### Information Security - 5 - Secure Systems Engineering Professor Chester Rebeiro Indian Institute of Technology, Madras Gdb-Demo

(Refer Slide Time: 0:22)



Hello, and welcome to this lecture so this is the demo in the course for in secure systems engineering. So I hope by now that you are actually install the virtual box and you are actually have this Ubuntu running as shown over here. So in this first demo we will actually look at the basics we will see how this stack is presented in a program and it also uses a tool called gdb which can be used to actually debug these programs.

(Refer Slide Time: 0:48)



So the first thing to do is to start the terminal, so you could do it like this (())(0:52) terminal.



(Refer Slide Time: 0:57)

If you had downloaded the codes from the website that we specified you would actually see that there is a directory called NPTEL codes, go there and in this video we will be looking at module (0), okay. So this particular module (0)(1:15) to the code that we have seen in the presentation.

(Refer Slide Time: 1:22)



So what it does is that it has a main function and an evocation to this function which is passed parameters 1, 2 and 3 and as we have seen in the previous videos this function just defines two buffers, buffer 1 of 5 bytes and buffer 2 of 10 bytes.

(Refer Slide Time: 1:40)



Now what we will see in this particular video is how we could actually analyse the stack for this particular program.

### (Refer Slide Time: 1:50)



Okay, the first thing we do is to actually create the executable for this particular program we do that by running make. So this make depends on what is known as a make file and it is just a script we will not go into the details about what is present in a make file but it is just a script that would commit us to invoke gcc specify various parameters for gcc and finally obtain the corresponding executable.

So this particular line is what the make file executes and it has created an elf file example 1, so some of these parameters for gcc like from minus F no stack protector, minus Z X 6 stack and so on. We will be actually seeing in a later video, for this particular video we would explain two specific parameters at minus M32 and minus G, so minus M32 indicates that the executable that is getting created in this case example 1 is a 32 bit executable and since this is

an Intel virtual box or Intel machine therefore, the executable that gets generated is a Intel 32 bit executable.

The minus G option that we specified over here ensures that debugging symbols get added into the executable. Now as we seen in the second video today this particular file the example 1 is an elf executable, so we have seen commands such as read elf minus H and we can look at the header of this particular executable and as we have seen in the previous video we would obtain the header for example 1.

So as seen this is the identifier 7f 45 4c 46 which actually transforms to elf in (())(4:01). Other things to actually note is that for the Intel 80386 which indicates that it is a 32 bit executable, so as we have seen the entry point for this executable is at the location 0x8048ce0 and we have also seen the various other aspects such as the section headers and so on. So in particular we can also use other read elf features such as read elf minus S to get more information about the section headers.

So read elf minus S dot slash example 1 would give you more information about the various sections that are present in this particular executable, okay.

	1 🖾 🕫
0 0 matelitesbases -/natel codes/madulat	
39: BBB+Bfbc B DB3ECT LOCAL DEFAULT 20 de alabel diors eux fin	
40 00048360 0 FUNC LOCAL DEFAULT 14 Frame_dumy	
411 08049708 0 0801CT LOCAL DEFAULT 19 _Trame_Bouny_Init_stray_	
(A) 00000000 0 FILE LOCAL DEFAULT ADS (ristoff.c	
44: 08048599 0 082ECT LOCAL DEFAULT 18 _FRAME_END_	
45: 08949710 0 0831C7 LOCAL DEFAULT 21 _3CR_END_	
49) 00000000 0 File LOGA, DEFAULT ASS 47: 0004970C 0 NOTVPE LOGAL DEFAULT 19 Init array end	
48: 08049714 0 082ECT LOCAL DEFAULT 22 DYNAMIC	
49) 08949708 0 NOTYPE LOCAL DEFAULT 19 LAIL Array start	
10: DEGREGER D DIGTOR LOCAL DEFINIT 17 DOULD FRAME FOR	
52) 08944409 2 FUNC GLOBAL DEFAULT 14 _11bc csu fiat	
53: 00000000 0 NOTYPE MEAK DEFAULT UND TH deregisterThCloneTab	
54: 08044310 4 PUNC GLOBAL HIDDIN 14 _x80.pt1_pc_thunk.bx	
56: 08044018 0 NOTYPE GLOBAL DEFAULT 25 _edata	
57: 00048464 0 FUNC GLOBAL DEFAULT 15 FLAL	
18: 04044010 0 NOTYPE GLOBAL DEFAULT IS0418_11471	
69: 08944014 0 063ECT GLOBAL HIDDEN 25 Toko Sandle	
61: 080-6847c 4 082ECT GLOBAL DEFAULT 16 TO_stdta_used	
63: D0000000 0 FURC CLOBAL DEFAULT UNDATABBBCLIEC_	
641 08944014 0 NOTYPE CLOBAL DEFAULT 20 _end	
651 000482e0 0 FUNC GLOBAL DEFAULT 14 start	
65: 08044479 4 083ECT GLOBAL DEFAULT 16 FD_hm	
GBI DBD48344 3D FUNC GLOBAL DEFAULT 14 mala	
69: 000483db 9 FUNC GLOBAL DEFAULT 14 function	
To beside a NOTYPE WEAK DEFAULT UND DV RegisterClasses	
72: 00000000 0 NOTYPE WEAK DEFAULT UND ITR registerTecloneTable	
731 0004828c 0 FUNC GLOBAL DEFAULT 11 LAIT	
nptelBosBoses:-/nptel_codes/module0\$ e: Intel 80306	
entrollententer (- metroll codes/metrollents 14	
example: example1.c Makefile	
<pre>npteleoshoxes:-/nptel_codes/nodule4\$ objdumpdisassemble-all ./example1 &gt; exam</pre>	
and hardward - (note), and a look and	
	00000000

(Refer Slide Time: 4:58)

Another important tool that we will actually look at to go more into detail about the example 1 executable is this objdump, so if I do something like objdump minus minus is assemble all example 1 and it would actually store the output in example 1 dot list, what this particular tool would do is that it would disassemble example 1 and store the disassembly in this file called example 1 dot list.

### (Refer Slide Time: 5:45)



So we could open another tab and look at example 1 dot list which looks like this we can search for main, it is here which is the main function and we could also open the corresponding C code and C that in mean there is a invocation to this function which is essentially done by this particular instruction, in this location 80483ED. So these (())(6:32) here is the machine code for this instruction call to function.

And as we have seen in the previous videos the three parameters 1, 2 and 3 which is passed to the function a pushed on to the stack, so it is pushed from the right that is push 3, push 2 and push 1, over here we see the disassembly for this particular function and as we know there is a preamble in the function where the stack frame is created for this particular function, there are space for the local variables over here and find P the function returns.

So in this particular instruction we see that there are 0 X1 0 that is 16 bytes for the stack frame, so this is because the total size for the local variables is 16 bytes. The next tool that we will actually look at is the most important so that is known as the gdb gnu debugger and this tool can be used to actually debug this program look at the various registers, look at the stack contents and so on.

# (Refer Slide Time: 7:48)



So the way to actually start this tool is to provide gdb with the executable example 1, so gdb would provide you a prompt like this and through this prompt you could actually send various commands such as the list command which would list the entire source code like this. Also you could send break points such as b to line number 10, so what this means is that the program will execute until the break point 10 act line number 10 is obtained. So this could mean that the program will execute until line number 10 and then it will wait for further action.

(Refer Slide Time: 8:38)



So let us now run this particular program, we run it by this command called r and what you see is that gdb tells you that break point 1 at line 10 is reached. So we can look into the

various registers which are present, now at this particular point we can look at the various contents of the various registers and this can be done with a command like info registers eip which stands for the instruction pointer, esp which stands for the stack pointer register and the ebp which is the frame pointer in the stack.

So we look at this and we see that eip is pointing to a location over here that is 80483e7, now if we go back to this code we see that the eip is actually pointing to this location over here essentially this instruction has to be executed. So we also see that the stack pointer is at the location 0xffffcf68.

(Refer Slide Time: 9:54)



So what we could do is we could actually print the contents of the stack with a command like this x slash 32x dollar esp and what this command does is that it essentially dumps the memory starting at the location specified by the stack pointer which in this case is ffffcf68 and this 32 over here indicates a number of memory locations that needs to be displayed, the second x over here specifies that the display should be in hexa decimal values.

So now we have seen that there is the instruction pointer pointing to this location push 03 which has not yet been executed, we have a stack pointer which is pointing to the location ffffcf68 and the base pointer in this particular case is also ffffcf68 the reason for this is that the stack pointer over here in this instruction move esp to be ebp the stack pointer is in fact copied to be base pointer and therefore the stack pointer and the base pointer are the same.

# (Refer Slide Time: 11:12)

	C C D control and the second s	
( ) ( ) spin@ubure: -puper(userprodukt ) ) ( ) ( ) ( ) ( ) ( ) ( ) ( )	<pre></pre>	ATU
PE DBBBBBD DBBBBBD DBBBBBD DBBBBBD DBBBBBB DBBBBBBB DBBBBBBBB DBBBBBBB DBBBBBBB DBBBBBBB DBBBBBBBB DBBBBBBBB DBBBBBBBB DBBBBBBBB DBBBBBBBB DBBBBBBBB DBBBBBBBB DBBBBBBBB DBBBBBBBB DBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB	Bill (1) - fract(1)	334
denbaldfa ollor: denbaldfa ollor: den def ar næmler dom. (mb) ti		

Now this similar disassembly what we see using objdump can be also obtained with gdb, so all we need to do is specify this command called disassemble and it would give us the exact same details. So we see that the instruction specified over here in gdb is the same instructions that are specified in this listing which we have obtained from objdump, what is specified in gdb because gdb tells you the run time status of the particular program is this additional point over here which tells you the current location where the program counter is present.

(Refer Slide Time: 12:12)

					OOO NAKA	(Bosherer -/hptel_codes/h	neduled	
					2 (	account one of one of o		
						Char buffer1[1]; char buffer1[1];		
000 m	elfenheur vientel i	ades/module0			ę )			
0+00040	f5 ++17+1 840							
0x00048	fő «+18+: leave				. (	function (1, 1, 1);		
End of ass	enter dump.				ii )			
(gdb) si					12			
(adb) d144	senhle	Function(1,2,						
Dump of as	embler code for fo	notion mate:						
0+00040	e4 codic: push	Sebp						
0+00045	e7 (+3): push	58x3						
+> Ex08048	e9 ex5x1 push	\$8x2						
0+00048	ed ++9>: call	8x80483db +fu						
010048	() <+14>: add	Souc,Wesp			example1.c		ine .	1,2
0x00046	f6 <+18>: leave							
0+00046	f7 ++19>1 ret				00010340 479	ection»)	math	taka .
(pdb) x/32	Sesp							Kesp, Kebp
Barritetes	8+8080883	8+80000000	8xf7e28637	6+0000001	00403601		140	\$0.10, <b>%esp</b>
Baffffcf84	8+8000000	8x17158888	0xf7ff6c04	8+17116000			Leave	
Baffffef94	8-8000000	8xf7fb8888	8+17158888	8+0000000	10103431			
Baffffcfb4	8+50000000	6x56645600	8+565482e5	Ex0000000	01011245 198			
Baffffefc4	Buf7feeffb	0xf7fe9880	8xf7ffd000	Ex80000001	2046344		push	Kebp
Codb) al	8×80848248	*********	0+00040301	8x80848344	00403001	(4 03	push	\$0.0
04040483eb		function(1,2,			10461091		push	
(gdb) 41		function (1.2			00403601	45 45 77 77 77	penh (all	post functions
(edb) 1/32	Seip							
earriterse	8+80000001	8x86666682	8x8000003	8+8000000	00403451		nep Leave	
earrrent cree	8+8688888	8+80000000	8+8000000	8477756000	101010	ä	ret	
BaffffcfBc	8xf7ff6c84	8xf7ff6000	8+0000000	6xf7fb8000	example1.lst		_	333,11
Baffffcfac	0+0000000	0+00000000	8+50775638	Ex50000001	HART HA			
Baffffefbe	8+88848248	8+80000000	exf7feeffe	8xf7fe9880				
duffffefec:	6xf7ffd000	8×80000001	0x000482e0	Ex30000000				
1000								

Now we could execute each instruction individually by a command known as single step or single instruction and we shorten this command like si so when we specify si a single instruction in this case the push instruction would get executed and we will actually look at this, so it says that this instruction which ends in 4a3e7 has executed and we could also see if we disassemble again that the program counter or the instruction pointer has moved to the next instruction.

We can also see the effect on the stack by dumping the stack contents as we have done before so that we have done by this x 32x followed 3esp and we see that there is a difference here the contents of 3 have been pushed on to the stack, single step through the next instruction like this and one more instruction like this and then look at the stack again and we see that all the parameters 1, 2 and 3 have been pushed on to the stack. The instruction pointer now is pointing to this call instruction.

So as we have seen in the previous video when there is a call instruction what is done is that this particular address 80483db is moved into the instruction pointer at the same time the instruction of the next address that is 080483f2 gets pushed on to the stack.

(Refer Slide Time: 13:40)

So let us see how this happens with a single instruction, okay. So we would look at the stack again and we will see that the next instruction 08483f2 which is the return address is pushed on to the stack, at the same time if you look at the contents of the registers we see that the instruction pointer that is the eip is pointing to a location 80483db which is the start of this particular function.

Now as we have seen in the previous video what this function initially does in its preamble is that it pushes into the stack that is the previous frame pointer into the stack, it moves the stack pointer to the base pointer and then this subtract instruction allocate 16 bytes of data in the stack.

(Refer Slide Time: 14:48)



So let us see this happening using 3 instructions, so I can just specify si3 which means that 3 instructions have to be executed, okay and if I disassemble it over here I see that these 3 instructions push ebp move esp to ebp and subtracts 16 from esp has been executed. So I could look at the contents of the stack again are like follows and what we see is that now the stack has actually gone down to ffffcf44 and the reason for this is because of this subtract instruction which is essentially allocating 16 bytes of data who hold buffer 1 and buffer 2.

(Refer Slide Time: 15:48)

		1 Bold function(int a, int b, i	(At c)	
		2 { the beffect(s);		
		4 char buffer2[10];		
		1		
🔵 🗇 ngtel@osboxex -/rgtel_codes/module0				
8xffffcfcc: 8xf7ff4000 8x80000001 8x800482e8	8+8000000	<pre>vold main()</pre>		
(pdb) st function (and the2 col) at manufal co2		10 function(1,1,1);		
runction (#+1, b+2, c+3) at example1.ct2		11 )		
(gdb) x/32x Sesp		12		
Suffff(fis) Substants Suffrance) Substants	Ex00000003			
8xffff(f)\$: 8xffffd00c 8x80000000 8x80000000	8+86666666			
Baffffcf88: Baf7fb8000 Baf7ff6c84 Baf7ff6000	8+0000000			
6x77764000 6x77764000 6x77764000 6x0000000	Exbd7fb61d			
8xffffcfb8: 8x00000001 8x000442c0 8x00000000	Baf7feeffb			
8xffff(f(8) 8xf7fe9880 8xf7ff6000 8x80000003	8+86848248			
(pdb) info registers elp esp ebp		example1.c	1.2	ALL
etp 0x80483db 0x80483db 4Function>		1040.1461 e9 15 ff ff ff	5mp 1010110 eregister_in_clones	
ebp Buffffcf68 Buffffcf68				
(pib) st 3		CICLE (CONCELSER)	and the	
(ath) discontile		00403441 00 45	nov Xeso Xebo	
Dump of assembler code for function function:			sub \$0x10,Nesp	
Babbbbbbbbb ++B+1 push Kebp		10403411 50	nep Taxaa	
Ballada Maria (alla) Nov Matp Matp		10103431 43	ret	
+> 6x886483e1 ++6+1 640				
debddeble2 exts: Leave		01010104 (MALAN)		
Babbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb		10413#\$1 07 #\$	nov Kesp Kebp	
(odb) x/32x Setp				
0xffff(f44) 0xffffd004 0xffffd00c 0x00048421	8xf7fb836c	80483491 64 02	pesh \$0x2	
8xffff(f54) 8xffff(f68 8x88848372 8x888683	8+8000002	southed at at ff ff ff	call shilldh cfunctions	
Auffffeffal Auffffahl Auffffahl Auffffahl	Ex 00000001	10413#21 03 64 Mc	add Stor, Nesp	
8xffff(f84) 8x80000000 8xf7fb8000 8xf7ff6(84	0x77774000			
8xffff(f94: 8x8000000 8xf7fb0000 8xf7fb0000	8+8000000	10111161 69	Leave	
Buffffcfa4: Bubd7fb63d Bubbeababd Bubbobbbb	Ex0000000	example1.1st	333.11	335
(odb) info registers eip esp ebp		iset nu		
elp dusb483e1 dusb483e1 «function=d»				
abo Auffferfed Buffferfed	_			
((()))				

So if I look at the contents of the registers info registers eip which is (())(16:00) ebp, you see that there is a distance between the stack pointer and the corresponding base pointer, so this region between the stack pointer and the base pointer is the active frame for that particular function. Now within this particular region between ffffcf44 and ffffcf54 is where the locals of this particular function resides.



(Refer Slide Time: 16:30)

So we can look at the locals for this function using this command info locals and we see that there are 2 locals specified in this function, so buffer 1 and buffer 2. We can also look at the address of buffer 1 and buffer 2 as follows at p which is a print slash x and ampersand buffer 1 would provide the address for buffer 1. Similarly p slash x and change it to buffer 2 which specify the address for buffer 2 and what we see over here is that buffer 1 and buffer 2 is both within the stack frame that is both are within the stack pointer and the base pointer.

# (Refer Slide Time: 17:25)



Now let us get back to the disassembly and what we see is that there are 2 instructions remaining one is the leave instruction and then the return instruction, the leave instruction essentially restores the stack frame such that the old frame corresponding to the previous function in this case the main function is restored and the return would then return back to the main function.

So let us see this happening, so I can do single instruction 3 which should execute 3 instructions and what we see is that the execution has gone back to the main function. Now as before we could also verify this, we could look at the various registers and we see that the instruction pointer now points to somewhere in main in fact it is pointing to the function soon after the call which is this which corresponds to the add function and what we also see is that the stack pointer and the base pointer have moved up in the stack.

So now we have completed executing execution of that particular function and since we have moved back to the main function, so the active frame now is corresponding to that for the main function. So all the local variables if any that are present in the main would come back and would actually be active over here. So gdb is one of the best debugging softwares that are available, it is (())(19:01) and what we actually capture is just the small glimpse of what gdb can do.

So you could also search and we will also try to share a document which comprises of all the various commands the gdb has and you can actually try various things with gdb and also try

to do such debugging with various other programs using gdb, look at the various registers and see how the stack and other local variables are maintained, thank you.