Министерство образования и науки Российской Федерации
Санкт-Петербургский Политехнический Университет Петра Великого
—
Институт компьютерных наук и кибербезопасности

ЛАБОРАТОРНАЯ РАБОТА № 2

«Изучение принципов поиска уязвимостей в программном обеспечении без исходных кодов»

по дисциплине «Модели безопасности компьютерных систем»

Выполнила
студентка гр. 5131001/10302 Куковякина Д. А.

<подпись>

Преподаватель Овасапян Т. Д.

<подпись>

Санкт-Петербург
2024

ПОСТАНОВКА ЗАДАЧИ

Цель работы – Изучение типовых ошибок и принципов поиска уязвимостей в программном обеспечении без исходных кодов.

Задачи:

1) реализовать программу, осуществляющую фаззинг формата файла;

2) разработать IDC/IDAPython-скрипт, осуществляющий в программе поиск функций ввода данных и небезопасные функции;

3) осуществить поиск уязвимости в выданной программе.

ХОД РАБОТЫ

Теоретические сведения

Тестирование программного обеспечения (Software Testing) — проверка соответствия реальных и ожидаемых результатов поведения программы.

Цель тестирования — проверка соответствия ПО предъявляемым требованиям, обеспечение уверенности в качестве ПО, поиск очевидных ошибок в программном обеспечении, которые должны быть выявлены до того, как их обнаружат пользователи.

Типовые ошибки в программном обеспечении:

1. Целочисленное переполнение (когда результат арифметической операции не умещается в выделенный для него диапазон целых чисел);

2. Отсутствие проверки длины копируемых данных (при копировании данных в буфер не проверяется их размер, что может привести к перезаписи памяти за пределами буфера);

3. Утечки памяти (программное обеспечение выделяет память, но не освобождает её после использования, что приводит к утечкам памяти и увеличению использования ресурсов системы);

4. Деление на 0 (может возникнуть при вычислениях, которые в итоге приводят к делению на 0);

5. Рекурсивные ошибки (неправильное использование рекурсии может привести к переполнению стека вызовов функций).

Традиционная практика тестирования предполагает, что на вход программы подаются самые разные данные: корректные, почти корректные и граничные случаи. Это позволяет убедиться, что код работает в целом правильно и серьёзных ошибок в логике работы нет. Однако при этом качество проверки зависит от качества тестов и их количества. Фиксированное количество тестов не может покрыть все возможные варианты входных данных. Поэтому всегда есть вероятность, что какой-то важный случай был упущен, не вошёл в фиксированный набор тестов, а код,

следовательно, всё-таки содержит ошибку, которую достаточно талантливый хакер сможет проэксплуатировать.

Фаззинг — это техника автоматизированного тестирования, при которой на вход программе подаются специально подготовленные данные, которые могут привести её к аварийному состоянию или неопределённому поведению. Фаззеры по операциям, которые будут совершаться над входными данными, делятся на:

– Mutation-Based Fuzzers: Этот тип фаззера проще всего создать, поскольку он изменяет существующие образцы данных для создания новых тестовых данных;

– Generation-Based Fuzzers: Этот тип фаззера создает новые тестовые данные на основе входной модели. Обычно он разбивает протокол или формат файла на фрагменты, которые затем выстраиваются в допустимом порядке, и эти фрагменты случайным образом распределяются независимо друг от друга.

Также их можно разделить по наличию обратной реакции от тестируемого приложения: feedback driven и not feedback driven.

Feedback-driven fuzzing — это вид фаззинга, при котором фаззер изменяет входные данные так, чтобы их обработка затрагивала как можно больше участков кода программы. Работа таких фаззеров возможна, благодаря их способности реагировать на отклик (feedback) программы. Обычно таким откликом является покрытие кода. Метрики покрытия кода отслеживают суммарное количество выполненных строк кода, базовых блоков, количество сделанных условных переходов. Задача фаззера — генерировать данные, которые приводят к увеличению покрытия кода.

Реализация фаззера

Функционал фаззера состоит из двух основных пунктов:

- Дозапись случайных байт в конец файла;
- Последовательное изменение байт в файле.

При выборе пункта «дозапись» необходимо указать количество тестов. В каждом из тестов дописывается $2^{номер\_теста}$ байт, байты генерируются случайно из диапазона английского алфавита нижнего регистра.

При выборе пункта «замена» указывается последний байт, до которого необходимо произвести замену. Затем вводится сам байт, на который будет заменяться, и количество байт, которые будут заменяться в рамках одного теста.

В рамках каждого теста происходит запуск исполняемого файла и подсчет покрытия кода при помощи динамической бинарной инструментации.

Динамическая бинарная инструментация (Dynamic Binary Instrumentation, DBI) заключается во вставке в бинарный исполняющийся код анализирующих процедур. Основное преимущество данного подхода заключается в том, что нет необходимости в исходном коде анализируемого приложения – работа происходит непосредственно с бинарным файлом.

За вставку дополнительного кода обычно отвечают инструментирующие процедуры, которые вызываются только раз при возникновении необходимого события и модифицируют целевую программу. Добавленный код представляет собой анализирующие процедуры. Эти процедуры отвечают за проведение необходимого анализа, модификации и мониторинга исследуемой программы и вызываются каждый раз при достижении определенного участка кода или возникновения в программе определенного события (создание процесса, возникновения исключения и т. д.).

Для реализации использовалась утилита DynamoRio, а именно её компонент drcov. Она генерирует файл с логами, содержащий следующие данные: информация о загружаемых модулях и таблицу базовых блоков, которая содержит список основных блоков, которые были выполнены при сборе информации о покрытии. Базовым блоком является последовательность инструкций, не содержащая инструкций передачи

управления. Для анализа покрытия было взято эталонное значение количества блоков – количество блоков при выполнении файла с правильными конфигурационными данными. Затем после внесения изменений считывается количество блоков и вычисляется их отношение. В случае если отношение близко к 1 (то есть код прошел столько же и больше блоков), то изменение сохраняется в файле, иначе конфигурационный файл откатывается до предыдущего состояния.

Далее рассмотрим случай выполнения программы с ошибкой. Для отслеживания используется структура DebugEvent, а именно её поле dwDebugEventCode, которое сообщает о возникновении исключения. Код исключения сохраняется в .u.Exception.ExceptionRecord.ExceptionCode, были обработаны основные исключения такие как переполнение, деление на ноль и т.д.

В случае возникновения исключения текущий конфигурационный буфер сохраняется в файл с логами, туда же записывается само исключение. Затем обрабатываются значения регистров (Wow64GetThreadContext) и состояние стека (ReadProcessMemory).

Состояние регистров и стека могут быть полезны при возникновении ошибок. Выводятся следующие значения:

- Eax – используется для арифметических операций и хранения результата;
- Ebx – используется как указатель на данные в памяти;
- Ecx – используется в циклах;
- Edx – используется для хранения данных и расширенных результатов умножения и деления;
- Esp – указатель на вершину стека;
- Ebp – используется для доступа к параметрам функции и локальным переменным;
- Esi – используется как источник данных для операций копирования и сравнения;

- Edi – используется как назначение для операций копирования и сравнения;
- Eflags – регистр флагов (Carry flag – результат больше максимального значения, Zero flag – результат равен 0, Sign flag – в соответствии со знаковым битом результата последней операции, Overflow flag – переполнение, Parity flag – 8 младших разрядов содержат четное число единиц).

Далее представлен пример работы программы (рисунок 1 – 5).

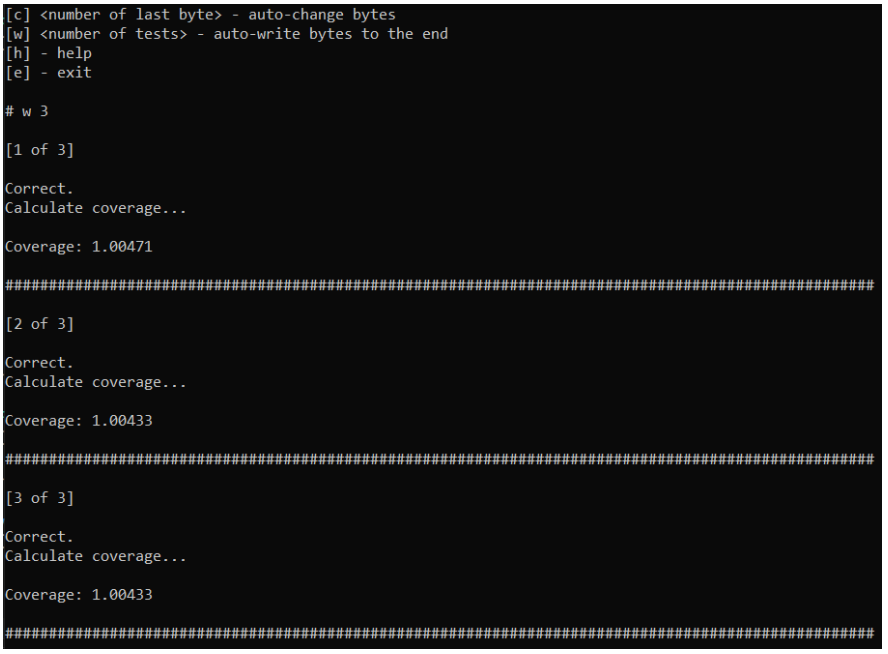

Рисунок 1 – Дозапись в файл

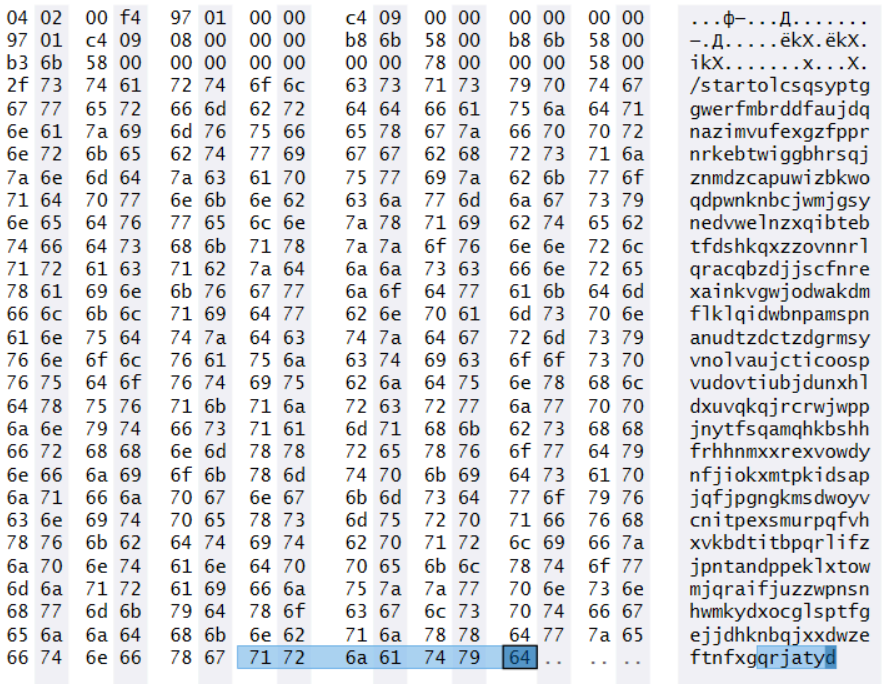

Рисунок 2 – 7 дописанных байт

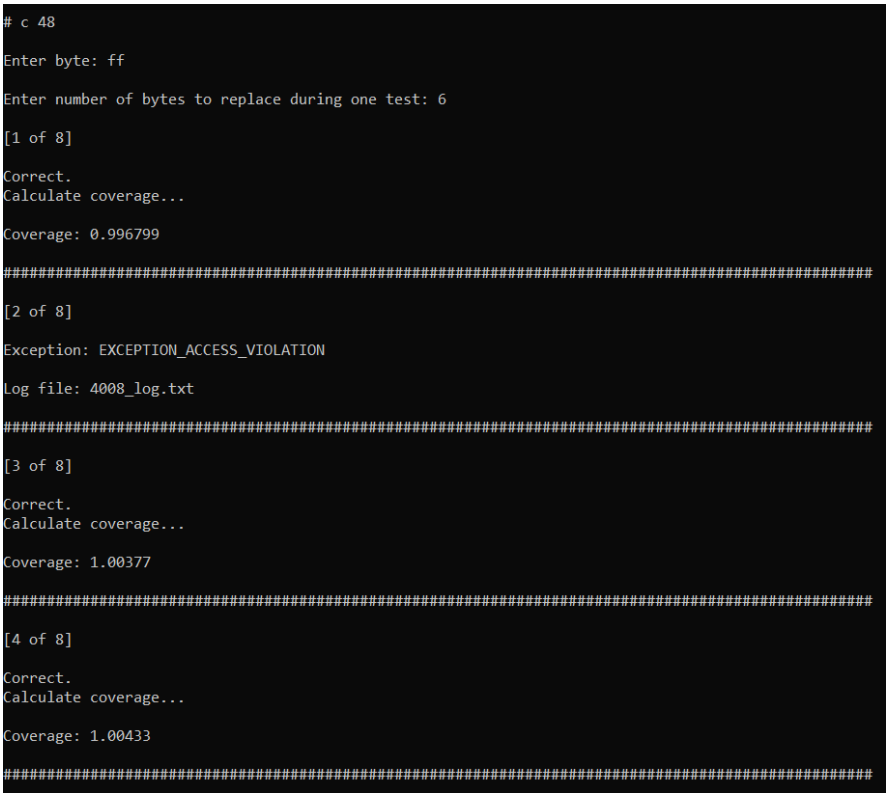

Рисунок 3 – Замена байт заголовка на FF

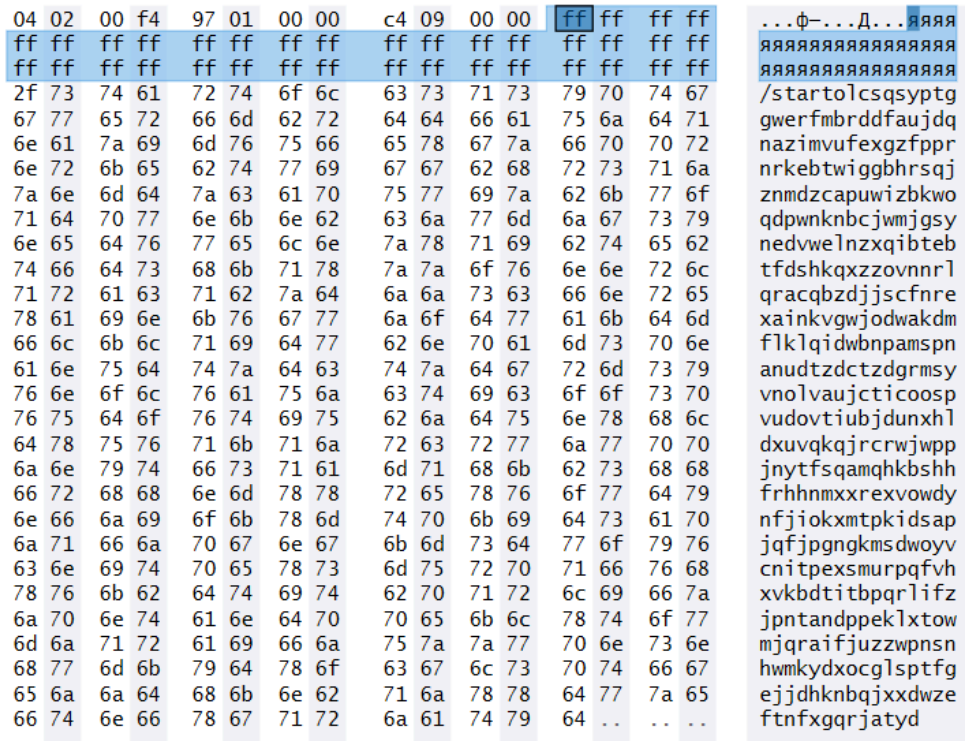

Рисунок 4 – Измененные байты

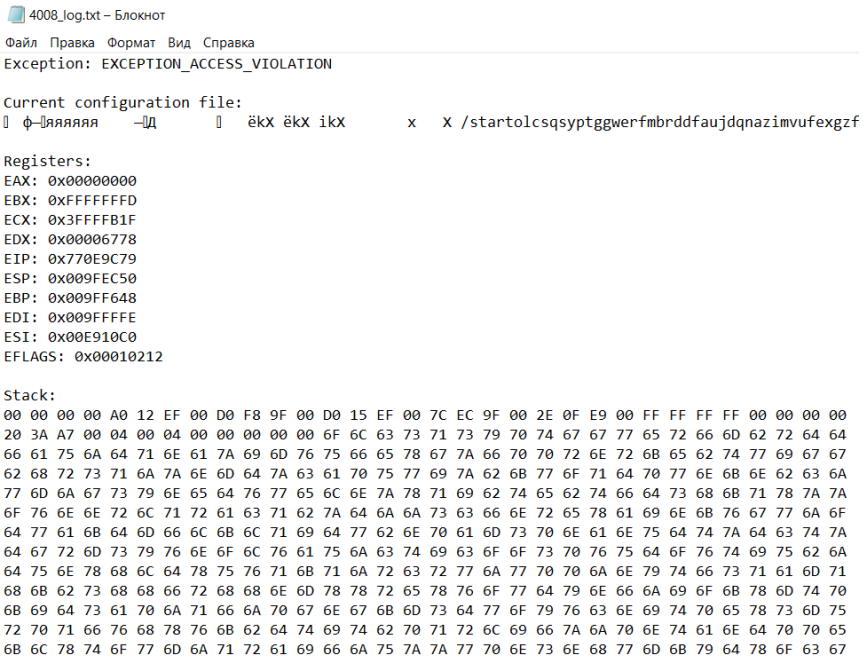

Рисунок 5 – Пример файла с логами

Скрипт

Далее было необходимо написать скрипт IDApython, который будет искать в программе функции ввода данных и вызов небезопасных функций и выводить адрес, откуда они вызываются. Для этого были использованы библиотеки idc, idautils, idaapi. А именно следующие функции:

- idautils.Heads(start, end) – получение списка заголовков (инструкций или данных)
- ida_ua.ua_mnem(ea) – предназначена для получения мнемоники инструкции в ассемблерном коде, мнемоника — символическое имя для одной исполняемой инструкции машинного языка
- print_operand(ea, n) – получение операнда инструкции, n – номер операнда

В результате запуска скрипта был получен следующий вывод (рисунок 6).

```
Analyze...
0x0040154e: call sprintf
0x004015af: call memset
0x004015cb: call strncpy
0x00401676: call malloc
0x0040169a: call fread
0x004016c7: call memcpy
0x00401d6f: call calloc
0x00401e3f: call free
0x00401fa4: call memcpy
0x00401fee: call memcpy
0x0040251d: call malloc
0x004027bf: call free
0x004027cf: call free
0x00402801: call realloc
0x0040289f: call free
0x004028e8: call memcpy
0x00402a69: call memcpy
0x00402b42: call malloc
0x00402c1f: call free
0x00402c34: call free
0x00402c5a: call memcpy
0x00402fd8: call realloc
0x00403009: call free
0x0040315a: call realloc
0x0040327a: call free
0x004035a6: call malloc
0x004035c9: call memcpy
0x00403662: call free
0x004036e6: call free
0x0040599b: call atoi
0x00405a45: call atoi
0x00405ba2: call atoi
0x00405cc9: call atoi
0x00405d3b: call atoi
0x0040705d: call memcpy
0x0040787f: call malloc
0x00407a23: call memcpy
Done
```

Рисунок 6 – Результат работы скрипта

Поиск уязвимости

Проанализируем функцию, отвечающую за парсинг полученного буфера, уже без заголовка (рисунок 7). В ней происходит рекурсия – проверяется наличие в буфере подстроки /start, если она есть, то parse_string вызывается еще раз с обрезанным буфером. Если же подстрока не найдена, то исполнение переходит к vuln_func, в которую передаются текущий буфер, shell_len, dst_len.

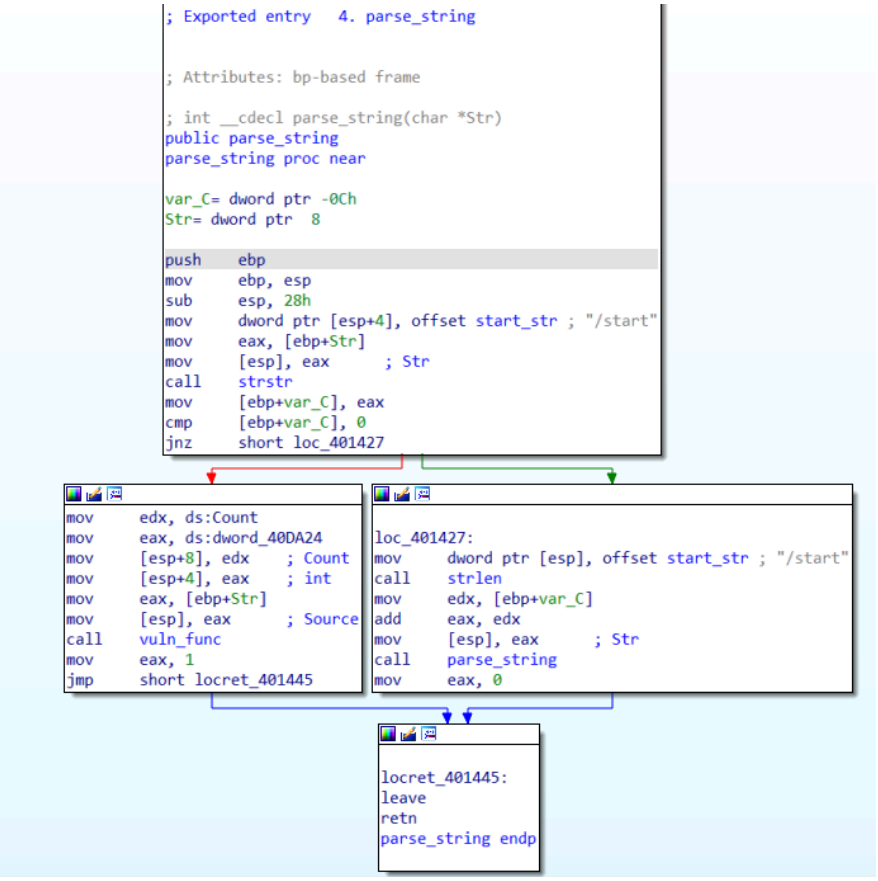

Рисунок 7 – Функция parse_string

На рисунках 8 и 9 представлен ассемблерный код и псевдокод этой функции соответственно. По коду видно, что создается буфер фиксированного размера – 2508. Затем он заполняется символами b (код 98). После чего при помощи strncpy происходит копирование из текущего буфера в этот созданный, при этом в качестве размера указывается переданный параметр dst_len. Исходя из этого становится ясно, что уязвимость программы связана с функцией strncpy.

```
                public vuln_func
vuln_func       proc near               ; CODE XREF: parse_string+3B↑p

Destination     = byte ptr -9CCh
Source          = dword ptr  8
Count           = dword ptr  10h

                push    ebp
                mov     ebp, esp
                sub     esp, 9E8h
                mov     dword ptr [esp+8], 9C4h ; Size
                mov     dword ptr [esp+4], 62h ; 'b' ; Val
                lea     eax, [ebp+Destination]
                mov     [esp], eax      ; void *
                call    memset
                mov     eax, [ebp+Count]
                mov     [esp+8], eax    ; Count
                mov     eax, [ebp+Source]
                mov     [esp+4], eax    ; Source
                lea     eax, [ebp+Destination]
                mov     [esp], eax      ; Destination
                call    strncpy
                lea     eax, [ebp+Destination]
                mov     [esp+4], eax
                mov     dword ptr [esp], offset aBufferS ; "buffer: %s\n"
                call    printf
                leave
                retn
vuln_func       endp
```

Рисунок 8 – Код vuln_func

```
int __cdecl vuln_func(char *Source, int a2, size_t Count)
{
  char Destination[2508]; // [esp+1Ch] [ebp-9CCh] BYREF

  memset(Destination, 98, 0x9C4u);
  strncpy(Destination, Source, Count);
  return printf("buffer: %s\n", Destination);
}
```

Рисунок 9 – Псевдокод vuln_func

Заменим в заголовке dst_len на 2510 (рисунок 10).

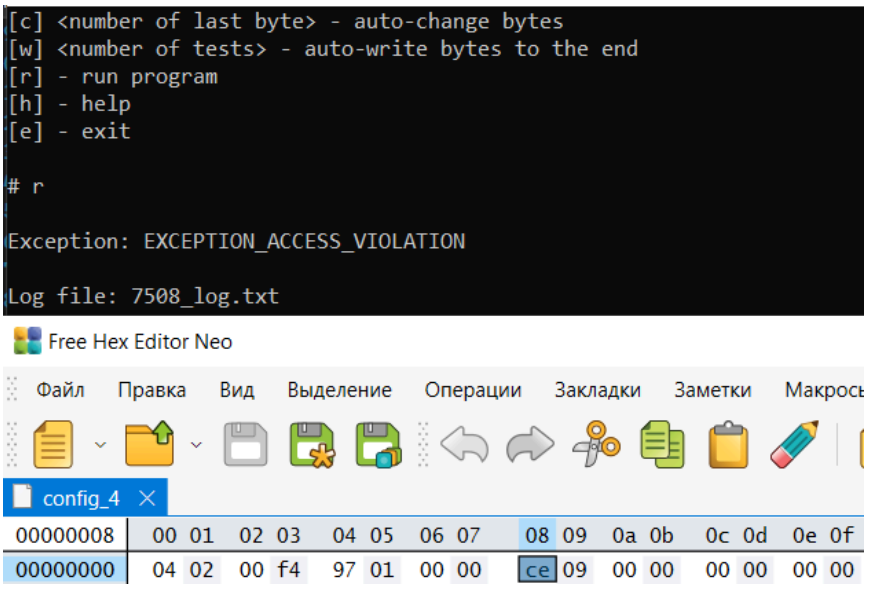

Рисунок 10 – Проверка уязвимости

На рисунке 11 представлено состояние стека и регистров на момент возникновения ошибки, полученное из IDA.

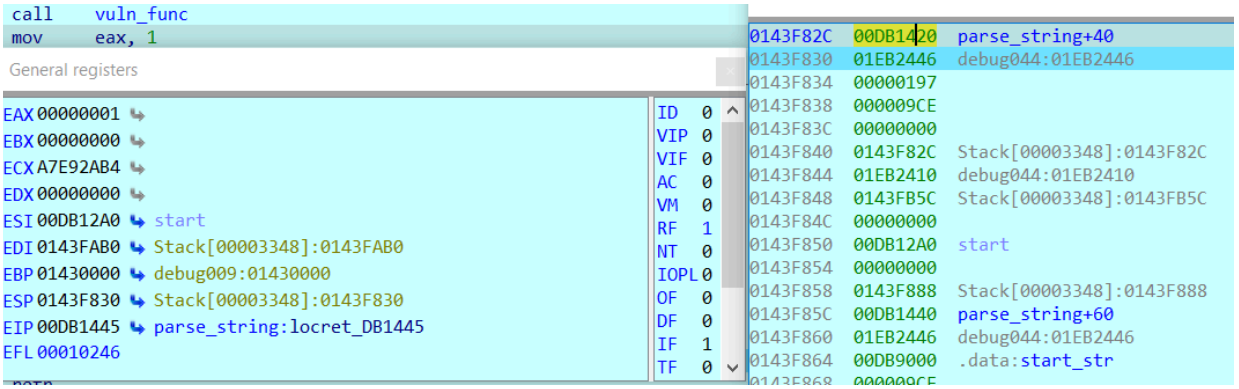

Рисунок 11 – Состояние регистров и стека

Анализ уязвимости

У функции strncpy можно выделить две основные уязвимости:

- не завершает буфер, в который копируется строка, null-символом;
- не осуществляет проверку параметра, переданного в качестве размера.

Рассмотрим два случая: изменение размера src буфера и изменение параметра size.

Исходный буфер всегда завершается нулем, даже если файл нулем не завершается (рисунок 12). В таком случае, если буфер-источник меньше или равен буферу-назначения, то первая уязвимость не возникнет. Однако, если

буфер-источник будет больше буфера-назначения (при этом параметр size имеет правильное значение), то символ завершения строки не будет добавлен, что может спровоцировать непредвиденное поведение, например, при вызове функции printf, которая выводит строку до встреченного нулевого символа.

```
Buffer = (char *)malloc(ElementCount + 1);
fread(Buffer, 1u, ElementCount, Stream);
Buffer[ElementCount] = 0;
if ( ElementCount > 0x2F )
  memcpy(a2, Buffer, 0x30u);
*a3 = Buffer + 48;
```

Рисунок 12 – Добавление нулевого символа

Для устранения данной уязвимости стоит использовать более безопасные с этой точки зрения функции, например strlcpy или snprintf, которые при копировании строк будут завершать буфер-назначение нулем.

При изменении параметра size до значения меньше, чем размер буфера-источника, может возникнуть такая же ситуация (рисунок 13). Тут также произойдет копирование строки без завершающего символа, что приведет к выходу за границы самого буфера при его печати.

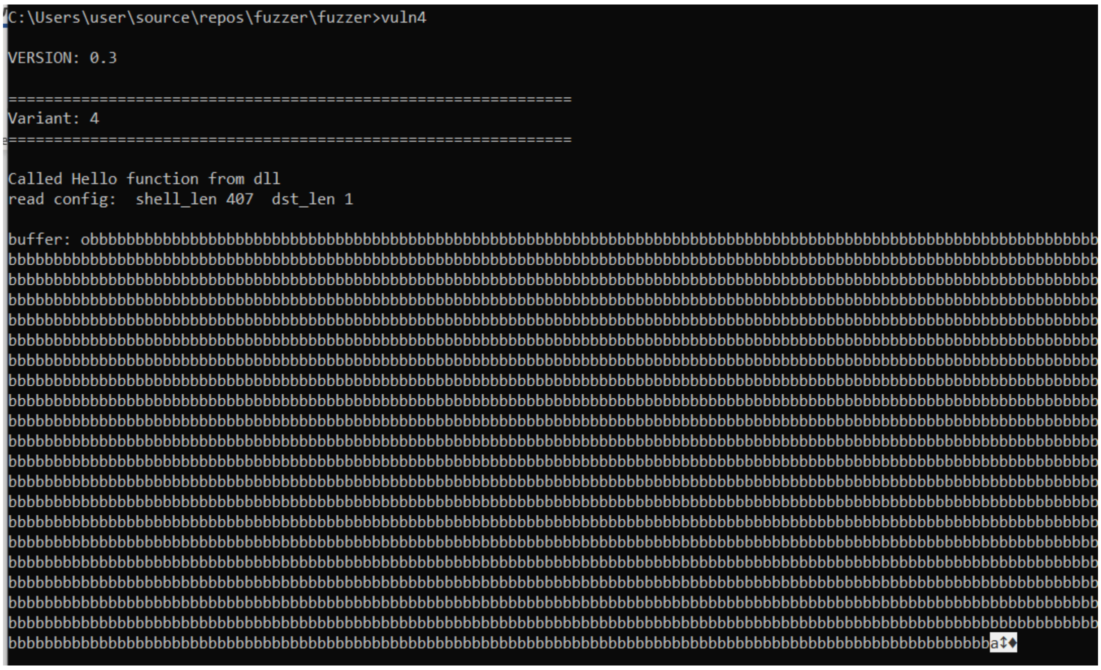

Рисунок 13 – Вывод "мусора"

При увеличении значения size произойдет перезапись данных, идущих после буфера-назначения. Данные могут быть заменены как нулевыми

символами (в случае если буфер-источник меньше size), так и значениями из буфера-источника, если позволяет его размер (рисунок 14 и 15). Для устранения данной уязвимости необходимо проверять считанное значение dst_len и при необходимости уменьшать его.

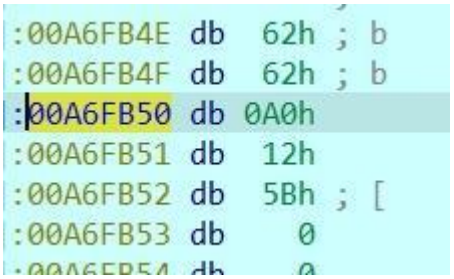

Рисунок 14 – Память до strncpy

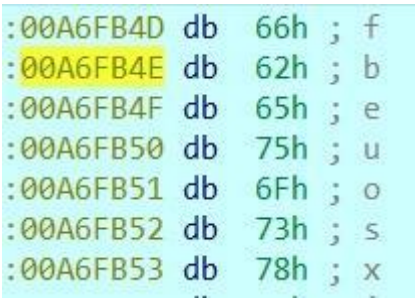

Рисунок 15 – Память после strncpy

После области памяти, хранящей данный буфер, идут два специальных поля "r" и "s", которые представляют собой обратный адрес и сохраненные регистры (рисунок 16). Поэтому при перезаписи данных полей возникает исключение, так как был затерт адрес возврата.

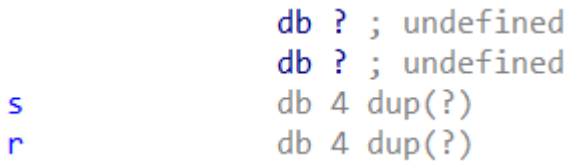

Рисунок 16 – Специальные поля

ВЫВОД

В ходе работы были изучены типовые ошибки и принципы поиска уязвимостей в программном обеспечении без исходных кодов. Была написана программа на языке C++, осуществляющая автоматический фаззинг формата-файла, и IDAPython-скрипт, осуществляющий поиск небезопасных функций.