Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Computer Science and
Information Technology

OSSEC-Labor: Einleitung Buffer Overflow, Teil 2

FH Zentralschweiz

Inhalt Heute

- Zeichenketten (Strings)
- amd64-Instruktionssatz
- Calling Convention und Aufbau des Stacks

RECAP: DATEN IM SPEICHER

Recap: Adressen

- CPU findet Daten und Instruktionen im Speicher über **Adressen**
- Adressen zeigen immer auf ein Byte

Recap: Adressen

- CPU findet Daten und Instruktionen im Speicher über **Adressen**
- Adressen zeigen immer auf ein Byte

- Wie adressiert man Multi-Byte-Werte? Strings? Objekte?
- Wie sehen diese im Speicher aus?

Adressierung

- Wie adressiert man Multi-Byte-Werte? Strings? Objekte?

Adressen zeigen immer auf das erste (nullte) Byte davon.

Werte im Speicher

- Wie sehen diese im Speicher aus?

- Text als Zeichenketten: Wir schauen uns C-Strings an
- «zero-terminated char arrays»

TEXT IM SPEICHER

Strings (Zeichenketten)

- Text wird als Folge von chars im Speicher abgelegt (Zeichenkette)
- char = 1-Byte-Wert
- Englisch «character» = «Zeichen»

- Folge von 1-Byte-Werten: «char array»

Encoding

- Welcher Wert im Byte steht für welchen Buchstaben?
- Encoding bestimmt das!
- ASCII und darauf aufbauende Encodings.

Encoding: ASCII

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	*[NULL]*	32	20	*[SPACE]*	64	40	@	96	60	`
1	1	*[START OF HEADING]*	33	21	!	65	41	A	97	61	a
2	2	*[START OF TEXT]*	34	22	"	66	42	B	98	62	b
3	3	*[END OF TEXT]*	35	23	#	67	43	C	99	63	c
4	4	*[END OF TRANSMISSION]*	36	24	$	68	44	D	100	64	d
5	5	*[ENQUIRY]*	37	25	%	69	45	E	101	65	e
6	6	*[ACKNOWLEDGE]*	38	26	&	70	46	F	102	66	f
7	7	*[BELL]*	39	27	'	71	47	G	103	67	g
8	8	*[BACKSPACE]*	40	28	(	72	48	H	104	68	h
9	9	*[HORIZONTAL TAB]*	41	29	)	73	49	I	105	69	i
10	A	*[LINE FEED]*	42	2A	*	74	4A	J	106	6A	j
11	B	*[VERTICAL TAB]*	43	2B	+	75	4B	K	107	6B	k
12	C	*[FORM FEED]*	44	2C	,	76	4C	L	108	6C	l
13	D	*[CARRIAGE RETURN]*	45	2D	-	77	4D	M	109	6D	m
14	E	*[SHIFT OUT]*	46	2E	.	78	4E	N	110	6E	n
15	F	*[SHIFT IN]*	47	2F	/	79	4F	O	111	6F	o
16	10	*[DATA LINK ESCAPE]*	48	30	0	80	50	P	112	70	p
17	11	*[DEVICE CONTROL 1]*	49	31	1	81	51	Q	113	71	q
18	12	*[DEVICE CONTROL 2]*	50	32	2	82	52	R	114	72	r
19	13	*[DEVICE CONTROL 3]*	51	33	3	83	53	S	115	73	s
20	14	*[DEVICE CONTROL 4]*	52	34	4	84	54	T	116	74	t
21	15	*[NEGATIVE ACKNOWLEDGE]*	53	35	5	85	55	U	117	75	u
22	16	*[SYNCHRONOUS IDLE]*	54	36	6	86	56	V	118	76	v
23	17	*[ENG OF TRANS. BLOCK]*	55	37	7	87	57	W	119	77	w
24	18	*[CANCEL]*	56	38	8	88	58	X	120	78	x
25	19	*[END OF MEDIUM]*	57	39	9	89	59	Y	121	79	y
26	1A	*[SUBSTITUTE]*	58	3A	:	90	5A	Z	122	7A	z
27	1B	*[ESCAPE]*	59	3B	;	91	5B	[	123	7B	{
28	1C	*[FILE SEPARATOR]*	60	3C	<	92	5C	\	124	7C	|
29	1D	*[GROUP SEPARATOR]*	61	3D	=	93	5D	]	125	7D	}
30	1E	*[RECORD SEPARATOR]*	62	3E	>	94	5E	^	126	7E	~
31	1F	*[UNIT SEPARATOR]*	63	3F	?	95	5F	_	127	7F	*[DEL]*

C-Strings

- Wie werden Strings in Programmen referenziert?
- Wir nehmen die Adresse des ersten (nullten) Chars!

- Woher wissen wir, wann der String fertig ist?
- Am Ende des Strings hängen wir ein Null-Byte (0x00) an.
- Dieser Wert ist kein gültiges Zeichen innerhalb des Strings
- Länge = Anzahl Bytes vor dem Null-Byte (ohne dieses)

- Damit spart man sich das Speichern einer Länge und die Bytes, in denen die Länge steht
- Ein Längenfeld hätte einen Maximalwert – was ist mit Strings, die länger sind als das?

C-Strings: Beispiel

Im Speicher steht diese Byte-Folge:
0x48, 0x61, 0x6c, 0x6c, 0x6f, 0x00

ASCII-Tabelle sagt uns:
'H', 'a', 'l', 'l', 'o', [NUL]

Es handelt sich also um den String "Hallo"!
Wichtig: Ohne Null-Byte am Ende wäre es kein gültiger String.

AMD64-INSTRUKTIONSSATZ

Assembly-Code: amd64 = x86_64 = 64bit-x86

- Instruktionssatz von modernen AMD- und Intel-CPUs
- Assembly kommt in NASM oder AT&T-Syntax

- Wichtig für das Lab: Wie sind diese aufgebaut

Wiederholung: Von Neumann-Maschine

- CPU rechnet mit Werten in Registern oder Konstanten = Immediate
- RAM enthält Daten
- Werte im RAM werden über Adressen referenziert

amd64: Assembly-Syntax

```
  mov     $0xe,%esi  source immediate, target register
  mov     %rax,%rdi  source register, target register
  callq   1040 <fgets@plt>
instruction   instruction parameters
name
```

- Links steht der Name der Instruktion
- Danach folgen Parameter

Parameter-Typen:

- Register – z.B. `%rax`
- Immediate – z.B. `$0xe` (Dezimal 14)
- Adressen – z.B. `1040`

  das `<fgets@plt>` sagt, was an dieser Adresse steht

amd64: Assembly-Syntax

```
mov     0x2ec8(%rip),%rdx
lea     -0x16(%rbp),%rax  LEA memory address syntax
```

Komplexere Parameter:

- LEA-Syntax = Scale-Index-Base-Offset – z.B. `-0x16(%rbp)`

- Wir wollen nicht den Wert in einem Register
  sondern einen Wert im Hauptspeicher
- Aber die ungefähre Adresse zum Wert steht in einem Register

- Der Wert ist um einen Offset von der Adresse im Register entfernt

amd64: Assembly-Syntax

```
mov     0x2ec8(%rip),%rdx
lea     -0x16(%rbp),%rax  LEA memory address syntax
```

Komplexere Parameter:

- `(%rbp)` Wert steht an der Adresse im Register `%rbp`
- `-0x16(%rbp)` Offset: Adresse in `%rbp` minus `0x16` Bytes

- `-0x16(%rbp,%rdi,4)` – Komplexere Strukturen

amd64: Funktionsaufruf

```
lea    0xe98(%rip),%rdi
callq  1030 <puts@plt>  function call
```

Call = Funktionsaufruf

- `<puts@plt>` – die externe Funktion puts() wird aufgerufen

- Funktionen kommen aus Bibliotheken («Libraries»)
- Wichtig für uns: C Standard Library (libc)
- Darin sind die wichtigsten Standard-Funktionen der Sprache C

- libc-Funktionen sind unter Linux in Manpages dokumentiert

Theorieteil Lab «Buffer Overflow»

CALLING CONVENTION AMD64

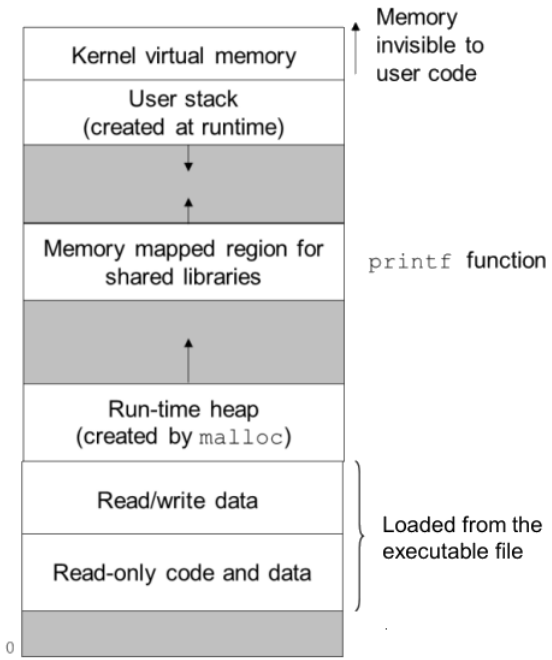
Stack

Start eines Programms:

- Programm bekommt eigenen virtuellen Speicherbereich
- Wird vom Betriebssystem aufgesetzt

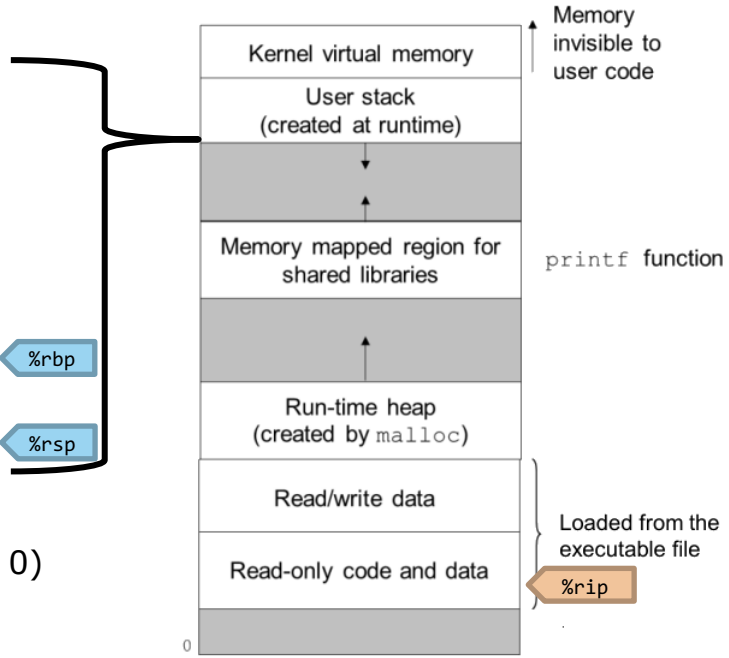
- Stack: Daten der aktuellen Funktion
- Heap: Funktions-unabhängige Daten

- Zusätzliche Details in der Vorlesung



Beispiel-Stack

Adresse	Wert	Bedeutung
0x7f48	0x0000000000007f60	Gespeicherter %rbp aufrufende Funktion
0x7f40	0x0000000000636261	Lokale Variable aufrufende Funktion
0x7f38	0x0000000012345678	Lokale Variable aufrufende Funktion
0x7f30	0x000000009abcdef0	Lokale Variable aufrufende Funktion
0x7f28	0x0000000055555412	Gespeicherte Rücksprungadresse %rip
0x7f20	0x0000000000007f48	Gespeicherter Basepointer %rbp
0x7f18	0x0000000000007f40	Gespeicherter Parameter %rdi
0x7f10	0x0000000000000000	Lokale Variable aktuelle Funktion



Stack wächst «von oben nach unten» (unten = Adresse 0)

…von höheren Adressen zu tieferen Adressen

In CPU-Registern stehen Adressen: Pointer auf Stack (%rbp, %rsp), Code (%rip), etc.

Stack vor Funktionsaufruf

Adresse	Wert	Bedeutung
0x7f48	0x0000000000007f60	Gespeicherter %rbp aufrufende Funktion
0x7f40	0x0000000000636261	Lokale Variable aufrufende Funktion
0x7f38	0x0000000012345678	Lokale Variable aufrufende Funktion
0x7f30	0x000000009abcdef0	Lokale Variable aufrufende Funktion

Base Pointer `%rbp`

Stack Pointer `%rsp`

Nächste Instruktion: **CALL** <function>

- push *instruction pointer* on stack
- change *instruction pointer* to start of new function

Zwischensituation: Nach *call*, vor Funktions-Prolog

Adresse	Wert	Bedeutung
0x7f48	0x0000000000007f60	Gespeicherter %rbp aufrufende Funktion
0x7f40	0x0000000000636261	Lokale Variable aufrufende Funktion
0x7f38	0x0000000012345678	Lokale Variable aufrufende Funktion
0x7f30	0x000000009abcdef0	Lokale Variable aufrufende Funktion
0x7f28	0x0000000055555412	Gespeicherte Rücksprungadresse %rip

Base Pointer `%rbp`

Stack Pointer `%rsp`

`%rip` changed

Nächste Instruktion (Funktions-Prolog 1): **PUSH** %rbp

- push *base pointer* on stack

Funktions-Prolog

Adresse	Wert	Bedeutung
0x7f48	0x0000000000007f60	Gespeicherter %rbp aufrufende Funktion
0x7f40	0x0000000000636261	Lokale Variable aufrufende Funktion
0x7f38	0x0000000012345678	Lokale Variable aufrufende Funktion
0x7f30	0x000000009abcdef0	Lokale Variable aufrufende Funktion
0x7f28	0x0000000055555412	Gespeicherte Rücksprungadresse %rip
0x7f20	0x0000000000007f48	Gespeicherter Basepointer %rbp

Base Pointer %rbp

Stack Pointer %rsp

Nächste Instruktion (Funktions-Prolog 2): **MOV** %rsp,%rbp

- set *base pointer* to value of *stack pointer*

Funktions-Prolog

Adresse	Wert	Bedeutung
0x7f48	0x0000000000007f60	Gespeicherter %rbp aufrufende Funktion
0x7f40	0x0000000000636261	Lokale Variable aufrufende Funktion
0x7f38	0x0000000012345678	Lokale Variable aufrufende Funktion
0x7f30	0x000000009abcdef0	Lokale Variable aufrufende Funktion
0x7f28	0x0000000055555412	Gespeicherte Rücksprungadresse %rip
0x7f20	0x0000000000007f48	Gespeicherter Basepointer %rbp

Stack Pointer `%rsp`

Base Pointer `%rbp`

Nächste Instruktion (Funktions-Prolog 3): **SUB** $0x10,%rsp

- grow stack: subtract 16 from value of *stack pointer*

Nach Funktions-Prolog

Adresse	Wert	Bedeutung
0x7f48	0x0000000000007f60	Gespeicherter %rbp aufrufende Funktion
0x7f40	0x0000000000636261	Lokale Variable aufrufende Funktion
0x7f38	0x0000000012345678	Lokale Variable aufrufende Funktion
0x7f30	0x000000009abcdef0	Lokale Variable aufrufende Funktion
0x7f28	0x0000000055555412	Gespeicherte Rücksprungadresse %rip
0x7f20	0x0000000000007f48	Gespeicherter Basepointer %rbp
0x7f18	0x0000000000000000	(hier steht noch nichts)
0x7f10	0x0000000000000000	(hier steht noch nichts)

Base Pointer `%rbp`

Stack Pointer `%rsp`

Ab hier macht die Funktion dann ihre eigentliche Aufgabe.

(Funktion macht ihre eigentliche Aufgabe)

Ab hier macht die Funktion dann ihre eigentliche Aufgabe.

Rückgabewert, falls vorhanden, wird ins Register %rax geschrieben.

Ende der Funktion: Vor Funktions-Epilog

Adresse	Wert	Bedeutung
0x7f48	0x0000000000007f60	Gespeicherter %rbp aufrufende Funktion
0x7f40	0x0000000000636261	Lokale Variable aufrufende Funktion
0x7f38	0x0000000012345678	Lokale Variable aufrufende Funktion
0x7f30	0x000000009abcdef0	Lokale Variable aufrufende Funktion
0x7f28	0x0000000055555412	Gespeicherte Rücksprungadresse %rip
0x7f20	0x0000000000007f48	Gespeicherter Basepointer %rbp
0x7f18	0x0000000000007f40	(hier stehen Variablen der Funktion)
0x7f10	0x0000000000000000	(hier stehen Variablen der Funktion)

Base Pointer `%rbp`

Stack Pointer `%rsp`

Nächste Instruktion (Funktions-Epilog 1): **MOV** %rbp,%rsp

- set *stack pointer* to value of *base pointer*

Funktions-Epilog

Adresse	Wert	Bedeutung
0x7f48	0x0000000000007f60	Gespeicherter %rbp aufrufende Funktion
0x7f40	0x0000000000636261	Lokale Variable aufrufende Funktion
0x7f38	0x0000000012345678	Lokale Variable aufrufende Funktion
0x7f30	0x000000009abcdef0	Lokale Variable aufrufende Funktion
0x7f28	0x0000000055555412	Gespeicherte Rücksprungadresse %rip
0x7f20	0x0000000000007f48	Gespeicherter Basepointer %rbp

Base Pointer `%rbp`

Stack Pointer `%rsp`

Nächste Instruktion (Funktions-Epilog 2): **POP** %rbp

- remove value from stack and set *base pointer* to removed value

Nach Funktions-Epilog

Adresse	Wert	Bedeutung
0x7f48	0x0000000000007f60	Gespeicherter %rbp aufrufende Funktion
0x7f40	0x0000000000636261	Lokale Variable aufrufende Funktion
0x7f38	0x0000000012345678	Lokale Variable aufrufende Funktion
0x7f30	0x000000009abcdef0	Lokale Variable aufrufende Funktion
0x7f28	0x0000000055555412	Gespeicherte Rücksprungadresse %rip

Base Pointer %rbp

Stack Pointer %rsp

Return

Adresse	Wert	Bedeutung
0x7f48	0x0000000000007f60	Gespeicherter %rbp aufrufende Funktion
0x7f40	0x0000000000636261	Lokale Variable aufrufende Funktion
0x7f38	0x0000000012345678	Lokale Variable aufrufende Funktion
0x7f30	0x000000009abcdef0	Lokale Variable aufrufende Funktion
0x7f28	0x0000000055555412	Gespeicherte Rücksprungadresse %rip

Base Pointer `%rbp`

Stack Pointer `%rsp`

Nächste Instruktion: **RET**

- remove value from stack and set ***instruction*** *pointer* to removed value

(Dies ist keine POP-Instruktion, weil der Instruction Pointer zusätzliche Offsets via CS-Register unterstützt)

Nach Return

Adresse	Wert	Bedeutung
0x7f48	0x0000000000007f60	Gespeicherter %rbp aufrufende Funktion
0x7f40	0x0000000000636261	Lokale Variable aufrufende Funktion
0x7f38	0x0000000012345678	Lokale Variable aufrufende Funktion
0x7f30	0x000000009abcdef0	Lokale Variable aufrufende Funktion

Base Pointer `%rbp`

Stack Pointer `%rsp`

`%rip` changed

Stack sieht wieder aus wie vor dem Funktionsaufruf.

Instruction Pointer zeigt wieder auf Code der aufrufenden Funktion.

Rückgabewert der Funktion, falls vorhanden, liegt im Register %rax.