Министерство науки и высшего образования Российской Федерации

Санкт-Петербургский Политехнический Университет Петра Великого

—

Институт компьютерных наук и кибербезопасности

ЛАБОРАТОРНАЯ РАБОТА № 3

«Исследование подходов, применяемых при эксплуатации уязвимостей»

1. по дисциплине «Методы безопасности компьютерных систем»
2. Выполнил
3. студент гр. 5151004/10101 Веремейчук Я.Ю.

<подпись>

1. Проверил
2. преподаватель Овасапян Т.Д.

<подпись>

1. Санкт-Петербург
2. 2024
3. Цель работы

Изучение способов эксплуатации уязвимостей и механизмов противодействия.

1. Постановка задачи

При проведении исследования безопасности программного обеспечения одним из этапов работы является проверка найденных ошибок на возможность их эксплуатации. Для этого осуществляется написание, так называемых proof-of-concept эксплойтов (POC), демонстрирующих возможность эксплуатации найденной уязвимости.

Одной из наиболее распространенных ошибок является переполнение буфера. Оно может происходить как на стеке (stack overflow), так и на куче (heap overflow).

В рамках лабораторной работы необходимо осуществить эксплуатацию уязвимости, найденной в исполняемом файле в лабораторной работе №3. Необходимо реализовать полезную нагрузку в соответствии с указанным вариантом задания, которая будет выполняться в результате эксплуатации уязвимости.

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.
9. В отчете необходимо привести следующую информацию:

* перечень динамических библиотек, не использующих рандомизацию адресов;
* код, осуществляющий эксплуатацию уязвимости (exploit) с комментариями;
* схема эксплуатации уязвимости (описание стека при эксплуатации, описание приемов обхода защиты);
* формат файла, используемого для эксплуатации, с указанием значения полей, приводящих к исполнению кода;
* код полезной нагрузки в бинарном виде с комментариями;
* код полезной нагрузки на языке ассемблера / языке C с комментариями.

1. Теоретические исследования

Защитные механизмы, затрудняющие эксплуатацию уязвимостей:

**Data Execution Prevention (DEP)** – предотвращение выполнения данных, это набор программных и аппаратных технологий, позволяющих выполнять дополнительные проверки содержимого оперативной памяти и предотвращать запуск вредоносного кода. Функция DEP позволяет отразить целый класс атак. В частности, DEP позволяет блокировать вирусы и другие вредоносные программы, пытающихся выполнить свой код из областей системной памяти, резервированных для Windows и других авторизованных программ. Обнаружив, что программа использует системную память неправильно, средство DEP принудительно закрывает программу и выдает соответствующее сообщение.

**Address Space Layout Randomization (ASLR)** – рандомизация адресного пространства, это механизм обеспечения безопасности, включающий в себя рандомизацию виртуальных адресов памяти различных структур данных, чувствительных к атакам. Расположение в памяти целевой структуры сложно предугадать, поэтому шансы атакующего на успех малы.

**Security Cookies, GS –** глобальный cookie-файл безопасности используется для защиты от переполнения буфера в коде, скомпилированном с параметром */GS (проверка безопасности буфера),*и в коде, в котором используется структурная обработка исключений. При входе в функцию с защитой от переполнения cookie-файл помещается в стек, а при выходе значение в стеке сравнивается с глобальным cookie-файлом. Любое различие между ними указывает, что произошло переполнение буфера, что приводит к немедленному завершению работы программы.

**Supervisor Mode Execution Prevention (SMEP) –** предотвращение исполнения кода в режиме супервизора. Защита ОС строится на том, что пользовательские приложения не могут выполнять привилегированные операции, например, получить доступ к портам ввода-вывода, управляющим регистрам процессора и т.п. Кроме того, память, используемая в режиме ядра, защищена от доступа из пользовательского режима. Пользовательское приложение не может ни прочитать, ни изменить, ни выполнить код в памяти ядра напрямую. Взаимодействие с ядром ОС происходит опосредованно через интерфейс системных вызовов. Привилегированный режим в свою очередь не имеет никаких ограничений, если бы не SMEP. Если он включен, любая попытка выполнить код, находящийся в памяти пользовательского приложения, приведет к ошибке страницы (page fault). В частности, в обработчике ошибок страниц на Windows 8 данная ситуация вызовет bugcheck. Проще говоря, если какой-нибудь драйвер или системный модуль ядра попробует выполнить код, расположенный в памяти пользовательского приложения, закончится всё это синим экраном смерти.

**SAFESEH** – опция компоновщика, при указании которой он только создаст образ, если он может также создать таблицу изображения безопасных обработчиков исключений. Эта таблица указывает, какие обработчики исключений являются допустимыми для образа операционной системы. Если /SAFESEH не указан, компоновщик создает образ с таблицей обработчиков безопасных исключений, если все модули совместимы с функцией безопасной обработки исключений. Если все модули, не совместимый с функцией безопасной обработки исключений, полученный в результате образ не будет содержать таблицу безопасных обработчиков исключений.

**SEHOP –** Функция защиты от перезаписи структурированных исключений (SEHOP) предназначена для блокировки уязвимостей, использующих метод перезаписи структурированного обработчика исключений (SEH).

**Control Flow Guard (Guard CF, CFG)** — относительно новый механизм защиты Windows (exploit mitigation), нацеленный на то, чтобы усложнить процесс эксплуатации бинарных уязвимостей в пользовательских приложениях и приложениях режима ядра. Работа данного механизма заключается в валидации неявных вызовов (indirect calls), предотвращающей перехват потока исполнения злоумышленником (например, посредством перезаписи таблицы виртуальных функций). Мощная комбинация поддержки компиляции и времени выполнения от Control Flow Guard реализует целостность потока управления, которая жестко ограничивает места, где могут выполняться инструкции косвенного вызова. Он также определяет набор функций в приложении, которые могут быть потенциальными целями для косвенных вызовов. Таким образом, Control Flow Guard добавляет дополнительные проверки безопасности, которые могут обнаружить попытки взлома исходного кода.

Техники эксплуатации уязвимостей и способы обхода защитных механизмов:

**ret2libc –** техника эксплуатации, когда адрес возврата (RET) функции в стеке подменяется адресом иной функции в программе, и в последующую часть стека записываются параметры для вызываемой функции. Эта техника позволяет нападающему выполнить какую-либо существующую функцию без необходимости внедрения шеллкода в программу

**ROP (return-oriented programming)** – технология, которая позволяет обходить NX-бит. Идея ROP-цепочек довольно проста. Вместо того чтобы записывать и исполнять код на стеке, мы будем использовать так называемые гаджеты. Гаджет — это короткая последовательность команд, которые заканчиваются инструкцией ret. Комбинируя такие команды, мы можем добиться исполнения кода.

**Heap Spray** – метод для эксплуатации уязвимостей, дословно переводится как “распыление кучи”, потому что оно включает в себя запись серии байтов в различных местах кучи. Злоумышленник заставляет приложение выделить память под большое количество объектов, содержащих вредоносный код. При этом повышается вероятность успеха эксплойта, который переносит поток исполнения на некоторую позицию внутри кучи. Важно понимать, что без эксплойта, позволяющего изменять поток исполнения, heap spraying не нанесет какого-либо вреда.

**Stack Pivoting** – технология, являющаяся одним из примеров ROP-уязвимостей. Она заключается в подмене указателя стека на нужный злоумышленнику буфер для последующего эксплуатирования уязвимости.

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. Сегмент данных содержит данные, используемые командами программы. В сегменте стека хранятся переменные функций, переменные окружения п аргументы, которые передаются программе.

1. Ход работы

При выполнении лабораторной работы №2, которая заключалась в изучении принципов поиска уязвимостей в программном обеспечении без исходного кода. Для выданной программы была обнаружена уязвимость в виде функции strcpy, которая небезопасно передает буфер из конфигурационного файла.

Изображение выглядит как текст, снимок экрана, Шрифт, дисплей

Автоматически созданное описание

Рисунок 1 – Уязвимость программы

В этом случае, по причине отсутствия механизмов проверки, данные, содержащиеся в аргументе, переполняют буфер и затирают данные в стеке, которые следуют за буфером.

Изображение выглядит как стол

Автоматически созданное описание

Рисунок 2 – Структура стека на момент вызова функции *strcpy*

Изображение выглядит как стол

Автоматически созданное описание

Рисунок 3 – Структура стека после переполнения массива в функции *strcpy*

Данные из буфера перекрывают адрес возврата из функции. 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.

Итак, заданный исполняемый файл был открыт с помощью отладчика x64dbg. В отладочных символах была найдена функция vuln\_func, в которой и происходит запись из конфигурационного файла в программу.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Автоматически созданное описание

Рисунок 4 – Поиск функции, в которой происходит запись буфера в стек

После установки точки останова было найдено конкретное место в коде, где происходит запись в стек. Для наглядности в конфигурационный файл заранее были записаны символы ‘U’, имеющие код 55.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Рисунок 5 – Изменение конфигурационного файла

Изображение выглядит как текст, снимок экрана, число, программное обеспечение

Автоматически созданное описание

Рисунок 6 – размещение буфера в стеке

Можно заметить, что после выполнения команды call произошла запись в стек.

Также было отмечено, что место в буфере, которое не было наполнено символами из конфигурационного файла (если значение dst\_len больше, чем символов после слова start), заполняется символами ‘b’ с кодом 62. Соответственно, с помощью отладчика можно найти место, где находится адрес функции возврата, который можно затереть, чтобы проэксплуатировать уязвимость.

Изображение выглядит как текст, снимок экрана, дисплей, Шрифт

Автоматически созданное описание

Рисунок 7 – Место завершения записанного буфера

Так как изначально на буфер выделено 2508 байт, а при помощи функции memset значение 0x62 устанавливается на 2500 байт, в конфигурационном файле после метки /start необходимо иметь 2500 + 12 + 54 (длина заголовка с меткой /start). Соответственно, адрес возврата определяется битами под номерами 2569, 2568, 2567, 2566.

После установки адреса возврата на адрес jmp esp, так как уже была выполнена команда ret, esp указывает на инструкцию, непосредственно следующую за измененным адресом возврата, следовательно, шеллкод необходимо вставлять сразу за адресом возврата, то есть начиная с адреса 2567.

Затем был установлен отладчик Immunity Debugger и скрипт mona.py. Файл необходимо добавить в директорию PyCommands.

В результате выполнения команды !mona modules на экран выводятся используемые модули и их механизмы защиты.

Изображение выглядит как текст, снимок экрана, дисплей

Автоматически созданное описание

Рисунок 8 – Используемые модули

В результате были найдены 6 PE-файлов, с которыми компонуется программа: системные библиотеки KERNELBASE.dll, KERNEL32.dll, msvcrt.dll, ntdll.dll, apphelp.dll, реализованная для программы библиотека func.dll и сам исполняемый файл. Можно заметить, что func.dll не использует технологию рандомизации адресов (ASLR), значит, его можно использовать для внедрения shell-кода. С помощью скрипта mona.py и команды !mona jmp –r esp были найдены адреса нужных инструкций.

Изображение выглядит как текст, снимок экрана, дисплей, программное обеспечение

Автоматически созданное описание

Рисунок 9 – Найденные инструкции

В качестве перехода был использован первый адрес – 0x62501297.

Изображение выглядит как текст, снимок экрана, линия, Шрифт

Автоматически созданное описание

Рисунок 10 – Заполнение буфера 2564 байтами и запись адреса перехода

Изображение выглядит как текст, снимок экрана, число, программное обеспечение

Автоматически созданное описание

Рисунок 11 – Переполнение буфера и переход

При написании базонезависимого кода необходимо придерживаться правил:

1. Не должно быть явных импортов. Если необходимо вызвать какую-либо API-функцию, необходимо найти ее адрес, разбирая таблицу экспорта нужной библиотеки руками, предварительно подгрузив эту библиотеку с помощью LoadLibraryA (сама LoadLibraryA импортируется из библиотеки kernel32.dll, которая дефолтно грузится во все процессы).
2. Не должно быть глобальных переменных. Если в коде встречается глобальная переменная, компилятор помещает ее в секцию неинициализированных данных, и все обращения к ней осуществляются по абсолютному адресу, что в данном случае недопустимо.
3. Все строки должны формироваться в стеке, т.к. строки обычно помещаются в секцию инициализированных данных (то есть могут располагаться где-то в другом месте при выполнении программы). То есть вместо char str[] = "hello"; нужно писать char str[] = {‘h’,’e’,’l’,’l’,’o’,0}, такая строка будет динамически создаваться на стеке.

Общий алгоритм получения импортов, для шеллкода ():

1. Получить адрес PEB
2. Через PEB->Ldr->InMemoryOrderModuleList, найти:
   * + kernel32.dll (в большинство процессов загружен по умолчанию)
     + или ntdll.dll (если нужно использовать более низкоуровневую альтернативу)
3. Пройтись по таблице экспорта kernel32.dll (или ntdll), для поиска адресов:

* kernel32.LoadLibraryA (а по сути: ntdll.LdrLoadDLL)
* kernel32.GetProcAddress (а по сути: ntdll.LdrGetProcedureAddress)

1. Использовать LoadLibraryA (или LdrLoadDll) для загрузки необходимых DLL
2. Использовать GetProcAddress (или LdrGetProcedureAddress) для получения нужных функций

Но в ходе работы были рассмотрены функции (GetModuleHandleA, GetProcAddress, malloc, free), которые подгружает func.dll. Так как для данного файла отключена ASLR, адреса этих функций изменяться не будут. Остальные функции, необходимые для реализации полезной нагрузки шеллкода, будут подгружаться с помощью функций GetModuleHandleA, GetProcAddress.

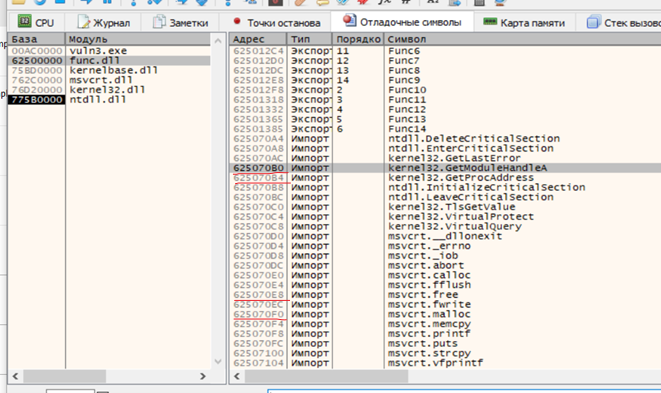


Рисунок 12 – Адреса необходимых функций

В соответствии с вариантом, полезной нагрузкой шеллкода является вывод списка открытых портов. Для начала была написана обычная программа, которая выводит такой список, а затем этот код был переписан в базонезависимый.

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

Рисунок 13 – Вывод списка портов

1. Вывод

Были изучены способы эксплуатации уязвимостей и механизмов противодействия.