**МИНОБРНАУКИ РОССИИ**

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«Гжельский государственный университет»** (ГГУ)

Колледж ГГУ

Специальность 09.02.07. Информационные системы и программирование.

**Реферат**

по предмету « МАПО»

на тему «Дизассемблирование».

ВЫПОЛНИЛ:

Студент группы ИСП-О-17

Белкин И.В.

ПРОВЕРИЛА:

Прокуронова А.Ю.

Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

п. Электроизолятор

2019 г.

Что такое дизассемблирование. Дизассемблирование – преобразование программы на машинном языке к ее ассемблерному представлению. Декомпиляция – получение кода языка высокого уровня из программы на машинном языке или ассемблере.

Декомпиляция – достаточно сложный процесс. Это обусловлено следующими причинами:

− Процесс компиляции происходит с потерями. В машинном языке нет имен переменных и функций, и тип данных может быть определен только по производимым над ними операциям. Наблюдая пересылку 32-х бит данных, требуется значительная работа, чтобы определить, являются ли эти данные целым числом, дробью или указателем.

− Компиляция это операция типа множество-множество.Компиляция и декомпиляция могут быть выполнены множеством способов. Поэтому результат декомпиляции может значительно отличаться от исходного кода.

− Декомпиляторы в значительной степени зависимы от конкретного языка и библиотек. Обрабатывая исполняемый файл, созданный компилятором Delphi, декомпилятором, разработанным для C, можно получить фантастический результат.

− Необходимо точное дизассемблирование исполняемого файла. Любая ошибка или упущение на фазе дизассемблирования практически наверняка размножатся в результирующем коде.

Прогресс средств декомпиляции происходит медленно, но верно. Наиболее сложный на сегодняшний день декомпилятор IDA, будет рассмотрен ниже.

Зачем нужно дизассемблирование. Цель инструментов дизассемблирования заключается в содействии исследованию функционирования программ, когда их исходные коды не доступны. Наиболее распространенные цели дизассемблирования:

− анализ вредоносного программного обеспечения;

− анализ уязвимостей программного обеспечения с закрытым исходным кодом;

− анализ совместимости программного обеспечения с закрытым исходным кодом;

− валидация компилятора;

− отображение команд программы в процессе отладки.

Анализ вредоносного программного обеспечения. Разработчики вредоносного программного обеспечения вряд ли предоставят исходный код своего детища. А без доступа к исходному коду, будет довольно сложно определить поведение вредоносной программы. Существуют два основных метода исследования вредоносных программ - динамический и статический анализ. Динамический анализ заключается в исполнении программы в тщательно контролируемом окружении (песочнице) и записи каждого ее действия. Статический анализ основан на разборе исходного кода, который, в случае вредоносной программы, в основном состоит из дизассемблированных листингов.

Анализ уязвимостей. Для простоты, можно разделить процесс аудита безопасности на три стадии: поиск уязвимостей, анализ уязвимостей, и разработка эксплойта. Одни и те же шаги предпринимаются вне зависимости от того, имеется ли у вас исходный код; однако, уровень трудоемкости резко возрастает, когда в вашем распоряжении есть лишь исполняемый файл. Первый шаг - исследование потенциально уязвимых условий в программе. Это зачастую достигается использованием динамических техник, таких как фаззинг, однако также может быть реализовано (обычно со значительно большими усилиями) посредством статического анализа. Как только проблема обнаружена, требуется определить, является ли она уязвимостью и если да, то при каких условиях.

Листинг дизассемблирования помогает понять, каким образом компилятор расположил переменные в памяти. Например, может быть полезно узнать, что объявленный программистом 70-байтный массив символов при распределении памяти компилятором был округлен в сторону 80 байт. Листинги дизассемблирования также предоставляют единственный способ понять, объявлены ли переменные глобально или внутри функций. Понимание реального расположения переменных в памяти жизненно важно при разработке эксплойтов.

Анализ совместимости. Когда программы доступны только в виде исполняемых файлов, сторонним разработчикам крайне сложно обеспечить совместимость с ними своих программ, а также расширить их функциональность. Например, если производитель не предоставил драйвер для аппаратного устройства, то реверс инжиниринг – практически единственное средство для разработки альтернативных драйверов.

Валидация компилятора. Дизассемблирование может быть средством для проверки соответствия работы компилятора его спецификации. Также исследователя может заинтересовать наличие дополнительных возможностей, оптимизирующих результат компиляции. С точки зрения безопасности важно быть уверенным, что код, генерируемый компилятором, не содержит черных ходов.

Отладка. К сожалению, дизассемблеры, встроенные в отладчики, зачастую малоэффективны (OllyDbg ─ исключение). Они неспособны к серийному дизассемблированию и иногда отказываются дизассемблировать, не будучи в состоянии определить границы функции. Поэтому, для лучшего контроля над процессом отладки, лучше использовать отладчик в сочетании с хорошим дизассемблером.

Как дизассемблировать. Типичные задачи, с которыми сталкивается дизассемблер: взять 100 КБ из исходного файла, отделить код от данных, преобразовать код к языку ассемблера, и главное ничего не потерять. В этот список можно добавить дополнительные пожелания, например, определение границ функций, распознавание таблиц переходов, выделение локальных переменных... Это значительно усложнит его работу. Качество результирующих листингов дизассемблирования определяется свойствами алгоритмов, а также уместностью их применения в конкретной ситуации.

Базовый алгоритм дизассемблирования

Шаг 1. Первым шагом в процессе дизассемблирования является идентификация кодового сегмента. Так как команды обычно смешаны с данными, то дизассемблеру необходимо их разграничить.

Шаг 2. Получив адрес первой команды, необходимо прочитать значение, содержащееся по этому адресу (или смещению в файле) и выполнить табличное преобразование двоичного кода операции в соответствую ему мнемонику языка ассемблера.

Шаг 3. Как только команда была обнаружена и декодирована, ее ассемблерный эквивалент может быть добавлен к результирующему листингу. После этого необходимо выбрать одну из разновидностей синтаксиса языка ассемблера.

Шаг 4. Далее необходимо перейти к следующей команде и повторить предыдущие шаги до тех пор, пока каждая команда файла не будет дизассемблирована.

Алгоритм линейной развертки

Алгоритм линейной развертки использует крайне прямолинейный подход при выборе очередной команды для дизассемблирования: где завершается код одной команды - начинается код новой. В результате, наиболее сложной задачей становится определение первой команды. Дизассемблирование начинается с первого байта в сегменте кода и последовательно продвигается, обрабатывая команды одну за другой, пока не будет достигнут конец сегмента. При этом не производится попыток понять логику передач управления в программе посредством распознавания команд перехода таких, например, как условия.

Главное преимущество алгоритма линейной развертки состоит в полном покрытии кодового сегмента. Одним из основных недостатков является невозможность распознать данные, если они совмещены с кодом. Это очевидно из листинга, демонстрирующего результат дизассемблирования функции при помощи данного алгоритма. Эта функция содержит конструкцию switсh. Компилятор принял решение реализовать switch как таблицу переходов. Более того, компилятор предпочел разместить таблицу переходов внутри самой функции. Конструкция jump по адресу 401250 ссылается на таблицу адресов начиная с 410257. К сожалению, дизассемблер рассматривает ее как набор команд и неверно генерирует соответствующее представление на языке ассемблера.

40123f: 55              pushebp

401240: 8bec            movebp,esp

401242: 33 c0           xor eax,eax

401244: 8b 55 08        mov edx,DWORD PTR [ebp+8]

401247: 83 fa 0c        cmp edx,0xc

40124a: 0f 87 90 00 00 00 ja 0x4012e0

401250: ff 24 95 57 12 40 00 jmp DWORD PTR [edx\*4+0x401257]

401257: e0 12           loopne 0x40126b

401259: 40              inc eax

40125a: 00 8b 12 40 00 90 add BYTE PTR [ebx-0x6fffbfee],cl

401260: 12 40 00        adc al,BYTE PTR [eax]

401263: 95              xchg ebp,eax

401264: 12 40 00        adc al,BYTE PTR [eax]

401267: 9a 12 40 00 a2 12 40 call 0x4012:0xa2004012

40126e: 00 aa 12 40 00 b2 add BYTE PTR [edx-0x4dffbfee],ch

401274: 12 40 00        adc al,BYTE PTR [eax]

401277: ba 12 40 00 c2 mov edx,0xc2004012

40127c: 12 40 00        adc al,BYTE PTR [eax]

40127f: ca 12 40        lret 0x4012

401282: 00 d2           add dl,dl

401284: 12 40 00        adc al,BYTE PTR [eax]

401287: da 12           ficom DWORD PTR [edx]

401289: 40              inc eax

40128a: 00 8b 45 0c eb 50 add BYTE PTR [ebx+0x50eb0c45],cl

401290: 8b 45 10        mov eax,DWORD PTR [ebp+16]

401293: eb 4b           jmp 0x4012e0

Алгоритм рекурсивного спуска

Алгоритм рекурсивного спуска использует концепцию передачи управления, определяющую, должна ли команда быть дизассемблирована по наличию или отсутствию на нее ссылок от других команд. Для понимания алгоритма рекурсивного спуска, полезно классифицировать команды в зависимости от их влияния на счетчик команд CPU.

Команды, не влияющие на счетчик команд. После выполнения такой команды управление переходит непосредственно к следующей команде. Примерами такой команды может послужить арифметические, такие как add; Для подобных команд процесс дизассемблирования такой же как и при линейной развертке.

Команды условного перехода. Команды условного перехода, такие как x86 jnz, образуют две возможных ветви исполнения. Поскольку в статическом контексте обычно невозможно определить исход проверки условия, алгоритм рекурсивного спуска дизассемблирует обе ветви. Адрес целевой ветви добавляется в список адресов для последующего дизассемблирования. Дизассемблирование продолжается последовательно, так, как если бы условие было ложно.

Команды безусловного перехода. Безусловные переходы приводят к нарушению последовательного порядка исполнения команд. Команда, получающая управление после выполнения безусловного перехода, может располагаться от него на значительном расстоянии. Кроме того, как видно из приведенного выше листинга, команды, следующие непосредственно за командой безусловного перехода не исполняются вообще. Таким образом, необходимость в их дизассемблировании отпадает.

Алгоритм рекурсивного спуска пытается определить адрес назначения безусловного перехода и занести его в список адресов для последующего дизассемблирования. К сожалению, не все безусловные переходы могут быть корректно обработаны данным алгоритмом. Когда адрес назначения перехода зависит от параметра, получаемого в процессе исполнения, его определение методами статического анализа становится невозможным.

Команды вызова функции. Команды вызова функции работают сходным образом с командами безусловных переходов (включая невозможность определить адрес назначения команды, такой как call eax), за исключением того, что после выполнения функции, управление обычно возвращается команде, следующей за ее вызовом. При этом, как и в случае с условными переходами, образуются две ветви исполнения. Адрес назначения команды call добавляется в список адресов для последующего дизассемблирования, в то время как команда, следующая за call, дизассемблируется с использованием алгоритма линейной развертки.

Алгоритм рекурсивного спуска может оказаться неэффективным в случае, если при возвращении из вызываемой функции поведение программы отклоняется от ожидаемого. Например, в коде функции может преднамеренно модифицироваться адрес возврата.

Команды возврата. В некоторых случаях, алгоритм рекурсивного спуска терпит неудачу. Команда возврата из функции (например, ret x86) не предоставляет информации о том, какая команда будет выполнена далее. Если бы программа была на самом деле запущена, управление было бы передано по адресу, расположенному на вершине стека. У дизассемблера нет возможности доступа к стеку. Вместо этого дизассемблирование внезапно останавливается. В этом случае алгоритм рекурсивного спуска обращается к списку отложенных адресов, и процесс дизассемблирования возобновляется. Этот рекурсивный процесс отражает смысл названия алгоритма.

Алгоритм рекурсивного спуска превосходно разделяет код и данные. Главным же недостатком такого подхода является неспособность распознавать ветви, образуемые таким командами как jump и call, использующими для адресации таблицы поиска. Несмотря на это, в сочетании с эвристиками для распознавания указателей на код, алгоритм рекурсивного спуска способен обеспечить хорошее покрытие кода в сочетании с прекрасным разделением кода и данных.