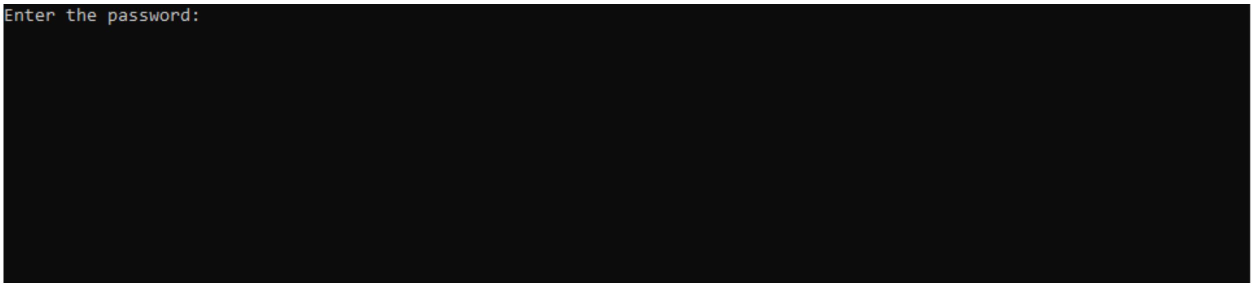
https://www.crackmes.one/crackme/674358839b533b4c22bd3159

Zenton's crackme for beginners:

For this one I will be using IDA.

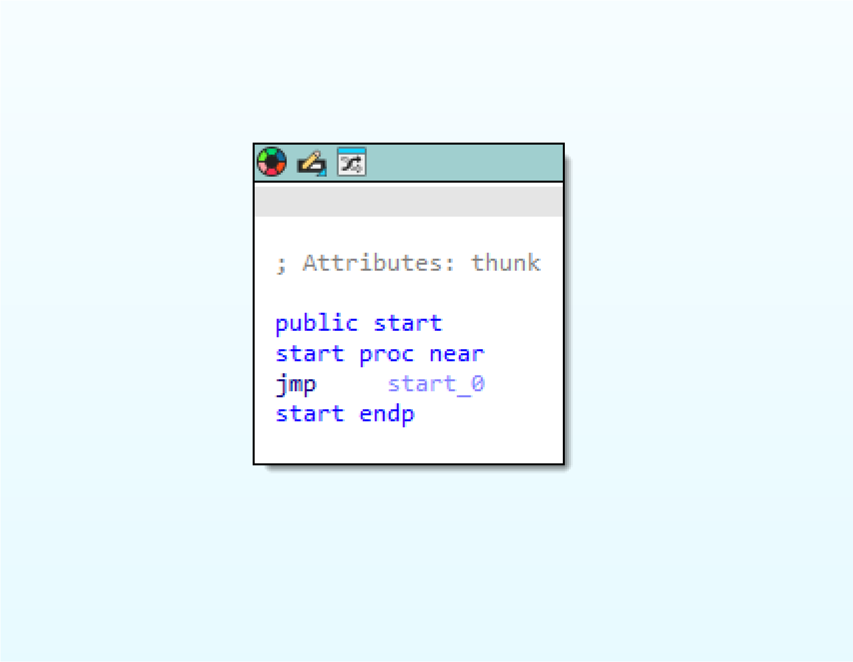
First thing to do is to run the program.



```
Enter the password:
```

This is all we get. But this gives us a string to work with.

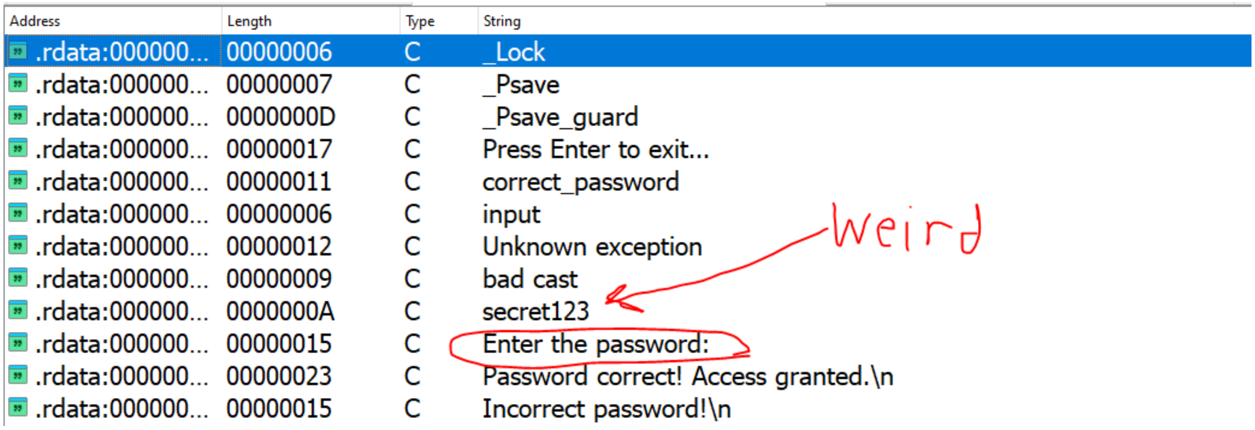
In IDA, this is what we start with:



```
; Attributes: thunk

public start
start proc near
jmp     start_0
start endp
```

This does not help us much, but a good starting point is to look for strings. By opening up the string subview in IDA, it provides us with a list of strings present in the program.

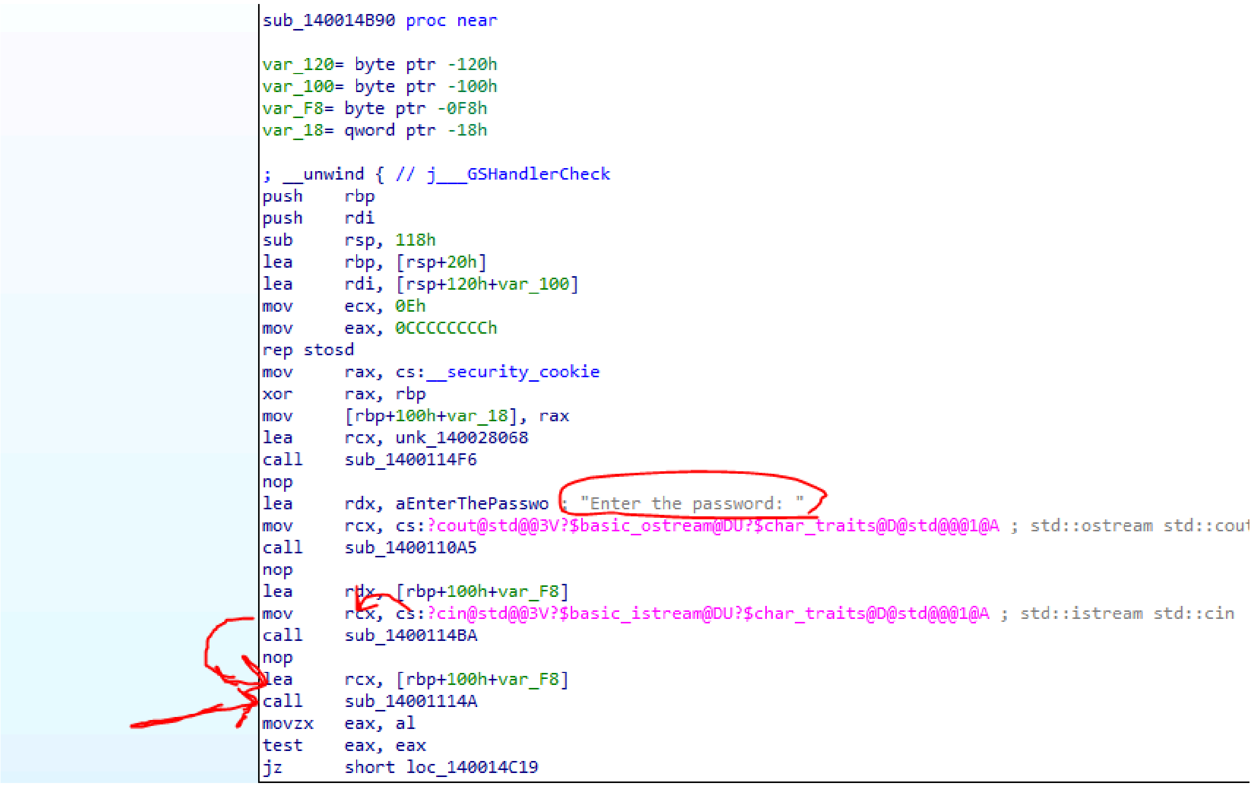


Address	Length	Type	String
.rdata:000000...	00000006	C	_Lock
.rdata:000000...	00000007	C	_Psave
.rdata:000000...	0000000D	C	_Psave_guard
.rdata:000000...	00000017	C	Press Enter to exit...
.rdata:000000...	00000011	C	correct_password
.rdata:000000...	00000006	C	input
.rdata:000000...	00000012	C	Unknown exception
.rdata:000000...	00000009	C	bad cast
.rdata:000000...	0000000A	C	secret123
.rdata:000000...	00000015	C	Enter the password:
.rdata:000000...	00000023	C	Password correct! Access granted.\n
.rdata:000000...	00000015	C	Incorrect password!\n

Here, we can see the string we knew about, “Enter the password,” But we also see something that kind of stands out,”secret123.”

Compared to the other strings around it, this is weird. We can check this as the password, but let's go a little deeper to confirm it.

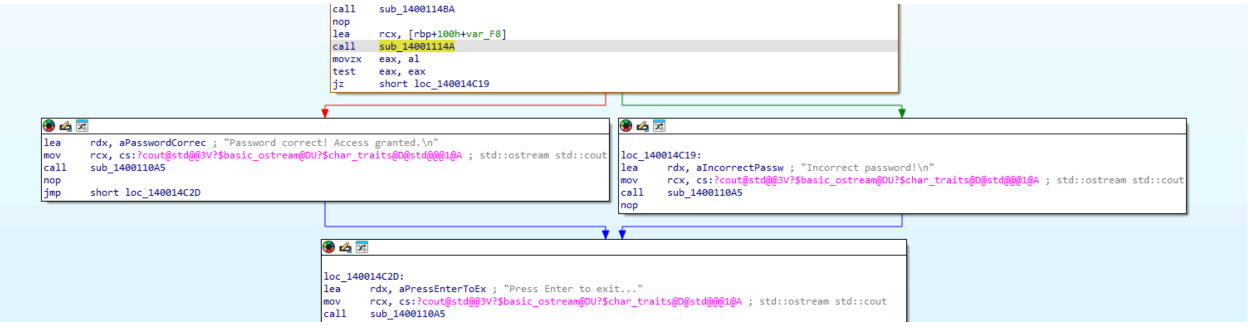
By navigating to where the Enter the password string shows up in the assembly, we can see most of the assembly for the program, outside of packages and imports.



```
sub_140014B90 proc near

var_120= byte ptr -120h
var_100= byte ptr -100h
var_F8= byte ptr -0F8h
var_18= qword ptr -18h

; __unwind { // j___GSHandlerCheck
push    rbp
push    rdi
sub     rsp, 118h
lea     rbp, [rsp+20h]
lea     rdi, [rsp+120h+var_100]
mov     ecx, 0Eh
mov     eax, 0CCCCCCCCh
rep stosd
mov     rax, cs:__security_cookie
xor     rax, rbp
mov     [rbp+100h+var_18], rax
lea     rcx, unk_140028068
call    sub_1400114F6
nop
lea     rdx, aEnterThePasswo ; "Enter the password: "
mov     rcx, cs:?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::ostream std::cou
call    sub_1400110A5
nop
lea     rdx, [rbp+100h+var_F8]
mov     rcx, cs:?cin@std@@3V?$basic_istream@DU?$char_traits@D@std@@@1@A ; std::istream std::cin
call    sub_1400114BA
nop
lea     rcx, [rbp+100h+var_F8]
call    sub_14001114A
movzx   eax, al
test    eax, eax
jz      short loc_140014C19
```



```
call    sub_1400114BA
nop
lea     rcx, [rbp+100h+var_F8]
call    sub_14001114A
movzx   eax, al
test    eax, eax
jz      short loc_140014C19
```

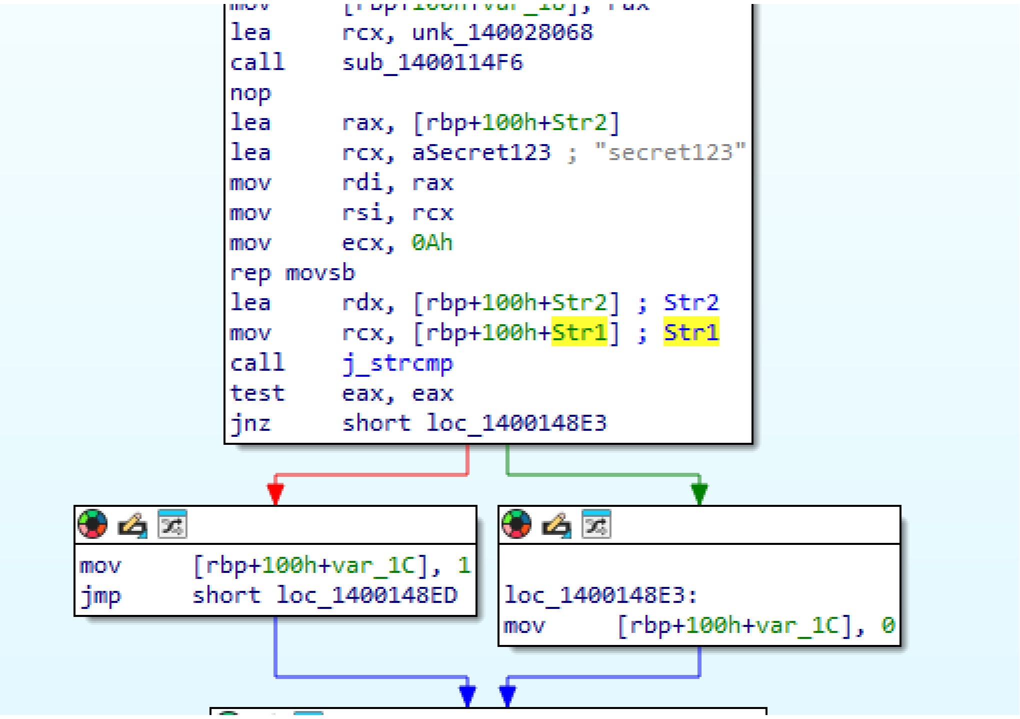
```
lea     rdx, aPasswordCorrec ; "Password correct! Access granted.\n"
mov     rcx, cs:?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::ostream std::cout
call    sub_1400110A5
nop
jmp     short loc_140014C2D
```

```
loc_140014C19:
lea     rdx, aIncorrectPassw ; "Incorrect password!\n"
mov     rcx, cs:?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::ostream std::cout
call    sub_1400110A5
nop
```

```
loc_140014C2D:
lea     rdx, aPressEnterToEx ; "Press Enter to exit..."
mov     rcx, cs:?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::ostream std::cout
call    sub_1400110A5
```

Here, we see the string we’re looking for. After this string we can see that it's calling sub_1400110A5, which through deducing, based on the line above it, that's our cout function, printing the string. Then by the same logic, sub1400111BA is our cin function, scanning for input. Then we have a final call, sub14001114A. We don’t know what this is yet, but after this call, we get two possible outcomes. Incorrect or correct. We see that above, the sub14001114A call, we have lea rcx, [rbp+100h+var_F8]. This is what is being passed into the sub14001114A call. Based on that, and the deduction that after this function the outcomes split, that is our comparison function.

Clicking on it, we see that it jumps to another subprocess, sub_140014860, which leads us here:

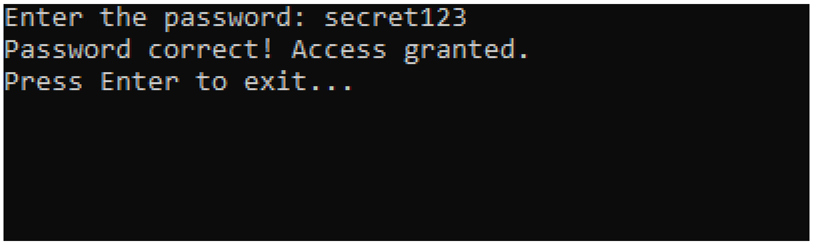


```
lea      rcx, unk_140028068
call     sub_1400114F6
nop
lea      rax, [rbp+100h+Str2]
lea      rcx, aSecret123 ; "secret123"
mov      rdi, rax
mov      rsi, rcx
mov      ecx, 0Ah
rep movsb
lea      rdx, [rbp+100h+Str2] ; Str2
mov      rcx, [rbp+100h+Str1] ; Str1
call     j_strcmp
test     eax, eax
jnz      short loc_1400148E3
```

```
mov      [rbp+100h+var_1C], 1
jmp      short loc_1400148ED
```

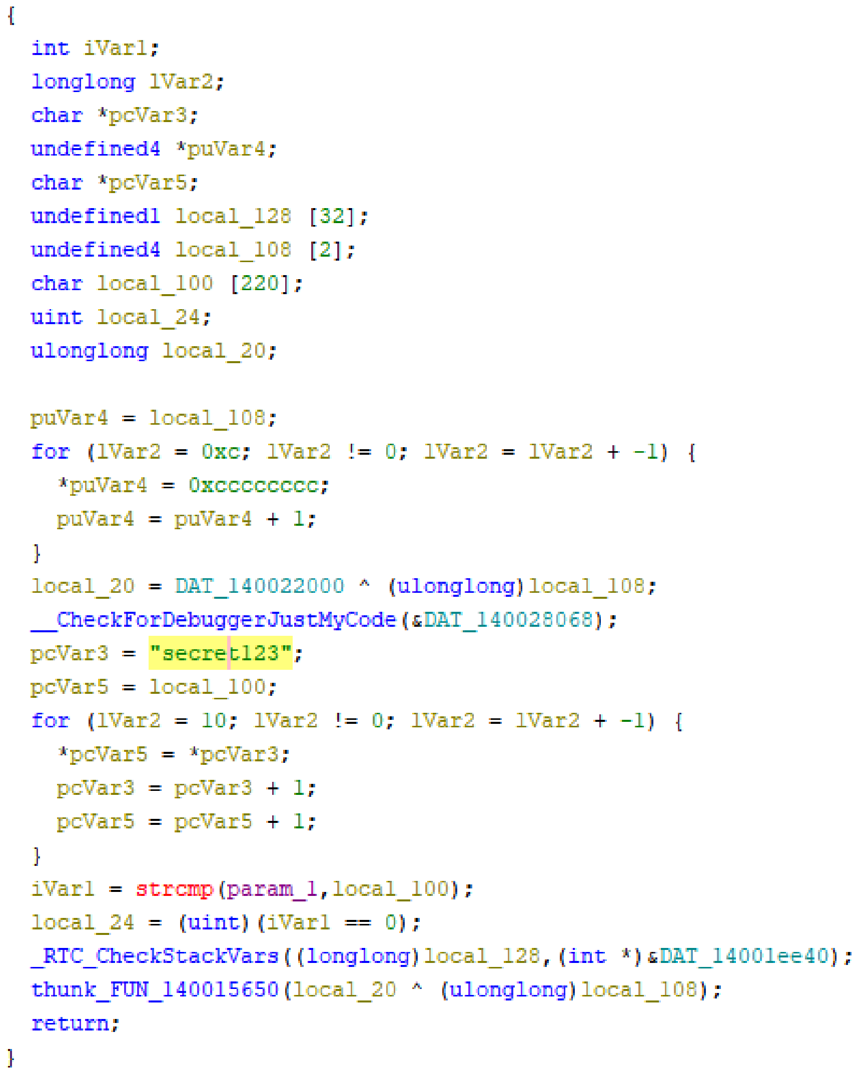
```
loc_1400148E3:
mov      [rbp+100h+var_1C], 0
```

Looking at this we see our “secret123” string, which is put in the rcx register, which holds a value, in this case, our string. Which is then moved down to above the call of “j_strcmp,” which takes two strings as parameters, and passes either a 1 or 0, true or false. So we can conclude that the function sub14001114A, can return a true or false value, and that our comparison is between str2, which is our input, the rdx register, and str1, “secret123.”
Meaning that we were right, secret123 is our password.



```
Enter the password: secret123
Password correct! Access granted.
Press Enter to exit...
```

But, for as simple as this program seems to be, I think this is entirely within my capabilities to reconstruct this back into code. So lets use the ghidra decompiler. When we go to where secret123 shows up in the code we get this:

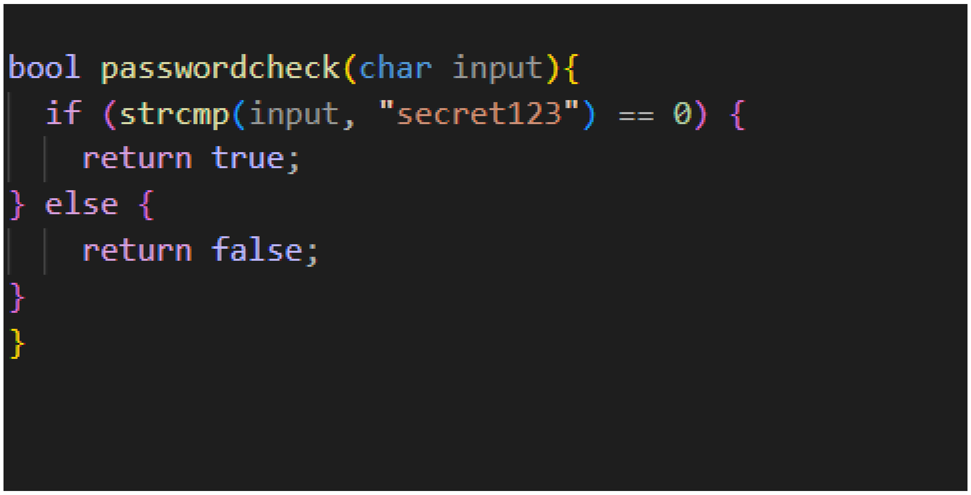


```
{
  int iVar1;
  longlong lVar2;
  char *pcVar3;
  undefined4 *puVar4;
  char *pcVar5;
  undefined1 local_128 [32];
  undefined4 local_108 [2];
  char local_100 [220];
  uint local_24;
  ulonglong local_20;

  puVar4 = local_108;
  for (lVar2 = 0xc; lVar2 != 0; lVar2 = lVar2 + -1) {
    *puVar4 = 0xcccccccc;
    puVar4 = puVar4 + 1;
  }
  local_20 = DAT_140022000 ^ (ulonglong)local_108;
  __CheckForDebuggerJustMyCode(&DAT_140028068);
  pcVar3 = "secret123";
  pcVar5 = local_100;
  for (lVar2 = 10; lVar2 != 0; lVar2 = lVar2 + -1) {
    *pcVar5 = *pcVar3;
    pcVar3 = pcVar3 + 1;
    pcVar5 = pcVar5 + 1;
  }
  iVar1 = strcmp(param_1,local_100);
  local_24 = (uint)(iVar1 == 0);
  _RTC_CheckStackVars((longlong)local_128,(int *)&DAT_14001ee40);
  thunk_FUN_140015650(local_20 ^ (ulonglong)local_108);
  return;
}
```

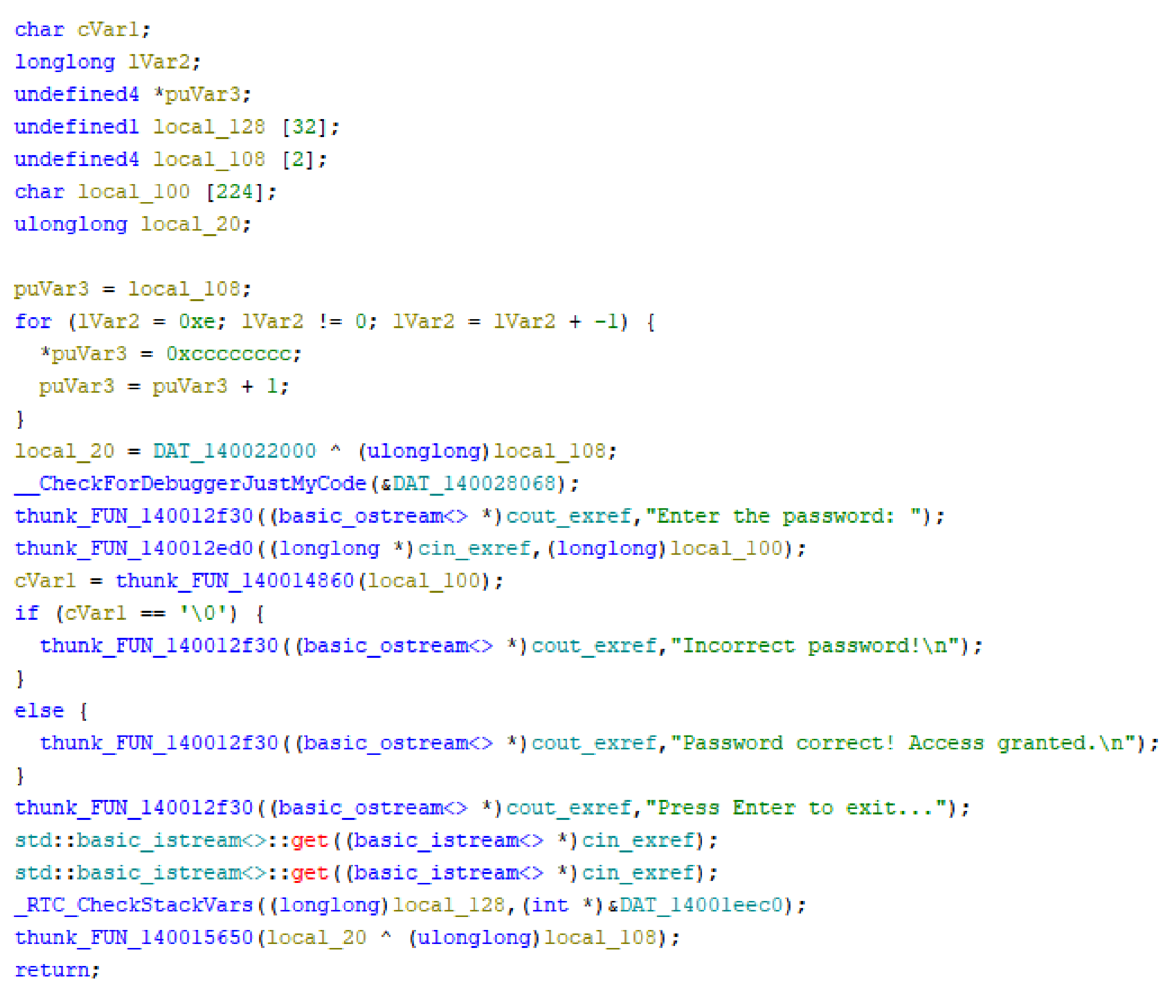
This shows us one, that there is a debugger checker, and that this function is a simple comparison.This also shows that this isn't the main function, so the comparison function is called within main.

So taking some liberties here and there we can come up with a reasonable function:



```
bool passwordcheck(char input){
  if (strcmp(input, "secret123") == 0) {
    return true;
} else {
    return false;
}
}
```

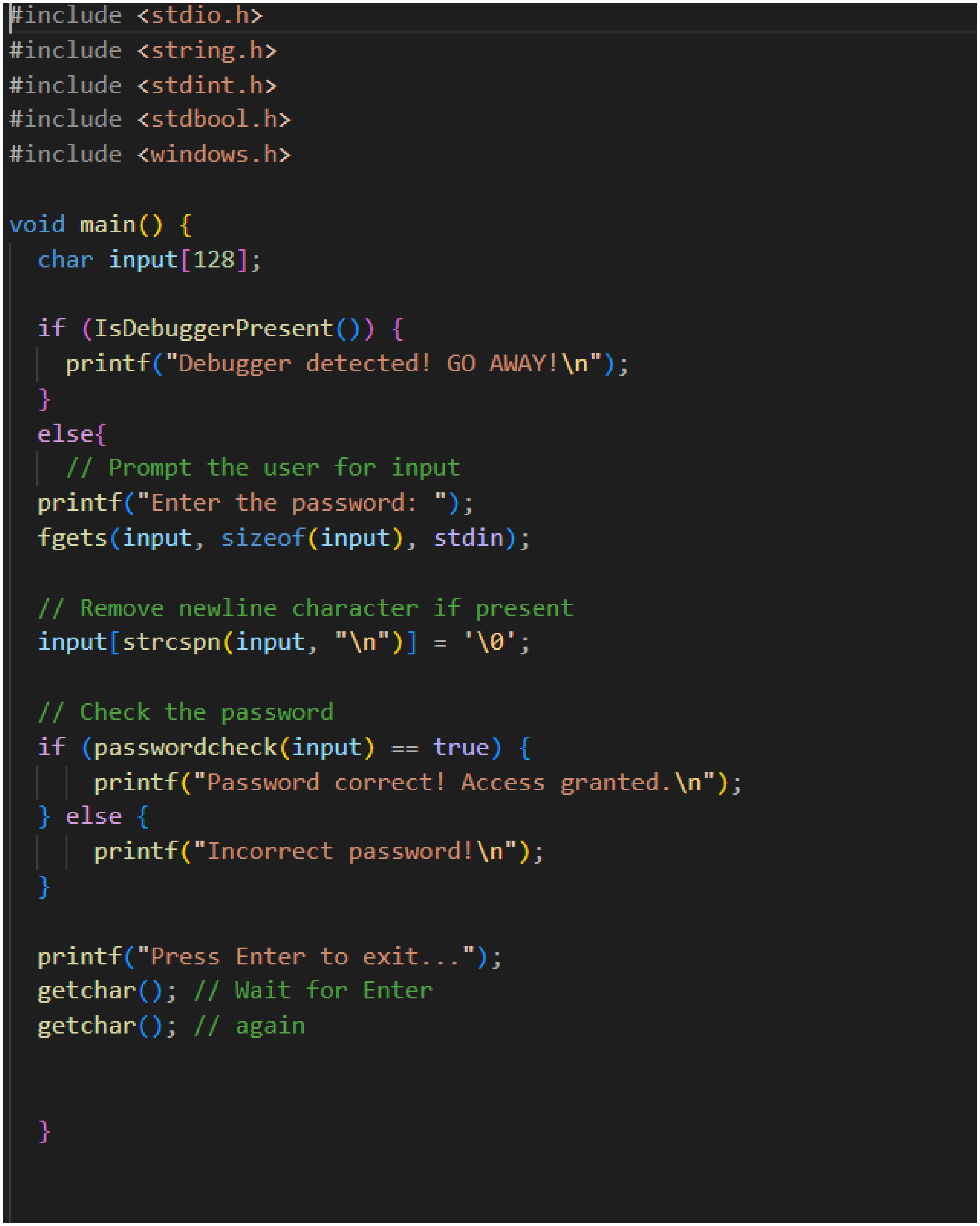
Next the main function:



```
char cVar1;
longlong lVar2;
undefined4 *puVar3;
undefined1 local_128 [32];
undefined4 local_108 [2];
char local_100 [224];
ulonglong local_20;

puVar3 = local_108;
for (lVar2 = 0xe; lVar2 != 0; lVar2 = lVar2 + -1) {
  *puVar3 = 0xcccccccc;
  puVar3 = puVar3 + 1;
}
local_20 = DAT_140022000 ^ (ulonglong)local_108;
__CheckForDebuggerJustMyCode(&DAT_140028068);
thunk_FUN_140012f30((basic_ostream<> *)cout_exref,"Enter the password: ");
thunk_FUN_140012ed0((longlong *)cin_exref,(longlong)local_100);
cVar1 = thunk_FUN_140014860(local_100);
if (cVar1 == '\0') {
  thunk_FUN_140012f30((basic_ostream<> *)cout_exref,"Incorrect password!\n");
}
else {
  thunk_FUN_140012f30((basic_ostream<> *)cout_exref,"Password correct! Access granted.\n");
}
thunk_FUN_140012f30((basic_ostream<> *)cout_exref,"Press Enter to exit...");
std::basic_istream<>::get((basic_istream<> *)cin_exref);
std::basic_istream<>::get((basic_istream<> *)cin_exref);
_RTC_CheckStackVars((longlong)local_128,(int *)&DAT_14001eec0);
thunk_FUN_140015650(local_20 ^ (ulonglong)local_108);
return;
```

This is essentially just a bunch of output functions, cout, which we know. And some CIN functions for input. SO, we can reasonably make the rest of the program here:



```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <stdbool.h>
#include <windows.h>

void main() {
  char input[128];

  if (IsDebuggerPresent()) {
    printf("Debugger detected! GO AWAY!\n");
  }
  else{
    // Prompt the user for input
  printf("Enter the password: ");
  fgets(input, sizeof(input), stdin);

  // Remove newline character if present
  input[strcspn(input, "\n")] = '\0';

  // Check the password
  if (passwordcheck(input) == true) {
      printf("Password correct! Access granted.\n");
  } else {
      printf("Incorrect password!\n");
  }

  printf("Press Enter to exit...");
  getchar(); // Wait for Enter
  getchar(); // again



  }
```

And by running the recreated executable, we can see it exhibits the same behavior. So maybe not perfect, but close enough!