
 leaq (%rsp), %rsi   # put address of rsp into %rsi, 2nd
 subq $8, %rsp
 call call_me        #call
 addq $8, %rsp
 movq %rbp, %rsp
 popq %rbp           #deconstruct stack frame
 ret

# int local_variables()
# {
```

Test results All succeeded

Test 1 · 1 out of 1; local_variables()

Lecture Recap

Linking: Symbol Resolution and Relocation

Linking and Loading

- Whats the “issue” with this C program?

Example C program

main.c

```c
int buf[2] = {1, 2};

int main()
{
  swap();
  return 0;
}
```

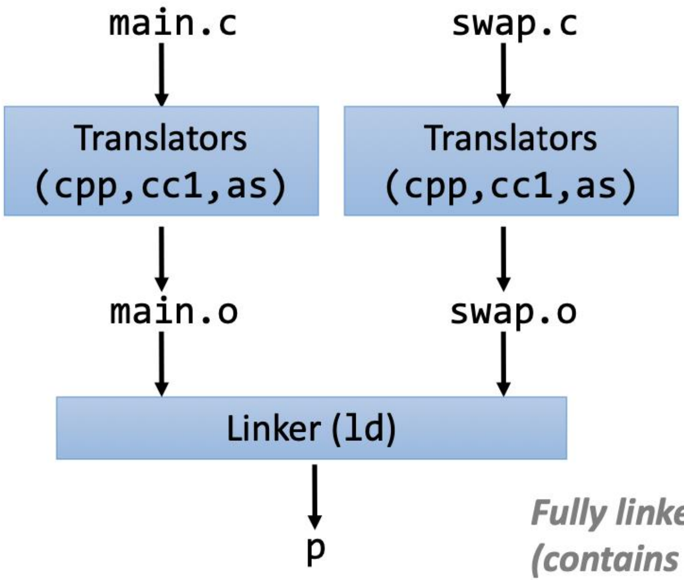
swap.c

```c
extern int buf[];

static int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1  = &buf[1];
  temp   = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

Static linking

- Programs are translated and linked using a *compiler driver*:

```
unix> gcc -O2 -g -o p main.c swap.c
unix> ./p
```



Source files

Separately compiled relocatable object files

Fully linked executable object file (contains code and data for all functions defined in main.c and swap.c

Static Linking in 2 Steps

- Step 1: Symbol resolution
  - Programs define and reference *symbols* (variables and functions):
    - `void swap() {…}    /* define symbol swap */`
    - `swap();            /* reference symbol swap */`
    - `int *xp = &x;      /* define xp, reference x */`
  - Symbol definitions are stored (by compiler) in *symbol table*.
    - Symbol table is an array of structs
    - Each entry includes name, type, size, and location of symbol.
  - Linker associates each symbol reference with exactly one symbol definition.

Static Linking in 2 Steps

What do linkers do?

- Step 2: Relocation
  - Merges separate code and data sections into single sections
  - Relocates symbols from their relative locations in the `.o` files to their final absolute memory locations in the executable.
  - Updates all references to these symbols to reflect their new positions.

Object files

3 kinds of object files (modules)

- Relocatable object file (.o file)
  - Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
  - Each .o file is produced from exactly one source (.c) file
- Executable object file
  - Contains code and data in a form that can be copied directly into memory and then executed.
- Shared object file (.so file)
  - Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
  - Called *Dynamic Link Libraries* (DLLs) by Windows

Object files: when a .c -> .o, how does this single .o get stored? That’s a **relocatable object file**

- Relocatable object file (.o file)
  - Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
  - Each .o file is produced from exactly one source (.c) file

ELF object file format

- Elf header
  - Word size, byte ordering, file type (`.o`, `exec`, `.so`), machine type, etc.
- Segment header table
  - Page size, virtual addresses memory segments (sections), segment sizes.
- `.text` section
  - Code
- `.rodata` section
  - Read only data: jump tables, ...
- `.data` section
  - Initialized global variables
- `.bss` section
  - Uninitialized global variables
  - “Block Started by Symbol”
  - “Better Save Space”
  - Has section header but occupies no space

ELF header
Segment header table (required for executables)
`.text` section
`.rodata` section
`.data` section
`.bss` section
`.symtab` section
`.rel.txt` section
`.rel.data` section
`.debug` section
Section header table

Resolving symbols: Global, External and Local

Resolving symbols

Global

```c
int buf[2] = {1, 2};

int main()
{
  swap();
  return 0;
}
```

main.c

External

External

Local

```c
extern int buf[];

static int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

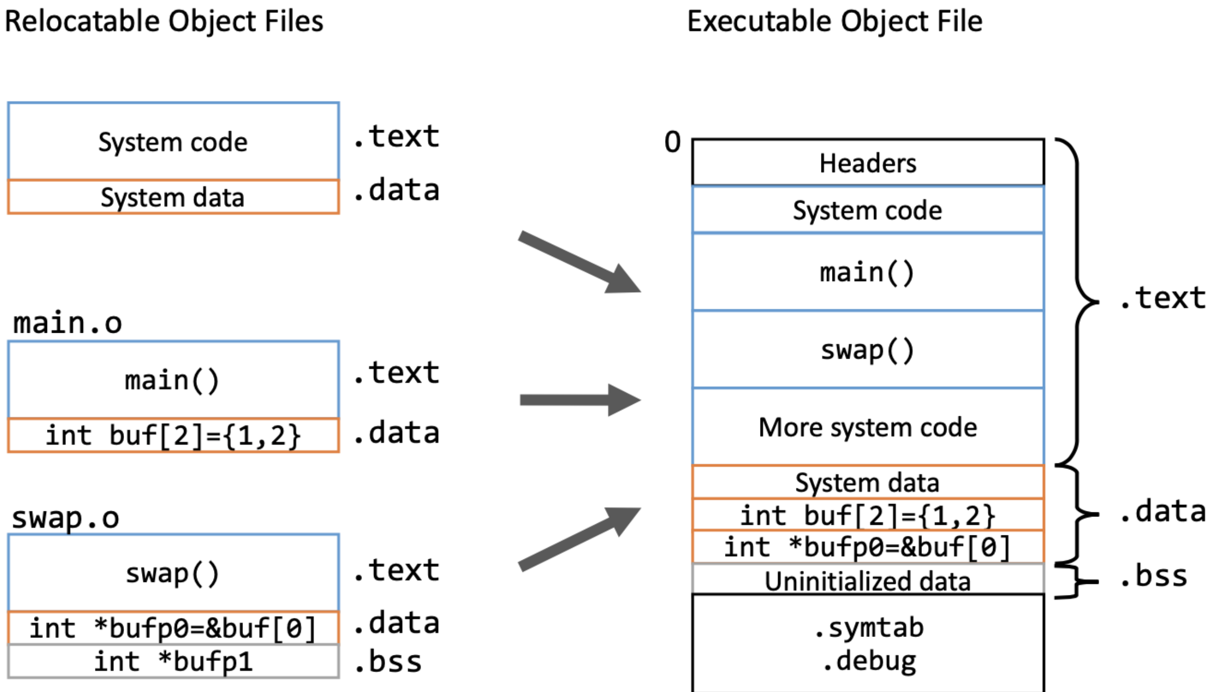
swap.c

Global

Linker knows nothing of temp

Put all the .o files (which are in ELF format) inside **ONE** big executable

Relocating code and data



Inside each .o file we are **still missing the references**: currently 0 as placeholder

Relocation info (`main`)

main.c

```
int buf[2] = {1,2};

int main()
{
  swap();
  return 0;
}
```

main.o

```
Disassembly of section .text:

0000000000000000 <main>:
   0:    48 83 ec 08       sub    $0x8,%rsp
   4:    b8 00 00 00 00    mov    $0x0,%eax
   9:    e8 00 00 00 00    callq  e <main+0xe>
                     a: R_X86_64_PC32 swap-0x4
   e:    b8 00 00 00 00    mov    $0x0,%eax
  13:    48 83 c4 08       add    $0x8,%rsp
  17:    c3                retq
```

```
Disassembly of section .data:

0000000000000000 <buf>:
   0:    01 00 00 00 02 00 00 00
```

Source: `objdump -D -r <file>`

After merging them: we can **check** in the symbol table where the function is in the final executable

- Now the linker can actually input the address of <swap> , but only after **everything got merged** since only then the final addresses are clear

Executable after relocation (.text)

```
00000000004004ed <main>:
  4004ed: 48 83 ec 08                 sub    $0x8,%rsp
  4004f1: b8 00 00 00 00              mov    $0x0,%eax
  4004f6: e8 0a 00 00 00              callq  400505 <swap>
  4004fb: b8 00 00 00 00              mov    $0x0,%eax
  400500: 48 83 c4 08                 add    $0x8,%rsp
  400504: c3                          retq
```

Lecture Recap

Linking: Issues with duplicate symbol definitions

Weak and Strong Symbols (UPDATED)

- Sometimes, **compiler** doesn’t know if symbol should be a **global symbol** or an **external** symbol
- So there is a concept of **strong and weak** symbols, for global variables

Recall 3 Type of Symbols (UPDATED)

Resolving symbols

Global

```
int buf[2] = {1, 2};

int main()
{
  swap();
  return 0;
}
```

main.c

External

External

Local

```
extern int buf[];

static int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

Global

swap.c

Linker knows nothing of temp

For **global symbols**, we have a further distinction (UPDATED)

Strong and weak symbols

- Program symbols are either strong or weak.
  By default:
  - ***Strong***: procedures and initialized globals
  - ***Weak***: uninitialized globals if `–fcommon`

p1.c

```
int foo=5;      <- strong

int p1() {      <- strong
}
```

p2.c

```
int foo;        <- weak

int p2() {      <- strong
}
```

Weak and Strong Symbols (UPDATED)

- **Note**: the concept of weak and strong linker symbols are related **exclusively** to **uninitialized global variables -f-common (old behaviour)**: puts uninitialised globals into a common block and they are weak symbols, allowing multiple uninitialized declerations across different .c files
- **-f-nocommon (new default)**: this is not the case anymore, uninitialized globals are **also classified as strong symbols**
- That means: the only weak to get weak symbols in C is to either compile with -f-common compiler flag or with #pragma weak

3 Linker Rules

Systems@ETH zürich

The linker's symbol rules

1. Multiple strong symbols are not allowed
   - Each item can be defined only once
   - Otherwise: Linker error

2. Given a strong symbol and multiple weak symbol, choose the strong symbol
   - References to the weak symbol resolve to the strong symbol

3. If there are multiple weak symbols, pick an arbitrary one
   - Can override this with `gcc -fno-common`

Will this yield an error?

Duplicate definitions

main.c:

```
int count = 0;
int main(int argc, char *argv[])
{
    count = 42;
    print_count();
    return 0;
}
```

other.c:

```
#include <stdio.h>
int count = 1;
void print_count()
{
    printf("Count is %d\n", count);
}
```

Will this yield an error?

- Yes: there are two definitions of the same symbol (but we can only have one)

Duplicate definitions

main.c:

```
int count = 0;
int main(int argc, char *argv[])
{
    count = 42;
    print_count();
    ret
}
```

other.c:

```
#include <stdio.h>
int count = 1;
void print_count()
{
    printf("Count is %d\n", count);
```

Link-time error: symbol multiply defined.

main.o:

```
0000000000000000 B count
000000000000000  T main
                 U print_count
```

Bss (uninitialized data)

other.o:

```
0000000000000000 D count
000000000000000  T print_count
                 U printf
```

Data segment (initialized data)

Will this yield an error?

One declaration and one definition

main.c:

```
extern int count;
int main(int argc, char *argv[])
{
    count = 42;
    print_count();
    return 0;
}
```

other.c:

```
#include <stdio.h>
int count = 1;
void print_count()
{
    printf("Count is %d\n", count);
}
```

Will this yield an error?

- **No**: the left extern int count (external linker symbol) refers to the RHS int count (global linker symbol)

One declaration and one definition

main.c:

```
extern int count;
int main(int argc, char *argv[])
{
    count = 42;
    print_count();
    ret
}
```

other.c:

```
#include <stdio.h>
int count = 1;
void print_count()
{
    printf("Count is %d\n", count);
```

No errors: program links and runs

main.o:

```
0000000000000000 U count
000000000000000  T main
                 U print_count
```

Undefined reference

other.o:

```
0000000000000000 D count
0000000000000000 T print_count
                 U printf
```

Will this yield an error?

Two declarations

main.c:

```
extern int count;
int main(int argc, char *argv[])
{
    count = 42;
    print_count();
    return 0;
}
```

other.c:

```
#include <stdio.h>
extern int count;
void print_count()
{
    printf("Count is %d\n", count);
}
```

Will this yield an error?

- **Yes**: we don't have a global symbol for count (no definition), only two references: but we would need a definition

Two declarations

main.c:

```
extern int count;
int main(int argc, char *argv[])
{
    count = 42;
    print_count();
    ret
}
```

other.c:

```
#include <stdio.h>
extern int count;
void print_count()
{
    printf("Count is %d\n", count);
```

Link-time error: undefined symbol count

main.o:

```
0000000000000000 U count
0000000000000000 T main
                 U print_count
```

other.o:

```
0000000000000000 U count
0000000000000000 T print_count
                 U printf
```

Will this yield an error?

What about this?

main.c:

```
int count;
int main(int argc, char *argv[])
{
    count = 42;
    print_count();
    return 0;
}
```

other.c:

```
#include <stdio.h>
int count = 1;
void print_count()
{
    printf("Count is %d\n", count);
}
```

Will this yield an error?

- **With –fcommon** the unitialised global is a weak symbol, as the other.c has definition for int count, the linker will turn “int count” i.e. the weak symbol into an external symbol and it links fine.
- **With –f-nocommon**: all global vars are strong, so we have 2 strong symbols which yields an error

With –fno-common
(default on very new compilers)

main.c:

```
int count;
int main(int argc, char *argv[])
{
    count = 42;
    print_count();
    ret
}
```

other.c:

```
#include <stdio.h>
int count = 1;
void print_count()
{
    printf("Count is %d\n", count);
```

Link-time error: symbol multiply defined.

main.o:

```
0000000000000000 B count
0000000000000000 T main
                 U print_count
```

other.o:

```
0000000000000000 D count
0000000000000000 T print_count
                 U printf
```

With –fcommon
(default pre-COVID)

main.c:

```
int count;
int main(int argc, char *argv[])
{
    count = 42;
    print_count();
    ret
}
```

other.c:

```
#include <stdio.h>
int count = 1;
void print_count()
{
    printf("Count is %d\n", count);
```

No errors: program links and runs

main.o:

```
0000000000000000 C count
000000000000000  T main
                 U print_count
```

“Common Block”:
weak symbol

other.o:

```
0000000000000000 D count
0000000000000000 T print_count
                 U printf
```

Will this yield an error?

Or this?

main.c:

```
int count;
int main(int argc, char *argv[])
{
    count = 42;
    print_count();
    return 0;
}
```

other.c:

```
#include <stdio.h>
int count;
void print_count()
{
    printf("Count is %d\n", count);
}
```

Will this yield an error?

- **No**: multiple weak symbols, it picks an arbitrary one (only works with -f-common)

Or this?

main.c:

```
int count;
int main(int argc, char *argv[])
{
    count = 42;
    print_count();
    ret
}
```

other.c:

```
#include <stdio.h>
int count;
void print_count()
{
    printf("Count is %d\n", count);
```

No errors: program links and runs

main.o:

```
0000000000000000 C count
0000000000000000 T main
                 U print_count
```

other.o:

```
0000000000000000 C count
0000000000000000 T print_count
                 U printf
```

Some Linker Puzzels (Assuming –fcommon)

`int x;` `int p1() {}`	`int p1() {}`	Link time error: two strong symbols (p1)
`int x;` `int p1() {}`	`int x;` `int p2() {}`	References to x will refer to the same uninitialized int. Is this what you really want?

`int x;`
`int y;`
`int p1() {}`
	`double x;`
	`int p2() {}`
		Writes to x in p2 might overwrite y!
		Evil!

`int x=7;`
`int y=5;`
`int p1() {}`
	`double x;`
	`int p2() {}`
		Writes to x in p2 will overwrite y!
		Nasty!

`int x=7;` `int p1() {}`	`int x;` `int p2() {}`	References to x will refer to the same initialized variable.

Lecture Recap

What are libraries?

Why do we need libraries?

Packaging commonly-used functions

- How to package functions commonly used by programmers?
  - Math, I/O, memory management, string manipulation, etc.

- Awkward, given the linker framework so far:
  - **Option 1:** Put all functions into a single source file
    - Programmers link big object file into their programs
    - Space and time inefficient
  - **Option 2:** Put each function in a separate source file
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer

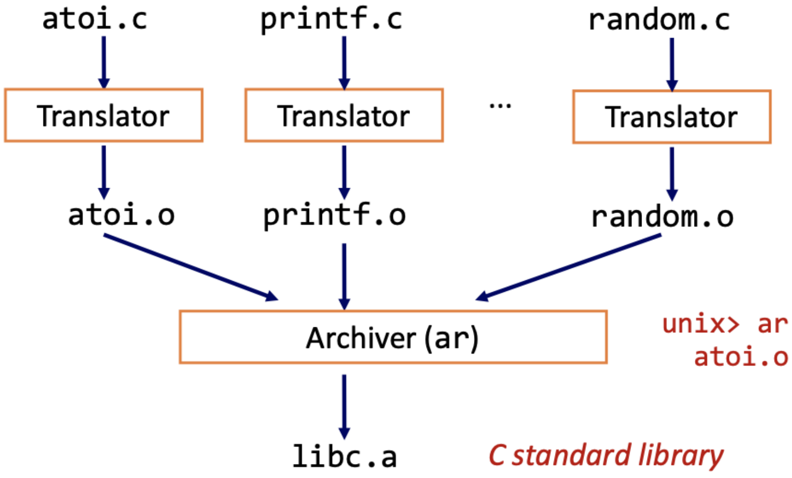
What are (static) libraries

Solution: static libraries

- Static libraries (`.a` archive files)
  - Concatenate related relocatable object files into a single file with an index (called an *archive*).
  - Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - If an archive member file resolves reference, link into executable.

What are (static) libraries

Creating static libraries




```
unix> ar rs libc.a \
  atoi.o printf.o … random.o
```

C standard library

Archiver allows incremental updates

Recompile function that changes and replace .o file in archive.

Example: check your linux system

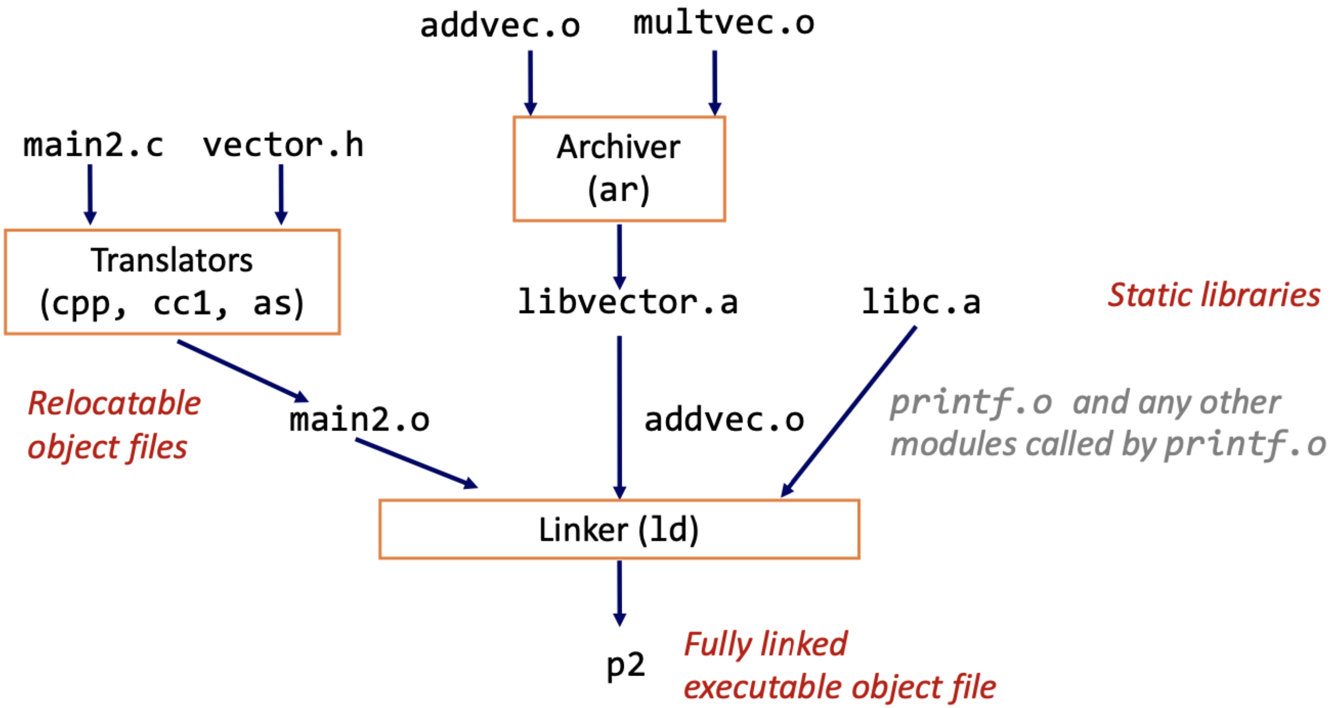
Commonly-used libraries

- `libc.a` (the C standard library)
  - 8 MB archive of 900 object files.
  - I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math
- `libm.a` (the C math library)
  - 1 MB archive of 226 object files.
  - floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% ar -t /usr/lib/libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
% ar -t /usr/lib/libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

Linking with static libraries



Assignment 4

Questions?

GDB

Overview

Quick Introduction to Debugging

GDB («Gnu DeBugger»)

Debugging Intro

- What is debugging?

- **So far (Eprog, Pprog)**: probably just printed out everything

Debugging Intro

- Might have worked then: but generally not a good idea, **especially not** when doing low level stuff

- **It has a** huge advantage, going instruction by instruction through your program, and in each step **you can check** which value is in which register, when you do which function call etc.
- **Seems abstract** but I will show you this later on an example, first some basics about **gdb**

Debugging Intro: Start gdb

- **Gdb (binary)** starts the given program: then we see nothing, now what?

```
user@4865b4f533e3:~/exs7/bomb510$ gdb bomb
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bomb...
(gdb) 
```

```
(gdb) layout asm
```

Debugging Intro: Start gdb

- **I recommend**: first start assembly view of the binary (its like **objdump** but directly as a view window)

```
0x1489 <main>           endbr64
0x148d <main+4>         push   %rbx
0x148e <main+5>         cmp    $0x1,%edi
0x1491 <main+8>         je     0x158f <main+262>
0x1497 <main+14>        mov    %rsi,%rbx
0x149a <main+17>        cmp    $0x2,%edi
0x149d <main+20>        jne    0x15c4 <main+315>
0x14a3 <main+26>        mov    0x8(%rsi),%rdi
0x14a7 <main+30>        lea    0x1b56(%rip),%rsi        # 0x3004
0x14ae <main+37>        call   0x1320 <fopen@plt>
0x14b3 <main+42>        mov    %rax,0x41f6(%rip)        # 0x56b0 <infile>
0x14ba <main+49>        test   %rax,%rax
0x14bd <main+52>        je     0x15a2 <main+281>
0x14c3 <main+58>        call   0x1b40 <initialize_bomb>
0x14c8 <main+63>        lea    0x1bb9(%rip),%rdi        # 0x3088
0x14cf <main+70>        call   0x1220 <puts@plt>
0x14d4 <main+75>        lea    0x1bed(%rip),%rdi        # 0x30c8
0x14db <main+82>        call   0x1220 <puts@plt>
0x14e0 <main+87>        call   0x1de2 <read_line>
0x14e5 <main+92>        mov    %rax,%rdi
0x14e8 <main+95>        call   0x15e7 <phase_1>
0x14ed <main+100>       call   0x1f1a <phase_defused>
0x14f2 <main+105>       lea    0x1bff(%rip),%rdi        # 0x30f8
0x14f9 <main+112>       call   0x1220 <puts@plt>
0x14fe <main+117>       call   0x1de2 <read_line>
0x1503 <main+122>       mov    %rax,%rdi
0x1506 <main+125>       call   0x160b <phase_2>
0x150b <main+130>       call   0x1f1a <phase_defused>
0x1510 <main+135>       lea    0x1b26(%rip),%rdi        # 0x303d
0x1517 <main+142>       call   0x1220 <puts@plt>
0x151c <main+147>       call   0x1de2 <read_line>

exec No process In:                                          L??   PC: ??
(gdb)
```

```
(gdb) layout asm
```

Debugging Intro: Start gdb

- **You** can now run your code and also give it arguments

```
   0x1489 <main>          endbr64
   0x148d <main+4>        push   %rbx
   0x148e <main+5>        cmp    $0x1,%edi
   0x1491 <main+8>        je     0x158f <main+262>
   0x1497 <main+14>       mov    %rsi,%rbx
   0x149a <main+17>       cmp    $0x2,%edi
   0x149d <main+20>       jne    0x15c4 <main+315>
   0x14a3 <main+26>       mov    0x8(%rsi),%rdi
   0x14a7 <main+30>       lea    0x1b56(%rip),%rsi        # 0x3004
   0x14ae <main+37>       call   0x1320 <fopen@plt>
   0x14b3 <main+42>       mov    %rax,0x41f6(%rip)        # 0x56b0 <infile>
   0x14ba <main+49>       test   %rax,%rax
   0x14bd <main+52>       je     0x15a2 <main+281>
   0x14c3 <main+58>       call   0x1b40 <initialize_bomb>
   0x14c8 <main+63>       lea    0x1bb9(%rip),%rdi        # 0x3088
   0x14cf <main+70>       call   0x1220 <puts@plt>
   0x14d4 <main+75>       lea    0x1bed(%rip),%rdi        # 0x30c8
   0x14db <main+82>       call   0x1220 <puts@plt>
   0x14e0 <main+87>       call   0x1de2 <read_line>
   0x14e5 <main+92>       mov    %rax,%rdi
   0x14e8 <main+95>       call   0x15e7 <phase_1>
   0x14ed <main+100>      call   0x1f1a <phase_defused>
   0x14f2 <main+105>      lea    0x1bff(%rip),%rdi        # 0x30f8
   0x14f9 <main+112>      call   0x1220 <puts@plt>
   0x14fe <main+117>      call   0x1de2 <read_line>
   0x1503 <main+122>      mov    %rax,%rdi
   0x1506 <main+125>      call   0x160b <phase_2>
   0x150b <main+130>      call   0x1f1a <phase_defused>
   0x1510 <main+135>      lea    0x1b26(%rip),%rdi        # 0x303d
   0x1517 <main+142>      call   0x1220 <puts@plt>
   0x151c <main+147>      call   0x1de2 <read_line>

exec No process In:                                        L??   PC: ??
(gdb)
```

- **Generally**

```
(gdb) run arg1 arg2
```

Debugging Intro: Moving inside gdb

- **After** having started the program we can move

```
(gdb) run arg1 arg2
```

- One **soure code** instruction forward, is “next”, one **assembly code** instruction is “nexti” or “ni” for short: next **does NOT step** into functions
- “step” and ”stepi” or “si” for short **steps into functions**
- **For** your lab “**ni**” will be the most important one

Debugging Intro: Breakpoints and Wachtpoints

- One **huge advantage** of debuggers: lets you set “breakpoints”, i.e. points where your debugger goes to, and then you can step through this function slowly, **instruction by instruction**

- While doing this you can **print values** in **registers, variables and in memory** (on the stack for instance)

Debugging Intro: Breakpoints and Wachtpoints

- "**Breakpoint <location>**" or "**b <location>**" for short to set

```
(gdb) b main            # Break at function `main`
(gdb) b 42              # Break at line 42
(gdb) b file.c:10       # Break at line 10 in file.c
```

- "**delete <breakpoint>**" or "**d <breakpoint**" for short to delete
- "**clear**" to **delete all** breakpoints

Debugging Intro: Printing and Expecting Variables

- “**print <expression>**” or “**p <expression>**” prints values of expressions, variables or registers

```
(gdb) p x           # Print variable `x`
(gdb) p $rdi        # Print value in `rdi` register
(gdb) p/x $rdi      # Print `rdi` register in hexadecimal
```

- “**x <expression>**” examines memory at a specific address

```
(gdb) x/x &x         # Examine memory at address of `x`, in hex
(gdb) x/d $rsp       # Examine memory at address in `rsp` register, in decimal
(gdb) x/s <address>  # Examine as a string
```

General Debugging Information

- **Recall**: we have “**objdump -d <binaryname>**” to look at the **assembly code from a binary** (reversing the step of compiling and assemlying)

- **Compiling**: source code -> assembly -> **executable**
- **Objdump**: **assembly** <- executable

General Debugging Information

- **If** assembly view from our gdb breaks do “ctrl l” for control load
- To **exit** either “q” or “ctrl d”

Debugging

"If debugging is the process of removing bugs, then programming must be the process of putting them in."

Problem

C Source

```c
int foo(char *a) {
    return strlen(a);
}

int main(...) {
    char *a = NULL;
    printf("%d", foo(a));
    return 0;
}
```

Output

Segmentation fault

Problem:

The output does not tell you where the Segmentation fault happened

Solution

Use a debugger to execute the program step by step

In our case this will be gdb https://sourceware.org/gdb/documentation/

With help from the binutils https://sourceware.org/binutils/docs/binutils/

Getting the Assembly

`objdump`:
display info about object files

Note: The generated code not necessarily looks that good.

`example.c`

```c
#include <stdio.h>

int foo(int a) {
    printf("%d", a);
    return a;
}

int main(int argc, char** argv) {
   int b = 10;
   int c = foo(b);
   return c;
}
```

Getting the Assembly

Stop after compiler

Example program assembly file with gcc: `gcc -S example.c`

```
cat example.s
        .file   "example.c"
        .text
        .section        .rodata
.LC0:
        .string "%d"
        .text
        .globl  foo
        .type   foo, @function
foo:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        movl    -4(%rbp), %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   foo, .-foo
        .globl  main
        .type   main, @function
main:
.LFB1:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $32, %rsp
        movl    %edi, -20(%rbp)
        movq    %rsi, -32(%rbp)
        movl    $10, -8(%rbp)
        movl    -8(%rbp), %eax
        movl    %eax, %edi
        call    foo
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE1:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
        .section        .note.GNU-stack,"",@progbits
```

Getting the Assembly

Comparison:

output of `objdump -d ./example`

```
000000000000064a <foo>:
 64a:   55                      push   %rbp
 64b:   48 89 e5                mov    %rsp,%rbp
 64e:   48 83 ec 10             sub    $0x10,%rsp
 652:   89 7d fc                mov    %edi,-0x4(%rbp)
 655:   8b 45 fc                mov    -0x4(%rbp),%eax
 658:   89 c6                   mov    %eax,%esi
 65a:   48 8d 3d c3 00 00 00    lea    0xc3(%rip),%rdi        # 724 <_IO_stdin_used+0x4>
 661:   b8 00 00 00 00          mov    $0x0,%eax
 666:   e8 b5 fe ff ff          callq  520 <printf@plt>
 66b:   8b 45 fc                mov    -0x4(%rbp),%eax
 66e:   c9                      leaveq
 66f:   c3                      retq

0000000000000670 <main>:
 670:   55                      push   %rbp
 671:   48 89 e5                mov    %rsp,%rbp
 674:   48 83 ec 20             sub    $0x20,%rsp
 678:   89 7d ec                mov    %edi,-0x14(%rbp)
 67b:   48 89 75 e0             mov    %rsi,-0x20(%rbp)
 67f:   c7 45 f8 0a 00 00 00    movl   $0xa,-0x8(%rbp)
 686:   8b 45 f8                mov    -0x8(%rbp),%eax
 689:   89 c7                   mov    %eax,%edi
 68b:   e8 ba ff ff ff          callq  64a <foo>
 690:   89 45 fc                mov    %eax,-0x4(%rbp)
 693:   8b 45 fc                mov    -0x4(%rbp),%eax
 696:   c9                      leaveq
 697:   c3                      retq
 698:   0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
 69f:   00
```

objdump

Param	Description
-d	**Display the assembly of the machine instructions (only sections which are expected to contain instructions)**
-D	Display the assembly of all sections
-l	Display line numbers when debugging information are present
-r	Print the relocation entries
-S	Display the source code (only if possible)
-t	**Display the symbol table entries**
-x	Equivalent to –a –f -h –p –r –t

Getting String Info

`strings`:
Prints printable character sequences > 3 chars with ‘\0’ termination.

This is helpful to get strings used in the `printf`s

```
printf(“Result %d”, 123);
```

strings

Param	Description
-a	Scan whole file, not just initialized and loaded sections
-n	Change minimum string length

So Far

Outputs show program structure

but no information about execution

→ Next step: run program in `gdb`

Debug Info

Include debug info in binary:

Compile with `-g` flag and have source code available

In the assignment:

`bomb.c` has debug info

`phase_N()` does not have debug info

Bomb: Debug Info

```
reto@reto-VirtualBox: ~/eth/casp2013/bomblab
74          phase_defused();                /* Drat!  They figured it out!
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/reto/eth/casp2013/bomblab/bomb

Breakpoint 1, main (argc=1, argv=0xbffff214) at bomb.c:36
36      {
(gdb) step
44          if (argc == 1) {
(gdb) step
45              infile = stdin;
(gdb) step
66          initialize_bomb();
(gdb) step
105     }
(gdb) step
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
72          input = read_line();            /* Get input                   */
(gdb) step
fff
73          phase_1(input);                 /* Run the phase               */
(gdb) stepi
0x08048afe      73          phase_1(input);                 /* Run the phase
   */
(gdb) stepi
0x08048bb0 in phase_1 ()
(gdb) stepi
0x08048bb3 in phase_1 ()
(gdb) 
```

debug info
(source code)

no debug info

GDB: Interactive Shell

gdb behaves pretty much like Linux shell

auto completion, history of commands, …

Not sure about a command?

- Online documentation
  http://www.gnu.org/software/gdb/documentation/

  Cheat Sheet
  http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf

- GDB help
  `(gdb) help [command]`

Start GDB

Start gdb with binary as argument:

```
gdb the_program
(gdb)
```

Start gdb and then load binary afterwards:

```
gdb
(gdb) file the_program
```

`(gdb)` at beginning of line indicates GDB running

Run Program

run program (also restart)

```
(gdb) run
```

program runs like in shell directly

Additional information like

- Function
- Line
- File

where the crash occurred is missing

Breakpoint: Set

In file `file.c` at line 123:

```
(gdb) break file.c:123
```

At function `foo()`:

```
(gdb) break foo
```

At address:

```
(gdb) break *0x80487dd
```

Breakpoint at next instruction to be executed:

```
(gdb) break
```

Delete breakpoint:

```
(gdb) delete <breakpoint>
```

Show information about all breakpoints:

```
(gdb) info breakpoints
```

Breakpoint: Execute

Until next source code line
(first instruction of new line)
(if debug info available)

```
(gdb) step [n]
```

Until next line of source code but function calls are one instruction.
(like step)

```
(gdb) next [n]
```

One assembly instruction

```
(gdb) stepi [n]
```

One instruction, but function calls are one instruction

```
(gdb) nexti [n]
```

`step` goes into function calls,
`next` goes over them

Breakpoint: Execute

Until next breakpoint

```
(gdb) continue
```

Until current function returns

```
(gdb) finish
```

Every time we hit a breakpoint:
The program pauses and gdb prompts for a command

Breakpoint: Conditional

Trigger breakpoint only if condition holds:

```
(gdb) break file.c:123 if variable > 456
```

Also works for watchpoints

Watchpoint

Be informed about changes to a variable

Like setting a breakpoint on the assignment operator for a certain variable

You will see the old and new values

Set a watchpoint:

```
(gdb) watch <variable>
```

Systems@ETH zürich

Program State

Print content of a variable:

```
(gdb) print variable|address
(gdb) print/x variable|address
```

Treat variable as string:

```
(gdb) x/s stringvariable|address
```

GDB printf:

```
(gdb) printf "%s\n", stringvariable|address
```

CPU registers:

```
(gdb) info registers
```

Program State

You can access pointers like in C

Pointer address:

```
(gdb) print ptr
```

Value of struct field:

```
(gdb) print ptr->field
```

All struct content

```
(gdb) print *ptr
```

Useful Commands

Stack trace at seg fault

```
(gdb) backtrace
```

Stack trace at current position (e.g. how did it get to this breakpoint)

```
(gdb) where
```

Show source/assembly code around current position

```
(gdb) list
(gdb) disassemble
```

GDB UI

activate nice “TUI” layout

```
(gdb) layout asm
```

default layout

```
(gdb) layout off
```

gdb-dashboard (**recommended**)
or another (ddd, pwndbg, …)

[gdb-dashboard]()

Binary Edit

To edit your binary use a hex editor, e.g. GHex

```
apt install ghex
ghex <file>
```

Demo of GDB on `simple_bomb`

Quick Introduction to the idea of the lab / following demo

- Given is some code, which calls a secret function, a “bomb” which you have to defuse by giving it a certain input
- However, you only see that the function is called and not what its actually doing: you **only have the executable** not the source code

- **First idea**?

Quick Introduction to the idea of the lab / following demo

- Given is some code, which ca
  which you have to defuse by
- However, you only see that t
  what its actually doing: you c   e
  source code

- **First idea**?
- Look at assembly

Quick Introductio following demo

- **On a second thought**: don’t want to read 1700 lines of assembly (for your lab)

```
0000000000002b0d <driver_post>:
    2b0d: f3 0f 1e fa             endbr64
    2b11: 53                      pushq   %rbx
    2b12: 4c 89 c3                movq    %r8, %rbx
    2b15: 85 c9                   testl   %ecx, %ecx
    2b17: 75 17                   jne     0x2b30 <driver_post+0x23>
    2b19: 48 85 ff                testq   %rdi, %rdi
    2b1c: 74 05                   je      0x2b23 <driver_post+0x16>
    2b1e: 80 3f 00                cmpb    $0x0, (%rdi)
    2b21: 75 33                   jne     0x2b56 <driver_post+0x49>
    2b23: 66 c7 03 4f 4b          movw    $0x4b4f, (%rbx)         # imm = 0x4B4F
    2b28: c6 43 02 00             movb    $0x0, 0x2(%rbx)
    2b2c: 89 c8                   movl    %ecx, %eax
    2b2e: 5b                      popq    %rbx
    2b2f: c3                      retq
    2b30: 48 8d 35 5a 0a 00 00    leaq    0xa5a(%rip), %rsi       # 0x3591 <array.0+0x3d1>
    2b37: bf 01 00 00 00          movl    $0x1, %edi
    2b3c: b8 00 00 00 00          movl    $0x0, %eax
    2b41: e8 ca e7 ff ff          callq   0x1310 <.plt.sec+0x120>
    2b46: 66 c7 03 4f 4b          movw    $0x4b4f, (%rbx)         # imm = 0x4B4F
    2b4b: c6 43 02 00             movb    $0x0, 0x2(%rbx)
    2b4f: b8 00 00 00 00          movl    $0x0, %eax
    2b54: eb d8                   jmp     0x2b2e <driver_post+0x21>
    2b56: 41 50                   pushq   %r8
    2b58: 52                      pushq   %rdx
    2b59: 4c 8d 0d 48 0a 00 00    leaq    0xa48(%rip), %r9        # 0x35a8 <array.0+0x3e8>
    2b60: 49 89 f0                movq    %rsi, %r8
    2b63: 48 89 f9                movq    %rdi, %rcx
    2b66: 48 8d 15 44 0a 00 00    leaq    0xa44(%rip), %rdx       # 0x35b1 <array.0+0x3f1>
    2b6d: be 6e 3b 00 00          movl    $0x3b6e, %esi           # imm = 0x3B6E
    2b72: 48 8d 3d 0b 0a 00 00    leaq    0xa0b(%rip), %rdi       # 0x3584 <array.0+0x3c4>
    2b79: e8 66 f5 ff ff          callq   0x20e4 <submitr>
    2b7e: 48 83 c4 10             addq    $0x10, %rsp
    2b82: eb aa                   jmp     0x2b2e <driver_post+0x21>

Disassembly of section .fini:

0000000000002b84 <_fini>:
    2b84: f3 0f 1e fa             endbr64
    2b88: 48 83 ec 08             subq    $0x8, %rsp
    2b8c: 48 83 c4 08             addq    $0x8, %rsp
    2b90: c3                      retq
```

Quick Introduction to the idea of the lab / following demo

- **That's** not even the issue: the actual issue is you **will blow up your bomb** all the time if you cant go through it step by step (ill show you later)

Quick Introduction to the idea of the lab / following demo

- That’s why we use **gdb**, the ”debugger” introduced before

- “**sscanf(input, "%*s %d", &middle)**”
- **Also** by now you should know:
- 1st argument → rdi
- 2nd argument → rsi
- 3rd argument → rdx
- 4th argument → rcx, etc.

Assignment 5

Bomb Lab

People here with ARM native macbooks using x86 Linux in Docker?

Systems@ETH zürich

- Apparently you cannot look into registers in this configuration
- **Use maximus** via ssh (ask me if you need help setting it up), since maximus is native x86

- **Also**, please don’t run anything **compute intensive**, and don’t something which crashes the cluster, also be aware it has **2 cores**, once multiple people are on it it gets slow

Backstory

Welcome Mr. Powers,

Here is your individual bomb.

I am friendly enough to give you the bomb’s main function, but it won’t help much.

Setup

Individual bomb (executable binary)

Different from everybody else’s

-> solution differs

`assignment5/bomb*/bomb`

Bomb’s main function given

`assignment5/bomb*/bomb.c`

Hints

Write key file & supply it via argument to avoid typing the known keys

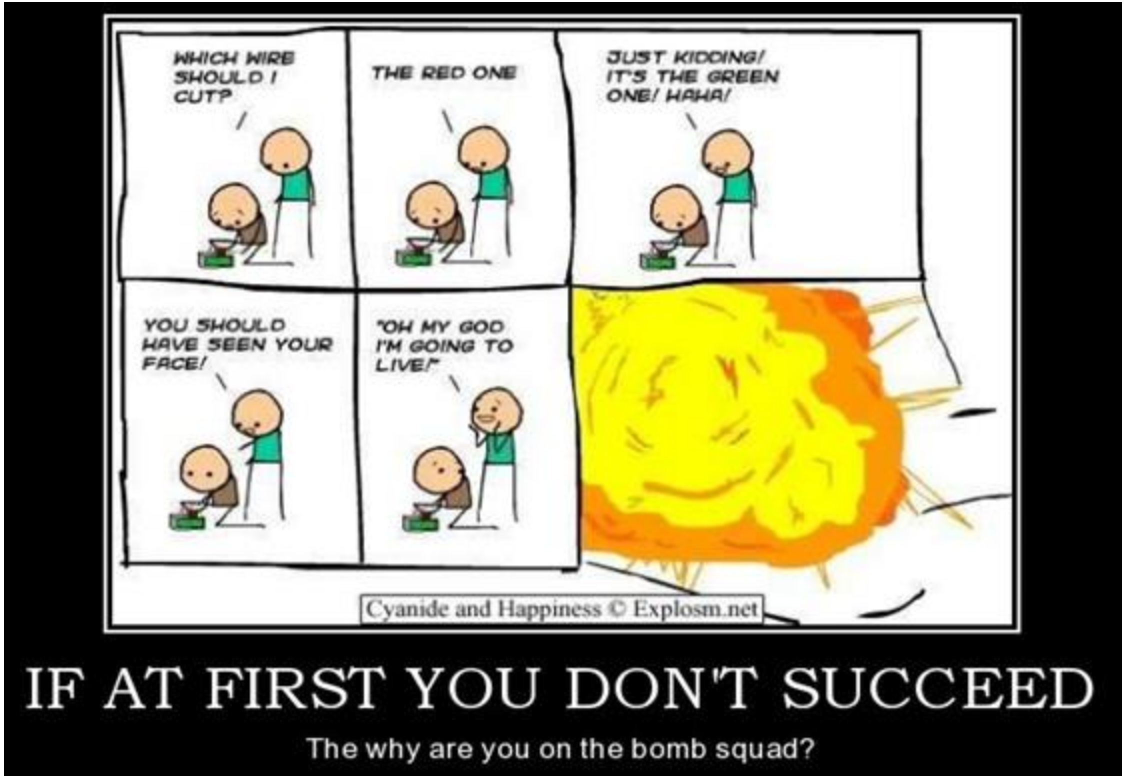
```
./bomb psol.txt
```

After using up all provided keys, bomb program switches to stdin (→ you can type)

Don’t go into C library functions

`printf()`, `malloc()`, etc.

Strategy


WHICH WIRE SHOULD I CUT?
THE RED ONE
JUST KIDDING! IT'S THE GREEN ONE! HAHA!
YOU SHOULD HAVE SEEN YOUR FACE!
"OH MY GOD I'M GOING TO LIVE!"
Cyanide and Happiness © Explosm.net
IF AT FIRST YOU DON'T SUCCEED
The why are you on the bomb squad?

Strategy

Get overview of program

Think of when to set breakpoints (functions, lines, …) or watchpoints (variables)

You don’t want the bomb to explode think about how to prevent that

Submission

Server graded

Follow instructions in assignment

Ensure path & filenames are as stated

Let me give helpful tips

- There are 6 phases (each one like on the RHS)

```c
    initialize_bomb();

    printf("Welcome to my fiendish little bomb. You have 6 phases with\n");
    printf("which to blow yourself up. Have a nice day!\n");

    /* Hmm...  Six phases must be more secure than one phase! */
    input = read_line();             /* Get input                   */
    phase_1(input);                  /* Run the phase               */
    phase_defused();                 /* Drat!  They figured it out!
                                      * Let me know how they did it. */
    printf("Phase 1 defused. How about the next one?\n");

    /* The second phase is harder.  No one will ever figure out
     * how to defuse this... */
    input = read_line();
    phase_2(input);
    phase_defused();
    printf("That's number 2.  Keep going!\n");

    /* I guess this is too easy so far.  Some more complex code will
     * confuse people. */
    input = read_line();
    phase_3(input);
    phase_defused();
    printf("Halfway there!\n");
```

Let me give you an **actual** preview, and helpful tips for the lab to get started

- As said before, you only have access to the binary and bomb.c file which calls functions where you don't have **access to the source code** => that's why we need objdump or gdb

```
al 72
xr-xr-x   1 benediktfalk   staff     30K 30 Okt 09:34 bomb*
-r--r--   1 benediktfalk   staff    4,0K 30 Okt 09:34 bomb.c
```

Let me give you an **actual** preview, and helpful tips for the lab to get started

- S.t. you don’t have to retype everything once you passed a level: write your strings in a file and call the binary with the file instead of an actual string
- So **create this file first** before you do anything, you don’t even have to write anything in it just “touch <filename>”
- I called mine ”**defuse**”

Let me give you an **actual** preview, and helpful tips for the lab to get started

- How to get started (example for first bomb): start gdb, set breakpoint at the first function that gets called

```
    input = read_line();             /* Get input                   */
    phase_1(input);                  /* Run the phase               */
    phase_defused();                 /* Drat!  They figured it out!
                                      * Let me know how they did it. */
```

```
user@4865b4f533e3:~/exs7/bomb510$ gdb bomb
```

```
(gdb) b phase_1
Breakpoint 1 at 0x15e7
(gdb) 
```

Let me give you an **actual** preview, and helpful tips for the lab to get started

- Then start assembly lay

```
(gdb) layout asm
```

```
0x1489 <main>          endbr64
0x148d <main+4>        push   %rbx
0x148e <main+5>        cmp    $0x1,%edi
0x1491 <main+8>        je     0x158f <main+262>
0x1497 <main+14>       mov    %rsi,%rbx
0x149a <main+17>       cmp    $0x2,%edi
0x149d <main+20>       jne    0x15c4 <main+315>
0x14a3 <main+26>       mov    0x8(%rsi),%rdi
0x14a7 <main+30>       lea    0x1b56(%rip),%rsi        # 0x3004
0x14ae <main+37>       call   0x1320 <fopen@plt>
0x14b3 <main+42>       mov    %rax,0x41f6(%rip)        # 0x56b0 <infile>
0x14ba <main+49>       test   %rax,%rax
0x14bd <main+52>       je     0x15a2 <main+281>
0x14c3 <main+58>       call   0x1b40 <initialize_bomb>
0x14c8 <main+63>       lea    0x1bb9(%rip),%rdi        # 0x3088
0x14cf <main+70>       call   0x1220 <puts@plt>
0x14d4 <main+75>       lea    0x1bed(%rip),%rdi        # 0x30c8
0x14db <main+82>       call   0x1220 <puts@plt>
0x14e0 <main+87>       call   0x1de2 <read_line>
0x14e5 <main+92>       mov    %rax,%rdi
0x14e8 <main+95>       call   0x15e7 <phase_1>
0x14ed <main+100>      call   0x1f1a <phase_defused>
0x14f2 <main+105>      lea    0x1bff(%rip),%rdi        # 0x30f8
0x14f9 <main+112>      call   0x1220 <puts@plt>
0x14fe <main+117>      call   0x1de2 <read_line>
0x1503 <main+122>      mov    %rax,%rdi
0x1506 <main+125>      call   0x160b <phase_2>
0x150b <main+130>      call   0x1f1a <phase_defused>
0x1510 <main+135>      lea    0x1b26(%rip),%rdi        # 0x303d
0x1517 <main+142>      call   0x1220 <puts@plt>
0x151c <main+147>      call   0x1de2 <read_line>

exec No process In:                                              L??   PC:
(gdb)
```

Let me give you an **actual** preview, and helpful tips for the lab to get started

- Now do “run” but **instead of passing** the string like in my demo, just write the name of your file, in my case “defuse”
- Then step through it like in the demo with “ni”, check registers and memory locations with p/x, x/s etc.

```
(gdb) run defuse
Starting program: /home/user/exs7/bomb510/bomb defuse
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
```

Let me give you an **actual** preview, and helpful tips for the lab to get started

- In case your gdb window freezes, don’t forget to use “Ctrl l” for “control load” to reload the window

Let me give you an **actual** preview, and helpful tips for the lab to get started

- **DO NOT FORGET TO SET YOUR BREAKPOINTS** or your bomb will **ALWAYS** blow up (especially don’t forget to set it the first time you do the lab: you are going to ruin your score)

Let me give you an **actual** preview, and helpful tips for the lab to get started

- **Everyone** has their **personal bomb**: I **stole** a bomb from someone, if you want we can look how to solve the first task together
- **Or** do you want to do it alone?

Submission

See you next week!