1. Санкт-Петербургский Политехнический Университет Петра Великого
2. **Институт кибербезопасности и защиты информации**

**ЛАБОРАТОРНАЯ РАБОТА № 1**

1. «Изучение методов исследования исполняемых файлов с элементами самозащиты»
2. по дисциплине «Теория и системы управления информационной безопасностью»
3. Вариант 2
4. Выполнил
5. студент гр. 4851003/70801 Гасанов Э. А.

<*подпись*>

1. Преподаватель
2. доцент, к.т.н. Жуковский Е.В.

<*подпись*>

1. Санкт-Петербург
2. 2021

**Цель работы**

Изучение методов защиты вредоносного программного обеспечения и способов их обхода. Изучение методов анализа бинарного кода.

**Задачи работы**

В ходе выполнения лабораторной работы необходимо выполнить следующие действия:

Получить у преподавателя исполняемые файлы, соответствующий индивидуальному номеру варианта задания. Полученная программа осуществляет считывание ключевой информации. В случае ее правильного ввода осуществляется печать на экране строки «Success!» и вывод «секретного кода», в ином случае выводится «Access denied!» и программа завершается. (2 ключа – преобразование входа и просто вывод)

Изучить подходы, применяемые для защиты программного обеспечения.

Осуществить анализ полученной программы на наличие защитных механизмов с использованием как статических, так и динамических методов анализа.

Обойти защитные механизмы, присутствующие в программе путем ее модификации и перехвата (mhook/minhook), используемых ей функций.

Определить в бинарном коде программы место вывода секретного кода.

Получить «секретный» код путем исполнения модифицированной программы.

Разработать скрипты IDAPython, находящие механизмы самозащиты в программе и обходящие их. Разработанные скрипты должны выполнять следующие действия:

* 1. Нахождение криптографических примитивов (xor).
  2. Нахождение закодированных строк (xor).
  3. Нахождение мест чтения PEB в коде.
  4. Нахождение мест замера времени (rdtsc, GetTickCount).
  5. Нахождение мест использования «редких» инструкций (int 2d, ss, и т.д.).
  6. Нахождение других признаков механизмов защиты бинарного кода.

Используя среду символьного выполнения (BARF, Triton, S2E, BitBlaze/FuzzBall, BAP, Fuzzgrind, MiAsm, angr) получить входное значение программы и секретный код.

Протестировать разработанные скрипты на исследуемом исполняемом файле.

Протестировать разработанные скрипты на исполняемом файле программы, соответствующей указанному варианту из архива bin2. Осуществить анализ полученного файла и решить crackme.

В отчете необходимо привести следующую информацию:

* описание алгоритма работы программы;
* пароль, запрашиваемый программой;
* «секретный код», полученный при анализе программы;
* место вывода «секретного кода» в бинарном коде программы;
* перечень механизмов защиты и фрагменты ассемблерного кода, отвечающие за их реализацию;
* описание принципа действия используемых механизмов защиты;
* описание процесса модификации программы и обхода защитных механизмов;
* описание решения crackme из архива bin2;
* исходный код реализованных скриптов IDAPython;
* исходный код скриптов для средств символьного исполнения.

**Ход работы**

Вариант задания — 2.

Был выполнен анализ программы crackme1.exe (нумерация с нуля) – это 32-битная программа, так как в HXD видно, что PE\_L (рисунок 1) ( 64 бит - PE..d†).

Изображение выглядит как стол

Автоматически созданное описание

Рисунок 1 – Разрядность программы.

Используется TlsCallback (рисунок 2), который мешает пользоваться отладчиком. TlsCallback позволяет авторам вредоносных программ выполнять вредоносный код до того, как отладчик сможет остановиться в традиционной EntryPoint. А также в SetUnhandledExceptionFilter может находится пользовательская функция обработки при обнаружении того, что производится отладка ( и выбросить исключение через SEH). После этого управление передаётся в программную точку входа (это будет видно так как они обе экспортируются). Это пример противодействия динамическому исследованию.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 2 – TLS-callback, мешающий отладчикам остановиться на EntryPoint.

IDA распознает main в списке функций. А также main можно найти через экспорт точки входа (рисунок 3 и 4).

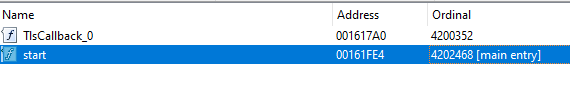


Рисунок 3 – Экспорт.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 4 – Экспортируемая точка входа.

IDA ошиблась при нахождении границ функции main из-за антиотладочного механизма, находящего, что процесс находится под отладкой (рисунок 5). Если возвращаемое значение истина, то вызывается деление на 0, что кладёт всю программу (рисунок 6).

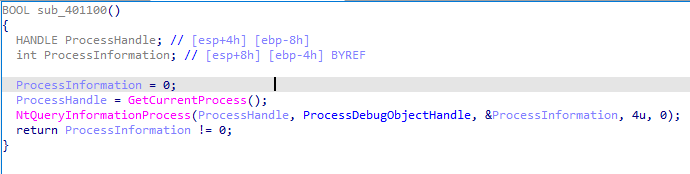


Рисунок 5 – Обнаружение того, что процесс под отладкой. Возвращается истина – 1!=0 .

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 6 – Деление на ноль.

Защиту от отладчика легко обойти с помощью непосредственно самого отладчика. Так как проверка выполняется в известном месте, то управление (EIP) можно вручную перенести далее, минуя нежелательный код.

IDA ошиблась при нахождении границ функции main из-за антиотладочного механизма -злоупотребление инструкцией возврата. Но на рисунке 7 видно, что return мешает доконца распознать main. Это пример противодействия статическому исследованию.

Изображение выглядит как стол

Автоматически созданное описание

Рисунок 7 – Ошибка распознавания границ main до конца.

Из-за этого программа определяла это место как оборвавшуюся на середине функцию и не анализировала код дальше. Это было устранено вручную, путем ручного определения границ функции (рисунок 8). Затем функция main была декомпилирована в псевдокод для удобного анализа.

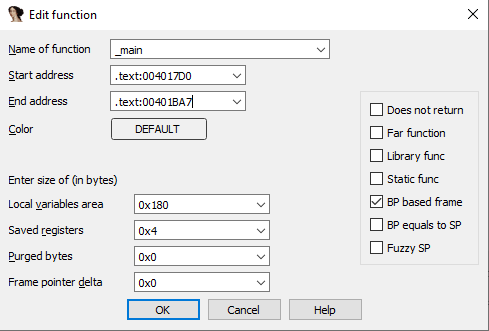


Рисунок 8 – Изменение границ функции.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 9 - Получившийся main.

Затем был произведён дальнейший анализ кода. Выяснилось, что проверка пароля выполняется в процедуре sub\_401450. При этом до этого выполняется проверка, что введённая строка больше 37 и не превышает длину в 255 символов, и что последний символ строки равен ‘b’.

Процедура проверки пароля также была декомпилирована. Псевдокод процедуры проверки пароля приведён на рисунке 10.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 10 – Проверка введенного пароля.

Введенный пароль конкатенируется в конец строки SЌQerR….

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 11 – Конкатенация статической строки с введенным паролем.

Процедура проверки сводятся к посимвольному изменению некоторых символов введенного пароля. Затем сравниваются первые 37 символов с присоединенным (и преобразованным) паролем пользователя. Если все символы совпадают, то проверка считается успешно пройденной. Для получения правильного пароля процедура преобразования была проведена в обратном порядке над эталонной строкой.

Было найдено несколько вариантов верных паролей (рисунки 12-15).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 12 – Первый пароль.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 13 – Второй пароль.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 14 – Третий пароль.



Рисунок 15 – Четвертый пароль.

3322,3323,3320,3321 — ключи, полученные преобразованием над строкой (сумма значений всех символов), а 1 — фиксированная часть.

Изображение выглядит как стол

Автоматически созданное описание

Рисунок 16 - Вывод секретных ключей.

В программе также были обнаружены другие защитные механизмы: так, например, программа получает список названий всех окон в системе и ищет в них ключевые слова (например, IDA). При обнаружении таких окон в консоль выводится текстовое предупреждение с требованием закрыть окно, однако это никак не влияет на дальнейший ход программы. На рисунке 17 представлено место вызова этой проверки. На рисунке 18 — код самой проверки.



Рисунок 17 – Вызов проверки окон.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 18 – Начало функции с проверкой окон.

Также существует функция проверки целостности процедуры проверки пароля. Например, если поставить программный breakpoint, то контрольная сумма будет нарушена и произойдет деление на ноль. На рисунке 19 и 20 видно, что если мы выставим breakpointы в проверяющую процедуру, то контрольная сумма изменится и произойдёт исключение (деление на 0).

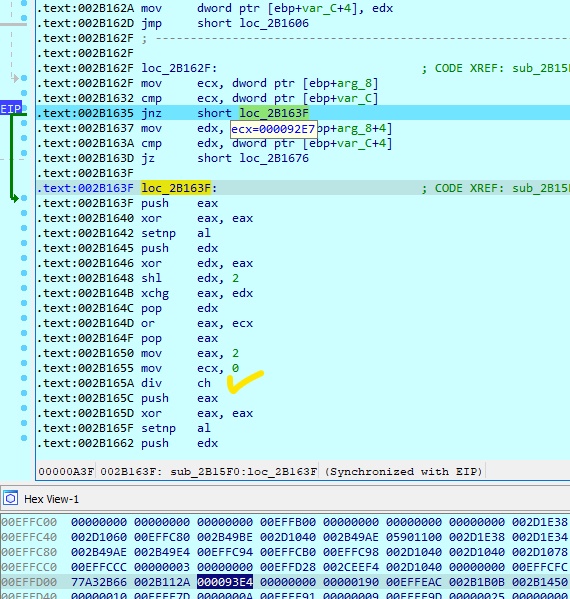


Рисунок 19 – Провал проверки контрольной суммы.

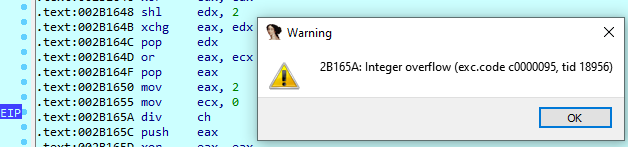


Рисунок 20 - Исключение.

Были реализованы скрипты согласно требованиям задания. Скрипт №1, а также скрипты №3-5 построены по одинаковому принципу и заключаются в прохождении по всем инструкциям программы и поиске среди них искомых инструкций (xor, rdtsc, int 2d и т.д.). Ниже приведены исходные коды скриптов.

1. Поиск всех инструкций xor.

from idaapi import \*

from idautils import \*

from idc import \*

start\_address = inf\_get\_min\_ea()

if start\_address == BADADDR:

qexit(1)

print("Start")

cnt = 0

for address in Heads(get\_segm\_start(start\_address), get\_segm\_end(start\_address)):

if is\_code(get\_full\_flags(address)):

if generate\_disasm\_line(address,0).startswith("xor "):

cnt+=1

print ("Found ", cnt," xors")

print ("End")

1. Нахождение закодированных строк.

from idaapi import \*

from idautils import \*

from idc import \*

import string

CODE\_REFERENCE = "Code"

DATA\_REFERENCE = "Data"

start\_address = inf\_get\_min\_ea()

if start\_address == BADADDR:

qexit(1)

print ("Start")

cnt = 0

good=0

string\_tmp=""

string\_candidates=list()

#Get list of segments (sections) in the binary image

for segea in Segments():

for funcea in Functions(segea, get\_segm\_end(segea)):

functionName = get\_func\_name(funcea)

for (startea, endea) in Chunks(funcea):

for address in Heads(startea, endea):

try:

if is\_code(get\_full\_flags(address)):

if "mov" == print\_insn\_mnem(address) and print\_operand(address, 1) == "0" and good > 3:

string\_candidates.append(string\_tmp)

good = 0

string\_tmp=""

cnt+=1

if "mov" == print\_insn\_mnem(address) and idc.get\_operand\_type(address,1) == 5:

string\_tmp+=chr(int(print\_operand(address, 1)[:-1],base=16) ^ 0x22)

good+=1

else:

good = 0

string\_tmp=""

except Exception:

continue

print ("Found ", cnt," xored strings")

print (string\_candidates)

print ("End")

1. Нахождение мест чтения PEB в коде.

Смещение TEB 0x30, то есть адрес FS: [0x30] содержит указатель на PEB, то есть адрес, где и будет лежать структура PEB:

struct \_PEB {

0x000 BYTE InheritedAddressSpace;

0x001 BYTE ReadImageFileExecOptions;

0x002 BYTE BeingDebugged; -- используется isDebugPresent

0x003 BYTE SpareBool;

from idaapi import \*

from idautils import \*

from idc import \*

start\_address = inf\_get\_min\_ea()

if start\_address == BADADDR:

qexit(1)

print ("Start")

cnt = 0

for address in Heads(get\_segm\_start(start\_address), get\_segm\_end(start\_address)):

if is\_code(get\_full\_flags(address)):

if "mov" in print\_insn\_mnem(address) and ("fs:30" in print\_operand(address, 1) or "fs:[30" in print\_operand(address, 1)):

print(address," : ",generate\_disasm\_line(address,0))

cnt+=1

print ("Found ", cnt," PEB reads")

print ("End")

1. Нахождение мест замера времени (rdtsc, GetTickCount).

from idaapi import \*

from idautils import \*

from idc import \*

start\_address = inf\_get\_min\_ea()

if start\_address == BADADDR:

qexit(1)

print ("Start")

cnt = 0

for address in Heads(get\_segm\_start(start\_address), get\_segm\_end(start\_address)):

if is\_code(get\_full\_flags(address)):

if "rdtsc" in print\_insn\_mnem(address) or "GetTickCount" in print\_insn\_mnem(address):

print(address," : ",generate\_disasm\_line(address,0))

cnt+=1

print ("Found ", cnt," time measurments")

print ("End")

1. Нахождение мест использования «редких» инструкций (int 2d, ss, и т.д.).

from idaapi import \*

from idautils import \*

from idc import \*

start\_address = inf\_get\_min\_ea()

if start\_address == BADADDR:

qexit(1)

print ("Start")

cnt = 0

for address in Heads(get\_segm\_start(start\_address), get\_segm\_end(start\_address)):

if is\_code(get\_full\_flags(address)):

if ("int" == print\_insn\_mnem(address) and "2d" == print\_operand(address, 0)) or ("ss" == print\_insn\_mnem(address)):

print(address," : ",generate\_disasm\_line(address,0))

cnt+=1

print ("Found ", cnt," rare instructions")

print ("End")

1. Нахождение других признаков механизмов защиты бинарного кода.

Данный скрипт ищет вызов функции OutputDebugString, которая часто применяется для проверки факт наличия отладчика.

from idaapi import \*

from idautils import \*

from idc import \*

start\_address = inf\_get\_min\_ea()

if start\_address == BADADDR:

qexit(1)

print ("Start")

for address in Heads(get\_segm\_start(start\_address), get\_segm\_end(start\_address)):

if is\_code(get\_full\_flags(address)):

if "call" in print\_insn\_mnem(address) and "OutputDebugString" in print\_operand(address, 0):

print(address," : ",generate\_disasm\_line(address,0))

cnt+=1

print ("Found ", cnt," OutputDebugString")

print ("End")

Разработанные скрипты были протестированы на обоих исследуемых исполняемых файлах. В первом файле были успешно найдены и раскодированы строки, найдено одно чтение PEB, два замера времени, а также одна проверка на наличие отладчика с помощью функции OutputDebugString(). Во втором исполняемом файле все скрипты дали отрицательный результат.

Затем был исследован файл crackme1\_crack\_me.exe из bin2\_\_crackme2021. Он использовал 13 байтов своих собственных начальных инструкций и c ними делал операцию AND 7. Каждый преобразованный байт был позицией в слове “BGOTHXIY”. Таким образом, создавался перестановочный шифр (BXXGYYYBGIBXX). Сравнение было посимвольное с введенным словом.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 21 - Байты начальных инструкций.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 22 - Шифр-код в 2 crackme.

Было получено решение для первой программы с помощью среды символьного выполнения angr. Для этого в angr были указаны адреса начала и конца функции, выполняющей непосредственную проверку пароля, а также адреса в этой функции, сигнализирующие об неуспешной проверке. Была задана строка для поиска в символьном виде, с ограничениями, что каждый символ строки должен быть одним из печатаемых латинских символов. Также была вручную ограничена длина строки. Решение было найдено за несколько секунд и имеет следующий вид: “SuperSe`retBackdnnrP`ssvnrd0102446789b”. Скрипт для поиска ключа приведён ниже.

import angr

import claripy

import logging

def main():

flag = claripy.BVS('flag', 37 \* 8, explicit\_name=True) # 37 мин размер длины пароля

buf = 0x606000 # место в памяти для буффера

func = 0x401450 # функция проверки

goal = 0x004015E8 # выход з функции

avoids = [0x004014B7, 0x004015B4,0x00401AE0] # 1 - проверка на длину; 2 - если проверка байтов не прошла v2=0; 3- избегать ветки, где нет b на конце

proj = angr.Project(r".\crackme1.exe")

state = proj.factory.blank\_state(addr=func)

state.memory.store(buf, flag, endness='Iend\_BE')

state.stack\_push(buf)

state.stack\_push(buf)

state.memory.store(buf + 37, b'\0')

for i in range(36):

state.solver.add(flag.get\_byte(i) >= 0x30) # диапозон символов для пароля

state.solver.add(flag.get\_byte(i) <= 0x7f)

simgr = proj.factory.simulation\_manager(state)

simgr.explore(find=goal, avoid=avoids)

found = simgr.found[0]

print(found.solver.eval(flag, cast\_to=bytes))

return 1

logging.getLogger('angr.sim\_manager').setLevel(logging.DEBUG)

main()

**Результаты работы**

Таким образом, были статически и динамически проанализированы 2 исполняемых файла. В первом файле было найдено несколько средств защиты, препятствующих анализу и отладке приложения. Во втором файле был обнаружен перестановочный шифр. Для обоих программ были найдены верные пароли, а так получен алгоритмы их работы.

Были написаны скрипты для IDAPython согласно заданию. Скрипты были протестированы на исследуемых исполняемых файлах и была продемонстрирована их работоспособность.

Был написан скрипт для средства символьного выполнения angr, позволивший найти пароль для первой программы за несколько секунд, однако требующий предварительного анализа программы.

**Вывод**

Были изучены различные методы защиты исполняемых файлов от их статического и динамического анализа. Были также изучены способы поиска и обхода рассмотренных способов защиты. Были получены навыки работы в популярных средствах анализа исполняемых файлов: IDA и angr.