**Ida Pro**

## 1. program3

## 2. c program

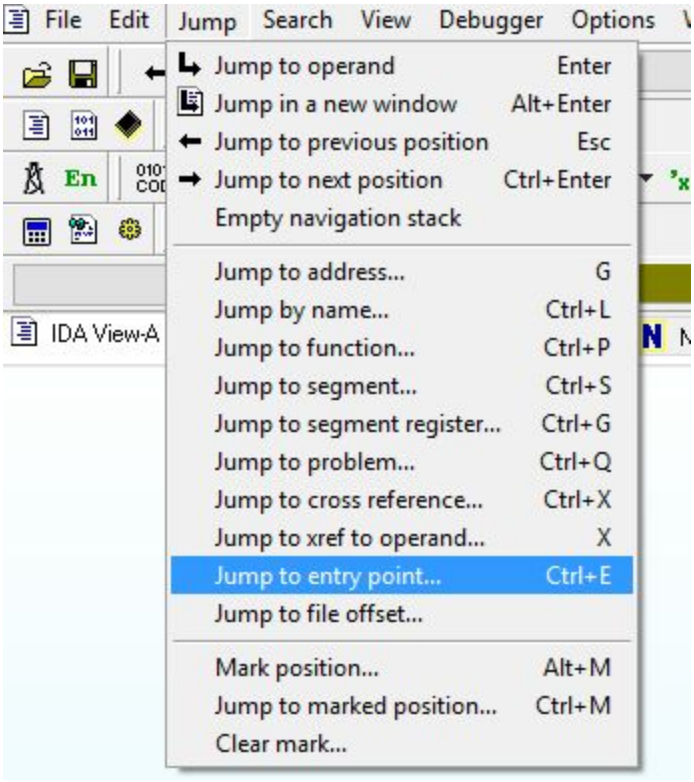
The c program does the exact same thing as program3, except that it is written in c, so it will look different compiled.

**Program 3 thorough explanation:**

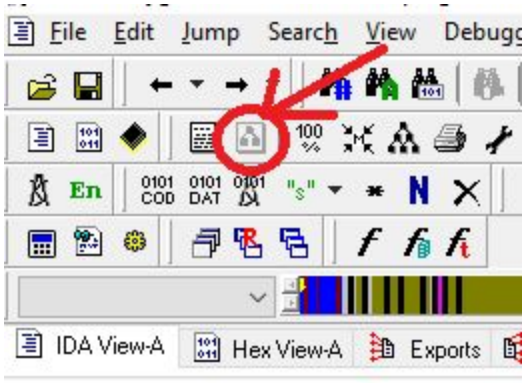
Here is the source code, for reference purposes.

```
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
using namespace std;
int main() {
	string m;
	char c;

	ifstream f("example");
	getline (f,m);
	char k = 's';
	f.close();
	for (int t = 0; t < m.size(); t++)
		m[t] ^= k;

	string o = "bitsadmin.exe /transfer 'JobName' ";
	o.append(m);
	o.append(" C:/Users/student/Desktop/code.txt");

	cout << o;
	system(o.c_str());
	cin >> c;

	return 0;
}
```

1) Go to entry point.

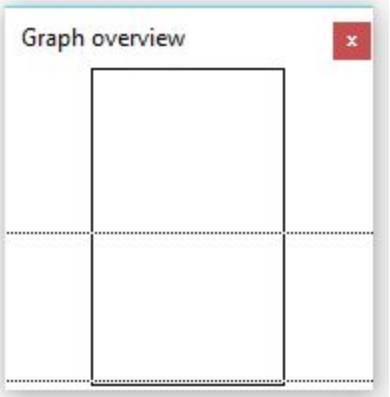



You can change to graph view (the blue sections in the bar at the top) or you can also see the plain assembly code from top to bottom. You will spend most of the time in IDA-View-A.



2) Go to the main function by double clicking on the first function that will run. In this case, double click sub_4011D0. Sub refers to the user defined functions, so there is only one function in this program. The null function calls are used by the compiler for garbage collection, debug functions, etc.

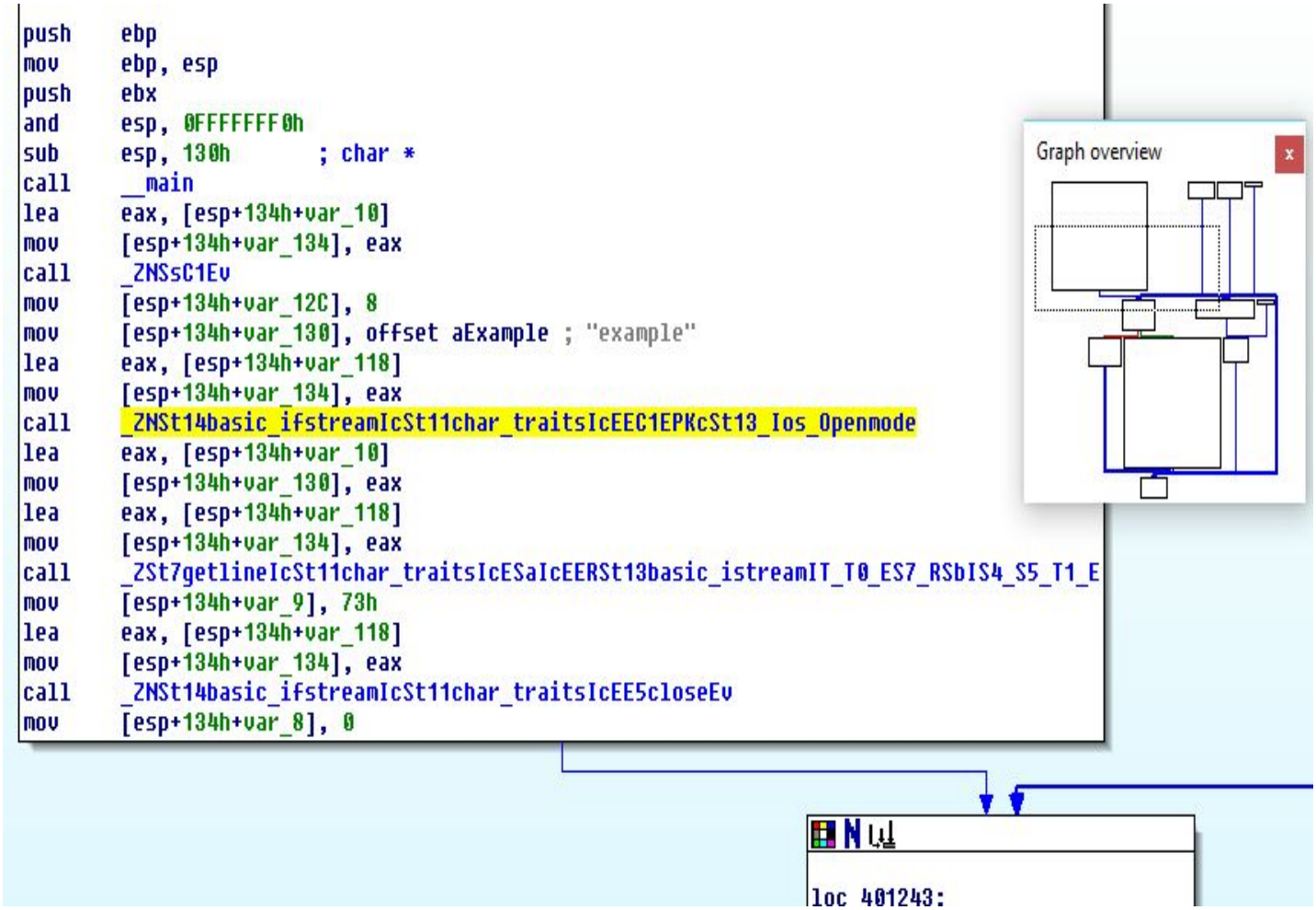
```
public start
start proc near

var_18= dword ptr -18h
var_14= dword ptr -14h
var_10= dword ptr -10h

push    ebp
mov     ebp, esp
sub     esp, 18h
and     esp, 0FFFFFFF0h
mov     [esp+18h+var_18], offset sub_4011D0
call    sub_4014E0
mov     [esp+18h+var_10], 0
mov     [esp+18h+var_14], 0
mov     [esp+18h+var_18], 0
call    nullsub_1
mov     [esp+18h+var_10], 0
mov     [esp+18h+var_14], 0
mov     [esp+18h+var_18], 0
call    nullsub_2
mov     [esp+18h+var_10], 0
mov     [esp+18h+var_14], 0
mov     [esp+18h+var_18], 0
call    nullsub_3
mov     [esp+18h+var_10], 0
mov     [esp+18h+var_14], 0
mov     [esp+18h+var_18], 0
call    nullsub_4
leave
retn
start endp
```



3) Once in main, you can see the allocation for a char pointer, and the string m which we can see is var_10. The first function call is a generic function by the compiler to declare the string.

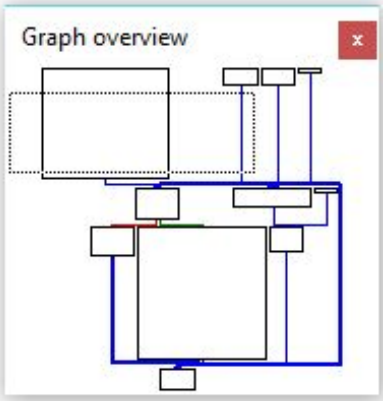
The second call is basic ifstream. A quick google search (http://en.cppreference.com/w/cpp/io/basic_ifstream) tells us this:

1. This is a cpp generated file and now we know the language this was written in.
2. This program is opening a file, since the string “example” was loaded into the eax, this is the file name.
3. It gets exactly 1 line from the file, then closes it. (the getline function call).



4) The line gets saved to var_10, a string.

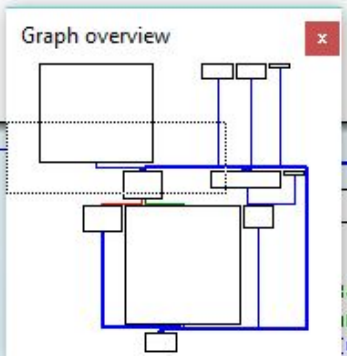
```
var_12C= dword ptr -12Ch
var_11C= dword ptr -11Ch
var_118= dword ptr -118h
var_10= dword ptr -10h
var_9= byte ptr -9
var_8= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
push    ebx
and     esp, 0FFFFFFF0h
sub     esp, 130h       ; char *
call    __main
lea     eax, [esp+134h+var_10]
mov     [esp+134h+var_134], eax
call    _ZNSsC1Ev
mov     [esp+134h+var_12C], 8
mov     [esp+134h+var_130], offset aExample ; "example"
lea     eax, [esp+134h+var_118]
mov     [esp+134h+var_134], eax
call    _ZNSt14basic_ifstreamIcSt11char_traitsIcEEC1EPKcSt13_Ios_Openmode
lea     eax, [esp+134h+var_10]
mov     [esp+134h+var_130], eax
lea     eax, [esp+134h+var_118]
mov     [esp+134h+var_134], eax
call    _ZSt7getlineIcSt11char_traitsIcESaIcEERSt13basic_istreamIT_T0_ES7_RSbIS4_S5
mov     [esp+134h+var_9], 73h
lea     eax, [esp+134h+var_118]
mov     [esp+134h+var_134], eax
```




5) After the getline function, the hex value 73h was moved to var_9. 73h is the letter s.

After the file was closed, a variable var_8 (an int) was set to 0. It was compared over and over (when following the arrows in the graph), we can safely assume this is a while or for loop. (See the add 1 to var_8).

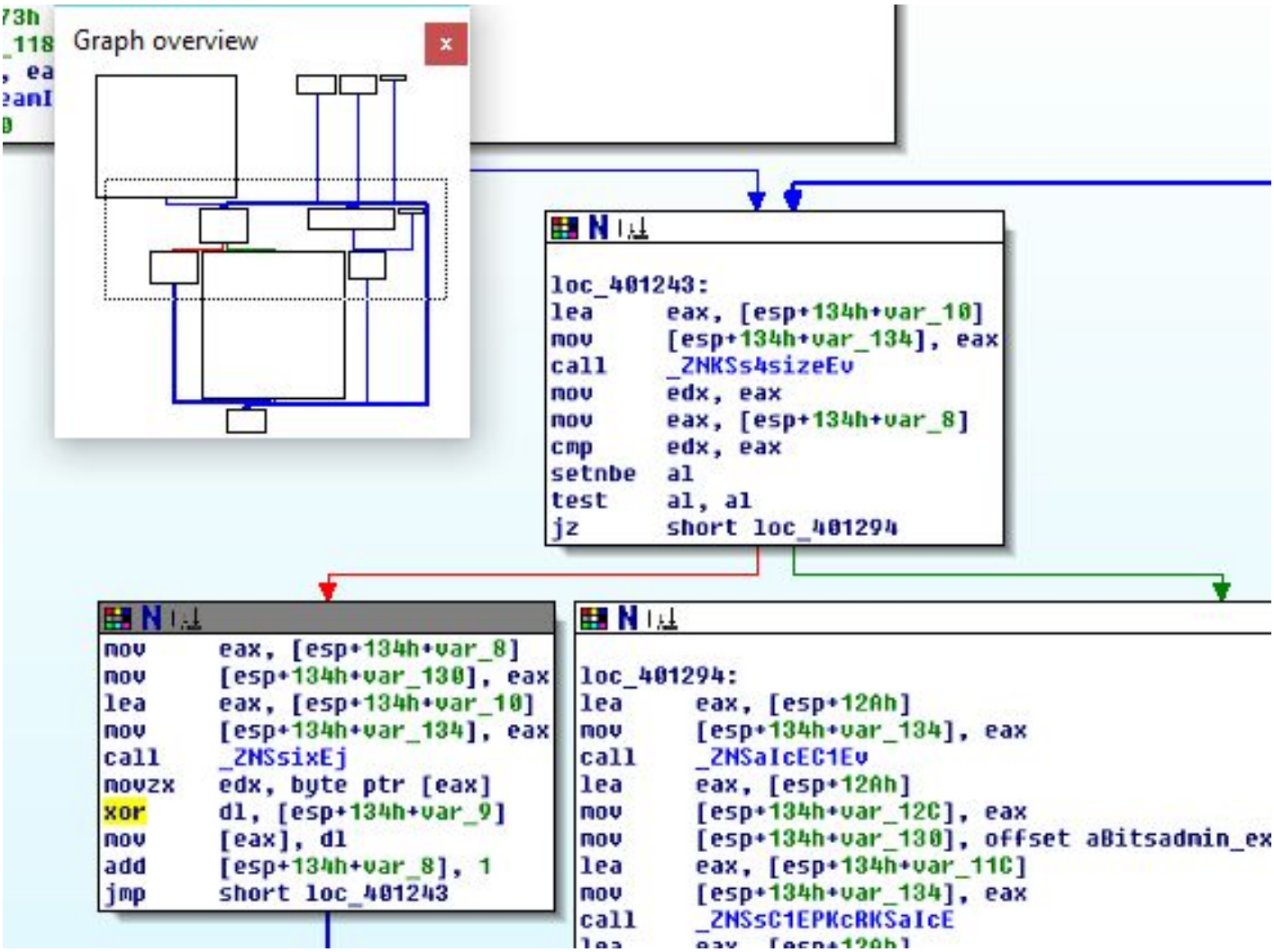
```
call    _ZSt7getlineIcSt11char_traitsIcESaIcEERSt13basic_istreamIT_T0_ES7_RSbIS4_S5_T1_E
mov     [esp+134h+var_9], 73h
lea     eax, [esp+134h+var_118]
mov     [esp+134h+var_134], eax
call    _ZNSt14basic_ifstreamIcSt11char_traitsIcEE5closeEv
mov     [esp+134h+var_8], 0
```



6) The loop takes the string's size, compares it with var_8 (look closely at the function call, you can see it is size). Red arrows mean failure, so it loops again. Green means success.

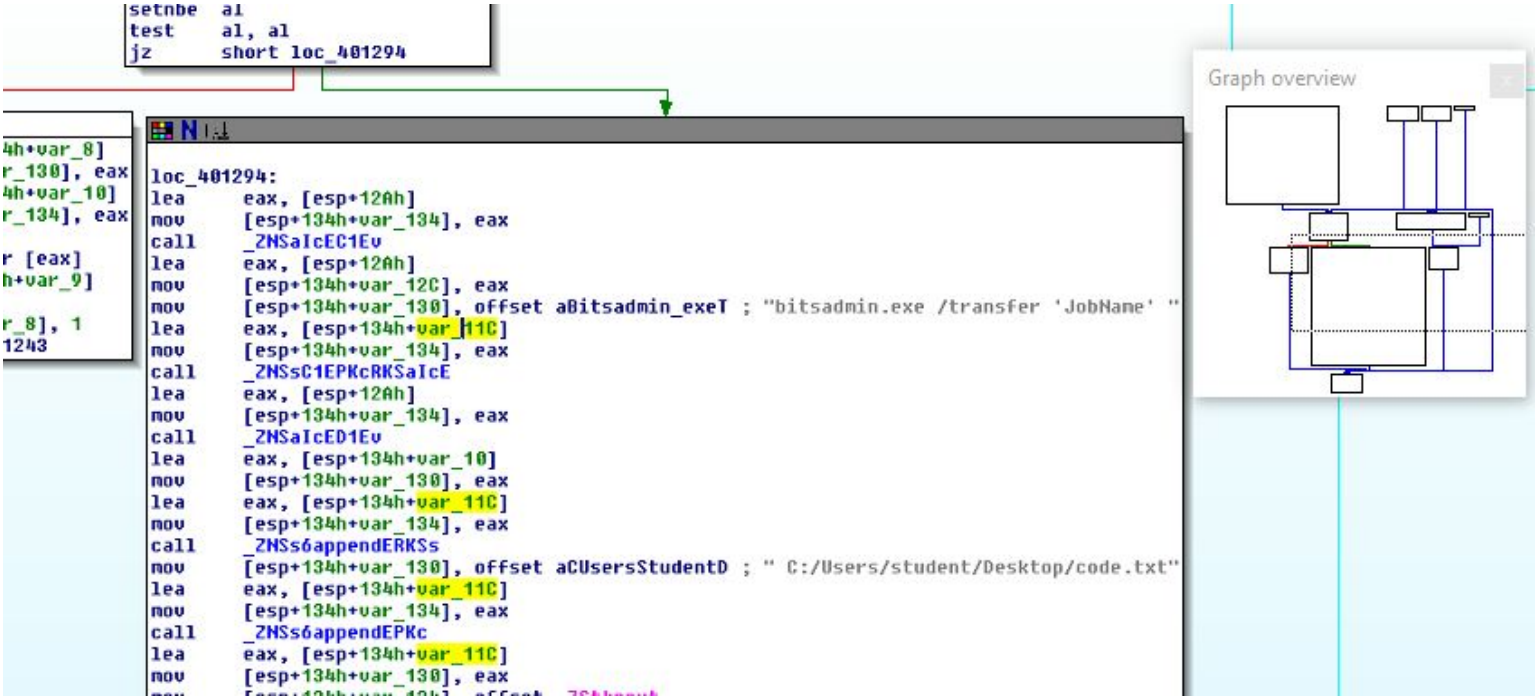
If you follow the red path, you will see var_10, the string, being xor'ed with the letter s, byte by byte (byte pointer),

Following the blue arrow after that leads back to the loop.

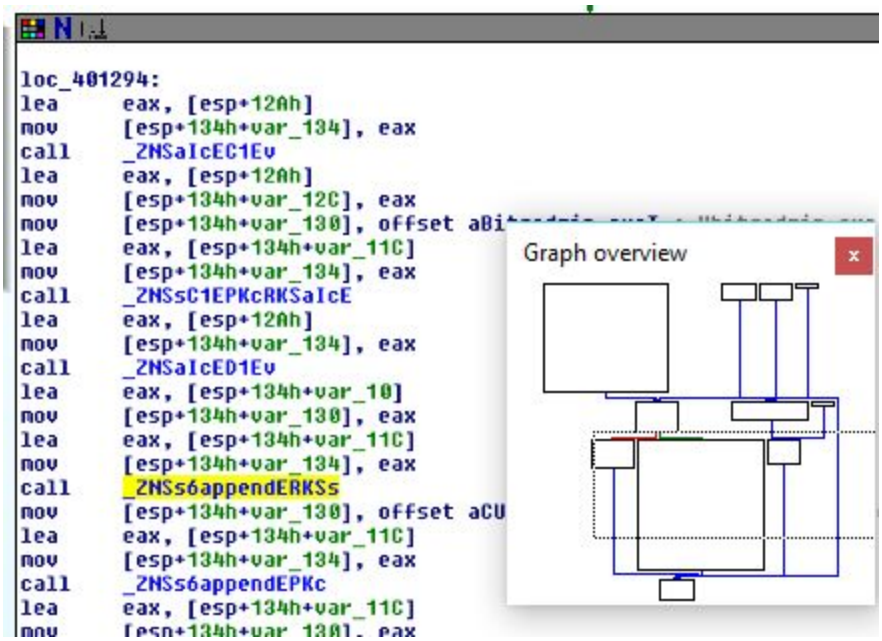


7) Green path, you can see lots of things happen:

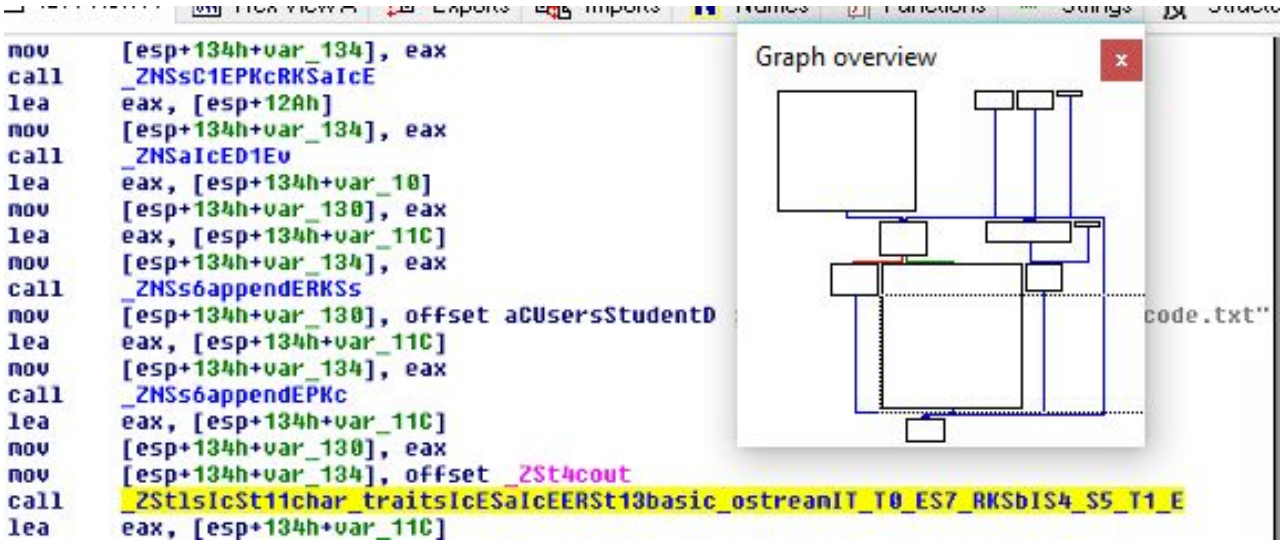
1. A long string (var_11C) containing the command “bitsadmin.exe” (https://msdn.microsoft.com/en-us/library/aa362813(v=vs.85).aspx)
2. The var_10 string was appended to it.
3. Another string with a “location” was appended as well.



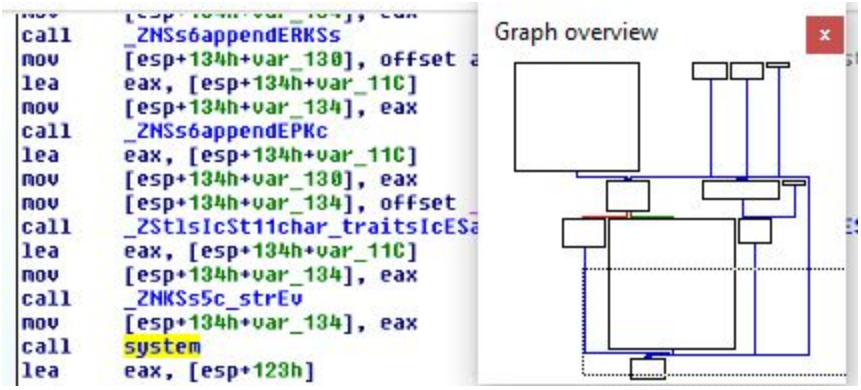

You can see the 2 append functions:



8) The full string was printed (cout). Then it was converted into a pointer of the string:

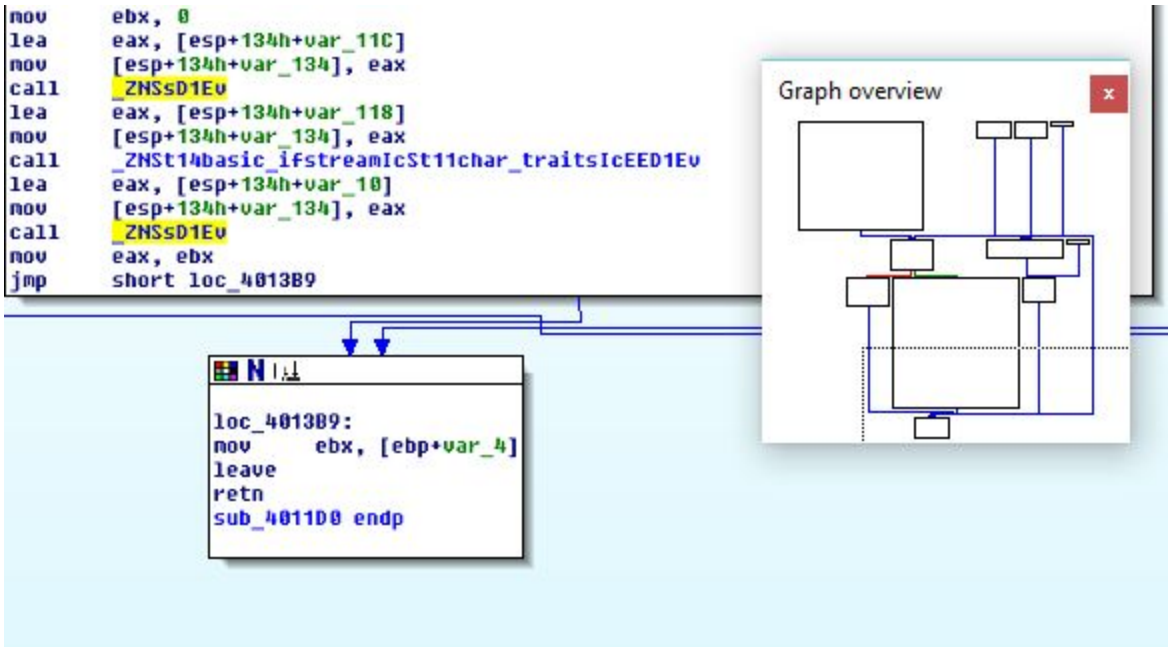


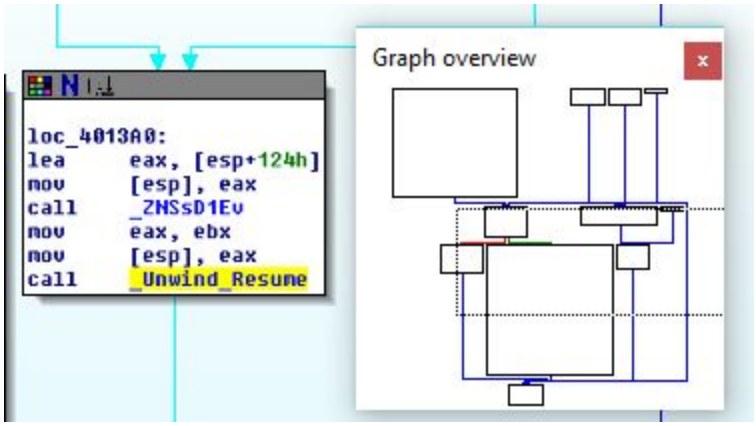

This string (var_11C)'s pointer was placed into a system call as a parameter,



9) After that the program pauses until user input (cin).

The program does nothing with the input, does memory clean up (unwind resume).

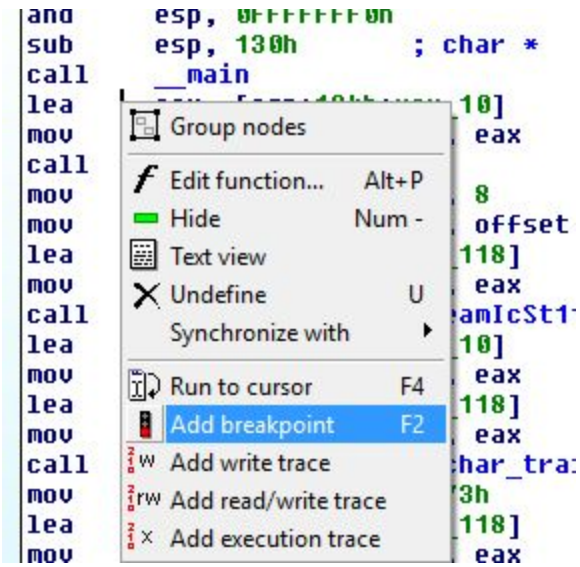


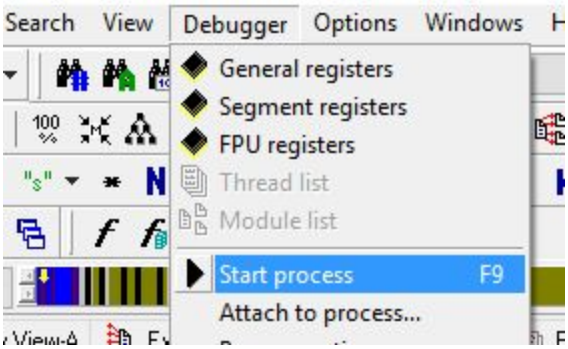

The program returns, runs the destructor functions, and exits.

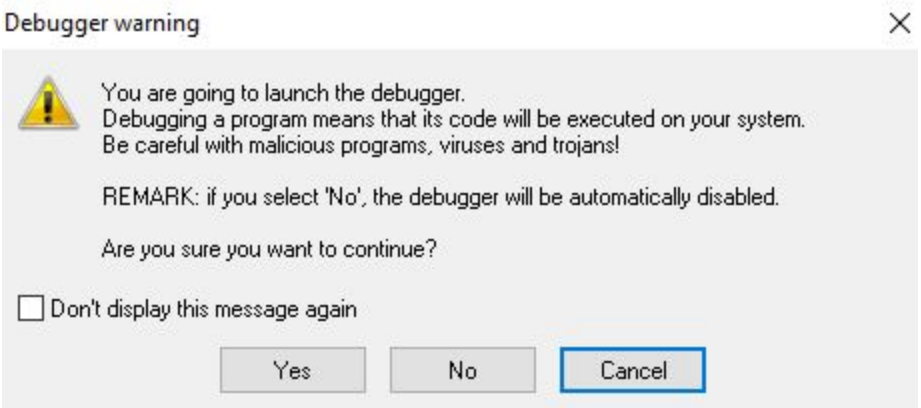
## Tracing with debugger:

Add a breakpoint right when main runs

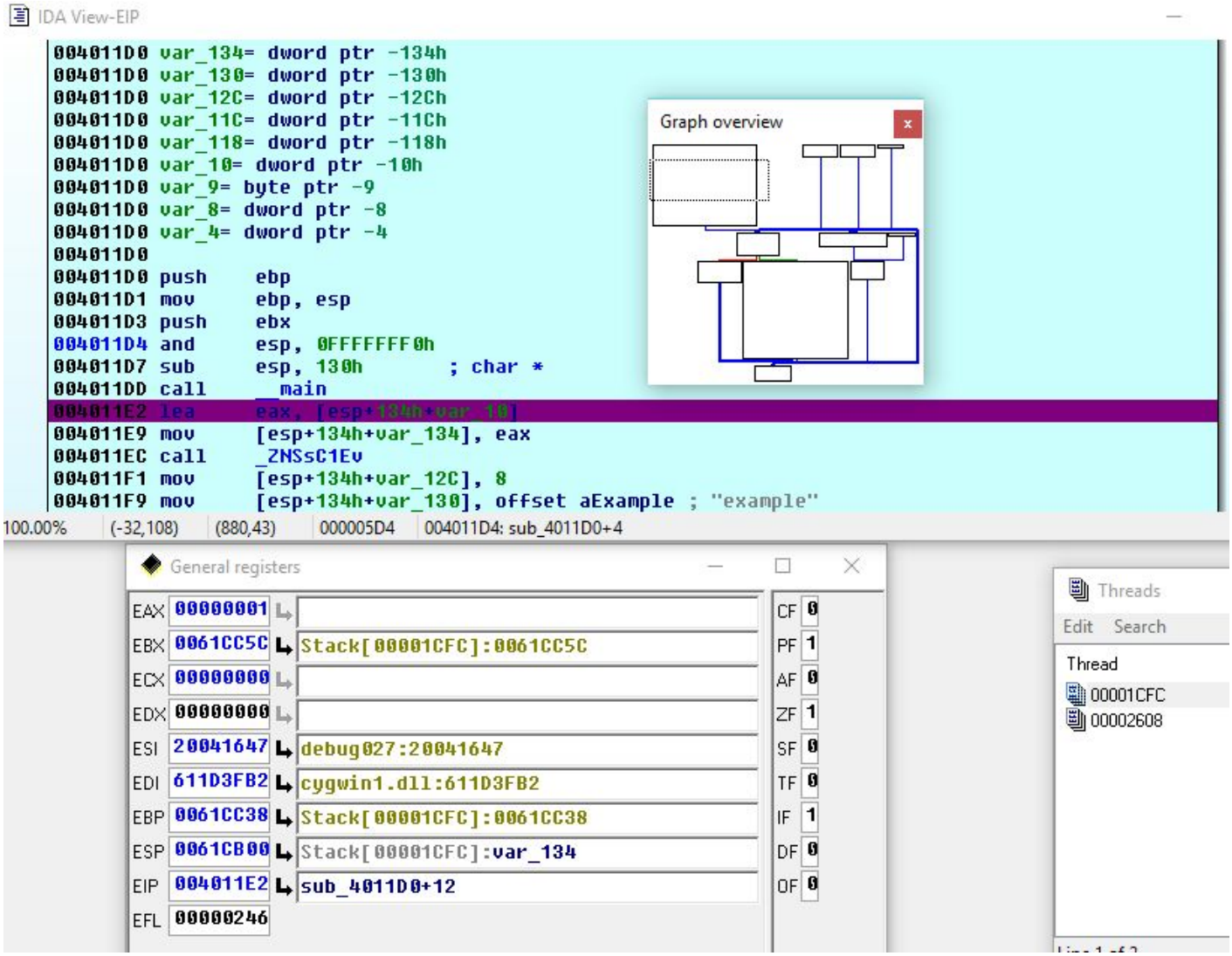



Go to start process and you will warnings, click yes and ok.

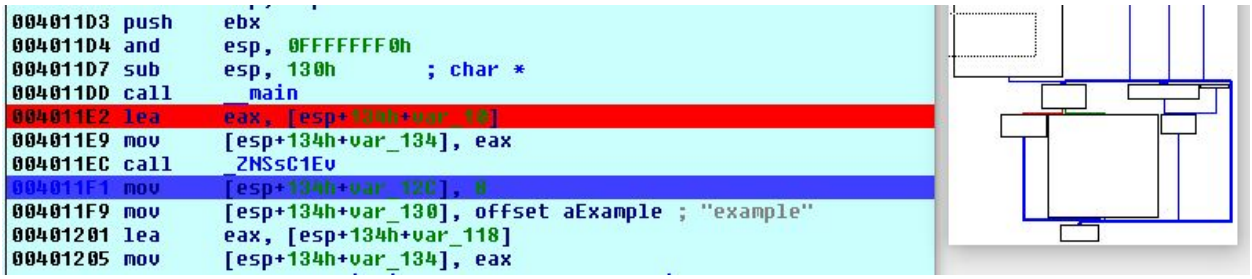


If you step into (F7) you can see the register values and a flow chart view of the code, you can disable the graph view by pressing spacebar.



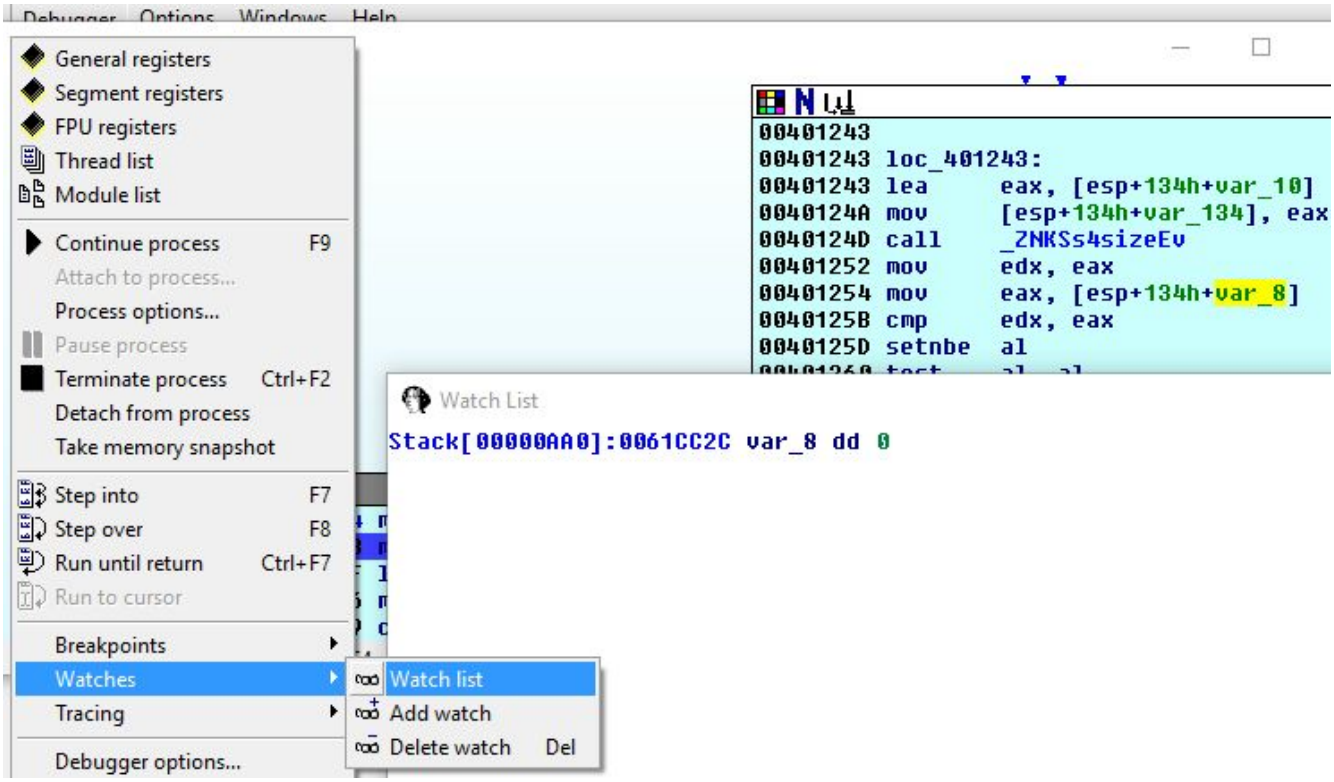
Make sure you press F8 to step over function calls, or else you will go into the functions the compiler did to add strings together, or open files, etc.




You can add specific variables to the watch list and see how they change as the program runs. The variable var_8 was the counter for a loop, and you can see it increase by 1 each time.

var_10 was a string that was being xor'ed, you can see it change byte by byte.

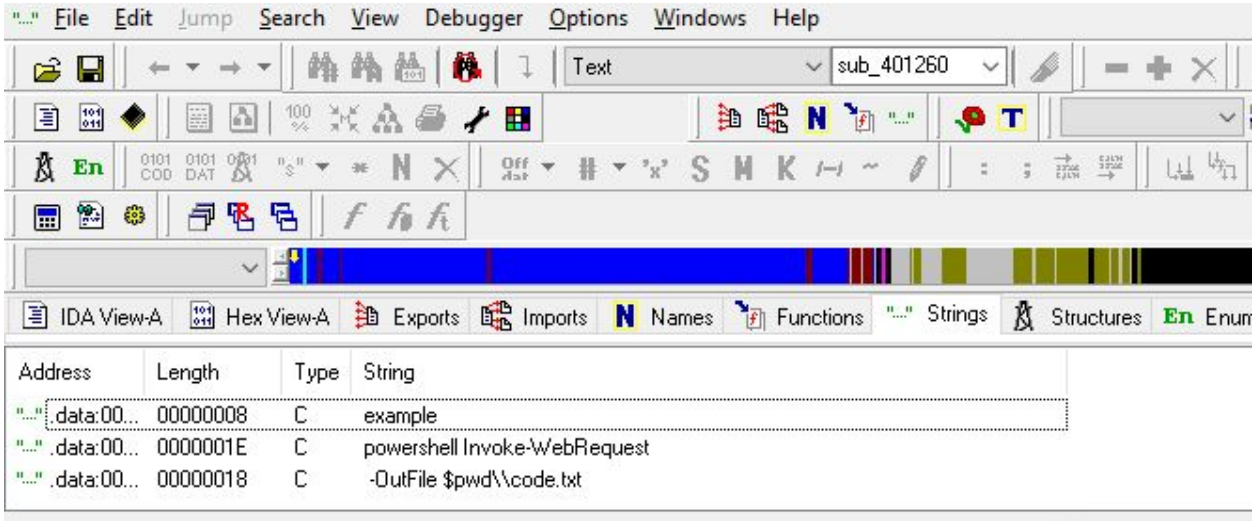
var_11C was the string that was being appended, and you can see the values change as well.



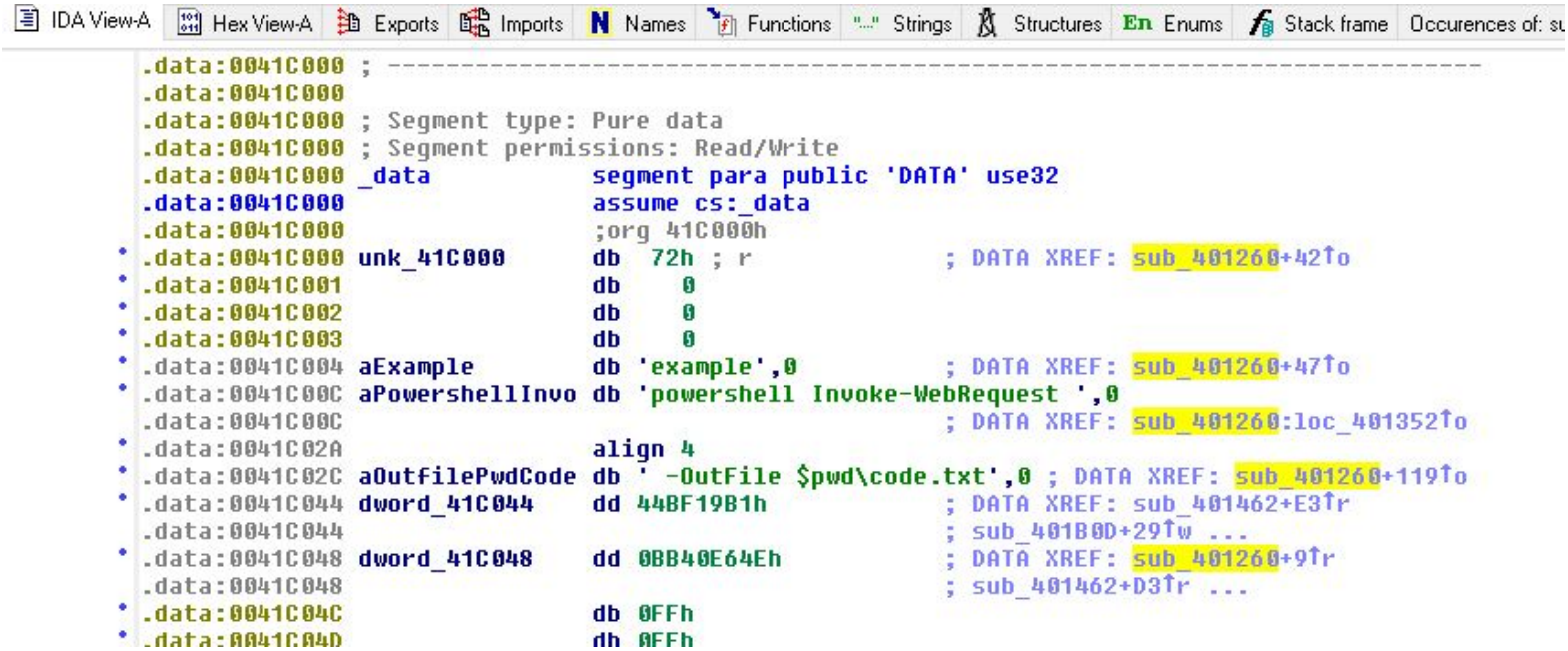
## C program

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

int main(char *argc, char* argv[]){
	char buffer[200];
	buffer[0] = '\0';

	FILE* file = fopen("example", "r");
	char line[46];
	fgets(line, sizeof(line), file);
	line[45]='\0';

	char k = 's';
	fclose(file);

	int t;
	for (t = 0; t < sizeof(line)-1; t++){
		line[t] ^= k;
	}

	strcpy(buffer,"powershell Invoke-WebRequest ");
	strcat(buffer,line);

	strcat(buffer," -OutFile $pwd\\code.txt");

	system(buffer);

	printf(buffer);
	getch();

	return 0;
}
```

The first thing you should do is investigate the strings tab:




You can see where strings are used by double clicking:



Double clicking sub_401260 will take you to the function where the strings are being used.

You can also see sub_401260 being called when you “jump to entry point”.

Between hModules because it is using a Windows system call to call a powershell.

```
loc_4017DB:                             ; hModule
push    edi
call    sub_402079
call    sub_405207
mov     edi, eax
call    sub_405201
mov     esi, eax
call    sub_404D6D
push    eax
push    dword ptr [edi]
push    dword ptr [esi]
call    sub_401260
mov     esi, eax
push    0                               ; hModule
call    sub_402115
```

It is the same as program3, in this case, the counter for the loop is var_104.