

Лекция 19

ПРО ПРОВЕРКУ ЦЕЛОСТНОСТИ И
ОБФУСКАЦИЮ МАШИННОГО КОДА

Проверка целостности

- Проверка целостности служит для того, чтобы никто не лез в вашу программу различными инструментами модификации кода (будь то Hex-редактор или Keypatch)
 - В основном применяется для того, чтобы прикрыть какие-то методы проверки лицензии
 - Особенно полезна для методов, основанных на асимметричной криптографии, чтобы никто не приставал к вашему публичному ключу
- В основном может применяться к следующему:
 - Исполняемый файл программы на диске
 - Представление программы в оперативной памяти
 - Загружаемые библиотеки

Проверка исполняемого файла

- Самый простой подход, вполне годится для начинающих
- Состоит в следующем:
 - Находим путь к текущему исполняемому файлу
 - Открываем его, читаем и берем контрольную сумму
 - Сравниваем с образцом
- Как ни странно, основные приключения в Linux происходят на первом шаге
- Чтобы получить имя текущего файла, можно использовать следующие способы:
 - `argv[0]`
 - `/proc/self/exe`
 - `/proc/self/maps`, `dladdr`
 - Наверняка, что-нибудь еще

Получаем имя исполняемого файла: argv[0]

- Обычно в массиве аргументов argv, передаваемом в функцию main(), присутствует и нулевой элемент, указывающий на имя запущенной программы
- В argv[0] может храниться как полный, так и локальный путь к файлу, а в случае, путь к файлу доступен в переменной окружения PATH – еще и просто имя файла
 - Самое печальное, что в argv[0] может храниться что-то вообще произвольное: системный вызов execve() позволяет вызывающей программе указывать любое значение argv[0]
- Несмотря на все недостатки и нюансы, у этого способа есть и преимущество: он не требует вообще никаких системных вызовов, что позволит вам не выдать себя перед аналитиком
- argv[0] часто используется программами, которые любят запускаться по-разному по разным символическим ссылкам (например, BusyBox)

Получаем имя исполняемого файла: argv[0]

```
n0n3m4@localhost:/bin$ ls
```

→ argv[0] == "ls"

```
n0n3m4@localhost:/bin$ /bin/ls
```

→ argv[0] == "/bin/ls"

```
n0n3m4@localhost:/bin$ ./ls
```

→ argv[0] == "./ls"

```
n0n3m4@localhost:~$ exec -a qwe ls
```

→ argv[0] == "qwe"

Получаем имя исполняемого файла: /proc/self/exe

- Гораздо более практически применимый способ, используется во многих программах Linux в качестве основного
- Основан на использовании символической ссылки /proc/self/exe, хранящей путь к текущему исполняемому файлу
 - Путь по этой ссылке можно узнать при помощи readlink()
 - Или же можно напрямую открыть этот файл при помощи fopen

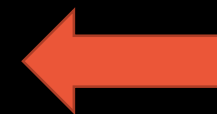
```
n0n3m4@localhost:~$ ls -la /proc/self/exe
lrwxrwxrwx 1 n0n3m4 n0n3m4 0 Apr  4 13:37 /proc/self/exe -> /bin/ls
```

Получаем имя исполняемого файла: /proc/self/maps

- Альтернативный способ, который можно использовать, если вы хотите скрыть попытку узнать имя процесса
 - /proc/self/exe очень заметен и может вызвать подозрения
- В файле /proc/self/maps хранится информация обо всех сегментах памяти
 - В том числе и об отображенных на память файлах, к которым относится ваша программа
 - Также можно использовать /proc/self/smaps
- Правда, в этом же файле будут присутствовать и библиотеки, так что нужно как-то отличить свою программу от них
 - Это легко сделать, собрав ее без флага PIE, например (тогда ее адрес всегда будет около 0x400000)
 - Можно также сопоставить адрес какой-нибудь функции (main) с адресами из списка

Получаем имя исполняемого файла: /proc/self/maps

```
n0n3m4@localhost:~$ cat /proc/self/maps
7f976168d000-7f9761800000 r--p 00000000 00:00 116741 /usr/lib/locale/C.UTF-8/LC_COLLATE
7f9761800000-7f97619e7000 r-xp 00000000 00:00 115203 /lib/x86_64-linux-gnu/libc-2.27.so
7f97619e7000-7f97619f0000 ---p 001e7000 00:00 115203 /lib/x86_64-linux-gnu/libc-2.27.so
7f97619f0000-7f9761be7000 ---p 000001f0 00:00 115203 /lib/x86_64-linux-gnu/libc-2.27.so
7f9761be7000-7f9761beb000 r--p 001e7000 00:00 115203 /lib/x86_64-linux-gnu/libc-2.27.so
7f9761beb000-7f9761bed000 rw-p 001eb000 00:00 115203 /lib/x86_64-linux-gnu/libc-2.27.so
7f9761bed000-7f9761bf1000 rw-p 00000000 00:00 0
7f9761c00000-7f9761c26000 r-xp 00000000 00:00 115179 /lib/x86_64-linux-gnu/ld-2.27.so
7f9761c26000-7f9761c27000 r-xp 00026000 00:00 115179 /lib/x86_64-linux-gnu/ld-2.27.so
7f9761c8c000-7f9761e27000 r--p 00000000 00:00 116753 /usr/lib/locale/locale-archive
7f9761e27000-7f9761e28000 r--p 00027000 00:00 115179 /lib/x86_64-linux-gnu/ld-2.27.so
7f9761e28000-7f9761e29000 rw-p 00028000 00:00 115179 /lib/x86_64-linux-gnu/ld-2.27.so
7f9761e29000-7f9761e2a000 rw-p 00000000 00:00 0
7f9761ee0000-7f9761f02000 rw-p 00000000 00:00 0
7f9761f0f000-7f9761f40000 r--p 00000000 00:00 116742 /usr/lib/locale/C.UTF-8/LC_CTYPE
...
7f9761f67000-7f9761f68000 r--p 00000000 00:00 116743 /usr/lib/locale/C.UTF-8/LC_IDENTIFICATION
7f9762000000-7f9762007000 r-xp 00000000 00:00 112947 /bin/cat
7f9762007000-7f9762008000 r-xp 00007000 00:00 112947 /bin/cat
7f9762207000-7f9762208000 r--p 00007000 00:00 112947 /bin/cat
7f9762208000-7f9762209000 rw-p 00008000 00:00 112947 /bin/cat
7ffff743e000-7ffff745f000 rw-p 00000000 00:00 0 [heap]
7ffffe239000-7ffffea39000 rw-p 00000000 00:00 0 [stack]
7ffffeaf7000-7ffffeaf8000 r-xp 00000000 00:00 0 [vdso]
```



Имя запущенного файла

Получаем имя исполняемого файла: dladdr

- Функция `dladdr` позволяет найти информацию о функции по ее адресу
 - В том числе к информации относится и путь к разделяемой библиотеке, в которой лежит функция
- Таким образом, при помощи `dladdr(&main, ...)` можно получить информацию о пути к «библиотеке», где она находится
 - Исполняемый файл тоже почему-то считается библиотекой
- Однако, по умолчанию с функцией `dladdr (glibc)` в таком сценарии есть беда – она возвращает то же самое, что и `argv[0]`, что делает ее не слишком полезной
 - Существует функция `dladdr1`, которая может вернуть дополнительную информацию в виде структуры `link_map`, но, увы, там исполняемый файл (даже PIE) библиотекой уже не считается, поскольку загружается ядром, так что путь остается пустым

Получаем имя исполняемого файла: dladdr

```
#define _GNU_SOURCE
#include <dlfcn.h>
int main()
{
    Dl_info dli;
    dladdr(&main, &dli);
    printf("Executable location (dladdr): %s\n", dli.dli_fname);
}
```

```
n0n3m4@localhost:/tmp$ ./dladdr
Executable location (dladdr): ./dladdr
n0n3m4@localhost:/tmp$ /tmp/dladdr
Executable location (dladdr): /tmp/dladdr
n0n3m4@localhost:/tmp$ bash -c 'exec -a qwe ./dladdr'
Executable location (dladdr): qwe
```

Проверка программы в оперативной памяти

- Данный подход был поверхностно описан в прошлой лекции
- Этот подход помогает не только от модификации самого исполняемого файла, но и от программ, которые вмешиваются в работу непосредственно в процессе работы
 - Например т.н. «лоадеров», взламывающих программу каждый раз в оперативной памяти после ее запуска (но оставляющих ее файл на диске неизменным)
- Основан на прочтении кода программы как данных и, соответственно, подсчете контрольной суммы этих данных
 - Соответственно, требует возможности читать свой код (можно попробовать вручить себе эти права при помощи mprotect, если это почему-то невозможно на вашей ОС)
- Основной проблемой является необходимость знания адресов (и особенно длины) функций в памяти

Узнаем адреса и длину кода

- Самый простой способ – прочитать содержимое `/proc/self/maps`
 - Все аналогично получению имени файла, только теперь вместо имени – адреса в памяти
 - Наиболее полезны сегменты с правами на исполнение вроде "r-xp"
- Однако, нужно быть осторожным с границами страниц: в зависимости от операционной системы (например, WSL 1 или Ubuntu) и (потенциально) версий ядра неиспользуемые участки файла могут по-разному отображаться на память
 - В Ubuntu если в одной и той же странице памяти находится и код и данные – они будут отображены и туда и туда (дважды)
 - В WSL 1 неиспользуемые данные честно забиваются нулями
 - Лучше просто пропускать последнюю страницу, если размер кода позволяет (а также если в той странице нет ничего важного)

Пример защиты с /proc/self/maps

```
#include <stdio.h>
#include <stdint.h>
int main() {
    char tmp[4096];
    FILE * f = fopen("/proc/self/maps", "rb");
    uint64_t sum = 0;
    while (fgets(tmp, 4096, f))
        if (strstr(tmp, "mapscheck") && strstr(tmp, "r-xp"))
        {
            uintptr_t start, end;
            sscanf(tmp, "%llx-%llx", &start, &end);
            for (uint64_t* p = start; p < end; p++)
                sum ^= *p;
        }
    printf("XOR sum is %llx\n", sum);
}
```

Узнаем адреса и длину кода

- Другой способ – использовать особенности компилятора и линкера
 - Например, тот факт, что функции в исполняемом коде часто оказываются в том же порядке, в котором следуют в исходном коде (а значит конец функции находится примерно там же, где и начало следующей)
 - Также можно использовать специальные переменные, создаваемые линкером для начала и конца секции .text: `__executable_start` и `__etext` (конкретные переменные зависят от вашего компилятора)
- Эти способы менее устойчивы к смене компилятора, однако более стабильны (поскольку вся информация об адресах доступна во время компиляции) и не требуют открытия каких-либо файлов
- Первый больше подходит для защиты отдельных функций, а второй – для всего кода программы

Пример защиты одной функции

```
#include <stdio.h>
#include <stdint.h>

void protectedfun() {
    printf("Much protected, wow\n");
}
void endoffun() { }

int main() {
    uint64_t* start = &protectedfun;
    uint64_t* end = &endoffun;
    uint64_t sum = 0;
    for (uint64_t* p = start; p < end; p++)
        sum ^= *p;
    printf("XOR sum is %llx\n", sum);
}
```

Пример защиты с __executable_start и __etext

```
#include <stdio.h>
#include <stdint.h>

extern void* __executable_start;
extern void* __etext;

int main()
{
    uint64_t* start = &__executable_start;
    uint64_t* end = &__etext;
    uint64_t sum = 0;
    for (uint64_t* p = start; p < end; p++)
        sum ^= *p;
    printf("XOR sum is %llx\n", sum);
}
```


Проверка целостности

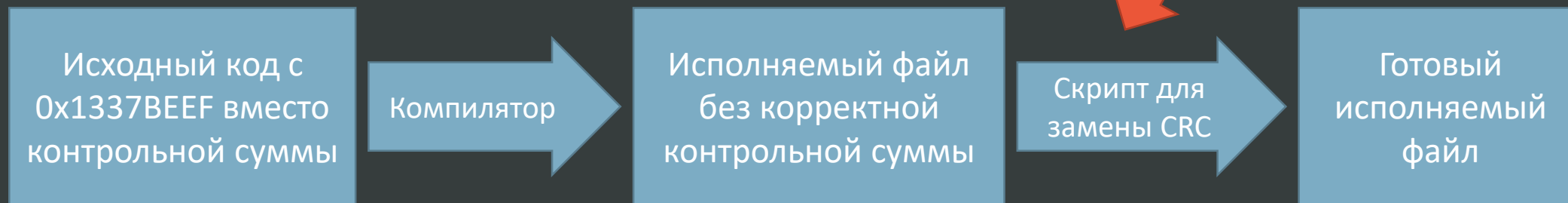
- После получения машинного кода любым из способов, нужно его как-то проверить на целостность
- Обычно сначала к коду применяется какая-то контрольная или хэш-сумма
 - В результате из длинного кода получится строка фиксированного размера, которую легче использовать для проверки
- Проверку можно осуществлять следующими способами:
 - Прямым сравнением на равенство
 - Использованием полученного результата в дальнейшей работе программы

Сравнение на равенство

- В случае, если контрольная сумма не оправдала ваших ожиданий, можно:
 - Завершить работу приложения
 - Начать мелко пакостить, испортив какие-нибудь переменные в программе или вообще стерев незадачливому реверс-инженеру папку /home
- Обычно, корректная контрольная сумма встраивается в исполняемый файл уже после сборки (поскольку до сборки исполняемого файла она может быть неизвестна)
- В большинстве случаев при таком сравнении вам придется исключить контрольную сумму из данных, подвергающихся взятию контрольной суммы, поскольку в противном случае она сама будет влиять на результат

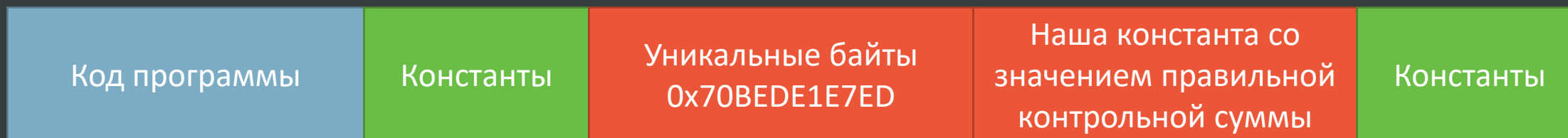
Сравнение на равенство

Обычно пишется самостоятельно,
ищет в файле 0x1337BEEF и заменяет
корректной контрольной суммой



Как не суммировать лишнего

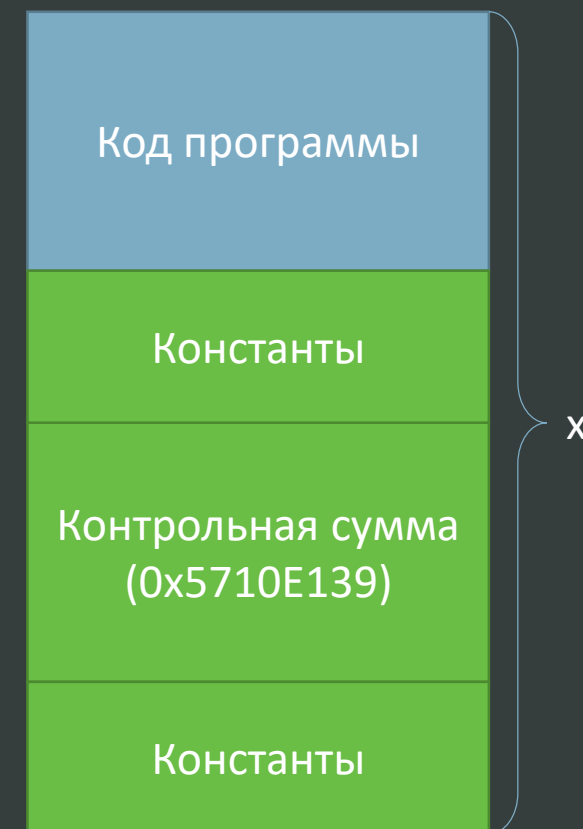
- Как пропустить само значение контрольной суммы в файле?
 - Можно окаймить его какими-то уникальными последовательностями байтов
 - А можно его и не пропускать
- Первый вариант выглядит следующим образом:



- В этом случае при подсчете контрольной суммы встретившись с байтами 0x070BEDE1E7ED можно пропустить <длина контрольной суммы> байтов после них

Как не суммировать лишнего

- Второй вариант годится для коротких контрольных сумм вроде CRC32
- Можно просто менять контрольную сумму на случайную до тех пор, пока итоговая контрольная сумма файла не совпадет с ней
 - Сложность этого перебора составит порядка 2^{32}
 - Успех перебора не гарантируется (чтобы увеличить его вероятность, можно добавить еще пару бесполезных байтов, которые можно менять на случайные)
- Задача аналогична поиску строки, содержащей свою CRC
 - Как в задаче "CRC32 challenge" из темы «криптография»
- Вариант очень удобен тем, что код становится проще
 - Однако время сборки программы существенно возрастает



$\text{CRC}(x) == 0x5710E139$

Использование CRC в логике программы

- Вариант для более продвинутых программистов
 - Зато «пакости» в данном варианте идут в комплекте абсолютно бесплатно
- Данный вариант проверки не подразумевает прямого сравнения контрольной суммы с корректной, вместо этого предполагается использование ее значения из предположения, что оно равно корректному
- Возможны следующие варианты:
 - Отправка этой контрольной суммы на удаленный сервер (программа не будет в курсе, что с суммой что-то не так, а вот сервер может изменить свое поведение)
 - Генерация ключей шифрования от этой контрольной суммы (для расшифровки игровых ресурсов, например)
 - Использование частей контрольной суммы как математических констант (например, при неудачном взломе число π может уйти в минус)
 - Все, чем ограничена ваша фантазия

Использование CRC в логике программы

```
#include <stdio.h>
#include <zlib.h>
static unsigned int pi = 0x4048f5c3 ^ 0x41f34aa9;
int checkkey(char* x) {
    if (!strcmp(x, "S3cr3t"))
        return 1;
    else
        return 0;
}
int main() {
    char tmp[1024];
    printf("Enter key: ");
    pi ^= crc32(0, &checkkey, &main - &checkkey);
    gets(tmp);
    if (checkkey(tmp))
        printf("Ok, length is %f\n",
            2 * *((float*)&pi) * 1);
}
```

```
n0n3m4@localhost:/tmp$ ./original
Enter key: S3cr3t
Ok, length is 6.280000
```

```
n0n3m4@localhost:/tmp$ ./patched
Enter key: qweqwe
Ok, length is -1687430144.000000
```



Проверка ключа запатчена

Проверка загружаемых библиотек

- Еще один способ нарушить работу программы, напрямую не покрываемый ни проверкой целостности, ни антиотладкой – добавление в процесс своего кода в составе разделяемой библиотеки
- Атакующий может использовать следующие методы:
 - LD_PRELOAD / LD_AUDIT
 - Замена системных библиотек (вроде libc.so)

Противодействие LD_PRELOAD / LD_AUDIT

- LD_PRELOAD и LD_AUDIT подгружают в процесс еще одну библиотеку
 - Именно тут их и можно поймать – библиотек станет на одну больше, этот способ сработает и с /etc/ld.so.preload
- Обнаружить лишние библиотеки можно прочитав /proc/self/maps
 - Очень важно для открытия и чтения файлов использовать только системные вызовы – они не подвержены замене
- Или можно просто собрать свой исполняемый файл статически
- Можно посмотреть, нет ли в переменных окружения LD_PRELOAD и LD_AUDIT...
 - Но это совсем простой способ, ведь эту переменную окружения можно убрать после загрузки, разве что атакующий не знает о функциях-конструкторах

Противодействие LD_PRELOAD



main:

```
fd = syscall(SYS_open, "/proc/self/maps", O_RDONLY)
```

```
syscall(SYS_read, fd, tmp, 4096)
```



Здесь syscall() – собственная реализация на ассемблере, не функция из libc



main:

```
f = fopen("/proc/self/maps", "r")
```

```
fread(tmp, 1, 4096, f)
```

evil.so:

```
fopen
```

```
fread
```

Замена системных библиотек

- Можно опять же собрать свой исполняемый файл статически, насколько это возможно
 - Впрочем, ядро при желании тоже можно заменить
- Также можно ввести белый список системных библиотек
 - Впрочем, это довольно бесперспективно на Linux, учитывая количество существующих дистрибутивов
- В большинстве случаев с возможностью замены системных библиотек приходится смириться и пытаться обнаруживать вредоносную замену по косвенным признакам
 - Например, рандом стал слишком статистически плох
 - Или скриншоте компьютерной игры врагов стало видно через стены

Борьба с проверкой целостности

Как бороться с проверкой целостности?

- Для начала ее нужно обнаружить
- Существует два принципиальных подхода:
 - Попробовать поймать проверку целостности отладкой (в этом случае антиотладка может вам помешать)
 - Спровоцировать программу на падение

Ловим проверку целостности отладкой

- Для того, чтобы поймать программу за чтением собственного файла, достаточно использовать `strace`
 - Особенно хороши для этого флаги `-i` и `-k`, позволяющие узнать текущий адрес на момент выполнения вызова и посмотреть весь стек вызовов до системного вызова соответственно
 - Готовая команда может выглядеть как `"strace -i -k -e open,openat ./program"`
- Для того, чтобы поймать программу за чтением кода в памяти, можно использовать точку останова на чтение памяти
 - Такую точку останова можно поставить при помощи команды GDB `"rwatch * <адрес>"`
 - Можно расставить точки останова по подозрительным местам программы (в частности, на место, которое вы хотите изменить) и посмотреть, что будет
 - В WSL 1 `rwatch` может не заработать (у меня не заработал)

```
n0n3m4@localhost:/tmp$ strace -i -k -e open,openat /tmp/main
```

```
...
```


```
> /lib/x86_64-linux-gnu/ld-2.27.so() [0x1098]
[00007f23cba1ccdd] openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
> /lib/x86_64-linux-gnu/ld-2.27.so(_dl_catch_error+0x101d) [0x1ccdd]
> /lib/x86_64-linux-gnu/ld-2.27.so() [0x5c07]
> /lib/x86_64-linux-gnu/ld-2.27.so() [0x93ce]
> /lib/x86_64-linux-gnu/ld-2.27.so(_dl_rtlld_di_serinfo+0x4782) [0xe312]
> /lib/x86_64-linux-gnu/ld-2.27.so(_dl_catch_exception+0x5b) [0x1bc6b]
> /lib/x86_64-linux-gnu/ld-2.27.so(_dl_rtlld_di_serinfo+0x4a88) [0xe618]
> /lib/x86_64-linux-gnu/ld-2.27.so() [0x3ea6]
> /lib/x86_64-linux-gnu/ld-2.27.so(__get_cpu_features+0x13a0) [0x1add0]
> /lib/x86_64-linux-gnu/ld-2.27.so() [0x2128]
> /lib/x86_64-linux-gnu/ld-2.27.so() [0x1098]
[00007f23cb36fc8e] openat(AT_FDCWD, "/proc/self/exe", O_RDONLY) = 3
> /lib/x86_64-linux-gnu/libc-2.27.so(__open64+0x4e) [0x7ee8e]
> /lib/x86_64-linux-gnu/libc-2.27.so(_IO_file_fopen+0xda) [0x7ee3a]
> /lib/x86_64-linux-gnu/libc-2.27.so(fopen+0x7a) [0x7eeaa]
> /tmp/main() [0x1133]
> /lib/x86_64-linux-gnu/libc-2.27.so(__libc_start_main+0xe7) [0x21b9]
> unexpected_backtracing_error [0x500000001]
```

/proc/self/exe читается из
нашей программы /tmp/main,
это подозрительно

Ловим проверку целостности отладкой

```
(gdb) rwatch * 0x400651
Hardware read watchpoint 1: * 0x400651
(gdb) r
Starting program: /tmp/bigpi
```

Ставим точку останова на чтение инструкции, которую хотим запатчить



```
Hardware read watchpoint 1: * 0x400651
```

```
Value = -75032
```

```
0x00007ffff7bbacb8 in crc32_z () from /lib/x86_64-linux-gnu/libz.so.1
```

```
(gdb) bt
```

```
#0 0x00007ffff7bbacb8 in crc32_z () from /lib/x86_64-linux-gnu/libz.so.1
```

```
#1 0x00000000004006ba in main ()
```

```
(gdb)
```



Смотрим, кто решил почитать код при помощи bt

Провоцируем программу на падение

- Некоторые разработчики не проверяют, доступен ли реально файл или память для чтения
 - Например я во всех задачах курса не теряю на это время
 - Это можно использовать: программа, написанная таким образом, скорее всего упадет ровно там, где пыталась сделать что-то интересное
- Для того, чтобы поймать программу за чтением собственного файла достаточно просто отобрать у всех права на чтение программы (`chmod 111 <программа>`)
 - Работает только если вы не запускаете программу из под рута, наверняка есть и другие способы (в т.ч. с рутом), но этот самый простой
- С чтением памяти программы, увы, сложнее: x86 не позволяет сделать память исполняемой, но нечитаемой (на ARM64 такое возможно)
 - Возможно с технологией Intel MPK и инструментом <https://github.com/intel/xom-switch>, но я не смог это проверить: нужен дорогой Хеон, технология так и осталась экзотикой

Провоцируем программу на падение


- К сожалению, Linux не будет автоматически создавать файлы дампа (coredump) для программ, для которых вы не имеете разрешений на чтение
- Поэтому, у вас есть следующие варианты:
 - Использовать LD_PRELOAD, установив обработчик SIGSEGV и посмотрев трейс оттуда (если LD_PRELOAD не заблокирован)
 - Использовать strace (теперь вам не нужно будет следить за системными вызовами, оно само упадет, очень удобно, хотя это все еще будет считаться отладкой)
 - Как-то выключить эти ограничения в ядре
 - Использовать костыли (в WSL 1 не работает)
- По крайней мере, однако, поймав сам факт падений, вы узнаете, что программа проверяет целостность
 - Это уже само по себе очень полезно

Используем костыли

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char** argv) {
    // Создаем подпроцесс
    if (!fork()) {
        // Выдаем файлу права на исполнение
        chmod(argv[1], 0777);
        // Запускаем его
        execv(argv[1], argv + 1);
        printf("Oops, maybe need more time in usleep?\n");
    }
    else {
        // Ждем несколько микросекунд (по вкусу)
        usleep(1);
        // Отбираем у программы права на чтение самой себя
        chmod(argv[1], 0000);
    }
}
```

Используем костыли

```
# Указываем имя файла с дампом, нейтрализуя Ubuntu Apport и прочее
n0n3m4@localhost:~$ sudo sh -c 'echo core > /proc/sys/kernel/core_pattern'
# Убираем лимит на размер файла с дампом (по умолчанию там 0 и он не создается)
n0n3m4@localhost:~$ ulimit -c unlimited
# Запускаем исследуемую программу через костыль, отбирающий права
n0n3m4@localhost:~$ ./kostyl ./main
# Возвращаем права на чтение main и запускаем GDB с новым файлом дампа
n0n3m4@localhost:~$ chmod 777 main && gdb main --core=core
Core was generated by `./main'.
Program terminated with signal SIGSEGV, Segmentation fault.
(gdb) bt
#0  __GI_fseek (fp=0x0, offset=0, whence=??) at fseek.c:35
#1  0x000000000401153 in ?? ()
#2  0x00007f6bef221b97 in __libc_start_main
#3  0x0000000004009ea in ?? ()
(gdb)
```



А вот и адрес, где живет не в меру любопытная проверка

Обфускация

Обфускация

- Обфускация – процесс запутывания кода
- Ранее мы познакомились с обфускацией для исходного кода (она запутывала код, делая его нечитаемым, например, удаляя имена переменных)
- В случае машинного кода обфускаторы также применяют различные преобразования, призванные затруднить анализ:
 - Замена констант на выражения, которые в конечном счете приводят к такому же результату
 - Замена условий и циклов в программе одним конечным автоматом (control flow flattening) – портит графики в дизассемблерах вроде IDA
 - Виртуализация – замена машинного кода на виртуальную машину некоторого нового языка и представление этого кода на ее языке
 - Вставка всегда истинных или ложных условий (с мусорным кодом)
 - Многое другое

Известные обфускаторы

- [Movfuscator](#) – обфускатор с открытым исходным кодом для 32-битного x86
 - Превращает все инструкции программы в MOV 0_o
- [Obfuscator-LLVM](#) – классический обфускатор с открытым исходным кодом
 - Основан на LLVM
 - Содержит множество ставших классикой преобразований: <https://crypto.junod.info/spro15.pdf>
 - Является наиболее часто используемым в практических целях обфускатором
 - Выкуплен Snap Inc. (Snapchat), так что оригинальными авторами не обновляется
- [Tigress](#) – обфускатор из исходного кода на языке Си в исходный код на языке Си
 - Исходный код не предоставляется под надуманными предлогами (что, однако, не мешает авторам публиковать научные статьи)
 - Написан на OCaml, так что закрытость его кода, наверное, и к лучшему
 - Крайне нетривиален в использовании

Obfuscator-LLVM

```
; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near
; __unwind {
push    rbp
mov     rbp, rsp
lea     rdi, format      ; "Hello, please enter key: "
mov     eax, 0
call    _printf
lea     rsi, string
lea     rdi, a1023        ; "%1023[^\n]"
mov     eax, 0
call    __isoc99_scanf
lea     rax, string
mov     eax, [rax]
cmp     eax, 'pm1S'
jnz     short loc_763
```

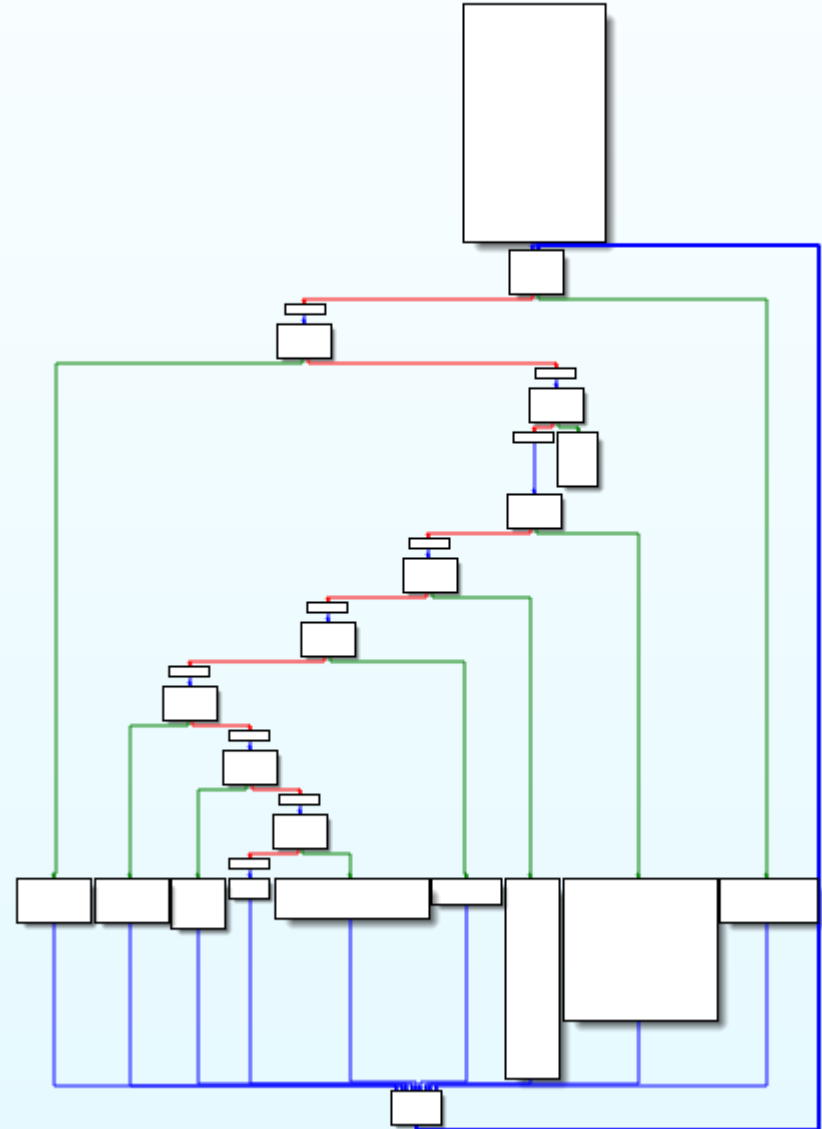
```
lea     rax, unk_201044
mov     eax, [rax]
cmp     eax, 'eke1'
jnz     short loc_763
```

```
lea     rax, unk_201048
mov     eax, [rax]
cmp     eax, 79h ; 'y'
jnz     short loc_763
```

```
lea     rdi, s            ; "Correct key, congratulations!"
call    _puts
jmp     short loc_76F
```

```
loc_763:
lea     rdi, aWrongKey    ; "Wrong key"
call    _puts
```

```
loc_76F:
mov     eax, 0
pop     rbp
retn
; } // starts at 6FA
main endp
```



Movfuscator

```

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near
; __unwind {
push    rbp
mov     rbp, rsp
lea     rdi, format      ; "Hello, please enter key: "
mov     eax, 0
call    _printf
lea     rsi, string
lea     rdi, a1023        ; "%1023[^\n]"
mov     eax, 0
call    __isoc99_scanf
lea     rax, string
mov     eax, [rax]
cmp     eax, 'pm15'
jnz     short loc_763

```

```

lea     rax, unk_201044
mov     eax, [rax]
cmp     eax, 'ekel'
jnz     short loc_763

```

```

lea     rax, unk_201048
mov     eax, [rax]
cmp     eax, 79h ; 'y'
jnz     short loc_763

```

```

lea     rdi, s          ; "Correct key, congratulations!"
call    _puts
jmp     short loc_76F

```

```

loc_763:
lea     rdi, aWrongKey ; "Wrong key"
call    _puts

```

```

loc_76F:
mov     eax, 0
pop     rbp
retn
; } // starts at 6FA
main endp

```



```

.text:08048C0C
.text:08048CE1
.text:08048CE8
.text:08048CEE
.text:08048CF0
.text:08048CF5
.text:08048CFA
.text:08048CFF
.text:08048D01
.text:08048D06
.text:08048D0B
.text:08048D11
.text:08048D16
.text:08048D1D
.text:08048D23
.text:08048D29
.text:08048D2B
.text:08048D30
.text:08048D36
.text:08048D3B
.text:08048D42
.text:08048D48
.text:08048D4A
.text:08048D4F
.text:08048D54
.text:08048D5E
.text:08048D63
.text:08048D68
.text:08048D72
.text:08048D78
.text:08048D7F
.text:08048D86
.text:08048D89
.text:08048D90
.text:08048D97
.text:08048D9A
.text:08048DA1
.text:08048DA7
.text:08048DAD
.text:08048DB4
.text:08048DBB
.text:08048DBE
.text:08048DC5
.text:08048DCC
.text:08048DCF
.text:08048DD6
.text:08048DDC
.text:08048DE1
.text:08048DE3
.text:08048DE8
.text:08048DED
.text:08048DF3
.text:08048DF8

```

```

mov     data_p, eax
mov     eax, sel_data[edx*4]
mov     edx, off_83F5160
mov     [eax], edx
mov     eax, offset aHelloPleaseEnt ; "Hello, please enter key: "
mov     R3, eax
mov     eax, R3
mov     eax, eax
mov     stack_temp, eax
mov     eax, offset off_83F5160
mov     edx, on
mov     data_p, eax
mov     eax, sel_data[edx*4]
mov     edx, off_83F5160
mov     edx, [edx-200068h]
mov     [eax], edx
mov     eax, off_83F5160
mov     edx, on
mov     data_p, eax
mov     eax, sel_data[edx*4]
mov     edx, stack_temp
mov     [eax], edx
mov     eax, 88048E4Ah
mov     alu_x, eax
mov     alu_y, 80000000h
mov     eax, 0
mov     ecx, 0
mov     alu_c, 0
mov     ax, word ptr alu_x
mov     cx, word ptr alu_y
mov     edx, alu_add16[edx*4]
mov     edx, [edx+ecx*4]
mov     cx, word ptr alu_c+2
mov     edx, alu_add16[edx*4]
mov     edx, [edx+ecx*4]
mov     word ptr alu_s, dx
mov     alu_c, edx
mov     ax, word ptr alu_x+2
mov     cx, word ptr alu_y+2
mov     edx, alu_add16[edx*4]
mov     edx, [edx+ecx*4]
mov     cx, word ptr alu_c+2
mov     edx, alu_add16[edx*4]
mov     edx, [edx+ecx*4]
mov     word ptr alu_s+2, dx
mov     alu_c, edx
mov     eax, alu_s
mov     eax, eax
mov     stack_temp, eax
mov     eax, offset off_83F5160
mov     edx, on
mov     data_p, eax
mov     eax, sel_data[edx*4]

```

Обфускация

- Многие обфускаторы (в том числе Movfuscator и Obfuscator-LLVM) представляют собой компиляторы языка Си и C++
 - Компилятор Си называется movcc в случае movfuscator
 - Компиляторы Си и C++ называются clang и clang++ в случае Obfuscator-LLVM
- Соответственно, чтобы применить их к своей программе, достаточно просто сменить компилятор на обфусцирующий
 - С Tigress это так не работает, например
- Важное замечание: в случае с Obfuscator-LLVM все обфусцирующие преобразования нужно включать отдельно, об этом можно прочитать на странице <https://github.com/obfuscator-llvm/obfuscator/wiki/Features> или просто дописать к флагам компиляции "-mllvm -fla -mllvm -sub -mllvm -bcf"

Противодействие обфускации

- Деобфускация – в общем случае очень нетривиальная задача, которая часто становится темой научных работ
- Готовых инструментов для OLLVM и Movfuscator немного:
 - <https://github.com/RPISEC/llvm-deobfuscator> – плагин для дизассемблера Binary Ninja, обрабатывает только control flow flattening (да и неизвестно, работает ли: нужен платный Binary Ninja)
 - <https://github.com/yqw1212/demovfuscator> (это, правда, форк с исправлениями) – деобфускатор для Movfuscator, много что пытается делать, но нормально получается у него только вернуть поток исполнения
 - <https://gitlab.com/eshard/d810> – Python-плагин для автоматической деобфускации во время декомпиляции в IDA Pro, на вид очень хорош, имеет настройки против OLLVM. Не забудьте поставить Z3, как советует автор, запуск по Ctrl-Shift-D. Работает только в декомпиляторе, граф и дизассемблеровый листинг так и останутся уродливыми

Противодействие обфускации

- Также код, обфусцированный OLLVM можно с некоторым успехом читать вручную
 - По крайней мере, с гораздо большим, чем код Movfuscator
- Обфусцированные программы все еще можно рассматривать как черный ящик
 - strace, ltrace, дампы памяти и прочее работают как прежде (если нет виртуализации)

Demovfuscator

- Этот инструмент крайне весело собирается из исходного кода, поэтому я собрал его для вас (под Ubuntu 18.04)
 - Скачать исполняемый файл можно по ссылке <https://n0n3m4.ru/nsuctf/demov>
 - Для его запуска вам все равно потребуется установить libz3-dev (через apt-get)
- Используется инструмент следующим образом:

```
n0n3m4@localhost:~$ ./demov ./main -g main.dot -o demain  
n0n3m4@localhost:~$ cat main.dot | dot -Tpng > main.png
```

- В файле demain будет доступна «деобфусцированная» версия программы
- В файле main.png будет доступен график потока исполнения с адресами
 - То есть, условные переходы и все в этом духе

Спасибо за внимание!
Задачи доступны на

nsuctf.ru

- Пожалуйста, используйте имя пользователя формата “Фамилия Имя”
 - e-mail можно забить любой, сервером он не проверяется
- Для вопросов по задачам рекомендую присоединиться к @NSUCTF в Telegram
 - Только, пожалуйста, без спойлеров