University of Stuttgart
Institute of
Information Security

System and Web Security
Summer Term 2018
Prof. Dr. Ralf Küsters

Homework 4

Submission via ILIAS until May 15, 2018, 07:59.

General Notes

- If you encounter difficulties, you SHOULD[1] ask the teaching assistants in the consultation hour.
- To solve the homework, you SHOULD form teams of 3 people.
- Your team size MUST NOT exceed 3 people.
- You MUST submit your homework in ILIAS. (In the ILIAS course, you can find the *Homework Submissions* item. There, you can register your team and submit your solution.)
- You are free to choose whether you write your solutions in German or in English.
- Your submission MUST include all team member's names and matriculation numbers.
- You MUST use the System Security Virtual Machine to work on all problems (except if stated otherwise).
- You MUST submit your solution as a PDF or TXT document. If your solution contains source code, you MUST submit your source code in its native file format separately. In your main document, it is sufficient to point to these source code files.
- When you carry out attacks, you SHOULD take a screenshot to document that your exploit worked (and include this screenshot in your submission).
- Your team gains one point for each problem if solved reasonably.
- If you do not adhere to these rules, you risk loosing points.

Information for Solving the Problems

- Many problems of the homework depend on the computer architecture you use. In ILIAS, you can download the *System Security Virtual Machine (VM)*. This VM serves as our reference system. Other systems may (or most likely will) produce different results that we cannot grade as correct. **You MUST use this VM to work on all problems (except if stated otherwise).**
- Please note that you can find further information on how to install the VM in ILIAS.
- When you compile source code, you MUST use `gcc` with the following options:

  -fno-stack-protector disables several defence mechanisms

  -O0 disables optimizations

  -ggdb3 adds meta information for the debugger to the binary

  Although we have configured the shell of the VM such that the `gcc` command uses these options automatically, you should not omit these (especially when writing your own make files, etc.). For example, you compile a C program (contained in file `myprog.c`) like this:

```
gcc -fno-stack-protector -O0 -ggdb3 -o myprog myprog.c
```

[1]SHOULD, MUST, and MUST NOT are used as defined in RFC2119.

Problem 1: Off-by-One Exploit

If a buffer can be overflown by one byte (if, for example, <= is used instead of < in loops), an attacker can use this for a so-called *off-by-one exploit*. Inform yourself about off-by-one exploits. See, for example, Section 8 of *Buffer Overflows for Dummies*.[2]

In the program below, it is possible to overwrite the least significant byte of the saved frame pointer in the stack frame of `b`. Use a suitable input for the program to change this byte such that after the program returns from the function `b`, the program continues running in a forged stack frame that has the address of the shellcode as the return address. Document your attack (including screenshots).

Hint: You do not need to write a program that generates an exploit for this task. You can provide your input string directly in the shell. For this, you need to properly escape this string (for the shell). One way to escape your string is to use the `echo -e` command. You can, for example, pass your string as in the following command: `./obo-exploit-1-target $(echo -e "abc\x78...")`

Another hint: You can add data to the environment or add additional arguments in order to "shift" the stack to lower memory addresses if needed.

File: code/obo–exploit–1–target.c

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

static char shellcode_with_nops[] = "\x90\x90\x90\x90\x90\x90\x90"
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

int i;

void b(char *input) {
  char buffer[32];
  printf("Address of buffer in exploitable program: %p\n", buffer);
  printf("Framepointer before copy: %x\n", *((int*)buffer+8));
  for (i=0; i<= 32; i++) {
    buffer[i] = input[i];
  }
  printf("Framepointer after copy: %x\n", *((int*)buffer+8));
}

void a(char *input) {
  b(input);
}

main(int argc, char *argv[]) {
  printf("Shellcode at: %p\n", shellcode_with_nops);
  if (argc > 1)
    a(argv[1]);
}
```

[2]See https://www.sans.org/reading-room/whitepapers/threats/buffer-overflows-dummies-481

Problem 2: BSS-segment-based Overflow with Function Pointer

For the C program below, find a String `str` such that the command `./bss-exploit-2-target $(echo -e "str")` opens a shell (`/bin/sh`). As usual, provide a screenshot of the successful attack.

Hint: Use the same technique as in *w00w00 on Heap Overflows*, Lines 556–603. Overwrite error_handler with the address of the method system.

File: code/bss–exploit–2–target.c

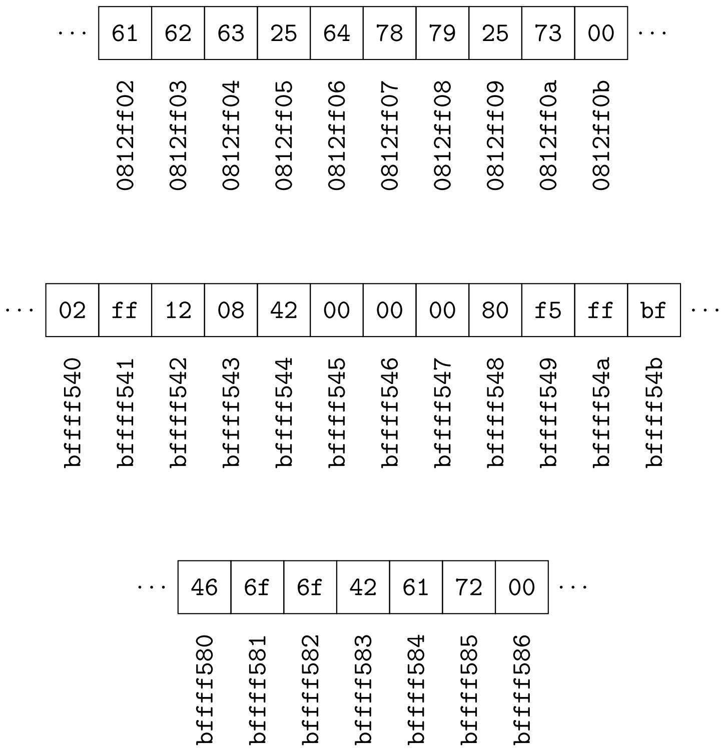
```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int print_error (const char * str);
main(int argc, char * argv[]) {
  static int (*error_handler)(const char * str);
  static char buf[12];
  error_handler = (int (*)(const char * str))print_error;
  if (argc < 2) {
    printf ("Please provide an argument\n");
    return 1;
  }
  strcpy(buf, argv[1]);
  if (buf[0] != '/') {
    (void)(*error_handler)(argv[1]);
  } else {
    printf ("Everything is OK\n");
  }
  printf ("Addr of system:  %p\n", system);
}
int print_error (const char * str) {
  printf("Not a valid input:  %s\n", str);
  return 0;
}
```

Problem 3: printf

Three excerpts of a program's memory are shown in Figure 1. At address `0xbffff540`, there are three arguments for a `printf` call, i.e., `printf` expects a pointer to a format string at address `0xbffff540` followed by more arguments.

All values in Figure 1 are shown hexadecimal. In particular, string characters are provided in their hexadecimal representation. An ASCII table is provided in Figure 2.

Your task is to play the role of `printf` step-by-step. Use the file `fstr-result-table.txt` as a template to write down your steps. After each step, write down the addresses where the printf-internal loop pointer and where the printf-internal stack pointer points to and what the (complete) output up to this step is (compare Slide Set 3, Slides 8ff.).



···	61	62	63	25	64	78	79	25	73	00	···
	0812ff02	0812ff03	0812ff04	0812ff05	0812ff06	0812ff07	0812ff08	0812ff09	0812ff0a	0812ff0b	

···	02	ff	12	08	42	00	00	00	80	f5	ff	bf	···
	bffff540	bffff541	bffff542	bffff543	bffff544	bffff545	bffff546	bffff547	bffff548	bffff549	bffff54a	bffff54b	

···	46	6f	6f	42	61	72	00	···
	bffff580	bffff581	bffff582	bffff583	bffff584	bffff585	bffff586	

Figure 1: Memory Excerpts.

Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char
20	␣	30	0	40	@	50	P	60	‘	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	¡	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	¿	4E	N	5E	ˆ	6E	n	7E	˜
2F	/	3F	?	4F	O	5F	_	6F	o		

Figure 2: ASCII Table (abbreviated).