УДК 004.056
ББК 32.973
Б64

Бирюков А. А.
Б64 Реверсивный инжиниринг приложений под Windows. – М.: ДМК Пресс, 2024. – 376 с.: ил.

ISBN 978-5-93700-292-1

Реверсивный инжиниринг считается одним из наиболее сложных направлений в информационной безопасности (ИБ). В книге автор предлагает приоткрыть завесу тайны над этой темой и с помощью практических примеров рассмотреть, как работают приложения под ОС Windows, а также разобраться в том, как эксплуатировать уязвимости переполнения буфера, размещать свой код в выполнимых файлах, находить полезную информацию в дампах памяти и многое другое.

Книга предназначена как для начинающих специалистов, желающих разобраться в реверс-инжиниринге, так и для опытных специалистов по ИБ, интересующихся данной темой.

УДК 004.056
ББК 32.973

ISBN 978-5-93700-292-1

ОГЛАВЛЕНИЕ

0x0CF11 ВСТУПЛЕНИЕ

«Ален ноби, ностра алис!
Что означает: если один человек построил,
другой завсегда разобрать может!»
Х/ф «Формула любви»

О пользе реверсинга

Реверсивный инжиниринг, или обратная разработка, появился задолго до создания компьютеров. Техническим специалистам всегда было интересно, как работает то или иное устройство или агрегат. А государства и промышленники хотели заполучить готовые технологии без затрат на разработку собственных аналогов или покупку лицензий.

Традиционно двигателем технологий являются военные конфликты, и здесь обратная разработка пришлась как нельзя кстати, ведь, захватив трофейную

технику, инженеры сразу же начинали изучать принципы ее работы, для того чтобы, с одной стороны, выявить слабые места вражеского вооружения, а с другой – понять, какие решения у противника лучше своих, дабы затем применить эти решения уже на своей технике. Кстати, не стоит думать, что реверсивный инжиниринг – это «тупое» копирование чужого готового решения.

Казалось бы, просто разобрал готовое устройство, измерил детали, перенес на чертежи – и готово, можешь делать свое такое же. Однако на практике это далеко не так. Возьмем, к примеру, реверсинг самолета. Прежде всего отметим, что такое сложное устройство состоит из тысяч деталей, собираемых вместе в определенной последовательности, поэтому инженерам сначала придется немало потрудиться, чтобы его разобрать и понять принципы взаимодействия различных узлов. А затем мы можем столкнуться с неожиданными проблемами. Так, при изучении американских самолетов и другой техники, поставлявшейся в СССР по ленд-лизу, инженерам пришлось переводить все размеры деталей из дюймов в метрическую систему. А так как это соотношение является иррациональным числом, то задача «копирования» становится совсем нетривиальной.

Таким образом, обратная разработка является одним из инструментов технологического развития экономики и государства в целом.

Зачем нужен реверсинг

Но вернемся к компьютерам. Возможно, у многих читателей возник вопрос, почему книга посвящена реверсингу приложений под Windows, хотя всем известно, что в свете последних геополитических событий в России взят курс на замещение иностранного ПО российскими аналогами. Однако во многих отраслях на сегодняшний день есть проприетарные приложения, работающие только под Windows. Так, например, на многих объектах критической инфраструктуры, прежде всего на промышленных предприятиях, для управления технологическим оборудованием используются ОС Windows и проприетарные SCADA-системы. Для того чтобы заменить данные решения на отечественные разработки, необходимо понимать, как функционирует та или иная SCADA-система, как работают протоколы обмена данными и драйверы оборудования.

Таким образом, реверсинг может быть востребован при решении задач по замене проприетарного программного обеспечения на собственные разработки.

Кроме того, противостояние может вестись в том числе и в виртуальном пространстве, и противники могут использовать различное вредоносное программное обеспечение для достижения своих целей. Соответственно, реверсинг может помочь как тем, кто защищается от вражеских атак (например, вирусным аналитикам и специалистам по тестированию на проникновение), так и тем, кто выполняет противоположные задачи.

Код практически любого приложения может содержать ошибки. Самый простой способ исправить эти ошибки – это найти в исходном коде проблемную команду или функцию и внести соответствующие исправления в код. Но что делать, когда по тем или иным причинам исходный код нам не доступен, как в случае с проприетарным ПО? Если у нас имеется только откомпилированный выполнимый файл – артефакт. Например, для выявления уязвимостей, недокументированных возможностей и т. д. В таком случае нам тоже потребуется

обратная разработка. Результатом обратной разработки является построение детального алгоритма работы программы, а также выявление уязвимостей и других интересующих исследователя аспектов работы программного обеспечения.

Таким образом, обратная разработка нужна тем, кто занимается поиском уязвимостей с целью улучшения защищенности программного обеспечения, – багхантерам.

Также реверсивный инжиниринг требуется программистам, работающим с низкоуровневыми языками программирования (например, разработчикам драйверов), где без знания ассемблера не обойтись.

Не стоит забывать, что вредоносный код – это тоже программное обеспечение, и его тоже необходимо реверсить для выявления принципов заражения и для того, чтобы понять, как от них можно защититься. Этими задачами занимаются вирусные аналитики.

Ну что ж, надеюсь, мне удалось наглядно обосновать необходимость реверсивного инжиниринга для решения задач ИТ и ИБ, и теперь мы можем перейти непосредственно к описанию разделов этой книги.

О чем эта книга

Реверсинг приложений традиционно считается одним из самых сложных направлений в информационных технологиях. Многих пугает необходимость изучать язык ассемблера, необходимый для работы с дизассемблерами и отладчиками. Также программистам, работавшим только с языками высокого уровня, не всегда понятны принципы работы со стеком, обратная запись байтов в памяти и манипуляции с регистрами.

Опыт автора по созданию и сопровождению курса по реверсивному инжинирингу в Академии Кодебай и проведению занятий в других учебных центрах по данной тематике лег в основу этой книги.

В своей книге я постарался как можно больше внимания уделить практической части процесса обратной разработки, поэтому здесь большое количество иллюстраций, в которых я постарался зафиксировать все выполняемые на практике действия.

В начале книги мы уделим внимание основам языка ассемблера, поговорим об основных инструкциях, регистрах и организации памяти. Но потом сразу же начнем решать Crack Me (крякмиксы, задачки «взломай меня»). Это маленькие приложения, в которых необходимо ввести определенный пароль или выполнить иное действие, для того чтобы получить сообщение об успешном решении. Решение крякмиксов позволяет быстрее понять основные принципы реверсинга.

Далее мы поговорим о переполнении буфера – одной из самых распространенных ошибок программистов. Мы откомпилируем уязвимое приложение на C и затем напишем к нему эксплойт. Эта тема будет наиболее интересна специалистам по анализу защищенности приложений.

Потом обсудим инъекцию своего кода в выполнимые файлы. В частности, рассмотрим формат PE, следы, которые оставляют компиляторы в файлах, понятие code cave и многое другое. Эта тема будет полезна прежде всего вирусным аналитикам и специалистам по анализу защищенности.

Следующая глава будет посвящена тоже инъекциям, но уже процессов. Материал будет полезен вирусным аналитикам.

Затем речь пойдет об антиотладке, то есть о тех средствах, к которым могут прибегнуть разработчики, для того чтобы защитить приложение от отладки. Эта глава будет интересна всем, кто занимается реверсингом и разработкой, так как программистам нужно знать, как можно защитить приложение, а аналитикам нужно знать, как можно обнаружить защиту.

В следующей главе речь пойдет о том, где может спрятаться приложение, если ему необходимо закрепиться в системе и запускаться при каждой загрузке ОС. Этот материал будет полезен как вирусным аналитикам, так и разработчикам средств защиты. Очень часто агенты, которые запущены на пользовательских узлах, должны быть как можно менее заметны для пользователя, например при использовании систем предотвращения утечек класса DLP или «следилок» наподобие Стахановца.

После этого мы поговорим о реверсинге приложений, написанных на .NET. Здесь речь пойдет о самом фреймворке Microsoft .NET, языке CIL и инструментах, необходимых для реверсинга. Также в этой главе будет разобрано несколько крякмиксов.

Затем коснемся еще одного средства защиты от реверсинга – обфускации кода и упаковки. Этот материал также будет полезен разным специалистам.

Ну а в следующих двух главах мы подробно рассмотрим анализ нескольких вредоносных файлов, а также методику построения ROP-цепочек. Эта глава будет интересна прежде всего вирусным аналитикам и специалистам, занимающимся разработкой средств защиты.

Далее речь пойдет о работе со средствами эмуляции типа «песочница». Здесь потребуется определенный навык работы с ОС Linux, так как именно под этой ОС мы будем ставить нашу песочницу.

Затем мы обсудим работу с инструментами расследования компьютерных инцидентов и, в частности, со средствами анализа дампа памяти. Эта глава будет интересна как вирусным аналитикам, так и специалистам по компьютерным инцидентам.

И в завершение мы разберем еще один инструмент, который может существенно помочь специалистам по анализу вредоносов, – специализированный дистрибутив Remnux.

Заключение к вступлению

При написании книги использовались некоторые мои статьи из блога УЦ Отус на Хабре, также в списке использованных источников вы можете найти полезные книги и статьи. Для тех, кто хотел бы лучше изучить язык ассемблера, я отдельно рекомендую книгу В. Ю. Пирогова «Ассемблер для Windows». Примеры на MASM взяты как раз из этой книги.

Ну и прежде чем перейти к самой книге, я предлагаю разгадать шифр, который загадал наш белый хакер – кот Реверс, встречающий нас в начале каждой главы книги. В заголовке каждой главы указан номер в десятичном виде и какое-то шестнадцатеричное число. Читателю предлагается разгадать алгоритм, по которому номер главы сопоставляется данному числу. Сразу скажу, что здесь нет серьезной криптографии уровня AES/ГОСТ, но присутствует логика крякмикса среднего уровня. Ответ я приведу в заключительной части этой книги.

ГЛАВА 1
0x00f13 О коде, инструментах и крякмиксах

Основы реверсинга

В этой главе мы обсудим основы ассемблера и затем плавно перейдем к теме реверсинга. Конечно, эта книга не ставит перед собой цель обучения языку ассемблера, для этого уже издано большое количество книг, некоторые из которых представлены в библиографии к данной книге. Также можно воспользоваться различными ресурсами в сети Интернет.

Здесь мы приведем основные понятия и инструкции ассемблера. Полный список инструкций ассемблера процессора Intel можно найти в приложениях к книге. Начнем с основ.

Регистры и стек

Обработку данных в процессоре осуществляют специальные ячейки, известные как регистры. Для простоты понимания регистры можно сравнить с пере-

менными в языках высокого уровня. В них можно хранить значения и выполнять определенные операции, но количество этих регистров ограничено архитектурой процессора.

Регистры в процессоре x86-64 можно разделить на четыре категории: регистры общего назначения, специальные регистры для приложений, сегментные регистры и специальные регистры режима ядра. В рамках выполняемых задач нас будут интересовать прежде всего регистры общего назначения (general-purpose registers), которые в основном и используются в приложениях на ассемблере.

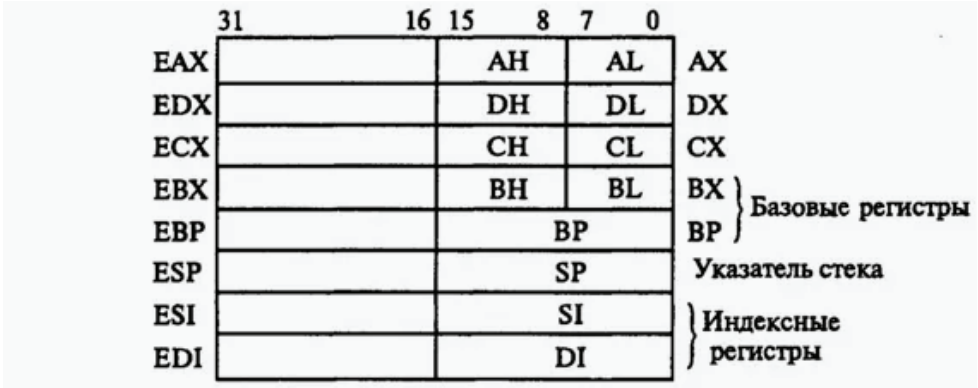
Далее в книге мы будем говорить преимущественно о 32-битной архитектуре процессора, так как все рассматриваемые крякмиксы, вредоносы и другие примеры кода написаны именно под нее.

Но в этой главе будут также упомянуты различные особенности, свойственные для 64-битной архитектуры.

Начнем с регистров для 32 бит. Процессор архитектуры x86 имеет восемь 32-битных регистров общего назначения, регистр флагов и указатель инструкций. Регистры общего назначения:

- EAX (Accumulator): для арифметических операций;
- ECX (Counter): для хранения счетчика цикла;
- EDX (Data): для арифметических операций и операций ввода-вывода;
- EBX (Base): указатель на данные;
- ESP (Stack pointer): указатель на верхушку стека;
- EBP (Base pointer): указатель на базу стека внутри функции;
- ESI (Source index): указатель на источник при операциях с массивом;
- EDI (Destination index): указатель на место назначения в операциях с массивами;
- EIP: указатель адреса следующей инструкции для выполнения;
- EFLAGS: регистр флагов, содержащий биты состояния процессора.

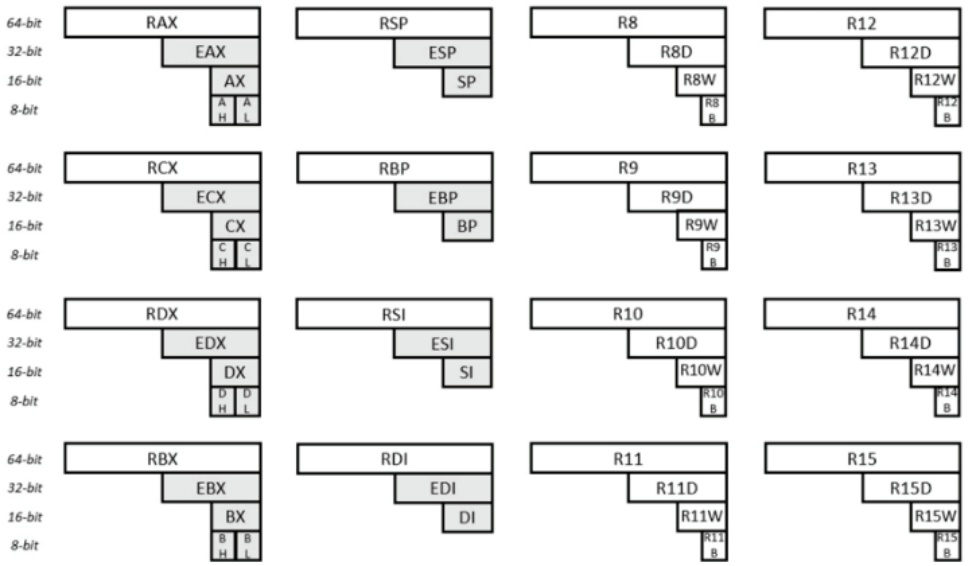
Можно получить доступ к частям 32-битных регистров с меньшей разрядностью. Например, младшие 16 бит 32-битного регистра EAX обозначаются как AX. К регистру AX можно обращаться как к отдельным байтам, используя имена AH (старший байт) и AL (младший байт). Но не для всех регистров можно получить такой доступ. На иллюстрации ниже представлены регистры общего назначения и наличие возможности доступа к их младшим частям.



Регистры общего назначения, базовые и индексные регистры

Как видно, младшие части доступны только у EAX, EDX, ECX, EBX. При этом если сами регистры (например, EAX) имеют разрядность 32 бита, то младшая часть AX будет содержать 16 бит. И мы можем обращаться к младшей части (AL) и старшей части (AH) регистра AX. К старшей части регистра EAX (биты 16–31) мы не можем обратиться напрямую, для этого необходимо прибегнуть к дополнительным действиям, например операциям побитового сдвига.

Теперь про 64 бита. В архитектуре x64 эти регистры были расширены до 64 бит, а новые расширенные регистры получили имена, которые начинаются с буквы R, например RAX, RBX и т. д. Кроме того, были добавлены 8 новых 64-битных регистров R8–R15.



Для обращения к 32-, 16- и 8-битным частям 64-разрядных регистров используются стандартные для архитектуры x86 имена регистров. Для доступа к подрегистрам новых 64-битных регистров R8–R15 применяется соответствующий суффикс:

- **D**: для получения младших 32 бит регистра, например R11D;
- **W**: для получения младших 16 бит регистра, например R11W;
- **B**: для получения младших 8 бит регистра, например R11B.

Также существуют еще шесть сегментных регистров:

- сегмент стека (SS). Указатель на стек (S означает «стек»);
- сегмент кода (CS). Указатель на код (C означает «код»)»
- сегмент данных (DS). Указатель на данные (D означает «данные»);
- дополнительный сегмент (ES). Указатель на дополнительные данные (E означает «дополнительный»; E следует после D);
- сегмент F (FS). Указатель на дополнительные данные (F следует после E);
- сегмент G (GS). Указатель на еще больше дополнительных данных (G следует за F).

Большинство приложений в современных операционных системах (таких как FreeBSD, Linux или Microsoft Windows) содержат модель памяти, которая указывает значения почти всех сегментных регистров в одно и то же место в памяти (и вместо них использует файл-подкачку), эффективно отключая их применение. Обычно применение FS или GS является исключением из этого правила, вместо этого они используются для указания на данные, относящиеся к конкретному потоку. В частности, с FS мы еще будем неоднократно сталкиваться при подселении своего кода в чужие процессы.

Флаги

EFLAGS – это 32-разрядный регистр, используемый в качестве набора битов, представляющих логические значения, для хранения результатов операций и состояния процессора.

Ниже представлены названия битов-флагов из этого 32-разрядного регистра.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

Биты с именами 0 и 1 являются зарезервированными битами и не должны изменяться.

Эти флаги используются по-разному.

0. CF: флаг переноса. Устанавливается, если последняя арифметическая операция перенесла (сложение) или заимствовала (вычитание) немного больше размера регистра. Затем это проверяется, когда за операцией следует операция добавления с переносом или вычитания с заимствованием для обработки значений, слишком больших для хранения только в одном регистре.

2. PF: флаг четности. Устанавливается, если количество установленных битов в младшем значащем байте кратно 2.

4. AF: флаг настройки. Перенос арифметических операций с десятичными числами в двоичном коде (BCD).

6. ZF: нулевой флаг. Устанавливается, если результат операции равен нулю (0).

7. SF: флаг знака. Устанавливается, если результат операции отрицательный.

8. TF: флаг ловушки. Устанавливается, если выполняется пошаговая отладка.

9. IF: флаг прерывания. Устанавливается, если прерывания включены.

10. DF: флаг направления. Направление потока. Если установлено, строковые операции будут уменьшать свой указатель, а не увеличивать его, считывая память в обратном направлении.

11. OF: флаг переполнения. Устанавливается, если арифметические операции со знаком приводят к значению, слишком большому для хранения в регистре.

12–13. IOPL: поле уровня привилегий ввода-вывода (2 бита). Уровень привилегий ввода-вывода текущего процесса.

14. NT: флаг вложенной задачи. Управляет цепочкой прерываний. Устанавливается, если текущий процесс связан со следующим процессом.

16. RF: флаг возобновления. Ответ на исключения отладки.

17. VM: режим виртуальной машины 8086. Устанавливается в режиме совместимости с 8086.

18. AC: проверка выравнивания. Устанавливается, если выполняется проверка выравнивания ссылок на память.

19. VIF: флаг виртуального прерывания. Виртуальный образ IF.

20. VIP: флаг ожидания виртуального прерывания. Устанавливается, если прерывание находится в ожидании.

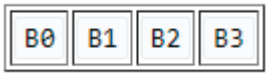
21. ID: флаг идентификации. Поддержка инструкции CPUID, если она может быть установлена.

Еще один регистр, с которым мы будем работать при исследовании переполнения буфера, – это EIP, указатель инструкции.

Регистр EIP содержит адрес следующей инструкции, которая будет выполнена, если в программе не происходит ветвления. EIP может быть прочитан через стек только после команды call. Подробнее о том, что такое стек, мы поговорим чуть позже.

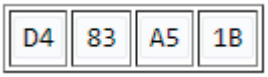
Архитектура x86 является little-endian, что означает, что многобайтовые значения записываются сначала младшим значащим байтом. (Важно понимать, что это относится только к порядку байтов, а не к битам.)

Таким образом, 32-разрядное значение 0xB3B2B1B0 на x86 было бы представлено в памяти как



Представление в порядке убывания порядка строк

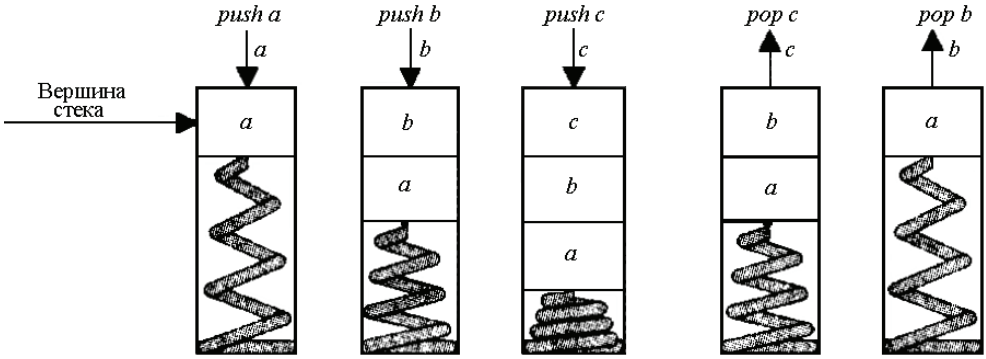
Например, 32-разрядное двойное слово 0x1BA583D4 (0x обозначает шестнадцатеричное число) было бы записано в память как



Пример с младшим порядком строк

Это будет отображаться как 0xD4 0x83 0xA5 0x1B при выполнении дампа памяти.

Рассмотрим еще одну важную тему – работу со стеком. Стек – это структура данных, работающая по принципу: первым вошел – последним вышел (First In Last Out). Мы можем использовать стек для хранения значений регистров. Для того чтобы поместить значение регистра в стек, используется инструкция PUSH, а для извлечения – инструкция POP.



Инструкции

Продолжая тему инструкций, по сути команд, используемых в ассемблере, рассмотрим несколько основных. Работу с регистром с помощью PUSH и POP мы уже разобрали. Пожалуй, самой распространенной инструкцией является MOV. Она копирует данные из одного места в другое и имеет следующий синтаксис:

```
MOV destination, source
```

Инструкция принимает два операнда. Первый операнд destination представляет расположение, куда надо поместить данные. В качестве такого места может выступать регистр процессора или адрес в памяти. Второй операнд source указывает на источник данных, в качестве которого может выступать регистр процессора, адрес в памяти или непосредственный операнд – число. То есть инструкция mov копирует данные из source в destination. При этом оба операнда не могут быть одновременно адресами в памяти.

Вот несколько примеров:

```
MOV EAX, 0x0a
```

Загружаем в EAX значение 0x0a.

```
MOV EBX, EAX
```

Загружаем в EBX значение EAX.

```
MOV AH, AL
```

В этом примере мы работаем с 16-битными частями EAX, загружая в AH значение AL.

```
MOV [BX], AX
```

А здесь мы загружаем в адрес [BX] значение регистра AX.

Но конструкции вида

```
MOV [BX], [AX]
```

недопустимы, так как оба операнда не могут быть одновременно адресами в памяти.

Инструкции для работы со стеком PUSH и POP, так же как и инструкция MOV, могут работать с различными сущностями. Так, мы можем поместить в стек содержимое всего регистра:

```
PUSH ECX
```

первые 16 бит:

```
PUSH DX
```

или целочисленное значение:

```
PUSH 0xfa
```

При этом, поместив в стек значение одной сущности, мы можем затем извлечь из стека это значение в другую сущность. Так, в предыдущих примерах мы последней поместили в стек константу 0xfa, но извлечь ее мы можем в AL с помощью инструкции

```
POP AL
```

Сохраненное значение DX мы извлечем в BX:

```
POP BX
```

И наконец, значение EAX мы извлечем в ячейку памяти, расположенную по адресу 0x87654321:

```
POP [0x87654321]
```

Еще одна полезная инструкция при решении крякмиксов – это CMP. Она используется для сравнения двух операндов. То есть эта команда сравнивает два числа, проверяя их равенство.

```
CMP ЗНАЧЕНИЕ1, ЗНАЧЕНИЕ2
```

При этом ЗНАЧЕНИЕ1 может быть одним из следующих: область памяти, регистр общего назначения. А ЗНАЧЕНИЕ2 может быть областью памяти, также регистром общего назначения или непосредственным значением (например, числом).

И команды JMP и CALL. Команда JMP выполняет безусловный переход в указанное место.

```
JMP METKA
```

При этом METKA – это адрес перехода, которому передается выполнение кода.

Команда CALL выполняет похожие действия – вызывает процедуру. Синтаксис:

```
CALL ИМЯ
```

Но здесь в поле ИМЯ может быть имя процедуры, метка, переменная, регистр или непосредственное значение адреса.

После выполнения вызова с помощью CALL при получении команды RET будет выполнен возврат к инструкции, которая следовала после CALL.

```
…
CALL Label1
MOV AX,0x0d
…
Label1:
      …
      RET
```

В приведенном примере в соответствии с инструкцией CALL будет выполнен переход по метке Label1, далее по команде RET мы вернемся к инструкции MOV AX,0x0d.

Режимы адресации

На языке ассемблера x86 режимы адресации определяют, как операнды памяти указываются в инструкциях. Режимы адресации позволяют программисту получать доступ к данным из памяти или эффективно выполнять операции с операндами. Архитектура x86 поддерживает различные режимы адресации, каждый из которых предлагает разные способы обращения к памяти или регистрам. Вот несколько распространенных режимов адресации в x86:

Адресация регистра
(адрес операнда R находится в поле адреса)

```
mov ax, bx ; перемещает содержимое регистра bx в ax
```

Моментальный
(фактическое значение указано в поле)

```
mov ax, 1 ; перемещает значение 1 в регистр ax
```

или

```
mov ax, 010Ch ; перемещает значение 0x010C в регистр ax
```

Прямая адресация памяти
(адрес операнда указан в поле адреса)

```
.data
my_var dw 0abcdh ; my_var = 0xabcd
.code
mov ax, [my_var] ; скопируем содержимое my_var в ax (ax=0xabcd)
```

Адресация с прямым смещением
(использует арифметику для изменения адреса)

```
byte_table db 12, 15, 16, 22 ; байты
mov al, [byte_table + 2]
mov al, byte_table[2] ; аналогично предыдущей инструкции
```

Регистр косвенный
(поле указывает на регистр, содержащий адрес операнда)

```
mov ax, [di]
```

Представленные здесь операции с памятью мы впоследствии увидим при разборе реальных примеров, представленных далее в этой книге.

Теперь поговорим о режимах работы процессора.

Реальный режим

Реальный режим – это пережиток оригинального процессора Intel 8086, появившегося много десятилетий назад. Как правило, вам не нужно ничего знать о нем (если только вы не программируете для системы на базе DOS или, что более вероятно, не пишете загрузчик, который напрямую вызывается BIOS).

В книге мы не рассматриваем работу с загрузчиками, но на самом деле проникновение вредоносов в загрузочный сектор – тоже довольно важная тема, и те, кто планирует далее заниматься вирусной аналитикой, вполне могут с этим столкнуться.

Intel 8086 обращался к памяти, используя 20-разрядные адреса. Но поскольку сам процессор был 16-разрядным, Intel изобрела схему адресации, которая обеспечивала способ преобразования 20-разрядного адресного пространства в 16-разрядные слова. Современные процессоры x86 запускаются в так называемом реальном режиме, который представляет собой режим работы, имитирующий поведение 8086, с некоторыми очень незначительными отличиями, для обеспечения обратной совместимости.

В реальном режиме сегмент и регистр смещения используются вместе для получения конечного адреса памяти. Значение в регистре сегмента умножается на 16 (сдвигается на 4 бита влево), и к результату добавляется смещение. Это обеспечивает доступное адресное пространство в 1 МБ. Однако особенность схемы адресации позволяет получить доступ сверх лимита в 1 МБ, если используется адрес сегмента 0xFFFF (максимально возможный); на моделях 8086 и 8088 все обращения к этой области ограничены нижним пределом памяти, но на моделях 80286 и более поздних версиях таким образом можно адресовать до 65 520 байт, превышающих отметку в 1 МБ, если включена адресная строка A20.

Защищенный режим

Если вы программируете в современной 32-разрядной операционной системе (такой как Linux, Windows), вы, по сути, программируете в плоском 32-разрядном режиме. При адресации может использоваться любой регистр, и, как правило, более эффективно использовать полный 32-разрядный регистр вместо 16-разрядной части регистра. Кроме того, сегментные регистры обычно не используются в плоском режиме, и их использование в плоском режиме не считается наилучшей практикой.

Многосегментная модель памяти

Используя 32-разрядный регистр для адресации памяти, программа может получить доступ (почти) ко всей памяти современного компьютера. Для бо-

лее ранних процессоров (только с 16-разрядными регистрами) применялась модель сегментированной памяти. Регистры CS, DS и ES используются для указания на разные сегменты памяти. Для небольшой программы (маленькой модели) CS=DS=ES. Для моделей памяти большего размера эти «сегменты» могут указывать на разные местоположения.

Длительный режим (Long Mode)

Термин «длительный режим» относится к 64-разрядному режиму. В компьютерной архитектуре x86-64 длительный режим – это режим, в котором 64-разрядная операционная система может получить доступ к 64-разрядным инструкциям и регистрам, о которых мы говорили чуть выше. 64-разрядные программы выполняются в подрежиме, называемом 64-разрядным режимом, в то время как 32-разрядные программы и 16-разрядные программы в защищенном режиме выполняются в подрежиме, называемом режимом совместимости. Реальный или виртуальный режим 8086 программы изначально не может запускаться в длительном режиме.

На этом нам пока теории будет достаточно. Конечно, тем, кто не слишком хорошо знаком с ассемблером, я рекомендую поработать с дополнительными ресурсами в сети Интернет, в частности с сайтами из библиографии, представленными в разделе «Изучение ассемблера». В части разработки собственных приложений под Windows я рекомендую обратить особое внимание на книгу В. Ю. Пирогова «Ассемблер для Windows», которая также представлена в библиографии.

Еще в приложении приводится полный список инструкций для 32- и 64-битных архитектур.

Теперь давайте перейдем к практической части.

О реверсинге

Обратная разработка, или реверсивный инжиниринг, – это исследование программного обеспечения (в том числе вредоносного) с целью изучения принципов его работы. Мы начнем с рассмотрения необходимых для исследования программного обеспечения инструментов, а затем погрузимся в саму специфику обратной разработки; решая CrackMe – небольшие приложения, мы рассмотрим основы языка ассемблера.

Ассемблер и компиляторы

Итак, откомпилированное приложение представляет собой машинные коды, которые понимает процессор. Однако для того чтобы программистам было удобнее понимать машинные инструкции, разработали язык ассемблера, который позволяет использовать более удобные для человека мнемонические (символьные) обозначения команд. Язык ассемблера – язык программирования «низкого уровня». Существуют языки высокого уровня, такие как C или Ру-

thon. Эти языки гораздо удобнее для программирования, поскольку не требуют для выполнения той или иной команды написания большого количества кода. Однако для выполнения программы на языке высокого уровня необходимо сначала перевести на язык ассемблера, чтобы компьютер их понял и смог исполнить. Таким образом, для того чтобы понимать, как работает программа, реверсеру необходимо знать язык ассемблера.

Для написания программ на языке ассемблера нам потребуются компиляторы. Совершенно очевидно, что для того чтобы научиться реверсингу, необходимо уметь программировать на языке ассемблера. Кроме того, зачастую при анализе приложений возникает необходимость написания собственного кода. Так, при исследовании приложений на переполнение буфера нам необходимо «скормить» уязвимому приложению произвольный код, который затем оно должно выполнить. Поэтому навык программирования на ассемблере реверсеру необходим.

Наиболее распространенным и, что немаловажно, бесплатным компилятором для 32- и 64-битных архитектур является FASM (Flat Assembler). Скачать его можно со страницы проекта: *https://flatassembler.net/*.

Загружаем архив с последней версией. Никакая установка не требуется, просто распаковываем архив и запускаем файл fasmw.exe. Для проверки корректности работы компилятора откроем в папке, куда был распакован FASM, каталог Examples, далее Hello и файл hello.asm.

```
; example of simplified Windows programming using complex macro features
include 'win32ax.inc' ; you can simply switch between win32ax, win32wx, win64ax
and win64wx here
.code
  start:
        invoke  MessageBox,HWND_DESKTOP,"May I introduce myself?",invoke
GetCommandLine,MB_YESNO
        .if eax = IDYES
                invoke  MessageBox,HWND_DESKTOP,"Hi! I'm the example
program!","Hello!",MB_OK
        .endif
        invoke  ExitProcess,0
.end start
```

Далее жмем **Run** и общаемся с диалоговыми окнами. Строго говоря, тот пример кода, который мы выполнили, – это не совсем чистый ассемблер, здесь используются так называемые макросы – шаблоны для генерации кода. Один раз создав макрос, мы можем использовать его во многих местах в коде программы. Макросы существенно упрощают жизнь разработчикам, делая процесс программирования на ассемблере более приятным и простым, а код программы получается понятнее, более схожим с кодом на языках высокого уровня.

Однако, как мы увидим дальше, при реверсинге макросы не видны, поэтому представленный выше пример используется лишь для того, чтобы быстро и просто проверить корректность работы компилятора.

Помимо FASM, существуют также иные компиляторы, такие как NASM, MASM и другие. Некоторые из них не поддерживают 64-битную архитектуру и давно не используются. На практике синтаксис написания кода может отличаться в зависимости от используемого компилятора, однако при необходимости можно привлечь любой компилятор.

В данной книге некоторые примеры кода будут представлены в нотации MASM. Настоящий реверсер, да и не только реверсер, но и разработчик, не должен привязываться к одному инструменту или технологии. Необходимо отталкиваться от решаемых задач и использовать те инструменты, которые лучшим образом подходят для решения.

Поэтому большинство примеров кода написаны под FASM, но в некоторых случаях используется MASM. Загрузить компилятор можно с этого ресурса: *https://masm32.com/download.htm*.

Отладчики и дизассемблеры

Основными инструментами реверс-инженера являются отладчик и дизассемблер. Разберемся, чем эти инструменты отличаются друг от друга.

Отладчик запускает целевую программу в контролируемых условиях, которые позволяют программисту отслеживать ее текущие операции и мониторить изменения в компьютерных ресурсах (чаще всего областях памяти, используемых целевой программой или операционной системой компьютера), которые могут указывать на неисправность кода.

Дизассемблер – это транслятор, преобразующий машинный код, объектный файл или библиотечные модули в текст программы на языке ассемблера.

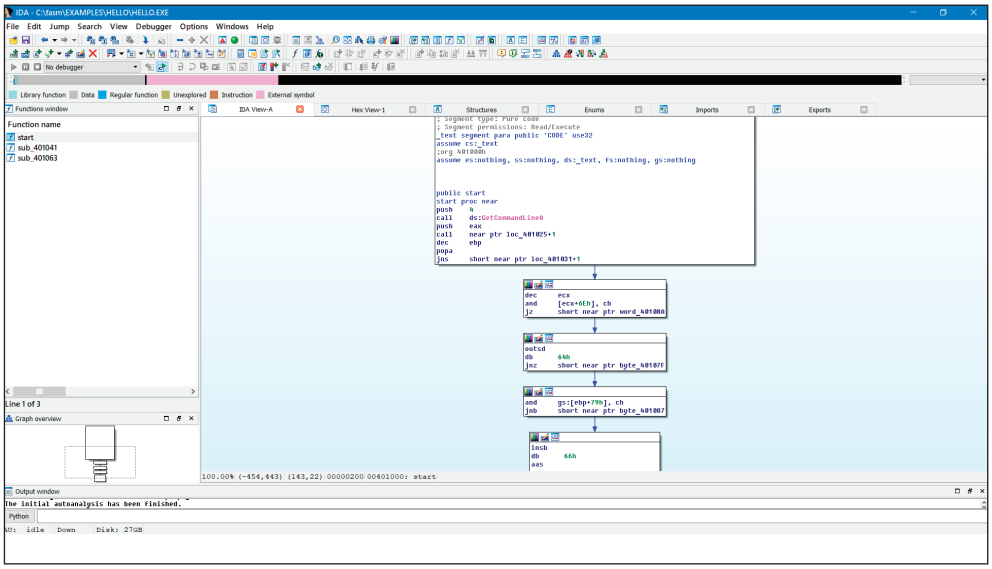
Таким образом, отладчик является мощным инструментом, который позволяет отслеживать и исправлять ошибки на всех этапах разработки. Также он может быть использован для отладки программы на разных уровнях – от отдельных функций до целого приложения.

Дизассемблер транслирует машинный код обратно в инструкции ассемблера. Машинный код – это бинарное представление исполняемого кода, который использует процессор для выполнения задач внутри компьютера.

Самым известным дизассемблером является IDA – Interactive Disassembler. IDA Pro Disassembler отличается исключительной гибкостью, наличием встроенного командного языка, поддерживает множество форматов исполняемых файлов для большого числа процессоров и операционных систем.

Для начала можно воспользоваться бесплатной редакцией IDA Free, которую можно загрузить по адресу *https://hex-rays.com/ida-free/*.

После установки запустим IDA Free, в появившемся окне выберем **New** и укажем наш файл hello.exe, который был получен в результате компиляции в FASM.

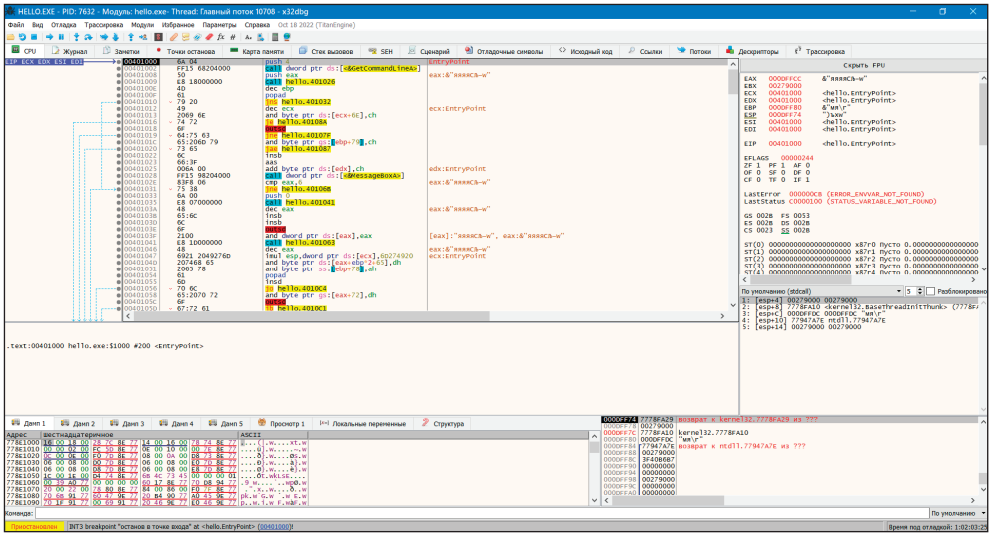


В результате мы получили дизассемлированный вид нашего выполняемого файла.

Теперь поговорим об отладчике. В принципе, в IDA тоже есть свой отладчик, но я предлагаю использовать x64dbg в качестве основного средства отладки. Этот отладчик является также бесплатным и работает как в 32-битной, так и в 64-битной архитектуре. Загрузить его можно со страницы *https://x64dbg.com/*.

После установки отладчик становится доступным при нажатии правой кнопки мыши на любом выполнимом файле. Выберем наш выполнимый файл hello.exe и после нажатия правой кнопки мыши выберем x64dbg.

В открывшемся окне с кодом необходимо один раз нажать **Выполнить**  . Далее мы увидим, как код программы разместился в памяти.



Пока нам этого достаточно, для того чтобы анализировать крякмиксы. Однако для более серьезного реверсинга необходимы также еще некоторые средства, о которых речь пойдет в соответствующих главах.

Только виртуализация

Для решения крякмиксов можно использовать хостовую машину, так как они не представляют какой-либо угрозы. Однако для анализа различных подозрительных файлов, хакерских инструментов и вредоносов обязательно нужно использовать изолированную среду. Лучше всего применять виртуальные машины, например Virtual Box или бесплатные редакции VMWare Workstation. При большом желании можно, конечно, использовать контейнеры, но тогда необходимо позаботиться о сохранении данных.

В качестве рабочей ОС я обычно использую Windows 7. С одной стороны, эта устаревшая ОС поддерживается большинством приложений, включая и вредоносные. С другой стороны, в ней нет «новейших механизмов защиты от Microsoft», которые по факту только мешают отладке.

Основы реверсинга, начинаем ломать

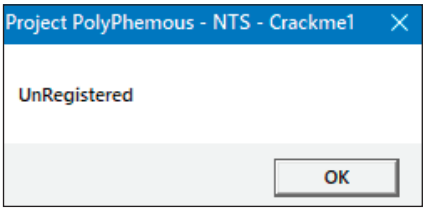
До этого мы установили все необходимые для реверсинга инструменты: отладчик, дизассемблер и другие дополнительные средства. Далее давайте разберем на примерах основные моменты, связанные с реверсивным инжинирингом, посмотрим разбор Crackme.

О патчинге

Программы используют различные механизмы защиты. Типичным случаем является необходимость ввести код для отключения какого-либо защитного механизма, альтернативой выступает отключение защиты после активации по сети. При этом по сети также передается некоторый код. Решить проблему с кодом можно двумя способами: изменив код в исходном приложении, например убрав проверку правильности ввода (патчинг), или же можно исследовать алгоритм, по которому генерируется правильный код, и затем сгенерировать нужный код своими силами. В первом случае мы вносим изменения в исходное приложение, во втором случае нет.

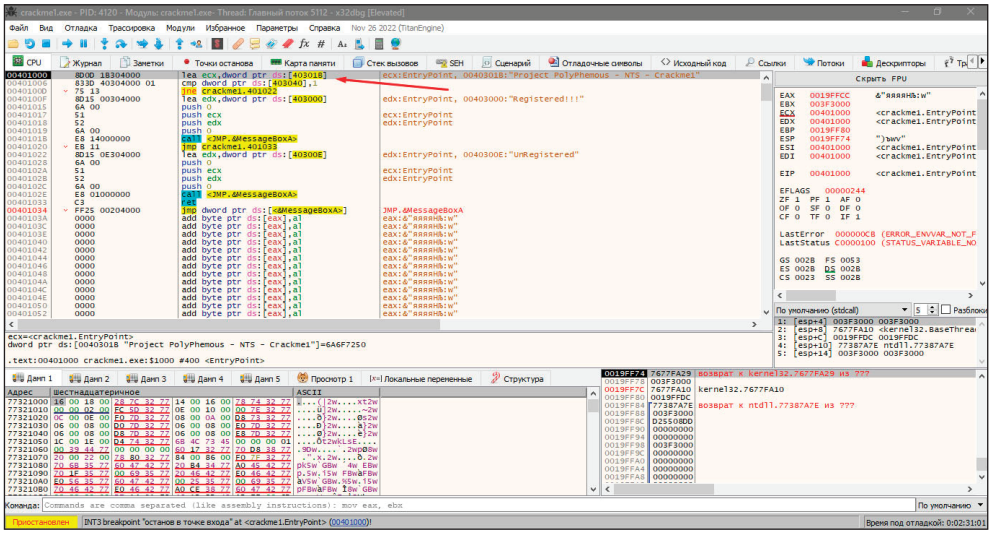
Посмотрим пример патчинга. В качестве подопытного у нас выступит очень простой Crackme.

Простой запуск выполнимого файла приведет к появлению следующего сообщения:



Собственно, нас даже ни о чем не спросили, а сразу выдали это сообщение. Приступим к исследованию.

Откроем файл в x64dbg. Нажав один раз на выполнение, мы попадаем в начальную точку, с которой начнется выполнение кода программы.



Первой идет еще незнакомая нам инструкция LEA. Эта команда вычисляет эффективный адрес источника, в данном случае 0x403018, и помещает его в приемник – регистр ECX. На скриншоте правее от этой строки отладчик показывает, что именно находится по этому адресу: текст «Project PolyPhemous...».

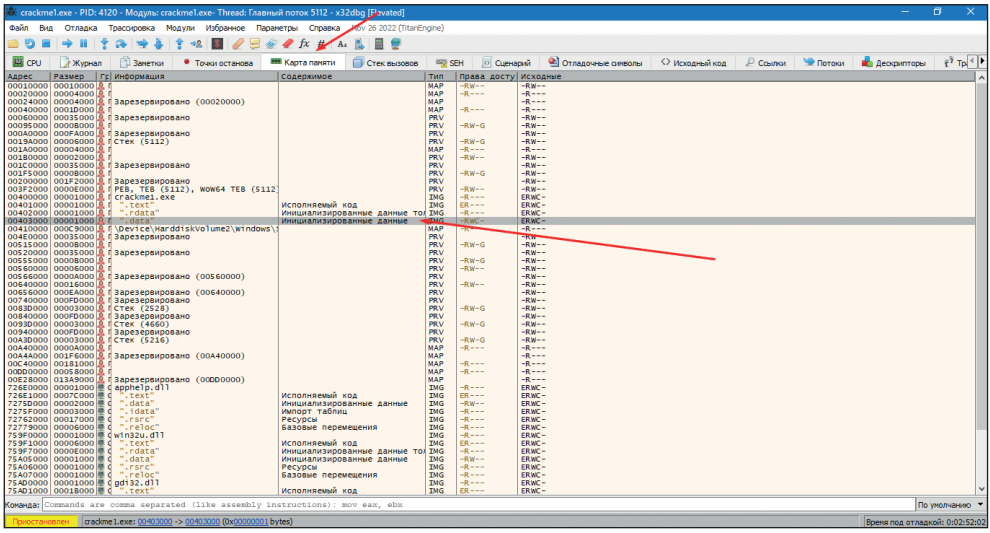
Второй строкой идет инструкция CMP, которая сравнивает содержимое ячейки памяти 0x403040 с единицей. В случае если значение не равно единице (инструкция JNE в следующей строке), выполняется переход на адрес 0x401022. В противном случае последовательно выполняем инструкции, идущие далее, в частности загружаем в EDX адрес 0x403000, по которому находится строка Registered!!! Также в стек помещается несколько служебных значений (команды PUSH), и вызывается функция операционной системы MessageBoxA, которая выводит окно с сообщением.

В случае перехода на адрес 0x401022 формируется сообщение Unregistered, далее также выводится окно.

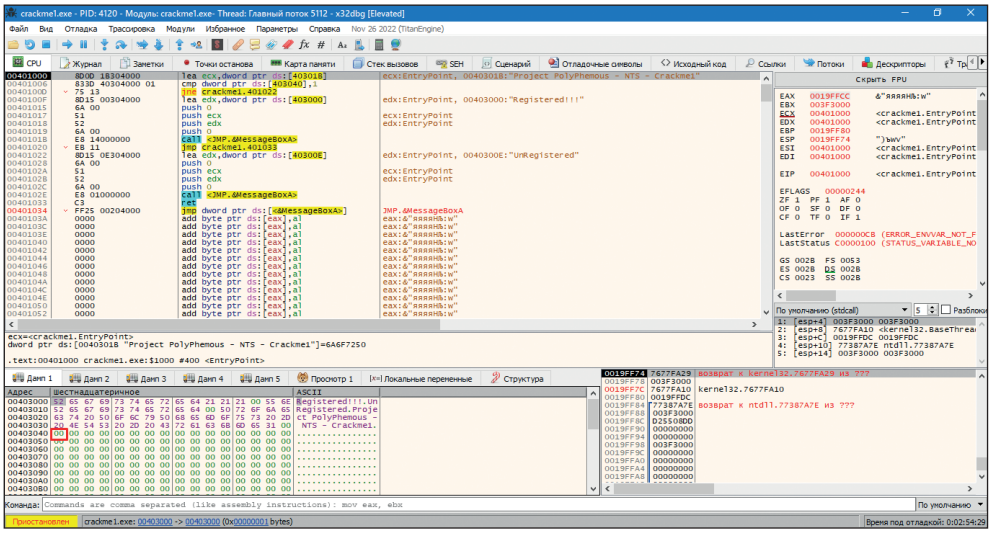
Здесь возможны несколько вариантов решения: можно убрать или поменять проверку условия, можно убрать условный переход, можно поправить область памяти.

Но для начала в целях обучения выполним по шагам каждую из приведенных инструкций. Нажимая **F7**, по очереди выполним каждую из команд. Очевидно, что текущий код выведет сообщение Unregistered, так как выполняется переход по команде JNE. Давайте посмотрим, какое значение находится по адресу 0x403040.

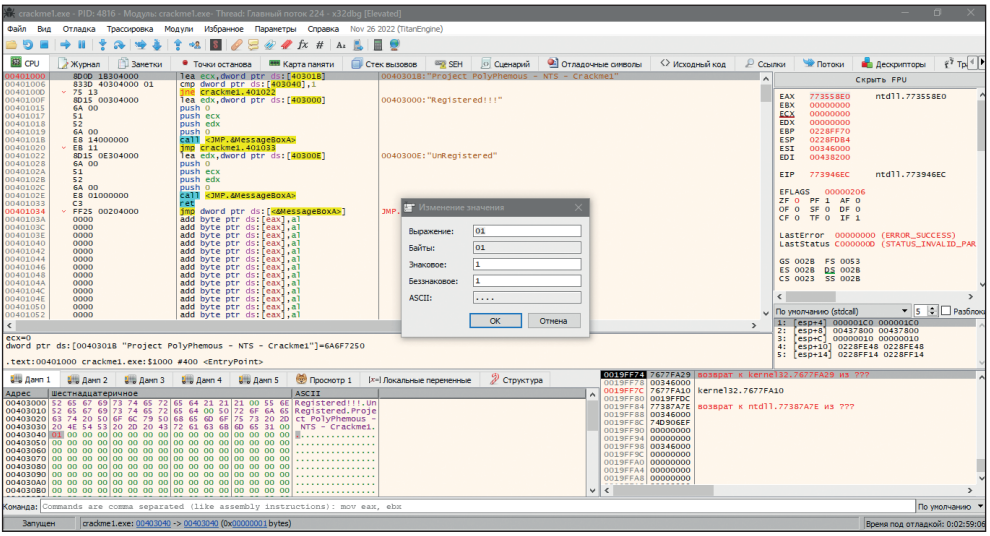
Для этого в меню отладчика выберем вкладку **Карта памяти** и далее сегмент **.data**.



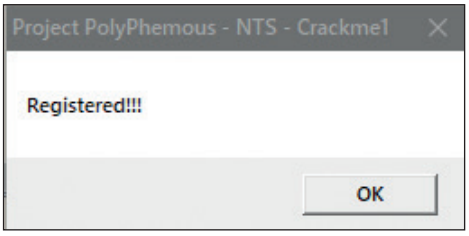
В современных приложениях сегменты кода и данных разделены. Выберем этот сегмент и убедимся, что по адресу 0x403040 находится 0.



Давайте попробуем поменять это значение. Для начала нажмем **Ctrl**+**F2** и перезапустим приложение в отладчике. Далее нажмем **F9**, после этого снова перейдем в карту памяти и отобразим содержимое адреса 0x403040. По правой кнопке мыши выберем **Изменить значение** и укажем единицу.

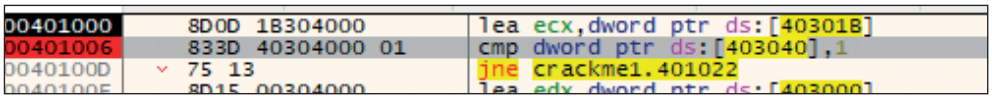


Снова запустим программу на выполнение. Получаем сообщение Registered.



Прежде чем реализовать другие способы патчинга, познакомимся с таким полезным инструментом отладки, как **Точки останова** (Breakpoints). Мы можем остановить выполнение программы на любом шаге, просто поставив брейк-пойнт на этот адрес.

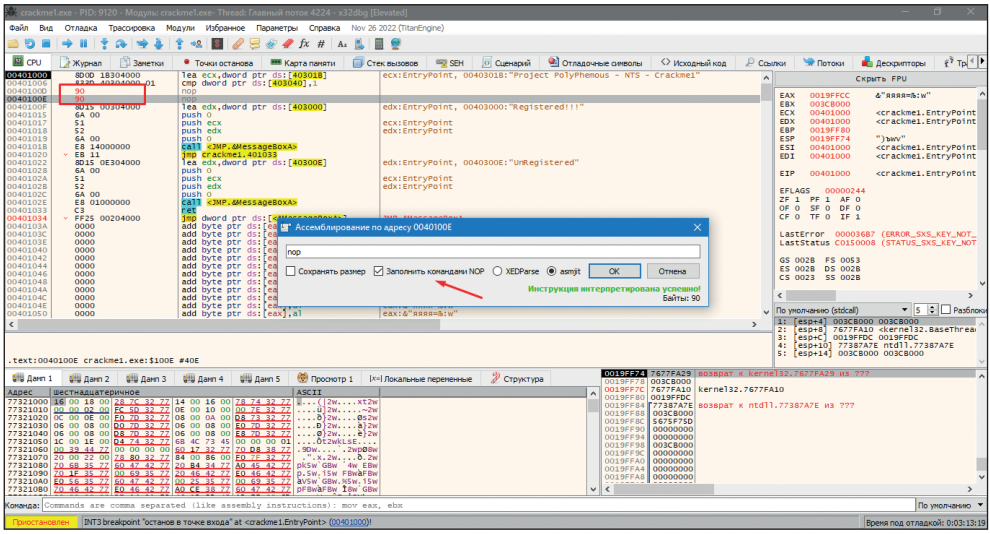
Снова перезапустим программу, нажав **Ctrl**+**F2**. Выберем вторую строку CMP… и нажмем на ней **F2**. Точка останова поставлена. Теперь если запустить программу на выполнение, она остановится на этой строке.



Далее попробуем убрать проверку JNE… Для простоты мы заменим эту команду инструкциями NOP. NOP – сокращение от No Operations, то есть эта команда ничего не делает. В первых процессорах данная команда использовалась для искусственного замедления работы программ, аналог команды sleep в языках высокого уровня.

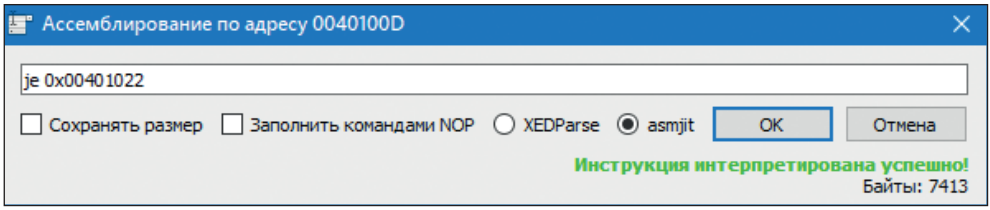
Мы не можем просто удалить ненужную команду в отладчике, но вместо этого мы можем ее заменить. Как видно в среднем столбце на рисунке, инструкции JNE соответствуют байты 0x75 и 0x13. Мы заменим NOP оба этих байта. Для этого нажмите правую кнопку мыши на строке с JNE и выберите

Ассемблировать. Далее укажите в строке ввода **nop** и выберите **Заполнить командами NOP**.



Инструкция JNE успешно заменена. В результате получаем значение Registered.

И в завершение рассмотрим еще один способ патчинга. Мы не будем заменять JNE командами NOP, а вместо этого заменим ее командой JE – переход, если равно. Для этого также откроем окно **Ассемблировать** на строке с JNE и заменим эту команду JE. При этом адрес перехода оставим неизменным. К слову, если напишем некорректную команду, в правом нижнем углу будет выведено соответствующее сообщение. Если мы ввели все корректно, то будет сообщение **Инструкция интерпретирована успешно!**.



И снова получаем сообщение Registered.

Подведем промежуточные итоги

В этом разделе мы рассмотрели пример патчинга – изменения выполнимого файла с целью обхода каких-либо проверок или защитных механизмов. В реальности различные кряки обходят механизмы проверки лицензий в приложениях, просто заменяя несколько байтов в выполнимых файлах. Но у патчинга есть существенный недостаток – изменение даже одного бита в файле приводит к изменению его контрольной суммы – параметра, который используют средства защиты для контроля целостности файла. Выполнимые файлы

не изменяются в процессе работы, поэтому их контрольная сумма должна оставаться неизменной.

Альтернативой патчингу является написание генератора ключей – кейгена. То есть мы не будем вносить изменения в файл, а вместо этого попытаемся разгадать принцип, по которому генерируются ключи.

Давайте рассмотрим написание кейгена.

Пишем кейген

Ранее мы осуществили патчинг, то есть поменяли несколько байтов в выполнимом файле, для того чтобы изменить логику работы программы, но теперь мы ни одного бита в выполнимых файлах менять не будем. Вместо этого мы попытаемся понять, по какому алгоритму осуществляется проверка, и передать программе нужный ключ.

Алгоритм проверки ключа может работать по-разному. В простейшем случае можно просто осуществлять проверку переданного ключа на соответствие какому-то заранее зашитому в программе набору байтов. В случае полного совпадения получаем Success, в противном случае – Fail.

Однако наиболее распространены алгоритмы проверки ключа, генерирующие правильный ключ в процессе работы программы. Так, часто ключ генерируется на основе имени узла, на котором запускается программа, времени запуска, IP-адреса и других параметров.

Есть и более творческие варианты проверок. Допустим можно обращаться по сети к какому-либо узлу, например по протоколу HTTP, и загружать с него нужный контент для проверки ключа. Также можно проверять наличие файла с ключом или процесса в памяти.

Кстати, все представленные методы могут использоваться для защиты реальных программ. От простейших проверок зашитого кода и до генерации сложных ключей, активации по сети или через файлы.

Виды механизмов защиты

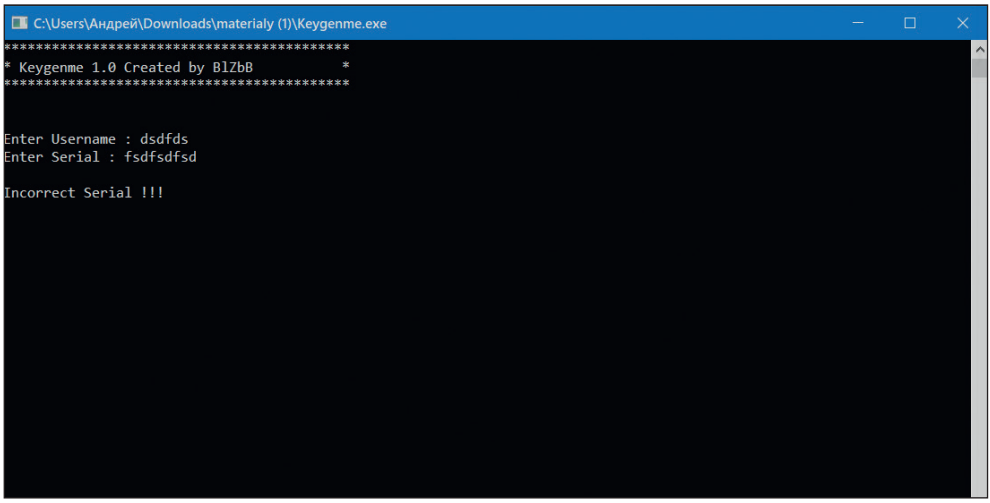
Для варианта с жестко захардкоженным ключом многое зависит от того, смогли ли мы правильно определить место в коде, где осуществляется проверка ключа и далее необходимо найти, где этот ключ, собственно, хранится.

Для борьбы с подобными механизмами защиты никаких дополнительных приложений, как правило, писать не нужно. Ключ – один для всех инсталляций, и нам необходимо лишь его узнать. Но на практике такую защиту используют лишь на очень старых программах, гораздо интереснее формировать нужный ключ динамически, в зависимости от определенных условий.

Далее посмотрим вариант разбора крякми с генерацией пароля. Здесь в качестве рабочего инструмента мы будем использовать дизассемблер IDA. С его помощью мы разберем, как выглядит код нашего крякмикса, и попробуем понять принцип генерации пароля.

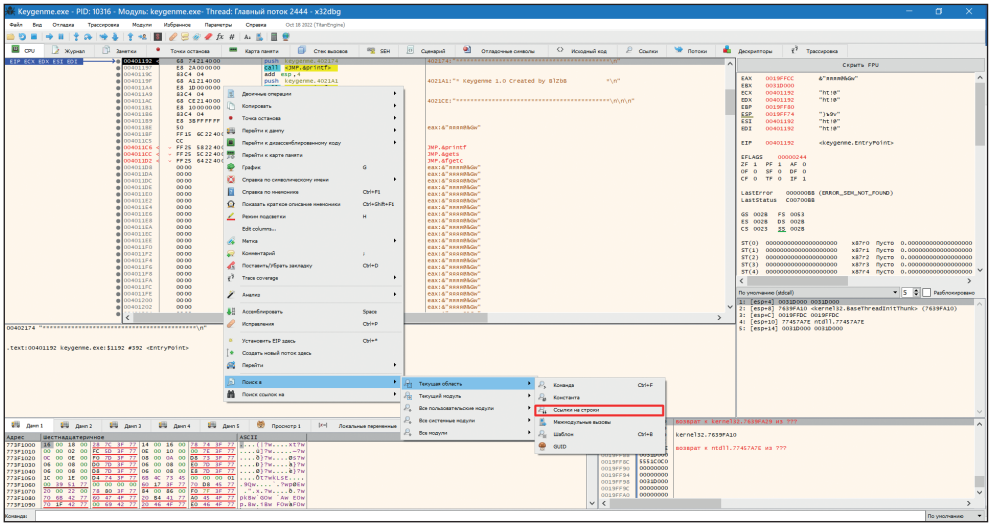
Разбор крякми

Для начала просто запустим наш крякмикс keygenme и посмотрим, что происходит. У нас запрашивают логин и пароль. После ввода случайных значений предсказуемо получаем сообщение о некорректном серийнике.

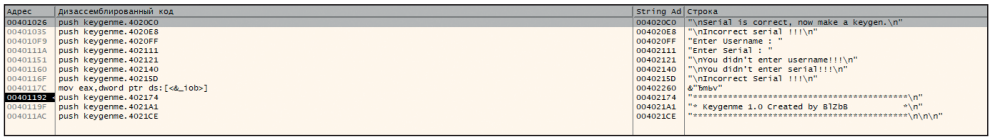


Откроем этот файл в отладчике и поищем вхождения всех текстовых строк. Этот прием далеко не всегда является полезным, иногда текстовые вхождения могут только еще больше запутать исследователя кода, но в простых случаях с незащищенными приложениями он тоже может помочь.

Нажимаем правую кнопку мыши, выбираем **Поиск в** -> **Текущая область** -> **Ссылки на строки**. В принципе, для полного поиска по всему коду можно выбрать **Все модули** -> **Ссылки на строки**, но такой поиск займет больше времени.

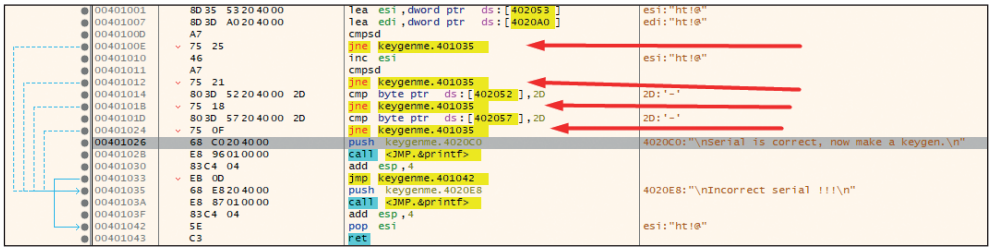


Получим информацию о найденных текстовых строках.



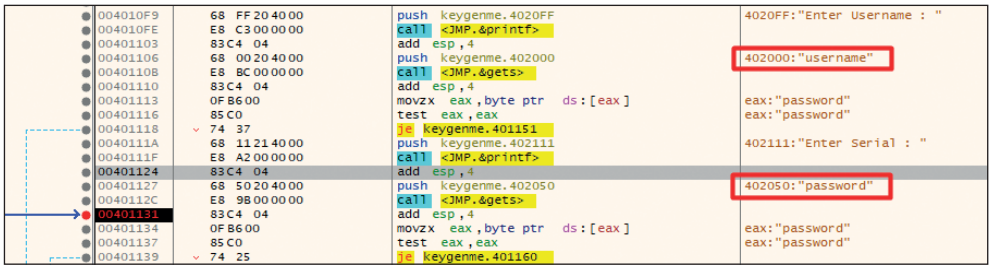
В случае если при таком поиске ничего найти не удалось, можно, во-первых, поискать по всем модулям, а во-вторых, в процессе выполнения программы, остановившись на каком-либо брейк-пойнте, еще раз попробовать поискать. Иногда это помогает найти новые строковые вхождения.

Теперь нам достаточно будет перейти на нужный адрес по строке **Serial is correct...**. Далее мы видим целый ряд проверок, по результатам каждой мы можем оказаться в блоке с Incorrect serial.



Если бы мы говорили о патчинге, то нам было бы достаточно просто забить NOP'ами соответствующие команды. Но сейчас мы хотим разобраться в том, как проверяется серийный номер.

Для этого можно, конечно, поставить брейк-пойнт сразу на адрес 0x401001, где начинается блок проверки, но есть некоторая вероятность, что мы до него не дойдем, так как ранее встречаются другие проверки. Поэтому лучше переместиться немного подальше, найти блок с вводом Username и Serial и поставить брейк-пойнт там с помощью клавиши **F2**. Далее запустим программу и введем осмысленные значения, например username и password. Я поставил брейк сразу после вызова функции, считывающей серийник.



Теперь мы знаем, по каким адресам в памяти хранятся Username и Serial. Далее можно пошагово перемещаться по программе, выполняя команды последовательно. Переходим на адрес 0x4010c0. Здесь довольно интересный блок кода. В нем программа берет поочередно блоки по четыре байта из серийного номера и на выходе возвращает в ECX количество этих блоков. Для

значения password это будет 2, а после возращения из этой подпрограммы выполняется sub ECX,3 (вычитание из ECX). В нашем случае значение будет отрицательным, и следующая команда jne отправит нас на Incorrect Serial. Таким образом, мы узнали, что длина серийника должна быть не меньше 12 символов. Проверим это. Введем password1234. Теперь ECX равно 3, и переход по jne не произойдет.

Следующий блок кода 0x401049 обрабатывает значение username. Обратим внимание на инструкции cld и repne scasb. Команды CLD и STD позволяют сбросить или установить флаг направления DF (Direction Flag). Команда CLD (Clear DF) сбрасывает флаг в значение 0, а команда STD (Set DF) устанавливает его в значение 1. Проще говоря, CLD говорит, что читать строку мы будем слева направо. Scasb – это поиск в строке байтов, а при использовании префикса repne scas сканирует строку в поисках первого элемента, равного значению в регистре al (мы используем только младшую часть EAX).

```
00401049    53              push  ebx
0040104A    56              push  esi                          esi:"ht!@"
0040104B    57              push  edi                          edi:"ht!@"
0040104C    33C0            xor   eax,eax
0040104E    8D3D 00204000   lea   edi,dword ptr  ds:[402000]  edi:"ht!@", 00402000:"username"
00401054    B9 FF000000     mov   ecx,FF
00401059    FC              cld
0040105A    F2:AE           repne  scasb
0040105C    F6D1            not   cl
0040105E    49              dec   ecx
0040105F    33DB            xor   ebx,ebx
00401061    8D35 00204000   lea   esi,dword ptr  ds:[402000]  esi:"ht!@", 00402000:"username"
00401067    33C0            xor   eax,eax
00401069    AC              lodsb
0040106A    F6E1            mul   cl
0040106C    03D8            add   ebx,eax
0040106E    47              inc   edi                          edi:"ht!@"
0040106F    FEC9            dec   cl
00401071    75 F4           jne   keygenme.401067
00401073    81F3 13131313   xor   ebx,13131313
00401079    F7D3            not   ebx
0040107B    81F3 CDAB3412   xor   ebx,1234ABCD
00401081    8BC3            mov   eax,ebx
00401083    25 0F0F0F0F     and   eax,F0F0F0F
00401088    81E3 F0F0F0F0   and   ebx,F0F0F0F0
0040108E    C1EB 04         shr   ebx,4
00401091    A3 A0204000     mov   dword ptr  ds:[4020A0],eax
00401096    891D A4204000   mov   dword ptr  ds:[4020A4],ebx
0040109C    BE A0204000     mov   esi,keygenme.4020A0          esi:"ht!@"
004010A1    B9 08000000     mov   ecx,8
004010A6    803E 09         cmp   byte ptr  ds:[esi],9         esi:"ht!@", 9:'\t'
004010A9    77 05           ja    keygenme.4010B0
004010AB    800E 30         or    byte ptr  ds:[esi],30        esi:"ht!@"
004010AE    EB 03           jmp   keygenme.4010B3
004010B0    8006 37         add   byte ptr  ds:[esi],37        esi:"ht!@"
004010B3    46              inc   esi                          esi:"ht!@"
004010B4    49              dec   ecx
004010B5    75 EF           jne   keygenme.4010A6
004010B7    5F              pop   edi                          edi:"ht!@"
004010B8    5E              pop   esi                          esi:"ht!@"
004010B9    5B              pop   ebx
004010BA    C3              ret
004010BB    CC              int3
```

В результате преобразований мы получим набор байтов, находящихся по адресу 0x4020a0.

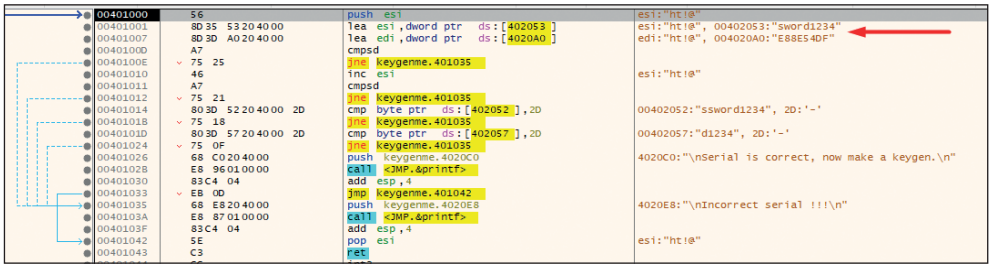
Адрес	Шестнадцатеричное																ASCII
00402030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00402040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00402050	70	61	73	73	77	6F	72	64	31	32	33	34	00	00	00	00	password1234....
00402060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00402070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00402080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00402090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
004020A0	45	38	38	45	35	34	44	46	00	00	00	00	00	00	00	00	E88E54DF .....
004020B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
004020C0	0A	53	65	72	69	61	6C	20	69	73	20	63	6F	72	72	65	.Serial is corre

Следующим блоком кода, на который будет выполнен переход, является переход на адрес 0x401000. Здесь мы видим наш сгенерированный набор байтов

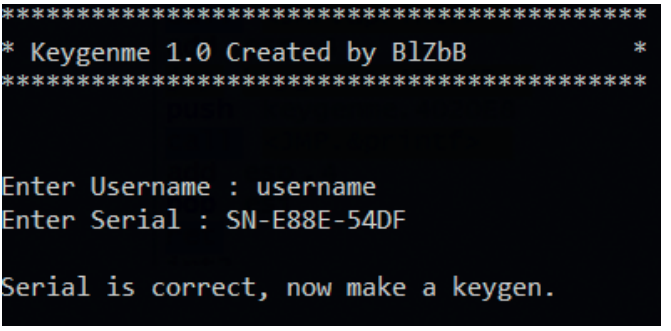
по адресу 0x4020a0 и фрагмент исходного серийника, начинающийся с адреса 0x402053, то есть с четвертого байта.

Далее команда CMPSD сравнивает двойное слово из памяти по адресу DS:SI с двойным словом по адресу ES:DI. Аналогична по действию команде CMP. Очевидно, что в нашем случае значения не совпадут.

Давайте сразу посмотрим на код дальше. На третьей и восьмой позициях в серийнике должны быть знаки разделителя «-».



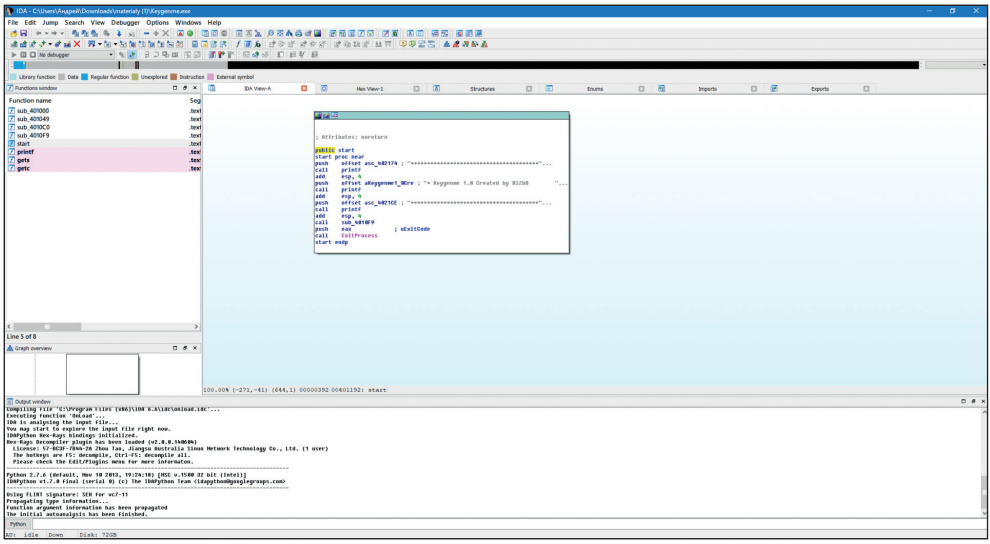
То есть формат серийника XX-XXXX-XXXX. По сути, мы уже можем на основе этих данных подготовить пару Username/Serial. В качестве имени пользователя оставляем слово username, а в качестве серийника указываем SN-E88E-54DF, где SN – это любые два символа, их все равно не проверяют. Давайте проверим.



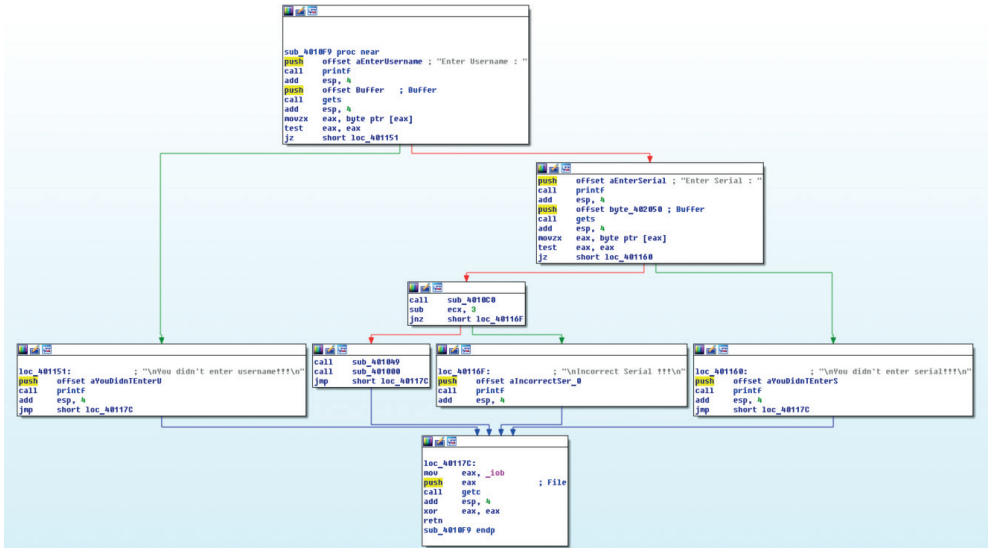
Как видно, мы правильно нашли области кода, в которых осуществляются генерация и проверка ключа, теперь давайте рассмотрим основные принципы написания генератора ключей и заодно поработаем в дизассемблере IDA.

Пишем кейген

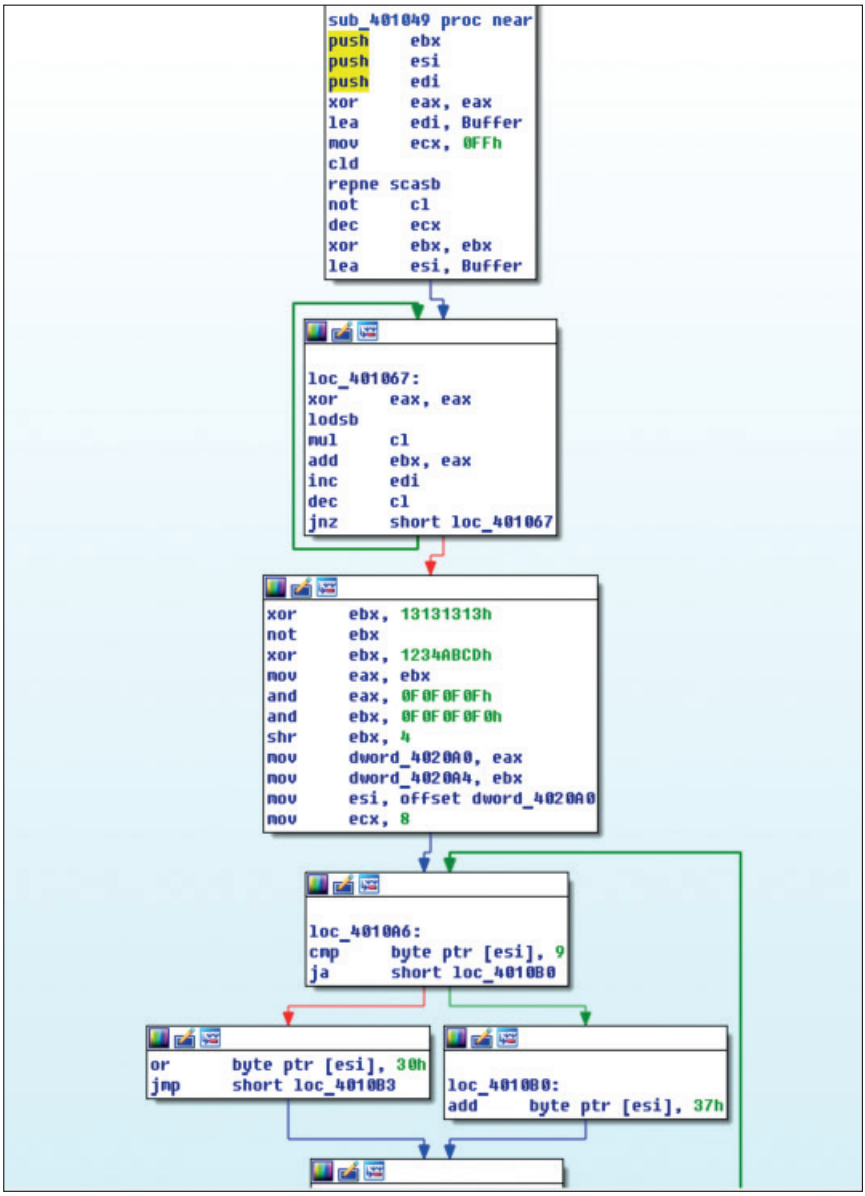
Откроем файл Crackme в IDA. Здесь мы видим несколько блоков кода, представленных в виде блоков со связями. Визуально такое отображение упрощает понимание принципов работы алгоритма.



Как можно увидеть, в левой части экрана указаны вызовы подпрограмм и, в частности, уже знакомые нам подпрограммы по адресам 0x4010F9 и 0x401049.



С точки зрения написания кейгена нас больше всего будет интересовать процедура, находящаяся по адресу 0x401049.



Если вы собираетесь писать свой кейген на языке высокого уровня, то вам необходимо будет реализовать все эти преобразования на соответствующем языке, но если вы будете использовать ассемблер, то можно просто скопировать фрагмент кода процедуры и поместить его в свой кейген. Для этого в IDA можно на нужной процедуре нажать правую кнопку мыши и выбрать **Text view**, а затем просто скопировать нужный фрагмент кода.

Ниже представлен фрагмент кода кейгена, который получает на вход username и serial как параметры командной строки и затем генерирует по ним нужный серийный номер.

```
START:
PUSH STD_OUTPUT_HANDLE
CALL GetStdHandle@4
MOV HANDL,EAX
CALL NUMPAR
CMP EAX,1
JE NO_PAR
MOV EDI,2
LEA EBX,USERNAME
CALL GETPAR
push    ebx
push    esi
push    edi
xor     eax, eax
lea     edi, USERNAME
mov     ecx, 0FFh
cld
repne scasb
not     cl
dec     ecx
xor     ebx, ebx
lea     esi, USERNAME

label4:
xor     eax, eax
lodsb
mul     cl
add     ebx, eax
inc     edi
dec     cl
jnz     short label4

xor     ebx, 13131313h
not     ebx
xor     ebx, 1234ABCDh
mov     eax, ebx
and     eax, 0F0F0F0Fh
and     ebx, 0F0F0F0F0h
shr     ebx, 4

LEA     esi,PASSWORD
mov     [ESI], eax
mov     [ESI+4], ebx
mov     ecx, 8

label7:
cmp     byte ptr [esi], 9
ja      short label5

or      byte ptr [esi], 30h
jmp     short label6
```

```
label5:
add     byte ptr [esi], 37h ; '7'

label6:
inc     esi
dec     ecx
jnz     label7

mov   ESI,offset PASSWORD
mov   EDI,offset PASSWORD1+3
MOV EAX,[ESI]
MOV [EDI],EAX
mov   ESI,offset PASSWORD+4
mov   EDI,offset PASSWORD1+8
MOV EAX,[ESI]
MOV [EDI],EAX


PUSH 0
PUSH OFFSET NUMW
PUSH 12
PUSH OFFSET PASSWORD1
PUSH HANDL
CALL WriteConsoleA@20

pop     edi
pop     esi
pop     ebx

NO_PAR:
PUSH 0
CALL ExitProcess@4
;retn

NUMPAR PROC
CALL GetCommandLineA@0
MOV ESI,EAX ;указатель на строку
XOR ECX,ECX ;счетчик
MOV EDX,1 ;признак
L1:
CMP BYTE PTR [ESI],0
JE L4
CMP BYTE PTR [ESI],32
JE L3
ADD ECX,EDX ;номер параметра
MOV EDX,0
JMP L2
L3:
OR EDX,1
L2:
INC ESI
JMP L1
```

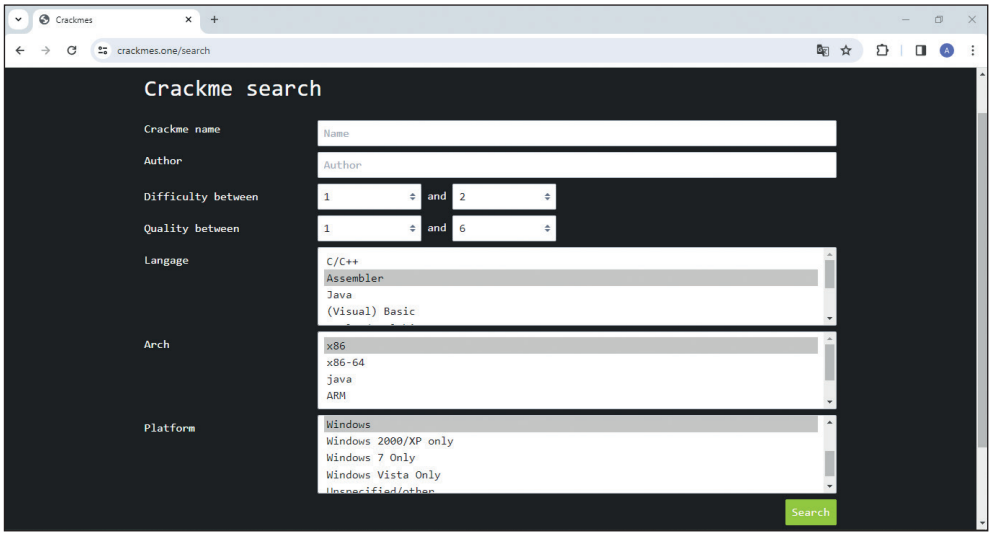
```
L4:
MOV EAX,ECX
RET
NUMPAR ENDP
;получить параметр из командной строки
;EBX указывает на буфер, куда будет помещен параметр
;в буфер помещается строка с нулем на конце
;EDI – номер параметра
GETPAR PROC
CALL GetCommandLineA@0
MOV ESI,EAX ;указатель на строку
XOR ECX,ECX ;счетчик
MOV EDX,1 ;признак
L1:
CMP BYTE PTR [ESI],0
JE L4
CMP BYTE PTR [ESI],32
JE L3
ADD ECX,EDX ;номер параметра
MOV EDX,0
JMP L2
L3:
OR EDX,1
L2:
CMP ECX,EDI
JNE L5
MOV AL,BYTE PTR [ESI]
MOV BYTE PTR [EBX],AL
INC EBX
L5:
INC ESI
JMP L1
L4:
MOV BYTE PTR [EBX],0
RET
GETPAR ENDP
_TEXT ENDS
END START
```

Здесь мы разобрали вариант защиты с использованием генерации серийного ключа в зависимости от имени пользователя либо других значений. Но в реальности можно встретить и более экзотические механизмы защиты. Например, можно встретить проверку, основанную на взаимодействии с узлом в сети и выдающую соответствующее сообщение в зависимости от результата. Здесь мы можем ограничиться отладчиком, хотя при работе с сетью может также потребоваться снифер, например Wireshark. Еще возможны варианты, когда проверяющий код ищет какой-либо файл на диске или процесс в сети.

Заключение

В этой главе мы рассмотрели основы языка ассемблера: регистры, инструкции, стек, управление памятью. Далее поговорили об основах реверсинга и разобрали несколько крякмиксов, проанализировали написание кейгена. Также мы узнали, как строится работа с дизассемблером и другими инструментами для реверсинга.

Для самостоятельной практики я бы рекомендовал воспользоваться ресурсом crackmes.one. На сайте выберите Search и далее настройте фильтр аналогично тому, как это представлено на рисунке.



По сложности для начала я бы рекомендовал установить верхнюю границу не выше 2. Ко многим, особенно старым, крякмиксам имеются решения, но я бы рекомендовал сначала постараться самостоятельно решить эти задачки и заглядывать в раздел Solutions только в крайнем случае.

На этом вводную часть в этой книге я предлагаю завершить и перейти к специализированным темам. И начнем мы с рассмотрения такой старой, но актуальной темы, как переполнение буфера.