Министерство науки и высшего образования Российской Федерации

Санкт-Петербургский Политехнический Университет Петра Великого

—

**Институт кибербезопасности и защиты информации**

**ЛАБОРАТОРНАЯ РАБОТА №2**

**Исследование подходов, применяемых при эксплуатации уязвимостей**

по дисциплине «Модели безопасности компьютерных систем»

Выполнил

студент гр. 4851001/80201 С.И. Ветров

<*подпись*>

Проверил

преподаватель Е.В. Жуковский

<*подпись*>

Санкт-Петербург

2021

# **Постановка задачи**

Изучение способов эксплуатации уязвимостей и механизмов противодействия.

В ходе выполнения лабораторной работы необходимо выполнить следующие действия:

1. Использовать полученные от преподавателя файлы для лабораторной работы №1 в соответствии со своим вариантом;
2. Получить вариант задания, в котором описан функционал полезной нагрузки;
3. Изучить защитные механизмы, затрудняющие эксплуатацию уязвимостей (DEP, ASLR, GS, Security Cookies, SMEP, SAFESEH, SEHOP и другие);
4. Изучить техники эксплуатации уязвимостей и способы обхода защитных механизмов (ret2libc, ROP, Stack Pivoting, Heap Spray и другие);
5. Используя скрипт mona.py для отладчиков Immunity Debugger/WinDBG или другой с аналогичной функциональностью осуществить получение списка загруженных в процесс модулей, не использующих технологию рандомизации адресов. Реализовать скрипт для отладчика;
6. Найти в динамических библиотеках без рандомизации адресов опкоды инструкций, необходимых для передачи управления (call esp, jmp esp, …) на полезную нагрузку. Реализовать для этих целей скрипт для отладчика (Immunity Debugger/WinDBG/IDA Pro);
7. Провести эксплуатацию найденной уязвимости, с использованием полезной нагрузки (shellcode) в соответствии с выданным вариантом;
8. Эксплуатацию уязвимости осуществлять под ОС семейства Windows версии не ниже Windows 7 x64. При эксплуатации запрещено изменение исполняемого файла, запуск сторонних исполняемых файлов, либо выполнение консольных команд. Полезная нагрузка должна быть написана на языке ассемблера, либо на языке C с использованием приемом разработки базонезависимого кода.

# **Ход работы**

# **Подмена адреса возврата**

Первым делом для эксплуатирования уязвимости в программе vuln.exe первым делом необходимо заменить адрес возврата из уязвимой функции. В первой лабораторной работе уже было реализовано переполнение буфера, однако из-за того, что адрес возврата затирался «мусорными» данными, то при переходе по такому произвольному адресу происходит выход с исключением.

Наша задача заключается в том, чтобы с помощью замена адреса возврата передать управление в стек, в котором будет размещен наш шелл-код (бинарные инструкции, являющиеся полезной нагрузкой).

Для этого необходимо определить адрес команды call esp или jmp esp в библиотеках, которые подтягивает при исполнении уязвимая программа. Для этих целей был реализован скрипт для программы immunity Debugger, который определяет адреса желаемых инструкций в dll файлах.



Рисунок 1 - Результаты работы скрипта

При анализе результатов работы скрипта важно брать во внимание, что в большинстве dll-библиотек, используемых программой подключен ASLR, что не дает возможность использовать абсолютные адреса инструкций, т.к. в данном случае из-за рандомизации адресов используется относительная адресация. Для того, чтобы определить, используется ли в библиотеке ASLR, также был реализован скрипт на Python для immunity Debugger. Ниже приведен результат вывода скрипта.

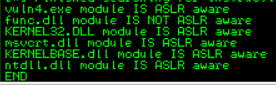


Рисунок 2 - Результат работы скрипта

В результате было принято решение использовать инструкция jmp esp, т.к. call esp присутствует только в библиотеках, что поддерживают ASLR. Вместо адреса возврата запишем адрес функции jmp esp в библиотеке func.dll. Адрес этой инструкции в шестнадцатеричном виде выглядит как “65 50 12 97”. Эти байты необходимо разместить в шелл-коде на месте адреса возврата, чтобы затереть его.

Подробнее рассмотрим процесс затирания адреса возврата в стеке в процессе работы программы. При вызове уязвимой функции strncpy происходит конкатенация двух строк – пустой строки в памяти программы и строки, что подается из конфигурационного файла. По причине того, что размер принимающей строки может быть слишком маленьким для того, чтобы «вместить» в себе присоединяемую строку, и это никак не контролируется со стороны программы, мы можем реализовать переполнение буфера. Для этого достаточно записать в строку большое количество мусорных данных и перезаписать адрес возврата. Место для локальных переменных, в том числе и для массивов выделяется в стеке, что и позволит с помощью переполнения поменять адрес возврата.

Ниже приведен процесс изменения стека во время отладки в процессе работы уязвимой функции.

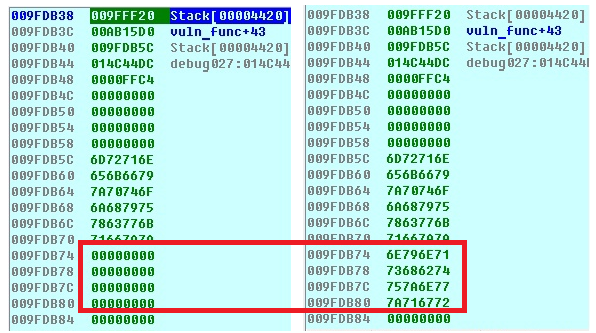


Рисунок 3 - Процесс изменения стека

Теперь достаточно рассчитать необходимое количество мусорных данных для подмена адреса возврата. В случае, если всё было рассчитано правильно, по возврата из уязвимой функции программа попадет на инструкцию jmp esp. Ниже приведен рисунок, на практике показывающий данный процесс.

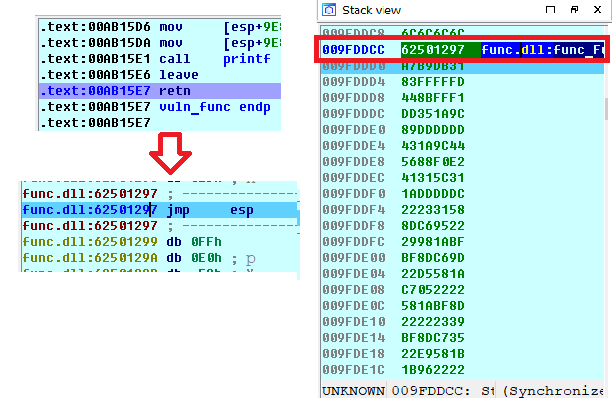


Рисунок 4 - Процесс перехода по новому адресу возврата

После перехода по данной инструкции программа передает управление на стек. Следующим этапом будет написание шелл-кода, который согласно условию работы должен осуществлять сканирование портов и выводит в файл список открытых в системе портов.

# **Реализация шелл-кода**

Для удобства реализации шелл-кода было принято решение писать его на языке Си вместо языка ассемблера. Одной из трудностей реализации шелл-кода является тот факт, что функция strncat() отслеживает нули при копировании. Это говорит о том, что в шелл-коде не должно встречаться нулей. В случае, если таковые попадутся, необходимо осуществить замену этих символов на другие. Чтобы упростить этот процесс, было принято решение применять операцию ксор на весь шелл-код с помощью ключа-символа, который не встречается в бинарном коде.

Ниже приведен ассемблерный код, который будет осуществлять ксор с шелл-кодом.

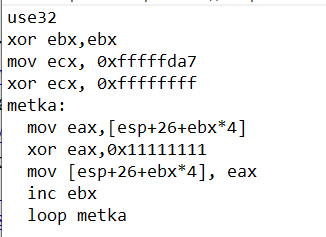


Рисунок 5 - Код для ксора шелл-кода

Для компиляции вышеупомянутого кода был использован FASM, на выходе которого мы получаем бинарный код, который мы можем вставить в конфигурационный файл для программы.

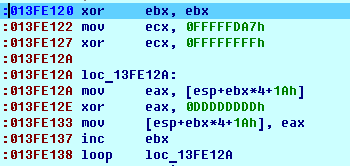


Рисунок 6 - Шелл-код при отладке уязвимой программы

Прежде чем приступить к написанию кода на Си, необходимо предварительно поменять некоторые настройки проекта:

1. Отключить подстановки:  
   Project Propeties -> C++\C -> Code generation  
   Busic Runtime Checks = Uninitialized variables (/RTCu);
2. Включить Enable Function-Level Linking, чтобы функции укладывались согласно их порядку в исходнике  
   Project Propeties -> C++\C -> Code generation  
   Enable Function-Level Linking = Yes (/Gy);

Выключить Incremental Linking, чтобы в промежутке между функциями не вставлялись инструкции JMP  
Project Propeties -> Linker -> General  
Enable Incremental Linking = No (/INCREMENTAL:NO).

При реализации программы важно придерживаться несколько правил: не использовать строки (использовать последовательности символов), а также вместо функций использовать указатели на них. Для использования внешних библиотек, а также функций будем использовать методы GetModuleHandleA и GetProcAddress, адреса которых можно найти в func.dll. Ниже приведен пример, отражающий используемые методы.

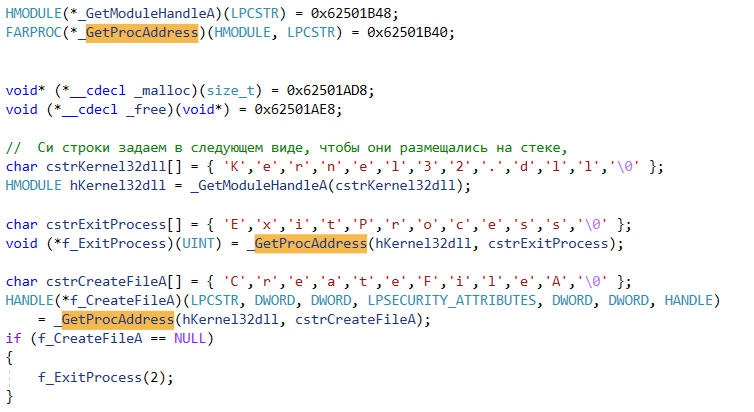


Рисунок 7 - Пример кода

При запуске программы данный код записывается в бинарный файл, который затем нужно «отксорить» и дописать его содержимое в конфигурационный файл, который идет на вход vuln.exe. По итогу всех манипуляций имеем следующую структуру конфигурационного файла:

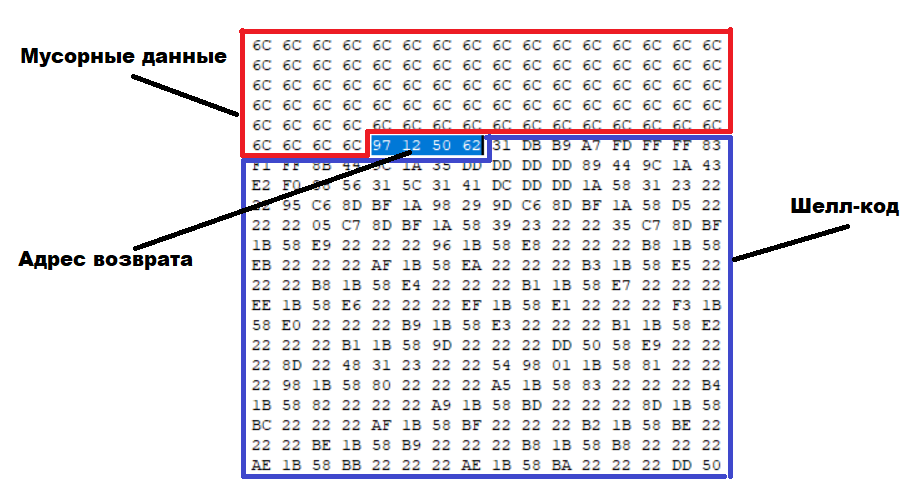


Рисунок 8 - Конфигурационный файл

# **Эксплуатация уязвимости**

По причине того, что при прослушивании портов с помощью функции connect из библиотеки ws2\_32 занимает неприлично много времени (около 20 секунд на порт) было принято решение использовать GetTCPTable. Ниже приведен результат вывода шелл-кода.

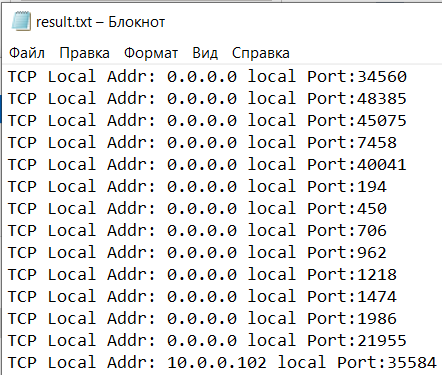


Рисунок 9 - Результат работы уязвимой программы

# **Выводы**

В ходе выполнения лабораторной работы были изучены различные механизмы, затрудняющие эксплуатацию уязвимостей (DEP, ASLR, GS, Security Cookies, SMEP, SAFESEH, SEHOP и другие). Также был реализован шелл-код, использующий уязвимость переполнения буфера в программе vuln.exe, соответствующей варианту программы. В процессе работы происходит сканирование открытых портов и вывод их в файл.