

# Программирование в Linux

Запуск и исполнение программ



# Hello world!


```
home > dmis > LinuxEgs > C hello_world.c > ...
```

```
1  #include <stdio.h>
2
3  int main(int argc, char* argv[]) {
4      printf("Hello World\n");
5      return 0;
6  }
7
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
dmis@dmis-MS-7A15:~/LinuxEgs$ gcc -o hello hello_world.c
dmis@dmis-MS-7A15:~/LinuxEgs$ ./hello
Hello World
```

1. bash -> fork
2. bash -> exec
3. exec -> check ELF
4. exec -> new address space
5. exec -> mmap binary module
6. exec -> dynamic linker
7. dynamic linker -> recursive mmap deps
8. dynamic linker -> lib relocation
9. ret, ret, ret...
10. pass args & envs
11. run entry point (main)



что происходит в этот момент?

# Будем разбираться с ELFами



*Executable & Linkable format*

**Executable**  
programs shared  
libs (иногда)

**Linkable**  
shared libs  
programs  
(иногда)

**Relocatable**  
object files  
static libs  
kernel modules

**Core**  
dump

В начале было слово. И слово было  
тридцатидвухбитным. И слово было 0x7F 0x45 0x4C  
0x46

### Elf header

Поле	Тип данных	Смещение
e_ident	Unsigned char	0x00
e_type	Elf32_Half	0x10
e_machine	Elf32_Half	0x12
e_version	Elf32_Word	0x14
e_entry	Elf32_Addr	0x18
e_phoff	Elf32_Off	0x1C
e_shoff	Elf32_Off	0x20
e_flags	Elf32_Word	0x24
e_ehsize	Elf32_Half	0x28
e_phentsize	Elf32_Half	0x2A
e_phnum	Elf32_Half	0x2C
e_shentsize	Elf32_Half	0x2E
e_shnum	Elf32_Half	0x30
e_shstrndx	Elf32_Half	0x32

В заголовке содержатся

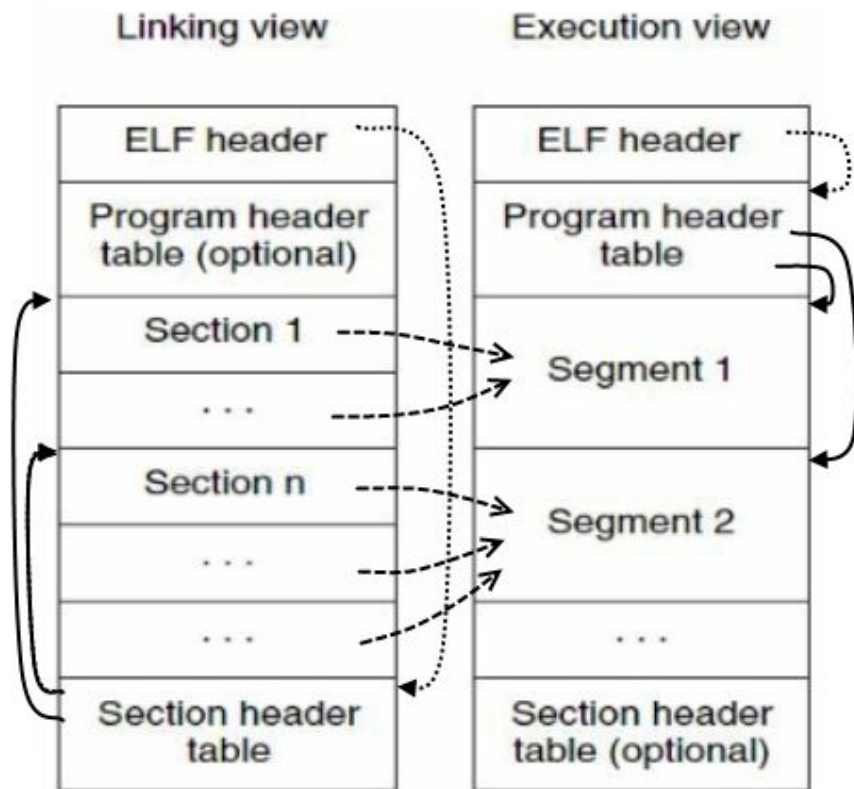
- Версия формата
- Целевая архитектура
- Ссылки на части файла  
(секции/сегменты)
- Адрес точки входа

```
dmis@dmis-MS-7A15:~/LinuxEgs$ readelf -h hello
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                              ELF64
  Data:                               2's complement, little endian
  Version:                            1 (current)
  OS/ABI:                             UNIX - System V
  ABI Version:                        0
  Type:                               DYN (Shared object file)
  Machine:                            Advanced Micro Devices X86-64
  Version:                            0x1
  Entry point address:                 0x1060
  Start of program headers:            64 (bytes into file)
  Start of section headers:            14712 (bytes into file)
  Flags:                               0x0
  Size of this header:                  64 (bytes)
  Size of program headers:              56 (bytes)
  Number of program headers:            13
  Size of section headers:              64 (bytes)
  Number of section headers:            31
  Section header string table index:    30
```

Поизучать ELF можно с помощью утилит **readelf** и **objdump**

Чтобы ковырять руками: `#include <linux/elf.h>` (`Elf64_Ehdr`)

# Секции и сегменты ELF



## Сегменты

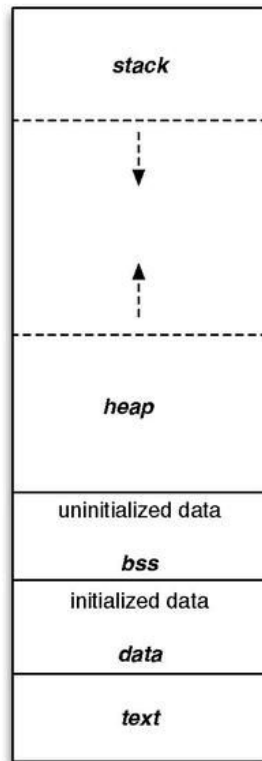
- Используются при загрузке на выполнение
- Собираются линковщиком из секций с одинаковыми правами доступа
- На сегменты накладываются права доступа (RWX)

## Секции

- Используются линковщиком
- Генерируются из единиц трансляции
- Переупорядочиваются и группируются линковщиком
- Участвуют в LTO (link time optimization)

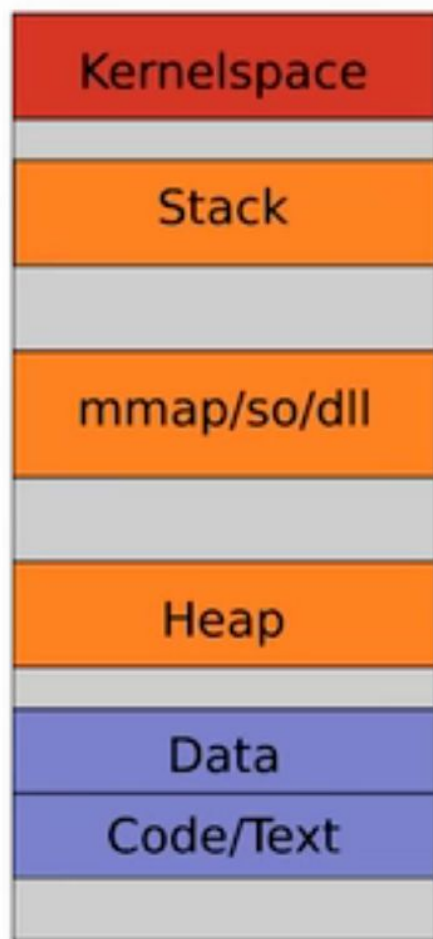
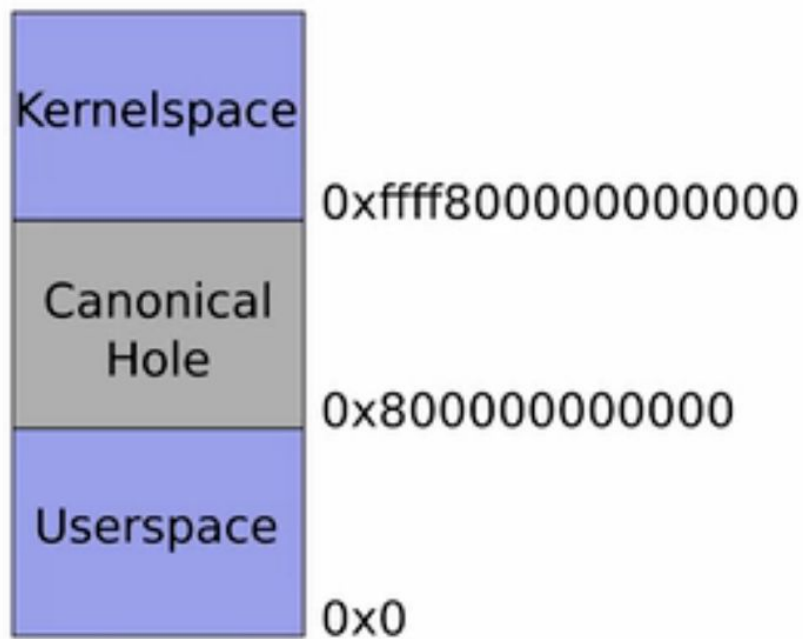
# Наиболее распространенные секции

- `.text` -- секция кода. RX
- `.data` -- секция  
инициализированных данных,  
RW
- `.rodata` -- секция  
инициализированных  
констант, R
- `.bss` --  
неинициализированные  
данные, RW, в ELF  
реального места не занимает





x86-64



# Что если есть библиотеки-зависимости?

## **Статические**

ничего не надо делать

## **Динамические**

как грузить?

# Динамические библиотеки

- Можно экономить место
  - Можно экономить память!
  - Можно грузить по требованию -- плагины!
  - Можно заменять без пересборки
- По какому адресу грузить?
  - Фиксированному?
  - Не фиксированному?
  - Как по этому адресу обращаться программе?
  - Как библиотеке обращаться к своему коду?

## Компоновка vs Загрузка динамических библиотек

Переменные/функции -- адреса могут быть неизвестны при компиляции .o

Но компоновщик справляется -- "символьные ссылки"

Адреса подставляются в самом конце

# Секция PT\_INTERP

Читаем имя файла динамического линковщика

Запускаем его вместо нашей программы

Направляем на загруженный ELF.

Ему осталось определить все адреса

Релокация  
(как статический линкер)

PIC



# Релокация

```
1 // main
2 #include "hello.h"
3
4 int main() {
5     hello();
6     return 0;
7 }

1 // libhello
2 #include "hello.h"
3
4 #include <stdio.h>
5
6
7 void hello_private() {
8     puts("Hello, world!\n");
9 }
10
11
12 void hello() {
13     hello_private();
14 }
```

1. Грузим main. Адрес hello неизвестен
2. Грузим libhello по какому-то адресу
3. Теперь мы знаем адрес hello -> подставляем его **в каждой** точке вызова в main
4. Внутри libhello теперь известен адрес hello\_private -- подставляем его **в каждой** точке вызова
5. И так для каждого символа!

# Релокация

## За

- Очень простой алгоритм
- Исполняемый код столь же эффективен, как и при статической компоновке

## Против

- Долгая загрузка
- Секция кода библиотеки доступна на запись!
- Экземпляр библиотеки невозможно разделить между процессами!

# Position Independent Code

Идея: функции и переменные используются чаще чем объявляются

```
1 // libhello
2 #include "hello.h"
3
4 #include <stdio.h>
5
6
7 void hello_private() {
8     puts("Hello, world!\n");
9 }
10
11
12 static FUNCT_ADDRS_T FUNC_ADDRS[....]; // будет заполнено
13                                         // при загрузке библиотеки
14
15 void hello() {
16     // hello_private();
17     (current_instruction_pointer +
18      offset_in_dll_to(FUNC_ADDRS))[0]();
19 }
```



## Position Independent Code

### За

- Быстрая загрузка
- Резолвить символы можно лениво
- Секция кода не модифицируется
- Можно разделять между процессами

### Против

- Косвенная адресация
- Чуть менее эффективный код
- Больше обращений к памяти -- меньше надежность в особых условиях работы

# Как резолвить лениво?

```
1 #include <iostream>
2
3 using Func = void(void);
4
5 void resolve();
6
7 static Func* GOT[] = {resolve}; // линкер записывает адрес вспомогательного
8                                 // кода сюда,
9                                 // а не сразу ищет адрес нужной функции
10
11 void actual_func() {
12     std::cout << "actual func\n";
13 }
14
15 void resolve() {
16     std::cout << "resolve!\n";
17     // при первом вызове отработывает код линкера,
18     // находящий нужный адрес
19     (GOT[0] = actual_func)();
20 }
21
22
23 void call_func_plt(){
24     GOT[0]();
25 }
26
27 int main() {
28     call_func_plt();
29     call_func_plt();
30     return 0;
31 }
```

# В ELF нет связи между зависимостями и импортируемыми именами!

```
1  #include <stdio.h>
2
3  extern void hello(const char* name);
4
5  int main(int argc, char* argv[]) {
6      ... hello("world");
7      ... return 0;
8  }
```

```
1  #include <stdio.h>
2
3  void hello(const char* name) {
4      ... printf("hello %s\n", name);
5  }
```

```
dmis@dmis-MS-7A15:~/LinuxEgs$ readelf --dyn-syms hello
```

Symbol table '.dynsym' contains 7 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	__ITM_deregisterTMCloneTab
2:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	__libc_start_main@GLIBC_2.2.5 (2)
3:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	hello
4:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__
5:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	__ITM_registerTMCloneTable
6:	0000000000000000	0	FUNC	WEAK	DEFAULT	UND	__cxa_finalize@GLIBC_2.2.5 (2)

```
dmis@dmis-MS-7A15:~/LinuxEgs$ readelf -d hello
```

Dynamic section at offset 0x2db8 contains 28 entries:

Tag	Type	Name/Value
0x0000000000000001	(NEEDED)	Shared library: [libexample.so]
0x0000000000000001	(NEEDED)	Shared library: [libc.so.6]

```

1  #include <stdio.h>
2
3  extern void hello(const char* name);
4
5  extern int run() {
6      hello("run");
7  }
8
9  int main(int argc, char* argv[]) {
10     hello("world");
11     return 0;
12 }
13

```

T -- символы из секции text  
(экспортируемые)

U -- неопределенные символы  
(импортируемые)

```

dmis@dmis-MS-7A15:~/LinuxEgs$ nm -s hello
0000000000004010 B __bss_start
0000000000004010 b completed.0
                                w __cxa_finalize@@GLIBC_2.2.5
0000000000004000 D __data_start
0000000000004000 W data_start
00000000000001090 t deregister_tm_clones
00000000000001100 t __do_global_dtors_aux
00000000000003db0 d __do_global_dtors_aux_fini_array_entry
0000000000004008 D __dso_handle
00000000000003db8 d _DYNAMIC
0000000000004010 D _edata
0000000000004018 B _end
00000000000001208 T _fini
00000000000001140 t frame_dummy
00000000000003da8 d __frame_dummy_init_array_entry
00000000000002184 r __FRAME_END__
00000000000003fb8 d _GLOBAL_OFFSET_TABLE_
                                w __gmon_start__
00000000000002010 r __GNU_EH_FRAME_HDR
                                U hello
00000000000001000 t _init
00000000000003db0 d __init_array_end
00000000000003da8 d __init_array_start
00000000000002000 R _IO_stdin_used
                                w _ITM_deregisterTMCloneTable
                                w _ITM_registerTMCloneTable
00000000000001200 T __libc_csu_fini
00000000000001190 T __libc_csu_init
                                U __libc_start_main@@GLIBC_2.2.5
00000000000001160 T main
000000000000010c0 t register_tm_clones
00000000000001149 T run
00000000000001060 T _start
0000000000004010 D __TMC_END__

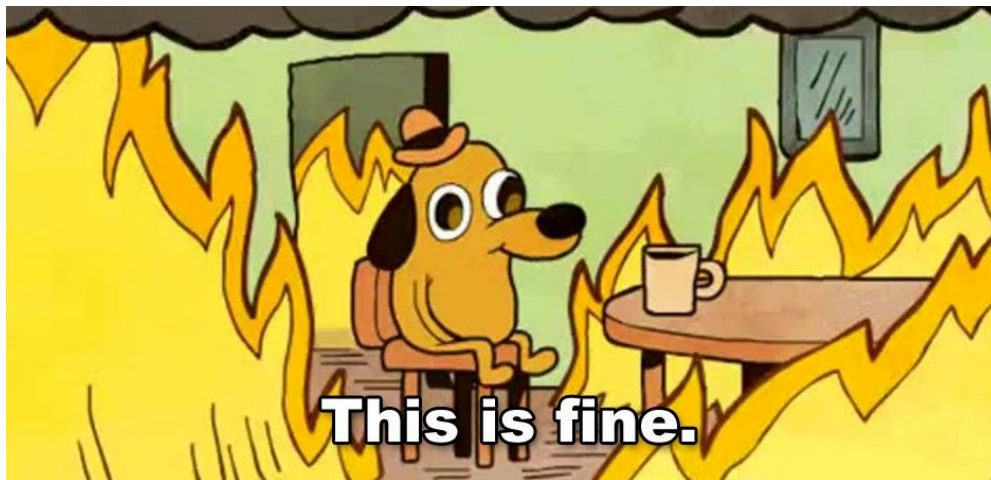
```

Что если две библиотеки определяют один и тот же символ?

**ODR Violation: UB!**

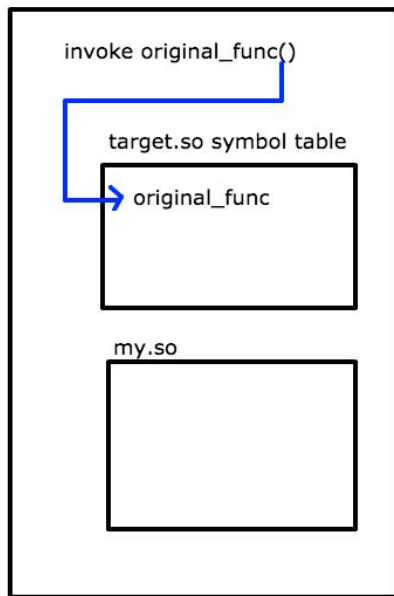
**Но контролируемое!**

Давайте эксплуатировать!

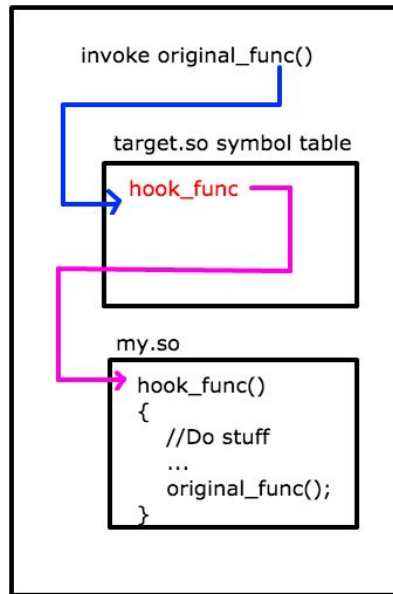


# LD\_PRELOAD и symbol hooking

*/etc/ld.so.preload*



Before Hook



After Hook

Нужно знать *точную* сигнатуру  
подменяемой функции:

имя, возвращаемое значение,  
аргументы и **конвенцию вызова**

# Конвенции вызовов

В ASM нет привычных функций (есть `jmp`, `longjmp` и обертка `-- call`). Аргументы могут передаваться различными способами

## `cdecl`

- аргументы передаются через стек, справа-налево
- вызывающий очищает стек
- вызывающий сохраняет регистры

## `stdcall/winapi`

- почти как `cdecl`, но стек очищает вызываемый

## `fastcall`

- не стандартизирован
- передача аргументов через регистры (`fast`)

## `thiscall`

- конвенция вызова методов объектов (куда класть `this`)

## осторожно, mangling!

```
1 // mangling_eg.cpp
2
3 int hello_mangle(const char*) {
4     return 1;
5 }
6
7 extern "C" {
8     int hello_nomangle(const char*){
9         return 1;
10    }
11 }
```

```
1 $ g++ -std=c++17 -o mangling_eg -shared -fPIC mangling_eg.cpp
2 $ objdump -T mangling_eg
3
4 mangling_eg:      file format elf64-x86-64
5
6 DYNAMIC SYMBOL TABLE:
7 0000000000000000 w D *UND* 0000000000000000 __cxa_finalize
8 0000000000000000 w D *UND* 0000000000000000 _ITM_registerTMCloneTable
9 0000000000000000 w D *UND* 0000000000000000 _ITM_deregisterTMCloneTable
10 0000000000000000 w D *UND* 0000000000000000 __gmon_start__
11 000000000000010f9 g DF .text 0000000000000013 _Z12hello_manglePKc
12 0000000000000110c g DF .text 0000000000000013 hello_nomangle
```



```

1 // hello.h
2 #pragma once
3
4 void hello(const char* msg);
5
6 // hello.c
7 #include "hello.h"
8
9 #include <stdio.h>
10
11 void hello(const char* msg) {
12     printf("hello, %s\n", msg);
13 }
14
15 //main.c
16 #include "hello.h"
17
18 int main(){
19     hello("World");
20 }

```

```

1 // hook.cpp
2 // #define _GNU_SOURCE // для RTLD_NEXT
3 // в старых версиях
4
5 #include <dlfcn.h>
6
7 #include <type_traits>
8 #include <iostream>
9
10 using hello_func = std::decay_t<void(const char*)>;
11
12 extern "C" {
13
14     void hello(const char* msg) {
15         std::cout << "All your base are belong to us\n";
16         auto original_hello =
17             reinterpret_cast<hello_func>(
18                 dlsym(RTLD_NEXT, "hello")
19             );
20         original_hello(msg);
21     }
22
23 }

```

```

1 $ gcc -shared -fPIC -o libhello.so hello.c
2 $ gcc -o hello main.c -L . -lhello -Wl,-rpath=.
3 $ g++ -std=c++17 -o libhook.so -shared -fPIC hook.cpp -ldl
4
5 $ ./hello
6 hello, World
7 $ LD_PRELOAD=./libhook.so ./hello
8 All your base are belong to us
9 hello, World

```

# Видимость символов (linkage)

Какие определения можно импортировать из динамической библиотеки?

- Под Windows (msvc) по умолчанию все символы не видны извне
- Под Unix (gcc) по умолчанию все видно
- Статические и локальные символы не видны никогда

```
1 void hello(char*) {...} // видимость по умолчанию
2                             // (настраивается флагом -fvisibility)
3
4 static void hello(char*) {...} // никогда не видно извне
5
6 __attribute__((visibility ("default"))) void hello(char*) {...} // будет видно
7
8 __attribute__((visibility ("hidden"))) void hello(char*) {...} // будет видно
```

**Прячьте все, кроме публичного интерфейса dll!  
компилируйте с -fvisibility=hidden**

[подробнее](#)

## Типичные вспомогательные макросы

```
1 #if defined _WIN32 || defined __CYGWIN__
2     #ifdef BUILDING_DLL
3         #ifdef __GNUC__
4             #define DLL_PUBLIC __attribute__ ((dllexport))
5         #else
6             #define DLL_PUBLIC __declspec(dllexport)
7         #endif
8     #else
9         #ifdef __GNUC__
10            #define DLL_PUBLIC __attribute__ ((dllimport))
11        #else
12            #define DLL_PUBLIC __declspec(dllimport)
13        #endif
14    #endif
15    #define DLL_LOCAL
16 #else
17     #if __GNUC__ >= 4
18         #define DLL_PUBLIC __attribute__ ((visibility ("default")))
19         #define DLL_LOCAL __attribute__ ((visibility ("hidden")))
20     #else
21         #define DLL_PUBLIC
22         #define DLL_LOCAL
23     #endif
24 #endif

1 extern "C" DLL_PUBLIC void function(int a);
2 class DLL_PUBLIC SomeClass
3 {
4     int c;
5     DLL_LOCAL void privateMethod();
6 public:
7     Person(int _c) : c(_c) { }
8     static void foo(int a);
9 };
```

Visibility распространяется  
на типы! Помните, когда  
бросаете исключения

# Можно ли защититься от подмены символов?

- Проверка переменной LD\_PRELOAD (функция getenv)
- Проверка файла /etc/ld.so.preload
- Проверка массива environ / третий аргумент main
- Проверка информации из /proc/pid/environ
- Перепроверить адреса функций с помощью dlsym
- Использовать системные вызовы напрямую (ASM)

нужные для этого  
функции можно  
подменить

ptrace

# Полезные ссылки

1. <https://eklitzke.org/position-independent-executables>
2. <https://gcc.gnu.org/wiki/Visibility>
3. Про ELF <https://habr.com/ru/post/480642/>
4. Про детект LD\_PRELOAD <https://habr.com/ru/post/479858/>
5. Про подмену функций <https://habr.com/ru/post/106107/>
6. PIC <https://habr.com/ru/company/badoo/blog/323904/>
7. <https://www.opennet.ru/docs/RUS/gas/gas-4.html>
8. [http://www.skyfree.org/linux/references/ELF\\_Format.pdf](http://www.skyfree.org/linux/references/ELF_Format.pdf)