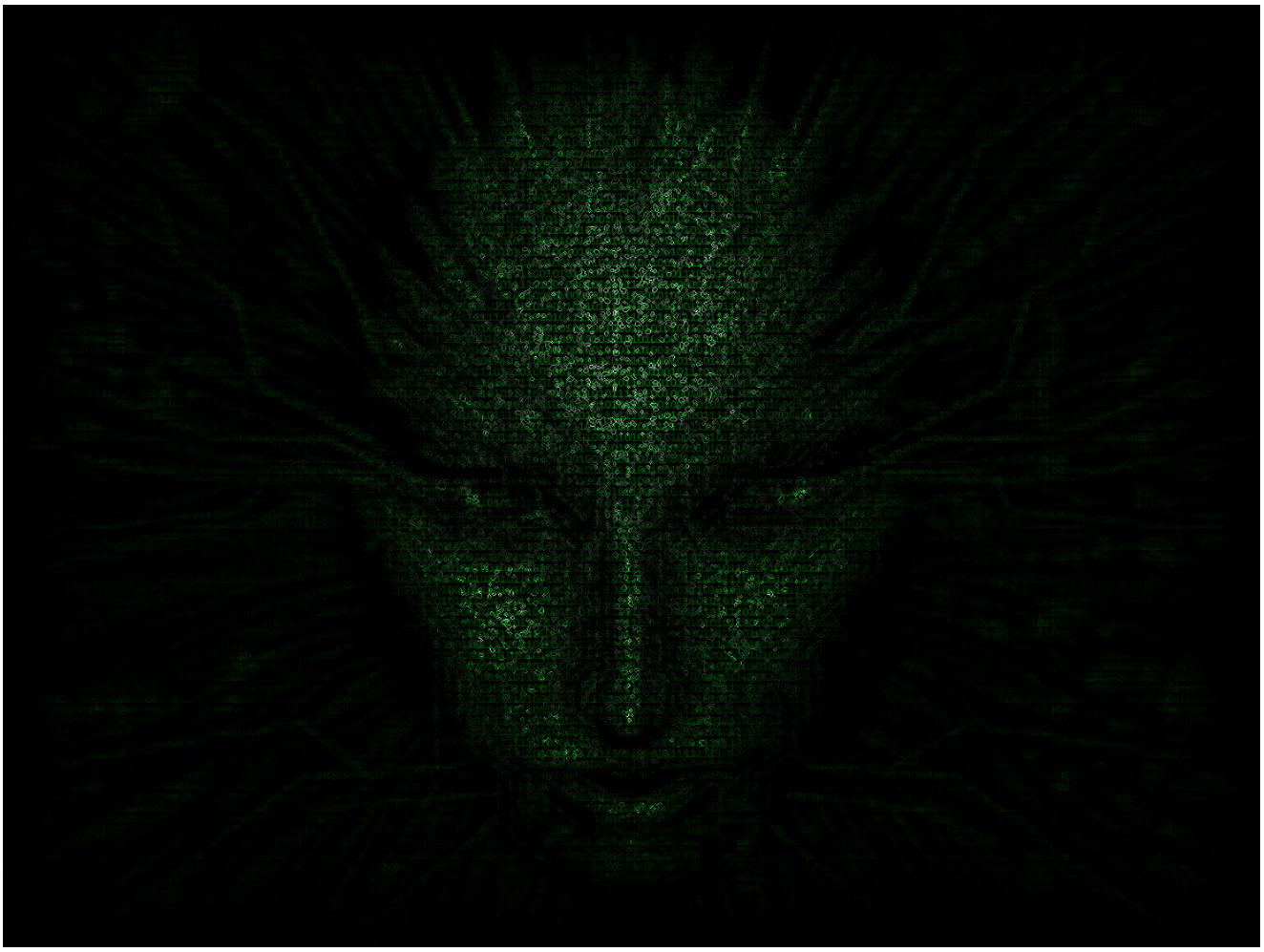
Статья LKM с рефлективным акцентом

xss.is/threads/54996



Привет, DaMaGeLaB!
Сегодня мы с тобой поговорим про ядро линукса. Непосредственно про Kernel dev. Вспомним с чего все начиналось, и посмотрим что мы имеем на сегодняшний день. Данная статья нацелена на то, что бы тот человек, который реально любит и хочет этим заниматься - получил нужный толчек в нужном направлении. То, с чего можно начать.
В то же время, это мой хлеб, так что я обещаю стараться себя сдерживать :3 Ато набегут с соседних тредов и будут делал бабки на паблик темах

0) Рефлексия
In computer science, reflective programming or reflection is the ability of a process to examine, introspect, and modify its own structure and behavior.
Отражение (рефлексия; холоним интроспекции, англ. reflection) — процесс, во время которого программа может отслеживать и модифицировать собственную структуру и

поведение во время выполнения.
Парадигма программирования, положенная в основу отражения, является одной из форм метапрограммирования[1] и называется рефлексивным программированием.

Метапрограммирование! Пнятненько?

Окей, если на серьезе -- писать "в рефлексии" куда сложнее, особенно в ядре. Зато весьма интереснее. Что бы ты примерно понимал, можно написать модуль собранный в 250кб, а в нем не единого
внешнего заголовочного файла. Все на чистом С.

Однако, нам не нужно полной рефлексии. Нам достаточно частичной. То есть у нас не стоит задача написать, скажем, малварь на Си, которая пишет себе подругу на АСМ в рантайме. Нет.
Наша задача -- аморфность. Быть как вода. Ты наливаешь воду в чашку - вода становится чашкой, в чайник - она становится чайником. Будь как вода, друг мой! (С)Баста.

1) Зачем это нужно?
Вопрос волне резонный. Ну, к примеру, лично меня не устраивает тот факт, что мой ядерный модуль должен быть скомпилирован под каждое ядро отдельно. Как и лбой другой ядерный модуль в линуксе в принципе.

Что бы продолжать разговор нужно вернуться немножко в прошлое. Когда-то давно, Торвальда Линуса укусил в зоопарке пингвин, а потом он написал Линукс. Вернее, его ядро. По скольку он был молод,
энергия била ключем, он часто допиливал в ядре всякие фичи. Структуры росли. Добавлялись новые типы данных, новые функции. И, разумеется, оффсеты съезжали с каждым новым релизом.
Я не буду разжевывать тут что такое оффсет, чтиво и так будет очень интересным но нихуя не понятным. Сори.

Так вот, и кароче решили что проблемы индейцев шерифа не волнуют. Хочешь? Собирай под каждое ядро, под каждый частный случай отдельно, персонально, свой билд (стаб) своего драйвера.
Линус гупнул кулаком по столу и все пошли собирать. И писать письма. В мейн-ланн. Уже 30 лет собирают. И пишут. И от релиза к релизу по 400 000 строк кода.
И что? Жизни же не хватит что бы все перечитать на бутлине (elexir bootlin) и все выучить, скажешь ты. И будешь прав. Не хватит. А кроме ваниллы есть еще дистрибьюторы. Дай бог здоровья Дебиану.
Смерть центосу. Опять х#йни напридумывали в убунте. Соптимизировали и заинлайнили половину системного апи. Нету символа. По этому в рефлексии.

2) Unix
Вообще, если взглянуть как рождается на свет LKM на фряхе - то можно задаться вопросом, типа, это что, какой-то специальный трюк? Типа как бы маркер? Или что?
C:

```
    /*
https://github.com/xcellerator/freebsd_kernel_hacking/blob/master/chapter1_lkm/1.3_hello_worlc
 */
    #include <sys/param.h>
    #include <sys/module.h>
    #include <sys/kernel.h>
    #include <sys/systm.h>

    /* The function called at load/unload. */
    static int load(struct module *module, int cmd, void *arg)
    {
        int error = 0;
        switch(cmd)
        {
            case MOD_LOAD:
                uprintf("Hello, world!\n");
                break;
            case MOD_UNLOAD:
                uprintf("Good-bye, cruel world!\n");
                break;
            default:
                error = EOPNOTSUPP;
                break;
        }
        return (error);
    }

    /* The second argument of DECLARE_MODULE. */
    static moduledata_t hello_mod = {
        "hello",    /* module name */
        load,        /* event handler */
        NULL        /* extra data */
    };

    /* use the DECLARE_MODULE macro so the kernel can link and register the module with
itself */
    /* DECLARE_MODULE(nane, data, sub, order) */
    /* name = module name */
    /* data = event handler function, see above */
    /* sub = system startup interace, we will always use SI_SUB_DRIVERS, used for registering
device drivers */
    /* order = priority of initialization, we will always use SI_ORDER_MIDDLE */
    DECLARE_MODULE(hello, hello_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
```

Ну да, так и есть. Типа маркер. Только не маркер, а "признак". Так вот, в Сишечке есть одно такое очень фундаментальное, бл#ть, понятие. Компилятор берет твой С код, и делает из него ассемблер.
Вот так вот прям все что было в 1 длиииинный файл на ассемблере. Потом дергает гас (gnu as). Транслятор уже транслирует этот линстинг из асма в опкоды. Получается сырой релоцируемый объектный файл `main.o`
Старина Линус тоже решил сильно не заморачиваться и унаследовал идею названий объектных файлов у Unix. И Получится LKM.ko. Это очень важный момент, на самом деле. Ты уже должен был догадаться,
что `.o` - это объектный релоцируемый файл, а `.ko` - это `kernel .o`. Так вот. За фундаментальное я пояснял. Получается что из объектного релоцируемого файла нужно сделать уже исполняемый бинарный файл.
Делает это линковщик. Типа в объектном файле вместо адресов функций будут порядковые номера или пустоты. Либо там могут быть оффсеты, тогда код будет PIC - позиционно независимый. Нам нужно как раз последнее.

Так вот, потом линковщик, значит, линкует этот объектный файлик в исполняемый и получается, к примеру, elf64 готовый к запуску. В юзерариуме действуют такие законы.
В ядре же линковщика нету.
Верне он есть. Он есть в самом ядре. То есть ты подаешь ядру сырой объектный файл `*.ko` и ядро уже слинковывает его в рантайме в систему. Называется "примешивается" (taints).

Итого: код на С превращается в код на АСМ. Код на асм превращается в опкоды. Опкоды отдаются ядру в качестве модуля.
Вывод: вся структурность написанного тобой на С, за исключением табуляций и отступов, сохраняется до последнего и имеет значение.
Это и есть то самое фундаментальное понятие, понимания которого так не хватает нынешнему поколению программистов, которые где-то слышали что `K&R - Язык С` - - это книга так-себе...
Хотя, в соседних тредах легко нарваться на специолистов по другому СИ, которые специально юзают психологические приемы. Типа "да K&R х#йня", и в ответ всякие наивные олухи типа меня "да то ты сам х#йня, а K&R не х#йня, смотри как умею"....

P.S. Кстати, на раннх версиях линукса вместо `*.ko` ядро хавало `*.o` в качестве драйвера. Так что, мы на верном пути.

3) GCC -> AS -> LD == Binutils

Я не люблю C++. Как и Торвальд. Не только потому, что он сложный или что он перегружен ненужным. Еще потому, что программист на C++ учит не сам язык, а стандарты. Причем ему не успеть выучить их,
так как они развиваются стремительнее, чем человек способен их изучать. Это ИМХО.

Отложим рассуждения о стандартах и поговорим о насущном.

Давай возьмем вот этот код, и попроуем его скомпилировать на, скажем, дебиан 10.

C:

```
#define NULL (void*)0x00
extern long __attribute__((__weak__)) malloc(long rdi, long rsi, long rdx, long rcx);
extern long __attribute__((__weak__)) printf(long rdi, ...);
extern long __attribute__((__weak__)) _exit(long rdi);
void _start() __attribute__((weak, alias("main")));
void main(){
    void *ptr = NULL;
    ptr = (void*)malloc(0x1000, 0,0,0);
    printf((long)"ptr = %#llx\n", (unsigned long long)ptr);
    _exit(0x00);
}
```

Слегка не привычно, не так-ли? Ну у меня компилируется без единого варнинга.
-В смысле? Ты ебанутый? (подумал читатель)

А вот и нет.
`cc -fno-builtin -fno-builtin-printf -nostdlib -nostdinc -S relo_obj_example.c -o relo_obj_example.S`
`cat relo_obj_example.S`
Code:

```
    .file    "relo_obj_example.c"
    .text
    .section    .rodata
.LC0:
    .string    "ptr = %#llx\n"
    .text
    .globl    main
    .type    main, @function
main:
.LFB0:
    .cfi_startproc
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movq    $0, -8(%rbp)
    movl    $0, %ecx
    movl    $0, %edx
    movl    $0, %esi
    movl    $4096, %edi
    call    malloc@PLT
    movq    %rax, -8(%rbp)
    movq    -8(%rbp), %rax
    leaq    .LC0(%rip), %rdx
    movq    %rax, %rsi
    movq    %rdx, %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    $0, %edi
    call    _exit@PLT
    nop
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size    main, .-main
    .weak    _start
    .set    _start,main
    .weak    _exit
    .weak    printf
    .weak    malloc
    .ident  "GCC: (Debian b00st3r)"
    .section    .note.GNU-stack,"",@progbits
```

Оукей, теперь это надо транслировать в машинный код. Зовем GAS.
`as relo_obj_example.S -o relo_obj_example.o`
Без единого варнинга я сказал!

`file relo_obj_example.o`
`relo_obj_example.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped`

Ре-ло-ка-та-бле. Ни-чтяк.
Кстати, нот-стриппед, типа не оголенный, это нормально для релоцируемых объектов, т.к. если стрипнуть релоцируемый объект он не сможет слинковаться, т.к. потеряется инфа о релокациях.
Но, можно стрипнуть только ту инфу, которая не несет критической важности для дальнейшего процесса линковки.
`strip --strip-unneeded ./relo_obj_example.o`
`readelf -aW relo_obj_example.o`
Code:

```
        Раздел перемещения '.rela.text' по смещению 0x1e8 содержится 4 элемента:
            Смещение           Инфо             Тип                 Значение симв.  Имя
символа + Addend
        0000000000000025  0000000400000004 R_X86_64_PLT32         0000000000000000 malloc - 4
        0000000000000034  0000000200000002 R_X86_64_PC32          0000000000000000 .rodata -
4
        0000000000000044  0000000500000004 R_X86_64_PLT32         0000000000000000 printf - 4
        000000000000004e  0000000600000004 R_X86_64_PLT32         0000000000000000 _exit - 4

        Раздел перемещения '.rela.eh_frame' по смещению 0x248 содержится 1 элемент:
            Смещение           Инфо             Тип                 Значение симв.  Имя
символа + Addend
        0000000000000020  0000000100000002 R_X86_64_PC32          0000000000000000 .text + 0
        No processor specific unwind information to decode

        Таблица символов «.symtab» содержит 8 элементов:
           Чис:    Знач            Разм Тип     Связ   Vis      Индекс имени
             0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
             1: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
             2: 0000000000000000     0 SECTION LOCAL  DEFAULT    5 .rodata
             3: 0000000000000000    85 FUNC    GLOBAL DEFAULT    1 main
             4: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND malloc
             5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND printf
             6: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _exit
             7: 0000000000000000    85 FUNC    WEAK   DEFAULT    1 _start
```

Названия релоков прикольные, любых других в ядре нужно избегать. В ядре вообще их нужно избегать в принципе любых.
`objdump -Mintel -d relo_obj_example.o`
Code:

```
relo_obj_example.o:     формат файла elf64-x86-64
Дизассемблирование раздела .text:
0000000000000000 <main>:
   0:    55                       push   rbp
   1:    48 89 e5                 mov    rbp,rsp
   4:    48 83 ec 10              sub    rsp,0x10
   8:    48 c7 45 f8 00 00 00     mov    QWORD PTR [rbp-0x8],0x0
   f:    00
  10:    b9 00 00 00 00           mov    ecx,0x0
  15:    ba 00 00 00 00           mov    edx,0x0
  1a:    be 00 00 00 00           mov    esi,0x0
  1f:    bf 00 10 00 00           mov    edi,0x1000
  24:    e8 00 00 00 00           call   29 <main+0x29>
  29:    48 89 45 f8              mov    QWORD PTR [rbp-0x8],rax
  2d:    48 8b 45 f8              mov    rax,QWORD PTR [rbp-0x8]
  31:    48 8d 15 00 00 00 00     lea    rdx,[rip+0x0]        # 38 <main+0x38>
  38:    48 89 c6                 mov    rsi,rax
  3b:    48 89 d7                 mov    rdi,rdx
  3e:    b8 00 00 00 00           mov    eax,0x0
  43:    e8 00 00 00 00           call   48 <main+0x48>
  48:    bf 00 00 00 00           mov    edi,0x0
  4d:    e8 00 00 00 00           call   52 <main+0x52>
  52:    90                       nop
  53:    c9                       leave
  54:    c3                       ret
```

Обрати внимание на строчку под номером 0x24. Видишь, там после опкода мнемоники call пустота в 4 байта? Это и есть слот для релока. Сюда будет записан адрес вызова после линковки.
Осталось слинковаться.
`ld -melf_x86_64 -lc -o relo_obj_example.elf ./relo_obj_example.o -I/lib64/ld-linux-x86-64.so.2`
`./relo_obj_example.elf`
`ptr = 0xa812a0`

Как видишь, все работает просто отлично.
Внимательный читатель обратил внимание на слово binutils в заголовке этой главы. В сумме эти и другие утилиты входят в состав пакета инструментов под говорящим названием binutils.

4) Kernel MAKE

Мы с тобой уже говорили о том, на сколько сложна и объемна система сборки ядра? Ну, процесс сборки исполняемого файлика для пользовательского пространства тоже довольно объемен, если делать это руками.
Все что выше можно бы было заменить на одну единственную строчку `cc

./relo_obj_example.c -o ./relo_obj_example.elf`. Оно скомпилировалось бы и работало, пару раз покрыв матом программиста за такие топорные объявления.
Можно бы было сделать и так, но тогда бы мы так и не поняли как это все работает.

А в ядре не 11 строчек кода. Их там сотни тысяч. Сорцы в сжатом виде занимают более 100 мегабайт. 100+ мегабайт сжатого печатного текста исходного кода!!! Уму не постижимо как это все собрать в 1 большой файл, согласись.
По этому сообщество использует `make`. И великое множество всяких скриптов и `spec` файлов для линковки, перелинковки и так далее и тому подобное.
Мы же вообще отказываемся от всего ненужного нам. Т.к. чем проще - тем надежнее.
Мы линковщик использовать вообще не будем, а от `make` нам вынужденно придется отказаться, т.к. он просто не даст на сделать то, что нужно.

Но для начала, давай попробуем с тобой посмотреть как же все таки оно работает, хоть примерно, что бы понять почему так, а не иначе.

`cat example.c`
C:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

extern void foo(void);
__asm__("jmp foo");

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Example");
MODULE_DESCRIPTION("Basic");
MODULE_VERSION("0.01");


static int __init example_init(void)
{
    printk(KERN_INFO "Hello, World!\n");
    return 0;
}


static void __exit example_exit(void)
{
    printk(KERN_INFO "Goodbye, World!\n");
}


module_init(example_init);
module_exit(example_exit);
void foo(void){;}
```

`cat Makefile`
Code:

```
obj-m += example.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

test:
    sudo dmesg -C
    ls -allh ./example.ko
    sudo insmod example.ko
    sudo rmmod example.ko
    dmesg
```

Пробуем:

`make && make test`
Code:

```
make -C /lib/modules/5.10.24/build M=/C/LKM_reflective_2021_xss.is/0_4 modules
make[1]: вход в каталог «/usr/src/linux-5.10.24»
  CC [M]  /C/LKM_reflective_2021_xss.is/0_4/example.o
/C/LKM_reflective_2021_xss.is/0_4/example.o: warning: objtool: .text+0x0: unreachable
instruction
  MODPOST /C/LKM_reflective_2021_xss.is/0_4/Module.symvers
  CC [M]  /C/LKM_reflective_2021_xss.is/0_4/example.mod.o
  LD [M]  /C/LKM_reflective_2021_xss.is/0_4/example.ko
make[1]: выход из каталога «/usr/src/linux-5.10.24»
sudo dmesg -C
ls -allh ./*.ko
-rw-r--r-- 1 root root 59K авг  7 22:45 ./example.ko
sudo insmod example.ko
sudo rmmod example.ko
dmesg
[1053842.152751] Hello, World!
[1053842.188853] Goodbye, World!
```

59КБ!!! Казалось бы! Если интересно что там происходит - можешь повторить передав `make` в аргументах `V=1`, а я продолжу.
`objdump -Mintel -d example.ko `

Code:

```
example.ko:     формат файла elf64-x86-64

Дизассемблирование раздела .text:

0000000000000000 <foo-0x10>:
   0:    eb 0e                   jmp    10 <foo>
   2:    66 66 2e 0f 1f 84 00    data16 cs nop WORD PTR [rax+rax*1+0x0]
   9:    00 00 00 00
   d:    0f 1f 00                nop    DWORD PTR [rax]

0000000000000010 <foo>:
  10:    e8 00 00 00 00          call   15 <foo+0x5>
  15:    c3                      ret

Дизассемблирование раздела .init.text:

0000000000000000 <init_module>:
   0:    e8 00 00 00 00          call   5 <init_module+0x5>
   5:    48 c7 c7 00 00 00 00    mov    rdi,0x0
   c:    e8 00 00 00 00          call   11 <init_module+0x11>
  11:    31 c0                   xor    eax,eax
  13:    c3                      ret

Дизассемблирование раздела .exit.text:

0000000000000000 <cleanup_module>:
   0:    48 c7 c7 00 00 00 00    mov    rdi,0x0
   7:    e9 00 00 00 00          jmp    c <cleanup_module+0xc>
```

Обрати внимание на джампы. Строчка под номером 0x00 -- джамп на строчку под номером 0x10 -- ей соответствует наша строчка на C `__asm__("jmp foo");`
Однако, тут отработал какой-то из безчисленных скриптов и спеков любезно выполняемый через Make -- и функция foo расположилась не в конце файла, как мы написали, а над init_module.

Более того, в папке проекта появилось куча непонятных файликов:
Code:

```
итого 236K
drwxr-xr-x 2 root root 4,0K авг  7 22:48 .
drwxr-xr-x 6 root root 4,0K авг  7 22:48 ..
-rw-r--r-- 1 c4t  c4t   473 июл 14 07:30 example.c
-rw-r--r-- 1 root root  59K авг  7 22:48 example.ko
-rw-r--r-- 1 root root  258 авг  7 22:48 .example.ko.cmd
-rw-r--r-- 1 root root   45 авг  7 22:48 example.mod
-rw-r--r-- 1 root root  769 авг  7 22:48 example.mod.c
-rw-r--r-- 1 root root  163 авг  7 22:48 .example.mod.cmd
-rw-r--r-- 1 root root  47K авг  7 22:48 example.mod.o
-rw-r--r-- 1 root root  30K авг  7 22:48 .example.mod.o.cmd
-rw-r--r-- 1 root root  14K авг  7 22:48 example.o
-rw-r--r-- 1 root root  29K авг  7 22:48 .example.o.cmd
-rw-r--r-- 1 c4t  c4t   250 авг  7 22:44 Makefile
-rw-r--r-- 1 root root   45 авг  7 22:48 modules.order
-rw-r--r-- 1 root root  185 авг  7 22:48 .modules.order.cmd
-rw-r--r-- 1 root root    0 авг  7 22:48 Module.symvers
-rw-r--r-- 1 root root  228 авг  7 22:48 .Module.symvers.cmd
```

Среди них есть один весьма любопытный:
`bcat example.mod.c -`

C:

File: example.mod.c

```
 1  #include <linux/module.h>
 2  #define INCLUDE_VERMAGIC
 3  #include <linux/build-salt.h>
 4  #include <linux/vermagic.h>
 5  #include <linux/compiler.h>
 6
 7  BUILD_SALT;
 8
 9  MODULE_INFO(vermagic, VERMAGIC_STRING);
10  MODULE_INFO(name, KBUILD_MODNAME);
11
12  __visible struct module __this_module
13  __section(".gnu.linkonce.this_module") = {
14      .name = KBUILD_MODNAME,
15      .init = init_module,
16  #ifdef CONFIG_MODULE_UNLOAD
17      .exit = cleanup_module,
18  #endif
19      .arch = MODULE_ARCH_INIT,
20  };
21
22  #ifdef CONFIG_RETPOLINE
23  MODULE_INFO(retpoline, "Y");
24  #endif
25
26  static const struct modversion_info ____versions[]
27  __used __section("__versions") = {
28      { 0x29e7595d, "module_layout" },
29      { 0x85750110, "printk" },
30      { 0x123b6dbb, "__fentry__" },
31  };
32
33  MODULE_INFO(depends, "");
34
35
36  MODULE_INFO(srcversion, "1A5934567907043DF7DB963");
```

Вот оно! Весь цимес! Помнишь, на фряхе мы наблюдали странные признаки в исходнике ядерного драйвера? Они есть не только на Unix, но и на Linux. Просто скрыты от посторонних глаз. Как и очень очень много вещей в ядре ;3

В двух словах:
Строчка под номером 2 -- это строчка версии ядра. По ней идет сравнение перед тем как подмешать в ядро драйвер. Особенности можешь узнать нажав `man init_module`

Строчка под номером 7 -- так же, дефайн из сорцев ядра, под которые идет сборка.
Строчка под номером 9, 10, 23, 33, 36 -- это макросы, которые дополняют и заполняют структуру struct_module.
Строчка под номером 26 -- структура modversion_info, вернее целый символ ____versions из секции __versions. О них поговорим позже. Но, забегая вперед, скажу что отличительной чертой таких вещей является то, что
эти структуры будут обрабатываться ядерным линковщиком раньше, чем выполнение дойдет до нашего entry point. То бишь мы не властны там что-то выполнить или поменять, и это проблема.
Строчка под номером 12 -- символ __this_module типа struct module (читай как формы. есть круг, есть квадрат, а есть struct module). Причем это очень важный момент. Дело в том, что у каждого ядра размеры этой структуры
будут несколько отличаться. Там что-то добавили, там убрали, там с другими флагами собрали ядро. Ну и как следствие отличаться будут оффсеты. А эта структурка, будучи объектом холодным, парсится ядроом ДО
того, как ядро вызовет наш условный main().
Строчка под номером 15 -- это и есть символ, который должент быть вызван после релокаций. Это и есть точка входа в наш модуль ядром. И у него есть свой оффсет.
В смысле у частного случая ядра будет свой оффсет, по которому оно будет искать ентри поинт в нашем `*.ko` И он очень даже разнится на разных ядрах.

Становтся жарковато, не так-ли? :3

5) GCC vs. MAKE

И так, удобный и автоматизированный make -- это явно не наш случай. Мы не хотим всей этой елочной мишуры, мы хотим свободы.
Как же это сделать?

Я искренне верю, что среди всех людей, которые прочитают эту статью найдется хотя бы один человек который сейчас в уме правильно ответит на этот вопрос... Значит, я писал это не зря :3

Теперь по пунктам. Для начала - я буду писать и компилировать ядерный модуль под ядро своей виртуалки. Она крутится на 4.19.160. А работаю я на десктопном 5.10+. Но, учитывая что у нас нету не одного инклуда,
это не имеет особого значения. Единственное, что отделяет нас от того, что бы эта штучка заработала на разных ядрах -- это строчка №60 -- поле vermagic. Но человек вдумчиво прочитавший эту статью с легкостью
решит этот вопрос самостоятельно, я в этом твердо убежден.

Строки 1 - 51 -- это мне лично так захотелось повыделываться, и, за одно, сэкономить тебе пару месяцев времени. Начинать всегда не просто, потому, что не знаешь к чему конкретно нужно идти. Не на что опереться.
№47 -- это не баг, это фича. Я специально посмотрел адрес функции sys_ni_syscall на ядре вирты. Потом в отладчике я накинул туда бряк. Что бы выполнение остановилось в этом месте. Ловкость лап, и никакого мошенничества
№75 -- это место, с которого я хочу просмотреть работу (или баг, верней, его причину) модуля под отладчиком. Если ты не знаешь как пользоваться отладчиком или как собрать ядро пригодное для отладки -- на xakep.ru писали
недавно годный мануал. Он есть и у нас в разделе эксплойтинга. Рекомендую ознакомиться
№71 -- это, дружище, тебе мой подгон. Пригодится, если будешь продолжать этим заниматься.
Не все символы дадуться так просто, без боя :3 Этот момент заслуживает отдельного пояснения чуть ниже.
№91 -- Обнажённая функция. Не путать с голым бинарником.
Важным моментом является то, что даже если ты напишешь в ней `return 0x00;` оно не выполнится. Хотя ты можешь писать там сишный код в пределах разумного.
Я по коду в комментариях написал три основных пункта. Еще про то, что компилятор вставляет ud2 поговорим ниже. А что такое volatile -- это что бы компилятор не пытался ничего наоптимизировать.
Типа память может измениться в любое время, из вне, независимо, так что нужно атомарненько взять ее значение перед использованием.
№92 -- работа с регистром. В принципе можно использовать не только в __naked, а где угодно, но после этого, если не в __naked, нужно юзать барьеры памяти, которые говорят что память могла измениться,
или регистров, т.к. регистры могли быть разрушены. Делается такое через инлайн асм. Погуглишь если нужно будет.
№68 и №89 -- это просто тип данных что бы брать от него typedef и объявлять потом указатели типа `func_t` -- мой рефлективный тип функций (не канает для __naked)
№100 и ниже -- это наша холодная структура struct_module. Я так для себя прикинул, что на дебианах юзается оффсет 150, а на убунтах 175. Ну и все. Написал. Работает.
Указатель на выход из функции не обязателен. Но у нас он есть. Хотя наш модуль не загрузится, т.к. вернет отрицательный код. Одноко, мы вополним свой код в ядерном контексте, чего нам более чем предостаточно!
Остально чисто под оффсеты. name на всех ядрах идет со смещением в 0x18 от начала структуры. Так что хардкодим.

Вроде бы все? С типами данных и классами данных разберешься и так, если еще не разобрался. Имхо пример более чем исчерпывающий.

Ааа, еще нужно скомпилировать и собрать же все это! Точно.

Code:

```
    rm -f ./*.ko 2&>/dev/nill ; \
    cc \
        -mcmodel=kernel \
        -mtune=generic \
        -mno-red-zone \
        -fno-plt \
        -fpie \
        -mno-80387 \
        -mno-fp-ret-in-387 \
        -fPIC -fno-PIE -Wno-attributes -Wno-int-conversion -nostdlib -nostartfiles -
nodefaultlibs \
        -m64 -fno-gimple \
        -Xassembler -mrelax-relocations=no \
        -mpreferred-stack-boundary=4 \
        -O2 \
        -std=gnu1x \
        reflective_example.c -c -o reflective_example.ko
```

`./make.sh && ls -allh ./*.ko`
`-rw-r--r-- 1 root root 4,4K авг 8 01:47 ./reflective_example.ko`

Чуть по компактнее, чем через make
Кстати, еще одна фишка тебе, дорогой читатель. Для объектных файлов gcc не добавляет отладочную инфу. Что бы он это сделал нужно дописать во флагах `-g3 -ggdb3` и желательно вырубить оптимизацию.
В таком случае брякнувшись на ините ты сможешь глянуть по какому адресу загружен твой модуль в данны момент с помощью команды отладчику `lx-lsmod`
После чего применить символы для адреса памяти с помощью команды `add-symbol-file reflective_example.ko 0xffffffff11223344`, оффсет, логично, берем из предыдущей команды.

С отладочными символами выкрученными до упора у меня получилось:
`-rw-r--r-- 1 root root 34K авг 8 01:52 reflective_example.ko`
что все равно почти в 2 раза меньше версии из под make

`readelf -aW reflective_example.ko`

Code:

```
Заголовок ELF:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Класс:                             ELF64
  Данные:                            дополнение до 2, от младшего к старшему
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  Версия ABI:                        0
  Тип:                               REL (Перемещаемый файл)
  Машина:                            Advanced Micro Devices X86-64
  Версия:                            0x1
  Адрес точки входа:                 0x0
  Начало заголовков программы:       0 (байт в файле)
  Начало заголовков раздела:         3240 (байт в файле)
  Флаги:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         21
  Section header string table index: 20

Заголовки разделов:
  [Нм] Имя              Тип             Адрес             Смещ   Разм   ES Флг Лк Инф Al
  [ 0]                  NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text            PROGBITS        0000000000000000 000040 00009e 00  AX  0   0  1
  [ 2] .rela.text       RELA            0000000000000000 000a48 000030 18   I 18   1  8
  [ 3] .data            PROGBITS        0000000000000000 0000e0 000008 00  WA  0   0  8
  [ 4] .rela.data       RELA            0000000000000000 000a78 000018 18   I 18   3  8
  [ 5] .bss             NOBITS          0000000000000000 0000e8 000000 00  WA  0   0  1
  [ 6] .modinfo         PROGBITS        0000000000000000 000100 00014e 00   A  0   0 32
  [ 7] .rodata          PROGBITS        0000000000000000 000250 000030 00   A  0   0  8
  [ 8] .init.text       PROGBITS        0000000000000000 000280 000030 00  AX  0   0  1
  [ 9] .rela.init.text  RELA            0000000000000000 000a90 000048 18   I 18   8  8
  [10] .exit.text       PROGBITS        0000000000000000 0002b0 00001c 00  AX  0   0  1
  [11] .rela.exit.text  RELA            0000000000000000 000ad8 000030 18   I 18  10  8
  [12] .gnu.linkonce.this_module PROGBITS        0000000000000000 0002e0 000338 00  WA  0
0 32
  [13] .rela.gnu.linkonce.this_module RELA            0000000000000000 000b08 000048 18
I 18  12  8
  [14] .comment         PROGBITS        0000000000000000 000618 00001f 01  MS  0   0  1
  [15] .note.GNU-stack  PROGBITS        0000000000000000 000637 000000 00      0   0  1
  [16] .eh_frame        PROGBITS        0000000000000000 000638 0000f0 00   A  0   0  8
  [17] .rela.eh_frame   RELA            0000000000000000 000b50 0000a8 18   I 18  16  8
  [18] .symtab          SYMTAB          0000000000000000 000728 000210 18     19   6  8
  [19] .strtab          STRTAB          0000000000000000 000938 00010b 00      0   0  1
  [20] .shstrtab        STRTAB          0000000000000000 000bf8 0000ae 00      0   0  1
Обозначения флагов:
  W (запись), A (назнач), X (исполняемый), M (слияние), S (строки),
  I (инфо), L (порядок ссылок), O (требуется дополнительная работа ОС),
  G (группа), T (TLS), C (сжат), x (неизвестно), o (специфич. для ОС),
  E (исключён),
```

```
  D (mbind), l (большой), p (processor specific)

    В этом файле нет групп разделов.

    В этом файле нет заголовков программы.

    В этом файле нет динамического раздела.

    Раздел перемещения '.rela.text' по смещению 0xa48 содержится 2 элемента:
        Смещение           Инфо             Тип                Значение симв.  Имя символа +
Addend
    000000000000008c  0000001000000004 R_X86_64_PLT32         0000000000000000 example_init -
4
    0000000000000097  0000001200000004 R_X86_64_PLT32         0000000000000000 example_exit -
4

    Раздел перемещения '.rela.data' по смещению 0xa78 содержится 1 элемент:
        Смещение           Инфо             Тип                Значение симв.  Имя символа +
Addend
    0000000000000000  0000000f00000001 R_X86_64_64            0000000000000000 printk + 0

    Раздел перемещения '.rela.init.text' по смещению 0xa90 содержится 3 элемента:
        Смещение           Инфо             Тип                Значение симв.  Имя символа +
Addend
    0000000000000010  0000000200000002 R_X86_64_PC32          0000000000000000 .data - 4
    0000000000000017  000000030000000b R_X86_64_32S           0000000000000000 .rodata + 0
    000000000000002a  0000000100000002 R_X86_64_PC32          0000000000000000 .text + 69

    Раздел перемещения '.rela.exit.text' по смещению 0xad8 содержится 2 элемента:
        Смещение           Инфо             Тип                Значение симв.  Имя символа +
Addend
    0000000000000007  0000000200000002 R_X86_64_PC32          0000000000000000 .data - 4
    000000000000000e  000000030000000b R_X86_64_32S           0000000000000000 .rodata + 1f

    Раздел перемещения '.rela.gnu.linkonce.this_module' по смещению 0xb08 содержится 3
элемента:
        Смещение           Инфо             Тип                Значение симв.  Имя символа +
Addend
    0000000000000150  0000001300000001 R_X86_64_64            0000000000000087 init_module +
0
    0000000000000178  0000001100000001 R_X86_64_64            0000000000000000 entry + 0
    0000000000000330  0000001400000001 R_X86_64_64            0000000000000092 cleanup_module
+ 0

    Раздел перемещения '.rela.eh_frame' по смещению 0xb50 содержится 7 элементов:
        Смещение           Инфо             Тип                Значение симв.  Имя символа +
Addend
    0000000000000020  0000000400000002 R_X86_64_PC32          0000000000000000 .init.text + 0
    0000000000000040  0000000100000002 R_X86_64_PC32          0000000000000000 .text + 2
    0000000000000060  0000000500000002 R_X86_64_PC32          0000000000000000 .exit.text + 0
    0000000000000080  0000000100000002 R_X86_64_PC32          0000000000000000 .text + d
```

```
00000000000000a0  0000000100000002 R_X86_64_PC32          0000000000000000 .text + 6d
00000000000000b4  0000000100000002 R_X86_64_PC32          0000000000000000 .text + 87
00000000000000d4  0000000100000002 R_X86_64_PC32          0000000000000000 .text + 92
No processor specific unwind information to decode

Таблица символов «.symtab» содержит 22 элемента:
   Чис:    Знач           Разм Тип     Связ   Vis      Индекс имени
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    3 .data
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    7 .rodata
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    8 .init.text
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT   10 .exit.text
     6: 0000000000000002    11 FUNC    GLOBAL DEFAULT    1 foo
     7: 0000000000000000    12 OBJECT  GLOBAL DEFAULT    6 __UNIQUE_ID_license53
     8: 0000000000000020    38 OBJECT  GLOBAL DEFAULT    6 __UNIQUE_ID_author54
     9: 0000000000000060    63 OBJECT  GLOBAL DEFAULT    6 __UNIQUE_ID_description55
    10: 00000000000000a0    13 OBJECT  GLOBAL DEFAULT    6 __UNIQUE_ID_version56
    11: 00000000000000b0    12 OBJECT  GLOBAL DEFAULT    6 __UNIQUE_ID_retpoline57
    12: 00000000000000c0    24 OBJECT  GLOBAL DEFAULT    6 __UNIQUE_ID_name58
    13: 00000000000000e0    35 OBJECT  GLOBAL DEFAULT    6 __UNIQUE_ID_srcversion59
    14: 0000000000000120    46 OBJECT  GLOBAL DEFAULT    6 __UNIQUE_ID_vermagic60
    15: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND printk
    16: 0000000000000000    48 FUNC    GLOBAL DEFAULT    8 example_init
    17: 0000000000000000    48 FUNC    WEAK   DEFAULT    8 entry
    18: 0000000000000000    28 FUNC    GLOBAL DEFAULT   10 example_exit
    19: 0000000000000087    11 FUNC    GLOBAL DEFAULT    1 init_module
    20: 0000000000000092    12 FUNC    GLOBAL DEFAULT    1 cleanup_module
    21: 0000000000000000   824 OBJECT  GLOBAL DEFAULT   12 __this_module
```

Не плохо. Конченных релокаций не наблюдаются. Они могу появиться, когда пишешь конструкции типа `int a = 1, *aptr = &a;` Вообще ты должен понимать что все подобные объявления с инициализацией -- это лишняя работа для линковщика.
Линковщик, его логику, чуть поменяли где-то на 4.4.* релизах и пропатчили binutils. У меня это было проблемой до такой степени, что я пропатчил себе `gnu as`, но это уже совсем другая история :3
Тут же все наши секции которые мы объявляли через аттрибуты. Кстати, аттрибуты -- это прямое окно в ассемблер. Очень серьезный хак на самом деле. Если не забуду напишу про него чуть ниже. Давай глянем Листинг что у нас вышло.

`objdump -Mintel -d reflective_example.ko`

Code:

```
reflective_example.ko:     формат файла elf64-x86-64


Дизассемблирование раздела .text:

0000000000000000 <sexi-0x10>:
   0:    eb 3e                   jmp    40 <foo>
   2:    66 66 2e 0f 1f 84 00    data16 cs nop WORD PTR [rax+rax*1+0x0]
   9:    00 00 00 00
   d:    0f 1f 00                nop    DWORD PTR [rax]

0000000000000010 <sexi>:
  10:    48 89 fa                mov    rdx,rdi
  13:    b8 00 00 00 00          mov    eax,0x0
  18:    48 89 d0                mov    rax,rdx
  1b:    48 69 c0 39 05 00 00    imul   rax,rax,0x539
  22:    c3                      ret
  23:    b8 ef be ad de          mov    eax,0xdeadbeef
  28:    0f 0b                   ud2
  2a:    66 0f 1f 44 00 00       nop    WORD PTR [rax+rax*1+0x0]

0000000000000030 <cleanup_module>:
  30:    48 c7 c7 00 00 00 00    mov    rdi,0x0
  37:    31 c0                   xor    eax,eax
  39:    ff 25 00 00 00 00       jmp    QWORD PTR [rip+0x0]        # 3f
<cleanup_module+0xf>
  3f:    90                      nop

0000000000000040 <foo>:
  40:    c3                      ret
  41:    66 66 2e 0f 1f 84 00    data16 cs nop WORD PTR [rax+rax*1+0x0]
  48:    00 00 00 00
  4c:    0f 1f 40 00             nop    DWORD PTR [rax+0x0]

0000000000000050 <init_module>:
  50:    48 83 ec 08             sub    rsp,0x8
  54:    48 c7 c0 40 61 0a 81    mov    rax,0xffffffff810a6140
  5b:    ff d0                   call   rax
  5d:    48 c7 c7 00 00 00 00    mov    rdi,0x0
  64:    31 c0                   xor    eax,eax
  66:    ff 15 00 00 00 00       call   QWORD PTR [rip+0x0]        # 6c
<init_module+0x1c>
  6c:    48 c7 c7 ff ff ff ff    mov    rdi,0xffffffffffffffff
  73:    e8 98 ff ff ff          call   10 <sexi>
  78:    48 83 c4 08             add    rsp,0x8
  7c:    c3                      ret

Дизассемблирование раздела .exit.text:

0000000000000000 <example_exit>:
   0:    48 c7 c7 00 00 00 00    mov    rdi,0x0
```

```
       7:    31 c0                   xor    eax,eax
       9:    ff 25 00 00 00 00       jmp    QWORD PTR [rip+0x0]        # f
<example_exit+0xf>

    Дизассемблирование раздела .init.text:

    0000000000000000 <example_init>:
       0:    48 83 ec 08             sub    rsp,0x8
       4:    48 c7 c0 40 61 0a 81    mov    rax,0xffffffff810a6140
       b:    ff d0                   call   rax
       d:    48 c7 c7 00 00 00 00    mov    rdi,0x0
      14:    31 c0                   xor    eax,eax
      16:    ff 15 00 00 00 00       call   QWORD PTR [rip+0x0]        # 1c
<example_init+0x1c>
      1c:    48 c7 c7 ff ff ff ff    mov    rdi,0xffffffffffffffff
      23:    e8 00 00 00 00          call   28 <example_init+0x28>
      28:    48 83 c4 08             add    rsp,0x8
      2c:    c3                      ret
```

Так вроде без нареканий. В example_init на 0x04 строчке хардкод нашего бряка. Он не будет релочиться, т.к. адрес захардкожен. Остальные заполнены нулями. Пока что. Странно то, что init_module дублирует example_init.
Ну ничего, пусть уже будет как есть. Лень скрины переделывать

И так, переходим к тестам.

На вирте №1 отработало. Потом мы добавили символы и насладились нашей работой в полной мере.
На вирте №2 тоже отработало. Только кое-кто проебал убрать захардкоженный брейкпоинт. Но я не буду уже переделывать скриншоты, суть передеана :3

6) Эпилог

Кароче, мне надоело вот это вот руками подбирать ключи к `insmod`, к тому-же его может и не быть. Например в бизибоксе его нету.

По этому, давай напишем программу, которая это будет делать за нас.

Написали. Ничтяк. Какие мы молодцы. Расписывать и пояснять тут особо нечего, по коду все и так предельно ясно. Теперь собираем.

Code:

```
xxd -i ./../0_5/reflective_example.ko > ./lkm.raw
cc inj3ct0r.c -o inj3ct0r.elf
strip ./inj3ct0r.elf
ls -allg ./inj3ct0r.elf && \
echo "Done."
```

Во. Теперь нормально. Первой системой был последний обновленный в хлам 11 дебиан буллсай. Второй системой был Oracle Unbreakable Linux с самыми последними апдейтами 2021 года.
Более того, в 4.17 поменялось соглашение о вызовах, если до 4.17 во все вызовы передавались все аргументы, то после 4.17 во все вызовы передается один аргумент в $rdi -- struct pt_regs.

Вот как-то вот так это все и работает.

7) Last word

Есть такая штука LKRG. Это тоже LKM только для защиты. Он каждые N времени чекает хешсуммы ядерных сегментов в памяти, тех, которые protected.
Я предлагаю надрать ему задницу :3

Что мы сделаем:

С первого взгляда может быть не ясно, но там все просто. 2 экстерна, 2 объявления с инициализациями (так делать не надо без патченного бинютилс).
Потом инклуд. Когда инклудится `*.c` в `*.c` -- получается один большой `*.c` так что видно static функции. Зато static функции не видно потом в kallsyms :3 Ну и у них (у статик) еще есть пару основных свойств, это побочное.
Это я если что сейчас про выхлоп colordiff писал.

А если говорить о сорцах `fuck_lkrg.c` -- то там вообще ничего сложного, просто нужно знать очень много особенностей. Одна из них, к примеру, stop_machine. Она паркует все ядерные потоки в определенном месте,
и передает управление на колл-бек. Получается что зайдя в коллбек мы единственный поток, который выполняется на системе.
Далее, что бы никому не показалось мало, мы аппаратно вешаем на себя защиту от прерываний -- строчка 38. После работы отпускаем этот режим -- строчка 42.
Строчки 7-16 -- аналог cr0 регистра, которого может не быть, и который сущее палево, что для гипервизора что для систем мониторинга. Кароче cr0 -- это плохо.
Пнятненько?

Ну и типа патчим память -- строчки 60 - 66. В названиях переменных отображается смысл.
За то, как работает врапер системного вызова, или почему разыменовав первое поле из таблицы системных вызовов я уверен что это sys_read() объяснять, надеюсь, не нужно?

Что из этого получится:

Я нарочно вернул положительное значение на выходе из инита. По двум причинам - 1 это эффектно . 2 - поскольку теперь моя функция новый системный вызов sys_read() -- то ей следует присутствовать в ядре.
Если бы мой инит вернул -1337 -- то ядро бы выгрузило все ресурсы модуля, который сообщил что не может работать по не известной причине. Так что в качествы быстрого ворк-эраунда я поменял - на +

8) Gre33t3n5s to:
XSS.is
club1337
signed volatile static long long int register asm("rax") __from_C4T_w1th_l0v3_:3 ;
Last edited: Aug 7, 2021