System Programming – Bomb Lab

1. Description - Binary bombs

This practical task is defusing binary bomb. A binary bomb is a program that consists of a sequence of phase. Each phase requires you to type a particular string on stdin. If you type the correct string, then you can enter to next phase. Otherwise, the bomb explodes with printing "BOOM!!!" and then program will be terminated. You need to find out correct strings for all phases to defuse the bomb.

2. What to do

Your job for this lab is to defuse 6 phases.

The bomb ignores blank input lines. If you run your bomb with a command line argument, for example,

$./bomb answer.txt

then it will read the input lines from answer.txt until it reaches EOF (end of file), and then switch over to stdin. In a moment of weakness, Dr. Evil added this feature so you don't have to keep retyping the solutions to phases you have already defused.

Ex.

Phase 1 solution : Iamhappy

Phase 2 solution : 12345

Phase 3 solution : 2 23 4 5 2

Phase 4 solution : WOW!!

Phase 5 solution : what is this

Phase 6 solution : 12345 6 12

Then, in answer.txt file you have to write

Iamhappy

12345

2 23 4 5 2

WOW!!

what is this

12345 6 12

You can test your answer.txt in command line.

$./bomb answer.txt

To avoid accidentally detonating the bomb, you will need to learn how to single-step through the assembly code and how to set breakpoints. You will also need to learn how to inspect both the registers and the memory states. One of the nice side-effects of doing the lab is that you will get very good at using a debugger. This is a crucial skill that will pay big dividends the rest of your career.

Hints (Please read this!)

There are many ways of defusing your bomb. You can examine it in great detail without ever running the program, and figure out exactly what it does. This is a useful technique, but it not always easy to do. You can also run it under a debugger, watch what it does step by step, and use this information to defuse it. This is probably the fastest way of defusing it. We do make one request, please do not use brute force! You could write a program that will try every possible key to find the right one.

There are many tools which are designed to help you figure out both how programs work, and what is wrong when they don't work. Here is a list of some of the tools you may find useful in analyzing your bomb, and hints on how to use them.

**gdb, objdump

- objdump : It is for disassembling the binary code.

- gdb : The GNU debugger. It will be used for tracing through a program line by line,

examining memory and registers, set breakpoints and etc.

Disassemble binary file

You can disassemble binary file by objdump with option -d.

```
$objdump -d bomb
```

It will print the disassembled code of bomb. If you want to store the output of dump,

you can redirect the result using '>' or '>>' command. (If you don't know how, please

search on google "How to redirect output on unix system?")

You can print out the bomb's symbol table with option -t.

```
$objdump -t bomb
```

The symbol table includes the names of all functions and global variables in the bomb, the names of all the functions the bomb calls, and their addresses. You may learn something by looking at the function names!

Run program with gdb

The GNU debugger, this is a command line debugger tool available on virtually every platform. You can trace through a program line by line, examine memory and registers, look at both the source code and assembly code (we are not giving you the source code for most of your bomb), set breakpoints, set memory watch points, and write scripts.

Below command will execute gdb with the given program.

```
$gdb bomb
```

Then start the bomb running with the command:

```
(gdb) run answer.txt
```

To pause execution at a particular phase, use the following notation before running the bomb:

```
(gdb) b phase_1
```

To print out current position's disassembled code

```
(gdb) disas
```

To continue execution - This command continues execution after the breakpoint.

```
(gdb) c
```

To single step – execute exactly one instruction and then pause:

(gdb) si

To step over procedure call - If the instruction is a procedure call and you want the procedure to be called and return before pausing at the next instruction, use the ni command

(gdb) ni

To step out procedure

(gdb) finish

To print out the contents of registers

(gdb) info reg

To examine memory - Specify the starting address and the format for dumping the data. x/[number][unit][format] [memory address]

(gdb) x/32xb 0x400500

To print every variables which location pointed by current eip

(gdb) info locals

To print every global variables list for current state

(gdb) info variables

To quit gdb

(gdb) quit

If you need more information about gdb, you can search in Google. I will give you

some useful url about gdb : http://www.yolinux.com/TUTORIALS/GDB-Commands.html

Hint Table

Registers

64-bit register	Low 32 bits	Low 16 bits	Low 8 bits
%rax	%eax	%ax	%al

%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rbx	%ebx	%bx	%bl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rsp	%esp	%sp	%spl
%rbp	%ebp	%bp	%bpl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

Instruction Operands

Type	Form	Operand value
Immediate	$Imm	Imm
Register	r	R[r]
Memory	Imm(rb,ri,s)	M[Imm+R[rb]+R[ri]*s]

Instructions

Each of the following 64-bit (quadword) instructions has variants for byte operations (ending in b), 16-bit word operations (ending in w) and 32-bit longword operations (ending in l).

Instruction		Effect	Description
movq	S,D	D ← S	Move
xchgq	S,D	D ↔ S	Exchange S with D
incq	D	D ← D + 1	Increment
decq	D	D ← D – 1	Decrement
negq	D	D ← -D	Negate
notq	D	D ← ~D	Complement

addq	S,D	D ← D + S	Add
subq	S,D	D ← D – S	Subtract
imulq	S,D	D ← D * S	Signed Multiply
xorq	S,D	D ← D ^ S	Exclusive-or
andq	S,D	D ← D & S	And
orq	S,D	D ← D | S	Or
salq	k,D	D ← D << k	Shift left
shlq	k,D	D ← D << k	Shift left (same as salq)
sarq	k,D	D ← D >>A k	Arithmetic shift right
shrq	k,D	D ← D >>L k	Logical shift right
cmpq	S2,S1	CC ← compare(S1 - S2)	Compare S1 with S2
testq	S2,S1	CC ← test(S1 & S2)	Test bits
leaq	S,D	D ← &S	Load effective address. Does not set condition codes.
pushq	S	%rsp ← %rsp – 8 M[%rsp] ← S	Push
popq	D	D ← M[%rsp] %rsp ← %rsp + 8	Pop
movsbw	S,D	D ← signextend(S)	Move sign-extended byte to word
movsbl	S,D	D ← signextend(S)	Move sign-extended byte to longword
movswl	S,D	D ← signextend(S)	Move sign-extended word to longword
movsbq	S,D	D ← signextend(S)	Move sign-extended byte to quadword
movswq	S,D	D ← signextend(S)	Move sign-extended word to quadword
movslq	S,D	D ← signextend(S)	Move sign-extended longword to quadword
movzbw	S,D	D ← zeroextend(S)	Move zero-extended byte to word
movzbl	S,D	D ← zeroextend(S)	Move zero-extended byte to double word
movzwl	S,D	D ← zeroextend(S)	Move zero-extended word to double word
movzbq	S,D	D ← zeroextend(S)	Move zero-extended byte to quadword
movzwq	S,D	D ← zeroextend(S)	Move zero-extended word to quadword
movzlq	S,D	D ← zeroextend(S)	Move zero-extended longword to quadword
nop		No change	No operation

outb	S,D	IO[D] ← S	Output byte
inb	S,D	D ← IO[S]	Input byte

Jump instructions

Instruction		Synonyms	Description
call	label		Procedure call
call	*operand		Indirect procedure call
ret			Return from call
jmp	label		Direct jump
jmp	*operand		Indirect jump
je	label	jz	Equal/zero
jne	label	jnz	Not equal/not zero
js	label		Negative
jns	label		Non-negative
jg	label	jnle	Greater (signed >)
jge	label	jnl	Greater or equal (signed >=)
jl	label	jnge	Less (signed <)
jle	label	jng	Less or equal (signed <=)
ja	label	jnbe	Above (unsigned >)
jae	label	jnb	Above or equal (unsigned >=)
jb	label	jnae	Below (unsigned <)
jbe	label	jna	Below or equal (unsigned <=)

Set instructions

There is a set instruction corresponding to each conditional jump. Only a couple of examples are shown here. The set instructions use a single byte destination register D which is set to 1 if the condition is true.

Instruction		Synonym	Effect	Description
sete	D	setz	D ← ZF	Equal / zero
sets	D		D ← SF	Negative
setl	D	setnge	D ← SF ^ OF	Less (signed <)
setb	D	setnae	D ← CF	Below (unsigned <)

Conditional move

There is a conditional move corresponding to each conditional jump. In x64, conditional move instructions are available for 16-bit, 32-bit and 64-bit registers. The register names determine the size of the object being moved.

Instruction		Synonym	Effect	Description
cmove	S,D	cmovz	if (ZF) D ← S	Move if equal / zero
cmovs	S,D		if (SF) D ← S	Move if negative

ASCII Reference

Hex	Char	Hex	Char	Hex	Char	Hex	Char
0	NUL	20	Space	40	@	60	`
1	SOH	21	!	41	A	61	a
2	STX	22	"	42	B	62	b
3	ETX	23	#	43	C	63	c
4	EOT	24	$	44	D	64	d
5	ENQ	25	%	45	E	65	e
6	ACK	26	&	46	F	66	f
7	BEL	27	'	47	G	67	g
8	BS	28	(	48	H	68	h
9	TAB	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	-	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t

15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	{
1C	FS	3C	<	5C	₩	7C	|
1D	GS	3D	0	5D	]	7D	}
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	Delete

Linux procedure call conventions

Parameters: %rdi, %rsi, %rdx, %rcx, %r8, %r9. Additional parameters on stack.

Return value: %rax

Caller-save: Parameters and %rax, %r10, %r11

Callee-save: %rbx, %rbp, %r12, %r13, %r14, %r15

Special: %rsp

***This assignment is referenced from the following:*

1. *CMU system programming bomblab assignment*