SSD **Name:** _____
Winter Term 2020
Exam 1 **Immatriculation Number:** _____
18.12.2020
Time Limit: 30 Minutes

You may *not* use your books, notes, or any additional material on this exam.

Since this is an oral exam with a limited time frame, the student will develop the answers in interaction with the examiner.

Question	Points	Score
1	10	
2	10	
3	10	
4	10	
Total:	40	

1. (10 points) **Memory Safety**

   You find the following code snippet:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    char* a = malloc(8);
    strcpy(a, "AAAA");
    char* b = malloc(8);
    strcpy(b, "BBBB");
    realloc(a, 32);
    char* c = malloc(8);
    strcpy(a, "aaaaaa");
    strcpy(c, "CCCC");

    printf("%s\n", a);
}
```

   (a) (2 points) What is/are the bug(s) in the program? Pinpoint code location(s) and type of the bug(s).

   (b) (2 points) What is a potential output of this program?

   (c) (4 points) Sketch the heap after each allocation (i.e. malloc and realloc).

   (d) (2 points) How would you fix the program?

2. (10 points) **Exploits**

I found a 32-bit strange-looking code snippet on my system and decompiled it:

```
00000000  6631C0            xor eax,eax
00000003  6650              push eax
00000005  66682F7A7368      push dword 0x68737a2f
0000000B  66682F62696E      push dword 0x6e69622f
00000011  6689E3            mov ebx,esp
00000014  6631C9            xor ecx,ecx
00000017  6631D2            xor edx,edx
0000001A  B00B              mov al,0xb
0000001C  CD80              int 0x80
0000001E  6631C0            xor eax,eax
00000021  B001              mov al,0x1
00000023  CD80              int 0x80
```

(a) (1 point) Which register holds the syscall number in the x86_32 Linux calling convention?

(b) (1 point) Which Linux syscall is used to load and run an ELF binary?

(c) (3 points) Sketch the stack layout.

(d) (4 points) What does the above snippet do? How is such a code snippet usually called?

(e) (1 point) How would you prevent this issue?

3. (10 points) **Finding Bugs**

A customer called you at 11pm to ask what has happened to his binary. He managed to run it with the address sanitizer ASAN. Here's the output he sent you.

```
==2671==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffd5cc814ea at pc 0
    x7fbaba6f7709 bp 0x7ffd5cc814a0 sp 0x7ffd5cc80c48
WRITE of size 11 at 0x7ffd5cc814ea thread T0
    #0 0x7fbaba6f7708  (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x62708)
    #1 0x400924 in main (/home/ssd/main+0x400924)
    #2 0x7fbaba2eb83f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2083f)
    #3 0x400788 in _start (/home/ssd/main+0x400788)

Address 0x7ffd5cc814ea is located in stack of thread T0 at offset 42 in frame
    #0 0x400865 in main (/home/ssd/main+0x400865)

  This frame has 1 object(s):
    [32, 42) 'password' <== Memory access at offset 42 overflows this variable
...
SUMMARY: AddressSanitizer: stack-buffer-overflow ??:0 ??
Shadow bytes around the buggy address:
  0x10002b988280: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x10002b988290: 00 00 00 00 00 00 00 00 f1 f1 f1 f1 00[02]f4 f4
  0x10002b9882a0: f3 f3 f3 f3 00 00 00 00 00 00 00 00 00 00 00 00
  0x10002b9882b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa        Stack use after scope:   f8
  Heap right redzone:      fb        Global redzone:          f9
  Freed heap region:       fd        Global init order:       f6
  Stack left redzone:      f1        Poisoned by user:        f7
  Stack mid redzone:       f2        Container overflow:      fc
  Stack right redzone:     f3        Array cookie:            ac
  Stack partial redzone:   f4        Intra object redzone:    bb
  Stack after return:      f5        ASan internal:           fe
```

(a) (5 points) Provide a small code snippet that could lead to this kind of crash.

(b) (2 points) Which bugs can be detected by ASAN?

(c) (2 points) What are limitations of ASAN?

(d) (1 point) How does ASAN ensure it catches those bugs?

4. (10 points) **Defensive Programming**

```c
#include <stdio.h>
#include <assert.h>

#define PRINT_ERROR(msg, err) fprintf(stderr, \
        "%s threw an error: %s, returning %d\n", \
        __func__, (msg), (int)(err)); \
    return (err);

int main(int argc, char** argv) {
    assert(argc >= 3);
    FILE* f = fopen(argv[1], "w");
    if (!f) { PRINT_ERROR("Unable to open file", -1); }
    // ...
    assert(f != NULL);
    assert(argc >= 3);
    assert(fprintf(f, "Your input: %s\n", argv[2]) > 0);
    // ...
    assert(f != NULL);
    fclose(f);
    return 0;
}
```

(a) (8 points) Explain four defensive programming principles which are related to the above code. Were these principles followed or not?

(b) (1 point) Describe what the purpose of asserts is and how they shall be used

(c) (1 point) How can compilers assist in improving code quality?

Appendix: ASCII Table

Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char
0x00	0	NULL (null)	0x20	32	space	0x40	64	@	0x60	96	‘
0x01	1	SOH (start of heading)	0x21	33	!	0x41	65	A	0x61	97	a
0x02	2	STX (start of text)	0x22	34	"	0x42	66	B	0x62	98	b
0x03	3	ETX (end of text)	0x23	35	#	0x43	67	C	0x63	99	c
0x04	4	EOT (end of transmission)	0x24	36	$	0x44	68	D	0x64	100	d
0x05	5	ENQ (enquiry)	0x25	37	%	0x45	69	E	0x65	101	e
0x06	6	ACK (acknowledge)	0x26	38	&	0x46	70	F	0x66	102	f
0x07	7	BELL (bell)	0x27	39	’	0x47	71	G	0x67	103	g
0x08	8	BS (backspace)	0x28	40	(	0x48	72	H	0x68	104	h
0x09	9	TAB (horizontal tab)	0x29	41	)	0x49	73	I	0x69	105	i
0x0a	10	LF (new line)	0x2a	42	*	0x4a	74	J	0x6a	106	j
0x0b	11	VT (vertical tab)	0x2b	43	+	0x4b	75	K	0x6b	107	k
0x0c	12	FF (form feed)	0x2c	44	,	0x4c	76	L	0x6c	108	l
0x0d	13	CR (carriage return)	0x2d	45	-	0x4d	77	M	0x6d	109	m
0x0e	14	SO (shift out)	0x2e	46	.	0x4e	78	N	0x6e	110	n
0x0f	15	SI (shift in)	0x2f	47	/	0x4f	79	O	0x6f	111	o
0x10	16	DLE (data link escape)	0x30	48	0	0x50	80	P	0x70	112	p
0x11	17	DC1 (device control 1)	0x31	49	1	0x51	81	Q	0x71	113	q
0x12	18	DC2 (device control 2)	0x32	50	2	0x52	82	R	0x72	114	r
0x13	19	DC3 (device control 3)	0x33	51	3	0x53	83	S	0x73	115	s
0x14	20	DC4 (device control 4)	0x34	52	4	0x54	84	T	0x74	116	t
0x15	21	NAK (negative ack)	0x35	53	5	0x55	85	U	0x75	117	u
0x16	22	SYN (synchronous idle)	0x36	54	6	0x56	86	V	0x76	118	v
0x17	23	ETB (end transmission)	0x37	55	7	0x57	87	W	0x77	119	w
0x18	24	CAN (cancel)	0x38	56	8	0x58	88	X	0x78	120	x
0x19	25	EM (end of medium)	0x39	57	9	0x59	89	Y	0x79	121	y
0x1a	26	SUB (substitute)	0x3a	58	:	0x5a	90	Z	0x7a	122	z
0x1b	27	FSC (escape)	0x3b	59	;	0x5b	91	[	0x7b	123	{
0x1c	28	FS (file separator)	0x3c	60	<	0x5c	92	\	0x7c	124	|
0x1d	29	GS (group separator)	0x3d	61	=	0x5d	93	]	0x7d	125	}
0x1e	30	RS (record separator)	0x3e	62	>	0x5e	94	^	0x7e	126	~
0x1f	31	US (unit separator)	0x3f	63	?	0x5f	95	_	0x7f	127	DEL

Appendix: C Function Reference

This appendix provides a short summary of C library functions used in the code snippets. The descriptions are partly taken from "The C Library Reference Guide" by Eric Huss.

fopen: `FILE *fopen(const char *path, const char *mode)`

Opens the filename pointed to by filename. The mode argument may be one of the following constant strings:

"r" read text mode

"w" write text mode (truncates file to zero length or creates new file)

"r+" read and write text mode

On success a pointer to the file stream is returned. On failure a null pointer is returned.

fprintf: `int fprintf(FILE *stream, const char *format, ...)`

Performs printf functionality on the provided FILE stream (instead of stdio) and returns the number of characters written, excluding the string null terminator. On error, a negative value is returned.

fclose: `int fclose(FILE *stream)`

Closes the stream. All buffers are flushed. If successful, it returns zero. On error it returns EOF.

strcpy: `char *strcpy(char *str1, const char *str2)`

Copies the string pointed to by str2 to str1. Copies up to and including the null character of str2. If str1 and str2 overlap the behavior is undefined. Returns the argument str1.

strncpy: `char *strncpy(char *str1, const char *str2, size_t n)`

Copies up to n characters from the string pointed to by str2 to str1. Copying stops when n characters are copied or the terminating null character in str2 is reached. If the null character is reached, the null characters are continually copied to str1 until n characters have been copied. Returns the argument str1.

malloc: `void *malloc(size_t size)`

Allocates the requested memory and returns a pointer to it. The requested size is size bytes. The value of the space is indeterminate. On success a pointer to the requested space is returned. On failure a null pointer is returned.

realloc: `void *realloc(void *ptr, size_t size)`

Attempts to resize the memory block pointed to by ptr that was previously allocated with a call to malloc or calloc. The contents pointed to by ptr are unchanged. If the value of size is greater than the previous size of the block, then the additional bytes have an undeterminate value. If the value of size is less than the previous size of the block, then the difference of bytes at the end of the block are freed. On success a pointer to the memory block is returned (which may be in a different location as before). On failure or if size is zero, a null pointer is returned.

system: `int system(const char *string)`

The command specified by string is passed to the host environment to be executed by the command processor. A null pointer can be used to inquire whether or not the command processor exists. If string is a null pointer and the command processor exists, then zero is returned. All other return values are implementation-defined.

printf: `int printf(const char *format, ...)`

This function takes the format string specified by the format argument and applies each following argument to the format specifiers in the string in a left to right fashion. Each character in the format string is copied to the stream except for conversion characters which specify a format specifier.

execv: `int execv(const char *path, char *const argv[])`

Replaces the current process image with a new process image specified in path. The execv() function provide an array of pointers (argv) to null-terminated strings that represent the argument list available to the new program. The first argument should point to the filename associated with the file being executed. The array of pointers must be terminated by a null pointer.

Appendix: 32-bit Linux Syscall List

Nr.	Name	EAX	EBX	ECX	EDX	ESI	EDI
1	sys_exit	0x01	int exit_code	-	-	-	-
2	sys_fork	0x02	-	-	-	-	-
3	sys_read	0x03	unsigned int fd	char *buf	size_t count	-	-
4	sys_write	0x04	unsigned int fd	const char *buf	size_t count	-	-
5	sys_open	0x05	const char *filename	int flags	int mode	-	-
6	sys_close	0x06	unsigned int fd	-	-	-	-
7	sys_waitpid	0x07	pid_t pid	int *stat_addr	int options	-	-
8	sys_creat	0x08	const char *pathname	int mode	-	-	-
9	sys_link	0x09	const char *oldname	const char *newname	-	-	-
10	sys_unlink	0x0a	const char *pathname	-	-	-	-
11	sys_execve	0x0b	const char *filename	const char **argv	const char **envp	-	-
12	sys_chdir	0x0c	const char *filename	-	-	-	-
13	sys_time	0x0d	time_t *tloc	-	-	-	-
14	sys_mknod	0x0e	const char *filename	int mode	unsigned dev	-	-
15	sys_chmod	0x0f	const char *filename	mode_t mode	-	-	-
16	sys_lchown16	0x10	const char *filename	old_uid_t user	old_gid_t group	-	-
19	sys_lseek	0x13	unsigned int fd	off_t offset	unsigned int origin	-	-
20	sys_getpid	0x14	-	-	-	-	-
26	sys_ptrace	0x1a	long request	long pid	long addr	long data	-
37	sys_kill	0x25	int pid	int sig	-	-	-
88	sys_reboot	0x58	int magic1	int magic2	unsigned int cmd	void *arg	-
125	sys_mprotect	0x7d	unsigned long start	size_t len	unsigned long prot	-	-

Appendix: 64-bit Linux Syscall List

Nr.	Name	RAX	RDI	RSI	RDX	R10	R8
0	sys_read	0x00	unsigned int fd	char *buf	size_t count	-	-
1	sys_write	0x01	unsigned int fd	const char *buf	size_t count	-	-
2	sys_open	0x02	const char *filename	int flags	int mode	-	-
3	sys_close	0x03	unsigned int fd	-	-	-	-
10	sys_mprotect	0x0a	unsigned long start	size_t len	unsigned long prot	-	-
59	sys_execve	0x3b	const char *filename	const char **argv	const char **envp	-	-
60	sys_exit	0x3c	int exit_code	-	-	-	-