P1.1

A linguagem java e python impedem vulnerabilidades de *buffer*, logo, quando tentamos escrever mais que 10 elementos, num *buffer* cuja memória alocada apenas tem espaço para 10 elementos, invoca uma exceção, interrompendo a execução do programa.

Por outro lado, a linguagem de c++ não interrompe a execução do programa.

Como existe duas stacks a serem utilizadas, *initialized data segment* (data) e *uninitialized data segment* (bss), estas vão ser preenchidas dependendo do facto de as variáveis conterem valores ou não. Inicialmente preencherá do maior endereço para o menor, a *initialized data segment* com a variável “test”. De seguida preencherá a stack *uninitialized data segment* com o array “tests” e a variável “num_elems”.

Aquando a atribuição de valor à variável “num_elems”, esta vai passar para a stack de data, e será retirada da stack bss. Quando criamos a variável “i” com valor 0, esta será escrita na stack data. Quando atribuímos o primeiro valor ao array “tests”, este array será escrito abaixo da variável “i” na stack data. Como a escrita se realiza do menor endereço para o maior, se for escrito mais que 10 elementos, o 11 elemento irá escrever sobre a variável “i”, na stack data.

P1.2

Como explicado na pergunta 1, a linguagem java e python impedem vulnerabilidades de buffer.

A linguagem de c++, como não interrompe a execução do programa, caso tentemos escrever mais que 10 elementos, o programa irá para sempre correr, pois quando tenta escrever o 11 elemento, irá escrever sobre a variável “i” com o valor 0, retornando o ciclo ao ponto inicial, com “i” = 0.

P1.3

No ficheiro “RootExploit.c”, como a variável “pass”, variável que será testada para a atribuição de privilégios, foi atribuída o valor 0, esta será guardada na stack de dados. No entanto, o *buffer* onde será guardada a palavra passe foi guardado na stack *bss*, visto que não estava preenchido.

Quando se introduz a palavra passe no *buffer*, este será guardado na stack de dados, no endereço inferior à variável “pass”, mas como o buffer é preenchido do menor endereço para o maior, caso exista um *overflow* do *buffer*, irá escrever sobre a variável “pass”. Como o *buffer* apenas tem alocado memória para 4 caracteres, basta escrever uma palavra em que a 5 caracter é 1. A palavra passe não será a correta, mas serão atribuídas permissões de root/admin, pois o 5 caracter será escrito na variável “pass”.

No ficheiro “0-simple.c” ocorre o mesmo processo, para escrever sobre a variável “control”, é necessário escrever 64 caracteres, seguidos por um 1, de modo a escrever na variável “control”, o valor 1.

P1.4

Analisando e testando o código escrito em C presente em ReadOverflow.c, podemos observar os seguintes resultados:

```
Insira numero de caracteres: 10
Insira frase: 1234567890
ECO: |1234567890|
Insira numero de caracteres: 9
Insira frase: 1234567890
ECO: |123456789|
Insira numero de caracteres: 11
Insira frase: 1234567890
ECO: |1234567890.|
Insira numero de caracteres: 51
Insira frase: 123456789012345678901234567890123456789012345678901
ECO: |123456789012345678901234567890123456789012345678901|
Insira numero de caracteres: 51
Insira frase: 123456789012345678901234567890123456789012345678901
ECO: |123456789012345678901234567890123456789012345678901|
Insira numero de caracteres: 101
Insira frase: abcd
ECO: |abcd..78901234567890123456789012345678901234567890
Insira numero de caracteres:
```

```
Insira numero de caracteres: 10
Insira frase: 1234567890
ECO: |1234567890|
Insira numero de caracteres: 9
Insira frase: 1234567890
ECO: |123456789|
Insira numero de caracteres: 11
Insira frase: 1234567890
ECO: |1234567890.|
Insira numero de caracteres: 51
Insira frase: 123456789012345678901234567890123456789012345678901
ECO: |123456789012345678901234567890123456789012345678901|
Insira numero de caracteres: 51
Insira frase: 123456789012345678901234567890123456789012345678901
ECO: |123456789012345678901234567890123456789012345678901|
Insira numero de caracteres: 101
Insira frase: abcd
ECO: |abcd..789012345678901234567890123456789012345678901
```

Devido à maneira como o código está estruturado podemos verificar que se a string inserida tiver comprimento inferior ao número de caracteres indicado anteriormente, vai ser imprimida no ecrã informação em memória que não é a pretendida, este tipo de vulnerabilidade a não validação do comprimento dos inputs resulta em falhas de segurança graves como a Heartbleed em que era indicado um número superior de bytes ao utilizado e na resposta, um atacante conseguiria obter informações da memória que não as que lhe era permitido obter.

Quando é passado um número e uma string maiores que o que o buffer tem capacidade, o número de caracteres do ciclo seguinte, irá ser obtido do que ficou em memória não sendo inserido pelo utilizador.

Como verificamos na última imagem foi possível aceder a uma zona da memória que não a do programa e obter informação da memória imprimida na consola.

Uma maneira de mitigar esta vulnerabilidade seria validar o input da string, ou seja,

após ser indicado o número de caracteres, o programa apenas iria buscar do utilizador esse número de caracteres, e após a string ser inserida, era verificado o seu tamanho e apenas seriam imprimidos na consola o número de caracteres correspondentes ao comprimento da string, garantindo assim que não haveria acessos a zonas de memória não permitidas.

P1.5

Observando o código da função "1-match.c" entende-se que é necessário alterar o valor do inteiro "control" para 0x6162364 de modo a "vencer" o jogo.

Inicialmente temos de compilar o código:

gcc -o 1match 1-match.c

Tendo em conta a maneira como a stack funciona sabemos que o buffer tem 64bytes(sendo 1 byte o espaço alocado em memória para cada caracter), o endereço do buffer ocupa 8 bytes, assim apesar de termos pedido apenas para reservar 64bytes para o buffer este ocupa 72bytes em memória de seguida teremos o endereço de retorno ocupando 4 bytes e depois o espaço reservado para o valor da variável control, assim teremos de escrever 76bytes antes de conseguirmos escrever em memória na variável control.

O valor a obter é 0x61626364, tendo em conta a tabela ascii isto será o mesmo que a string "abcd" (ver na tabela na resposta à pergunta P1.6), uma vez que os processadores intel trabalham em little-endian, ou seja o primeiro valor a ser passado é o menos significativo esta string teria de ser escrita em memória na ordem inversa: "dcba", assim para vencermos este jogo temos de chamar o programa com um argumento de 76 bytes seguido da string em questão.

Deste modo executamos o programa da seguinte forma:

./1match
1234567890123456789012345678901234567890123456789012345678901234567890123
456dcba

```
user@CSI:~/Aulas/Aula10/codigofonte$ ./1match 12345678901234567890123456789012345
678901234567890123456789012345678901234567890123456dcba
You win this game if you can change variable control to the value 0x61626364'
Congratulations, you win!!! You correctly got the variable to the right value
user@CSI:~/Aulas/Aula10/codigofonte$
```

P1.6

Observando o código da função "*2-functions.c*" entende-se que é necessário alterar o valor da variável **fp** para um valor que seja diferente de 0. Contudo, visto que o objetivo final é invocar a função "*win*", é necessário ainda que a execução desta variável na linha 27 seja substituída por uma invocação à função "*win*". Assim, podemos alterar o valor da variável para o próprio endereço de memória da função que se pretende invocar, fazendo,

consequentemente, com que o seu valor seja diferente de 0. De seguida demonstram-se todos os passos necessários para conseguir imprimir no ecrã a mensagem "*Congratulations, you win!*".

1. Primeiro que tudo, é necessário compilar o código. Para tal utiliza-se a ferramenta **gcc**, sendo que se executa o seguinte comando na diretoria onde se encontra o documento a compilar:

   gcc -o 2-functions 2-functions.c -g

2. Com o código compilado num programa executável, utilizemos agora o *debugger* para proceder à sua execução, auxiliando a identificar endereços de memória importantes. Para isto utilize-se:

   gdb 2-functions

3. O primeiro passo já no interior do *debugger* será identificar o endereço de memória no qual a função de "*win*" se encontra, isto pode ser efetuado com o comando:

   p win

   Este produz um *output* contendo o endereço desta mesma função. O *output* pode ser observado na figura seguinte.

```
(gdb) p win
$1 = {<text variable, no debug info>} 0x555555554740 <win>
```

4. Após anotado o endereço de memória da função que se pretende invocar, podemos proceder à execução deste. Para tal utiliza-se o comando **run**. Este executa a função, sendo que é então pedido ao utilizador a introdução de uma *string*. Como demonstra a figura seguinte.

```
(gdb) run
Starting program: /home/user/1718-EngSeg/TPraticas/Aula10/codigofonte/2-functions
You win this game if you are able to call the function win.'
```

5. A introdução desta *string* é exatamente um dos aspetos mais importantes para explorar esta vulnerabilidade. Sabemos que o *buffer* tem um tamanho de 64 bytes, e que se aloca o valor da variável *fd* nos 4 bytes seguintes, mais 4 para o *base pointer*, até escrevemos no endereço de retorno. Assim, para substituir o endereço de memória para a qual a variável *fd* aponta é necessário escrever 72 bytes.

   Sabemos assim que a nossa *string* necessita de ter um tamanho igual a 72 até que se possa atribuir o valor do endereço de memória. Como estes valores são apresentados em bytes e em valores hexadecimais, é necessário introduzir os caracteres corretos para assegurar que a tradução para endereço seja correta. Para tal pode-se recorrer a uma tabela *ASCII*. Tenhamos a seguinte tabela em mente.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

6. Sabemos que o valor que estamos a tentar atribuir é correspondente a: **0x555555554740**, este pode ser expresso pela tabela *ASCII* supracitada, como: **UUUUG@**.

   Sabendo ainda que nas arquiteturas Intel atuais, encontramo-nos num modelo *little-endian*, é necessário introduzir este valor pela ordem inversa, sendo que a *string* final será algo como 72 caracteres quaisquer, seguidos por **@GUUUU**.

   Procedeu-se então à sua introdução no programa, e verificou-se que a função "*win*" tinha sido invocada sem alterar código, como pode ser visível na figura seguinte.

```
gdb-peda$ run
Starting program: /home/user/1718-EngSeg/TPraticas/Aula10/codigofonte/2-functions
You win this game if you are able to call the function win.'
123456789012345678901234567890123456789012345678901234567890123456789012@GUUUU
calling function pointer, jumping to 0x555555554740
Congratulations, you win!!! You successfully changed the code flow
[Inferior 1 (process 18715) exited normally]
Warning: not running or target is remote
```

P1.7

O processo de resolução desta pergunta é exatamente igual à anterior, no entanto deparamo-nos com o problema de que a posição de memória da função "*win*" ser diferente, visto que desta vez é: **0x5555555546f0**. Isto apresenta um problema, pois o caracter *ASCII* 240, equivalente a F0 não existe. Tentaram-se várias abordagens, mas nenhuma conseguiu modificar efetivamente o valor do registo **%ip** de modo a que este apontasse para a função desejada. Desta forma conclui-se que a única maneira de resolver este problema seria

alterar o código (inserindo um print do próprio buffer, por exemplo depois da função *gets*), levando a que o endereço da função "*win*" voltasse a ser o mesmo que apresentava na pergunta supracita. Neste caso, a *string* de *input* seria exatamente a mesma.