Lab 13: PasswordLab Tuesday April 2nd

Collaboration: In lab, we encourage collaboration and discussion as you work through the problems. These activities, like recitation, are meant to get you to review what we've learned, look at problems from a different perspective and allow you to ask questions about topics you don't understand. We encourage discussing problems with other students in this lab!

Setup: Download the lab handout and code from the course website https://cs.cmu.edu/ 15122/handouts/code, and move it to your private directory in your unix.qatar.cmu.edu machine. Following that create a directory, move the handout to it, and unzip the handout file by executing the following commands:

```
% mkdir lab_13
% mv 13-handout.tgz lab_13
% cd lab_13
% tar -xvf 13-handout.tgz
```

Submission:
Create a tar file by executing the command below and submit it to autolab, under the lab name:

```
% tar cfzv handin.tgz answers.txt
```

Dr. Evil's passwords

Genius supervillian Dr. Evil is on the loose! Known for a series of devilishly tricky yet completely vulnerable assembly bombs, Dr. Evil has left a trail of destruction across Carnegie Mellon's undergraduate computer science curriculum. Authorities have been unable to track the whereabouts of this mastermind, but we have new intelligence on Dr. Evil's Super Secret Evil Plan™ to investigate.

You have been hired as an agent to crack the code of Dr. Evil's Super Secret Evil Plan. It seems that she left her secret plans in a password protected c0 binary file, accessible to you on the cluster computers by typing `evilplan`. She also accidentally left her C0VM bytecode in a public folder! She seems to have deleted most of the helpful comments, though, so we'll need help figuring out the passwords by hand. We were also able to acquire the `main` function's source code in `password-main.c0`, but it relies on functions that only appear in the bytecode file `password.bc0`.

You'll need to read through `password.bc0` to figure out some of the function calls — namely, the function calls `password1()`, `password2()`, `password3()`, etc.

Each of the password functions either takes in a password as input, and returns a boolean, or simply returns the password as an integer. Some passwords are numbers while others are strings. For the first four passwords the user's input is passed to `parse_int`, but for the last password the string is passed directly to the function. We've filled in the bytecode file with all the intelligence we have, so you'll have to figure out the rest.

A description of relevant C0VM bytecode instructions is given in appendix.

To check if you're correct, just run the password binary file, and type in the passwords you think are correct:

```
% evilplan
Welcome to Dr. Evil's Super Secret Evil Plan Terminal
This terminal should only be run by Dr. Evil to read the
Super Secret Evil Plan.
If you are anyone else, get OUT.
```

```
Password1:
```

(1.a) Dr. Evil's first password function seems pretty simple. It seems to return an integer. What is it?

1.5pt

(1.b) Dr. Evil's second password is a bit more complicated. It uses `vload` and `vstore` to store some local variables. Figure out what integer `password2` returns!

(1.c) Dr. Evil's third password is definitely more complicated. It uses `ildc` to load integers from the integer pool. What's going on there? For this password, note that returning `1` is equivalent to returning `true`, and returning `0` is equivalent to returning `false`.

3pt

(1.d) Dr. Evil's fourth password has a loop! The function jumps around, doing something to an integer input. What's the password?

(1.e) Dr. Evil's fifth and final password calls a helper function, func5. Figure out what it's doing, and crack the last password! The ASCII table to the right, which includes both integer and hex values, may come in handy.

4pt

(1.f) For the most clever of agents, Dr. Evil seems to have left a hidden 6th password. She didn't activate it in the source code file, which means it must have been so complicated even she didn't want to deal with it! Figure it out through the bytecode, and tell your TA if you think you got it.

Partial ASCII Table

32	20	␣	64	40	@	96	60	‘
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_			

Appendix: Selected C0VM bytecode reference

```
Stack operations
0x57 pop              S, v -> S
0x59 dup              S, v -> S, v, v
0x5F swap             S, v1, v2 -> S, v2, v1

Arithmetic
0x60 iadd             S, x:w32, y:w32 -> S, x+y:w32
0x64 isub             S, x:w32, y:w32 -> S, x-y:w32
0x68 imul             S, x:w32, y:w32 -> S, x*y:w32
0x6C idiv             S, x:w32, y:w32 -> S, x/y:w32
0x70 irem             S, x:w32, y:w32 -> S, x%y:w32
0x7E iand             S, x:w32, y:w32 -> S, x&y:w32
0x80 ior              S, x:w32, y:w32 -> S, x|y:w32
0x82 ixor             S, x:w32, y:w32 -> S, x^y:w32
0x78 ishl             S, x:w32, y:w32 -> S, x<<y:w32
0x7A ishr             S, x:w32, y:w32 -> S, x>>y:w32

Constants
0x10 bipush <b>       S -> S, x:w32       (x = (w32)b, sign extended)
0x13 ildc <c1,c2>     S -> S, x:w32       (x = int_pool[(c1<<8)|c2])
0x14 aldc <c1,c2>     S -> S, a:*         (a = &string_pool[(c1<<8)|c2])
0x01 aconst_null      S -> S, null:*

Local Variables
0x15 vload <i>        S -> S, v           (v = V[i])
0x36 vstore <i>       S, v -> S           (V[i] = v)

Assertions and errors
0xBF athrow           S, a:* -> S         (c0_user_error(a))
0xCF assert           S, x:w32, a:* -> S  (c0_assertion_failure(a) if x == 0)

Control Flow
0x00 nop              S -> S
0x9F if_cmpeq <o1,o2> S, v1, v2 -> S      (pc = pc+(o1<<8|o2) if v1 == v2)
0xA0 if_cmpne <o1,o2> S, v1, v2 -> S      (pc = pc+(o1<<8|o2) if v1 != v2)
0xA1 if_icmplt <o1,o2> S, x:w32, y:w32 -> S (pc = pc+(o1<<8|o2) if x < y)
0xA2 if_icmpge <o1,o2> S, x:w32, y:w32 -> S (pc = pc+(o1<<8|o2) if x >= y)
0xA3 if_icmpgt <o1,o2> S, x:w32, y:w32 -> S (pc = pc+(o1<<8|o2) if x > y)
0xA4 if_icmple <o1,o2> S, x:w32, y:w32 -> S (pc = pc+(o1<<8|o2) if x <= y)
0xA7 goto <o1,o2>     S -> S              (pc = pc+(o1<<8|o2))

Functions
0xB8 invokestatic <c1,c2>  S, v1, v2, ..., vn -> S, v
                          (function_pool[c1<<8|c2] => g, g(v1,...,vn) = v)
0xB0 return                ., v -> .      (return v to caller)
0xB7 invokenative <c1,c2>  S, v1, v2, ..., vn -> S, v
                            (native_pool[c1<<8|c2] => g, g(v1,...,vn) = v)

Memory
0xBB new <s>          S -> S, a:*         (*a is now allocated, size <s>)
0x2E imload           S, a:* -> S, x:w32  (x = *a, a != NULL, load 4 bytes)
0x4E imstore          S, a:*, x:w32 -> S  (*a = x, a != NULL, store 4 bytes)
0x2F amload           S, a:* -> S, b:*    (b = *a, a != NULL, load address)
0x4F amstore          S, a:*, b:* -> S    (*a = b, a != NULL, store address)
```

```
0x34 cmload         S, a:* -> S, x:w32   (x = (w32)(*a), a != NULL, load 1 byte)
0x55 cmstore        S, a:*, x:w32 -> S   (*a = x & 0x7f, a != NULL, store 1 byte)
0x62 aaddf <f>      S, a:* -> S, (a+f):* (a != NULL; f field offset in bytes)
0xBC newarray <s>   S, n:w32 -> S, a:*   (a[0..n) now allocated,
                                          each array element has size <s>)
0xBE arraylength    S, a:* -> S, n:w32   (n = \length(a))
0x63 aadds          S, a:*, i:w32 -> S, (a->elems+s*i):*
```