**Практика 6. Обратная разработка приложений. Бинарные уязвимости.**

**Целью** данной работы является получение практических навыков эксплуатации простейших бинарных уязвимостей.

Отчёт по данной работе должен содержать подробное описание уязвимости и её наглядную демонстрацию (скриншот с эксплуатацией).

**Краткая теория (охватывает не все задания)!**

**1. Общие понятия о бинарных уязвимостях.**

Бинарные уязвимости являются одним из самых сложных классов уязвимостей в современном мире практической ИБ связанной с анализом кода. Данные уязвимости сложно обнаружить и зачастую очень сложно поэксплуатировать полностью. Первым усложняющим задачу фактором является анализ двоичного кода, а не исходного кода. При анализе двоичного кода даже большой командой профессионалов всё равно велик шанс упустить важные детали из виду.

Хорошим примером может служить ОС Windows которую исследуют и разрабатывают большое количество профессионалов и несмотря на это уязвимостей в ней находят очень часто, в том числе и критические. Сложно переоценить значимость данных уязвимостей, ведь бинарные уязвимости также и находят в критически важных инфраструктурах, например, в ПЛК АСУ ТП, которые обеспечивают целые города благами жизни, или регулируют ядерное оружие. Всё высоконагруженное и высокотехничное оборудование работает на ПО разработанном на компилируемых языка и соответственно сразу потенциально подвержено такого рода уязвимостям.

Существует два больших класса уязвимостей: кода и архитектуры. Уязвимости кода непосредственно связаны с неправильными обработками данных или недостаточной проверкой получаемых значений. Уязвимости архитектуры представляют собой реализацию каких-либо возможностей системы вне задуманных целей для это возможности. Другими словами, вы используете те функции, которые вам доступны, но не в тех целях, для которых они подразумевались, при этом проверки целей не происходит концептуально.

**2. Переполнение буфера.**

Данная уязвимость считается самой популярной при исследованиях, несмотря на то, что разработано уже множество методик, позволяющих компилятору защищаться от подобного рода уязвимостей, они всё равно обнаруживаются и эксплуатируются, особенно в старом ПО, которое не редко входит в состав нового ПО как некие полезные модули.

Суть данной уязвимости заключается в том, что размер входного буфера данных зачастую не проверяется при перемещение его из одного места в другое или просто при считывании. При этом данная уязвимость не обязательно должна быть связана с переполнением стека или кучи с возможностью выполнения кода, также можно и перезаписывать некоторые локальные переменные при возникновении переполнений буфера. Рассмотрим на примере.

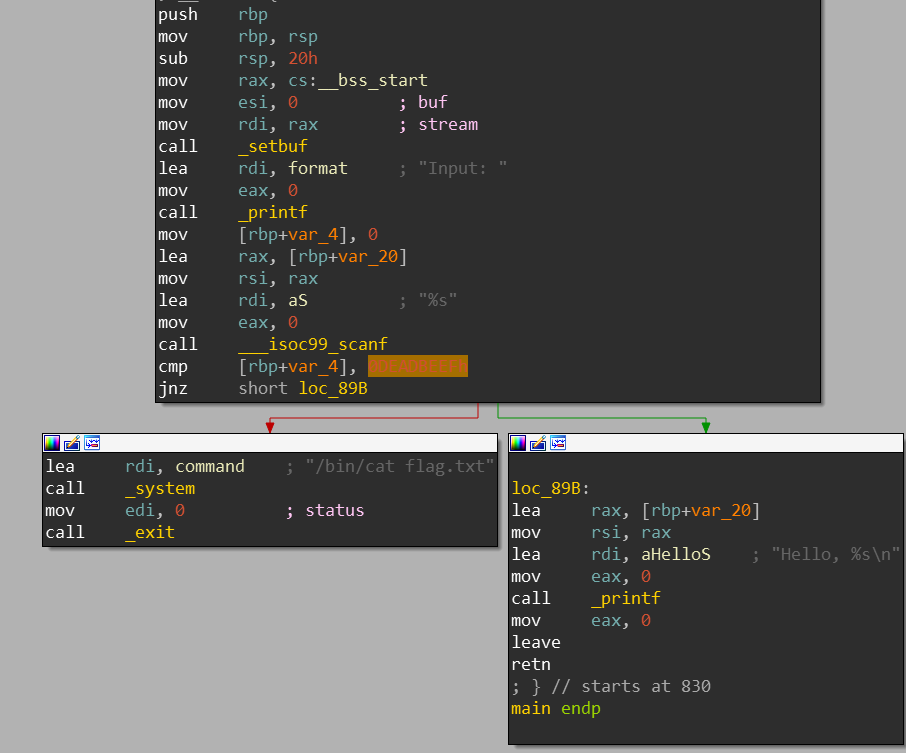


Рисунок 1 - Пример уязвимости переполнения буфера.

На Рис. 1 представлен пример переполнения буфера, при этом в данном случае при переполнение будет произведена перезапись локальной переменной, но если попытаться перезаписывать значения дальше вплоть до адреса возврата, то произойдёт ошибка, связанная с нарушением целостности стека. Эта ошибка связана с наличием защищающих механизмов в компиляторе. Одним из таких механизмов является так называемая стековая канарейка, которая находится перед адресом возврата и соответственно нельзя перезаписать адрес возврата не перезаписав её, а её перезапись влечёт за собой аварийное завершение работы программы, так как обработчик обнаружит нарушение целостности стека. Это довольно простая технология очень хорошо влияет на защиту ПО, если программист вдруг допустил ошибку.

Перезапись переменной произойдёт потому, что в функцию scanf() не был явно передан аргумент отвечающий за количество считываемых символов, однако массив в который данные символы будут считываться ограничен 16 байтами, которые располагаются на стеке. Там-же и располагается локальная переменная и если правильно высчитать смещение и передать необходимое количество байт для достижения адреса данной переменной, то последующие байты перезапишут значение данной переменной. В данном примере это показано довольно искусственно, т.к. по условиям видно, что для того чтобы выполнилась команда чтения флага одна из локальных переменных должна быть равна значению **0xdeadbeef**, однако данная переменная нигде не изменяет своё значение из чего можно сделать вывод что от нас требуется изменить это значение при помощи переполнения буфера.

Для того, чтобы высчитать верное количество, байт которое нужно передать, чтобы попасть на перезапись необходимой переменной можно воспользоваться IDA и посмотреть разбиение стека для данной функции. Данную операцию можно выполнить с помощью горячих клавиш “Ctrl+K”, после их нажатия откроется новая вкладка, в которой будет указано разбиение стека для данной функции. Как можно заметить по Рис. 2, от начала нашего буфера, который хранит введённые значения до адреса необходимой переменной 0x20-0x4 = 0x16 – байт, в переводе на десятичную систему 32-4=28 байт. То есть после передачи 28 байт следующие байты будут перезаписывать данную переменную.

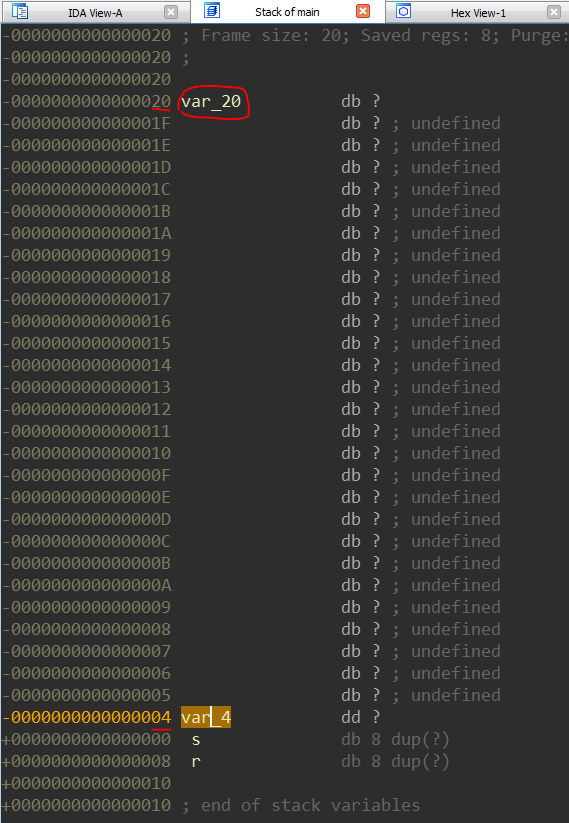


Рисунок 2 - Разбиение стека функции main.

Данный вывод довольно просто проверяется при помощи запуска исполняемого файла с передачей в него необходимых параметров Рис. 3.

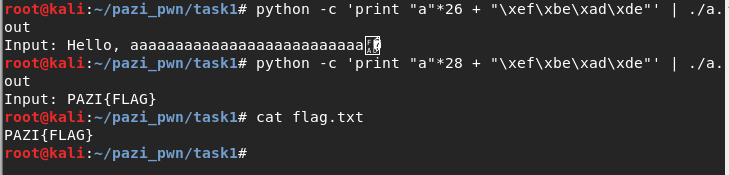


Рисунок 3 - Эксплуатация уязвимости.

Данный пример довольно искусственен, однако хорошо демонстрирует базовый принцип уязвимости переполнения буфера.

**3. Уязвимость форматной строки.**

Суть данной уязвимости заключается в том, что введённые пользователем данные могут передаваться в функции вывода на экран, при этом формат данных не будет указан явно, что позволяет пользователю выводит те данные на экран которые ему необходимы, например, содержание стека функции. В контексте атаки данная уязвимость может быть полезна, например, для получения как раз той самой стековой канарейки, про которую было написано выше, при её получении мы можем перезаписать её верным значением и после перезаписать необходимым нам значением адрес возврата.

Рассмотрим пример. На Рис. 4 видно, что введённые пользователем данные в программ считываются и добавляются в буфер, далее данный буфере предаётся в функцию printf() однако, формат передаваемых данных никак не указывается, что позволяет манипулировать выводом данной функции в различных целях. Целью данного примера является возможность чтения контрольного значения PAZI{flag} которое можно заметить в функции. Если рассматривать данный тип заданий, встречающихся на различных CTF соревнованиях, то они обычно характеризуются тем, что исполняемый файл никак не предоставляется, а предоставляется только сетевой сервис, на котором запущен данный файл, то есть самого факта обратной разработки нет, это фактически слепые догадки. Однако и на практике можно с пользой применять данную уязвимость, например, если она была найдена во встраиваемой системе под которую нет отладчика, с помощью данной уязвимости можно следить за значениями на стеке.

Для выполнения цели предусмотренной данным искусственным примером достаточно произвести определённый форматный ввод и немного обработать вывод. Ввод представлен на Рис. 5, после получения объёма шестнадцатеричных цифр можно выделить в них ASCII-коды формата флага и найти флаг.



Рисунок 4 - Пример уязвимости форматной строки.

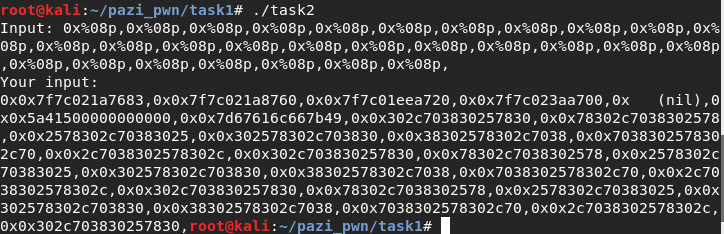


Рисунок 5 - Пример эксплуатации уязвимости форматной строки.