**Лекция 5 Исследование безопасности приложений. Основы.**

В данном курсе под исследованием безопасности приложений подразумевается обратная разработка и поиск уязвимостей в исполняемых файлах для различных операционных систем. По-другому обратную разработку часто называют ревёрс инжинирингом. Данное понятие включает в себя различные сферы деятельности, однако нас будет интересовать только сфера, связанная с исследованием исполняемых файлов. Существуют целые компании, занимающиеся антивирусной деятельностью, например, Eset, Kaspersky, Dr. Web, Avast и другие.



В основу их деятельности положена обратная разработка исполняемых файлов, детектирование на основе результатов обратной разработки зловредных файлов и занесение их в антивирусные базы для дальнейшего обнаружения этих зловредных файлов на других компьютерах.

Антивирусная деятельность не является основополагающим направлением использования обратной разработки, хоть и занимает довольно большую отдельную нишу. Основной деятельностью обратной разработки всегда было и будет являться обнаружение уязвимостей.

Для начального представления об обратной разработке рассмотрим следующий пример. Вы получаете квадратную коробку, с одной стороны у коробки есть ручка, регулируя которую, вы выставляете различные значения на вход, после фиксации ручки вы можете нажать на копку и получить значение на другой стороне коробки, которая считается выходом. То есть по сути у вас есть чёрный ящик, которому вы можете предоставлять входные данные и получать выходные.



Теперь представим, что вам дали только этот ящик и никаких документов к нему нет, то есть, как он работает, вы не знаете, вы просто можете подавать значения на вход и получать значения на выходе, как они связаны вы не понимаете, но хотите разобраться. Зачем? Ну, например, для того, чтобы сделать такой же ящик или чтобы найти в ящике ошибку – момент, когда он будет выдавать одинаковые результаты на разные входные данные, хотя так не должно быть, или что-то другое. Ваши цели могут быть совсем иные, важно лишь то, что вам нужно понять, как он работает или как работают его отдельные части.

Чтобы решить эту проблему, вы можете использовать разные варианты, например, вы можете проанализировать связь входных и выходных данных и найти формулу, по которым они меняются, но здесь вас может подстерегать проблема: вдруг внутри машины есть заранее определённые константы, о которых вы не знаете, да и для подтверждения вашей формулы вам придётся проверить все пары входных и выходных данных, а их может быть очень много.

Хорошим решением данной проблемы будет разбор ящика и анализ его внутреннего устройства. Для разбора ящика необходимы инструменты, а они могут быть различны в зависимости от задач, однако есть основные мультизадачные инструменты. Например, с помощью отвёртки можно раскрутить каркас ящика, крепления внутренних элементов, полностью разобрать его и достать необходимые элементы наружу, однако мы всё равно не получим представление о работе ящика, если просто всё вытащим из него. Нам необходимо проанализировать взаимосвязь всех компонентов при их совместной работе, в идеале понять через какие элементы ящика проходят наши входные значения, как преобразуются и как обрабатываются на различных этапах, в общем, полностью понять алгоритмы обработки наших значений. Лучшим вариантом, конечно, будет, пошаговое наблюдение за работой ящика, однако это не всегда возможно как в ящиках, так и в реальных анализах программ.

Считается, что обратная разработка является самой сложной частью информационной безопасности, наверное, так и есть, ведь, чтобы действительно проводить полноценные исследования, нужно знать очень много фундаментальных вещей, начиная от построения процессора и заканчивая алгоритмами и их реализациями, поэтому данная тематика считается очень сложной и оплачивается соответственно.



Перед тем как мы перейдём непосредственного к исследованию нашей первой программы, мы вспомним основы информатики, работы ЦП, устройства ОС и исполняемых файлов, вспомним про низкоуровневые языки программирования (далее ЯП) и многое другое. Всё это необходимо для того, чтобы у вас было хотя бы малейшее представление и понимание того, что вы должны будете сделать при исследовании приложения.

**Начнём с простого.**

Повторим, что такое ЦП (центральный процессор) и для чего он нужен.

Итак, ЦП служит для выполнения команд компьютером, он по сути выполняет ряд заложенных в него на стадии производства команд, современные процессоры могут выполнить за раз огромное количество операций. Все операции процессора очень просты, например, сложить два регистра (переменные) или выдать значение на шину данных, положить в регистр значение, взять значение с шины данных и другие простейшие операции. Были упомянуты следующие, возможно, незнакомые вам определения: регистр, шина данных, они будут объяснены далее. Нужно помнить, что фактически процессор воспринимает только последовательности 0 и 1, то есть он работает на очень низком уровне и оперирует только бинарным кодом. В зависимости от архитектуры длины команд могут быть различны, могут иметь фиксированную длину (так называемая RISC-архитектура), либо иметь нефиксированную длину команды (например, x86, x86-64). Человеку сложно воспринимать просто числа и считать их командами, для него нужна более высокая абстракция, которой как раз и является язык Ассемблера. Язык Ассемблера различен для различных архитектур процессоров. Язык ассемблера напрямую связан с двоичными кодами команд процессора, то есть каждая команда на ассемблере имеет однозначное прямое соответствие с опкодом процессора (опкод – это операционный код, довольно частое применяется для описания одной команды процессора). Посмотрим на примере. У нас есть команда на языке ассемблера для архитектуры x86-64:

mov eax, 1 – что означает положи в регистр eax значение 1

 В шестнадцатеричном представлении опкодов эта команда будет выглядеть так:

B8 01 00 00 00

Видно, что ассемблерное представление удобнее для человека и позволяет лучше понять, что происходит в данном участке кода.

На изображении слева вы можете увидеть шестнадцатеричное представление опкодов, а справа представление на языке ассемблера.



**Кратко рассмотрим строение процессора.**

Процессор помимо устройства управления и АЛУ содержит также запоминающее устройство, которое включает в себя регистры – небольшие ячейки сверх быстрой оперативной памяти (СОЗУ), которые используются самим процессором для выполнения каких-либо операций, в зависимости от архитектуры размер ячеек, их количество и общепринятые названия меняются. Также существуют регистры общего назначения и специальные регистры. Фактически можно сделать довольно сильное упрощение и просто представлять себе регистры, как переменные без типа, с фиксированным размером, доступ к которым осуществляется мгновенно.



Для базового представления о регистрах, надеюсь данной информации хватит, однако рекомендую поподробнее ознакомиться с данной темой (ссылки: https://en.wikipedia.org/wiki/Processor\_register, https://en.wikipedia.org/wiki/Central\_processing\_unit).

Итак, теперь мы примерно понимаем, как работает процессор, и у нас может возникнуть вопрос, как процессор отличает данные от команд, ведь по сути вся информация в компьютере является числами, которые так или иначе можно передать процессору в качестве команд. На самом деле эту проблему решает ОС, регулируя файлы через их формат. На рисунке ниже представлено краткое описание формата PE для OC Windows.



Например, если вы откроете в шестнадцатеричном редакторе две разные картинки с одинаковыми расширениями, то увидите, что в заголовках этих картинок будут находиться одинаковые байты, за исключением конечно байтов, связанных с расширением. ОС понимает по расширению файла, что этот файл не является исполняемым, однако расширение можно принудительно поменять, например, на \*.exe (расширение исполняемых файлов в Windows), тогда ОС будет думать, что это файл исполняемый, однако при попытке его запуска возникнет ошибка из-за того, что формат файла другой, ОС будет искать определённые секции в файле, необходимые заголовки и не найдёт их, так как мы просто поменяли расширение у картинки, а не полностью формат файла. Таким образом и регулируются команды, которые поступают к процессору.

Рассмотрим тему с форматом исполняемых файлов подробнее.

**Формат исполняемого файла.**

Разные ОС имеют разный формат исполняемых файлов: для Windows – это формат PE, для Linux – это формат ELF, для OS X – это формат Mach-O. Что это означает на практике? Если мы имеем одинаковый исходный код программы на 3-х разных ОС, её скомпилированный вид будет различным. 

То есть скомпилированная программа из одного исходного кода на одном и том же процессоре, но под разные ОС, будет различна в каждом случае. Запустить программу, скомпилированную под Linux на Windows, не представляется возможным, без использования специальных средств. Всё это происходит из-за того, что каждая операционная система имеет свой формат исполняемого файла. Формат – это набор определённых правил построения исполняемого файла, например, заголовок должен содержать определённую сигнатуру на такой-то позиции, потом идёт размер секций, потом – сами секции.

Файл строится по неким единым правилам для того, чтобы ОС при запуске файла смогла верно загрузить его в ОЗУ и передавать ЦП верные инструкции, которые находятся в этом файле, также ОС будет предоставлять исполняемому файлу необходимые библиотеки, данные и прочее. Всё это ОС поймёт из содержания файла, так как он построен по правилам формата исполняемого файла для данной ОС. Советую также самостоятельно ознакомиться с форматами исполняемых файлов, например, с форматом ELF (<https://ru.wikipedia.org/wiki/Executable_and_Linkable_Format>).

Теперь перейдём к тому, как получить исполняемый файл. Этот процесс называется компиляцией, и вы должны уже быть с ним знакомы с курса программирования на С++.

**Компиляция.**

Процесс преобразования исходного кода на каком-либо компилируемом ЯП высокого или среднего уровня в машинные коды называется компиляцией. По сути это превращение нашего файла с исходным кодом, например, на языке С, в исполняемый файл под Windows, который можно будет запустить и увидеть, как он исполнит заданные нами команды в исходном коде. Теория компиляции очень сложна, в некоторых ВУЗах на это отводят целые дисциплины в несколько семестров, однако нам нужно знать лишь некоторые фундаментальные моменты компиляции, но если вы хотите углубиться в эту сферу, то учтите, что вам придётся очень хорошо выучить математику.

Что нам интересно для наших целей: Во-первых, компиляция — это однонаправленный процесс, который невозможно полностью достоверно обратить, то есть по имеющимся машинным командам мы не сможем получить код на языке С или С++, так как процесс компиляции — это процесс с потерями. Во-вторых, если вы хотите внести какие-либо изменения в программу и править исходный код, то для внесения этих изменений в скомпилированную программу, необходимо перекомпилировать её. В-третьих, в скомпилированной программе отсутствуют такие понятия, как переменные, функции, циклы и прочее, скомпилированная программа это набор машинных инструкций. Пожалуй, это основные моменты, которые нам необходимы. Относительно полная схема компиляции изображена на рисунке ниже.



Может возникнуть вопрос: “Если невозможно получить исходные коды из скомпилированного приложения, то как мы будем его исследовать?”. Данный вопрос подводит нас к процессу преобразования скомпилированного приложения в человеко-читаемый вид. Этот процесс называется дизассемблирование.

**Дизассемблирование.**

Процесс преобразования (трансляции) машинных кодов в текст программы на языке ассемблера называется дизассемблированием. Если вы пролистаете лекцию чуть выше, то заметите, что, когда мы описывали машинные коды, мы учли, что каждому машинному коду соответствует одна команда на языке ассемблера, соответственно мы можем перевести машинный код в код ассемблера, который поудобнее анализировать, чем последовательности нулей и единиц. Данная задача решается сегодня лишь несколькими инструментами: IDA Pro и radare2 (только под Linux). IDA Pro являет несомненным лидером в данной области, однако она является платной, а бесплатные версии обладают крайне урезанным функционалом, но мы всё равно будем использовать её в качестве основного инструмента для исследования, потому что она обладает большим спектром функций, помимо основной функции – дизассемблирования.



На изображение показан интерфейс дизассемблера IDA Pro.

На этом моменте лекция по основам заканчивается, но, для лучшего понимания функционирования всех механизмов, советуем почитать по каждой из рассмотренных в данной лекции тематик дополнительную информацию, это будет полезно.