1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт компьютерных наук и технологий
5. **Кафедра «Информационная безопасность компьютерных систем»**

**ЛАБОРАТОРНАЯ РАБОТА № 2**

1. «**Изучение принципов поиска уязвимостей в программном обеспечении без исходных кодов**»
2. по дисциплине «Модели безопасности компьютерных систем»
3. Выполнил
4. студент гр. 33508/3 Князев П.В.

<*подпись*>

1. Преподаватель
2. Жуковский Е.В.

<*подпись*>

1. Санкт-Петербург
2. 2018

# Task

В рамках лабораторной работы необходимо получить в соответствии со своим вариантом исполняемый файл, конфигурационный файл и динамические библиотеки.

В ходе выполнения лабораторной работы необходимо выполнить следующие действия:

1. Изучить основные типы ошибок в программном обеспечении (целочисленное переполнение, отсутствие проверки длины копируемых данных, переполнение буфера и другие).
2. Получить у преподавателя файлы в соответствии со своим вариантом.
3. Реализовать программу, осуществляющую фаззинг формата файла.
4. Реализованная программа должна осуществлять следующие действия:

* осуществлять изменение оригинального файла (однобайтовая замена, замена нескольких байт, дозапись в файл);
* заменять байты на граничные значения (0x00, 0xFF, 0xFFFF, 0xFFFFFF, 0xFFFFFFFF, 0xFFFF/2, 0xFFFF/2+1, 0xFFFF/2-1 и т.д.);
* иметь автоматический режим работы, при котором производится последовательная замена байт в файле;
* находить в файле символы, разделяющие поля (“,:=;”);
* расширять значения полей в файле (дописывать в конец, увеличивать длину строк в файле);
* находить границы полей в файле на основании анализа нескольких конфигурационных файлов;
* осуществлять запуск исследуемой программы;
* обнаруживать возникновение ошибки в исследуемом приложении;
* получать код ошибки и состояния стека, регистров и другую информацию на момент возникновения ошибки;
* логировать в файл информацию о произошедших ошибках и соответствующих им входных параметрах (произведенные замены).

1. Разработать IDC/IDAPython-скрипт, осуществляющий следующие действия:
   * поиск в программе функций ввода данных (fread, fscanf, read, fgets, …);
   * поиск вызовов небезопасных функций (strcpy, sprintf, strncpy, memcpy, memmove, …);
   * определение последовательности вызовов (трассы исполнения) от функций ввода данных до вызова небезопасных функций.
2. Разработанный IDC/IDAPython-скрипт должен выводить следующую информацию:
   * название найденной функции;
   * адрес, откуда вызывается данная функция.
3. Используя разработанную программу для фаззинга формата файлов осуществить поиск в выданной программе уязвимости (целочисленное переполнение, отсутствие проверки длины данных), приводящей к переполнению буфера.
4. Сформировать файл, приводящий к краху исследуемого приложения.
5. Изучить с использованием дизассемблера и отладчика структуру программы и найти участок кода с уязвимостью. Провести исследование найденной уязвимости и условий ее эксплуатации.

# Result

В результате работы на языке python была написана программа – фаззер. Для того, чтобы провести фаззинг формата файла, нужно передать на вход программе следующие параметры:

* Имя конфигурационного файла, который будет изменяться фаззером и подаваться на вход уязвимой программе
* Путь к папке с различными файлами такого же формата для проведения анализа
* Имя программы, которая подлежит тестированию

В одну папку со скриптом фаззера следует скопировать конфигурационный файл и уязвимую программу.

**Анализ формата файла**

Перед фаззингом необходимо сначала проанализировать похожие файлы и получить примерную структуру файла. К каждой области структуры будут применяться различные методы по изменению файла. Для этого была использована библиотека difflib, позволяющая находить наибольшие совпадающие последовательности байт. Сам анализ происходит в функции analyze (Рис. 1). Эта функция совершает следующие действия:

* Просмотр всех файлов в папке
* Сравнение основного конфигурационного файла с со всеми остальными файлами, найденными в папке
* Нахождения последовательностей байт, которые не изменяются от файла к файлу
* На основании полученной информации идет разделение файла на регионы: постоянные и изменяющиеся.
* Вывод полученной структуры на экран

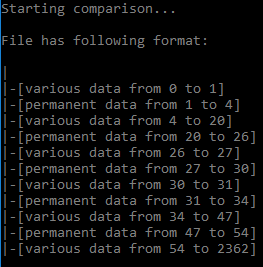


Рис. 1 «Получение структуры файла»

**Фаззинг**

Информация, полученная функцией analyze, используется для фаззинга различных регионов файла. Регионы с постоянными данными могут быть разделителями или особыми полями, удаление которых не принесет никакой выгоды. Потому для каждого из таких регионов используется следующий способ (fuzzStaticPieceOfFile):

Вход: регион файла R, последовательность байтов для фаззинга D

1. Если len(D) < len(R)
   1. I = 0
   2. Пока I < len(R) – len (D)
      1. Изменить данные R, начинающиеся с позиции R + I, на D
      2. Запуск программы и логирование (если произошел сбой)
      3. Восстановить предыдущее состояние
      4. I = I + 1

Регионы с непостоянными данными могут быть различной длины и содержать любые значения. Поэтому для каждого из таких регионов используется способ, описанный выше, и вдобавок метод специально для непостоянных данных (fuzzDynamicPieceOfFile). Его суть состоит в увеличении длины входных данных, что может привести к переполнению буфера.

**Запуск программы и обнаружение сбоев**

Для обнаружения сбоев в программе необходимо запустить её под отладкой и зарегистрировать callback – функции, которые будут вызваны при отладочных событиях. Для этого был создан класс ProcCrashTester, использующий библиотеку PyDbg. Принцип его работы состоит в следующем:

1. Инициализация (получение имени программы для запуска)
2. Запуск монитор – процесса. Его задача состоит в том, чтобы запустить процесс - отладчик и, если не произошел сбой после запуска, через некоторое время завершить запущенный процесс.
3. Запуск процесса – отладчика. Его задача состоит в том, чтобы запустить процесс под отладкой и настроить callback – функции, которая будет сохранять информацию о сбое если произойдет переполнение буфера, стека и т.д.

При сбое программы в папку Crash reports сохраняется копия конфигурационного файла, который приводит к сбою и контекст процесса в момент завершения: стек вызовов и регистры.

**Ручной режим работы**

Помимо автоматического режима работы программы – фаззера существует и ручной режим. Запускается он опцией --terminal. После включения этого режима на консоль выводится справка по использованию команд.

Поддерживаются следующие команды:

1. insert <start> <data> - вставка данных data, начиная с позиции start
2. insert <start> <hexbyte> <count> - вставка последовательности байт hexbyte длиной count, начиная с позиции start
3. change <start> <data> - замена исходных байт на байты data, начиная с позиции start
4. change <start> <hexbyte> <count> - замена исходных данных на последовательность байт hexbyte длиной count, начиная с позиции start
5. remove <start> <end> - вырезать данные из файла, начиная с позиции start и заканчивая позицией end
6. show <start> <end> - показать байты файла, начиная с позиции start и заканчивая позицией end
7. save <start> <end> - сохранить байты файла, начиная с позиции start и заканчивая позицией end
8. restore – восстановить байты, сохраненные командой save
9. run - запустить программу на исполнение и проверить на сбой
10. analyze – анализ формата файла
11. exit - выйти
12. help – справка

**Описание найденной уязвимости**

В варианте 23 была найдена уязвимость, заключающаяся в использовании небезопасной функции vsprintf. C помощью написанного IDAPython – скрипта удалось найти trace от функции ввода fread до вызова небезопасной функции:

from input func fread to unsafe func vsprintf:

\_main -> read\_config -> fread

\_main -> parse\_string -> vuln\_func -> vuln\_overflow\_vsprintf -> vsprintf

Уязвимость заключается в отсутствии проверки длины буфера a1. Максимально допустимая длина буфера 3000 байт. Если выйти за предел, то произойдет повреждение переменных и перезапись адреса возврата.

int vuln\_overflow\_vsprintf(char \*a1, ...)

{

char v2[3000];

va\_list va;

va\_start(va, a1);

memset(v2, ‘b’, 3000);

vsprintf(v2, a1, va);

return printf("buffer: %s\n", v2);

}

При этом Context dump выглядит следующим образом:

EIP: 48484848 Unable to disassemble at 48484848

EAX: 00000bd1 ( 3025) -> N/A

EBX: 00000000 ( 0) -> N/A

ECX: 758940fd (1971929341) -> N/A

EDX: 00000000 ( 0) -> N/A

EDI: 009ffea0 ( 10485408) -> 0P/}qu0Tt00tHwJlw0B/0PeuB/PPPjmw2tXXGJlwnw0 (stack)

ESI: 00ce12a0 ( 13505184) -> N/A

EBP: 48484848 (1212696648) -> N/A

ESP: 009ff040 ( 10481728) -> b .. b(stack)

+00: 00cea100 ( 13541632) -> N/A

+04: 00cc419e ( 13386142) -> H .. H (heap)

Видно, что адрес возврата был изменен на 48484848 (входные HHHH … ), а это говорит о том, что можно эксплуатировать данную уязвимость.

Данную уязвимость легко устранить путем использования функций vsnprintf, которая копирует n символов или vsprintf\_s.

# Conclusion

Ошибки переполнения буфера часто возникают при использовании низкоуровневых языков, таких как C и C++. Эти уязвимости могут быть эксплуатируемыми злоумышленником. Чтобы сократить риск эксплуатации уязвимости, нужно использовать secure – функции и проверку длины входных данных.

Приложение

IDAPython – скрипт

from idaapi import \*

def recWalk(function\_ea, level = 1):

"""

Walks around all functions, called from address function\_ea

"""

f\_start = function\_ea

f\_end = FindFuncEnd(function\_ea)

visitedLinks.append(f\_start)

print '%s Addr = %08x, Func = %s' % ('-' \* level, function\_ea, GetFunctionName(function\_ea))

# For each defined element in the function.

for head in Heads(f\_start, f\_end):

# If the element is an instruction

if isCode(GetFlags(head)):

# Get the references made from the current instruction

refs = CodeRefsFrom(head, 0)

if refs:

refs = [ x for x in refs if (x not in visitedLinks) and (x < f\_start or x > f\_end) ]

for r in refs:

func(r, level + 1)

def findPathToFunc(function\_ea, func\_to\_find, visitedLinks):

"""

Finds a way to function

function\_ea - address where the search will be started

"""

# print 'Found Addr = %08x, Func = %s' % (function\_ea, GetFunctionName(function\_ea))

# Found!

if function\_ea == func\_to\_find:

return GetFunctionName(function\_ea)

f\_start = function\_ea

f\_end = FindFuncEnd(function\_ea)

# Match address as visited to avoid a loop: func1 -> func2 -> func3 -> func1

visitedLinks.append(f\_start)

# For each defined element in the function.

for head in Heads(f\_start, f\_end):

# If the element is an instruction

if isCode(GetFlags(head)):

# Get the references made from the current instruction

# those can not be local

refs = CodeRefsFrom(head, 0)

if refs:

res = None

refs = [ x for x in refs if (x not in visitedLinks) and (x < f\_start or x > f\_end) ]

for r in refs:

res = findPathToFunc(r, func\_to\_find, visitedLinks)

if res is not None:

res = GetFunctionName(function\_ea) + ' -> ' + res

return res

return None

def trace(function\_ea, func\_to\_find):

visitedLinks = []

return findPathToFunc(function\_ea, func\_to\_find, visitedLinks)

def printCallSource(addr):

print 'Function [%s] %s is called from:' % (hex(addr), GetFunctionName(addr))

refs = CodeRefsTo(addr, 0)

for r in refs:

print "[%s] %s" % (hex(r), GetFunctionName(r))

print '\n'

inputFunctions = (

# stdin

'gets',

'scanf',

'getchar',

# file

'fgetc',

'getc',

'fread',

'fgets',

'fscanf'

)

unsafeFunctions = (

'memset',

'memcpy',

'memmove',

'strcpy',

'strcat',

'sprintf',

'vsprintf',

'sscanf',

'malloc',

'calloc',

'realloc',

'free'

)

def listFunctions(ea):

"""

From function list chose those are in

tuples unsafeFunctions and inputFunctions

"""

inputFuncMap = {}

unsafeFuncMap = {}

# Loop through all the functions

for function\_ea in Functions(SegStart(ea), SegEnd(ea)):

funcName = GetFunctionName(function\_ea)

funcAddr = function\_ea

if funcName in inputFunctions:

inputFuncMap[funcName] = funcAddr

elif funcName in unsafeFunctions:

unsafeFuncMap[funcName] = funcAddr

return inputFuncMap, unsafeFuncMap

def commonTrace(ea, addr\_from, addr\_to):

""" finds common traces for functions """

visitedLinks = trace(ea, addr\_to)

if visitedLinks is None:

return None, None

visited = visitedLinks.split(' -> ')

s1 = None

s2 = None

for v in reversed(visited):

s1 = trace(LocByName(v), addr\_from)

s2 = trace(LocByName(v), addr\_to)

if s1 is not None and s2 is not None:

return s1, s2

return None, None

# start here

# Wait until IDA has done all the analysis tasks.

# If loaded in batch mode, the script will be run before

# everything is finished, so the script will explicitly

# wait until the autoanalysis is done.

autoWait()

ea = ScreenEA()

print 'Script start'

input, unsafe = listFunctions(ea)

print '-' \* 60

print 'Input functions:'

for e in input.values():

printCallSource(e)

print '-' \* 60

print 'Unsafe functions:'

for e in unsafe.values():

printCallSource(e)

print '-' \* 60

print 'Traces:'

for name\_i, addr\_i in input.items():

for name\_u, addr\_u in unsafe.items():

s1, s2 = commonTrace(ea, addr\_i, addr\_u)

if s1 is None or s2 is None:

continue

print '-' \* 20

print 'from input func %s to unsafe func %s:' % (name\_i, name\_u)

print s1

print s2

print 'Script end'