PA193 - Secure coding principles and practices

LAB: Language level vulnerabilities: Buffer overflow, type overflow, strings

Łukasz Chmielewski *chmiel@fi.muni.cz (email me with your questions/feedback)*
Centre for Research on Cryptography and Security, Masaryk University

CROCS
Centre for Research on Cryptography and Security

PV286+PA193: Secure coding principles and practices

- ~~Main goal: secure coding~~
  - ~~How to write code in a more secure way~~
  - ~~So that the program is harder to be attacked/exploited~~
  - ~~Selected basic building blocks of security applications~~
- ~~PV286: > 80 students~~
  - ~~Lecture: 2 hours weekly~~
  - ~~Project: about 30-40 hours/person~~
- PA193: < 40 students
  - Seminar: 2 hours weekly, usually corresponding to the lecture
  - Homework: about 6-? hours/each
- In case of questions: please email me!

People

- Main contact: Łukasz Chmielewski (CRoCS@FI MU)
  - Office hours: Friday 9:30-11:00, A406


chmiel@fi.muni.cz,


https://keybase.io/grasshoppper

- Other lectures, seminars, and the project
  - Václav Lorenc (HERE Technologies), Marek Sýs (FI), Lukas Rucka (FI), Martin Čarnogurský (RootLUG), and Lumir Honus (AT&T)

Grading PA193

- Grading 60 (max)
  - A ≥ 54
  - B ≥ 48
  - C ≥ 42
  - D ≥ 36
  - E ≥ 30
  - F < 30
  - Z ≥ 30
- Assignments:
  - 6 assignments,10 points each
  - Published usually after the last seminar and you will have 1 week to work on them
  - For the first one today you will have 2 weeks due to the initial registration period

Pre-seminar preparation

- Prepare your IDE for compilation of file bufferoverflow.cpp (IS)
  - Use any suitable C/C++ IDE: Visual Studio, QT Creator, CLion, Eclipse, Xcode…
  - Make sure file compiles and can be debugged (breakpoints, next step…)
  - Locate where Memory debug window is
  - Locate where switch between source code and disassembly is
  - Find where to set compiler switch for stack protection (/GS, –fstack-protector …)

Plan for online seminar

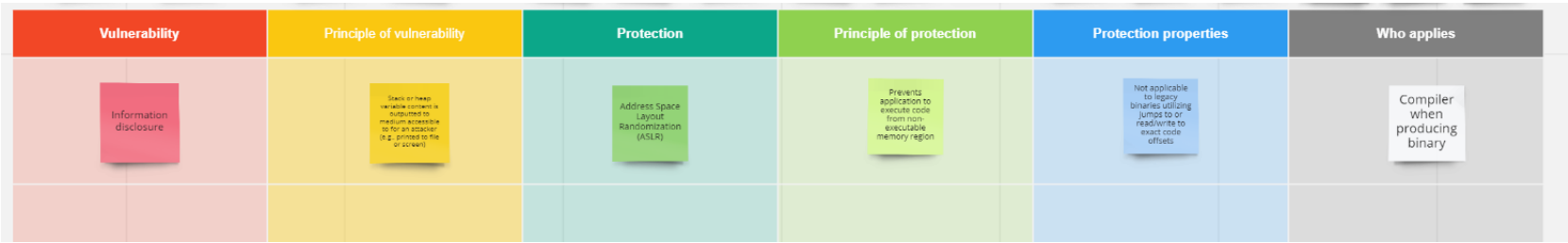
- Group activity – pairing Vulnerabilities and Defenses
- Fun with buffer overflow and memory layouts
- Impact of compilation switches
- Seminar: your IDE
- Assignment: your own environment

Note: password for the various activities

- If you are asked for password (Miro boards…) use ‘fimunicz’

Group activity: Pair topics (30 minutes)

- Groups of 3 students (breakout rooms), Discuss and reason within the group
- Distribute cards to the best match, make Vulnerability – Protection “rows”
- Link for Miro board will be provided in each breakout room
  - (one of you can Share the screen with Miro board for easier group discussion)



1. Pair name of vulnerability and its description
2. Pair name of protection and its description
3. Put protection pair atop of a vulnerability pair where protection is most effective
4. Add properties (adv/disadv/limits) of protection to its best match
5. Add who applies the protection

Miro links (Seminar PA193/01)

- Group 01: https://miro.com/app/board/uXjVPnmVaQc=/?share_link_id=575386214508
- Group 02: https://miro.com/app/board/uXjVPnl4PTo=/?share_link_id=980893964679
- Group 03: https://miro.com/app/board/uXjVPnl4Pe8=/?share_link_id=425798590612
- Group 04: https://miro.com/app/board/uXjVPnk21Xc=/?share_link_id=491507747684

Miro links (Seminar PA193/02)

- Group 01: https://miro.com/app/board/uXjVPnk21Tk=/?share_link_id=308394666113
- Group 02: https://miro.com/app/board/uXjVPnk2ylQ=/?share_link_id=853935301744
- Group 03: https://miro.com/app/board/uXjVPnk2yjo=/?share_link_id=216370791764
- Group 04: https://miro.com/app/board/uXjVPnk2yvg=/?share_link_id=865653713272

Miro links (Seminar PA193/03)

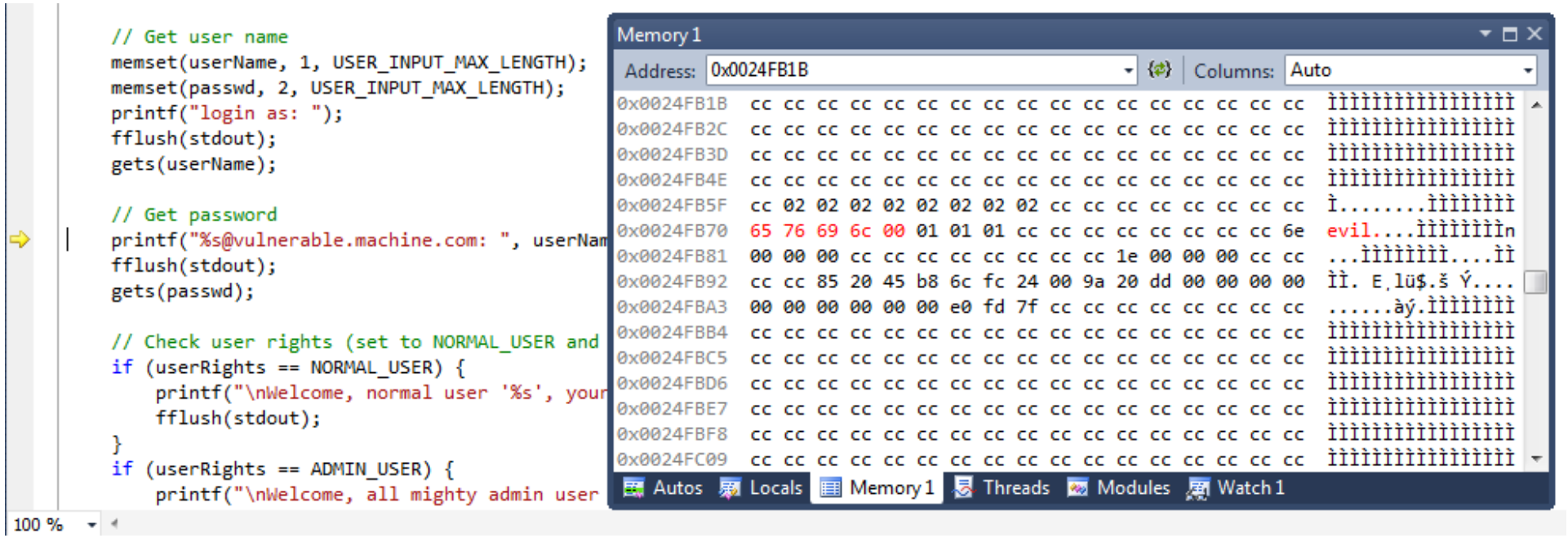
- Group 01: https://miro.com/app/board/uXjVPnk2yx8=/?share_link_id=948527080355
- Group 02: https://miro.com/app/board/uXjVPnk2y8I=/?share_link_id=165488852701
- Group 03: https://miro.com/app/board/uXjVPnk2y9o=/?share_link_id=387044453097
- Group 04: https://miro.com/app/board/uXjVPnk2y4w=/?share_link_id=357716600890

Group activity: Pair topics - analysis

- Take quick look at solution of another team (5 min)
  - Group 1 looks at Group 2, …
- Can you spot any differences to your solution?
  - Highlight one which was different, but also possible

INSPECTING MEMORY CORRUPTION

Live demo, explanation of the code



```c
// Note: GCC and MSVC uses different memory alignment
// Try "12345678DevilEvecosia" as a password for gcc build
// Try "1234567812345678Devil I am. Ha Ha" as a password for MSVC debug build

void demoBufferOverflowData() {
	int		unused_variable = 30;
#define NORMAL_USER		'n'
#define ADMIN_USER		'a'
	int		userRights = NORMAL_USER;
#define USER_INPUT_MAX_LENGTH  8
	char	userName[USER_INPUT_MAX_LENGTH];
	char	passwd[USER_INPUT_MAX_LENGTH];

	// print some info about variables
	printf("%-20s: %p\n", "userName", userName);
	printf("%-20s: %p\n", "passwd", passwd);
	printf("%-20s: %p\n", "unused_variable", &unused_variable);
	printf("%-20s: %p\n", "userRights", &userRights);
	printf("\n");

	// Get user name
	memset(userName, 1, USER_INPUT_MAX_LENGTH);
	memset(passwd, 2, USER_INPUT_MAX_LENGTH);
	printf("login as: ");
	fflush(stdout);
	gets(userName);

	// Get password
	printf("%s@vulnerable.machine.com: ", userName);
	fflush(stdout);
	gets(passwd);

	// Check user rights (set to NORMAL_USER and not changed in code)
	if (userRights == NORMAL_USER) {
		printf("\nWelcome, normal user '%s', your rights are limited.\n\n", userName);
		fflush(stdout);
	}
	if (userRights == ADMIN_USER) {
		printf("\nWelcome, all mighty admin user '%s'!\n", userName);
		fflush(stdout);
	}

	// How to FIX:
	//memset(userName, 0, USER_INPUT_MAX_LENGTH);
	//fgets(userName, USER_INPUT_MAX_LENGTH, stdin);
	//memset(passwd, 0, USER_INPUT_MAX_LENGTH);
	//fgets(passwd, USER_INPUT_MAX_LENGTH, stdin);
}
```

Setup

- Create new Visual Studio Project
  - File->New->Project->VisualC++->Win32 Console app
  - Turn off ‘Precompiled header’ and ‘SDL checks’
- Paste BufferOverflow.cpp from IS instead of project’s main file
- Try to compile (disable warning on gets() function)
  - #define _CRT_SECURE_NO_WARNINGS
- Insert breakpoint (begin of demoBufferOverflowData()) – F9
- Run program in debug mode – F5
- Execute next step of program – F10

```
void demoBufferOverflowData() {
    int         unused_variable = 30;
#define NORMAL_USER         'n'
#define ADMIN_USER          'a'
    int         userRights = NORMAL_USER;
#define USER_INPUT_MAX_LENGTH  8
    char        userName[USER_INPUT_MAX_LENGTH];
    char        passwd[USER_INPUT_MAX_LENGTH];

    // print some info about variables
    printf("%-20s: %p\n", "userName", userName);
    printf("%-20s: %p\n", "passwd", passwd);
    printf("%-20s: %p\n", "unused_variable", &unused_variable);
    printf("%-20s: %p\n", "userRights", &userRights);
    printf("\n");

    // Get user name
    printf("login as: ");
    gets(userName);

    // Get password
    printf("%s@vulnerable.machine.com: ", userName);
    gets(passwd);

    // Check user rights (set to NORMAL_USER and not changed in code)
    if (userRights == NORMAL_USER) {
        printf("\nWelcome, normal user '%s', your rights are limited.\n\n", userName);
    }
    if (userRights == ADMIN_USER) {
        printf("\nWelcome, all mighty admin user '%s'!\n", userName);
    }
}
```

Variable containing current access rights

Array with fixed length (will overflow)

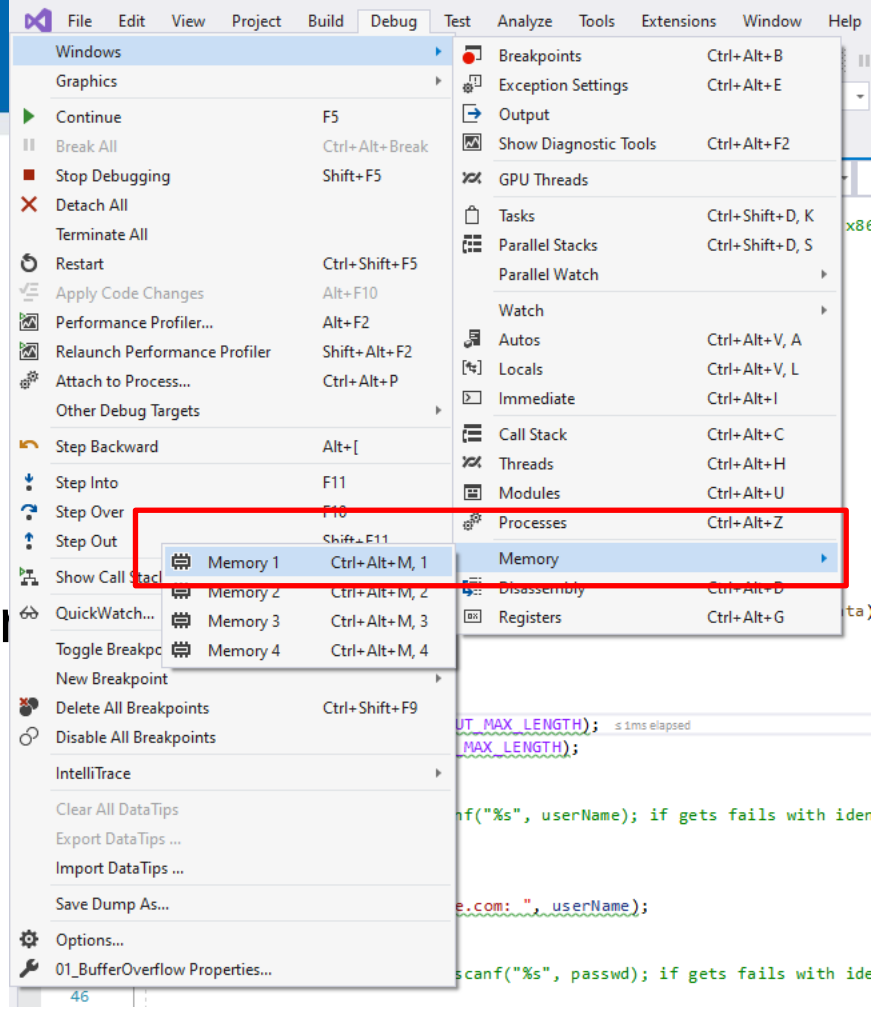
Help output of address of local variables stored on the stack

Reading username and password (no length checking)

Print information about current user rights

Setup – display memory content

- Display memory window
  - Debug → Windows → Memory
  - Program must be already in debugging session and
- Insert address to display surrounding memory
  - value of pointer (e.g., passwd)
  - address of variable (&userRights)
  - Just userRights is not OK (value inside variable)



Data in memory

```
void demoBufferOverflowData() {
    int     unused_variable = 30;
#define NORMAL_USER     'n'
#define ADMIN_USER      'a'
    int     userRights = NORMAL_USER;
#define USER_INPUT_MAX_LENGTH   8
    char    userName[USER_INPUT_MAX_LENGTH];
    char    passwd[USER_INPUT_MAX_LENGTH];

    // print some info about variables
    printf("%-20s: %p\n", "userName", userName);
    printf("%-20s: %p\n", "passwd", passwd);
    printf("%-20s: %p\n", "unused_variable", &unu
    printf("%-20s: %p\n", "userRights", &userRigh
    printf("\n");

    // Get user name
    memset(userName, 1, USER_INPUT_MAX_LENGTH);
    memset(passwd, 2, USER_INPUT_MAX_LENGTH);
    printf("login as: ");
    fflush(stdout);
```

100 %

Memory 1

Address: 0x0024FB1B Columns: Auto

```
0x0024FB1B  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0024FB2C  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0024FB3D  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0024FB4E  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0024FB5F  cc 02 02 02 02 02 02 02 02 cc cc cc cc cc cc cc cc
0x0024FB70  01 01 01 01 01 01 01 01 cc cc cc cc cc cc cc cc 6e
0x0024FB81  00 00 00 cc cc cc cc cc cc cc cc cc 1e 00 00 00 cc cc
0x0024FB92  cc cc 85 20 45 b8 6c fc 24 00 9a 20 dd 00 00 00 00
0x0024FBA3  00 00 00 00 00 00 e0 fd 7f cc cc cc cc cc cc cc cc
0x0024FBB4  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0024FBC5  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0024FBD6  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0024FBE7  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0024FBF8  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
0x0024FC09  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
```

Autos | Locals | Memory 1 | Threads | Modules | Watch 1

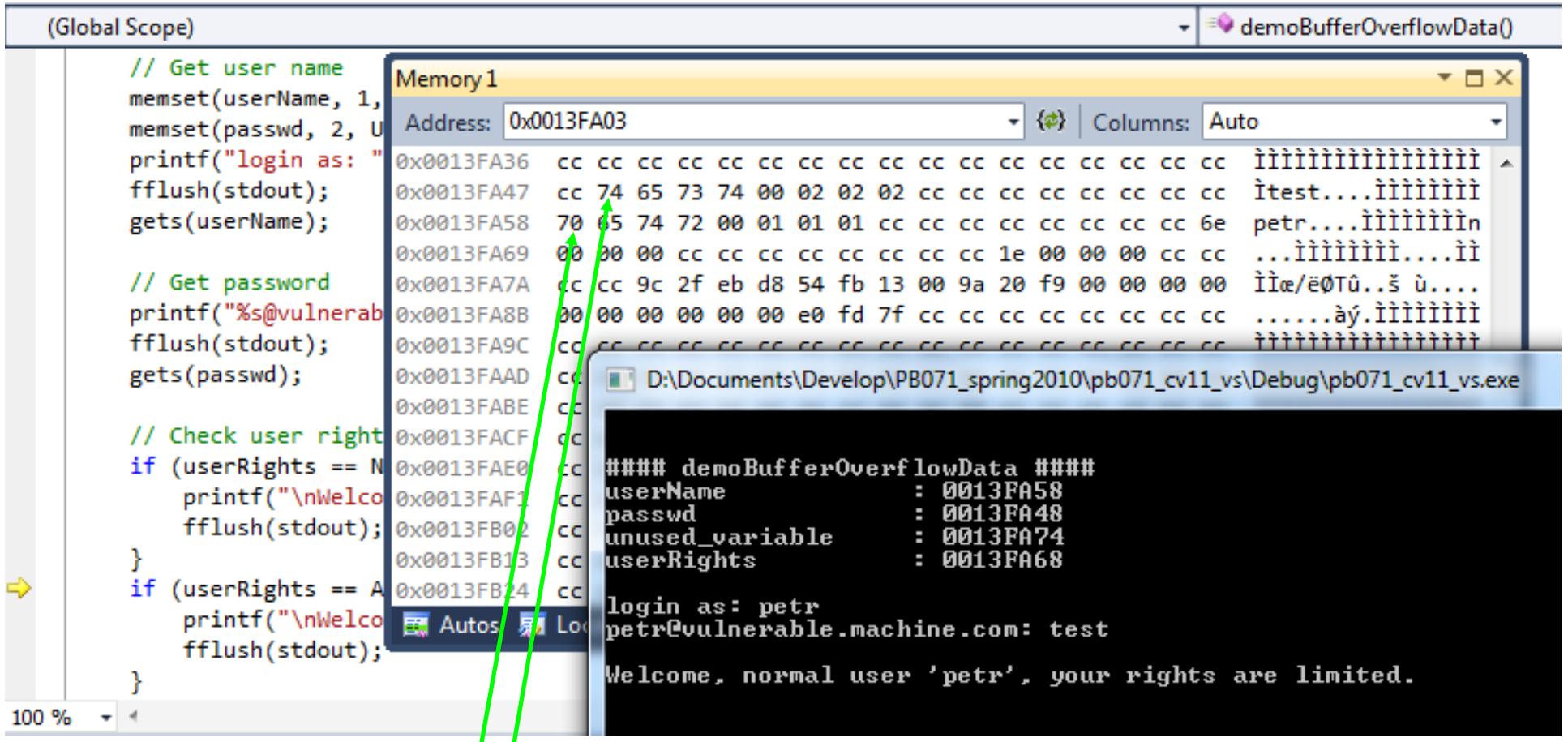
userName

passwd

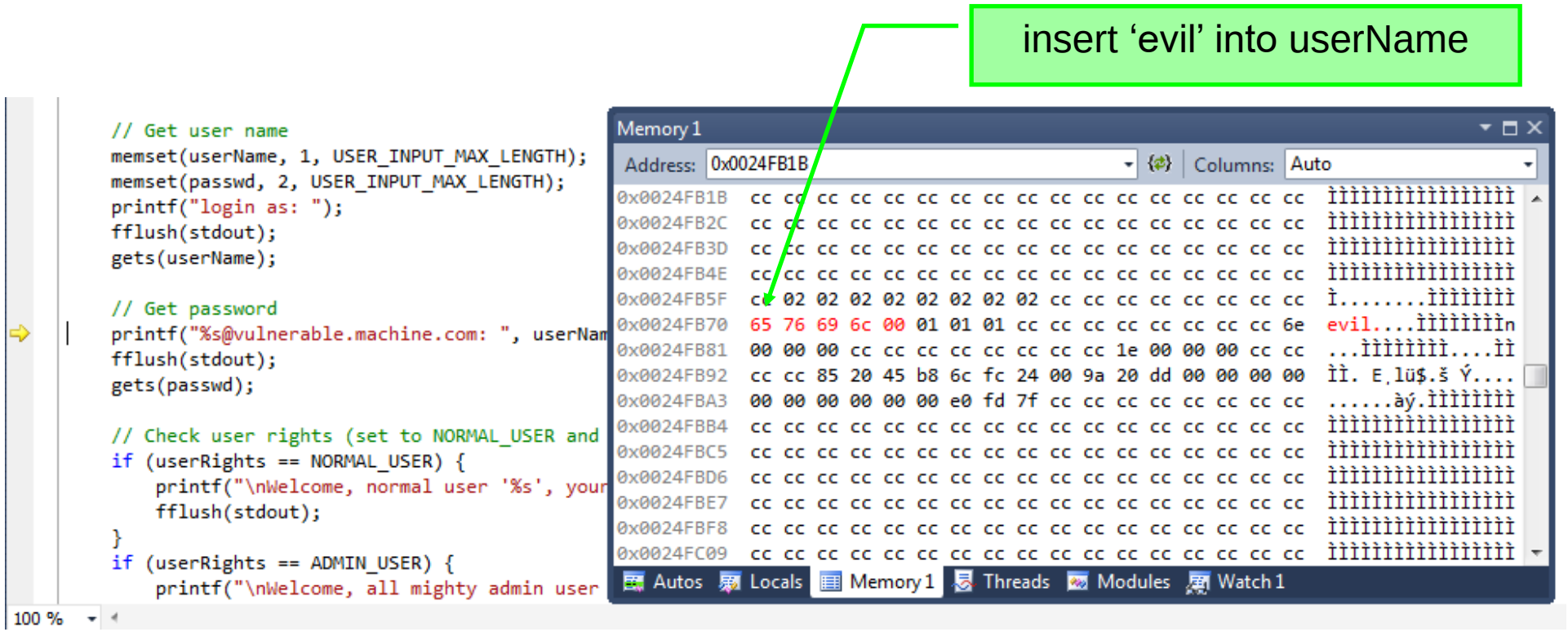
userRights

unused_variable

Running without malicious input



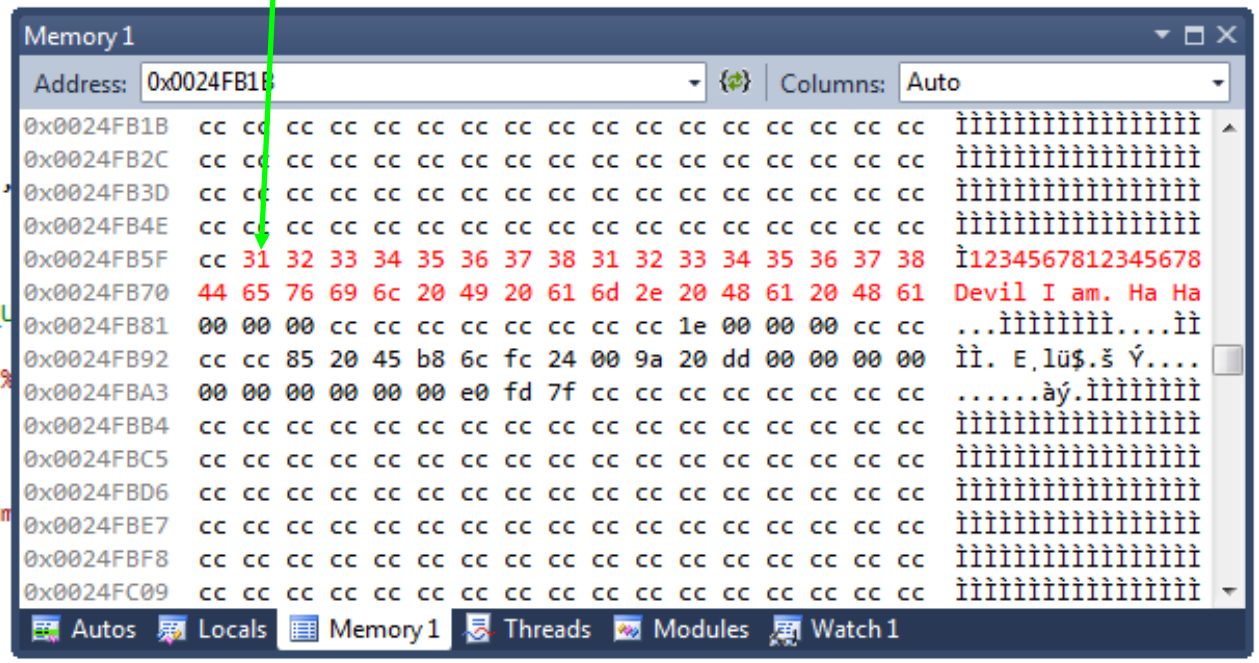
Running with malicious input – userName



Running with malicious input - passwd

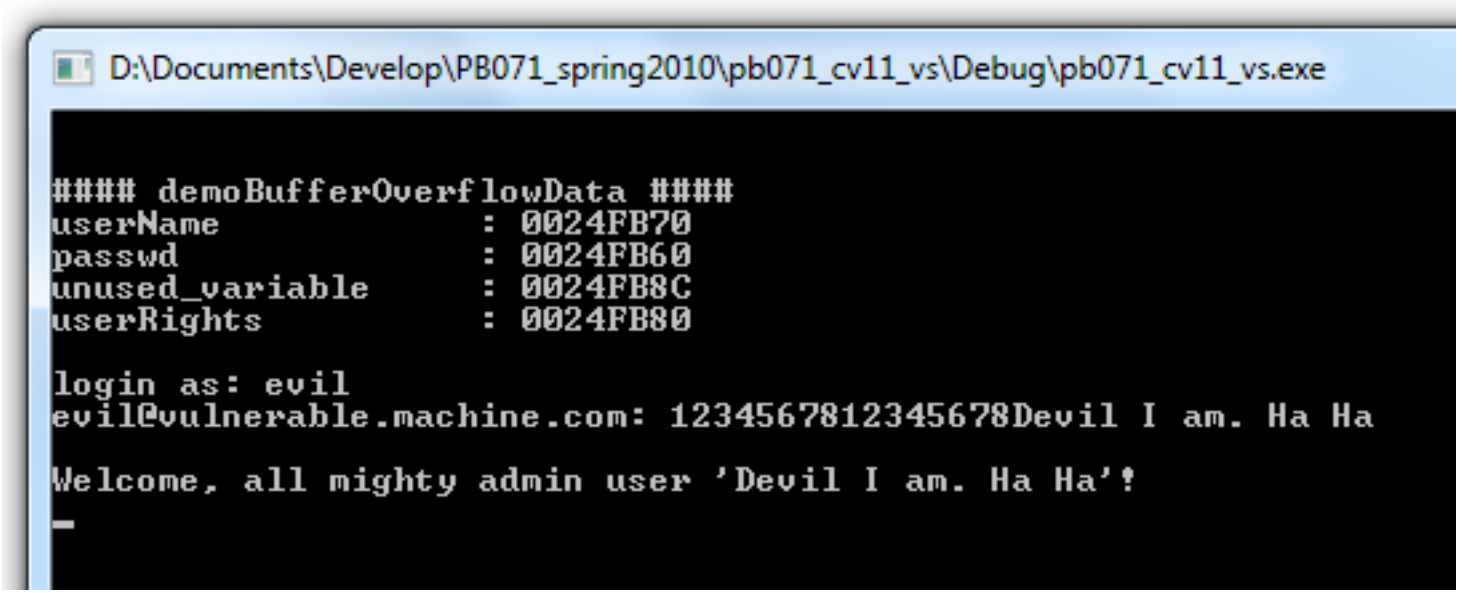
Insert
‘1234567812345678Devil I am. Ha Ha’
into passwd

```
printf("login as: ");
fflush(stdout);
gets(userName);

// Get password
printf("%s@vulnerable.machine.com: ",
fflush(stdout);
gets(passwd);

// Check user rights (set to NORMAL_U
if (userRights == NORMAL_USER) {
    printf("\nWelcome, normal user '%
    fflush(stdout);
}
if (userRights == ADMIN_USER) {
    printf("\nWelcome, all mighty adm
    fflush(stdout);
}

// How to FIX:
```



- Too long password overflow **`userName`** and **`userRights`**

Running with attacker input - result



Questions (debug mode)

- How are userName, password and userRights positioned in memory?
- How will you find memory location (address) of *userRights* variable?
- How many bytes you need to write into *userName* variable to change *userRights* ?
- Can you get admin rights by changing userName only?
- Is your system little or big endian?

Questions (debug mode)

- Why is program throwing debugger exception when finishing function demoBufferOverflowData()?
- How program was able to detect memory corruption?
- Why 0xcc bytes are here? How can you type 0xcc into terminal?
- Can you get admin rights without raising runtime exception (*memory around userName variable corrupted*) when leaving demoBufferOverflowData()?
  - What caused the runtime exception to be emitted?
- Where can you find the return address?
- What should be the return address value?
  - Try R-Click -> Go to Disassembly

Questions (release mode)

- Release mode, /GS on
  - What is memory layout with respect to debug mode?
  - Can you still execute buffer overflow and change userRights?
  - What is the value of canary word?
- Release mode, /GS off
  - What is the influence of /GS disabled?
  - What is the impact on addresses of variables?
  - Can you become admin in Release? Why?

STACK PROTECTION SWITCHES

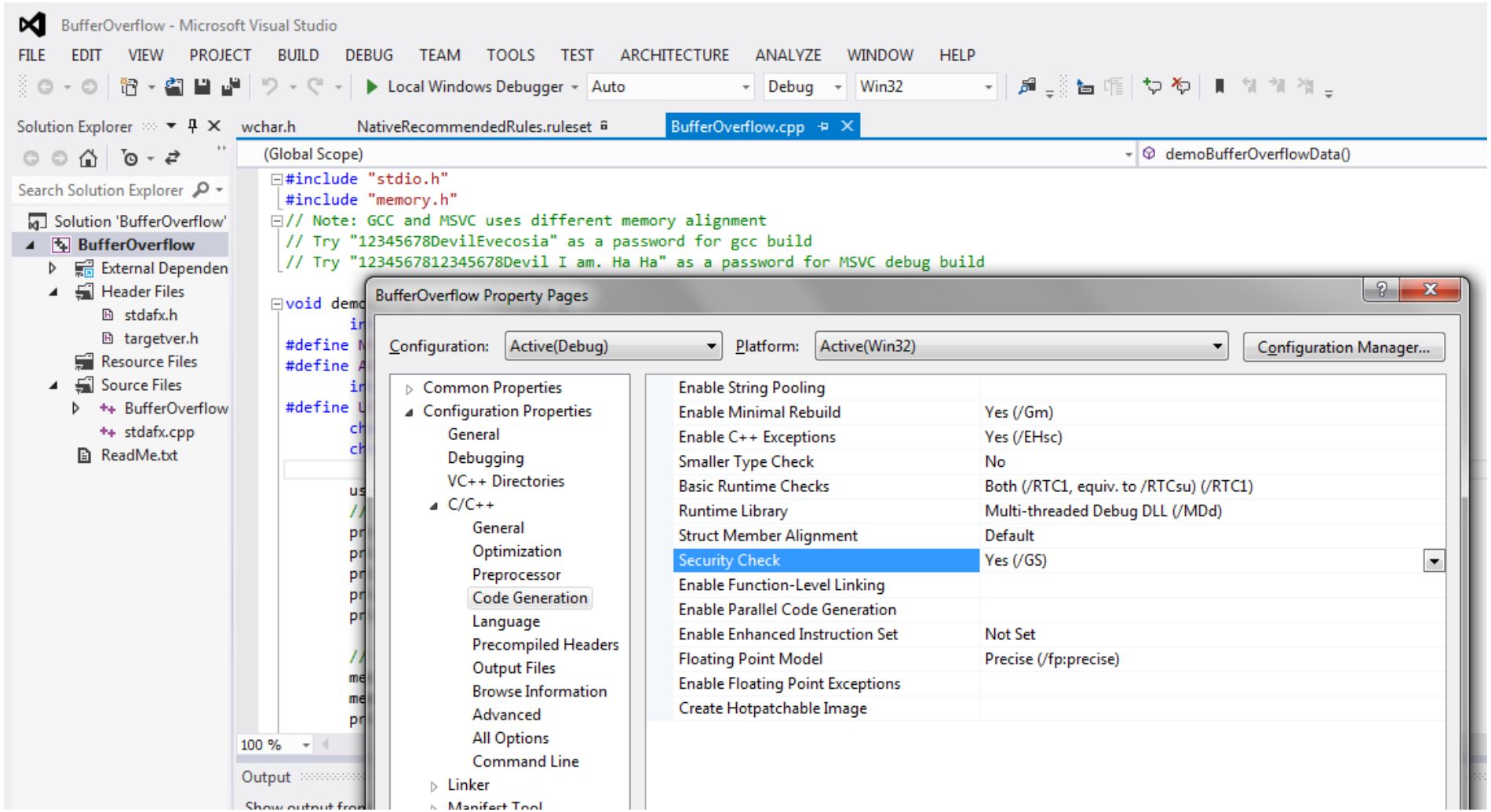
Lab – compiler protections

- GCC (e.g., QT Creator) & MSVC (Visual Studio)
  - list of compiler flags, release mode
- Compile program with/without compiler protection
  - bufferoverflowdemo.cpp::demoBufferOverflowData()
    - download from IS materials
  - return pointer smash behavior (crash, exception)
- Disassembly display of resulting binary
  - instruction-wise mode in IDE (Visual Studio), OllyDbg
  - existence of canary word (function with/without GS buffer)
- Display address of variable, function...,
    - run program multiple times – memory randomization (ASLR)

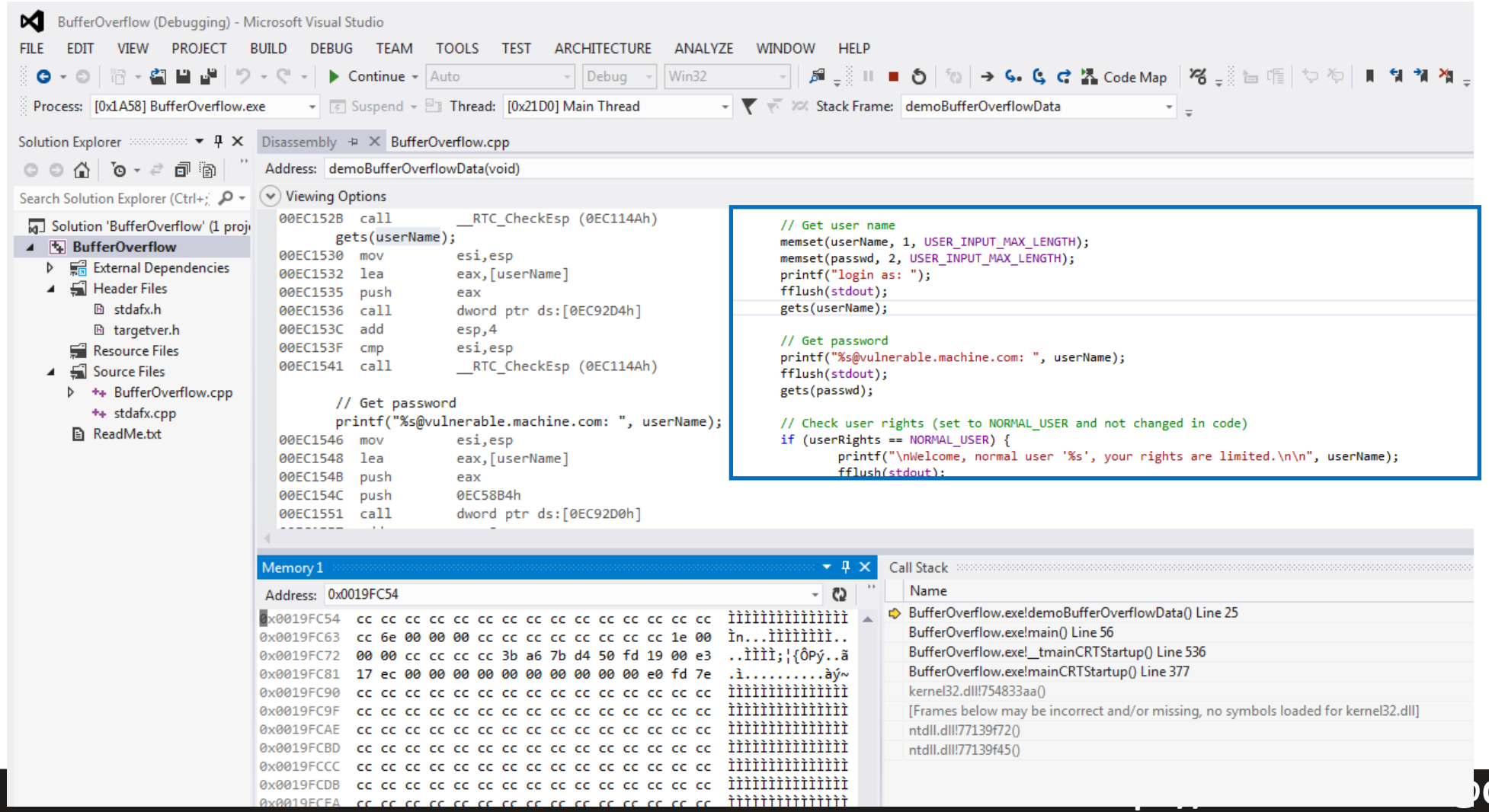
Compiler flags

- Locate all flags discussed during the lecture
- Visual Studio Projects Settings
- Observe memory layout for stack frame with and without a flag (/GS, /DYNAMICBASE)
  - what is changing?
  - what is missing?

Compiler settings for /DEP and /ASLR



Deeper look into disassembly



HOMEWORK ASSIGNMENT

Assignment 1: Stack memory layout

- Perform an analysis of stackframe memory layout as set by your compiler in different configurations
  1. Create small C/C++ program with three arguments (int, char[8] and struct) and three local variables (float, char[8], int[8])
  2. Compile at least in four configurations: (debug | release) and (with | without stack protection)
  3. For every configuration, obtain and annotate memory layout of:
     a) Arguments and local variables
     b) Position of stack canary
     c) Position of return address
- Describe how you obtained the memory layout

Assignment 1: Stack memory layout

- Produce short (max 2xA4) text with the description of your solution
  - Document all compiler and platform settings used (for replicability)
  - Present and discuss the results obtained
  - Explain why memory layout is impacting the attacker's capability to exploit a buffer overflow
  - Mention any surprising things you observed
- What to submit
  - Source code of the testing application
  - Text description of your solution (pdf)
- When and where to submit
  - Submit **before 27.2.2023 23:59** into IS HW vault
  - Soft deadline: -3 points for every started 24 hours

THANK YOU FOR COMING, SEE YOU NEXT WEEK

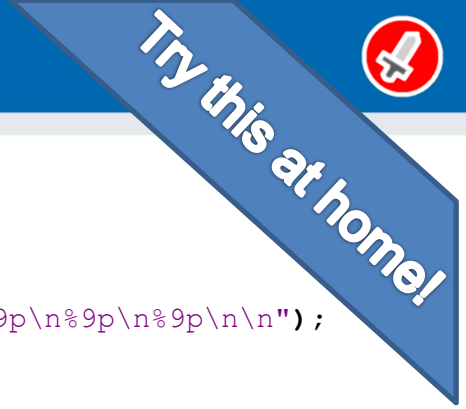
SUPPLEMENTARY MATERIAL – HAVE FUN!

Other stack overflow demos

1. Stack overflow – auth. privileges change
2. Adjacent memory overflow – reveal password
3. Smash function return address – attacker code exec
4. Type overflow – overflow integer causing BO
5. Buffer overflow - shell execution

- Example with debugging with instruction-wise mode


```
void demoAdjacentMemoryOverflow(char* userName, char* password) {
        // See more at http://www.awarenetwork.org/etc/alpha/?x=5
    // Once string is not null terminated, lot of functions will behave wrongly:
        // sprintf, fprintf, snprintf, strcpy, strcat, strlen, strstr, strchr, read...
        // memcpy, memmove - if length to copy is computed via strlen(string)

        char message[100];
        char realPassword[] = "very secret password nbu123";
        char buf[8];

        // print some info about variables
        printf("%-20s: %p\n", "message", message);
        printf("%-20s: %p\n", "userName", userName);
        printf("%-20s: %p\n", "password", password);
        printf("%-20s: %p\n", "realPassword", &realPassword);
        printf("%-20s: %p\n", "buf", &buf);
        printf("\n");

        memset(buf, 0, sizeof(buf));
        memset(message, 1, sizeof(message));
        strncpy(buf,userName,sizeof(buf));              // We will copy only characters which fits into buf

        // Now print username to standard output - nothing sensitive, right?
        sprintf(message, "Checking '%s' password\n",buf);
        printf("%s", message);
        if (strcmp(password, realPassword) == 0) {
                printf("Correct password.\n");
        }
        else {
                printf("Wrong password.\n");
        }

        // FIX: Do not allow to have non-terminated string
        // Clear buffer for text with zeroes (terminating zero will be there)
        // strncpy(buf,arg1,sizeof(buf) - 1);
}
```




```c
void vulnerable_function(const char* input) {
    char buf[10];
       memset(buf, 0, sizeof(buf));

       printf("%-20s: %9p\n", "buf", &buf);
    printf("My stack looks like:\n%9p\n%9p\n%9p\n%9p\n%9p\n%9p\n%9p\n%9p\n%9p\n%9p\n%9p\n%9p\n%9p\n\n");
    fflush(stdout);
    strcpy(buf, input);

    printf("Now the stack looks like:\n%9p\n%9p\n%9p\n%9p\n%9p\n%9p\n%9p\n%9p\n%9p\n%9p\n%9p\n%9p\n%9p\n\n");
    fflush(stdout);
}

/**
 * This function contains code that attacker likes to execute.
 * It may be otherwise inaccessible privileged function (e.g., something like "admin_createNewUser()")
 * or code not originally present in the program code, but inserted by an attacker into the memory
 * (with starting address set at the begin of inserted code)
 */
void attackers_code(void) {
    printf("Augh! I've been hacked!\n");
       system("mmc.exe lusrmgr.msc");
       //system("rm -rf");
}

void demoBufferOverunFunction() {
       // Example taken from Howard and LeBlanc, modified to run with gcc compiler
       // This is a bit of cheating to make life easier - we will print the addresses of functions
       // In real scenario, attacker needs to find it by himself (e.g., multiple tests on test machine, dump)
       printf("Address of vulnerable_function() = %p\n", vulnerable_function);
       printf("Address of attackers_code() = %p\n", attackers_code);
       printf("Address of demoBufferOverunFunction = %p\n", demoBufferOverunFunction);
       fflush(stdout);
       vulnerable_function("12345678");                              // No problem
       vulnerable_function("1234567812aaaaaaa");          // Writing behind the array, but still nothing happens
       vulnerable_function("1234567812aaaaaaaa\x14\xff\x22\x01\xde\x16\x40");  // But now its works  - attackers_code()!
}
```

BinScope Binary Analyzer

- Download Microsoft SDL's Binscope
  - https://blogs.microsoft.com/microsoftsecure/2012/08/15/microsofts-free-security-tools-binscope-binary-analyzer/
  - https://www.microsoft.com/en-us/download/details.aspx?id=44995
- Run BinScope Binary Analyzer (cmd or GUI)
  - **`binscope.exe`**
  - **`binscope.exe /o results.xml targetApp.exe`**
- Run on the binaries produced with different compiler settings
  - /GS...