Министерство науки и высшего образования Российской Федерации

Санкт-Петербургский Политехнический Университет Петра Великого

—

Институт компьютерных наук и кибербезопасности

ЛАБОРАТОРНАЯ РАБОТА № 2

«Изучение принципов поиска уязвимостей в программном обеспечении без исходных кодов»

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 типовых ошибок и принципов поиска уязвимостей в программном обеспечении без исходных кодов.

1. Постановка задачи

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);

— осуществлять запуск исследуемой программы;

— используя средство динамической бинарной инструментации (DBI) (Intel Pin / DynamoRIO) осуществлять измерение покрытия кода во время фаззинга;

— реализовать режим работы фаззера с обратной связью на основе покрытия кода, основанный на сохранении измененных байт в файле с учетом их влияния на покрытие кода программы;

— обнаруживать возникновение ошибки в исследуемом приложении;

— получать код ошибки и состояние стека и регистров на момент возникновения ошибки;

— логировать в файл информацию о произошедших ошибках и соответствующих им входных параметрах (произведенные замены).

5. 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, …).

6. Разработанный IDC/IDAPython-скрипт должен выводить следующую информацию:

— название найденной функции;

— адрес, откуда вызывается данная функция.

7. Используя разработанную программу для фаззинга формата файлов осуществить поиск в выданной программе уязвимости (целочисленное переполнение, отсутствие проверки длины данных), приводящей к переполнению буфера.

8. Сформировать файл, приводящий к краху исследуемого приложения.

9. Изучить с использованием дизассемблера и отладчика структуру программы и найти участок кода с уязвимостью. Провести исследование найденной уязвимости и условий ее эксплуатации.

10. В отчете необходимо привести следующую информацию:

— исходный код программы, осуществляющей фаззинг формата файла;

— описание методов, применяемых при фаззинге;

— участок кода с найденной уязвимостью (из дизассемблера), с комментариями;

— восстановленный исходный код (на языке С) участка с уязвимостью;

— описание найденной уязвимости (тип ошибки, причины, ограничение при эксплуатации);

— размер буфера, адрес начала буфера, адрес сохраненного на стеке адреса возврата (return address, за буфером);

— формат файла, используемого для эксплуатации, с указанием значения полей, приводящих к краху программы;

— описание уязвимости и механизмов ее эксплуатации;

— исходный код разработанных IDC/IDAPython-скриптов;

— описание возможного способа исправления уязвимости.

1. Теоретические исследования

Фаззинг — это техника тестирования программного обеспечения, часто автоматическая или полуавтоматическая, заключающаяся в передаче приложению на вход неправильных, неожиданных или случайных данных.

Предметом интереса являются падения и зависания, нарушения внутренней логики и проверок в коде приложения, утечки памяти, вызванные такими данными на входе. Фаззинг является разновидностью выборочного тестирования, часто используемого для проверки проблем безопасности в программном обеспечении и компьютерных системах.

Обычно фаззингом завершают процесс разработки, но фаззить можно и отдельные функции разрабатываемого продукта.

Преимущества фаззинга перед другими методами тестирования:

* фаззер можно запустить и забыть о нём до момента окончания тестирования, а работать уже с результатами;
* автоматизированное тестирование может выявить те ошибки, которые не удалось найти методом ручного тестирования, за счёт большего покрытия кода;
* позволяет собрать общее представление о защищенности тестируемого кода.

Существует несколько методов фаззинга: метод случайных данных, мутационного тестирования протокола вручную, мутационного тестирования и автоматически порождающего тестирования протокола.

Метод случайных данных наименее эффективен, однако может быть использован как самый быстрый способ поиска неверного кода. Подход с использованием случайных данных заключается в подстановке псевдослучайных данных в объект исследования. Самый простой пример такого подхода – бесконечный цикл, в котором случайные данные передаются на нужный адрес или порт.

Мутационное тестирование протокола вручную проще в использовании, чем метод случайных чисел. При ручном тестировании протокола автоматические фаззеры не применяются. Загрузив тестируемое приложение, тестер вручную вводит некорректные данные в попытке вызвать нежелательное поведение приложения. Чаще всего такой метод фаззинга применяется для веб-приложений.

Мутационное тестирование (тестирование методом грубой силы) представляет собой фаззер, который начинается с действующего образца протокола или формата данных и искажает каждые байт, слово, двойное слово в пакете данных или файле. Охват кода при подходе грубой силы зависит от того, сколько файлов тестируется. Большинство определений файлов довольно сложны, поэтому придется брать множество образцов, чтобы обеспечить сколько-нибудь приемлемый охват.

Автоматическое порождающее тестирование протокола – более продвинутый метод тестирования грубой силы. Для его реализации требуется предварительное исследование. Вместо того чтобы создавать образец для тестирования с жестко заданным кодом, создается грамматика, в которой описывается работа спецификации протокола. Таким образом определяются те порции пакета или файла, которые должны остаться неизменными, и те, которые служат переменными для фаззинга.

1. Ход работы

Для выполнения работы был изучен файл vuln3.exe. Для его запуска необходимо также скачать func.dll.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 1 – Выполнение исходной программы

Исполняемый файл подгружает библиотеку func.dll, считывает конфигурационный файл config\_3 и обрабатывает его. Для поиска уязвимости необходимо модифицировать файл config\_3.

Конфигурационный файл представляет из себя набор данных, содержащий заголовок – первые 48 байт, слово “/start” и буфер, который передается исходной программе. Подчеркнутые на рисунке 1 числа – числа, выводимые для значений shell\_len и dst\_len. Значение dst\_len отвечает за размер буфера, который принимает программа.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Рисунок 2 – Начало конфигурационного файла

При изучении исходной программы в IDA Pro была найдена функция strcpy, которая небезопасно передает буфер из конфигурационного файла, вызывая переполнение буфера.

Изображение выглядит как текст, снимок экрана, Шрифт, дисплей

Автоматически созданное описание

Рисунок 3 – Уязвимость в исходной программе

В ходе лабораторной работы был разработан фаззер.

Для начала было изучено средство динамической бинарной инструментации DynamoRIO (версии 9.0.1).

DynamoRIO - это виртуальная машина процесса, которая перенаправляет выполнение программы с ее исходного двоичного кода на копию этого кода. Инструментарий, выполняющий действия нужного инструмента, затем добавляется в эту копию. В исходную программу не вносится никаких изменений, ее не нужно каким-либо образом специально подготавливать. DynamoRIO работает полностью во время выполнения и обрабатывает устаревший код, динамически загружаемые библиотеки, динамически генерируемый код и самоизменяющийся код.

DynamoRIO отслеживает весь поток управления, чтобы зафиксировать выполнение целевой программы в целом.

Инструмент DynamoRIO drcov собирает информацию о том, какие базовые блоки были выполнены. Он записывает результаты в отдельный файл журнала для каждого процесса.

Параметры среды выполнения для этого инструмента включают:

-dump\_text: выводит файл журнала в текстовом формате.

-dump\_binary: включено по умолчанию, выводит файл журнала в двоичном формате.

-[no\_]nudge\_kills: только для Windows. По умолчанию включено. Использует nudge для уведомления процесса о завершении, чтобы было вызвано событие exit.

-logdir dir: задает каталог журнала, который по умолчанию равен ".".

Для измерения покрытия кода необходимо выполнить команду:

drrun.exe -t drcov -dump\_text -- vuln3.exe

В полученном файле утилита пишет, какие модули использовала программа и присваивает каждому id. В данном случае это исполняемый файл, библиотека func.dll, системные библиотеки и модули DynamoRIO.

Изображение выглядит как текст, снимок экрана, число, линия

Автоматически созданное описание

Рисунок 4 – Результат работы улилиты DynamoRIO

Ниже в файле для каждого модуля выводится относительный адрес базового блока и количество выполненных инструкций (байт). Модуль с идентификатором 0 – исходная программа.

Если перейти по одному из адресов в IDA Pro (предварительно выставив в Edit -> Segments -> Rebase Program value 0), можно попасть на блок данных с указанным количеством байт. Например, был найден блок, для которого выполнилось 14 байт. Для поиска необходимо нажать G.

Изображение выглядит как текст, программное обеспечение, Значок на компьютере, веб-страница

Автоматически созданное описание

Рисунок 5 – Поиск выполненного блока

Получается, если просуммировать все выполненные байты, получится покрытие кода (будет подсчитываться в фаззере). То есть количество кода, которое удалось выполнить. При изменении конфигурационного файла покрытие может изменяться. Чем оно больше, тем лучше, так как выше вероятность того, что при передаче программе некорректных данных мы найдем наибольшее количество уязвимостей.

Итак, фаззер. Пользователь может выбрать исходные директории.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 6 – Выбор директорий

Для начала выбирается случайный конфигурационный файл из заданной директории с помощью функции FileChoice(), и оттуда считываются данные.

Далее в функции Mutation() данные подвергаются одному из изменений: подстановка граничного значения для переменных программы, создание случайного количества байт и добавление в конец имеющегося буфера, подстановка рандомного значения для переменных программы, стирание данных из конца файла.

Затем происходит запуск исходной программы с измененным конфигурационным файлом с помощью дебагера, который выполняет все инструкции, возвращает исключения программы, регистры. Если тот или иной конфигурационный файл привел к краху программы, то он записывается, а также записывается состояние регистров.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Рисунок 7 – Логи программы после падения, код исключения

Затем рассчитывается покрытие кода и записывается файл, данные которого привели к такому состоянию.

Во время работы программа печатает время работы, количество запусков, падений и максимальное покрытие.

Изображение выглядит как текст, снимок экрана, меню

Автоматически созданное описание

Рисунок 8 – Информация, выводимая в процессе работы

Также при выполнении лабораторной работы был написан скрипт на Python для поиска уязвимых функций в исходном коде.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Рисунок 9 – Поиск небезопасных функций в коде

1. Вывод

В ходе работы были изучены типовые ошибки и принципы поиска уязвимостей в программном обеспечении без исходных кодов. Был реализована своя программа фазер для автоматического поиска уязвимостей, с помощью которого была протестирована тестируемая программа