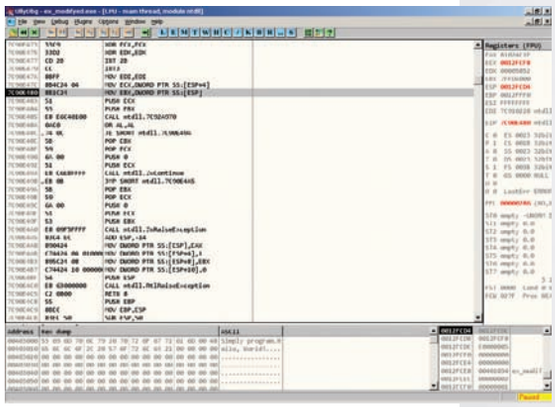
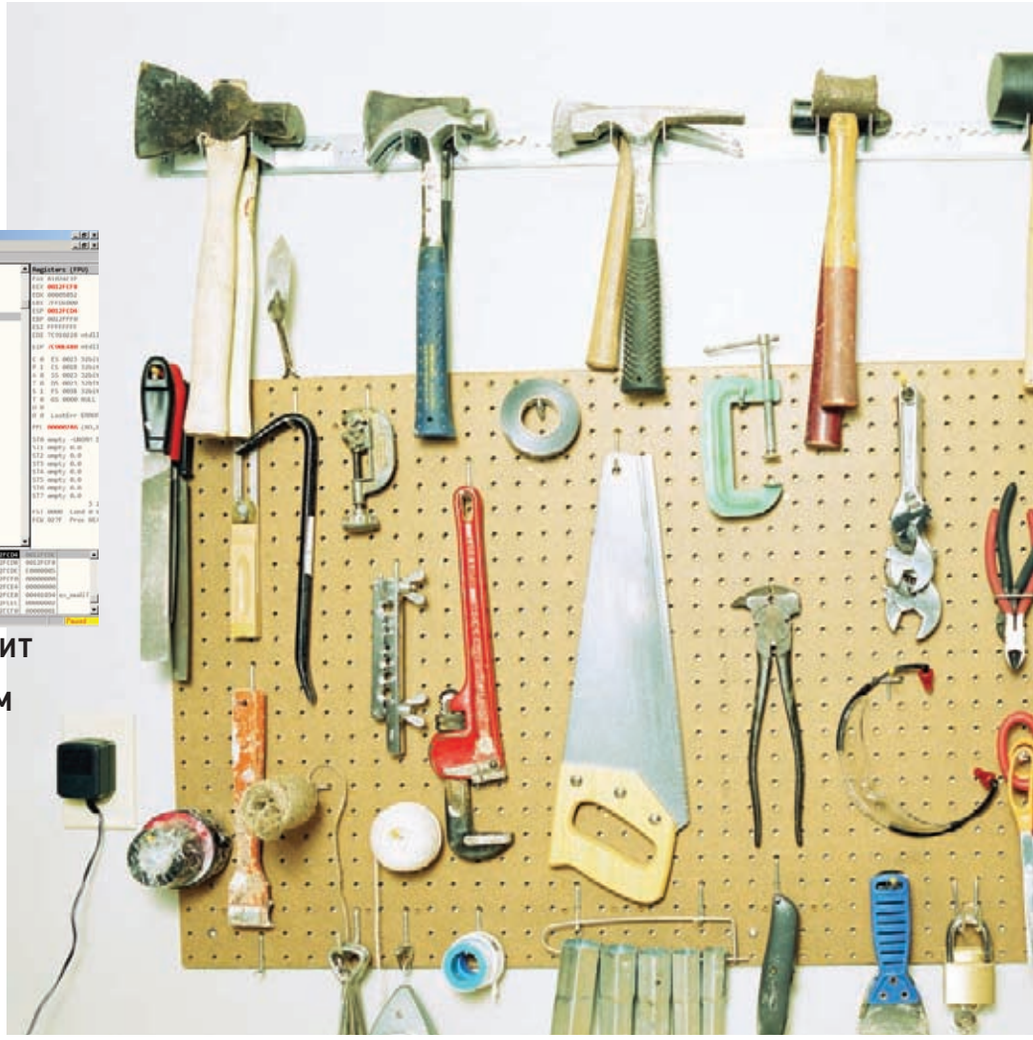
ЛЕОНИД «CR@WLER» ИСУПОВ / CRAWLER@XAKEP.RU /



«ДЕБРИ» КОДА, В КОТОРЫЕ УХОДИТ ЗАЩИЩЕННАЯ ПРОГРАММА, ВЫПОЛНЯЕМАЯ ПОД ОТЛАДЧИКОМ



АНТИОТЛАДОЧНЫЕ ТРЮКИ

Активно противодействуем отладке нашего приложения

В Сети можно найти огромное количество статей, посвященных реверсингу. Правда, большинство направлены на изучение методов снятия конкретных пакеров и протекторов. В то же время трудно найти хорошее руководство по антиотладке, которое бы описывало методы, применимые во многих случаях.

ВАШЕ ВРЕМЯ ВЫШЛО!

Факт выполнения кода под отладчиком скрыть достаточно сложно, это в особенности относится ко времени выполнения инструкций. Действительно, каждая инструкция выполняется процессором за определенное количество тактов. Зная тактовую частоту процессора, легко вычислить временной промежуток, необходимый для выполнения того или иного объема вычислений. Даже если невозможно определить, какой процессор воткнут в разъем материнской платы, можно рассчитать диапазон, в который теоретически укладывается скорость интерпретации машинных инструкций. Можно вводить поправку и на разницу в частотах, и на количество ядер процессора, и на технологии оптимизации выполнения кода, используемые в тех или иных процессорах. Но и это не столь важно, если рассматривать достаточно распространенное оборудование, например, процессоры Intel и AMD. Значит, можно использовать антиотладочные приемы, построенные на принципе замера времени исполнения некоторых инструкций.

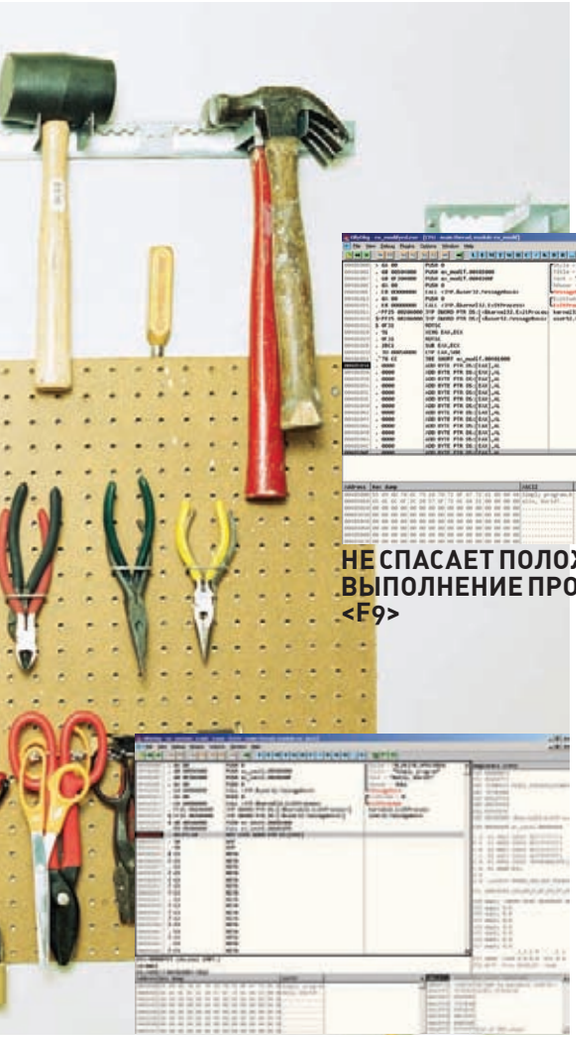
Работая над антиотладочным кодом, мы можем пойти по одному из следующих путей:

1. Замер времени выполнения инструкции, сравнение полученного времени со значением, которое укладывается в принятые рамки, но имеет «запас».

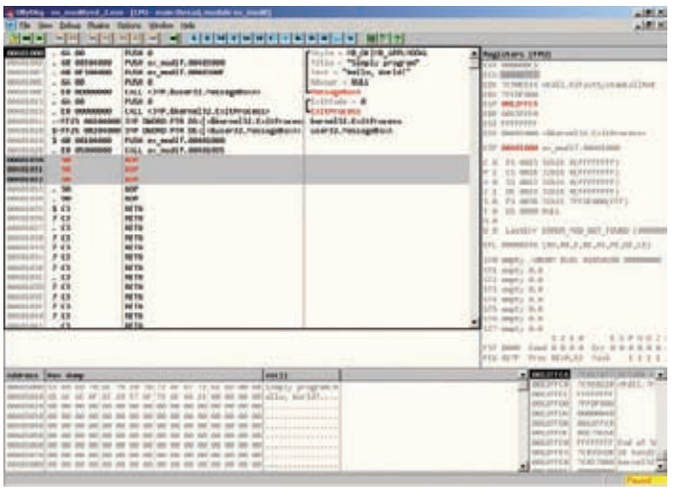
2. Точный замер времени выполнения ряда инструкций, например, очень большого цикла.

Как первый, так и второй способы имеют свои преимущества.

Процессор оперирует понятием такта, который



НЕ СПАСАЕТ ПОЛОЖЕНИЕ ДАЖЕ ВЫПОЛНЕНИЕ ПРОГРАММЫ ПО <F9>



В ПРИНЦИПЕ, ВЫХОД ЕСТЬ. ОБНУЛЕНИЕ СЧЕТЧИКА ИЛИ ЗАМЕНА ИНСТРУКЦИИ НА «NOP» ДАДУТ РЕЗУЛЬТАТ, НО ИСПОЛЬЗОВАНИЕ МОДИФИКАЦИЙ МЕТОДА ПОЗВОЛЯЕТ ИСКЛЮЧИТЬ И ЭТУ ВОЗМОЖНОСТЬ

КОД ВЫПОЛНЯЕТСЯ ПОД OLLYDBG ПО <F7>. РЕЗУЛЬТАТ — ВОЗНИКНОВЕНИЕ ИСКЛЮЧЕНИЯ

в некотором смысле является аналогом временного промежутка. Если каким-либо образом получить количество тактов, которое процессор выполнил с момента последнего сброса счетчика тактов, становится возможным использовать это значение, чтобы определить, выполняется ли поток машинного кода с положенной скоростью, или же программа запущена под отладчиком. Такая возможность существует, инструкция «rdtsc» предоставляет программисту средства для подсчета количества тактов, выполненных процессором с момента последнего сброса. Вот как описывает эту инструкцию «Википедия»:

> «**rdtsc** (Read Time Stamp Counter) — ассемблерная инструкция для платформы x86, читающая счетчик TSC (Time Stamp Counter) и возвращающая в регистрах EDX:EAX 64-битное количество тактов с момента последнего сброса процессора. rdtsc поддерживается в процессорах Pentium и более новых. Опкод: 0F 31. В многозадачных операционных системах инструкция может быть превращена в привилегированную (установлен 3 бит в управляющем регистре CR4), и ее использование приведет к генерации исключения в программе».

Для того чтобы использовать инструкцию RDTSC в антиотладочных целях, необходимо выполнить ее дважды: до и после выполнения кода, для которого будет производиться замер:

```
ADDRESS: RDTSC
ADDRESS_2: выполняемый код
ADDRESS_3:RDTSC
```

Естественно, код, для которого производится замер, может быть и не специально написанным набором инструкций, а какой-либо частью программы. Младшая часть 64-битной последовательности, содержащей количество тактов, помещается в регистр EAX. В большинстве случаев (для незначительного количества инструкций) изменен будет именно он, старшая же часть последовательности — регистр EDX — останется неизменной. Значит, если выполнить два замера количества тактов — до и после выполнения проверочного кода — и получить разность значений, которыми были инициализированы регистры EAX, полученное значение будет количеством тактов, которое процессор выполнил между замерами. Если предположить, что одна инструкция не может выполняться процессором за время, когда счетчик «наматывает» более 0x1000 тактов, можно реализовать антиотладочный прием следующим образом:

```
RDTSC
XCHG EAX, ECX
RDTSC
SUB EAX, ECX
CMP EAX, 1000
 JBE NOT_DEBUGGED
CALL Kernel32.TerminateProcess
…

NOT_DEBUGGED: выполнение программы
```

Инструкция «XCHG EAX, ECX» одновременно является и инструкцией, для которой замеряется «тактовый промежуток», и частью антиотладочного кода (производится сохранение содержимого регистра EAX в регистр ECX перед повторным получением количества тактов). После выполнения первых трех инструкций регистры EAX и ECX содержат значения, соответствующие количеству тактов, выполненных процессором в разное время (на моменты до и после вызова инструкции «XCHG»). Далее вычисляется их разность и ее сравнение со значением 0x1000. Полученная разность превысила заданную величину? Нас отлаживают, завершаем работу.

Попробуем использовать наш код в программе, написанной на ассемблере. Ее исходный код выглядит так:

```
.386

.model flat,stdcall

option casemap:none

; подключение необходимых библиотек:
include \masm32\include\windows.inc  ;
include \masm32\include\kernel32.inc ;
includelib \masm32\lib\kernel32.lib  ;
include \masm32\include\user32.inc   ;
includelib \masm32\lib\user32.lib    ;

; секция данных
.data
alert_upper  db "Simply program",0
alert_text   db "Hello, World!",0
```

```
; секция кода
.code

start:
  invoke MessageBox,
       NULL,
       addr alert_text,
       addr alert_upper,
       MB_OK

  invoke ExitProcess, NULL
end start
```

У тебя есть несколько путей реализации антиотладочного приема — можно внедрить нашу конструкцию непосредственно в исходный код, можно внести изменения в уже откомпилированный PE-файл. Я предпочитаю второй способ — он годится для защиты и тех программ, исходным кодом которых мы не располагаем. Значит, откомпилируем программу при помощи MASM («ml /c /coff /Cp имя_файла.asm»; «link /SUBSYSTEM:WINDOWS /LIBPATH:c:\masm32\lib /SECTION:.text,RWE имя_файла.obj»). И модифицируем ее любым отладчиком, например, OllyDBG. Для «подопытной», исходный код которой был рассмотрен выше, набор антиотладочных инструкций, базирующийся по адресу 0x401026, будет выглядеть так:

```
00401026  RDTSC
00401028  XCHG EAX,ECX
00401029  RDTSC
0040102B  SUB EAX,ECX
0040102D  CMP EAX,500
00401032  JBE SHORT ex_tickc.00401000
; переход к точке входа программы
```

Программиста не интересуют последствия выполнения кода, следующего после адреса 0x401032, хотя можно разместить ниже условного перехода инструкцию завершения работы программы. Результат обнадеживает: отладчик OllyDBG не справился с выполнением кода :).

ЖЕЛЕЗНАЯ АНТИОТЛАДКА

Особенности выполнения некоторых инструкций процессорами позволяет создавать антиотладочные методы, обойти которые способен не каждый реверсер. Это связано с особенностями архитектуры процессоров, которые не учитывают отладчики. В качестве примера можно привести особенности очереди предварительной выборки процессоров Intel. Сайт www.intel.com комментирует понятие «предварительная выборка»:

> «Исходя из содержания текущей команды или поставленной задачи, блок предварительной выборки определяет порядок запроса соответствующих данных и инструкций из командной кэш-памяти или системной памяти компьютера. По мере поступления инструкций важнейшей задачей блока предварительной выборки становится их правильное «выстраивание» и пересылка в блок декодировки».

Один из способов антиотладки, использующей механизм очереди предварительной выборки, — перезапись исполняемого кода. Рассмотрим псевдокод:

```
ADDRESS_01: CALL ADDRESS_03
ADDRESS_02: инструкции, подлежащие
исполнению
ADDRESS_03: MOV AL, 0C3h
MOV EDI, OFFSET ADDRESS_03
OR ECX, FFFFFFFF
REP STOSB
```

Исполнение этого набора инструкций приведет к различным результатам, в зависимости от того, как код был выполнен (под отладчиком или штатно), какие атрибуты имеет страница памяти, следующая за страницей, содержащей код. Если атрибут «writeable» секции не установлен, исполнение кода приведет к возникновению исключения, что неизбежно повлечет крах программы. Поэтому для его использования необходимо предусмотреть установку необходимых атрибутов на страницы памяти.

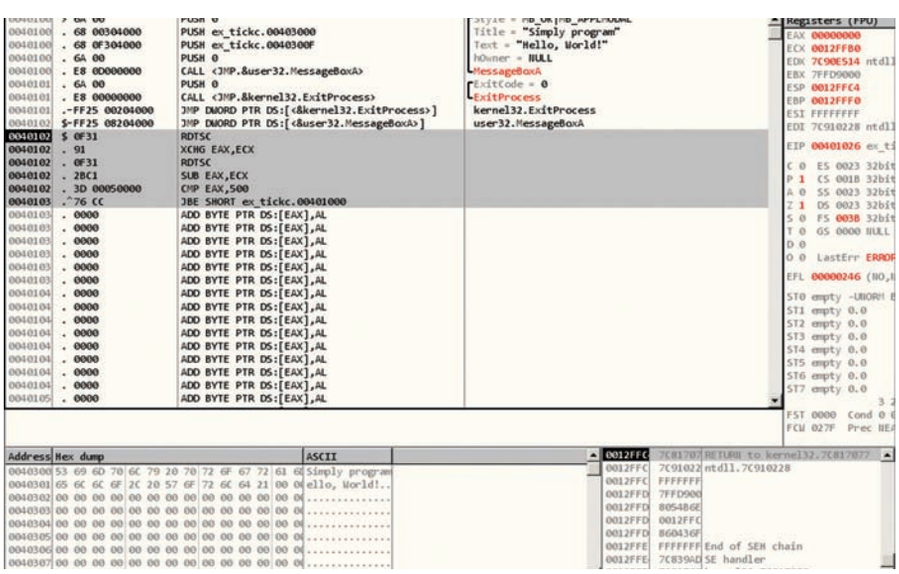
Итак, как уже было сказано, результат выполнения кода зависит от нескольких факторов. Функция данного кода, несложно догадаться — перезапись инструкций поверх уже существующих. Значит, поверх инструкции «REP STOSB» должен записаться код, соответствующий шестнадцатеричному значению 0xC3. Естественно предположить, что выполнение кода должно остановиться сразу после того, как инструкция REP STOSB будет перезаписана. Действительно, такое «поведение» процессора, выполняющего машинный код, кажется логичным — ведь на месте ранее выполнявшегося кода находится новый (машинный код 0xC3 соответствует инструкции RET). В случае если код выполнялся в контексте отладчика, все будет происходить именно таким образом — выполнение остановится, инструкция RET выполнится, возвращая управление по тому адресу, который был сохранен в стек. Если отладчик в памяти отсутствует, а процесс исполняется в контексте операционной системы без посредничества отладчика, произойдет исключение, тип которого может варьироваться в зависимости от способа размещения памяти, которая находится сразу после рассматриваемого кода.

В случае если память является обычной виртуальной областью, будет сгенерировано исключение «Ошибка доступа к памяти» (Access violation). Программа будет завершена, однако если обработчик исключений установлен, ошибка может быть обработана. В случае если обращения к виртуальной памяти не произошло, выполнение инструкции rep будет прекращено. Произойдет следующая последовательность событий: запись инструкции «RET» (ей соответствует размещенный в части регистра AX байт-код), ее выполнение и, соответственно, возврат к ADDRESS_02 (в стеке размещен адрес инструкции, следующей за REP).

Объяснение этому — особенность механизма предварительной выборки. Процессоры Intel младше Pentium при записи в адрес памяти, соответствующий адресу в очереди, не очищали очередь предварительной выборки автоматически. Очередь очищалась лишь тогда, когда вызывалось исключение (exception). Например, в случае с пошаговым исключением, которое используется отладчиками прикладного уровня, очередь автоматически очищалась. Таким образом, на процессорах данного типа, в отсутствии отладчика, выполнялась бы оригинальная машинная инструкция.

Если же отладчик присутствует, очередь будет

ВЫПОЛНЕНИЕ ЭТОГО КОДА ПОД ОТЛАДЧИКОМ ПРИВЕДЕТ К КРАХУ ПРОГРАММЫ!



ИЗМЕНЯЕМ ТОЧКУ ВХОДА ПРОГРАММЫ

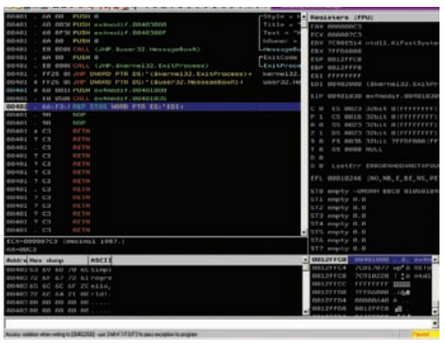
очищаться; соответственно, выполняться будет не оригинальная инструкция, а машинный код, которым она была перезаписана.

Эта особенность была изменена в процессорах Pentium и старше. Несмотря на это, команды MOVS и STOS с префиксом REP продолжают кэшироваться. Следовательно, они выполняются даже в том случае, когда произошла их перезапись. В нашем случае процессор выполняет очистку очереди предварительной выборки и выполняет операцию «RET». Эту особенность можно использовать в антиотладочных целях. Код, написанный нами, будет дробить любые попытки пошаговой отладки приложения.

Будем изучать действие кода на примере программы, рассмотренной выше. Перед внедрением защитного кода изменим точку входа программы на 0x401026, где он и будет размещен (используй LordPE). Откроем программу в OllyDbg и дополним ее кодом:

```
00401026  PUSH 00401000
0040102B  CALL 00401035
00401030  REP STOS WORD PTR ES:[EDI]
00401033  NOP
00401034  NOP
00401035  MOV AL,0C3
00401037  MOV EDI, 00401035
0040103C  MOV ECX, 0FCA
00401041  REP STOS BYTE PTR ES:[EDI]
```

Что произойдет, если программа, измененная подобным образом, выполнится обычным способом (вне отладчика)? Антиотладочный код работоспособен, так как вновь записанная инструкция RET, ведущая к REP STOS WORD PTR ES:[EDI], выполнится только после обнуления регистра ECX, ведь очередь предварительной выборки не будет очищена.



«БРАТ-БЛИЗНЕЦ» OLLYDBG — «ПИТОНОВЫЙ» IMMUNITY

Итак, вот как действует операционная система, выполняя набор инструкций:

1. Помещение в стек адреса 00401000.

2. Вызов кода, размещенного по адресу 00401035, и, соответственно, помещение в стек адреса 00401030. Это естественно, так как в момент вызова автоматически сохраняется адрес инструкции, следующей за командой «call», которая инициировала вызов.

3. Инициализация инструкции цикла «REP STOS» — в регистры будут помещены необходимые данные.

4. Выполнение инструкции «REP STOS» до полной отработки цикла (обнуление ECX).

5. Выполнение инструкции «RET». Инструкция REP STOS WORD PTR ES:[EDI], расположенная по адресу 0x401030, выполнена не будет, так как регистр-счетчик ECX будет содержать нулевое значение.

6. Выполнение возврата по адресу 0x401000, помещенному в стек ранее.

Естественно, все выполнится без заминок. А вот что произойдет, если программу будет отлаживать реверсер:

1. Помещение в стек адреса 0040100.

2. Вызов кода, размещенного по адресу 00401035, и, соответственно, помещение в стек адреса 00401030.

3. Инициализация инструкции цикла «REP STOS» — в регистры будут помещены необходимые данные.

4. Выполнение инструкции «REP STOS» до момента перезаписи кода — очередь предварительной выборки будет очищена.

5. Выполнение инструкции «RET». Инструкция REP STOS WORD PTR ES:[EDI], расположенная по адресу 0x401030, выполнится, так как регистр-счетчик ECX будет содержать значение, отличное от нулевого.

6. После того, как EDI достигнет значения 0x402000, произойдет исключение — ошибка записи в память («Access violation when writing to...»).

Программа уйдет в штопор, что вряд ли обрадует крякера :). Почему выполнение инструкции REP STOS WORD PTR ES:[EDI] (говоря проще, «REP STOSW») при ненулевом значении ECX приводит к краху? Чтобы понять это, разберем механизм действия команды «STOS». «STOre String» — команда, которая помещает по адресу, указанному в регистре EDI, байт, слово или двойное слово (для его указания используется регистр EAX или его части) и автоматически корректирует значение адресного регистра EDI. Команда «STOSB» использует в качестве операнда значение регистра AL, а инструкция «STOSW» — регистра AX.

Значит, при выполнении инструкции REP STOSW будет задействован регистр AX. Следовательно, адресный регистр (EDI) будет корректироваться не на 1 байт, а на 2, что приведет к достижению им критического значения 0x402000. Выполнение программы станет невозможно.

ЕСТЕСТВЕННО ПРЕДПОЛОЖИТЬ, ЧТО ВЫПОЛНЕНИЕ КОДА ДОЛЖНО ОСТАНОВИТЬСЯ СРАЗУ ПОСЛЕ ТОГО, КАК ИНСТРУКЦИЯ REP STOSB БУДЕТ ПЕРЕЗАПИСАНА.

«СОБИРАЕМ» ФАЙЛ ПРОГРАММЫ ПРИ ПОМОЩИ MASM32