Sanitizers and linker tricks

Alexey Veselovsky

A few words about me

~~A few words about me~~
Authority bias

Authority bias is the tendency to attribute greater accuracy to the opinion of an authority figure (unrelated to its content) and be more influenced by that opinion.

~~A few words about me~~
Authority bias

Authority bias is the tendency to attribute greater accuracy to the opinion of an authority figure (unrelated to its content) and be more influenced by that opinion.

Let’s develop it!

Authority bias

Worked in industries like:

- SCADA for natural gas compressors (for natural gas storages). In C++.

Authority bias

Worked in industries like:

- SCADA for natural gas compressors (for natural gas storages). In C++.
- VoIP

Authority bias

Worked in industries like:

- SCADA for natural gas compressors (for natural gas storages). In C++.
- VoIP
- Medtech (realtime patient monitoring, data acquisition and processing)

Authority bias

Worked in industries like:

- SCADA for natural gas compressors (for natural gas storages). In C++.
- VoIP
- Medtech (realtime patient monitoring, data acquisition and processing)
- Self driving harvesters and trains

Authority bias

Worked in industries like:

- SCADA for natural gas compressors (for natural gas storages). In C++.
- VoIP
- Medtech (realtime patient monitoring, data acquisition and processing)
- Self driving harvesters and trains

Correctness is critical in these industries.

Authority bias

Worked in industries like:

- SCADA for natural gas compressors (for natural gas storages). In C++.
- VoIP
- Medtech (realtime patient monitoring, data acquisition and processing)
- Self driving harvesters and trains

Correctness is critical in these industries.

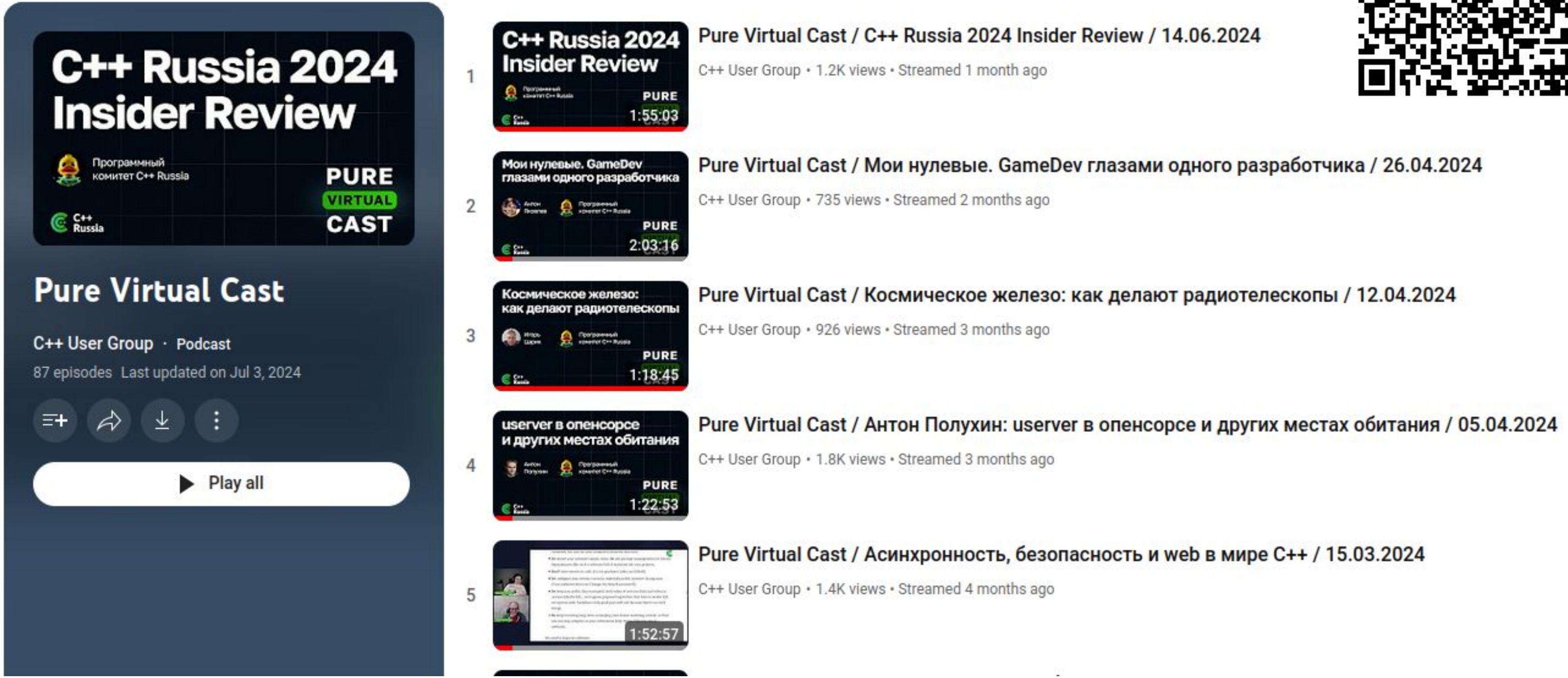
Sanitizers+proper testing are able to help you to catch the most dangerous errors early.

Authority bias

Podcast:

Authority bias

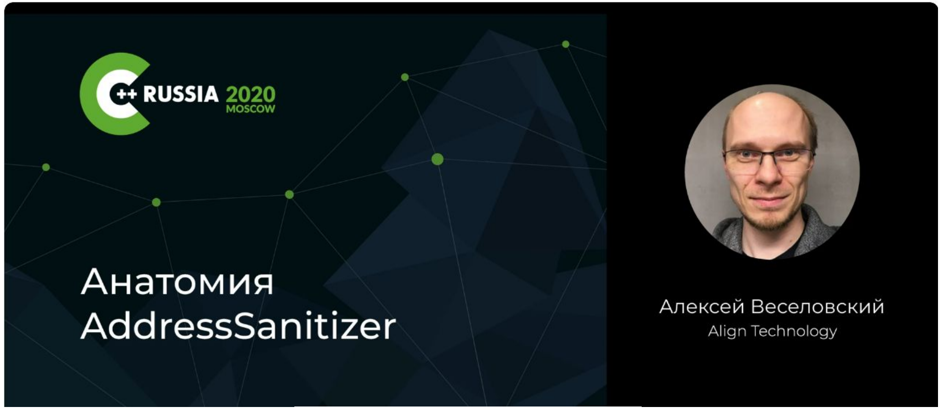
Podcast: Pure Virtual Cast



Authority bias

My talks:

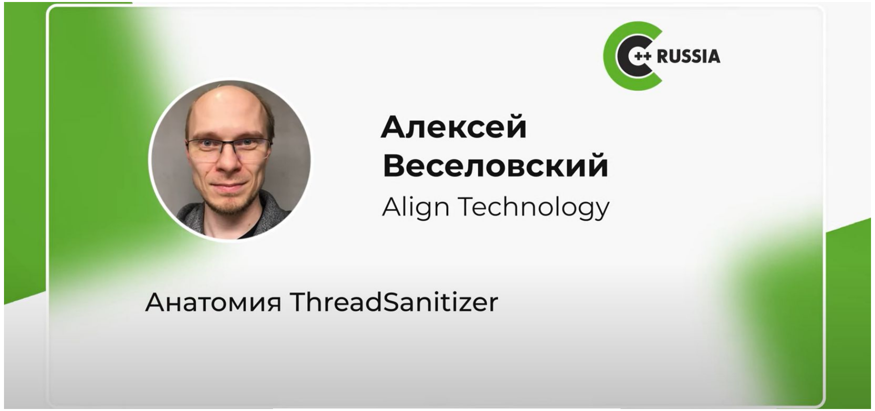
- Address Sanitizer Anatomy (2020)



Authority bias

My talks:

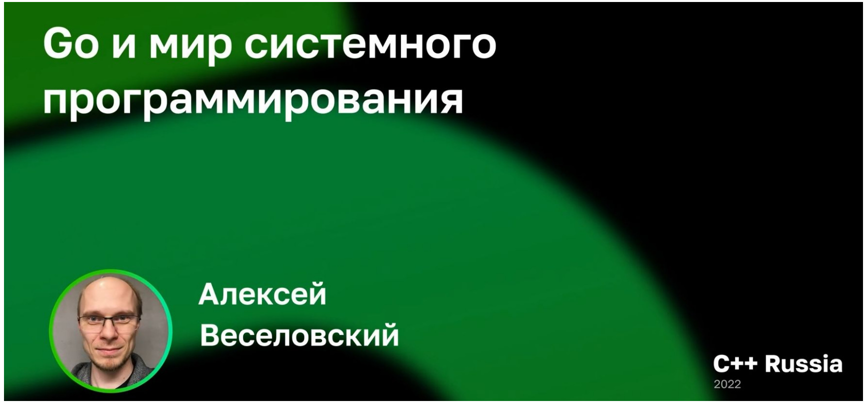
- Address Sanitizer Anatomy (2020)
- Thread Sanitizer Anatomy (2021)



Authority bias

My talks:

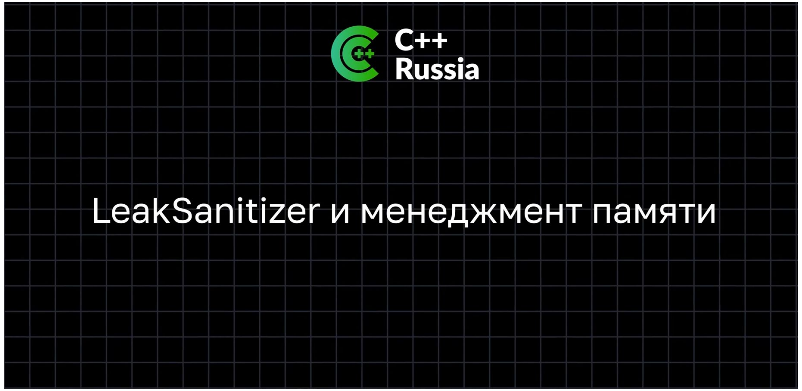
- Address Sanitizer Anatomy (2020)
- Thread Sanitizer Anatomy (2021)
- Go & world of system programming. Runtimeless Go. (2022)



Authority bias

My talks:

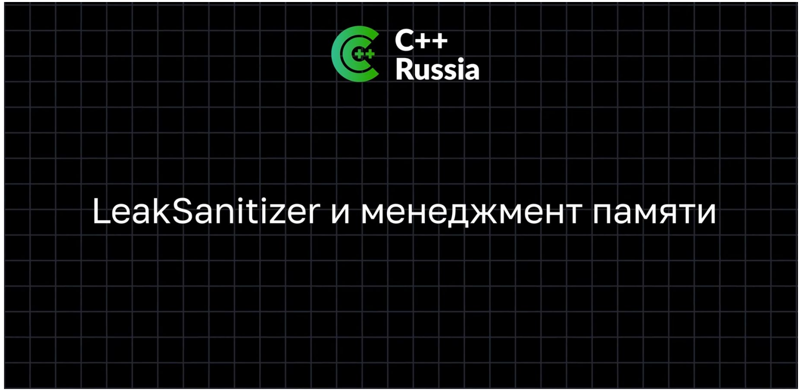
- Address Sanitizer Anatomy (2020)
- Thread Sanitizer Anatomy (2021)
- Go & world of system programming. Runtimeless Go. (2022)
- Leak Sanitizer & memory management (2024)



Sanitizer bias

My talks:

- **Address Sanitizer** Anatomy (2020)
- **Thread Sanitizer** Anatomy (2021)
- Go & world of system programming. Runtimeless Go. (2022)
- **Leak Sanitizer** & memory management (2024)



Sanitizers

- **Address Sanitizer**
- **Thread Sanitizer**

- **Leak Sanitizer**

Sanitizers

- **Address Sanitizer**
- **Thread Sanitizer**
- **Leak Sanitizer**

- Memory Sanitizer
- Undefined Behavior Sanitizer
- Data Flow Sanitizer

Interceptors

- **Address Sanitizer**
- **Thread Sanitizer**
- **Leak Sanitizer**

- **Memory Sanitizer**
- Undefined Behavior Sanitizer
- **Data Flow Sanitizer**

Interceptors inside

Interceptors. Why?

Let’s start from example…

Interceptors. Why?

Let’s start from example… ASAN

Interceptors. Why?

Let’s start from example… ASAN

```
int main() {
   char *arr = malloc(16);
   arr[16] = 16;
   free(arr);
}
```

Interceptors. Why?

Let’s start from example… ASAN

```
int main() {
>   char *arr = malloc(16);
    arr[16] = 16;
    free(arr);
}
```

Interceptors. Why?

```
int main() {
>  char *arr = malloc(16);
   arr[16] = 16;
   free(arr);
}
```

[-8;-1]	[0:7]	[8:15]	[16:23]

ASAN

1. Allocates >16 bytes

Interceptors. Why?

```
int main() {
>  char *arr = malloc(16);
   arr[16] = 16;
   free(arr);
}
```

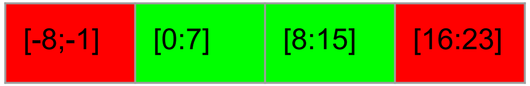
[-8;-1]	[0:7]	[8:15]	[16:23]

ASAN

1. Allocates >16 bytes
2. Saves metadata (stacktrace…)

Interceptors. Why?

```c
int main() {
  char *arr = malloc(16);
  arr[16] = 16;
  free(arr);
}
```

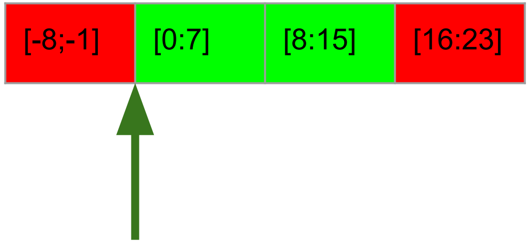


ASAN

1. Allocates >16 bytes
2. Saves metadata (stacktrace…)
3. Adds redzones

Interceptors. Why?

```
int main() {
>  char *arr = malloc(16);
   arr[16] = 16;
   free(arr);
}
```

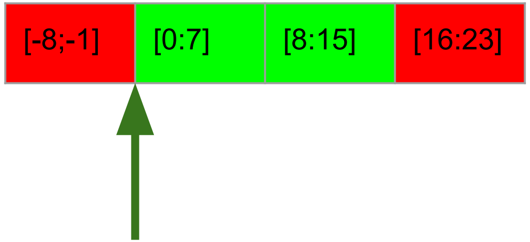


ASAN

1. Allocates >16 bytes
2. Saves metadata (stacktrace…)
3. Adds redzones
4. Returns pointer to user memory

Interceptors. Why?

```
int main() {
>  char *arr = malloc(16);
   arr[16] = 16;
   free(arr);
}
```



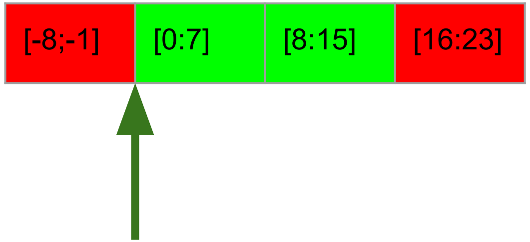
ASAN

1. Allocates >16 bytes
2. Saves metadata (stacktrace…)
3. Adds redzones
4. Returns pointer to user memory

Somehow it should replace malloc function

Interceptors. Why?

```
int main() {
>  char *arr = malloc(16);
   arr[16] = 16;
   free(arr);
}
```



ASAN

1. Allocates >16 bytes
2. Saves metadata (stacktrace…)
3. Adds redzones
4. Returns pointer to user memory

Somehow it should replace malloc function

This technique called **Interceptor**

Interceptors. Why?

Sanitizers intercepts not only functions which they are replacing…

Interceptors. Why?

Sanitizers intercepts not only functions which they are replacing…

But also which they are wrapping (pthread_create, strcpy…).

Interceptors. Why?

Sanitizers intercepts not only functions which they are replacing…

But also which they are wrapping (pthread_create, strcpy…).

```
ReturnType interceptor_for_Func(ArgType arg) {
   // do some sanitizer specific things
    ...
    return REAL(Func)(arg);
}
```

Interceptors. Requirements.

Interceptors. Requirements.

1. Able to replace some library function by our implementation

Interceptors. Requirements.

1. Able to replace some library function by our implementation
2. Able to call original function from our implementation

Interceptors. Requirements.

1. Able to replace some library function by our implementation
2. Able to call original function from our implementation
3. We don’t want to hardcode anything to compiler

Interceptors. How?

1. Able to replace some library function by our implementation
2. Able to call original function from our implementation
3. We don’t want to hardcode anything to compiler
4. ???

Interceptors. How?

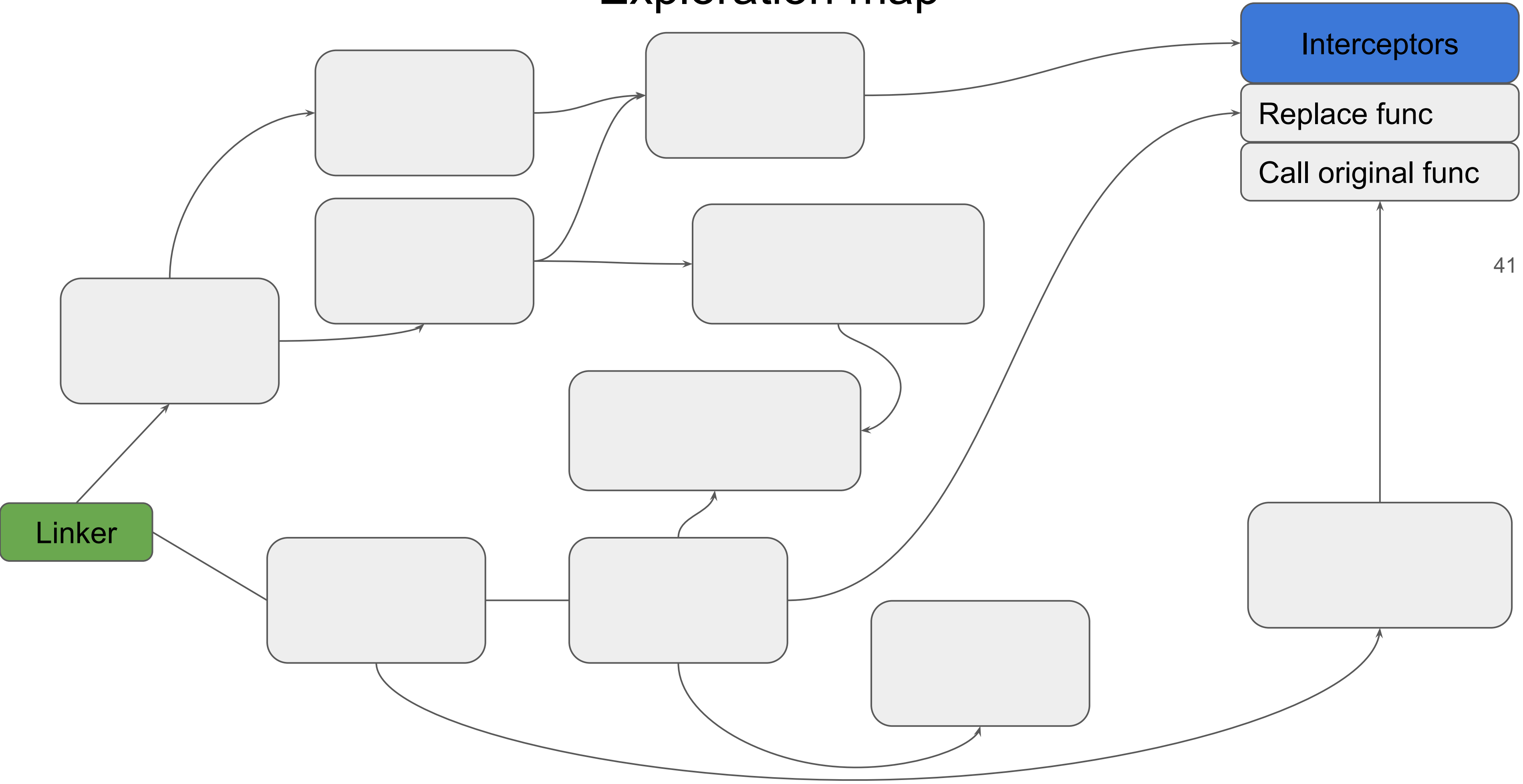
1. Able to replace some library function by our implementation
2. Able to call original function from our implementation
3. We don’t want to hardcode anything to compiler
4. ???
5. **LINKER!**

Interceptors. How?

1. Able to replace some library function by our implementation
2. Able to call original function from our implementation
3. We don’t want to hardcode anything to compiler
4. ???
5. **LINKER!**

Linux x86_64 only

Exploration map



Linker… things

Linker knows not so much about functions. Mainly it works with symbols.

Linker… things

Linker knows not so much about functions. Mainly it works with symbols.

```
/usr/bin/ld: /tmp/ccQVb21X.o: in function `foo()':
bar.cpp:(.text+0x0): multiple definition of `foo()'; /tmp/ccVAAISu.o:m.cpp:(.text+0x0): first
defined here
```

Linker… things

Linker knows not so much about functions. Mainly it works with symbols.

```
/usr/bin/ld: /tmp/ccQVb21X.o: in function `foo()':
bar.cpp:(.text+0x0): multiple definition of `foo()'; /tmp/ccVAAISu.o:m.cpp:(.text+0x0): first
defined here

/usr/bin/ld: /tmp/ccQVb21X.o:(.bss+0x0): multiple definition of `someVar';
/tmp/ccVAAISu.o:(.bss+0x0): first defined here
collect2: error: ld returned 1 exit status
```

Linker… things

Linker knows not so much about functions. Mainly it works with symbols.

```
/usr/bin/ld: /tmp/ccQVb21X.o: in function `foo()':
bar.cpp:(.text+0x0): multiple definition of `foo()'; /tmp/ccVAAISu.o:m.cpp:(.text+0x0): first
defined here

/usr/bin/ld: /tmp/ccQVb21X.o:(.bss+0x0): multiple definition of `someVar';
/tmp/ccVAAISu.o:(.bss+0x0): first defined here
collect2: error: ld returned 1 exit status

stderr: ld.lld: error: undefined symbol: foo
```

Linker ... things

Lin

```
/usr/bin
bar.cpp:                                          0): first
defined

/usr/bin
/tmp/ccV
collect2

stderr:
```

Linker: Sections and Symbols

Object file is list of headers and sections

Linker: Sections and Symbols

Object file is a list of headers and sections
Object file is...

Linker: Sections and Symbols

Object file is a list of headers and sections

Object file is…

compiled CU

```
$ cat m.c
void foo() {}
$ clang -c m.c
$ ls
m.c m.o
```

Linker: Sections and Symbols

Object file is a list of headers and sections

Object file is…

compiled CU

linked shared object

```
$ cat m.c
void foo() {}
$ clang -c m.c
$ ls
m.c m.o

$ clang -shared m.c -o m.so
$ ls
m.so
```

Linker: Sections and Symbols

Object file is a list of headers and sections

Object file is…

compiled CU

```
$ cat m.c
void foo() {}
$ clang -c m.c
$ ls
m.c m.o
```

linked shared object

```
$ clang -shared m.c -o m.so
$ ls
m.so
```

linked executable

```
$ cat m.c
void main() {}
$ clang m.c
$ ls
a.out
```

Linker: Sections and Symbols

Object file is a list of headers and sections: let’s look inside…

```
$ readelf -aW m.o
```

Linker: Sections and Symbols

There are a lot of sections here

```
$ readelf -aW m.o
ELF Header:
…
Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000000000 000040 00000b 00  AX  0   0  1
  [ 2] .data             PROGBITS        0000000000000000 00004b 000000 00  WA  0   0  1
  [ 3] .bss              NOBITS          0000000000000000 00004b 000000 00  WA  0   0  1
  [ 4] .comment          PROGBITS        0000000000000000 00004b 000027 01  MS  0   0  1
  [ 5] .note.GNU-stack   PROGBITS        0000000000000000 000072 000000 00      0   0  1
  [ 6] .note.gnu.property NOTE            0000000000000000 000078 000020 00   A  0   0  8
  [ 7] .eh_frame         PROGBITS        0000000000000000 000098 000038 00   A  0   0  8
  [ 8] .rela.eh_frame    RELA            0000000000000000 000140 000018 18   I  9   7  8
  [ 9] .symtab           SYMTAB          0000000000000000 0000d0 000060 18     10   3  8
  [10] .strtab           STRTAB          0000000000000000 000130 000009 00      0   0  1
  [11] .shstrtab         STRTAB          0000000000000000 000158 000067 00      0   0  1
```

Linker: Sections and Symbols

But we'll consider only few of them: .symtab

```
$ readelf -aW m.o
ELF Header:
…
Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000000000 000040 00000b 00  AX  0   0  1
  [ 2] .data             PROGBITS        0000000000000000 00004b 000000 00  WA  0   0  1
  [ 3] .bss              NOBITS          0000000000000000 00004b 000000 00  WA  0   0  1
  [ 4] .comment          PROGBITS        0000000000000000 00004b 000027 01  MS  0   0  1
  [ 5] .note.GNU-stack   PROGBITS        0000000000000000 000072 000000 00      0   0  1
  [ 6] .note.gnu.property NOTE            0000000000000000 000078 000020 00   A  0   0  8
  [ 7] .eh_frame         PROGBITS        0000000000000000 000098 000038 00   A  0   0  8
  [ 8] .rela.eh_frame    RELA            0000000000000000 000140 000018 18   I  9   7  8
  [ 9] .symtab           SYMTAB          0000000000000000 0000d0 000060 18     10   3  8
  [10] .strtab           STRTAB          0000000000000000 000130 000009 00      0   0  1
  [11] .shstrtab         STRTAB          0000000000000000 000158 000067 00      0   0  1
```

Linker: Sections and Symbols

But we'll consider only few of them: .symtab, .text

```
$ readelf -aW m.o
ELF Header:
…
Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000000000 000040 00000b 00  AX  0   0  1
  [ 2] .data             PROGBITS        0000000000000000 00004b 000000 00  WA  0   0  1
  [ 3] .bss              NOBITS          0000000000000000 00004b 000000 00  WA  0   0  1
  [ 4] .comment          PROGBITS        0000000000000000 00004b 000027 01  MS  0   0  1
  [ 5] .note.GNU-stack   PROGBITS        0000000000000000 000072 000000 00      0   0  1
  [ 6] .note.gnu.property NOTE            0000000000000000 000078 000020 00   A  0   0  8
  [ 7] .eh_frame         PROGBITS        0000000000000000 000098 000038 00   A  0   0  8
  [ 8] .rela.eh_frame    RELA            0000000000000000 000140 000018 18   I  9   7  8
  [ 9] .symtab           SYMTAB          0000000000000000 0000d0 000060 18     10   3  8
  [10] .strtab           STRTAB          0000000000000000 000130 000009 00      0   0  1
  [11] .shstrtab         STRTAB          0000000000000000 000158 000067 00      0   0  1
```

Linker: Sections and Symbols

But we'll consider only few of them: .symtab, .text, .data

```
$ readelf -aW m.o
ELF Header:
…
Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000000000 000040 00000b 00  AX  0   0  1
  [ 2] .data             PROGBITS        0000000000000000 00004b 000000 00  WA  0   0  1
  [ 3] .bss              NOBITS          0000000000000000 00004b 000000 00  WA  0   0  1
  [ 4] .comment          PROGBITS        0000000000000000 00004b 000027 01  MS  0   0  1
  [ 5] .note.GNU-stack   PROGBITS        0000000000000000 000072 000000 00      0   0  1
  [ 6] .note.gnu.property NOTE            0000000000000000 000078 000020 00   A  0   0  8
  [ 7] .eh_frame         PROGBITS        0000000000000000 000098 000038 00   A  0   0  8
  [ 8] .rela.eh_frame    RELA            0000000000000000 000140 000018 18   I  9   7  8
  [ 9] .symtab           SYMTAB          0000000000000000 0000d0 000060 18     10   3  8
  [10] .strtab           STRTAB          0000000000000000 000130 000009 00      0   0  1
  [11] .shstrtab         STRTAB          0000000000000000 000158 000067 00      0   0  1
```

Linker: Sections and Symbols

A simple example: 2 funcs, 2 vars

```
$ cat m.c
int global_var = 42;
static int static_var = 42;

void global_func() {}
static void static_func(){}
```

Linker: Sections and Symbols

Compile…

```
$ cat m.c
int global_var = 42;
static int static_var = 42;

void global_func() {}
static void static_func(){}


$ clang -c m.c
```

Linker: Sections and Symbols

Explore

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

Linker: Sections and Symbols

Each symbol has a name

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND 
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

Linker: Sections and Symbols

And a value

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

Linker: Sections and Symbols

Two rows has the same value. Why?

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

Linker: Sections and Symbols

Two rows has the same value. Why? They are located in the different sections!

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

```
$ readelf -aW m.o
ELF Header:
...
Section Headers:
  [Nr] Name
  [ 0]
  [ 1] .text
  [ 2] .data
  [ 3] .bss
  [ 4] .comment
  [ 5] .note.GNU-stack
  [ 6] .note.gnu.property
  [ 7] .eh_frame
  [ 8] .rela.eh_frame
  [ 9] .symtab
  [10] .strtab
  [11] .shstrtab
```

Linker: Sections and Symbols

Function in a .text section

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

```
$ readelf -aW m.o
ELF Header:
...
Section Headers:
  [Nr] Name
  [ 0]
  [ 1] .text
  [ 2] .data
  [ 3] .bss
  [ 4] .comment
  [ 5] .note.GNU-stack
  [ 6] .note.gnu.property
  [ 7] .eh_frame
  [ 8] .rela.eh_frame
  [ 9] .symtab
  [10] .strtab
  [11] .shstrtab
```

Linker: Sections and Symbols

Variable in a .data section

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

```
$ readelf -aW m.o
ELF Header:
...
Section Headers:
  [Nr] Name
  [ 0]
  [ 1] .text
  [ 2] .data
  [ 3] .bss
  [ 4] .comment
  [ 5] .note.GNU-stack
  [ 6] .note.gnu.property
  [ 7] .eh_frame
  [ 8] .rela.eh_frame
  [ 9] .symtab
  [10] .strtab
  [11] .shstrtab
```

Linker: Sections and Symbols

Two functions

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

```
$ readelf -aW m.o
ELF Header:
...
Section Headers:
  [Nr] Name
  [ 0]
  [ 1] .text
  [ 2] .data
  [ 3] .bss
  [ 4] .comment
  [ 5] .note.GNU-stack
  [ 6] .note.gnu.property
  [ 7] .eh_frame
  [ 8] .rela.eh_frame
  [ 9] .symtab
  [10] .strtab
  [11] .shstrtab
```

Linker: Sections and Symbols

Two functions with a different bindings

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

```
$ readelf -aW m.o
ELF Header:
...
Section Headers:
  [Nr] Name
  [ 0]
  [ 1] .text
  [ 2] .data
  [ 3] .bss
  [ 4] .comment
  [ 5] .note.GNU-stack
  [ 6] .note.gnu.property
  [ 7] .eh_frame
  [ 8] .rela.eh_frame
  [ 9] .symtab
  [10] .strtab
  [11] .shstrtab
```

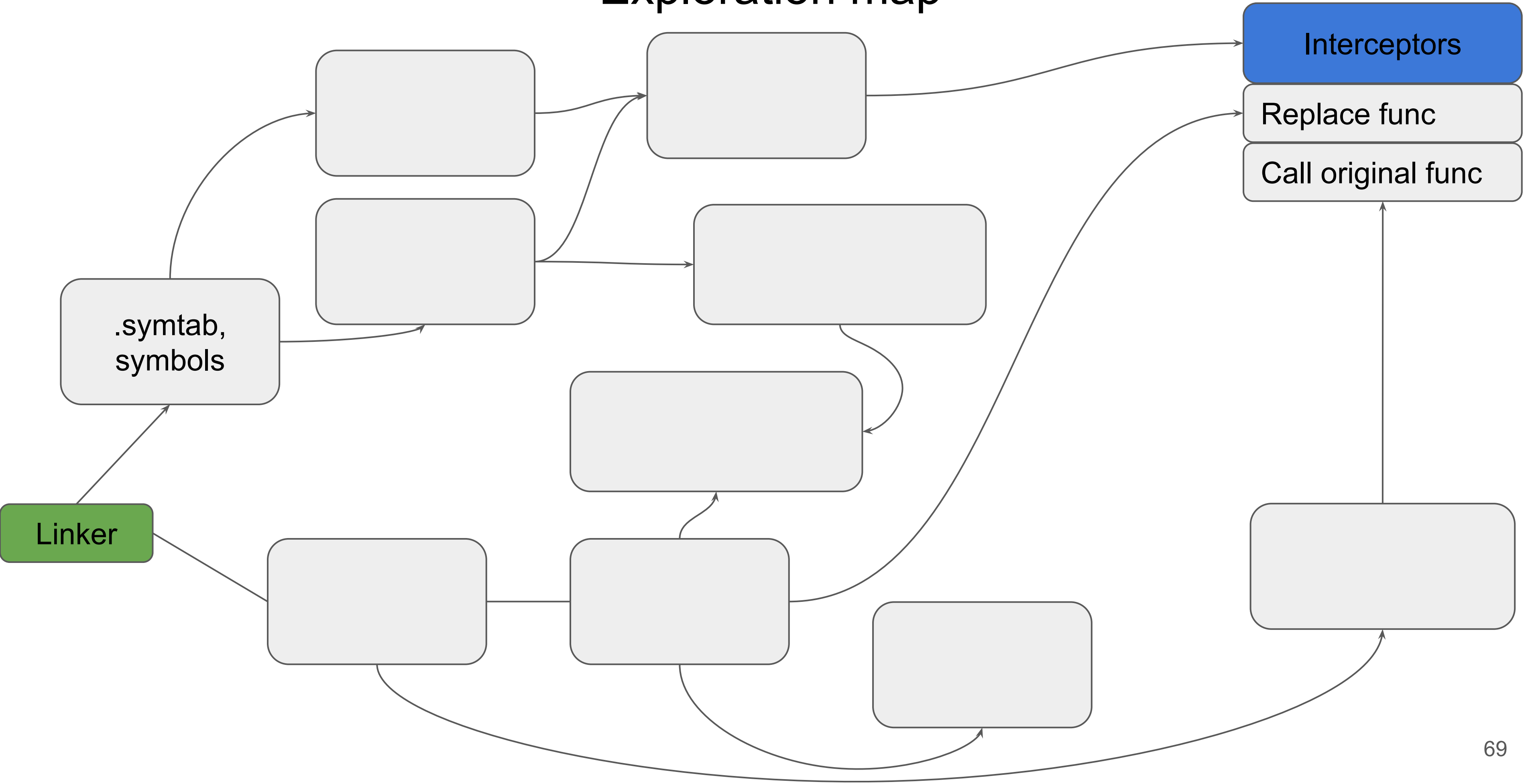
Linker: Sections and Symbols

Values are the different

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

```
$ readelf -aW m.o
ELF Header:
...
Section Headers:
  [Nr] Name
  [ 0]
  [ 1] .text
  [ 2] .data
  [ 3] .bss
  [ 4] .comment
  [ 5] .note.GNU-stack
  [ 6] .note.gnu.property
  [ 7] .eh_frame
  [ 8] .rela.eh_frame
  [ 9] .symtab
  [10] .strtab
  [11] .shstrtab
```

Exploration map



Linker: Sections and Symbols

What we can do with this table?

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

Linker: Sections and Symbols

Two (or multiple) rows with same value but different names

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

Linker: Sections and Symbols

Two (or multiple) rows with same value but different names

```
$ readelf -sW m.o

Symbol table '.symtab' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
     7: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func_2
```

Linker: Sections and Symbols

Let’s do it in code

```
$
```

Linker: Sections and Symbols

Let’s do it in code. It is not C or C++! It is **extension**.

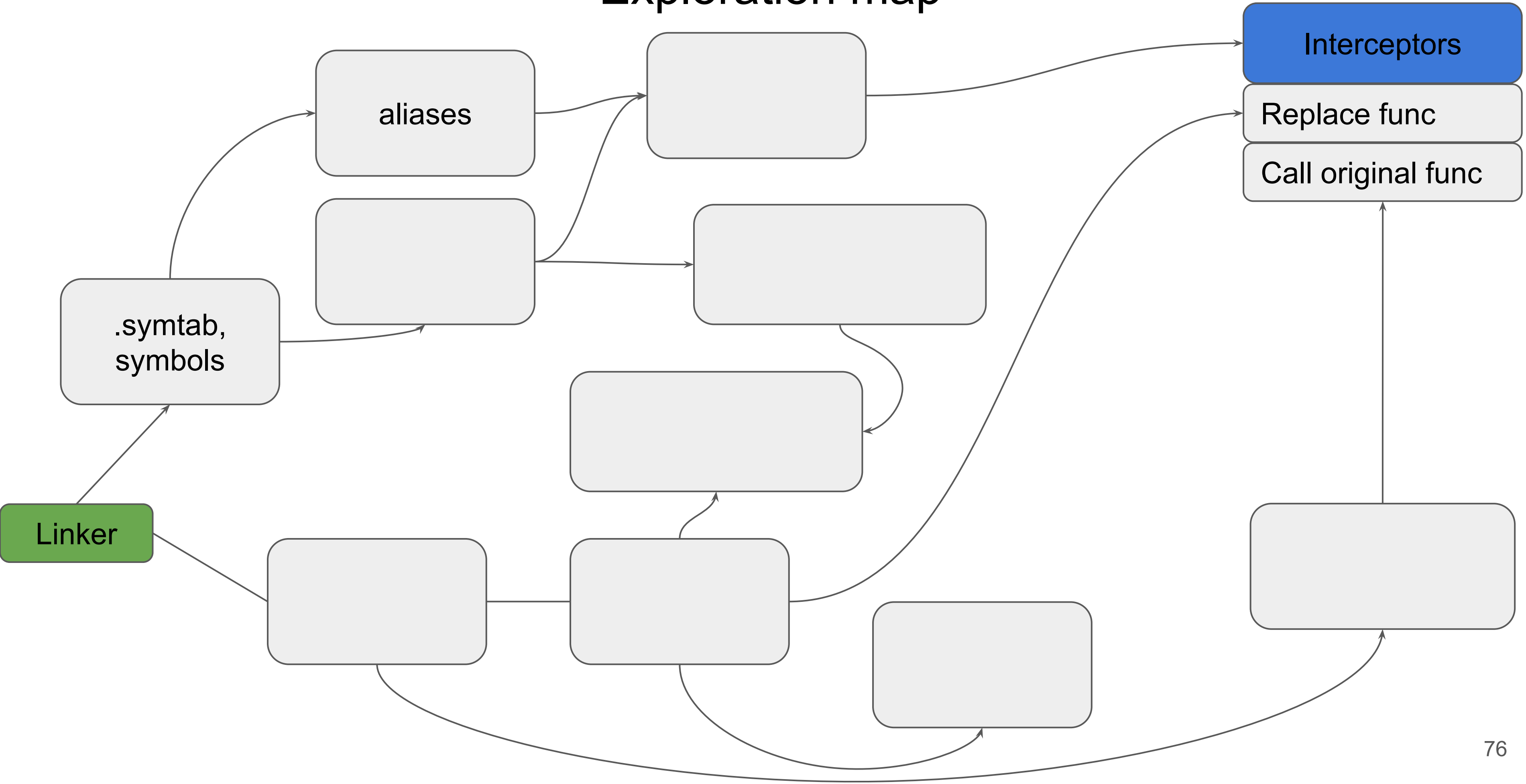
```
$ cat m.c
int global_var = 42;
static int static_var = 42;

void global_func() {}
void __attribute__((alias("global_func"))) global_func_2();

static void static_func(){}


$ clang -c m.c
```

Linker: Sections and Symbols

Result:

```
$ readelf -sW m.o

Symbol table '.symtab' contains 8 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND 
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    2 static_var
     4: 000000000000000b    11 FUNC    LOCAL  DEFAULT    1 static_func
     5: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    2 global_var
     6: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
     7: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func_2
```

Exploration map



Linker: Sections and Symbols

What about two values one name?

```
$
```

Linker: Sections and Symbols

What about two values one name?
Compiler don’t let us to define it in one compilation unit

```
$
```

Linker: Sections and Symbols

What about two values one name?
Compiler don’t let us to define it in one compilation unit
But in multiple…

```
$
```

Linker: Sections and Symbols

During linking linker merges .symtabs from multiple object files to one.

```
$
```

Linker: Sections and Symbols

During linking linker merges .symtabs from multiple object files to one.
If there are two symbols with the same name generally it is an error:

```
$ cat m.c
void global_func() {}
$ clang -c m.c
```

Linker: Sections and Symbols

During linking linker merges .symtabs from multiple object files to one.
If there are two symbols with the same name generally it is an error:

```
$ cat m.c
void global_func() {}
$ clang -c m.c

$ cat main.c
void global_func() {}
void main() {}
$ clang -c main.c
```

Linker: Sections and Symbols

During linking linker merges .symtabs from multiple object files to one.
If there are two symbols with the same name generally it is an error:

```
$ cat m.c
void global_func() {}
$ clang -c m.c

$ cat main.c
void global_func() {}
void main() {}
$ clang -c main.c
```

```
$ clang m.o main.o
/usr/bin/ld: main.o: in function `global_func':
main.c:(.text+0x0): multiple definition of `global_func';
m.o:m.c:(.text+0x0): first defined here
collect2: error: ld returned 1 exit status
```

Linker: Sections and Symbols

During linking linker merges .symtabs from multiple object files to one.
If there are two symbols with the same name generally it is an error:

```
main.c:(.text+0x0): multiple definition of `global_func';
m.o:m.c:(.text+0x0): first defined here
```

```
$ cat m.c
void global_func() {}
$ clang -c m.c

$ cat main.c
void global_func() {}
void main() {}
$ clang -c main.c
```

```
$ readelf -sW m.o
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
```

Linker: Sections and Symbols

During linking linker merges .symtabs from multiple object files to one.
If there are two symbols with the same name generally it is an error:

```
main.c:(.text+0x0): multiple definition of `global_func';
m.o:m.c:(.text+0x0): first defined here
```

```
$ cat m.c
void global_func() {}
$ clang -c m.c

$ cat main.c
void global_func() {}
void main() {}
$ clang -c main.c
```

```
$ readelf -sW m.o
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
$ readelf -sW main.o
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
     4: 000000000000000b    11 FUNC    GLOBAL DEFAULT    1 main
```

Linker: Sections and Symbols

But if we'll define function with **weak** attribute…

```
$ cat m.c
void __attribute__((weak))
global_func() {}
$ clang -c m.c

$ cat main.c
void global_func() {}
void main() {}
$ clang -c main.c
```

Linker: Sections and Symbols

But if we'll define function with **weak** attribute…
Linking will be ok

```
$ cat m.c
void __attribute__((weak))
global_func() {}
$ clang -c m.c

$ cat main.c
void global_func() {}
void main() {}
$ clang -c main.c
$ clang m.o main.o
```

```
$ readelf -sW m.o
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000000    11 FUNC    WEAK   DEFAULT    1 global_func
$ readelf -sW main.o
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
     4: 000000000000000b    11 FUNC    GLOBAL DEFAULT    1 main
```

Linker: Sections and Symbols

But if we'll define function with **weak** attribute…
Linking will be ok
Executable also contains .symtab

```
$ cat m.c
void __attribute__((weak))
global_func() {}
$ clang -c m.c

$ cat main.c
void global_func() {}
void main() {}
$ clang -c main.c
$ clang m.o main.o
```

```
$ readelf -sW a.out | grep global_func
36: 0000000000001134    11 FUNC    GLOBAL DEFAULT   14 global_func
```

Linker: Sections and Symbols

What if we'll remove global_func from main.c?

```
$ cat m.c
void __attribute__((weak))
global_func() {}
$ clang -c m.c

$ cat main.c
//void global_func() {}
void main() {}
$ clang -c main.c
$ clang m.o main.o
```

```
$
```

Linker: Sections and Symbols

What if we'll remove global_func from main.c?

Question to audience: will the result be the same?

```
$ cat m.c
void __attribute__((weak))
global_func() {}
$ clang -c m.c

$ cat main.c
//void global_func() {}
void main() {}
$ clang -c main.c
$ clang m.o main.o
```

```
$ readelf -sW a.out | grep global
...
```

Linker: Sections and Symbols

What if we'll remove global_func from main.c?

Question to audience: will the result be the same?

No. Linker really copies symbols as is.

```
$ cat m.c
void __attribute__((weak))
global_func() {}
$ clang -c m.c

$ cat main.c
//void global_func() {}
void main() {}
$ clang -c main.c
$ clang m.o main.o
```

```
$ readelf -sW a.out | grep global
...
36: 0000000000001129    11 FUNC    WEAK   DEFAULT   14 global_func
```

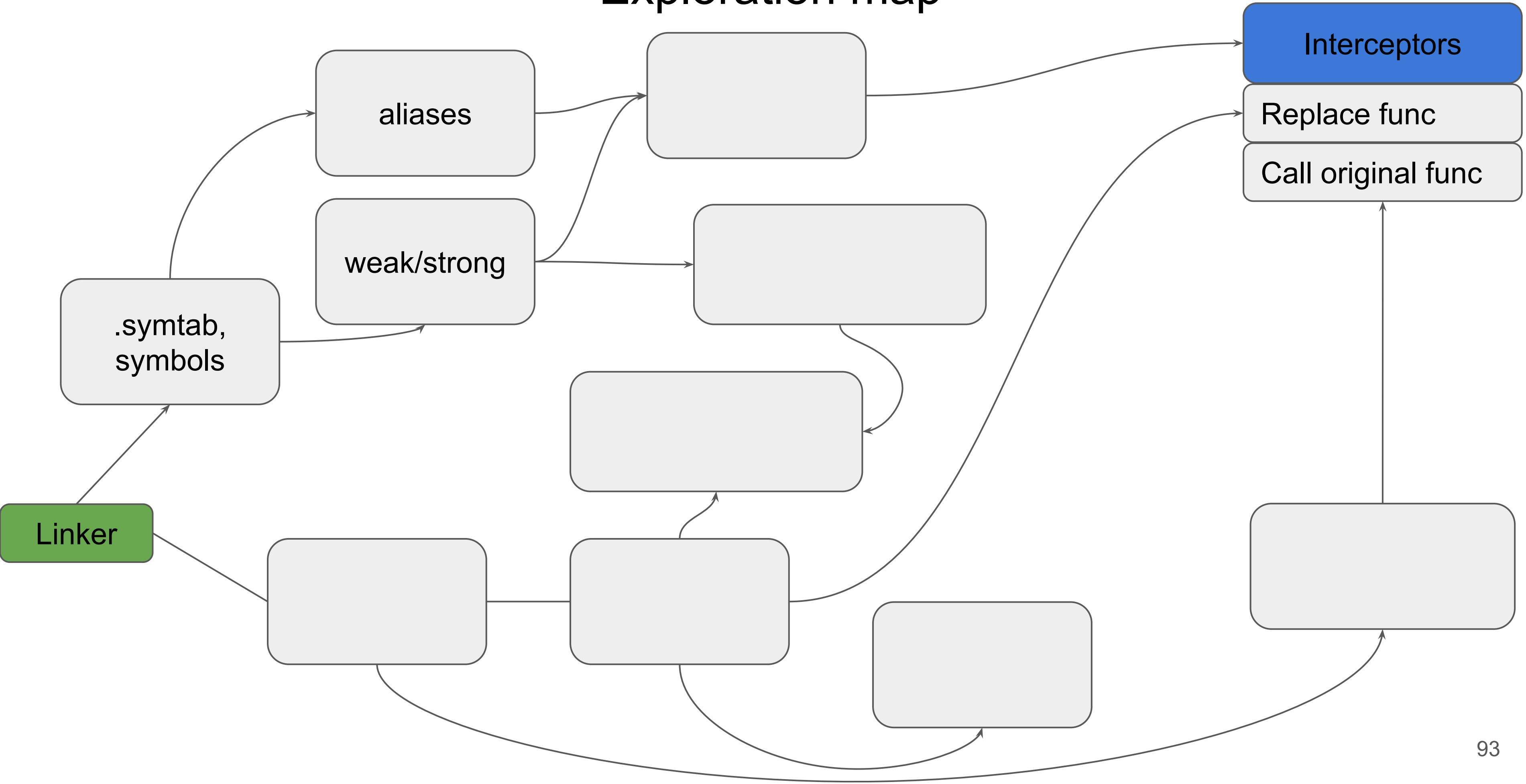
Linker: Sections and Symbols

STRONG (global) symbols always prevail over **WEAK** symbols

```
$ cat m.c
void __attribute__((weak))
global_func() {}
$ clang -c m.c

$ cat main.c
//void global_func() {}
void main() {}
$ clang -c main.c
$ clang m.o main.o
```

```
$ readelf -sW a.out | grep global
...
36: 0000000000001129    11 FUNC    WEAK   DEFAULT   14 global_func
```

Exploration map



Linker: Sections and Symbols

Aliases and Weak symbols are independent. That’s why we can combine them:
Create weak alias to some function.

```
$ cat m.c
void global_func() {}
void __attribute__((weak ,alias("global_func"))) global_func_2();
$ clang -c m.c
```

Linker: Sections and Symbols

Aliases and Weak symbols are independent. That's why we can combine them.
Create weak alias to some function.

```
$ cat m.c
void global_func() {}
void __attribute__((weak, alias("global_func"))) global_func_2();
$ clang -c m.c
$ readelf -sW m.o
Num:    Value          Size Type    Bind   Vis      Ndx Name
  0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
  1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
  2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
  3: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func
  4: 0000000000000000    11 FUNC    WEAK   DEFAULT    1 global_func_2
```

Linker: Sections and Symbols

Aliases and Weak symbols are independent. That's why we can combine them.
Question to the audience:

Can we create a **STRONG** alias to **WEAK** function?

```
$ cat m.c
void __attribute__((weak)) global_func() {}
void __attribute__((alias("global_func"))) global_func_2();
?
```

Linker: Sections and Symbols

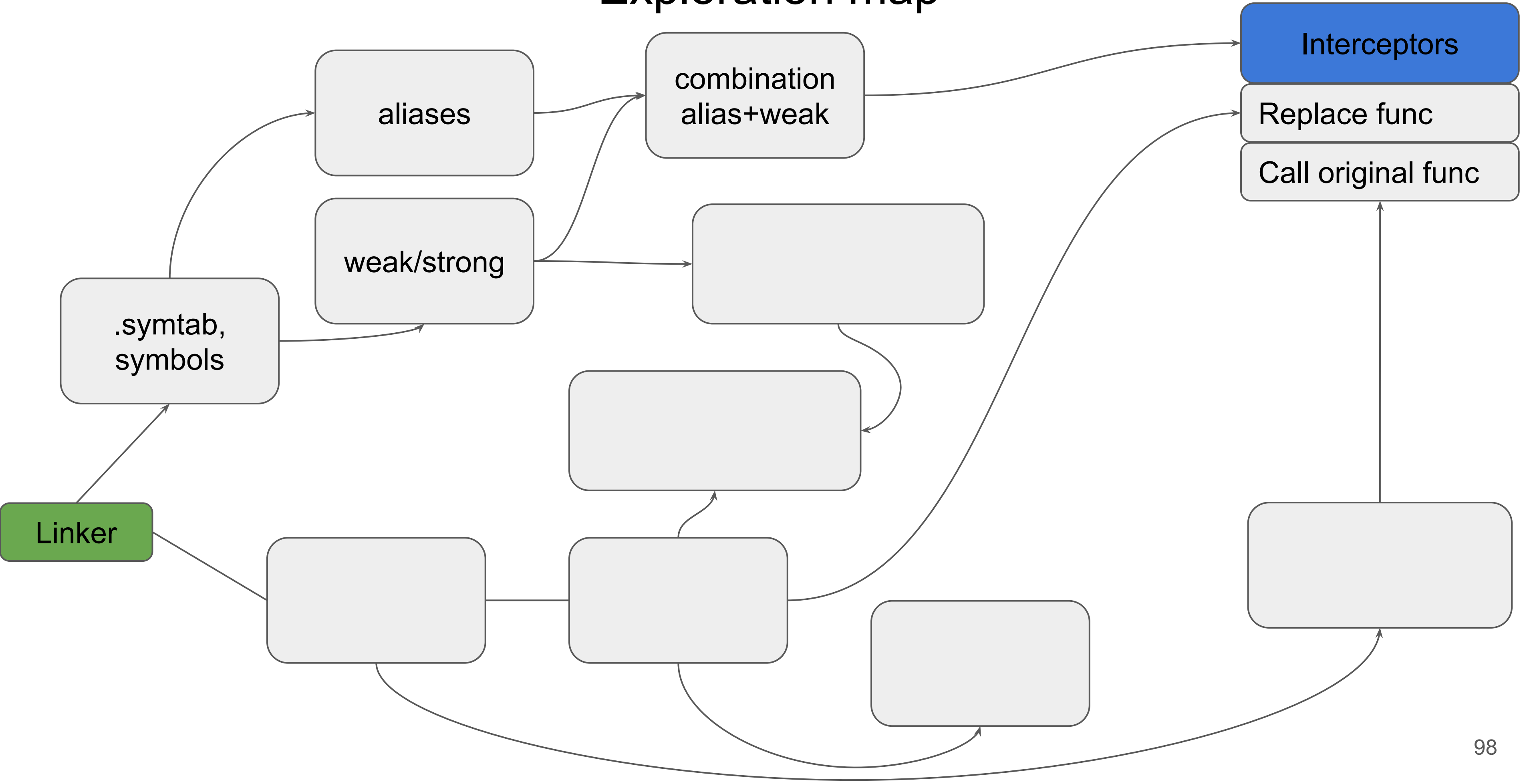
Aliases and Weak symbols are independent. That's why we can combine them.
Question to the audience:

Can we create a **STRONG** alias to **WEAK** function?

Yep. We can! All records in .symtab are equal. There is no primary record.

```
$ cat m.c
void __attribute__((weak)) global_func() {}
void __attribute__((alias("global_func"))) global_func_2();
$ clang -c m.c
$ readelf -sW m.o
Num:    Value            Size Type    Bind   Vis      Ndx Name
  0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
  1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS m.c
  2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
  3: 0000000000000000    11 FUNC    WEAK   DEFAULT    1 global_func
  4: 0000000000000000    11 FUNC    GLOBAL DEFAULT    1 global_func_2
```

Exploration map



Let’s use it!

Let’s use it!

ASAN public API has useful function

Let’s use it!

ASAN public API has useful function

```
// Prints stack traces for all live heap allocations ordered by total
// allocation size until top_percent of total live heap is shown. top_percent
// should be between 1 and 100. At most max_number_of_contexts contexts
// (stack traces) are printed.
// Experimental feature currently available only with ASan on Linux/x86_64.
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts);
```

Let’s use it!

ASAN public API has useful function

```
// Prints stack traces for all live heap allocations ordered by total
// allocation size until top_percent of total live heap is shown. top_percent
// should be between 1 and 100. At most max_number_of_contexts contexts
// (stack traces) are printed.
// Experimental feature currently available only with ASan on Linux/x86_64.
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts);
```

```
Live Heap Allocations: 475809 bytes in 6 chunks; quarantined: 0 bytes in 0 chunks; 17747 other chunks; total chunks:
17753; showing top 100% (at most 1000 unique contexts)

402000 byte(s) (84%) in 1 allocation(s)
    #0 0x6508136538d1 in operator new[](unsigned long) (/tmp/test/a.out+0x1058d1)
    #1 0x650813655b48 in main (/tmp/test/a.out+0x107b48)
    #2 0x7241ba62a1c9 in __libc_start_call_main csu/../sysdeps/nptl/libc_start_call_main.h:58:16
    #3 0x7241ba62a28a in __libc_start_main csu/../csu/libc-start.c:360:3
    #4 0x65081357a344 in _start (/tmp/test/a.out+0x2c344)

73728 byte(s) (15%) in 1 allocation(s)
    #0 0x650813615193 in malloc (/tmp/test/a.out+0xc7193)
…
```

Let’s use it!

ASAN public API has useful function

```
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts);
```

We want call this function if ASAN runtime is available

Let’s use it!

ASAN public API has useful function

```
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts);
```

We want call this function if ASAN runtime is available

We want to avoid recompilation. That’s why we can’t use conditional compilation.

Let’s use it!

Let’s just call it!

```
$ cat main.c
#include <sanitizer/asan_interface.h>
int main() {
   __sanitizer_print_memory_profile(100, 1000);
}
```

Let’s use it!

Let’s just call it!

```
$ cat main.c
#include <sanitizer/asan_interface.h>
int main() {
   __sanitizer_print_memory_profile(100, 1000);
}

$ clang -fsanitize=address main.c
```

Let’s use it!

Compiles!

```
$ cat main.c
#include <sanitizer/asan_interface.h>
int main() {
   __sanitizer_print_memory_profile(100, 1000);
}

$ clang -fsanitize=address main.c
```

Let's use it!

Ooops… Linkage error.

```
$ cat main.c
#include <sanitizer/asan_interface.h>
int main() {
   __sanitizer_print_memory_profile(100, 1000);
}

$ clang -fsanitize=address main.c
$ clang main.c
/usr/bin/ld: /tmp/main-04dc2e.o: in function `main':
main.c:(.text+0xf): undefined reference to `__sanitizer_print_memory_profile'
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

Let’s use it!

Let’s define undefined!

```
$ cat main.c
#include <sanitizer/asan_interface.h>
int main() {
   __sanitizer_print_memory_profile(100, 1000);
}

$ clang -fsanitize=address main.c
$ clang main.c
/usr/bin/ld: /tmp/main-04dc2e.o: in function `main':
main.c:(.text+0xf): undefined reference to `__sanitizer_print_memory_profile'
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

Let’s use it!

Let’s define undefined!

```
$ cat main.c
#include <sanitizer/asan_interface.h>
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}
int main() {
   __sanitizer_print_memory_profile(100, 1000);
}
$
```

Let’s use it!

Let’s define undefined!
Compiles!

```
$ cat main.c
#include <sanitizer/asan_interface.h>
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}
int main() {
   __sanitizer_print_memory_profile(100, 1000);
}
$ clang main.c
$
```

Let’s use it!

Ooops… Multiple definitions. As expected.

```
$ cat main.c
#include <sanitizer/asan_interface.h>
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}
int main() {
   __sanitizer_print_memory_profile(100, 1000);
}
$ clang main.c
$ clang -fsanitize=address main.c
/usr/bin/ld: /tmp/main-acadb5.o: in function `__sanitizer_print_memory_profile':
main.c:(.text+0x0): multiple definition of `__sanitizer_print_memory_profile';
/usr/lib/llvm-18/lib/clang/18/lib/linux/libclang_rt.asan-x86_64.a(asan_memory_profile.cpp.o):(.text.__sanitizer_print_m
emory_profile+0x0): first defined here
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

Let’s use it!

Let’s define undefined as weak symbol.

```
$ cat main.c
#include <sanitizer/asan_interface.h>
void __attribute__((weak))
__sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}

int main() {
   __sanitizer_print_memory_profile(100, 1000);
}

$ clang main.c
```

Let’s use it!

Let’s define undefined as weak symbol.
Links successfully with or without ASAN runtime

```
$ cat main.c
#include <sanitizer/asan_interface.h>
void __attribute__((weak))
__sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}

int main() {
   __sanitizer_print_memory_profile(100, 1000);
}

$ clang main.c
$ clang -fsanitize=address main.c
$
```

Let’s use it!

Links successfully with or without ASAN runtime

Let’s run it!

```
$ cat main.c
#include <sanitizer/asan_interface.h>
void __attribute__((weak))
__sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}

int main() {
   __sanitizer_print_memory_profile(100, 1000);
}

$ clang main.c
$ clang -fsanitize=address main.c
$ ./a.out
```

Let’s use it!

Links successfully with or without ASAN runtime

Let’s run it! Works!

```
$ cat main.c
#include <sanitizer/asan_interface.h>
void __attribute__((weak))
__sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}

int main() {
   __sanitizer_print_memory_profile(100, 1000);
}

$ clang main.c
$ clang -fsanitize=address main.c
$ ./a.out
Live Heap Allocations: 65 bytes in 2 chunks; quarantined: 0 bytes in 0 chunks; 9555 other chunks; total chunks: 9557;
showing top 100% (at most 1000 unique contexts)
41 byte(s) (63%) in 1 allocation(s)
    #0 0x5c7fa0f49193 in malloc (/tmp/test/a.out+0xc6193) (BuildId: 0089915e746c65b3f9e8c42abdd6dc6e56614062)
...
```

Let's use it!

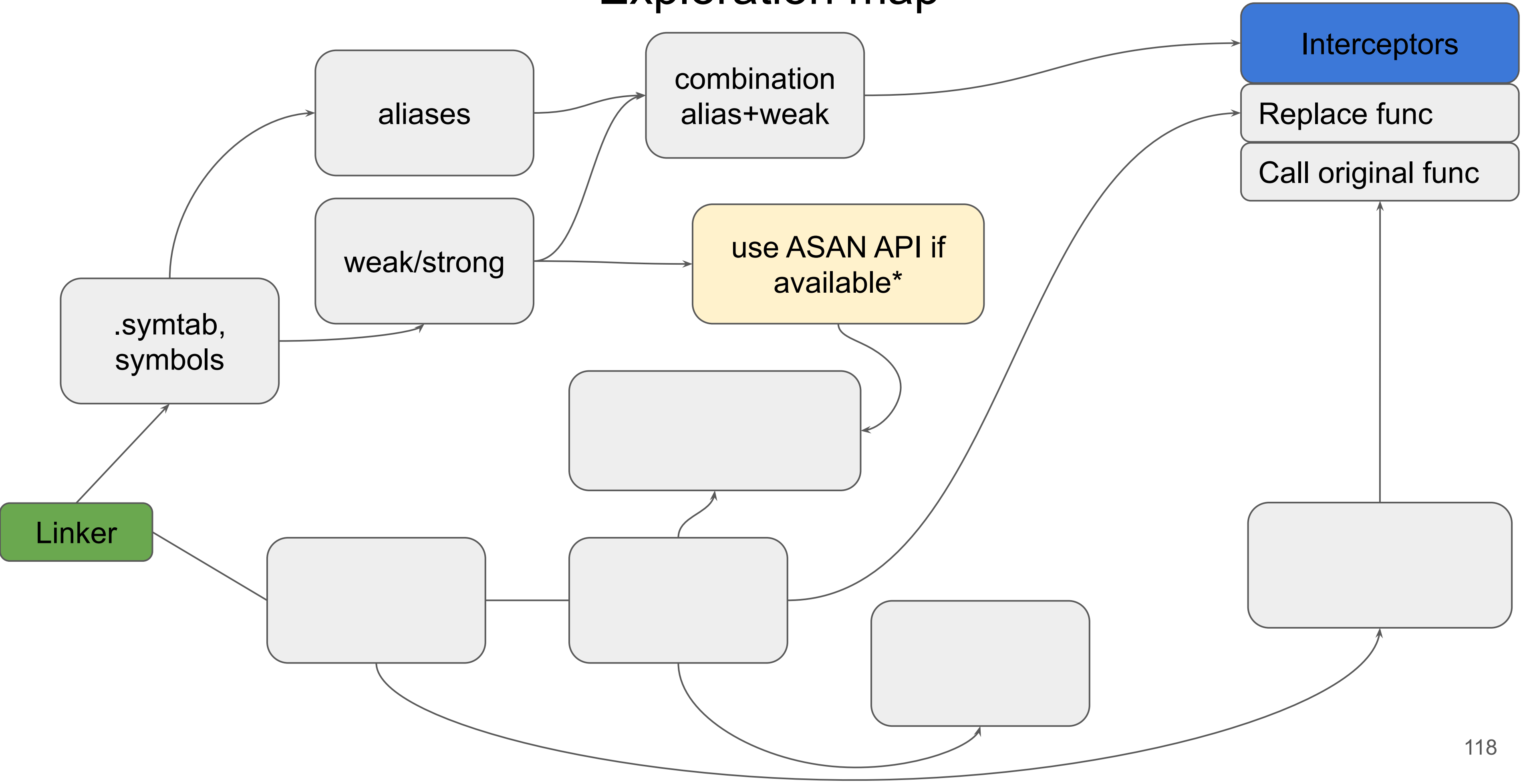
Links successfully with or without ASAN runtime

Let's run it! Works!

```
$ cat main.c
#include <sanitizer/asan_interface.h>
void __attribute__((weak))
__sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}

int main() {
   __sanitizer_print_memory_profile(100, 1000);
}

$ clang main.c
$ clang -fsanitize=address main.c
$ ./a.out
Live Heap Allocations: 65 bytes in 2 chunks; quarantined: 0 bytes in 0 chunks; 9555 other chunks; total chunks: 9557;
showing top 100% (at most 1000 unique contexts)
41 byte(s) (63%) in 1 allocation(s)
    #0 0x5c7fa0f49193 in malloc (/tmp/test/a.out+0xc6193) (BuildId: 0089915e746c65b3f9e8c42abdd6dc6e56614062)
...
```

SUCCESS!

Exploration map



Let's use it!

Let's test it with gcc

```
$ cat main.c
#include <sanitizer/asan_interface.h>
void __attribute__((weak))
__sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}

int main() {
   __sanitizer_print_memory_profile(100, 1000);
}

$ gcc main.c
$
```

Let’s use it!

Let’s test it with gcc
Compiles and links with or without ASAN

```
$ cat main.c
#include <sanitizer/asan_interface.h>
void __attribute__((weak))
__sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}

int main() {
   __sanitizer_print_memory_profile(100, 1000);
}

$ gcc main.c
$ gcc -fsanitize=address main.c
$
```

Let’s use it!

Compiles and links with or without ASAN
Let’s run it!

```
$ cat main.c
#include <sanitizer/asan_interface.h>
void __attribute__((weak))
__sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}

int main() {
   __sanitizer_print_memory_profile(100, 1000);
}

$ gcc main.c
$ gcc -fsanitize=address main.c
$ ./a.out
```

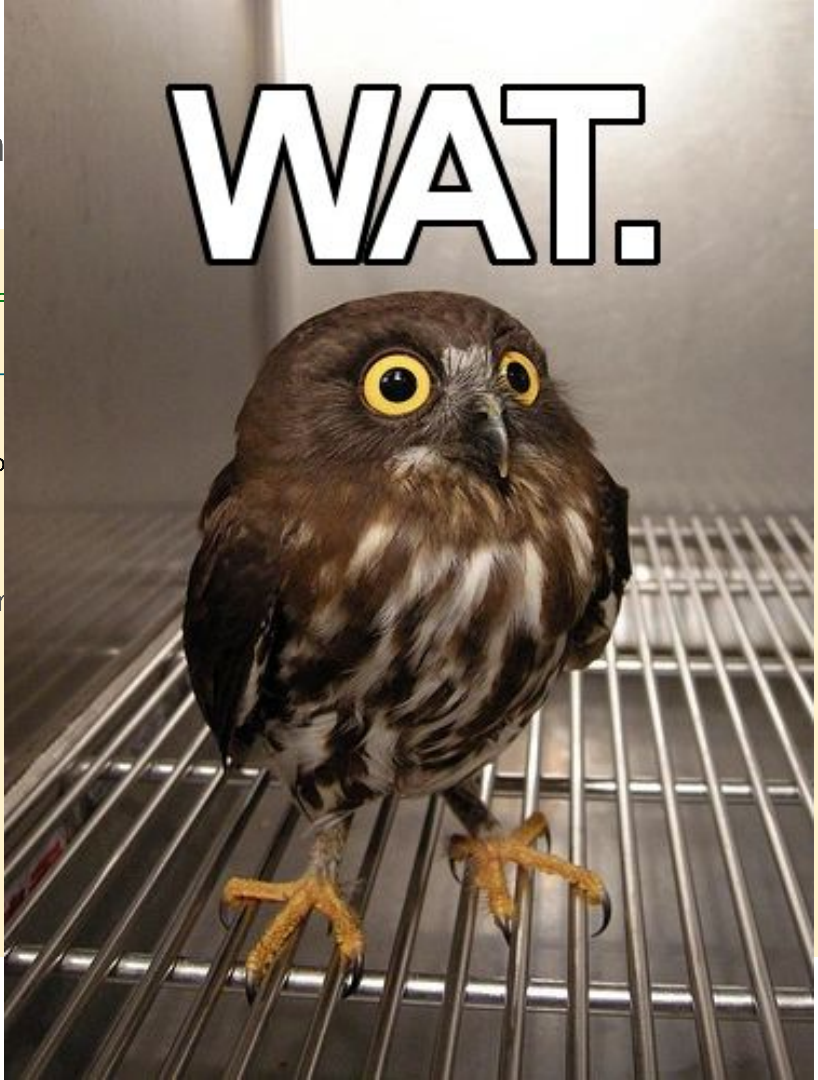
Let’s use it!

Compiles and links with or without ASAN
Let’s run it!

```
$ cat main.c
#include <sanitizer/asan_interface.h>
void __attribute__((weak))
__sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}

int main() {
   __sanitizer_print_memory_profile(100, 1000);
}

$ gcc main.c
$ gcc -fsanitize=address main.c
$ ./a.out
$
```

Compiles and links with
Let’s run it!

```
$ cat main.c
#include <sanitizer/asan_interf
void __attribute__((weak))
__sanitizer_print_memory_profil

int main() {
   __sanitizer_print_memory_pro
}

$ gcc main.c
$ gcc -fsanitize=address m
$ ./a.out
$
```



Let's use it!

clang uses **static** ASAN runtime (by default)
gcc uses **dynamic** ASAN runtime (by default)

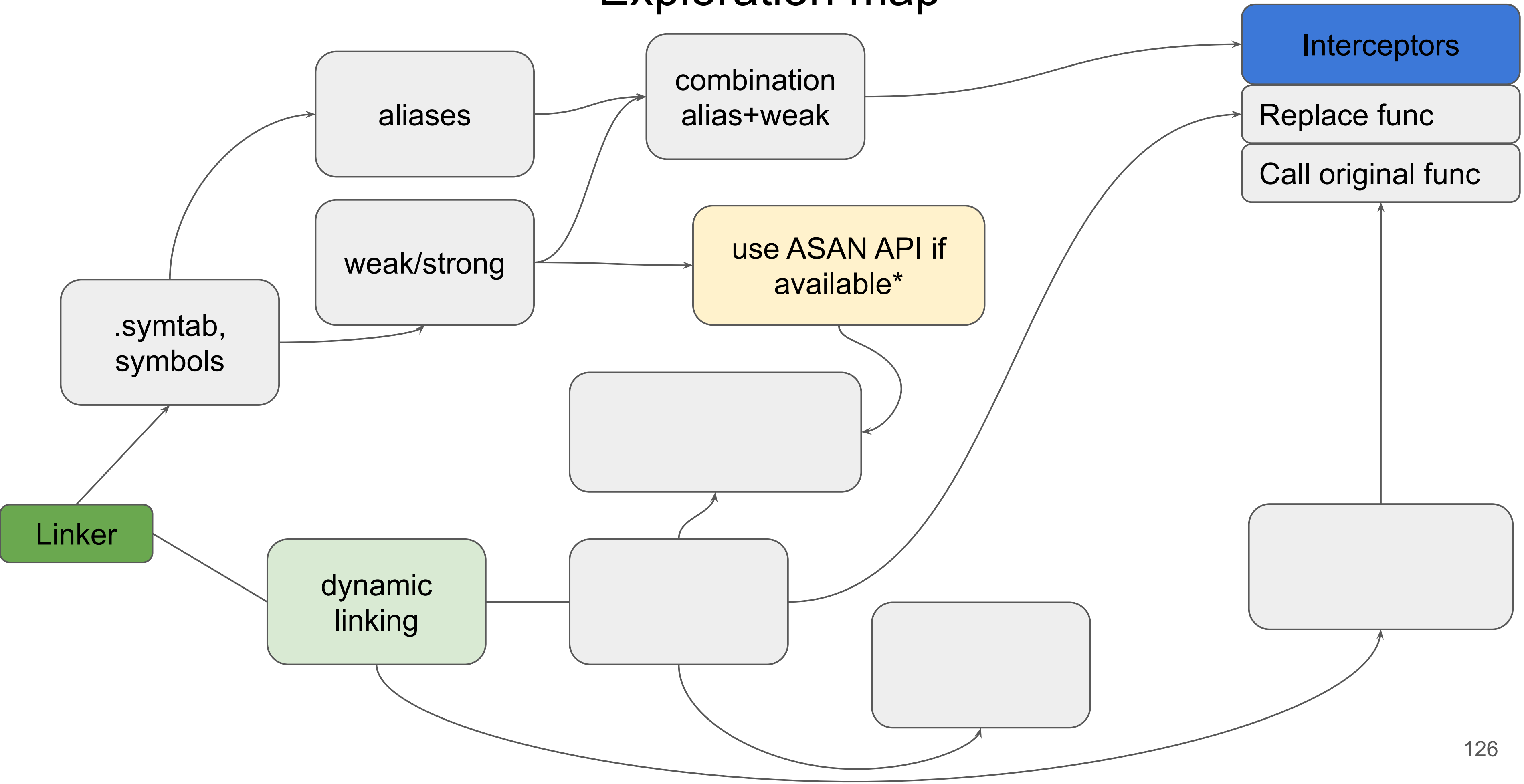
```
$ cat main.c
#include <sanitizer/asan_interface.h>
void __attribute__((weak))
__sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}

int main() {
   __sanitizer_print_memory_profile(100, 1000);
}

$ gcc main.c
$ gcc -fsanitize=address main.c
$ ./a.out
$
```

Dynamic linking

clang uses **static** ASAN runtime (by default)
gcc uses **dynamic** ASAN runtime (by default)

Seems like dynamic linker works differently

Exploration map



Dynamic linking

Seems like dynamic linker works differently

Dynamic linker much more lazier than static one.

Dynamic linking

Seems like dynamic linker works differently

Dynamic linker much more lazier than static one.

Static linker checks all symbols with the same name.

Dynamic linking

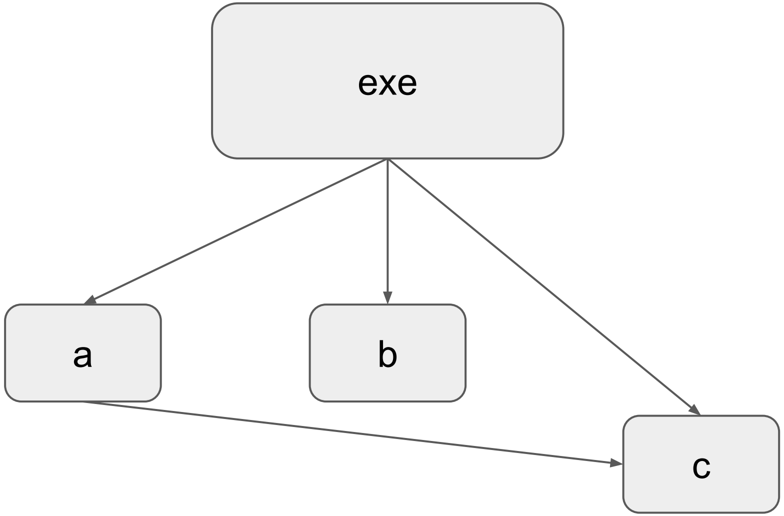
Seems like dynamic linker works differently

Dynamic linker much more lazier than static one.

Static linker checks all symbols with the same name.
Dynamic linker just uses first found symbol

Dynamic linking

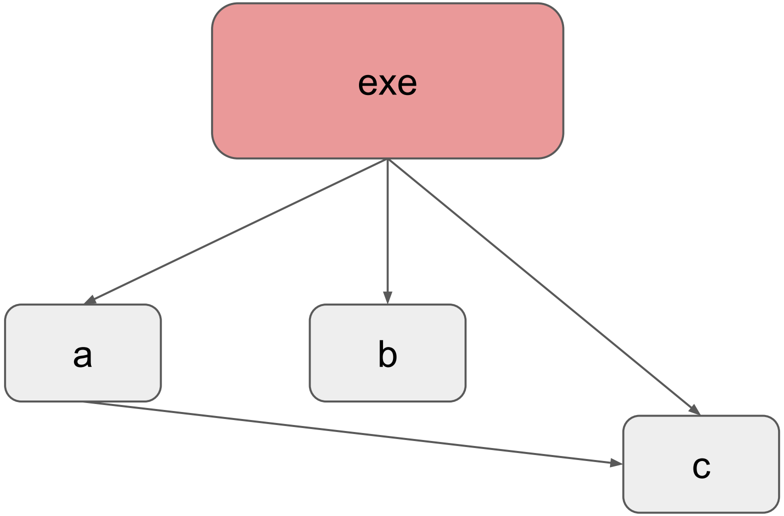
We have some executable with some dynamic dependencies. And dependencies have their own dependencies. The symbol search will be the following:



Dynamic linking

We have some executable with some dynamic dependencies. And dependencies have their own dependencies. The symbol search will be the following:

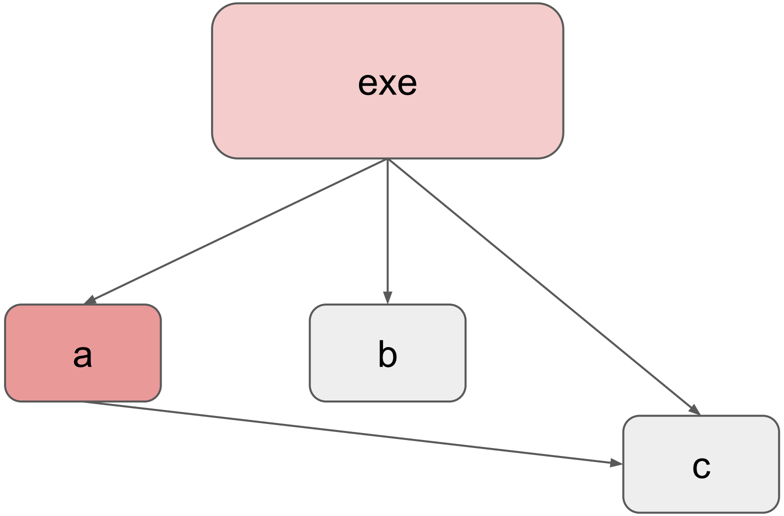
1. executable



Dynamic linking

We have some executable with some dynamic dependencies. And dependencies have their own dependencies. The symbol search will be the following:

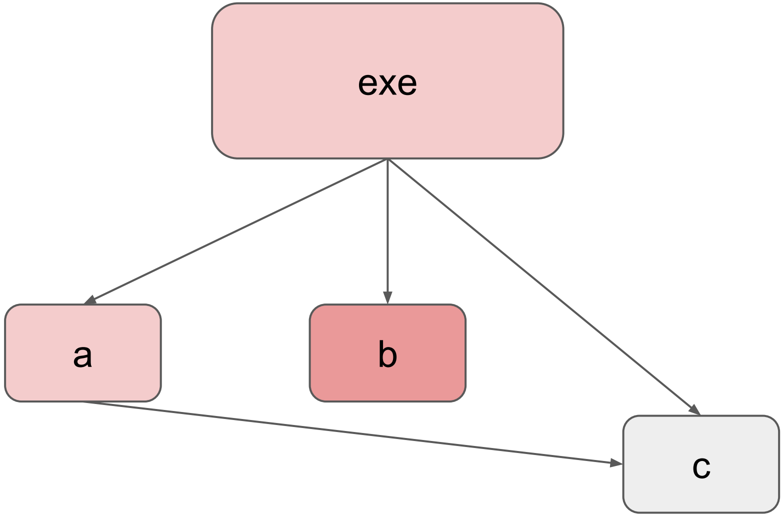
1. executable
2. a



Dynamic linking

We have some executable with some dynamic dependencies. And dependencies have their own dependencies. The symbol search will be the following:

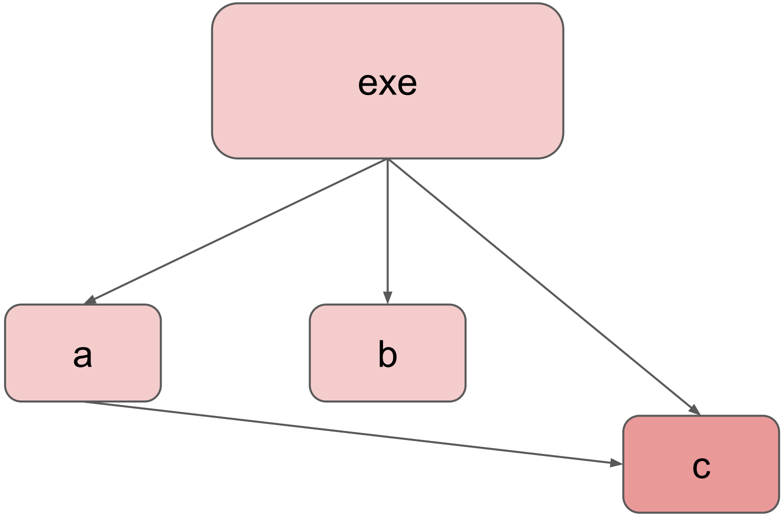
1. executable
2. a
3. b



Dynamic linking

We have some executable with some dynamic dependencies. And dependencies have their own dependencies. The symbol search will be the following:

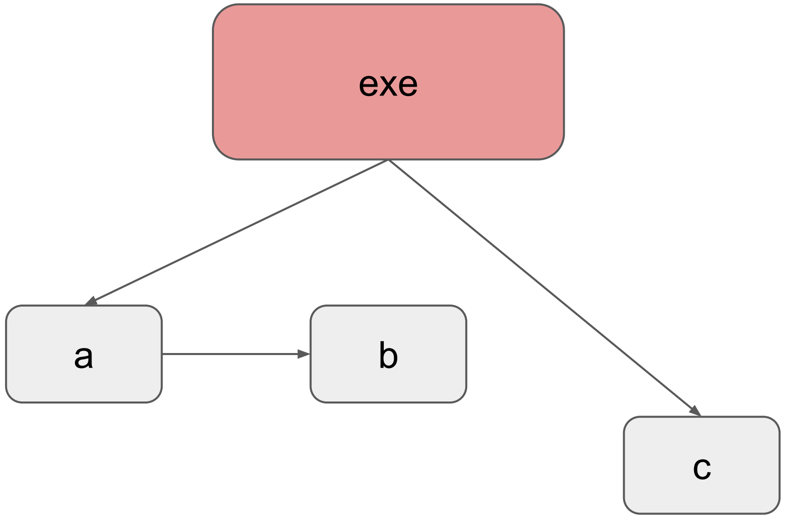
1. executable
2. a
3. b
4. c



Dynamic linking

We have some executable with some dynamic dependencies. And dependencies have their own dependencies. The symbol search will be the following:

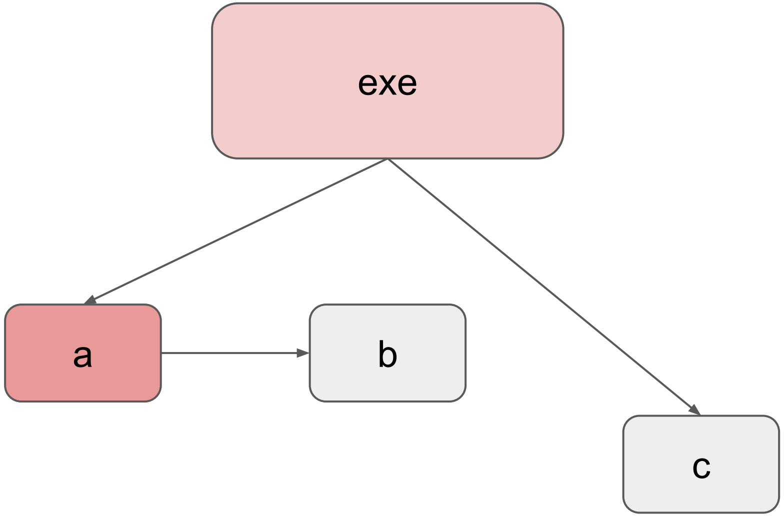
1. executable



Dynamic linking

We have some executable with some dynamic dependencies. And dependencies have their own dependencies. The symbol search will be the following:

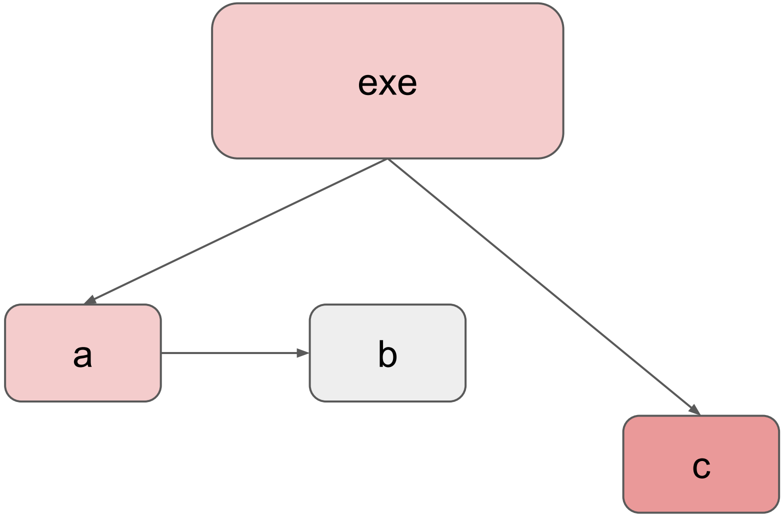
1. executable
2. a



Dynamic linking

We have some executable with some dynamic dependencies. And dependencies have their own dependencies. The symbol search will be the following:

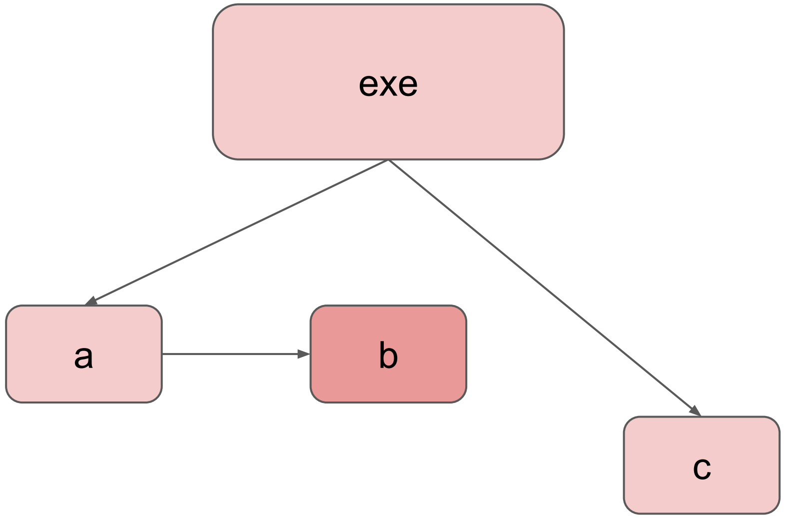
1. executable
2. a
3. c



Dynamic linking

We have some executable with some dynamic dependencies. And dependencies have their own dependencies. The symbol search will be the following:

1. executable
2. a
3. c
4. b



Dynamic linking

Let’s check how dynamic linker works with a simple app. What dependencies etc…

```
$ cat main.c
#include <stdlib.h>
int main() {
   int* a = malloc(sizeof(int));
}

$ gcc -fsanitize=address main.c
```

Dynamic linking

Let’s check dynamic dependencies.

```
$ gcc -fsanitize=address main.c
$ ldd ./a.out
	linux-vdso.so.1 (0x00007ffdd0bfd000)
	libasan.so.8 => /lib/x86_64-linux-gnu/libasan.so.8 (0x00007fc054e00000)
	libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fc054a00000)
	libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fc055545000)
	libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fc055518000)
	/lib64/ld-linux-x86-64.so.2 (0x00007fc055651000)
```

Dynamic linking

We can see that ASAN runtime is the **first** dependency (**vdso** is virtual shared object, `man vdso`)

```
$ gcc -fsanitize=address main.c
$ ldd ./a.out
      linux-vdso.so.1 (0x00007ffdd0bfd000)
      libasan.so.8 => /lib/x86_64-linux-gnu/libasan.so.8 (0x00007fc054e00000)
      libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fc054a00000)
      libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fc055545000)
      libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fc055518000)
      /lib64/ld-linux-x86-64.so.2 (0x00007fc055651000)
```

Dynamic linking

That’s how it intercepts all necessary functions.

```
$ gcc -fsanitize=address main.c
$ ldd ./a.out
      linux-vdso.so.1 (0x00007ffdd0bfd000)
      libasan.so.8 => /lib/x86_64-linux-gnu/libasan.so.8 (0x00007fc054e00000)
      libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fc054a00000)
      libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fc055545000)
      libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fc055518000)
      /lib64/ld-linux-x86-64.so.2 (0x00007fc055651000)
```

Dynamic linking

Let's check how `malloc` is being searched

```
$ cat main.c
#include <stdlib.h>
int main() {
   int* a = malloc(sizeof(int));
}
$ gcc -fsanitize=address main.c
```

Dynamic linking

Firstly it is being searched in executable (without success)
Secondally in lib**asan**. And it was been found in libasan.

```
$ cat main.c
#include <stdlib.h>
int main() {
   int* a = malloc(sizeof(int));
}
$ gcc -fsanitize=address main.c
$ LD_DEBUG=symbols 2>&1 ./a.out | grep malloc
    319140:   symbol=malloc;  lookup in file=./a.out [0]
    319140:   symbol=malloc;  lookup in file=/lib/x86_64-linux-gnu/libasan.so.8 [0]
```

Dynamic linking

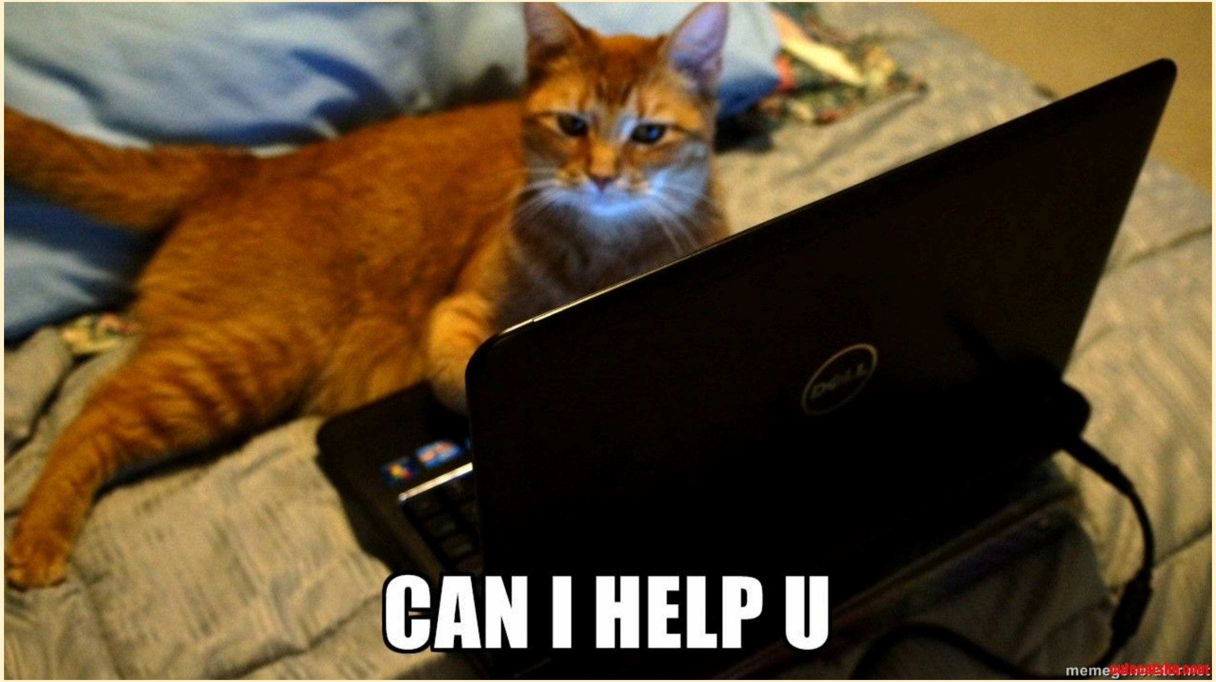
LD_DEBUG – very useful tool

```
$ LD_DEBUG=help cat
```

Dynamic linking

LD_DEBUG – very useful tool

```
$ LD_DEBUG=help cat
```



Dynamic linking

LD_DEBUG – very useful tool

```
$ LD_DEBUG=help cat
Valid options for the LD_DEBUG environment variable are:

  libs        display library search paths
  reloc       display relocation processing
  files       display progress for input file
  symbols     display symbol table processing
  bindings    display information about symbol binding
  versions    display version dependencies
  scopes      display scope information
  all         all previous options combined
  statistics  display relocation statistics
  unused      determined unused DSOs
  help        display this help message and exit

To direct the debugging output into a file instead of standard output
a filename can be specified using the LD_DEBUG_OUTPUT environment variable.
```

Dynamic linking

LD_DEBUG - why cat is able to talk?

```
$ LD_DEBUG=help cat
Valid options for the LD_DEBUG environment variable are:

  libs        display library search paths
  reloc       display relocation processing
  files       display progress for input file
  symbols     display symbol table processing
  bindings    display information about symbol binding
  versions    display version dependencies
  scopes      display scope information
  all         all previous options combined
  statistics  display relocation statistics
  unused      determined unused DSOs
  help        display this help message and exit

To direct the debugging output into a file instead of standard output
a filename can be specified using the LD_DEBUG_OUTPUT environment variable.
```

Dynamic linking

LD_DEBUG - why cat is able to talk?

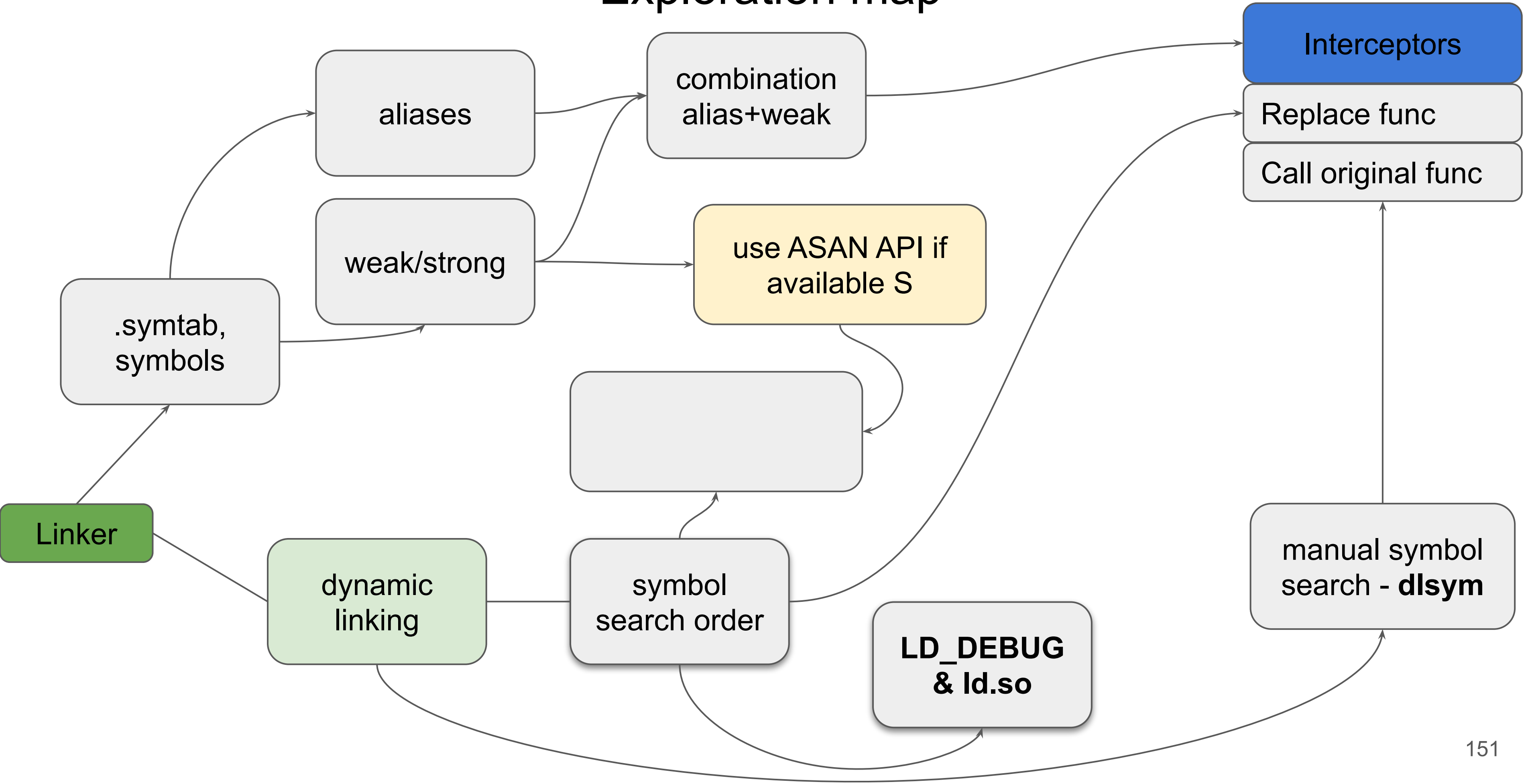
```
$ man ld.so
NAME
     ld.so, ld-linux.so - dynamic linker/loader

SYNOPSIS
     The dynamic linker can be run either indirectly by running some dynami-
     cally  linked  program  or shared object (in which case no command-line
     options to the dynamic linker can be passed and, in the ELF  case,  the
     dynamic linker which is stored in the .interp section of the program is
     executed) or directly by running:

     /lib/ld-linux.so.*  [OPTIONS] [PROGRAM [ARGUMENTS]]

DESCRIPTION
     The  programs  ld.so  and ld-linux.so* find and load the shared objects
     (shared libraries) needed by a program, prepare the program to run, and
     then run it.
```

Dynamic linking

LD_DEBUG, LD_PRELOAD, LD_LIBRARY_PATH, etc – are env var for **ld.so**

```
$ man ld.so
NAME
     ld.so, ld-linux.so - dynamic linker/loader

SYNOPSIS
     The dynamic linker can be run either indirectly by running some dynami-
     cally  linked  program  or shared object (in which case no command-line
     options to the dynamic linker can be passed and, in the ELF  case,  the
     dynamic linker which is stored in the .interp section of the program is
     executed) or directly by running:

     /lib/ld-linux.so.*  [OPTIONS] [PROGRAM [ARGUMENTS]]

DESCRIPTION
     The  programs  ld.so  and ld-linux.so* find and load the shared objects
     (shared libraries) needed by a program, prepare the program to run, and
     then run it.
```

Exploration map



It’s time to fix our application!

Do you see the problem?

```
$ cat main.c
#include <sanitizer/asan_interface.h>
void __attribute__((weak))
__sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}

int main() {
   __sanitizer_print_memory_profile(100, 1000);
}
```

It’s time to fix our application!

Do you see the problem?

Yes. Application will always use their own implementation (in case of dynamic ASAN)

```
$ cat main.c
#include <sanitizer/asan_interface.h>
void __attribute__((weak))
__sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}

int main() {
   __sanitizer_print_memory_profile(100, 1000);
}
```

It’s time to fix our application!

The solution is simple – move this function to a separate .so

```
$
```

It’s time to fix our application!

The solution is simple – move this function to a separate .so

```
$ cat fake_asan_profile.c
#include <stdlib.h>
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}
```

It’s time to fix our application!

The solution is simple – move this function to a separate .so

```
$ cat fake_asan_profile.c
#include <stdlib.h>
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}
$ cat main.c
#include <sanitizer/asan_interface.h>
int main() {
   __sanitizer_print_memory_profile(100, 1000);
}
```

It’s time to fix our application!

The solution is simple – move this function to a separate .so

```
$ cat fake_asan_profile.c
#include <stdlib.h>
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}
$ cat main.c
#include <sanitizer/asan_interface.h>
int main() {
   __sanitizer_print_memory_profile(100, 1000);
}
$ gcc -shared asan_profile.c -o libfake_asan_profile.so
$ gcc main_old.c -L. -lfake_asan_profile
```

It's time to fix our application!

The solution is simple – move this function to a separate .so

```
$ cat fake_asan_profile.c
#include <stdlib.h>
void __sanitizer_print_memory_profile(size_t top_percent, size_t max_number_of_contexts) {}
$ cat main.c
#include <sanitizer/asan_interface.h>
int main() {
   __sanitizer_print_memory_profile(100, 1000);
}
$ gcc -shared asan_profile.c -o libfake_asan_profile.so
$ gcc main_old.c -L. -lfake_asan_profile
$ ./a.out
$
```

It’s time to fix our application!

The solution is simple – move this function to a separate .so

```
$ gcc main_old.c -L. -lfake_asan_profile
$ ./a.out
$
$ ldd a.out
linux-vdso.so.1 (0x00007fffc1953000)
libfake_asan_profile.so => ./libfake_asan_profile.so (0x00007c73c5d20000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007c73c5a00000)
/lib64/ld-linux-x86-64.so.2 (0x00007c73c5d2c000)
```

It’s time to fix our application!

The solution is simple – move this function to a separate .so
Works as expected (without ASAN runtime)

```
$ gcc main_old.c -L. -lfake_asan_profile
$ ./a.out
$
$ ldd a.out
linux-vdso.so.1 (0x00007fffc1953000)
libfake_asan_profile.so => ./libfake_asan_profile.so (0x00007c73c5d20000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007c73c5a00000)
/lib64/ld-linux-x86-64.so.2 (0x00007c73c5d2c000)
$ LD_DEBUG=symbols 2>&1 ./a.out | grep __sanitizer_print_memory_profile
323573:     symbol=__sanitizer_print_memory_profile;  lookup in file=./a.out [0]
323573:     symbol=__sanitizer_print_memory_profile;  lookup in file=./libfake_asan_profile.so [0]
```

It’s time to fix our application!

Let’s check it with ASAN runtime

```
$ gcc -fsanitize=address main_old.c -L. -lfake_asan_profile
$ ./a.out
```

It’s time to fix our application!

Let’s check it with ASAN runtime.
Works!

```
$ gcc -fsanitize=address main_old.c -L. -lfake_asan_profile
$ ./a.out
Live Heap Allocations: 92 bytes in 2 chunks; quarantined: 0 bytes in 0 chunks; 1948 other chunks;
total chunks: 1950; showing top 100% (at most 1000 unique contexts)
68 byte(s) (73%) in 1 allocation(s)
    #0 0x7dcf952fbb37 in malloc ../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:69
    #1 0x7dcf95936db1 in malloc ../include/rtld-malloc.h:56
    #2 0x7dcf95936db1 in __GI__dl_exception_create_format elf/dl-exception.c:157
    #3 0x7dcf9593e641 in _dl_lookup_symbol_x elf/dl-lookup.c:809
    #4 0x7dcf94f8521c in do_sym elf/dl-sym.c:146
...
```

It’s time to fix our application!

Let’s check it with ASAN runtime.
libasan.so has higher priority. So we’ve created “default” implementation for this func.

```
$ gcc -fsanitize=address main_old.c -L. -lfake_asan_profile
$ ldd a.out
linux-vdso.so.1 (0x00007fff2018e000)
libasan.so.8 => /lib/x86_64-linux-gnu/libasan.so.8 (0x0000746a29200000)
libfake_asan_profile.so => ./libfake_asan_profile.so (0x0000746a29a31000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x0000746a28e00000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x0000746a29948000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x0000746a2991b000)
/lib64/ld-linux-x86-64.so.2 (0x0000746a29a59000)
```

It’s time to fix our application!

Let’s check it with ASAN runtime.
Symbol search check via LD_DEBUG:

```
$ gcc -fsanitize=address main_old.c -L. -lfake_asan_profile
$ LD_DEBUG=symbols 2>&1 ./a.out | grep __sanitizer_print_memory_profile
324222:    symbol=__sanitizer_print_memory_profile;  lookup in file=./a.out [0]
324222:    symbol=__sanitizer_print_memory_profile;  lookup in
file=/lib/x86_64-linux-gnu/libasan.so.8 [0]
```

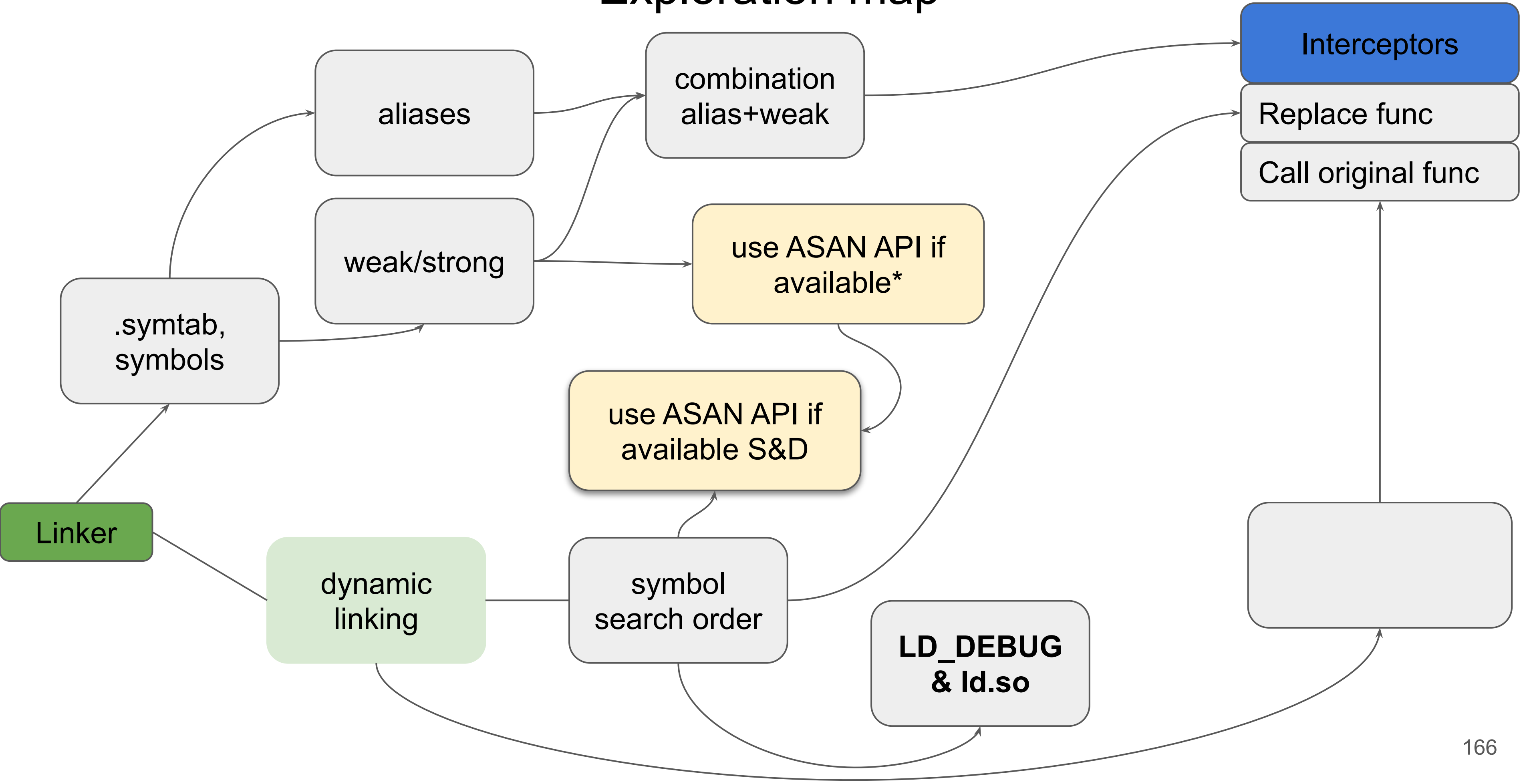
It’s time to fix our application!

With and without ASAN runtime

```
$ gcc -fsanitize=address main_old.c -L. -lfake_asan_profile
$ LD_DEBUG=symbols 2>&1 ./a.out | …
lookup in file=./a.out [0]
lookup in file=/lib/x86_64-linux-gnu/libasan.so.8 [0]
```

```
$ gcc main_old.c -L. -lfake_asan_profile
$ LD_DEBUG=symbols 2>&1 ./a.out | …
lookup in file=./a.out [0]
lookup in file=./libfake_asan_profile.so [0]
```

Exploration map



Sum up techniques

To create default implementation which could be replaced by some another one you can:

For static linking: create WEAK symbol

Executable	default weak **foo**()
Executable	Possible **strong** overload for **foo**()

Sum up techniques

To create default implementation which could be replaced by some another one you can:

For static linking: create WEAK symbol

For dynamic linking: move your symbol to dynamic lib. Load it AFTER possible overload.

Executable	default weak **foo**()
Executable	Possible **strong** overload for **foo**()

.so	possible overload for **boo**()

.so	default **boo**()

Sum up techniques

To create default implementation which could be replaced by some another one you can:

For static linking: create WEAK symbol

For dynamic linking: move your symbol to dynamic lib. Load it AFTER possible overload.

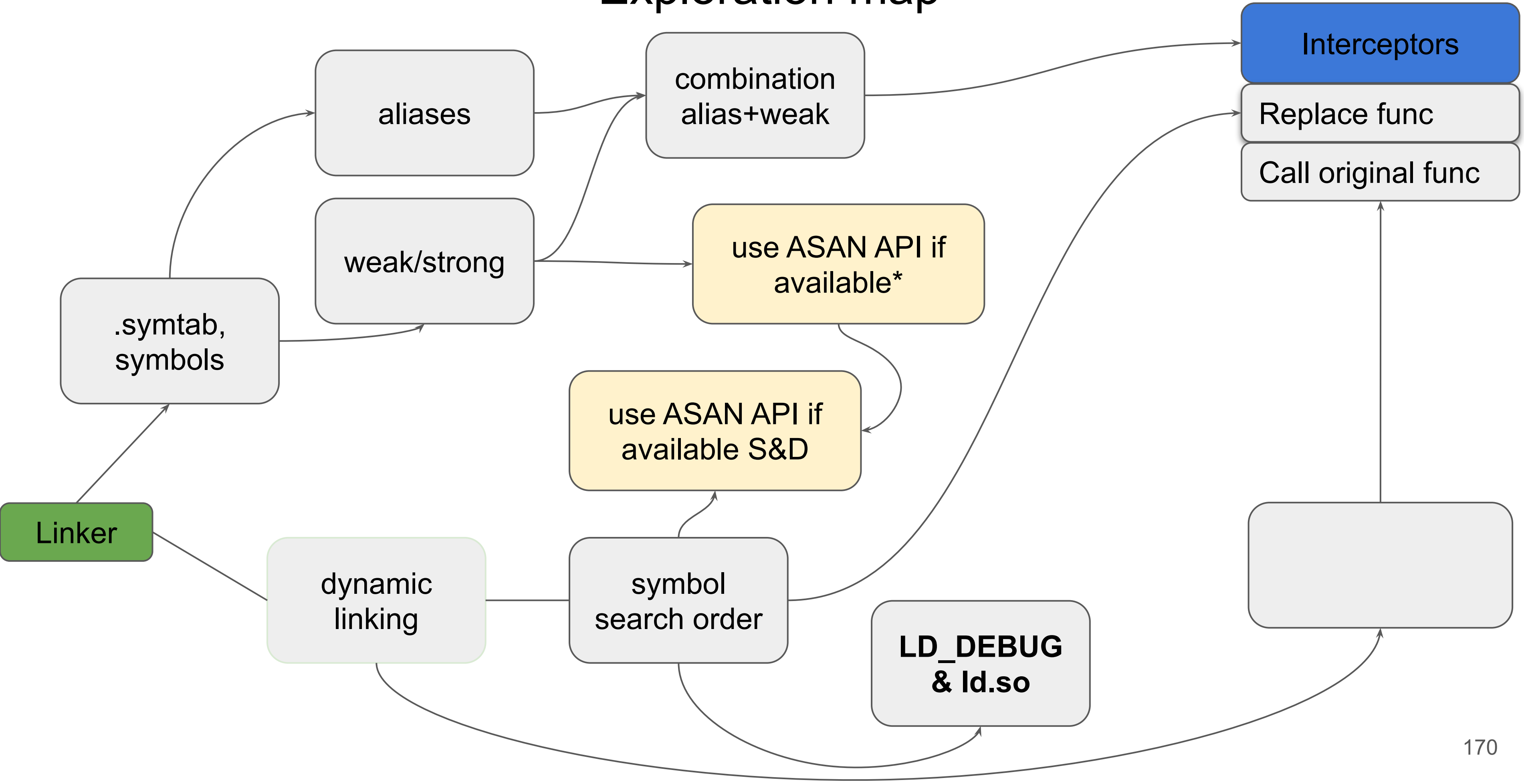
To redefine symbol define youth version in executable or in dynamic lib which will be loaded first.

Executable	default weak **foo**()
Executable	Possible **strong** overload for **foo**()
Executable	Possible overload for **boo**()

.so	possible overload for **boo**()

.so	default **boo**()

Exploration map



How to call redefined function?

How to call redefined function?

Just use **dlsym** call!

How to call redefined function?

Just use **dlsym** call!

```
$ cat my_malloc.c
#include <unistd.h>
#include <dlfcn.h>
typedef void*(*real_malloc)(size_t)
void* malloc(size_t size) {
   write(1, "my malloc\n", 11);
   void* func = dlsym(RTDL_NEXT, "malloc");
   real_malloc real_malloc = func;
   return real_malloc(size);
}
```

How to call redefined function?

Just use **dlsym** call!
dlsym RTDL_NEXT finds a **next** given symbol

```
$ cat my_malloc.c
#include <unistd.h>
#include <dlfcn.h>
typedef void*(*real_malloc)(size_t)
void* malloc(size_t size) {
   write(1, "my malloc\n", 11);
   void* func = dlsym(RTDL_NEXT, "malloc");
   real_malloc real_malloc = func;
   return real_malloc(size);
}
```

How to call redefined function?

Let's try it!

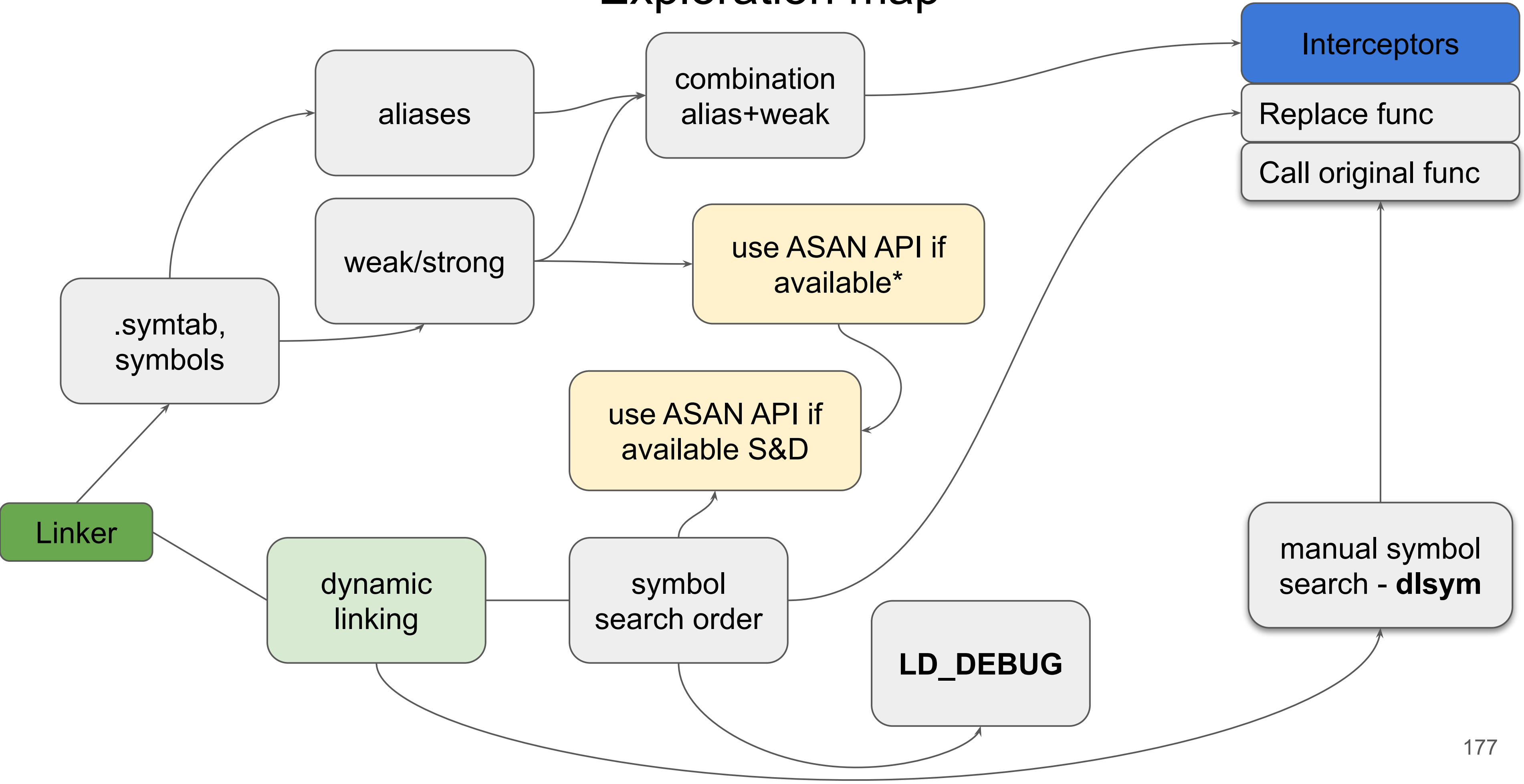
```
$ cat my_malloc.c
#include <unistd.h>
#include <dlfcn.h>
typedef void*(*real_malloc)(size_t)
void* malloc(size_t size) {
   write(1, "my malloc\n", 11);
   void* func = dlsym(RTDL_NEXT, "malloc");
   real_malloc real_malloc = func;
   return real_malloc(size);
}
$ clang -shared my_malloc.c -o my_malloc.so
$ cat main.c
int main() {
   printf("%p\n", malloc(10));
}
$ clang -L. -l:my_malloc.so main.c
```

How to call redefined function?

Let's run it!

```
$ cat my_malloc.c
#include <unistd.h>
#include <dlfcn.h>
typedef void*(*real_malloc)(size_t)
void* malloc(size_t size) {
   write(1, "my malloc\n", 11);
   void* func = dlsym(RTDL_NEXT, "malloc");
   real_malloc real_malloc = func;
   return real_malloc(size);
}
$ clang -shared my_malloc.c -o my_malloc.so
$ cat main.c
int main() {
   printf("%p\n", malloc(10));
}
$ clang -L. -l:my_malloc.so main.c
$ ./a.out
my malloc
my malloc
0x5dbd8acb12a0
```

Exploration map



Sanitizer interceptors IRL

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

It is so complex to allow several tools which have to intercept this functions work together

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

First function: trampoline. A short one. Just calls `__interceptor_malloc`

Two symbols one function. WEAK and STRONG

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a      5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a      5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50    575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50    575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

First function: trampoline. A short one. Just calls `__interceptor_malloc`
Two symbols one function. WEAK and STRONG

Second function: real ASAN malloc implementation.
Again: two symbols (WEAK and STRONG) one function.

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

How does it work normally?

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

How does it work normally?

Code calls **malloc**. Linker finds **malloc** symbol in ASAN runtime. **malloc is trampoline**

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

How does it work normally?

Code calls **malloc**. Linker finds **malloc** symbol in ASAN runtime. **malloc is trampoline**

Trampoline calls **__interceptor_malloc** symbol. **__interceptor_malloc** is actual malloc implementation in ASAN

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

Extension points: all WEAK symbols here.

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

Extension points: all WEAK symbols here. Other tool can redefine **malloc** or **__interceptor_malloc**

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

Extension points: all WEAK symbols here. Other tool can redefine **malloc** or **__interceptor_malloc**

In both cases redefinition should call a STRONG symbol from the same pair

Sanitizer interceptors IRL

```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

Extension points: all WEAK symbols here. Other tool can redefine **malloc** or **__interceptor_malloc**

In both cases redefinition should call a STRONG symbol from the same pair

malloc -> __interceptor_trampoline_malloc
__interceptor_malloc -> ___interceptor_malloc

Sanitizer interceptors IRL

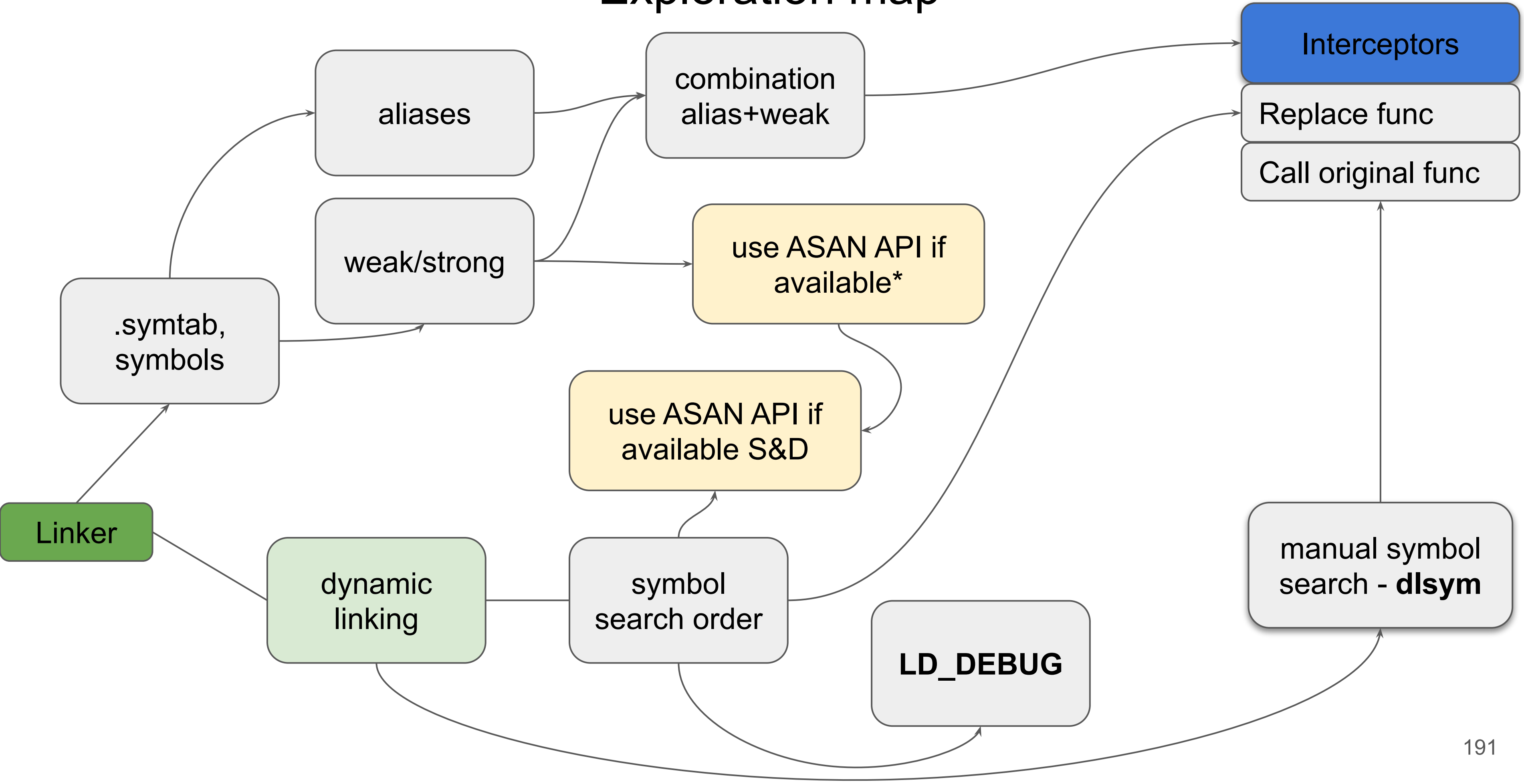
```
$ readelf -sW /lib/x86_64-linux-gnu/libasan.so.8 | grep malloc
  2439: 00000000000fa57a     5 FUNC    WEAK   DEFAULT   14 malloc
   902: 00000000000fa57a     5 FUNC    GLOBAL DEFAULT   14 __interceptor_trampoline_malloc
   214: 00000000000fba50   575 FUNC    GLOBAL DEFAULT   14 ___interceptor_malloc
  2173: 00000000000fba50   575 FUNC    WEAK   DEFAULT   14 __interceptor_malloc
```

If both are redefined we will have a chain of 3 interceptors (including sanitizer one):

malloc -> __interceptor_trampoline_malloc -> **__interceptor_malloc** -> **___interceptor_malloc**

Red - interceptors with custom logic

Exploration map



What we can do now? (bonus)

What we can do now? (bonus)

We can create a library for pretty drawing a Leak Sanitizer report. Which will work automatically.

What we can do now? (bonus)

We can create a library for pretty drawing a Leak Sanitizer report. Which will work automatically. Just like this:

```
$ cat main.cxx
struct N {N* next;};
int main() {
   auto a = new N{};
   auto b = new N{};
   a->next = b;
   b->next = a;
   return 0;
}
```

What we can do now? (bonus)

Compiling it WITHOUT Leak or Address sanitizer!

```
$ cat main.cxx
struct N {N* next;};
int main() {
   auto a = new N{};
   auto b = new N{};
   a->next = b;
   b->next = a;
   return 0;
}
$ g++ main.c
```

What we can do now? (bonus)

Run it!

```
$ cat main.cxx
struct N {N* next;};
int main() {
   auto a = new N{};
   auto b = new N{};
   a->next = b;
   b->next = a;
   return 0;
}
$ g++ main.c
$ LD_PRELOAD="./my_secret_lib.so /lib/x86_64-linux-gnu/liblsan.so.0" ./a.out
```

What we can do now? (bonus)

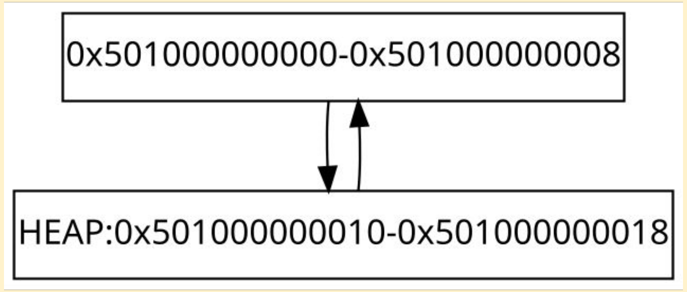
Draw leaked memory graph

```
$ cat main.cxx
struct N {N* next;};
int main() {
   auto a = new N{};
   auto b = new N{};
   a->next = b;
   b->next = a;
   return 0;
}
$ g++ main.c
$ LD_PRELOAD="./my_secret_lib.so /lib/x86_64-linux-gnu/liblsan.so.0" ./a.out
$ dot -Tpng lsan_scan.dot > lsan_scan.png
```

What we can do now? (bonus)

Show it

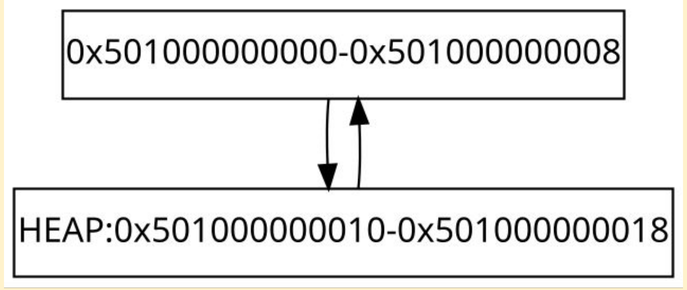
```
$ cat main.cxx
struct N {N* next;};
int main() {
   auto a = new N{};
   auto b = new N{};
   a->next = b;
   b->next = a;
   return 0;
}
$ g++ main.c
$ LD_PRELOAD="./my_secret_lib.so /lib/x86_64-linux-gnu/liblsan.so.0" ./a.out
$ dot -Tpng lsan_scan.dot > lsan_scan.png
$ cat lsan_scan.png
```



What we can do now? (bonus)

How? I’ll tell about it on the next talk :-)

```
$ cat main.cxx
struct N {N* next;};
int main() {
   auto a = new N{};
   auto b = new N{};
   a->next = b;
   b->next = a;
   return 0;
}
$ g++ main.c
$ LD_PRELOAD="./my_secret_lib.so /lib/x86_64-linux-gnu/liblsan.so.0" ./a.out
$ dot -Tpng lsan_scan.dot > lsan_scan.png
$ cat lsan_scan.png
```



Explore your tools

Explore your tools

Go deeper

Explore your tools

Go deeper

Use it in unusual ways!

Explore your tools

Go deeper

Use it in unusual ways!

Explore, learn, give a talk!

```
$ man ld.so
$ LD_DEBUG=help cat
$ ldd ./a.out
$ man readelf
$ man nm
$ readelf -aW your.o
$ readelf -sW m.o
$ LD_PRELOAD=/lib/x86_64-linux-gnu/liblsan.so.0 ./a.out
```

Additional materials are here ⇒

Questions for you:

- can sanitizer work if your application was linked statically with libc?
- how to load your shared library lately?
- where is dynamic dependencies are located?
- are exported symbols in shared libraries located in .symtab?
- is it possible to override strong symbol for static linking?
- how does C++ multiple definitions are working - linker&symbols level (inline functions in headers, templates the same instantations…)