CSE 410/510 Special Topics: Software Security

Instructor: Dr. Ziming Zhao

Location: NSC 220

Time: Monday 5:00PM - 7:50PM

First Half of This Class

1. Background
   a. System call
   b. Environment
   c. Tools
   d. ELF

Background Knowledge: System Calls

What is System Call?

When a process needs to invoke a kernel service, it invokes a procedure call in the operating system interface using special instructions (not a **call** instruction in x86). Such a procedure is called a system call.

The system call enters the kernel; the kernel performs the service and returns. Thus a process alternates between executing in user space and kernel space.

System calls are generally not invoked directly by a program, but rather via wrapper functions in glibc (or perhaps some other library).

Popular System Call

On Unix, Unix-like and other POSIX-compliant operating systems, popular system calls are open, read, write, close, wait, exec, fork, exit, and kill.

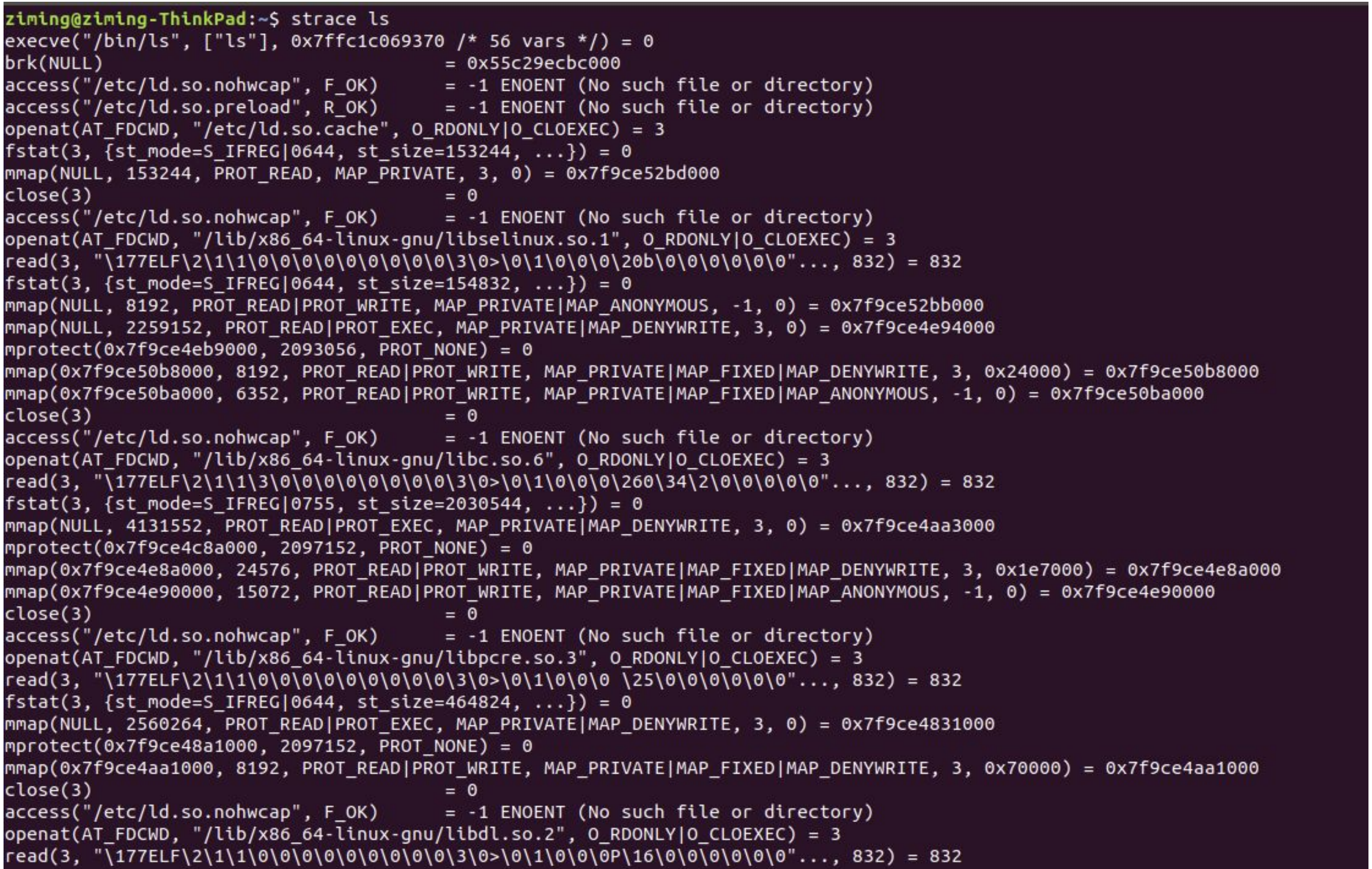
Many modern operating systems have hundreds of system calls. For example, Linux and OpenBSD each have over 300 different calls, FreeBSD has over 500, Windows 7 has close to 700.

Glibc interfaces

Often, but not always, the name of the wrapper function is the same as the name of the system call that it invokes.

For example, glibc contains a function chdir() which invokes the underlying "chdir" system call.

Tools: strace & ltrace



```
ziming@ziming-ThinkPad:~$ strace ls
execve("/bin/ls", ["ls"], 0x7ffc1c069370 /* 56 vars */) = 0
brk(NULL)                               = 0x55c29ecbc000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=153244, ...}) = 0
mmap(NULL, 153244, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f9ce52bd000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20b\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=154832, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f9ce52bb000
mmap(NULL, 2259152, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f9ce4e94000
mprotect(0x7f9ce4eb9000, 2093056, PROT_NONE) = 0
mmap(0x7f9ce50b8000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x24000) = 0x7f9ce50b8000
mmap(0x7f9ce50ba000, 6352, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f9ce50ba000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260\34\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2030544, ...}) = 0
mmap(NULL, 4131552, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f9ce4aa3000
mprotect(0x7f9ce4c8a000, 2097152, PROT_NONE) = 0
mmap(0x7f9ce4e8a000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e7000) = 0x7f9ce4e8a000
mmap(0x7f9ce4e90000, 15072, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f9ce4e90000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libpcre.so.3", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0 \25\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=464824, ...}) = 0
mmap(NULL, 2560264, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f9ce4831000
mprotect(0x7f9ce48a1000, 2097152, PROT_NONE) = 0
mmap(0x7f9ce4aa1000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x70000) = 0x7f9ce4aa1000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libdl.so.2", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\16\0\0\0\0\0\0"..., 832) = 832
```

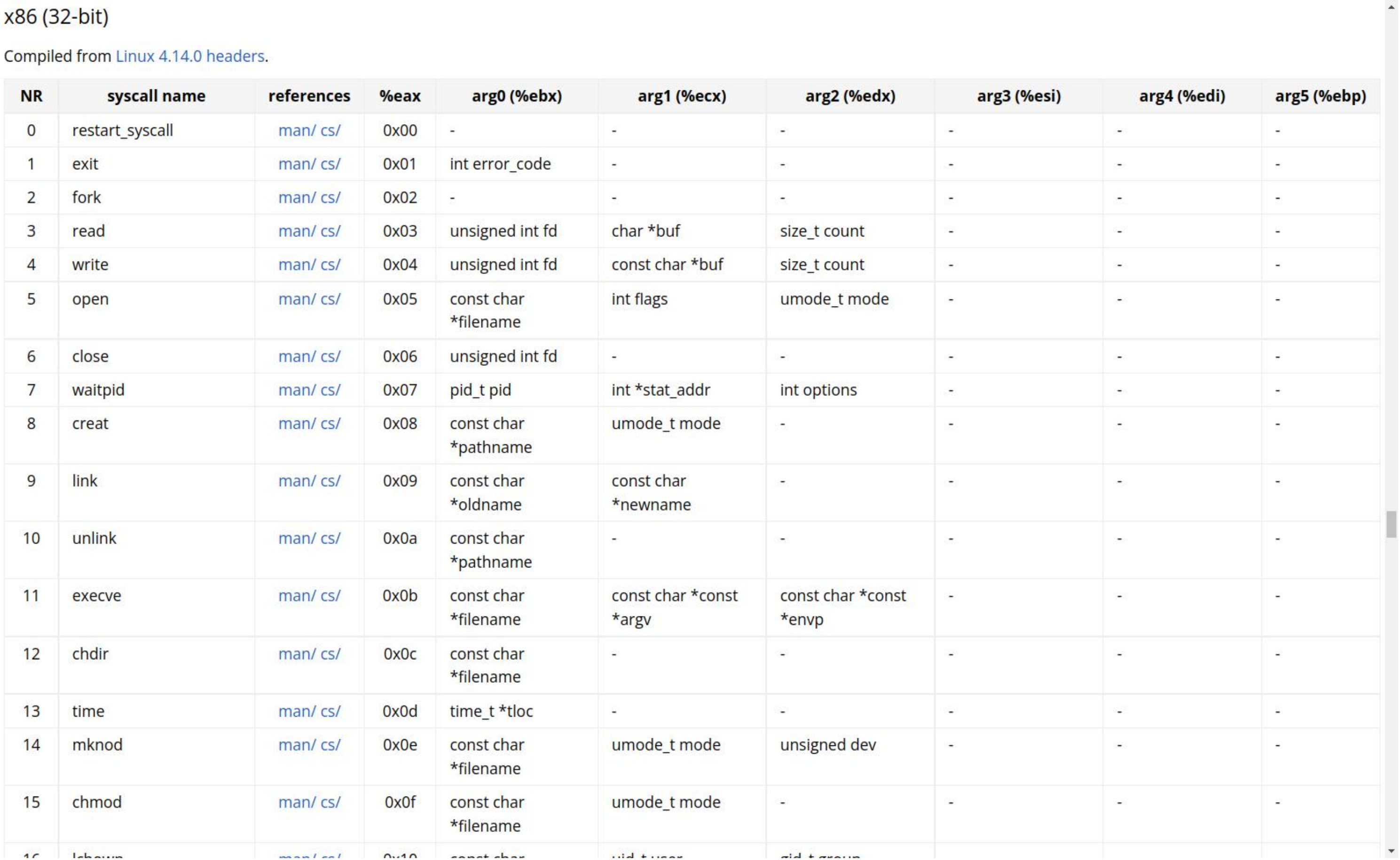
Making a System Call in x86/64 Assembly

On x86/x86-64, most system calls rely on the software interrupt.

A software interrupt is caused either by an exceptional condition in the processor itself, or a special instruction (the **int 0x80** instruction or **syscall** instruction).

For example: a divide-by-zero exception will be thrown if the processor's arithmetic logic unit is commanded to divide a number by zero as this instruction is in error and impossible.

Making a System Call in x86 Assembly (INT 0x80)



x86 (32-bit)

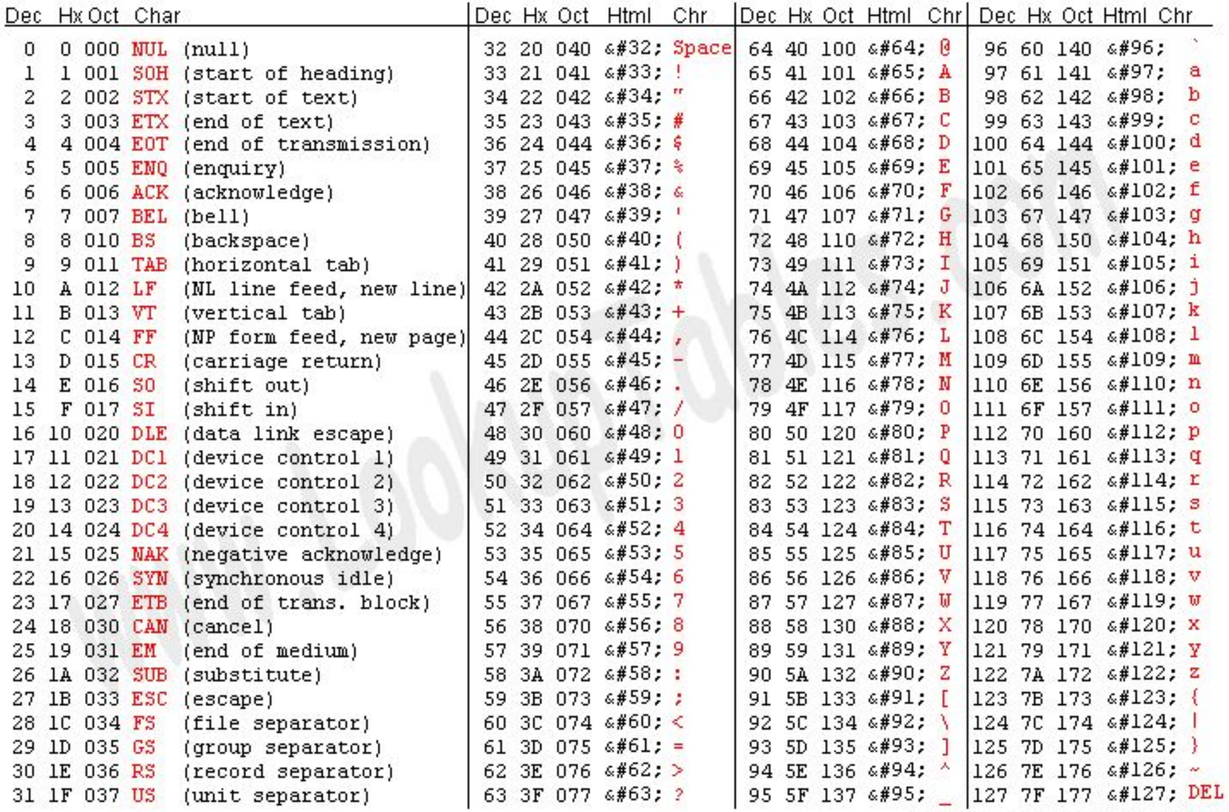
Compiled from Linux 4.14.0 headers.

NR	syscall name	references	%eax	arg0 (%ebx)	arg1 (%ecx)	arg2 (%edx)	arg3 (%esi)	arg4 (%edi)	arg5 (%ebp)
0	restart_syscall	man/ cs/	0x00	-	-	-	-	-	-
1	exit	man/ cs/	0x01	int error_code	-	-	-	-	-
2	fork	man/ cs/	0x02	-	-	-	-	-	-
3	read	man/ cs/	0x03	unsigned int fd	char *buf	size_t count	-	-	-
4	write	man/ cs/	0x04	unsigned int fd	const char *buf	size_t count	-	-	-
5	open	man/ cs/	0x05	const char *filename	int flags	umode_t mode	-	-	-
6	close	man/ cs/	0x06	unsigned int fd	-	-	-	-	-
7	waitpid	man/ cs/	0x07	pid_t pid	int *stat_addr	int options	-	-	-
8	creat	man/ cs/	0x08	const char *pathname	umode_t mode	-	-	-	-
9	link	man/ cs/	0x09	const char *oldname	const char *newname	-	-	-	-
10	unlink	man/ cs/	0x0a	const char *pathname	-	-	-	-	-
11	execve	man/ cs/	0x0b	const char *filename	const char *const *argv	const char *const *envp	-	-	-
12	chdir	man/ cs/	0x0c	const char *filename	-	-	-	-	-
13	time	man/ cs/	0x0d	time_t *tloc	-	-	-	-	-
14	mknod	man/ cs/	0x0e	const char *filename	umode_t mode	unsigned dev	-	-	-
15	chmod	man/ cs/	0x0f	const char *filename	umode_t mode	-	-	-	-
16	lchown	man/ cs/	0x10	const char	uid_t user	gid_t group			

https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86-32_bit

Making a System Call in x86 Assembly

```
xor    eax,eax
push   eax
push   0x68732f2f
push   0x6e69622f
mov    ebx,esp
push   eax
push   ebx
mov    ecx,esp
mov    al,0xb
int    0x80
```



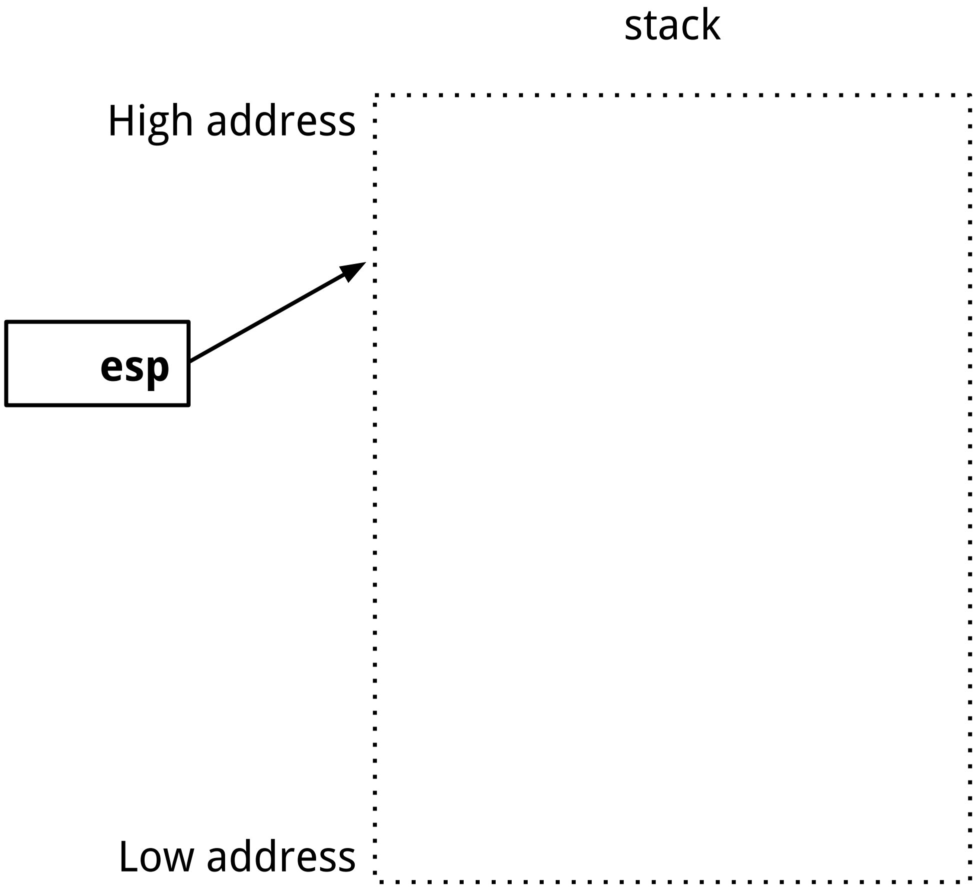
Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH	(start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX	(start of text)	34	22	042	&#34;	"	66	42	102	&#66;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

Source: www.LookupTables.com

http://shell-storm.org/shellcode/files/shellcode-827.php

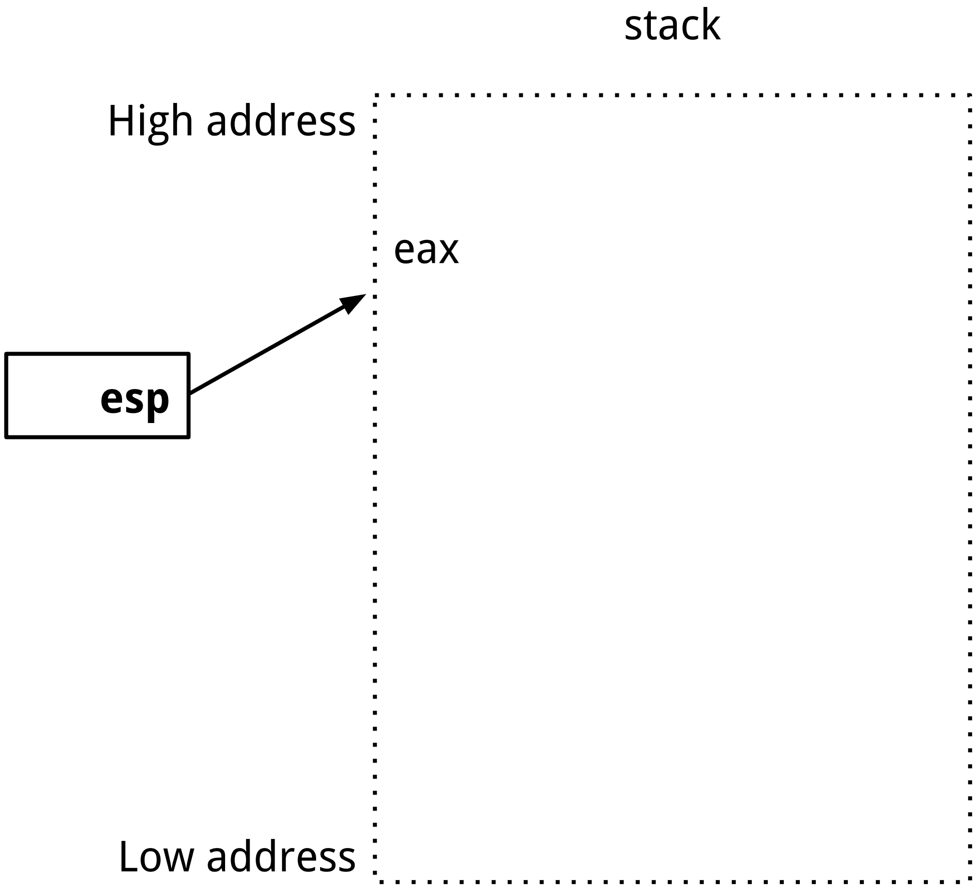
Making a System Call in x86 Assembly

```
xor    eax,eax
push   eax
push   0x68732f2f
push   0x6e69622f
mov    ebx,esp
push   eax
push   ebx
mov    ecx,esp
mov    al,0xb
int    0x80
```



Making a System Call in x86 Assembly

```
xor    eax,eax
push   eax
push   0x68732f2f
push   0x6e69622f
mov    ebx,esp
push   eax
push   ebx
mov    ecx,esp
mov    al,0xb
int    0x80
```



stack

High address

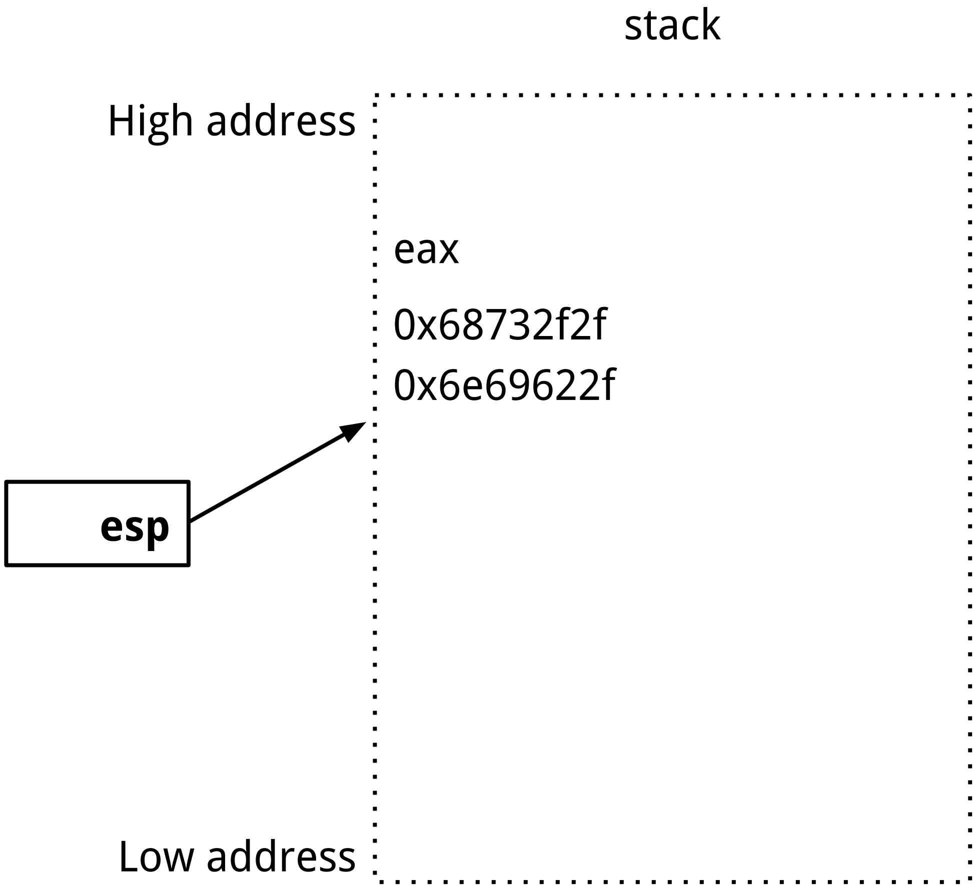
eax

esp

Low address

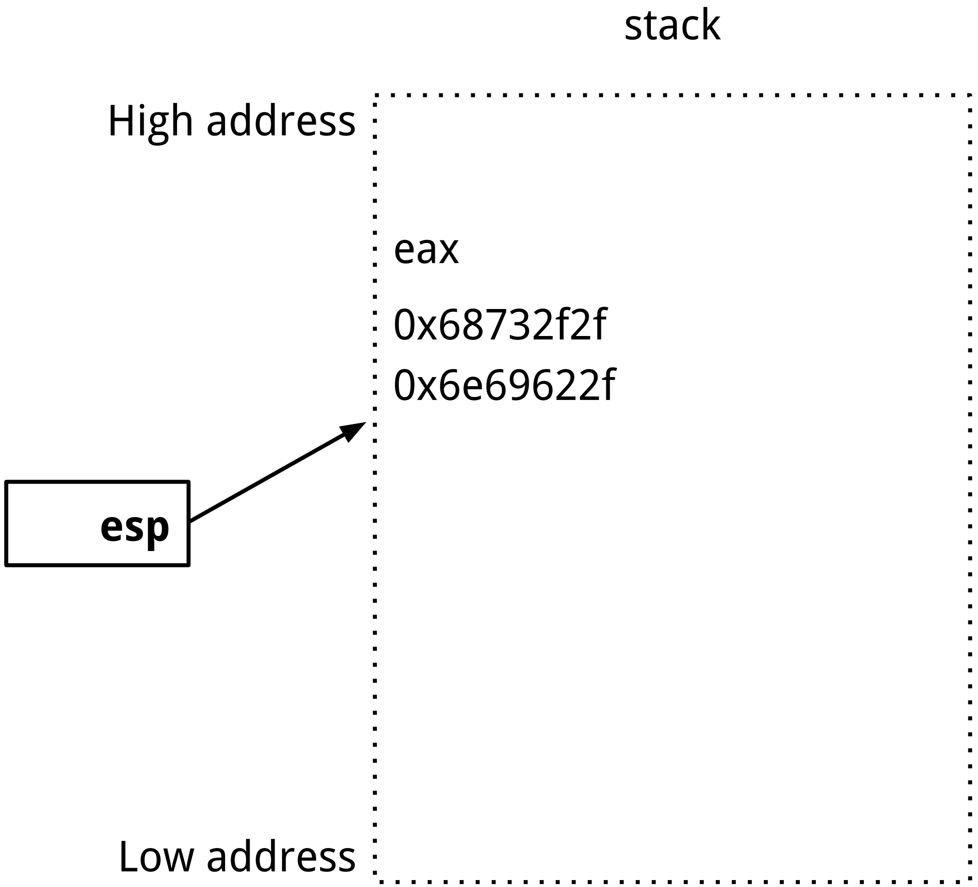
Making a System Call in x86 Assembly

```
xor    eax,eax
push   eax
push   0x68732f2f
push   0x6e69622f
mov    ebx,esp
push   eax
push   ebx
mov    ecx,esp
mov    al,0xb
int    0x80
```



Making a System Call in x86 Assembly

```
xor    eax,eax
push   eax
push   0x68732f2f
push   0x6e69622f
mov    ebx,esp
push   eax
push   ebx
mov    ecx,esp
mov    al,0xb
int    0x80
```



stack

High address

eax

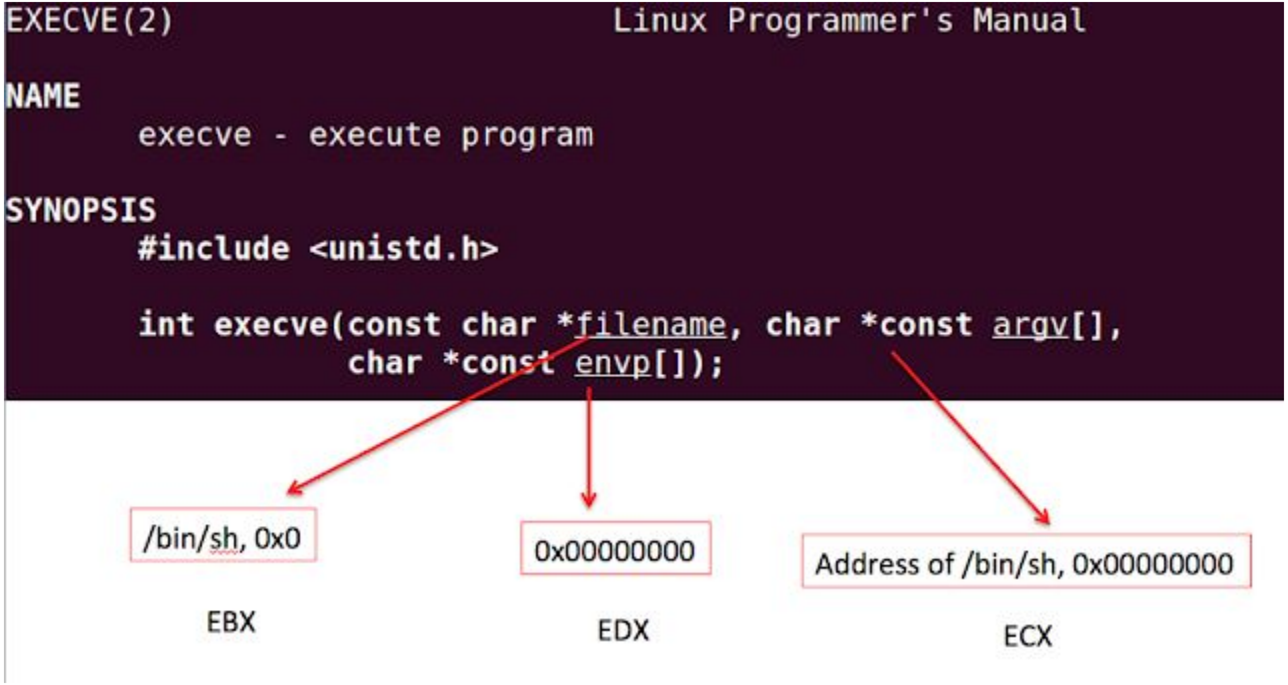
0x68732f2f

0x6e69622f

esp

Low address

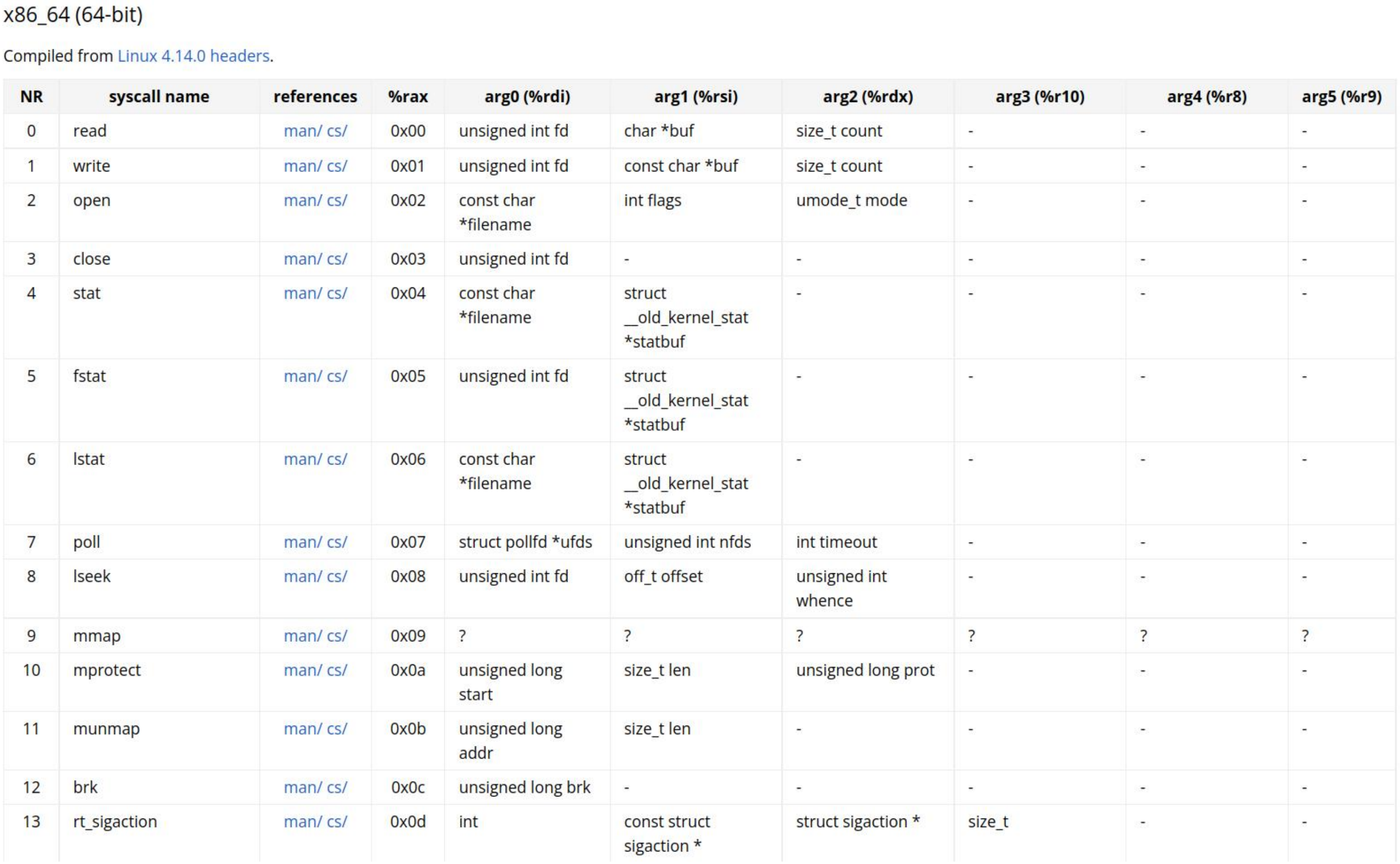
Making a System Call in x86 Assembly



```
EXECVE(2)                    Linux Programmer's Manual

NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

       int execve(const char *filename, char *const argv[],
                  char *const envp[]);
```

execve("/bin/sh", address of string "/bin/sh", 0)

Making a System Call in x86_64 (64-bit) Assembly



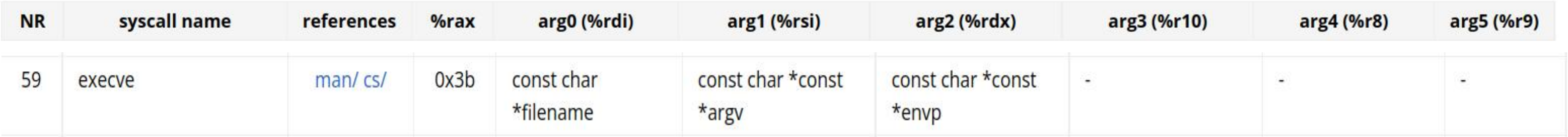
x86_64 (64-bit)

Compiled from Linux 4.14.0 headers.

NR	syscall name	references	%rax	arg0 (%rdi)	arg1 (%rsi)	arg2 (%rdx)	arg3 (%r10)	arg4 (%r8)	arg5 (%r9)
0	read	man/ cs/	0x00	unsigned int fd	char *buf	size_t count	-	-	-
1	write	man/ cs/	0x01	unsigned int fd	const char *buf	size_t count	-	-	-
2	open	man/ cs/	0x02	const char *filename	int flags	umode_t mode	-	-	-
3	close	man/ cs/	0x03	unsigned int fd	-	-	-	-	-
4	stat	man/ cs/	0x04	const char *filename	struct __old_kernel_stat *statbuf	-	-	-	-
5	fstat	man/ cs/	0x05	unsigned int fd	struct __old_kernel_stat *statbuf	-	-	-	-
6	lstat	man/ cs/	0x06	const char *filename	struct __old_kernel_stat *statbuf	-	-	-	-
7	poll	man/ cs/	0x07	struct pollfd *ufds	unsigned int nfds	int timeout	-	-	-
8	lseek	man/ cs/	0x08	unsigned int fd	off_t offset	unsigned int whence	-	-	-
9	mmap	man/ cs/	0x09	?	?	?	?	?	?
10	mprotect	man/ cs/	0x0a	unsigned long start	size_t len	unsigned long prot	-	-	-
11	munmap	man/ cs/	0x0b	unsigned long addr	size_t len	-	-	-	-
12	brk	man/ cs/	0x0c	unsigned long brk	-	-	-	-	-
13	rt_sigaction	man/ cs/	0x0d	int	const struct sigaction *	struct sigaction *	size_t	-	-

https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86-32_bit

Making a System Call in x86_64 (64-bit) Assembly



NR	syscall name	references	%rax	arg0 (%rdi)	arg1 (%rsi)	arg2 (%rdx)	arg3 (%r10)	arg4 (%r8)	arg5 (%r9)
59	execve	man/ cs/	0x3b	const char *filename	const char *const *argv	const char *const *envp	-	-	-

```
push rax
xor rdx, rdx
xor rsi, rsi
mov rbx,'/bin//sh'
push rbx
push rsp
pop rdi
mov al, 59
syscall
```

SYSTEM AND LIBRARY CALLS EVERY PROGRAMMER NEEDS TO KNOW

LINUX SYSTEM PROGRAMMING

O'REILLY®

ROBERT LOVE

Background Knowledge: Environment and Shell Variables

Environment and Shell Variables

Environment and Shell variables are a set of dynamic **named values**, stored within the system that are used by applications launched in shells.

```
KEY=value
KEY="Some other value"
KEY=value1:value2
```

The names of the variables are case-sensitive (UPPER CASE).
Multiple values must be separated by the colon : character.
There is no space around the equals = symbol.

Environment and Shell Variables

Environment variables are variables that are available system-wide and are inherited by all spawned child processes and shells.

Shell variables are variables that apply only to the current shell instance. Each shell such as zsh and bash, has its own set of internal shell variables.

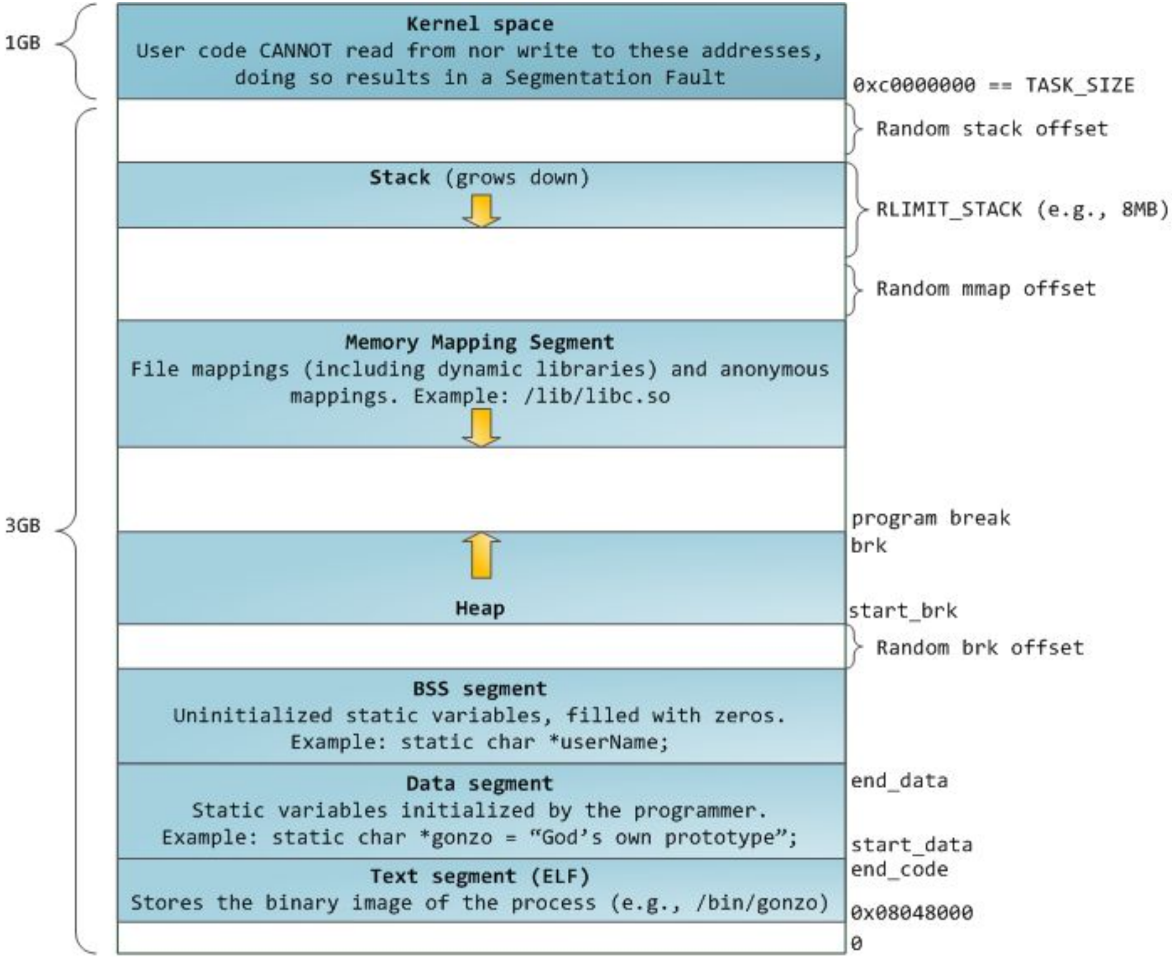
Common Environment Variables

USER - The current logged in user.
HOME - The home directory of the current user.
EDITOR - The default file editor to be used. This is the editor that will be used when you type edit in your terminal.
SHELL - The path of the current user’s shell, such as bash or zsh.
LOGNAME - The name of the current user.
PATH - A list of directories to be searched when executing commands.
LANG - The current locales settings.
TERM - The current terminal emulation.
MAIL - Location of where the current user’s mail is stored.

Commands

env – The command allows you to run another program in a custom environment without modifying the current one. When used without an argument it will print a list of the current environment variables.
printenv – The command prints all or the specified environment variables.
set – The command sets or unsets shell variables. When used without an argument it will print a list of all variables including environment and shell variables, and shell functions.
unset – The command deletes shell and environment variables.
export – The command sets environment variables

The environment variables live towards the top of the stack, together with command line arguments.



1GB

Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault

0xc0000000 == TASK_SIZE

Random stack offset

Stack (grows down)

RLIMIT_STACK (e.g., 8MB)

Random mmap offset

Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

3GB

program break
brk

Heap

start_brk

Random brk offset

BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;

Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = “God’s own prototype”;

end_data

start_data

Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)

end_code

0x08048000

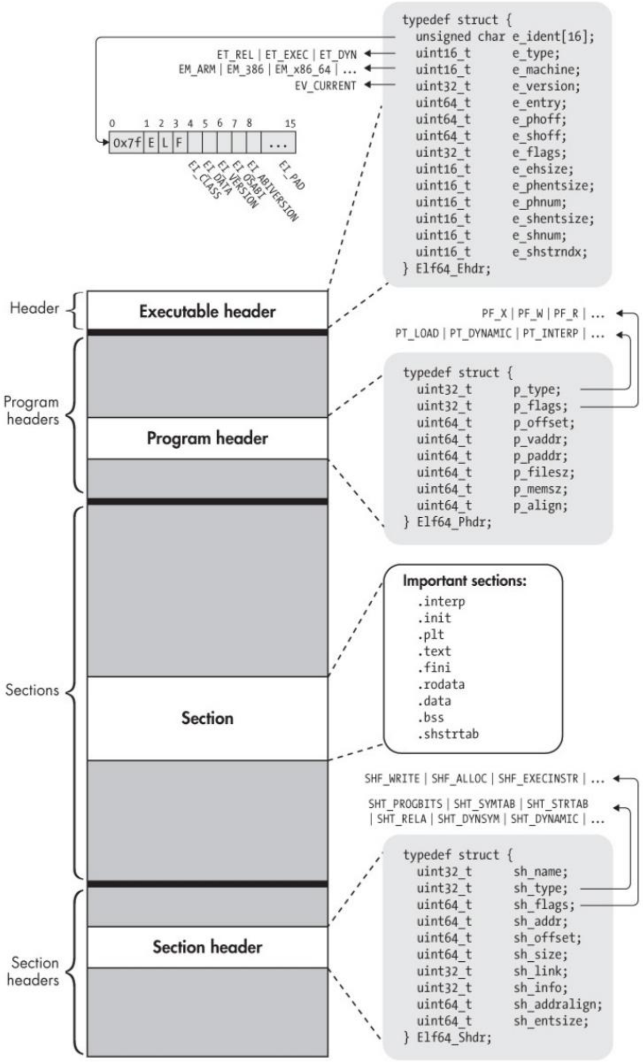
0

Executable and Linkable Format (ELF)

ELF Files

The **Executable** and **Linkable Format** (**ELF**) is a common standard file format for *executable files*, *object code*, *shared libraries*, and *core dumps*. Filename extension *none*, *.axf*, *.bin*, *.elf*, *.o*, *.prx*, *.puff*, *.ko*, *.mod* and *.so*

Contains the program and its data. Describes how the program should be loaded (program/segment headers). Contains metadata describing program components (section headers).


Header
Executable header
Program headers
Program header
Sections
Section
Section headers
Section header
ET_REL | ET_EXEC | ET_DYN
EM_ARM | EM_386 | EM_x86_64 | ...
EV_CURRENT
0 1 2 3 4 5 6 7 8 15
0x7f E L F ...
EI_CLASS
EI_DATA
EI_VERSION
EI_OSABI
EI_ABIVERSION
EI_PAD
PF_X | PF_W | PF_R | ...
PT_LOAD | PT_DYNAMIC | PT_INTERP | ...
SHF_WRITE | SHF_ALLOC | SHF_EXECINSTR | ...
SHT_PROGBITS | SHT_SYMTAB | SHT_STRTAB | SHT_RELA | SHT_DYNSYM | SHT_DYNAMIC | ...

```
typedef struct {
  unsigned char e_ident[16];
  uint16_t      e_type;
  uint16_t      e_machine;
  uint32_t      e_version;
  uint64_t      e_entry;
  uint64_t      e_phoff;
  uint64_t      e_shoff;
  uint32_t      e_flags;
  uint16_t      e_ehsize;
  uint16_t      e_phentsize;
  uint16_t      e_phnum;
  uint16_t      e_shentsize;
  uint16_t      e_shnum;
  uint16_t      e_shstrndx;
} Elf64_Ehdr;
```

```
typedef struct {
  uint32_t      p_type;
  uint32_t      p_flags;
  uint64_t      p_offset;
  uint64_t      p_vaddr;
  uint64_t      p_paddr;
  uint64_t      p_filesz;
  uint64_t      p_memsz;
  uint64_t      p_align;
} Elf64_Phdr;
```

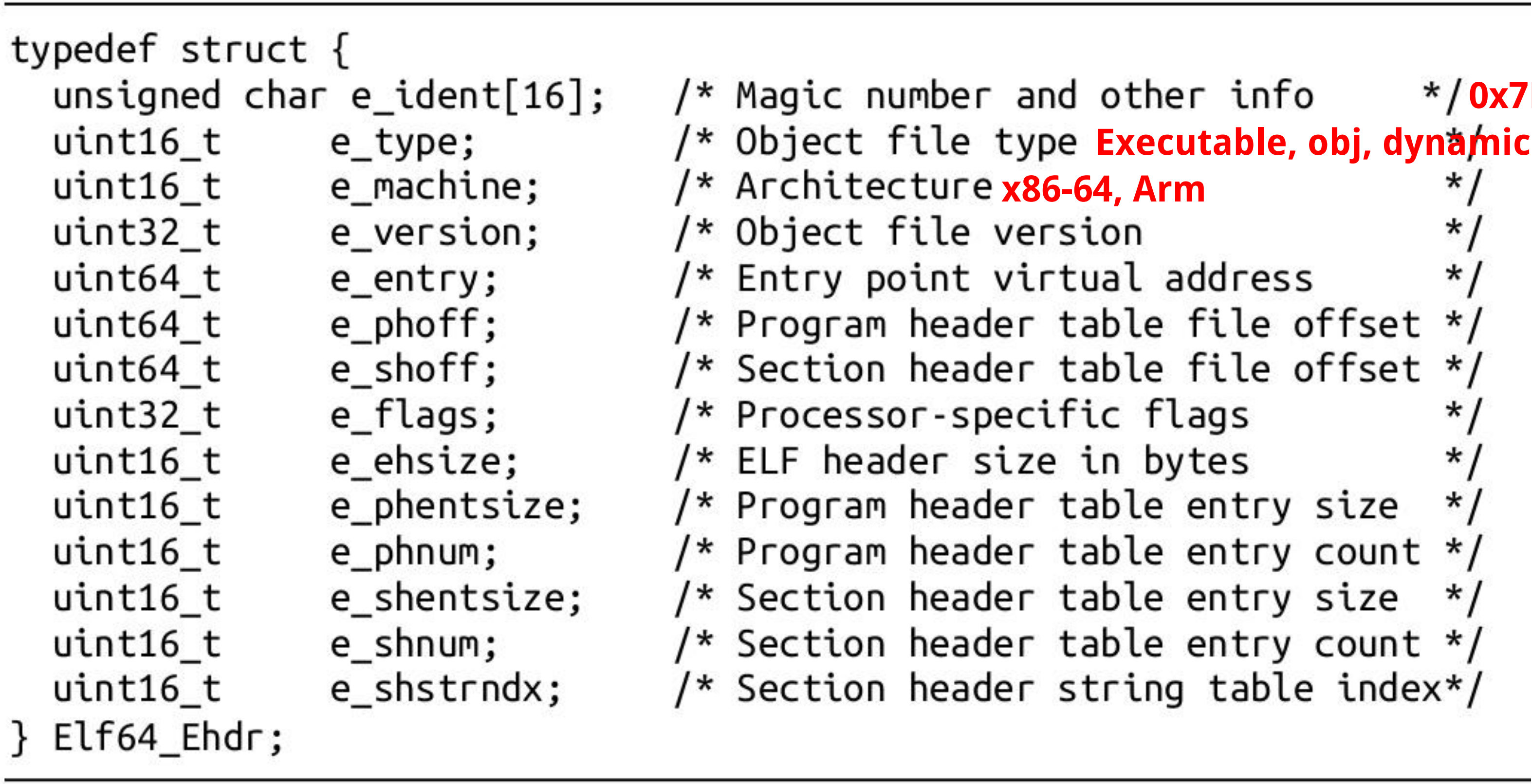
Important sections:

```
  .interp
  .init
  .plt
  .text
  .fini
  .rodata
  .data
  .bss
  .shstrtab
```

```
typedef struct {
  uint32_t      sh_name;
  uint32_t      sh_type;
  uint64_t      sh_flags;
  uint64_t      sh_addr;
  uint64_t      sh_offset;
  uint64_t      sh_size;
  uint32_t      sh_link;
  uint32_t      sh_info;
  uint64_t      sh_addralign;
  uint64_t      sh_entsize;
} Elf64_Shdr;
```

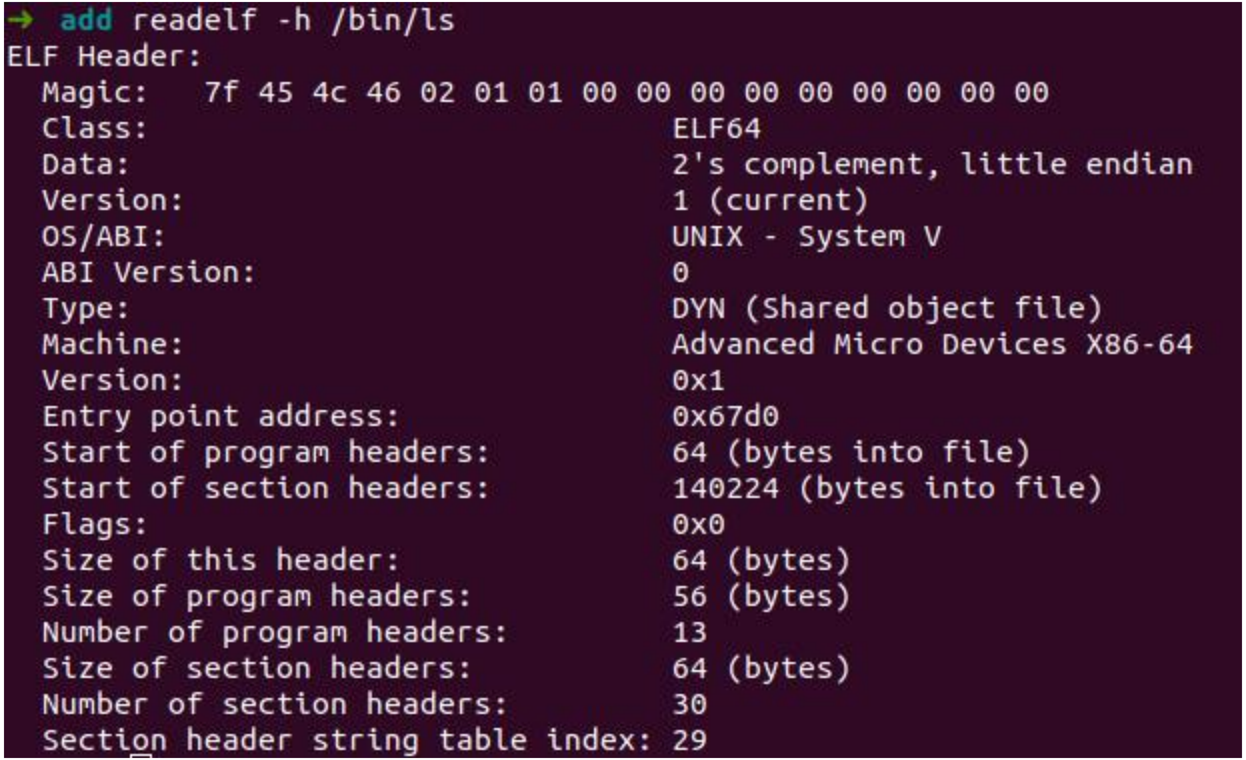
- Executable (a.out), object files (.o), shared libraries (.a), even core dumps.
- Four *types* of components: an **executable header**, a series of (optional) **program headers**, a number of **sections**, and a series of (optional) **section headers**, one per section.

Executable Header



```
typedef struct {
  unsigned char e_ident[16];   /* Magic number and other info      */ 0x7F ELF ..
  uint16_t      e_type;        /* Object file type Executable, obj, dynamic lib */
  uint16_t      e_machine;     /* Architecture x86-64, Arm         */
  uint32_t      e_version;     /* Object file version              */
  uint64_t      e_entry;       /* Entry point virtual address      */
  uint64_t      e_phoff;       /* Program header table file offset */
  uint64_t      e_shoff;       /* Section header table file offset */
  uint32_t      e_flags;       /* Processor-specific flags         */
  uint16_t      e_ehsize;      /* ELF header size in bytes         */
  uint16_t      e_phentsize;   /* Program header table entry size  */
  uint16_t      e_phnum;       /* Program header table entry count */
  uint16_t      e_shentsize;   /* Section header table entry size  */
  uint16_t      e_shnum;       /* Section header table entry count */
  uint16_t      e_shstrndx;    /* Section header string table index*/
} Elf64_Ehdr;
```

```
readelf -h a.out
```



```
→  add readelf -h /bin/ls
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x67d0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          140224 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 29
```

Sections

The code and data in an ELF binary are logically divided into contiguous non-overlapping chunks called sections. The structure of each section varies depending on the contents.

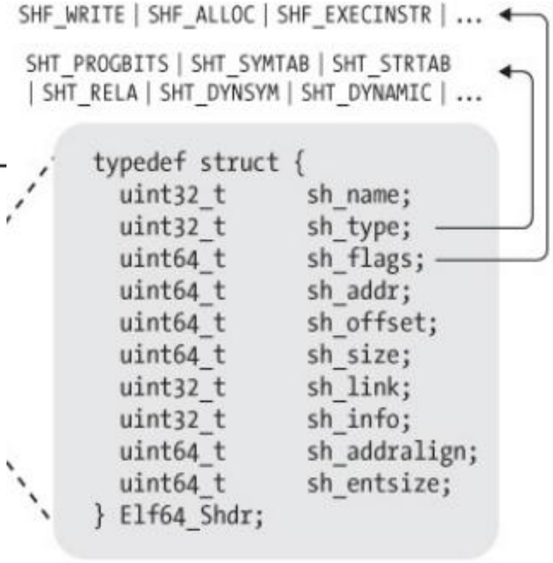
The division into sections is intended to provide a convenient organization for use by the ***linker***.

Section Header Format

```
typedef struct {
  uint32_t  sh_name;       /* Section name (string tbl index)   */
  uint32_t  sh_type;       /* Section type                      */
  uint64_t  sh_flags;      /* Section flags                     */
  uint64_t  sh_addr;       /* Section virtual addr at execution */
  uint64_t  sh_offset;     /* Section file offset               */
  uint64_t  sh_size;       /* Section size in bytes             */
  uint32_t  sh_link;       /* Link to another section           */
  uint32_t  sh_info;       /* Additional section information    */
  uint64_t  sh_addralign;  /* Section alignment                 */
  uint64_t  sh_entsize;    /* Entry size if section holds table */
} Elf64_Shdr;
```

Each section is described by its section header.

```
readelf -S a.out
```


SHF_WRITE | SHF_ALLOC | SHF_EXECINSTR | ...
SHT_PROGBITS | SHT_SYMTAB | SHT_STRTAB
| SHT_RELA | SHT_DYNSYM | SHT_DYNAMIC | ...

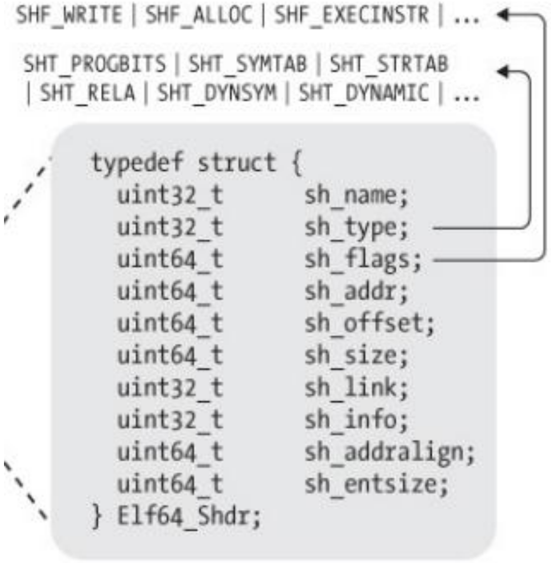
```
typedef struct {
  uint32_t      sh_name;
  uint32_t      sh_type;
  uint64_t      sh_flags;
  uint64_t      sh_addr;
  uint64_t      sh_offset;
  uint64_t      sh_size;
  uint32_t      sh_link;
  uint32_t      sh_info;
  uint64_t      sh_addralign;
  uint64_t      sh_entsize;
} Elf64_Shdr;
```

sh_flags

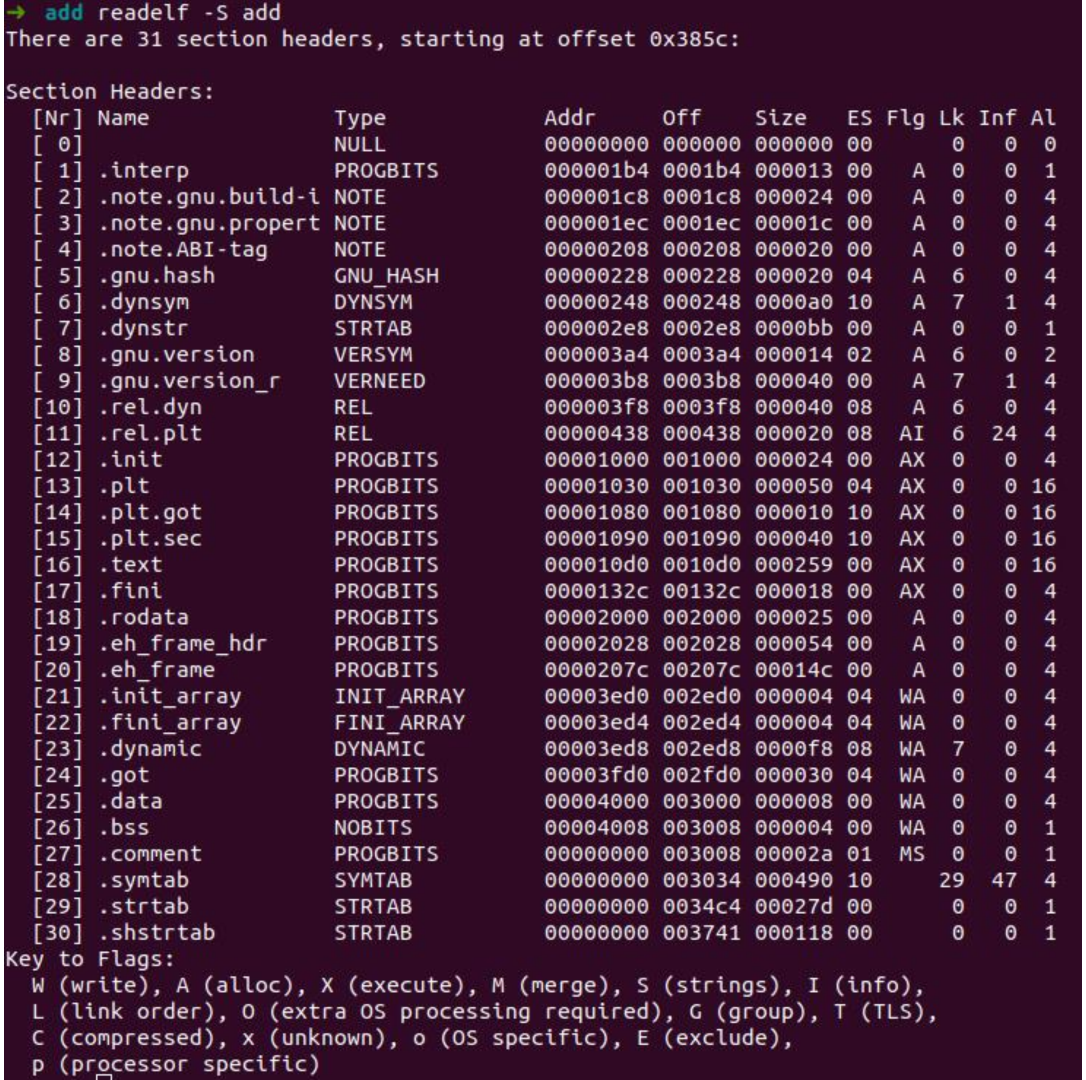
SHF_WRITE: the section is writable at runtime.

SHF_ALLOC: the contents of the section are to be loaded into virtual memory when executing the binary.

SHF_EXECINSTR: the section contains executable instructions.


SHF_WRITE | SHF_ALLOC | SHF_EXECINSTR | ...
SHT_PROGBITS | SHT_SYMTAB | SHT_STRTAB
| SHT_RELA | SHT_DYNSYM | SHT_DYNAMIC | ...

```
typedef struct {
  uint32_t      sh_name;
  uint32_t      sh_type;
  uint64_t      sh_flags;
  uint64_t      sh_addr;
  uint64_t      sh_offset;
  uint64_t      sh_size;
  uint32_t      sh_link;
  uint32_t      sh_info;
  uint64_t      sh_addralign;
  uint64_t      sh_entsize;
} Elf64_Shdr;
```



```
→  add readelf -S add
There are 31 section headers, starting at offset 0x385c:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        000001b4 0001b4 000013 00   A  0   0  1
  [ 2] .note.gnu.build-i NOTE            000001c8 0001c8 000024 00   A  0   0  4
  [ 3] .note.gnu.propert NOTE            000001ec 0001ec 00001c 00   A  0   0  4
  [ 4] .note.ABI-tag     NOTE            00000208 000208 000020 00   A  0   0  4
  [ 5] .gnu.hash         GNU_HASH        00000228 000228 000020 04   A  6   0  4
  [ 6] .dynsym           DYNSYM          00000248 000248 0000a0 10   A  7   1  4
  [ 7] .dynstr           STRTAB          000002e8 0002e8 0000bb 00   A  0   0  1
  [ 8] .gnu.version      VERSYM          000003a4 0003a4 000014 02   A  6   0  2
  [ 9] .gnu.version_r    VERNEED         000003b8 0003b8 000040 00   A  7   1  4
  [10] .rel.dyn          REL             000003f8 0003f8 000040 08   A  6   0  4
  [11] .rel.plt          REL             00000438 000438 000020 08  AI  6  24  4
  [12] .init             PROGBITS        00001000 001000 000024 00  AX  0   0  4
  [13] .plt              PROGBITS        00001030 001030 000050 04  AX  0   0 16
  [14] .plt.got          PROGBITS        00001080 001080 000010 10  AX  0   0 16
  [15] .plt.sec          PROGBITS        00001090 001090 000040 10  AX  0   0 16
  [16] .text             PROGBITS        000010d0 0010d0 000259 00  AX  0   0 16
  [17] .fini             PROGBITS        0000132c 00132c 000018 00  AX  0   0  4
  [18] .rodata           PROGBITS        00002000 002000 000025 00   A  0   0  4
  [19] .eh_frame_hdr     PROGBITS        00002028 002028 000054 00   A  0   0  4
  [20] .eh_frame         PROGBITS        0000207c 00207c 00014c 00   A  0   0  4
  [21] .init_array       INIT_ARRAY      00003ed0 002ed0 000004 04  WA  0   0  4
  [22] .fini_array       FINI_ARRAY      00003ed4 002ed4 000004 04  WA  0   0  4
  [23] .dynamic          DYNAMIC         00003ed8 002ed8 0000f8 08  WA  7   0  4
  [24] .got              PROGBITS        00003fd0 002fd0 000030 04  WA  0   0  4
  [25] .data             PROGBITS        00004000 003000 000008 00  WA  0   0  4
  [26] .bss              NOBITS          00004008 003008 000004 00  WA  0   0  1
  [27] .comment          PROGBITS        00000000 003008 00002a 01  MS  0   0  1
  [28] .symtab           SYMTAB          00000000 003034 000490 10     29  47  4
  [29] .strtab           STRTAB          00000000 0034c4 00027d 00      0   0  1
  [30] .shstrtab         STRTAB          00000000 003741 000118 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
```

readelf -S a.out

Sections

.init: executable code that performs initialization tasks and needs to run before any other code in the binary is executed.

.fini: code that runs after the main program completes.

.text: where the main code of the program resides.

Sections

.rodata section, which stands for “read-only data,” is dedicated to storing constant values. Because it stores constant values, .rodata is not writable.

The default values of initialized variables are stored in the .data section, which is marked as writable since the values of variables may change at runtime.

the .bss section reserves space for uninitialized variables. The name historically stands for “block started by symbol,” referring to the reserving of blocks of memory for (symbolic) variables.

Lazy Binding (.plt, .got, .got.plt Sections)

Binding at Load Time: When a binary is loaded into a process for execution, the dynamic linker resolves references to functions located in shared libraries. The addresses of shared functions were not known at compile time.

In reality - Lazy Binding: many of the relocations are typically not done right away when the binary is loaded but are deferred until the first reference to the unresolved location is actually made.

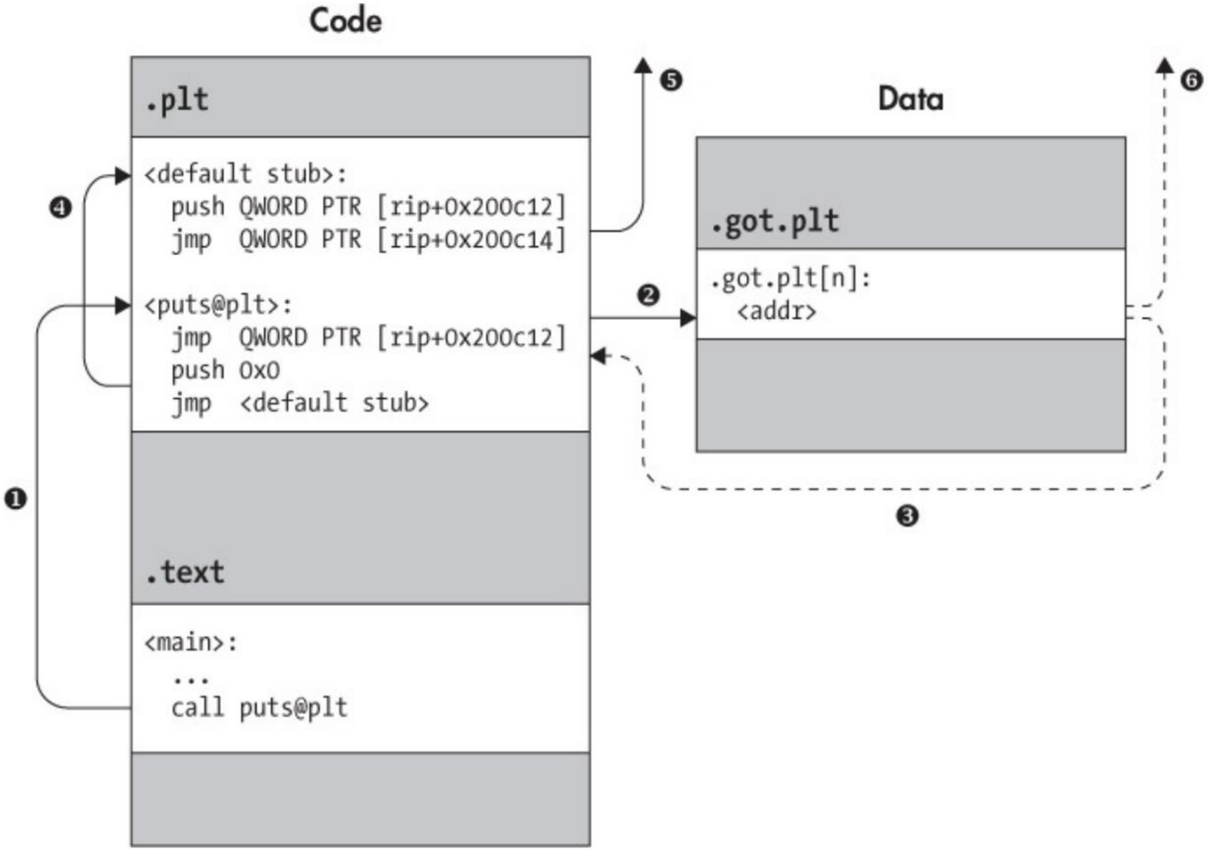
Lazy Binding (.plt, .got, .got.plt Sections)

Lazy binding in Linux ELF binaries is implemented with the help of two special sections, called the Procedure Linkage Table (.plt) and the Global Offset Table (.got).

.plt is a code section that contains executable code. The PLT consists entirely of stubs of a well-defined format, dedicated to directing calls from the .text section to the appropriate library location.

.got.plt is a data section.

Dynamically Resolving a Library Function Using the PLT

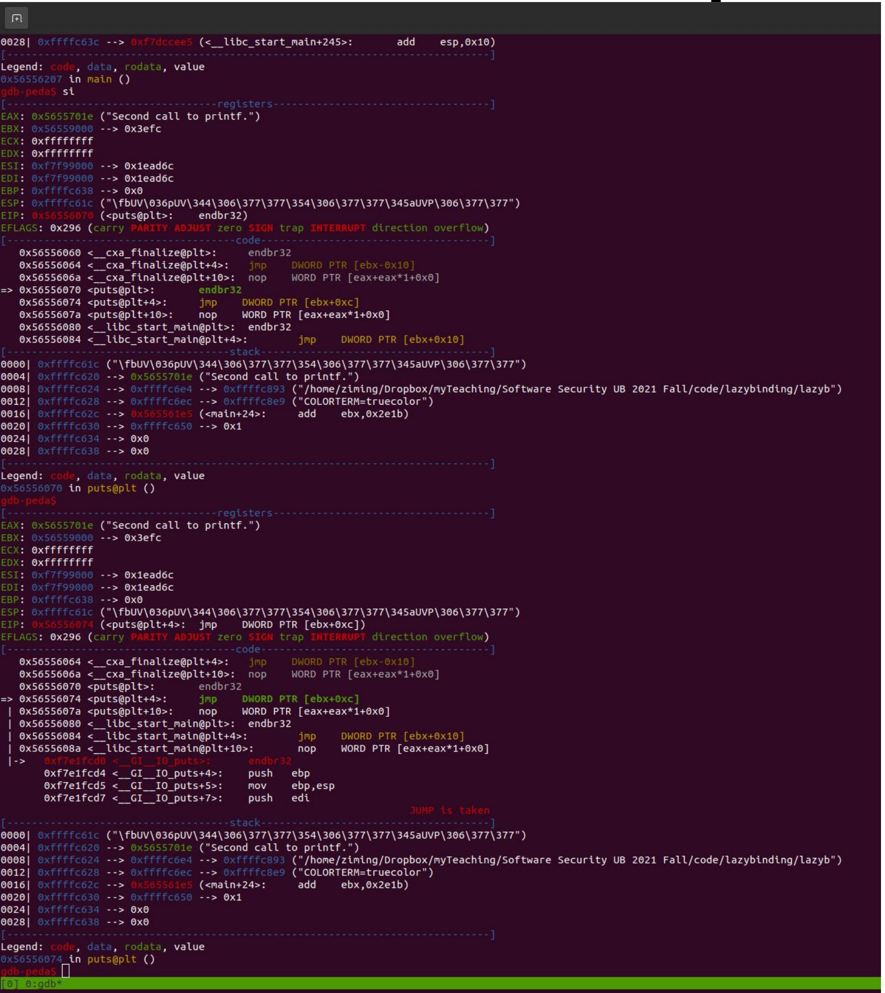


```
<default stub>:
  push QWORD PTR [rip+0x200c12]
  jmp  QWORD PTR [rip+0x200c14]

<puts@plt>:
  jmp  QWORD PTR [rip+0x200c12]
  push 0x0
  jmp  <default stub>
```

```
.got.plt[n]:
  <addr>
```

```
<main>:
  ...
  call puts@plt
```

Example: Debug code\lazyb



```
0028| 0xffffc63c --> 0xf7dccee5 (<__libc_start_main+245>:	add    esp,0x10)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x56556207 in main ()
gdb-peda$ si
[----------------------------------registers-----------------------------------]
EAX: 0x5655701e ("Second call to printf.")
EBX: 0x56559000 --> 0x3efc
ECX: 0xffffffff
EDX: 0xffffffff
ESI: 0xf7f99000 --> 0x1ead6c
EDI: 0xf7f99000 --> 0x1ead6c
EBP: 0xffffc638 --> 0x0
ESP: 0xffffc61c ("\fbUV\036pUV\344\306\377\377\354\306\377\377\345aUVP\306\377\377")
EIP: 0x56556070 (<puts@plt>:	endbr32)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x56556060 <__cxa_finalize@plt>:	endbr32
   0x56556064 <__cxa_finalize@plt+4>:	jmp    DWORD PTR [ebx-0x10]
   0x5655606a <__cxa_finalize@plt+10>:	nop    WORD PTR [eax+eax*1+0x0]
=> 0x56556070 <puts@plt>:	endbr32
   0x56556074 <puts@plt+4>:	jmp    DWORD PTR [ebx+0xc]
   0x5655607a <puts@plt+10>:	nop    WORD PTR [eax+eax*1+0x0]
   0x56556080 <__libc_start_main@plt>:	endbr32
   0x56556084 <__libc_start_main@plt+4>:	jmp    DWORD PTR [ebx+0x10]
[------------------------------------stack-------------------------------------]
0000| 0xffffc61c ("\fbUV\036pUV\344\306\377\377\354\306\377\377\345aUVP\306\377\377")
0004| 0xffffc620 --> 0x5655701e ("Second call to printf.")
0008| 0xffffc624 --> 0xffffc6e4 --> 0xffffc893 ("/home/ziming/Dropbox/myTeaching/Software Security UB 2021 Fall/code/lazybinding/lazyb")
0012| 0xffffc628 --> 0xffffc6ec --> 0xffffc8e9 ("COLORTERM=truecolor")
0016| 0xffffc62c --> 0x565561e5 (<main+24>:	add    ebx,0x2e1b)
0020| 0xffffc630 --> 0xffffc650 --> 0x1
0024| 0xffffc634 --> 0x0
0028| 0xffffc638 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x56556070 in puts@plt ()
gdb-peda$
[----------------------------------registers-----------------------------------]
EAX: 0x5655701e ("Second call to printf.")
EBX: 0x56559000 --> 0x3efc
ECX: 0xffffffff
EDX: 0xffffffff
ESI: 0xf7f99000 --> 0x1ead6c
EDI: 0xf7f99000 --> 0x1ead6c
EBP: 0xffffc638 --> 0x0
ESP: 0xffffc61c ("\fbUV\036pUV\344\306\377\377\354\306\377\377\345aUVP\306\377\377")
EIP: 0x56556074 (<puts@plt+4>:	jmp    DWORD PTR [ebx+0xc])
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x56556064 <__cxa_finalize@plt+4>:	jmp    DWORD PTR [ebx-0x10]
   0x5655606a <__cxa_finalize@plt+10>:	nop    WORD PTR [eax+eax*1+0x0]
   0x56556070 <puts@plt>:	endbr32
=> 0x56556074 <puts@plt+4>:	jmp    DWORD PTR [ebx+0xc]
 | 0x5655607a <puts@plt+10>:	nop    WORD PTR [eax+eax*1+0x0]
 | 0x56556080 <__libc_start_main@plt>:	endbr32
 | 0x56556084 <__libc_start_main@plt+4>:	jmp    DWORD PTR [ebx+0x10]
 | 0x5655608a <__libc_start_main@plt+10>:	nop    WORD PTR [eax+eax*1+0x0]
 |->   0xf7e1fcd0 <__GI__IO_puts>:	endbr32
       0xf7e1fcd4 <__GI__IO_puts+4>:	push   ebp
       0xf7e1fcd5 <__GI__IO_puts+5>:	mov    ebp,esp
       0xf7e1fcd7 <__GI__IO_puts+7>:	push   edi
                                                                  JUMP is taken
[------------------------------------stack-------------------------------------]
0000| 0xffffc61c ("\fbUV\036pUV\344\306\377\377\354\306\377\377\345aUVP\306\377\377")
0004| 0xffffc620 --> 0x5655701e ("Second call to printf.")
0008| 0xffffc624 --> 0xffffc6e4 --> 0xffffc893 ("/home/ziming/Dropbox/myTeaching/Software Security UB 2021 Fall/code/lazybinding/lazyb")
0012| 0xffffc628 --> 0xffffc6ec --> 0xffffc8e9 ("COLORTERM=truecolor")
0016| 0xffffc62c --> 0x565561e5 (<main+24>:	add    ebx,0x2e1b)
0020| 0xffffc630 --> 0xffffc650 --> 0x1
0024| 0xffffc634 --> 0x0
0028| 0xffffc638 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x56556074 in puts@plt ()
gdb-peda$
[0] 0:gdb*
```

GDB Cheatsheet:

https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf

Section View (Section Header)
vs.
Segment View (Program Header)

The program header table provides a segment view of the binary, as opposed to the section view provided by the section header table.

The section view of an ELF binary is meant for static linking purposes.

The segment view is used by the operating system and dynamic linker when loading an ELF into a process for execution to locate the relevant code and data and decide what to load into virtual memory.

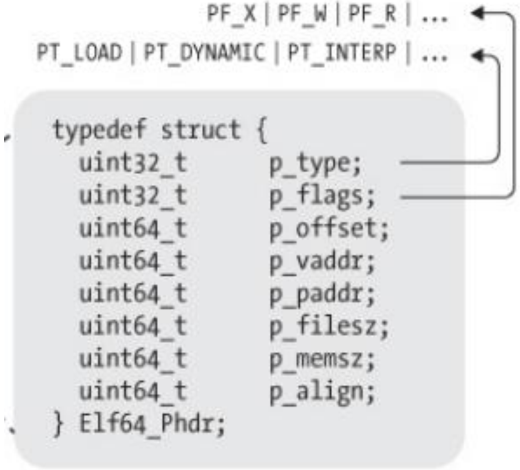
Segments are simply a bunch of sections bundled together.

Program Header Format

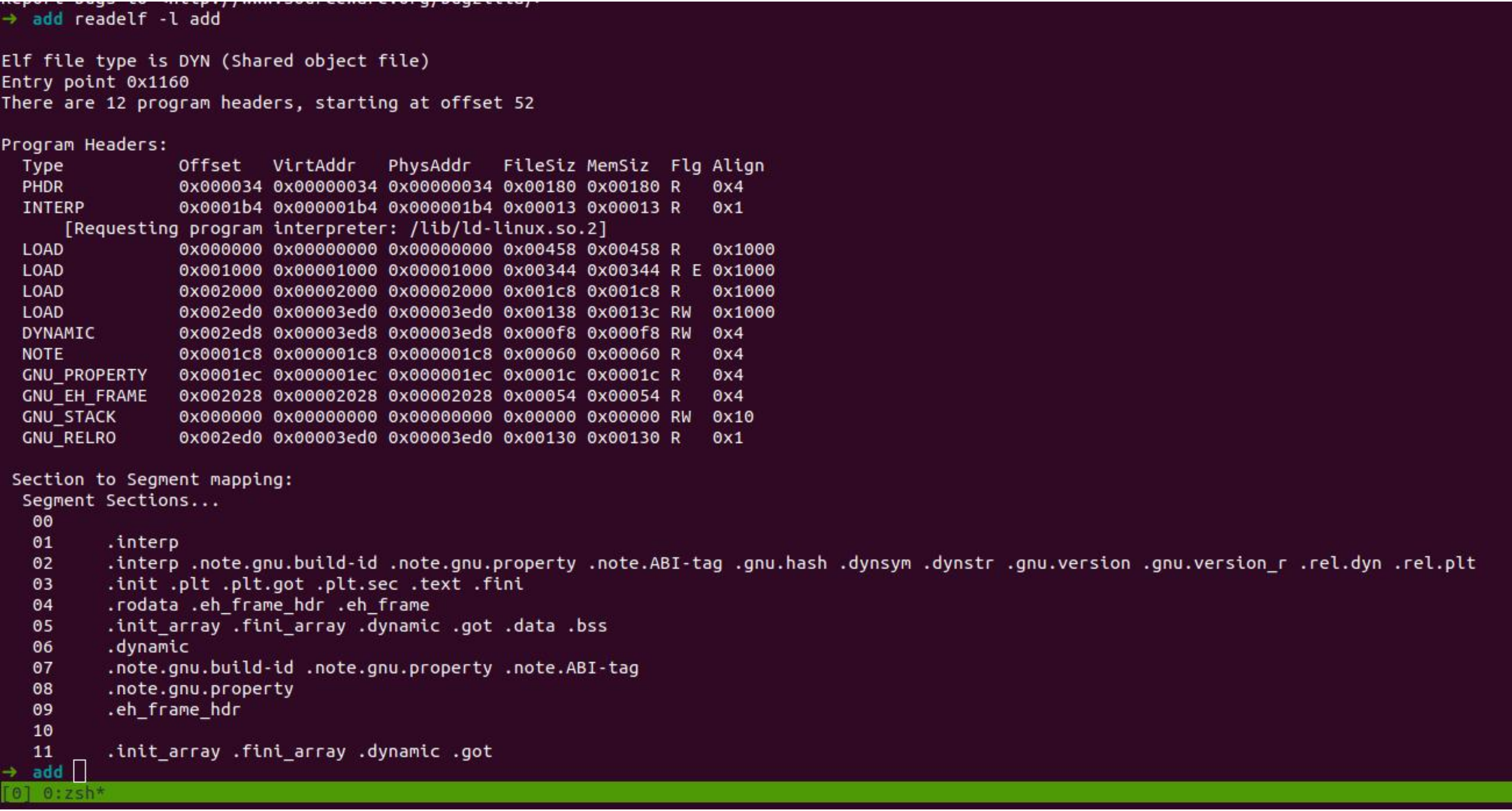
```
typedef struct {
  uint32_t  p_type;   /* Segment type             */
  uint32_t  p_flags;  /* Segment flags            */
  uint64_t  p_offset; /* Segment file offset      */
  uint64_t  p_vaddr;  /* Segment virtual address  */
  uint64_t  p_paddr;  /* Segment physical address */
  uint64_t  p_filesz; /* Segment size in file     */
  uint64_t  p_memsz;  /* Segment size in memory   */
  uint64_t  p_align;  /* Segment alignment        */
} Elf64_Phdr;
```

Each section is described by its section header.

```
readelf -l a.out
```



```
typedef struct {
  uint32_t      p_type;
  uint32_t      p_flags;
  uint64_t      p_offset;
  uint64_t      p_vaddr;
  uint64_t      p_paddr;
  uint64_t      p_filesz;
  uint64_t      p_memsz;
  uint64_t      p_align;
} Elf64_Phdr;
```



```
Report bugs to <http://www.sourceware.org/bugzilla/>
→ add readelf -l add

Elf file type is DYN (Shared object file)
Entry point 0x1160
There are 12 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x00180 0x00180 R   0x4
  INTERP         0x0001b4 0x000001b4 0x000001b4 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x00000000 0x00000000 0x00458 0x00458 R   0x1000
  LOAD           0x001000 0x00001000 0x00001000 0x00344 0x00344 R E 0x1000
  LOAD           0x002000 0x00002000 0x00002000 0x001c8 0x001c8 R   0x1000
  LOAD           0x002ed0 0x00003ed0 0x00003ed0 0x00138 0x0013c RW  0x1000
  DYNAMIC        0x002ed8 0x00003ed8 0x00003ed8 0x000f8 0x000f8 RW  0x4
  NOTE           0x0001c8 0x000001c8 0x000001c8 0x00060 0x00060 R   0x4
  GNU_PROPERTY   0x0001ec 0x000001ec 0x000001ec 0x0001c 0x0001c R   0x4
  GNU_EH_FRAME   0x002028 0x00002028 0x00002028 0x00054 0x00054 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
  GNU_RELRO      0x002ed0 0x00003ed0 0x00003ed0 0x00130 0x00130 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.build-id .note.gnu.property .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt
   03     .init .plt .plt.got .plt.sec .text .fini
   04     .rodata .eh_frame_hdr .eh_frame
   05     .init_array .fini_array .dynamic .got .data .bss
   06     .dynamic
   07     .note.gnu.build-id .note.gnu.property .note.ABI-tag
   08     .note.gnu.property
   09     .eh_frame_hdr
   10
   11     .init_array .fini_array .dynamic .got
→ add
[0] 0:zsh*
```

Background Knowledge:
Manual Binary Analysis Tools

Tools for this class

file
readelf
strings
nm
objdump
GDB
[optional] IDA Pro
[optional] ghidra
[optional] Binary Ninja

GDB Cheat Sheet

Start gdb using:

```
gdb <binary>
```

Pass initial commands for gdb through a file

```
gdb <binary> –x <initfile>
```

To start running the program

```
r <argv>
```

Use python output as stdin in GDB:

```
r <<< $(python -c "print '\x12\x34'*5")
```

Set breakpoint at address:

```
b *0x80000000
b main
```

Disassemble 10 instructions from an address:

```
x/10i 0x80000000
```

GDB Cheat Sheet

To put breakpoints (stop execution on a certain line)

```
b <function name>
b *<instruction address>
b <filename:line number>
b <line number>
```

To show breakpoints

```
info b
```

To remove breakpoints

```
clear <function name>
clear *<instruction address>
clear <filename:line number>
clear <line number>
```

GDB Cheat Sheet

Use “examine” or “x” command
x/32xw <memory location> to see memory contents at memory location, showing 32 hexadecimal words
x/5s <memory location> to show 5 strings (null terminated) at a particular memory location
x/10i <memory location> to show 10 instructions at particular memory location

See registers
info reg

Step an instruction
si

Shell Cheat Sheet

Run a program and use another program’s output as a parameter

```
program $(python -c "print '\x12\x34'*5")
```

10 Mins Break

This Class

1. Stack-based buffer overflow (Sequential buffer overflow)
    a. Brief history of buffer overflow
    b. Information C function needs to run
    c. C calling conventions (x86, x86-64)
    d. Overflow local variables
    e. Overflow RET address to execute a function
    f. Overflow RET and more to execute a function with parameters

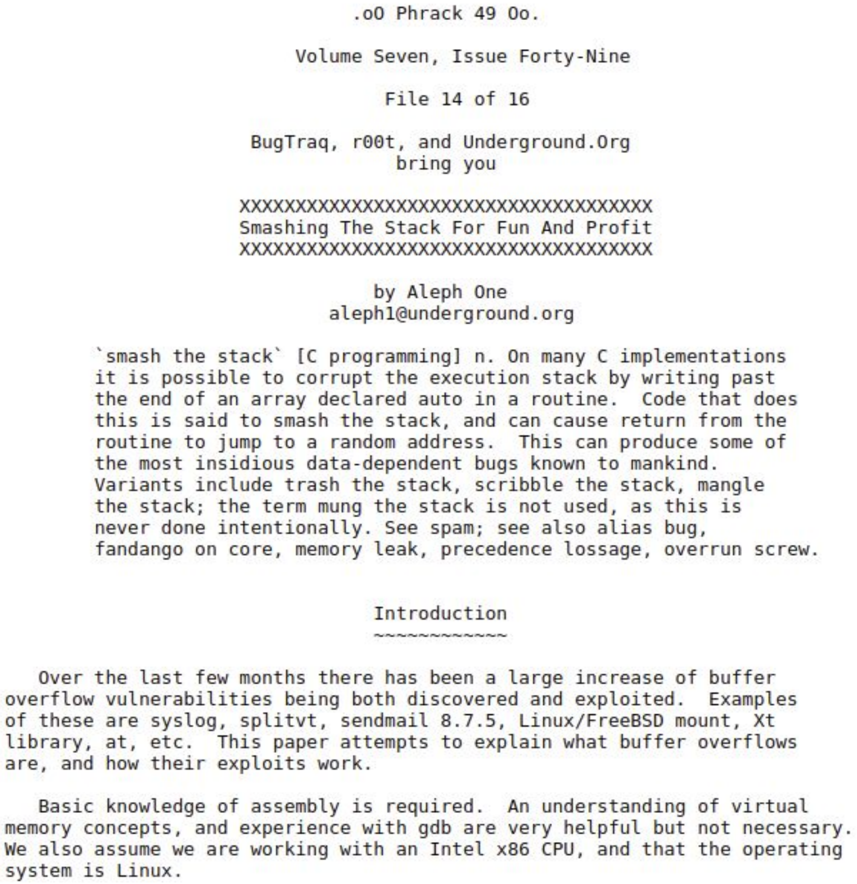
Stack-based Buffer Overflow

Objectives

1. Understand how stack works in Linux x86/64
2. Identify a buffer overflow in a program
3. Exploit a buffer overflow vulnerability

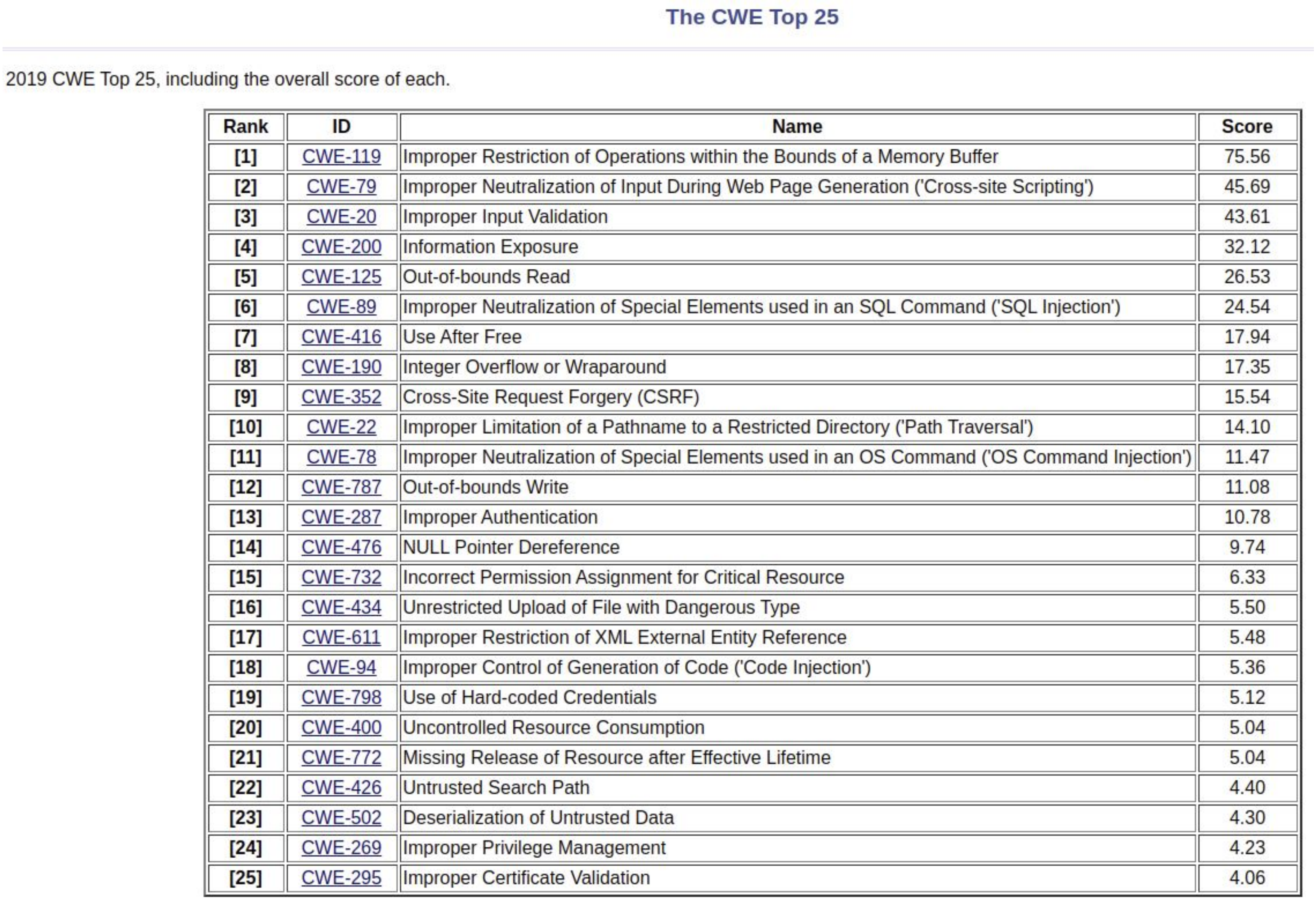
An Extremely Brief History of Buffer Overflow

The Morris worm (November 9, 1988), was one of the first computer worms distributed via the Internet, and the first to gain significant mainstream media attention. Morris worn used buffer overflow as one of its attack techniques.



```
                               .oO Phrack 49 Oo.

                          Volume Seven, Issue Forty-Nine

                                   File 14 of 16

                     BugTraq, r00t, and Underground.Org
                                    bring you

                     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                     Smashing The Stack For Fun And Profit
                     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

                                 by Aleph One
                             aleph1@underground.org

        `smash the stack` [C programming] n. On many C implementations
        it is possible to corrupt the execution stack by writing past
        the end of an array declared auto in a routine.  Code that does
        this is said to smash the stack, and can cause return from the
        routine to jump to a random address.  This can produce some of
        the most insidious data-dependent bugs known to mankind.
        Variants include trash the stack, scribble the stack, mangle
        the stack; the term mung the stack is not used, as this is
        never done intentionally. See spam; see also alias bug,
        fandango on core, memory leak, precedence lossage, overrun screw.


                                 Introduction
                                 ~~~~~~~~~~~~

   Over the last few months there has been a large increase of buffer
overflow vulnerabilities being both discovered and exploited.  Examples
of these are syslog, splitvt, sendmail 8.7.5, Linux/FreeBSD mount, Xt
library, at, etc.  This paper attempts to explain what buffer overflows
are, and how their exploits work.

   Basic knowledge of assembly is required.  An understanding of virtual
memory concepts, and experience with gdb are very helpful but not necessary.
We also assume we are working with an Intel x86 CPU, and that the operating
system is Linux.
```

1996-11-08



The CWE Top 25

2019 CWE Top 25, including the overall score of each.

Rank	ID	Name	Score
[1]	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	75.56
[2]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.69
[3]	CWE-20	Improper Input Validation	43.61
[4]	CWE-200	Information Exposure	32.12
[5]	CWE-125	Out-of-bounds Read	26.53
[6]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	24.54
[7]	CWE-416	Use After Free	17.94
[8]	CWE-190	Integer Overflow or Wraparound	17.35
[9]	CWE-352	Cross-Site Request Forgery (CSRF)	15.54
[10]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.10
[11]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	11.47
[12]	CWE-787	Out-of-bounds Write	11.08
[13]	CWE-287	Improper Authentication	10.78
[14]	CWE-476	NULL Pointer Dereference	9.74
[15]	CWE-732	Incorrect Permission Assignment for Critical Resource	6.33
[16]	CWE-434	Unrestricted Upload of File with Dangerous Type	5.50
[17]	CWE-611	Improper Restriction of XML External Entity Reference	5.48
[18]	CWE-94	Improper Control of Generation of Code ('Code Injection')	5.36
[19]	CWE-798	Use of Hard-coded Credentials	5.12
[20]	CWE-400	Uncontrolled Resource Consumption	5.04
[21]	CWE-772	Missing Release of Resource after Effective Lifetime	5.04
[22]	CWE-426	Untrusted Search Path	4.40
[23]	CWE-502	Deserialization of Untrusted Data	4.30
[24]	CWE-269	Improper Privilege Management	4.23
[25]	CWE-295	Improper Certificate Validation	4.06

C/C++ Function in x86

What information do we need to call a function at runtime? Where are they stored?

- Code
- Parameters
- Return value
- Global variables
- Local variables
- Temporary variables
- Return address
- *Function frame pointer*
- *Previous function Frame pointer*

Global and Local Variables in C/C++

Variables that are declared inside a function or block are called **local variables**. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own.

Global variables are defined outside a function. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

In the definition of function parameters which are called **formal parameters**. Formal parameters are similar to local variables.

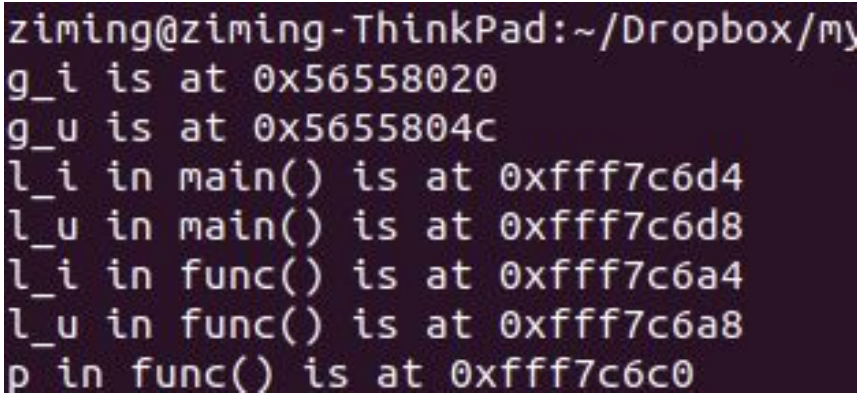
Global and Local Variables (code/globallocalv)

```
char g_i[] = "I am an initialized global variable\n";
char* g_u;

int func(int p)
{
  int l_i = 10;
  int l_u;

  printf("l_i in func() is at %p\n", &l_i);
  printf("l_u in func() is at %p\n", &l_u);
  printf("p in func() is at %p\n", &p);
  return 0;
}
```

```
int main(int argc, char *argv[])
{
  int l_i = 10;
  int l_u;

  printf("g_i is at %p\n", &g_i);
  printf("g_u is at %p\n", &g_u);

  printf("l_i in main() is at %p\n", &l_i);
  printf("l_u in main() is at %p\n", &l_u);

  func(10);
}
```

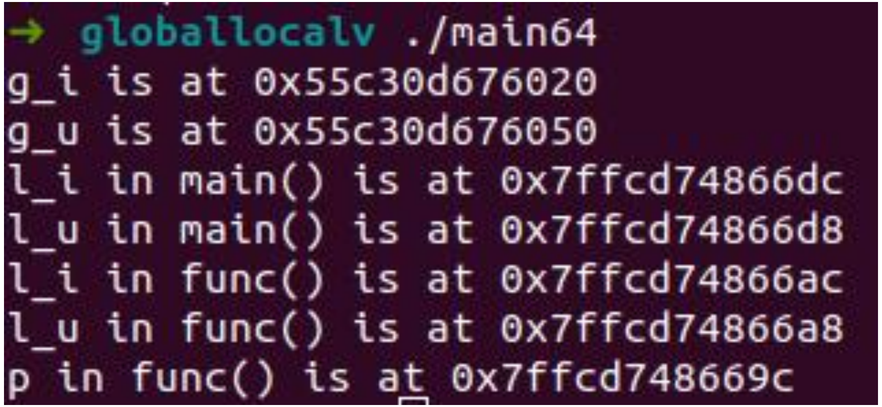
Tools: readelf; nm

Global and Local Variables (code/globallocalv 32bit)



```
ziming@ziming-ThinkPad:~/Dropbox/my
g_i is at 0x56558020
g_u is at 0x5655804c
l_i in main() is at 0xfff7c6d4
l_u in main() is at 0xfff7c6d8
l_i in func() is at 0xfff7c6a4
l_u in func() is at 0xfff7c6a8
p in func() is at 0xfff7c6c0
```

Global and Local Variables (code/globallocalv 64bit)



```
→  globallocalv ./main64
g_i is at 0x55c30d676020
g_u is at 0x55c30d676050
l_i in main() is at 0x7ffcd74866dc
l_u in main() is at 0x7ffcd74866d8
l_i in func() is at 0x7ffcd74866ac
l_u in func() is at 0x7ffcd74866a8
p in func() is at 0x7ffcd748669c
```

C/C++ Function in x86/64

What information do we need to call a function at runtime? Where are they stored?

- Code [.text]
- Parameters [mainly stack (32bit); registers + stack (64bit)]
- Return value [eax, rax]
- Global variables [.bss, .data]
- Local variables [stack; registers]
- Temporary variables [stack; registers]
- Return address [stack]
- Function frame pointer [ebp, rbp]
- Previous function Frame pointer [stack]

Stack

Stack is essentially scratch memory for functions

- Used in MIPS, ARM, x86, and x86-64 processors

Starts at high memory addresses, and grows down

Functions are free to push registers or values onto the stack, or pop values from the stack into registers

The assembly language supports this on x86

- **esp/rsp** holds the address of the top of the stack
- push eax/rax 1) decrements the stack pointer (esp/rbp) then 2) stores the value in eax/rax to the location pointed to by the stack pointer
- pop eax/rax 1) stores the value at the location pointed to by the stack pointer into eax/rax, then 2) increments the stack pointer (esp/rsp)

x86/64 Instructions that affect Stack

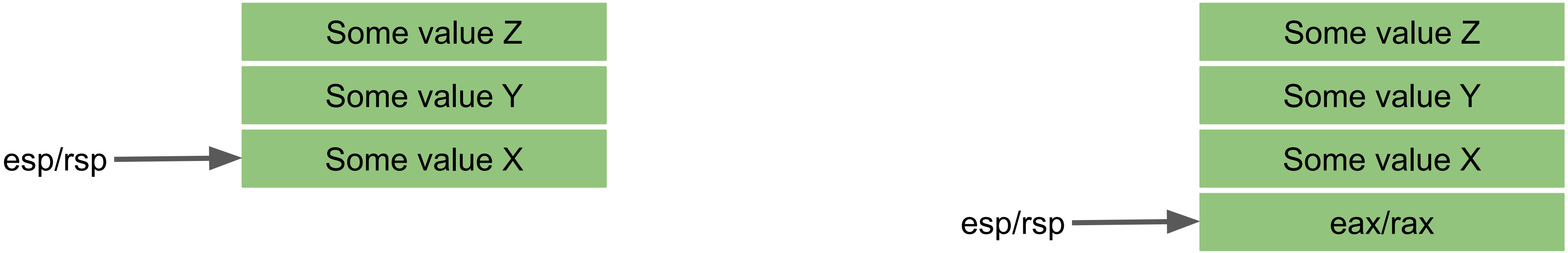
push, pop, call, ret, enter, leave

x86/64 Instructions that affect Stack

Before:

push eax/rax

After



Some value Z

Some value Y

esp/rsp → Some value X

Some value Z

Some value Y

Some value X

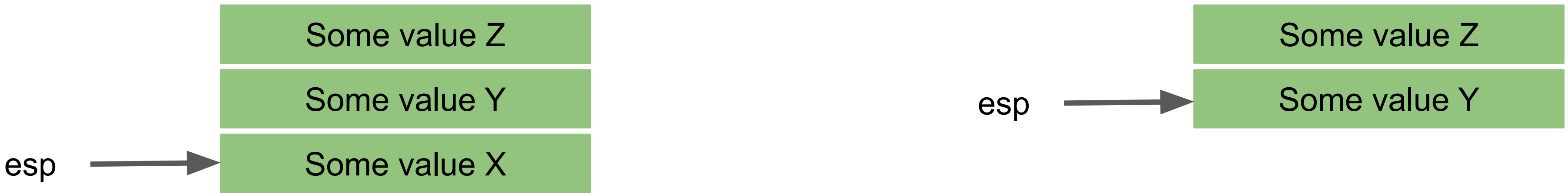
esp/rsp → eax/rax

x86 Instructions that affect Stack

Before:

pop eax

After: eax = X



Some value Z

Some value Y

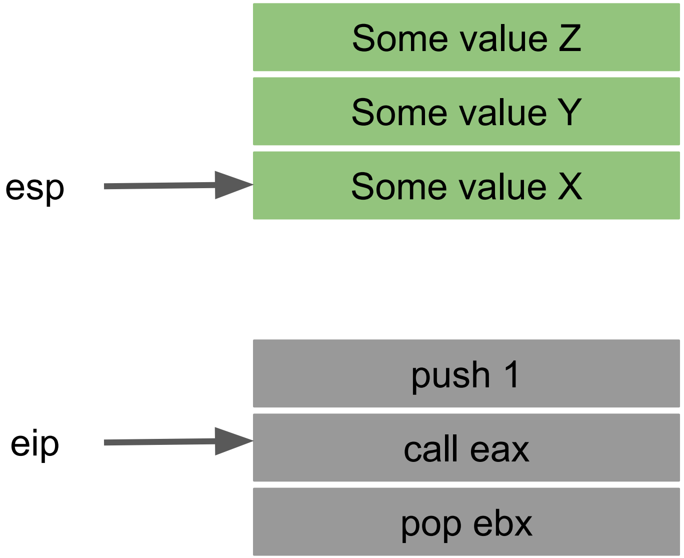
esp → Some value X

Some value Z

esp → Some value Y

x86 Instructions that affect Stack

Before:



Some value Z

Some value Y

esp → Some value X

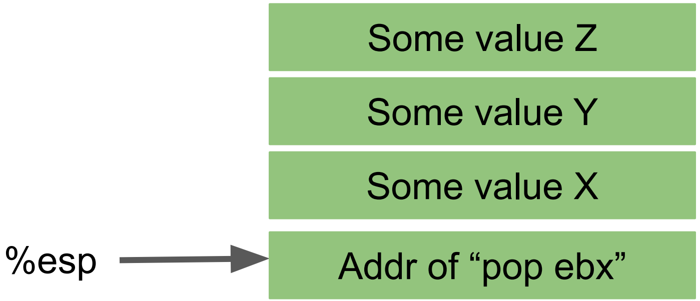
push 1

eip → call eax

pop ebx

call eax

After: eip = eax



Some value Z

Some value Y

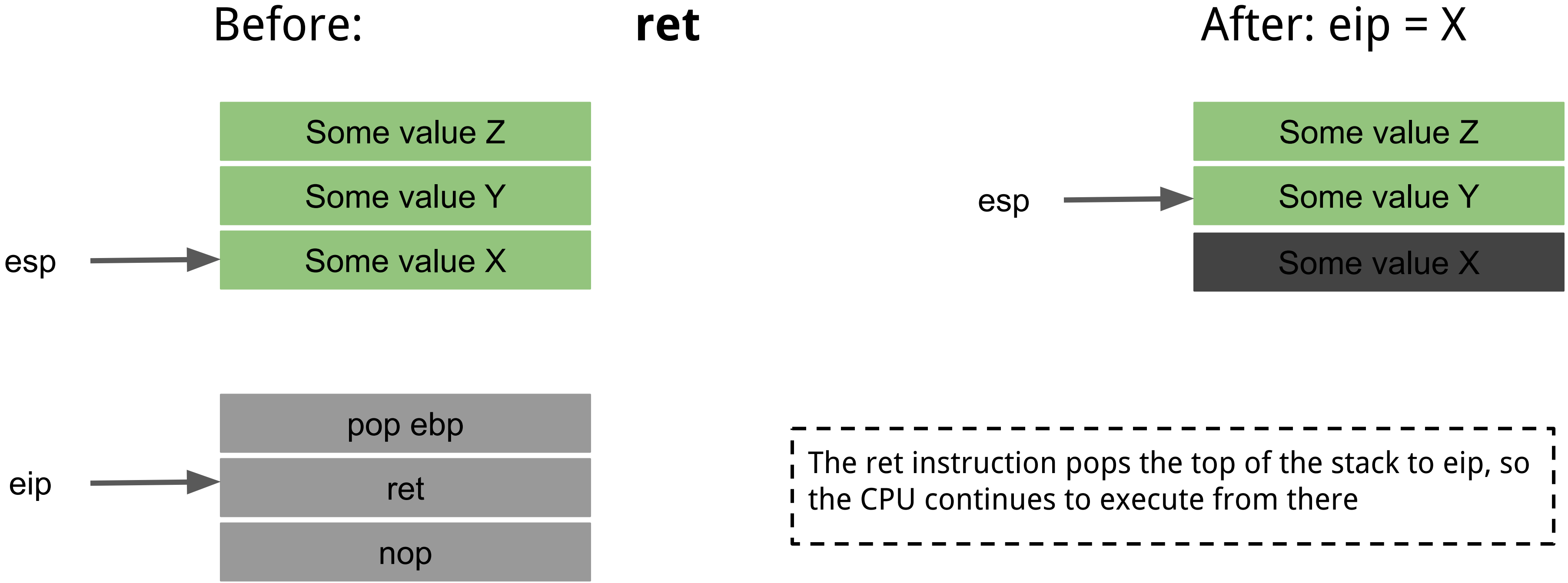
Some value X

%esp → Addr of “pop ebx”

The call instruction does two things:

1. Push the address of next instruction to the stack
2. Move the dest address to %eip

x86 Instructions that affect Stack



Before:

ret

After: eip = X

esp

Some value Z

Some value Y

Some value X

eip

pop ebp

ret

nop

esp

Some value Z

Some value Y

Some value X

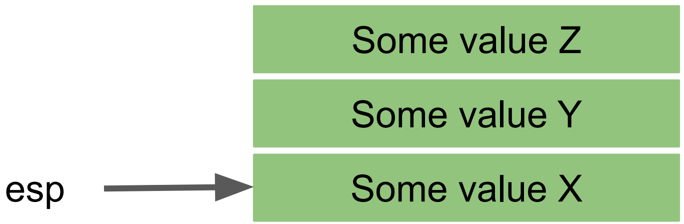
The ret instruction pops the top of the stack to eip, so the CPU continues to execute from there

x86 Instructions that affect Stack

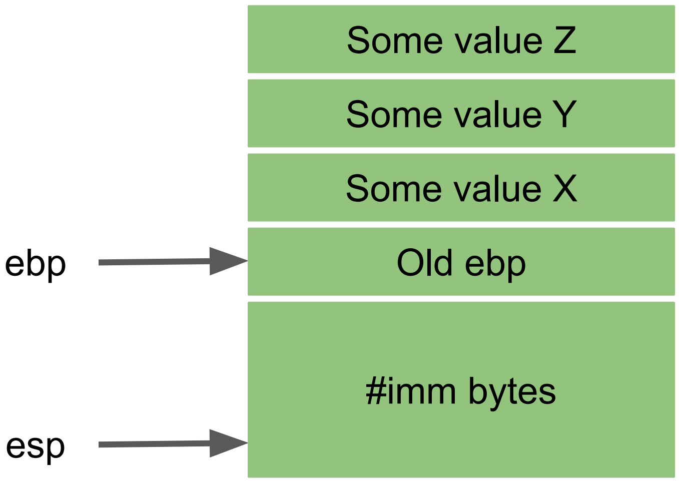
```
push ebp
mov ebp, esp
sub esp, #imm
```

enter

Before:



After:



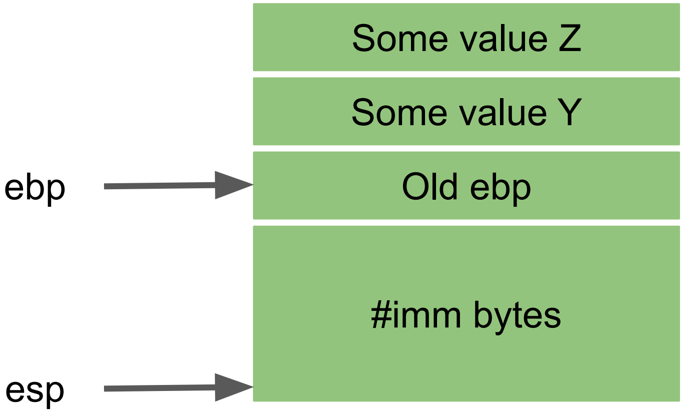
x86 Instructions that affect Stack

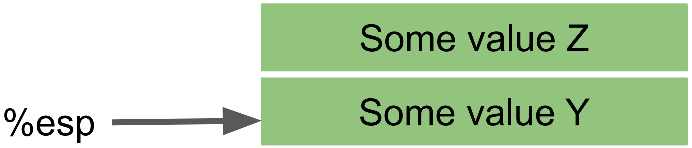
```
mov esp, ebp
pop ebp
```

Before:

leave

After: ebp = old ebp





Function Frame

Functions would like to use the stack to allocate space for their local variables. Can we use the stack pointer (esp/rsp) for this?

- Yes, however stack pointer can change throughout program execution

Frame pointer points to the start of the function's frame on the stack

- Each local variable will be (different) **offsets** of the frame pointer
- In x86/64, frame pointer is called the base pointer, and is stored in **ebp/rbp**

Function Frame

A function’s Stack Frame

- Starts with **where ebp/rbp points to**
- Ends with **where esp/rsp points to**

Calling Convention

Information, such as parameters, must be stored on the stack in order to call the function. Who should store that information? Caller? Callee?

Thus, we need to define a convention of who pushes/stores what values on the stack to call a function

- Varies based on processor, operating system, compiler, or type of call

x86 (32 bit) Linux Calling Convention (cdecl)

Caller (in this order)

- Pushes arguments onto the stack (in right to left order)
- Execute the **call** instruction (pushes address of instruction after call, then moves dest to **eip**)

Callee

- Pushes previous frame pointer onto stack (ebp)
- Setup new frame pointer (mov ebp, esp)
- Creates space on stack for local variables (sub esp, #imm)
- Ensures that stack is consistent on return
- Return value in **eax** register

Callee Allocate a stack (Function prologue)

Three instructions:

push ebp; (Pushes previous frame pointer onto stack)
mov ebp, esp; (change the base pointer to the stack)
sub esp, 10; (allocating a local stack space)

Callee Deallocate a stack (Function epilogue)

```
mov esp, ebp
pop ebp
ret
```

Global and Local Variables (code/globallocalv)

```
int func(int p)
{
  int l_i = 10;
  int l_u;

  printf("l_i in func() is at %p\n", &l_i);
  printf("l_u in func() is at %p\n", &l_u);
  printf("p in func() is at %p\n", &p);
  return 0;
}
```

Function main()

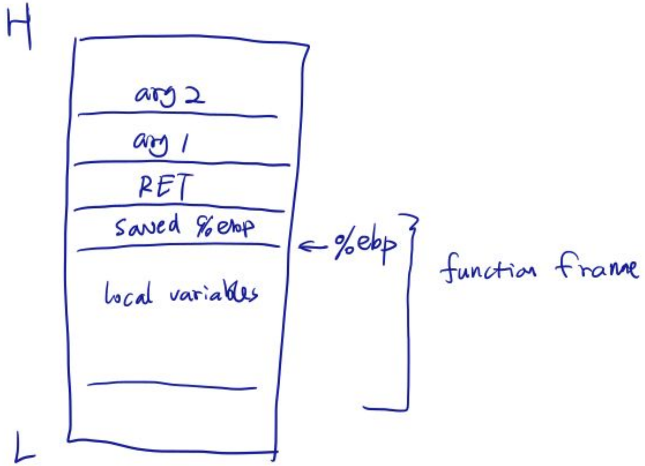
```
657:   83 ec 0c              sub    esp,0xc
 65a:   6a 0a                push   0xa
 65c:   e8 3c ff ff ff        call   59d <func>
 661:   83 c4 10              add    esp,0x10
```

Function func()

```
59d:   55                    push   ebp
59e:   89 e5                 mov    ebp,esp
5a0:   83 ec 18              sub    esp,0x18
5a3:   c7 45 f4 0a 00 00 00  mov    DWORD PTR [ebp-0xc],0xa
5aa:   83 ec 08              sub    esp,0x8
5ad:   8d 45 f4              lea    eax,[ebp-0xc]
5b0:   50                    push   eax
5b1:   68 00 07 00 00        push   0x700
5b6:   e8 fc ff ff ff        call   5b7 <func+0x1a>
5bb:   83 c4 10              add    esp,0x10
5be:   83 ec 08              sub    esp,0x8
5c1:   8d 45 f0              lea    eax,[ebp-0x10]
5c4:   50                    push   eax
5c5:   68 18 07 00 00        push   0x718
5ca:   e8 fc ff ff ff        call   5cb <func+0x2e>
5cf:   83 c4 10              add    esp,0x10
5d2:   83 ec 08              sub    esp,0x8
5d5:   8d 45 08              lea    eax,[ebp+0x8]
5d8:   50                    push   eax
5d9:   68 30 07 00 00        push   0x730
5de:   e8 fc ff ff ff        call   5df <func+0x42>
5e3:   83 c4 10              add    esp,0x10
5e6:   b8 00 00 00 00        mov    eax,0x0
5eb:   c9                    leave
5ec:   c3                    ret
```

Draw the stack (x86 cdecl)

x86 , cdel in a function



(%ebp) : saved %ebp

4(%ebp) : RET

8(%ebp) : first argument

-8(%ebp) : maybe a local variable

x86 Stack Usage (32bit)

- Negative indexing over ebp

```
mov eax, [ebp-0x8]
lea eax, [ebp-24]
```

- Positive indexing over ebp

```
mov eax, [ebp+8]
mov eax, [ebp+0xc]
```

- Positive indexing over esp

x86 Stack Usage (32bit)

- Accesses local variables (negative indexing over ebp)

mov eax, ebp-0x8 value at ebp-0x8
lea eax, ebp-24 address as ebp-0x24

- Stores function arguments from caller (positive indexing over ebp)

mov eax, ebp+8 1st arg
mov eax, ebp+0xc 2nd arg

- Positive indexing over esp

Function arguments to callee

Stack example: code/factorial

```
int fact(int n)
{
  printf("---In fact(%d)\n", n);
  printf("&n is %p\n", &n);

  if (n <= 1)
    return 1;

  return fact(n-1) * n;
}
```

```
int main(int argc, char *argv[])
{
  if (argc != 2)
  {
    printf("Usage: fact integer\n");
    return 0;
  }

  printf("The factorial of %d is %d\n.",
atoi(argv[1]), fact(atoi(argv[1])));
}
```

Stack example: code/fivepara

```
int func(int a, int b, int c, int d, int e)
{
  return a + b + c + d + e;
}

int main(int argc, char *argv[])
{
  func(1, 2, 3, 4, 5);
}
```

X86 disassembly

globallocalv_fast_32

fastcall

On x86-32 targets, the fastcall attribute causes the compiler to pass the first argument (if of integral type) in the register ECX and the second argument (if of integral type) in the register EDX. Subsequent and other typed arguments are passed on the stack. The called function pops the arguments off the stack. If the number of arguments is variable all arguments are pushed on the stack.

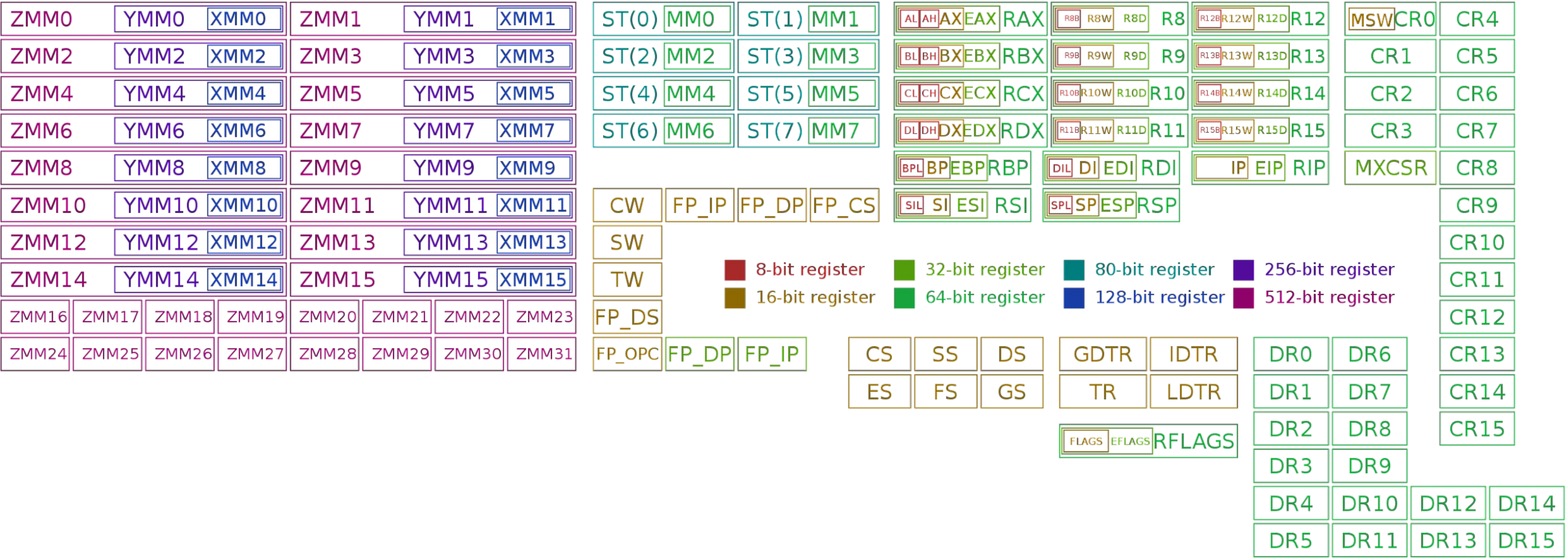
```
int __attribute__ ((fastcall)) func(int p)
```

x86-64 (64 bit) Linux Calling Convention

Caller

- Use registers to pass arguments to callee. Register order (1st, 2nd, 3rd, 4th, 5th, 6th, etc.) rdi, rsi, rdx, rcx, r8, r9, ... (use stack for more arguments)

Registers on x86-64



ZMM0 YMM0 XMM0 ZMM1 YMM1 XMM1 ST(0) MM0 ST(1) MM1 AL AH AX EAX RAX R8B R8W R8D R8 R12B R12W R12D R12 MSW CR0 CR4

ZMM2 YMM2 XMM2 ZMM3 YMM3 XMM3 ST(2) MM2 ST(3) MM3 BL BH BX EBX RBX R9B R9W R9D R9 R13B R13W R13D R13 CR1 CR5

ZMM4 YMM4 XMM4 ZMM5 YMM5 XMM5 ST(4) MM4 ST(5) MM5 CL CH CX ECX RCX R10B R10W R10D R10 R14B R14W R14D R14 CR2 CR6

ZMM6 YMM6 XMM6 ZMM7 YMM7 XMM7 ST(6) MM6 ST(7) MM7 DL DH DX EDX RDX R11B R11W R11D R11 R15B R15W R15D R15 CR3 CR7

ZMM8 YMM8 XMM8 ZMM9 YMM9 XMM9 BPL BP EBP RBP DIL DI EDI RDI IP EIP RIP MXCSR CR8

ZMM10 YMM10 XMM10 ZMM11 YMM11 XMM11 CW FP_IP FP_DP FP_CS SIL SI ESI RSI SPL SP ESP RSP CR9

ZMM12 YMM12 XMM12 ZMM13 YMM13 XMM13 SW CR10

ZMM14 YMM14 XMM14 ZMM15 YMM15 XMM15 TW CR11

ZMM16 ZMM17 ZMM18 ZMM19 ZMM20 ZMM21 ZMM22 ZMM23 FP_DS CR12

ZMM24 ZMM25 ZMM26 ZMM27 ZMM28 ZMM29 ZMM30 ZMM31 FP_OPC FP_DP FP_IP CS SS DS GDTR IDTR DR0 DR6 CR13

ES FS GS TR LDTR DR1 DR7 CR14

FLAGS EFLAGS RFLAGS DR2 DR8 CR15

DR3 DR9

DR4 DR10 DR12 DR14

DR5 DR11 DR13 DR15

🟥 8-bit register
🟫 16-bit register
🟩 32-bit register
🟩 64-bit register
🟦 80-bit register
🟦 128-bit register
🟪 256-bit register
🟪 512-bit register

https://en.wikipedia.org/wiki/X86

Stack example: code/fivepara

```
int func(int a, int b, int c, int d, int e)
{
  return a + b + c + d + e;
}

int main(int argc, char *argv[])
{
  func(1, 2, 3, 4, 5);
}
```

X86-64 disassembly

X86-64 Stack Usage

- Access local variables (negative indexing over rbp)

```
mov rax, [rbp-8]
lea rax, [rbp-0x24]
```

- Access function arguments from caller

```
mov rax, rdi
```

- Setup parameters for callee

```
mov rdi, rax
```

Overwrite Local Variables

Data-only Attack

Buffer Overflow Example: code/overflowlocal

```
int vulfoo(int i, char* p)
{
  int j = i;
  char buf[6];

  strcpy(buf, p);

  if (j)
    print_flag();
  else
    printf("I pity the fool!\n");

  return 0;
}

int main(int argc, char *argv[])
{
  if (argc == 2)
    vulfoo(0, argv[1]);
}
```

```
000012c4 <vulfoo>:
    12c4:     55                     push   ebp
    12c5:     89 e5                  mov    ebp,esp
    12c7:     83 ec 18               sub    esp,0x18
    12ca:     8b 45 08               mov    eax,DWORD PTR [ebp+0x8]
    12cd:     89 45 f4               mov    DWORD PTR [ebp-0xc],eax
    12d0:     83 ec 08               sub    esp,0x8
    12d3:     ff 75 0c               push   DWORD PTR [ebp+0xc]
    12d6:     8d 45 ee               lea    eax,[ebp-0x12]
    12d9:     50                     push   eax
    12da:     e8 fc ff ff ff         call   12db <vulfoo+0x17>
    12df:     83 c4 10               add    esp,0x10
    12e2:     83 7d f4 00            cmp    DWORD PTR [ebp-0xc],0x0
    12e6:     74 07                  je     12ef <vulfoo+0x2b>
    12e8:     e8 10 ff ff ff         call   11fd <print_flag>
    12ed:     eb 10                  jmp    12ff <vulfoo+0x3b>
    12ef:     83 ec 0c               sub    esp,0xc
    12f2:     68 45 20 00 00         push   0x2045
    12f7:     e8 fc ff ff ff         call   12f8 <vulfoo+0x34>
    12fc:     83 c4 10               add    esp,0x10
    12ff:     b8 00 00 00 00         mov    eax,0x0
    1304:     c9                     leave
    1305:     c3                     ret
```

Implementations of strcpy()

```
char *strcpy(char *dest, const char *src)
{
  unsigned i;
  for (i=0; src[i] != '\0'; ++i)
    dest[i] = src[i];

  //Ensure trailing null byte is copied
  dest[i]= '\0';

  return dest;
}
```

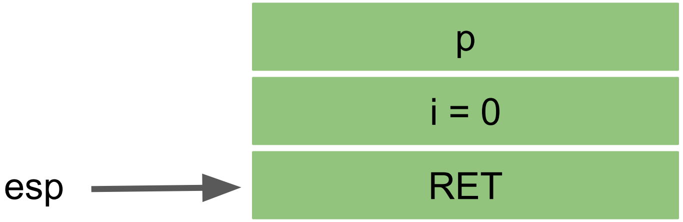
Implementations of strcpy()

```c
char *strcpy(char *dest, const char *src)
{
  unsigned i;
  for (i=0; src[i] != '\0'; ++i)
    dest[i] = src[i];

  //Ensure trailing null byte is copied
  dest[i]= '\0';

  return dest;
}
```

```c
char *strcpy(char *dest, const char *src)
{
   char *save = dest;
   while(*dest++ = *src++);
   return save;
}
```

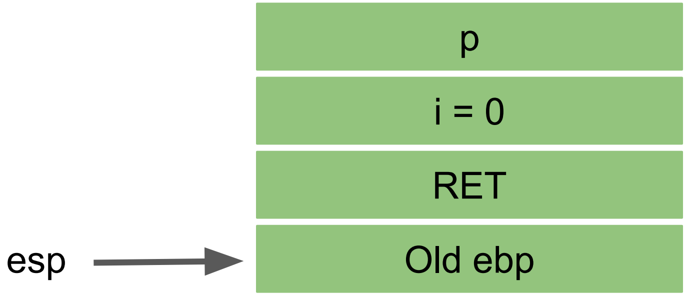
Buffer Overflow Example: code/overflowlocal

```
000012c4 <vulfoo>:
    12c4:   55                  push   ebp
    12c5:   89 e5               mov    ebp,esp
    12c7:   83 ec 18            sub    esp,0x18
    12ca:   8b 45 08            mov    eax,DWORD PTR [ebp+0x8]
    12cd:   89 45 f4            mov    DWORD PTR [ebp-0xc],eax
    12d0:   83 ec 08            sub    esp,0x8
    12d3:   ff 75 0c            push   DWORD PTR [ebp+0xc]
    12d6:   8d 45 ee            lea    eax,[ebp-0x12]
    12d9:   50                  push   eax
    12da:   e8 fc ff ff ff      call   12db <vulfoo+0x17>
    12df:   83 c4 10            add    esp,0x10
    12e2:   83 7d f4 00         cmp    DWORD PTR [ebp-0xc],0x0
    12e6:   74 07               je     12ef <vulfoo+0x2b>
    12e8:   e8 10 ff ff ff      call   11fd <print_flag>
    12ed:   eb 10               jmp    12ff <vulfoo+0x3b>
    12ef:   83 ec 0c            sub    esp,0xc
    12f2:   68 45 20 00 00      push   0x2045
    12f7:   e8 fc ff ff ff      call   12f8 <vulfoo+0x34>
    12fc:   83 c4 10            add    esp,0x10
    12ff:   b8 00 00 00 00      mov    eax,0x0
    1304:   c9                  leave
    1305:   c3                  ret
```



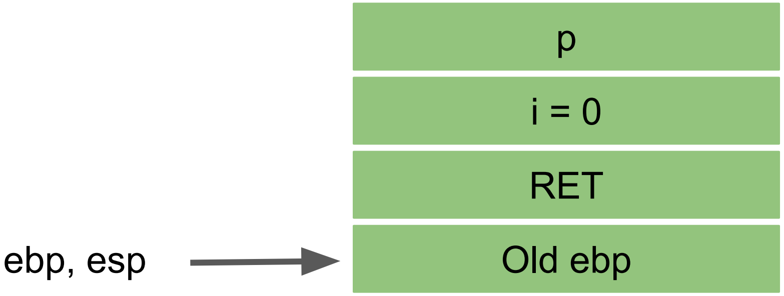
Buffer Overflow Example: code/overflowlocal

```
000012c4 <vulfoo>:
 12c4:    55                   push   ebp
 12c5:    89 e5                mov    ebp,esp
 12c7:    83 ec 18             sub    esp,0x18
 12ca:    8b 45 08             mov    eax,DWORD PTR [ebp+0x8]
 12cd:    89 45 f4             mov    DWORD PTR [ebp-0xc],eax
 12d0:    83 ec 08             sub    esp,0x8
 12d3:    ff 75 0c             push   DWORD PTR [ebp+0xc]
 12d6:    8d 45 ee             lea    eax,[ebp-0x12]
 12d9:    50                   push   eax
 12da:    e8 fc ff ff ff       call   12db <vulfoo+0x17>
 12df:    83 c4 10             add    esp,0x10
 12e2:    83 7d f4 00          cmp    DWORD PTR [ebp-0xc],0x0
 12e6:    74 07                je     12ef <vulfoo+0x2b>
 12e8:    e8 10 ff ff ff       call   11fd <print_flag>
 12ed:    eb 10                jmp    12ff <vulfoo+0x3b>
 12ef:    83 ec 0c             sub    esp,0xc
 12f2:    68 45 20 00 00       push   0x2045
 12f7:    e8 fc ff ff ff       call   12f8 <vulfoo+0x34>
 12fc:    83 c4 10             add    esp,0x10
 12ff:    b8 00 00 00 00       mov    eax,0x0
 1304:    c9                   leave
 1305:    c3                   ret
```



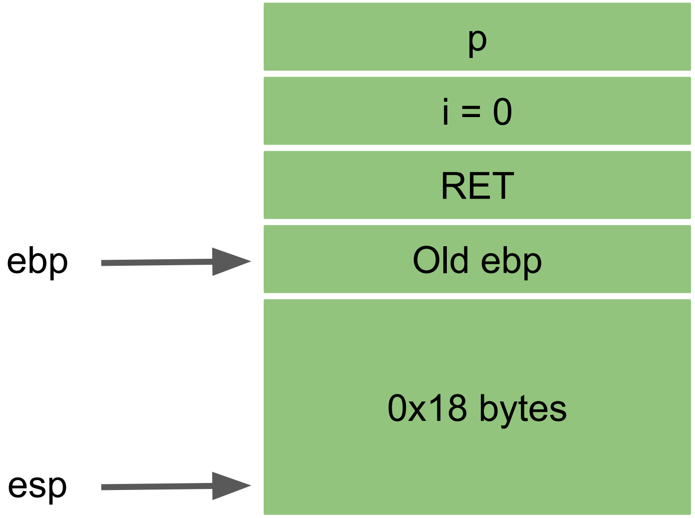
Buffer Overflow Example: code/overflowlocal

```
000012c4 <vulfoo>:
    12c4:    55                     push   ebp
    12c5:    89 e5                  mov    ebp,esp
    12c7:    83 ec 18               sub    esp,0x18
    12ca:    8b 45 08               mov    eax,DWORD PTR [ebp+0x8]
    12cd:    89 45 f4               mov    DWORD PTR [ebp-0xc],eax
    12d0:    83 ec 08               sub    esp,0x8
    12d3:    ff 75 0c               push   DWORD PTR [ebp+0xc]
    12d6:    8d 45 ee               lea    eax,[ebp-0x12]
    12d9:    50                     push   eax
    12da:    e8 fc ff ff ff         call   12db <vulfoo+0x17>
    12df:    83 c4 10               add    esp,0x10
    12e2:    83 7d f4 00            cmp    DWORD PTR [ebp-0xc],0x0
    12e6:    74 07                  je     12ef <vulfoo+0x2b>
    12e8:    e8 10 ff ff ff         call   11fd <print_flag>
    12ed:    eb 10                  jmp    12ff <vulfoo+0x3b>
    12ef:    83 ec 0c               sub    esp,0xc
    12f2:    68 45 20 00 00         push   0x2045
    12f7:    e8 fc ff ff ff         call   12f8 <vulfoo+0x34>
    12fc:    83 c4 10               add    esp,0x10
    12ff:    b8 00 00 00 00         mov    eax,0x0
    1304:    c9                     leave
    1305:    c3                     ret
```



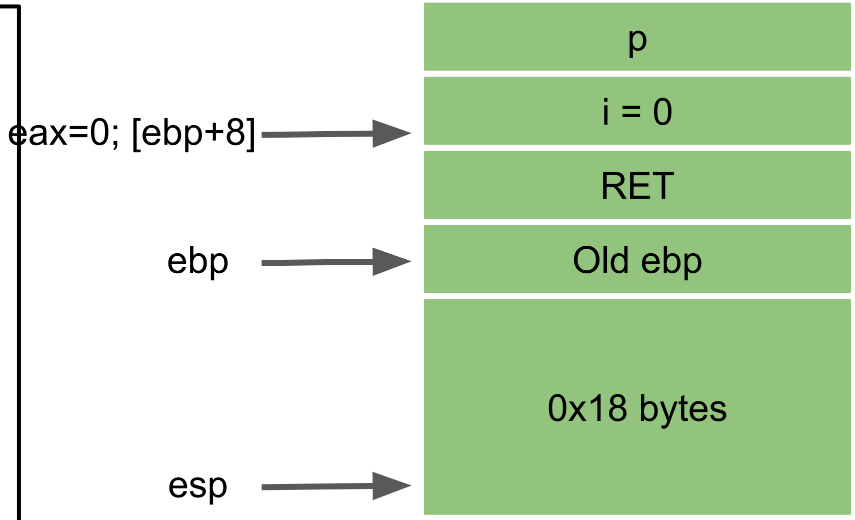
Buffer Overflow Example: code/overflowlocal

```
000012c4 <vulfoo>:
    12c4:     55                      push   ebp
    12c5:     89 e5                   mov    ebp,esp
    12c7:     83 ec 18                sub    esp,0x18
    12ca:     8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
    12cd:     89 45 f4                mov    DWORD PTR [ebp-0xc],eax
    12d0:     83 ec 08                sub    esp,0x8
    12d3:     ff 75 0c                push   DWORD PTR [ebp+0xc]
    12d6:     8d 45 ee                lea    eax,[ebp-0x12]
    12d9:     50                      push   eax
    12da:     e8 fc ff ff ff          call   12db <vulfoo+0x17>
    12df:     83 c4 10                add    esp,0x10
    12e2:     83 7d f4 00             cmp    DWORD PTR [ebp-0xc],0x0
    12e6:     74 07                   je     12ef <vulfoo+0x2b>
    12e8:     e8 10 ff ff ff          call   11fd <print_flag>
    12ed:     eb 10                   jmp    12ff <vulfoo+0x3b>
    12ef:     83 ec 0c                sub    esp,0xc
    12f2:     68 45 20 00 00          push   0x2045
    12f7:     e8 fc ff ff ff          call   12f8 <vulfoo+0x34>
    12fc:     83 c4 10                add    esp,0x10
    12ff:     b8 00 00 00 00          mov    eax,0x0
    1304:     c9                      leave
    1305:     c3                      ret
```



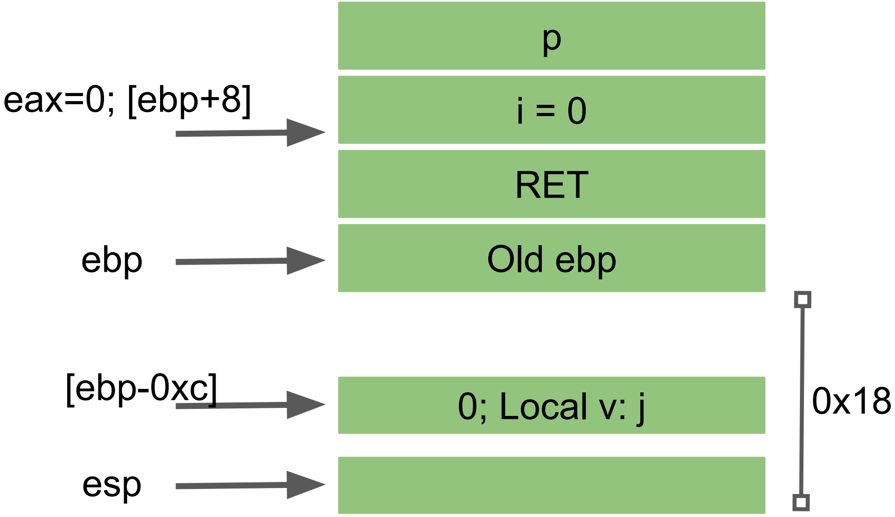
Buffer Overflow Example: code/overflowlocal

```
000012c4 <vulfoo>:
    12c4:      55                   push   ebp
    12c5:      89 e5                mov    ebp,esp
    12c7:      83 ec 18             sub    esp,0x18
    12ca:      8b 45 08             mov    eax,DWORD PTR [ebp+0x8]
    12cd:      89 45 f4             mov    DWORD PTR [ebp-0xc],eax
    12d0:      83 ec 08             sub    esp,0x8
    12d3:      ff 75 0c             push   DWORD PTR [ebp+0xc]
    12d6:      8d 45 ee             lea    eax,[ebp-0x12]
    12d9:      50                   push   eax
    12da:      e8 fc ff ff ff       call   12db <vulfoo+0x17>
    12df:      83 c4 10             add    esp,0x10
    12e2:      83 7d f4 00          cmp    DWORD PTR [ebp-0xc],0x0
    12e6:      74 07                je     12ef <vulfoo+0x2b>
    12e8:      e8 10 ff ff ff       call   11fd <print_flag>
    12ed:      eb 10                jmp    12ff <vulfoo+0x3b>
    12ef:      83 ec 0c             sub    esp,0xc
    12f2:      68 45 20 00 00       push   0x2045
    12f7:      e8 fc ff ff ff       call   12f8 <vulfoo+0x34>
    12fc:      83 c4 10             add    esp,0x10
    12ff:      b8 00 00 00 00       mov    eax,0x0
    1304:      c9                   leave
    1305:      c3                   ret
```



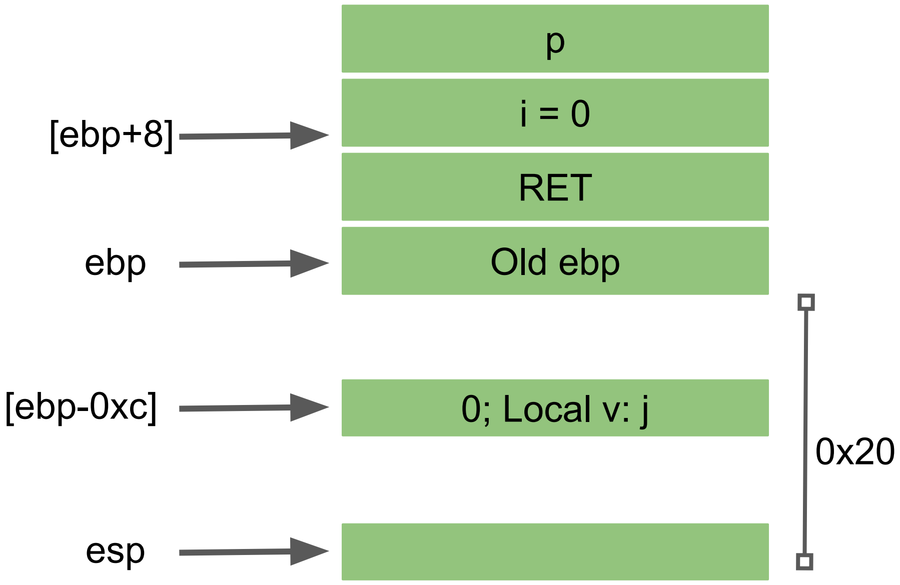
Buffer Overflow Example: code/overflowlocal

```
000012c4 <vulfoo>:
   12c4:   55                   push   ebp
   12c5:   89 e5                mov    ebp,esp
   12c7:   83 ec 18             sub    esp,0x18
   12ca:   8b 45 08             mov    eax,DWORD PTR [ebp+0x8]
   12cd:   89 45 f4             mov    DWORD PTR [ebp-0xc],eax
   12d0:   83 ec 08             sub    esp,0x8
   12d3:   ff 75 0c             push   DWORD PTR [ebp+0xc]
   12d6:   8d 45 ee             lea    eax,[ebp-0x12]
   12d9:   50                   push   eax
   12da:   e8 fc ff ff ff       call   12db <vulfoo+0x17>
   12df:   83 c4 10             add    esp,0x10
   12e2:   83 7d f4 00          cmp    DWORD PTR [ebp-0xc],0x0
   12e6:   74 07                je     12ef <vulfoo+0x2b>
   12e8:   e8 10 ff ff ff       call   11fd <print_flag>
   12ed:   eb 10                jmp    12ff <vulfoo+0x3b>
   12ef:   83 ec 0c             sub    esp,0xc
   12f2:   68 45 20 00 00       push   0x2045
   12f7:   e8 fc ff ff ff       call   12f8 <vulfoo+0x34>
   12fc:   83 c4 10             add    esp,0x10
   12ff:   b8 00 00 00 00       mov    eax,0x0
   1304:   c9                   leave
   1305:   c3                   ret
```



Buffer Overflow Example: code/overflowlocal

```
000012c4 <vulfoo>:
    12c4:      55                     push   ebp
    12c5:      89 e5                  mov    ebp,esp
    12c7:      83 ec 18               sub    esp,0x18
    12ca:      8b 45 08               mov    eax,DWORD PTR [ebp+0x8]
    12cd:      89 45 f4               mov    DWORD PTR [ebp-0xc],eax
    12d0:      83 ec 08               sub    esp,0x8
    12d3:      ff 75 0c               push   DWORD PTR [ebp+0xc]
    12d6:      8d 45 ee               lea    eax,[ebp-0x12]
    12d9:      50                     push   eax
    12da:      e8 fc ff ff ff         call   12db <vulfoo+0x17>
    12df:      83 c4 10               add    esp,0x10
    12e2:      83 7d f4 00            cmp    DWORD PTR [ebp-0xc],0x0
    12e6:      74 07                  je     12ef <vulfoo+0x2b>
    12e8:      e8 10 ff ff ff         call   11fd <print_flag>
    12ed:      eb 10                  jmp    12ff <vulfoo+0x3b>
    12ef:      83 ec 0c               sub    esp,0xc
    12f2:      68 45 20 00 00         push   0x2045
    12f7:      e8 fc ff ff ff         call   12f8 <vulfoo+0x34>
    12fc:      83 c4 10               add    esp,0x10
    12ff:      b8 00 00 00 00         mov    eax,0x0
    1304:      c9                     leave
    1305:      c3                     ret
```



p

[ebp+8] → i = 0

RET

ebp → Old ebp

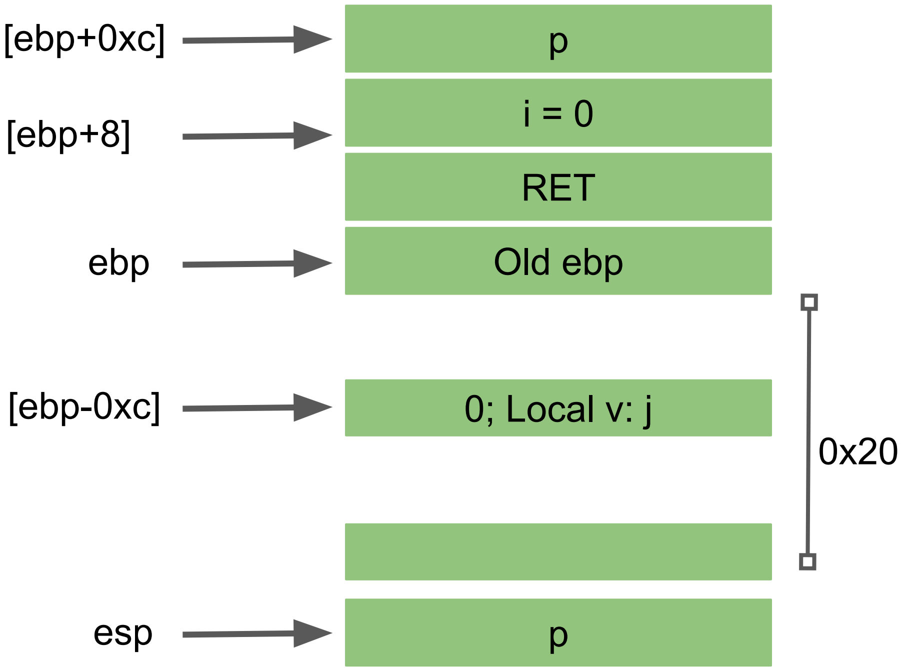
[ebp-0xc] → 0; Local v: j

esp →

0x20

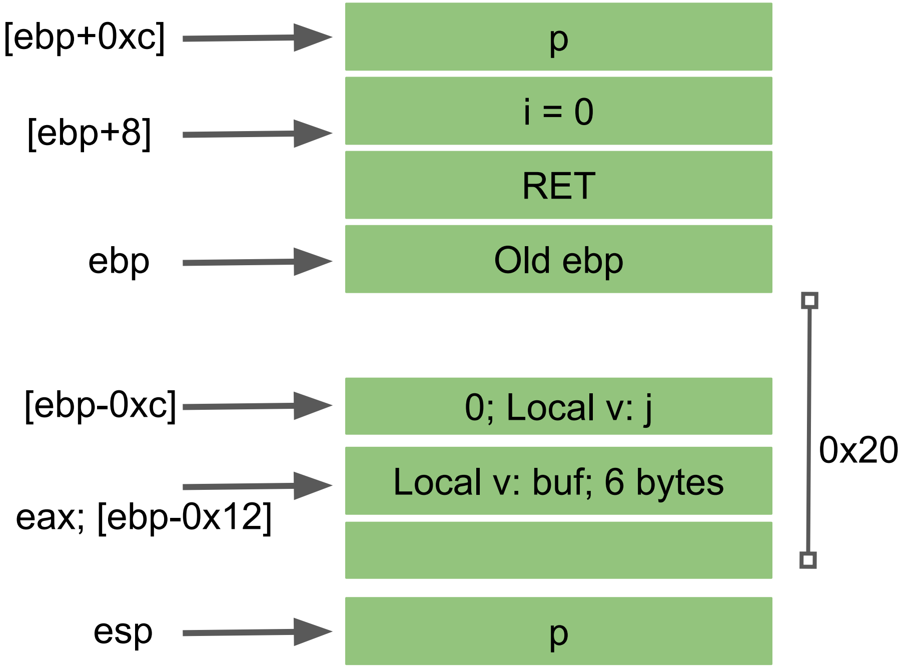
Buffer Overflow Example: code/overflowlocal

```
000012c4 <vulfoo>:
   12c4:    55                  push   ebp
   12c5:    89 e5                 mov    ebp,esp
   12c7:    83 ec 18               sub    esp,0x18
   12ca:    8b 45 08               mov    eax,DWORD PTR [ebp+0x8]
   12cd:    89 45 f4               mov    DWORD PTR [ebp-0xc],eax
   12d0:    83 ec 08               sub    esp,0x8
   12d3:    ff 75 0c              push   DWORD PTR [ebp+0xc]
   12d6:    8d 45 ee               lea    eax,[ebp-0x12]
   12d9:    50                  push   eax
   12da:    e8 fc ff ff ff        call   12db <vulfoo+0x17>
   12df:    83 c4 10              add    esp,0x10
   12e2:    83 7d f4 00             cmp    DWORD PTR [ebp-0xc],0x0
   12e6:    74 07                je     12ef <vulfoo+0x2b>
   12e8:    e8 10 ff ff ff         call   11fd <print_flag>
   12ed:    eb 10                jmp    12ff <vulfoo+0x3b>
   12ef:    83 ec 0c              sub    esp,0xc
   12f2:    68 45 20 00 00          push   0x2045
   12f7:    e8 fc ff ff ff       call   12f8 <vulfoo+0x34>
   12fc:    83 c4 10              add    esp,0x10
   12ff:    b8 00 00 00 00          mov    eax,0x0
   1304:    c9                  leave
   1305:    c3                  ret
```



Buffer Overflow Example: code/overflowlocal

```
000012c4 <vulfoo>:
    12c4:   55                  push   ebp
    12c5:   89 e5               mov    ebp,esp
    12c7:   83 ec 18            sub    esp,0x18
    12ca:   8b 45 08            mov    eax,DWORD PTR [ebp+0x8]
    12cd:   89 45 f4            mov    DWORD PTR [ebp-0xc],eax
    12d0:   83 ec 08            sub    esp,0x8
    12d3:   ff 75 0c            push   DWORD PTR [ebp+0xc]
    12d6:   8d 45 ee            lea    eax,[ebp-0x12]
    12d9:   50                  push   eax
    12da:   e8 fc ff ff ff      call   12db <vulfoo+0x17>
    12df:   83 c4 10            add    esp,0x10
    12e2:   83 7d f4 00         cmp    DWORD PTR [ebp-0xc],0x0
    12e6:   74 07               je     12ef <vulfoo+0x2b>
    12e8:   e8 10 ff ff ff      call   11fd <print_flag>
    12ed:   eb 10               jmp    12ff <vulfoo+0x3b>
    12ef:   83 ec 0c            sub    esp,0xc
    12f2:   68 45 20 00 00      push   0x2045
    12f7:   e8 fc ff ff ff      call   12f8 <vulfoo+0x34>
    12fc:   83 c4 10            add    esp,0x10
    12ff:   b8 00 00 00 00      mov    eax,0x0
    1304:   c9                  leave
    1305:   c3                  ret
```



[ebp+0xc]

[ebp+8]

ebp

[ebp-0xc]

eax; [ebp-0x12]

esp

p

i = 0

RET

Old ebp

0; Local v: j

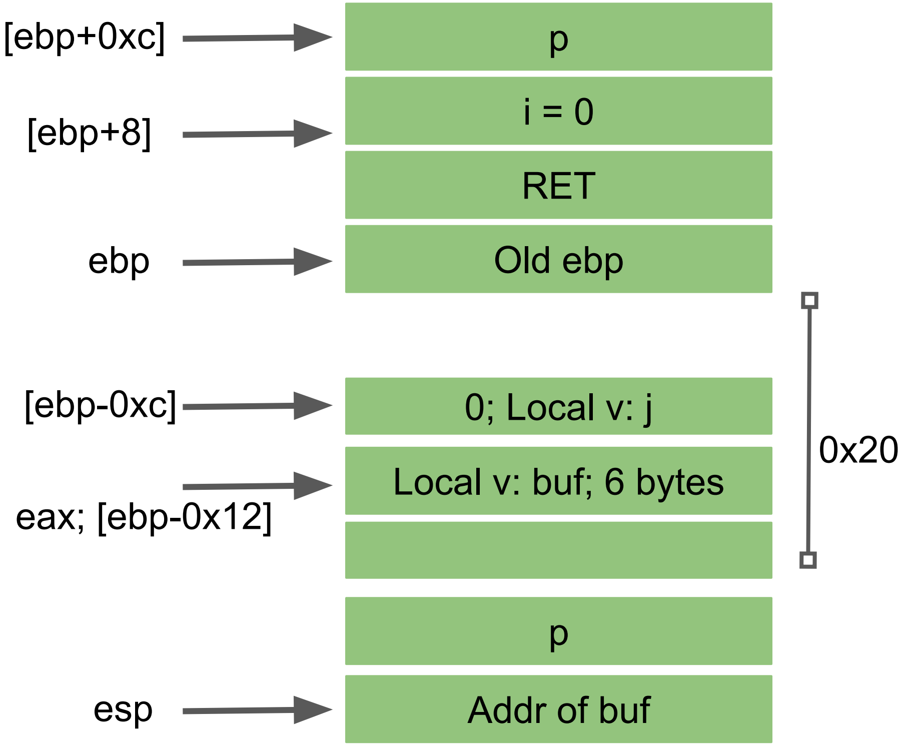
Local v: buf; 6 bytes

p

0x20

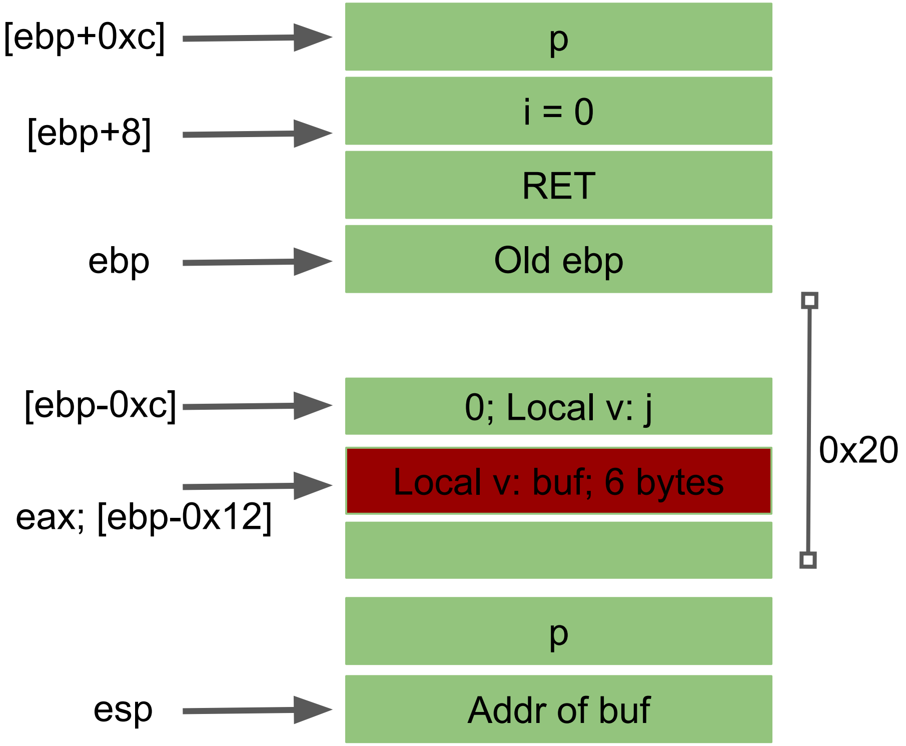
Buffer Overflow Example: code/overflowlocal

```
000012c4 <vulfoo>:
    12c4:      55                   push   ebp
    12c5:      89 e5                mov    ebp,esp
    12c7:      83 ec 18             sub    esp,0x18
    12ca:      8b 45 08             mov    eax,DWORD PTR [ebp+0x8]
    12cd:      89 45 f4             mov    DWORD PTR [ebp-0xc],eax
    12d0:      83 ec 08             sub    esp,0x8
    12d3:      ff 75 0c             push   DWORD PTR [ebp+0xc]
    12d6:      8d 45 ee             lea    eax,[ebp-0x12]
    12d9:      50                   push   eax
    12da:      e8 fc ff ff ff       call   12db <vulfoo+0x17>
    12df:      83 c4 10             add    esp,0x10
    12e2:      83 7d f4 00          cmp    DWORD PTR [ebp-0xc],0x0
    12e6:      74 07                je     12ef <vulfoo+0x2b>
    12e8:      e8 10 ff ff ff       call   11fd <print_flag>
    12ed:      eb 10                jmp    12ff <vulfoo+0x3b>
    12ef:      83 ec 0c             sub    esp,0xc
    12f2:      68 45 20 00 00       push   0x2045
    12f7:      e8 fc ff ff ff       call   12f8 <vulfoo+0x34>
    12fc:      83 c4 10             add    esp,0x10
    12ff:      b8 00 00 00 00       mov    eax,0x0
    1304:      c9                   leave
    1305:      c3                   ret
```



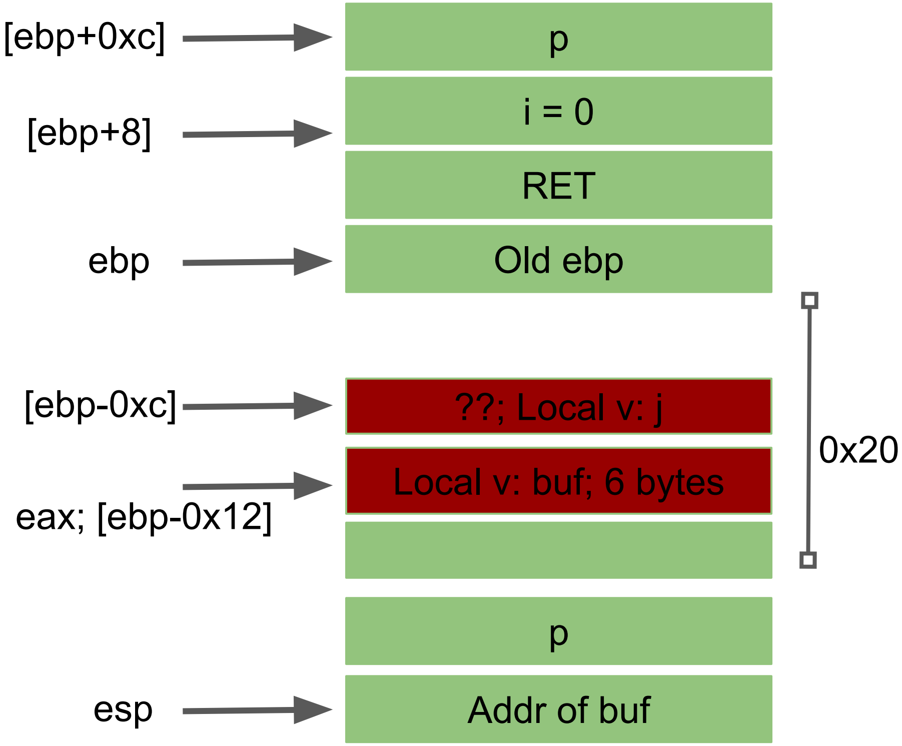
Buffer Overflow Example: code/overflowlocal

```
000012c4 <vulfoo>:
    12c4:    55                    push   ebp
    12c5:    89 e5                 mov    ebp,esp
    12c7:    83 ec 18              sub    esp,0x18
    12ca:    8b 45 08              mov    eax,DWORD PTR [ebp+0x8]
    12cd:    89 45 f4              mov    DWORD PTR [ebp-0xc],eax
    12d0:    83 ec 08              sub    esp,0x8
    12d3:    ff 75 0c              push   DWORD PTR [ebp+0xc]
    12d6:    8d 45 ee              lea    eax,[ebp-0x12]
    12d9:    50                    push   eax
    12da:    e8 fc ff ff ff        call   12db <vulfoo+0x17>
    12df:    83 c4 10              add    esp,0x10
    12e2:    83 7d f4 00           cmp    DWORD PTR [ebp-0xc],0x0
    12e6:    74 07                 je     12ef <vulfoo+0x2b>
    12e8:    e8 10 ff ff ff        call   11fd <print_flag>
    12ed:    eb 10                 jmp    12ff <vulfoo+0x3b>
    12ef:    83 ec 0c              sub    esp,0xc
    12f2:    68 45 20 00 00        push   0x2045
    12f7:    e8 fc ff ff ff        call   12f8 <vulfoo+0x34>
    12fc:    83 c4 10              add    esp,0x10
    12ff:    b8 00 00 00 00        mov    eax,0x0
    1304:    c9                    leave
    1305:    c3                    ret
```



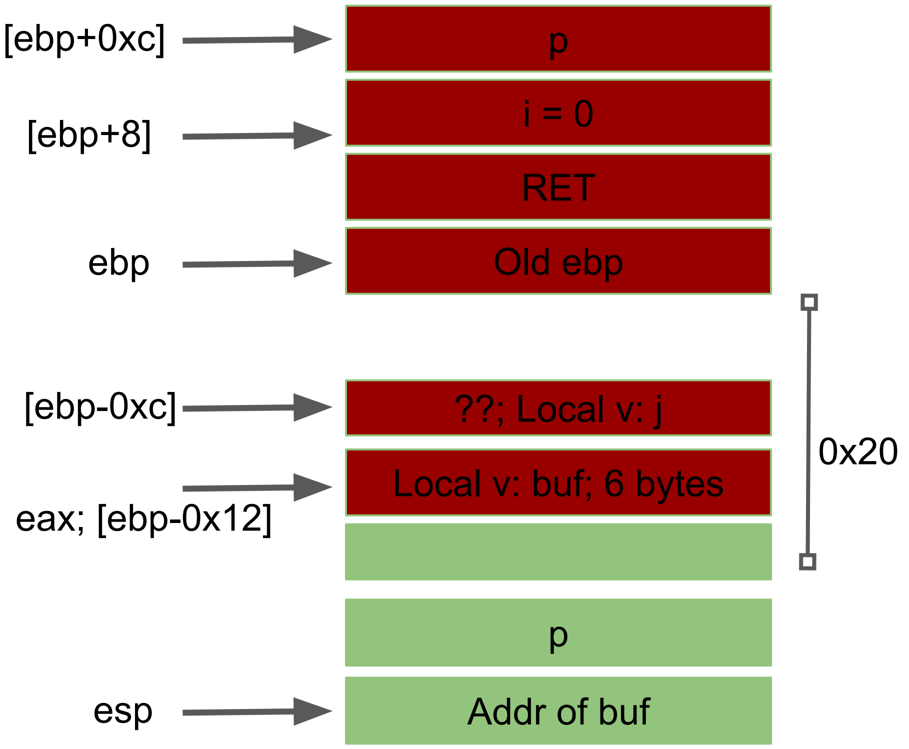
Buffer Overflow Example: code/overflowlocal

```
000012c4 <vulfoo>:
    12c4:   55                  push   ebp
    12c5:   89 e5               mov    ebp,esp
    12c7:   83 ec 18            sub    esp,0x18
    12ca:   8b 45 08            mov    eax,DWORD PTR [ebp+0x8]
    12cd:   89 45 f4            mov    DWORD PTR [ebp-0xc],eax
    12d0:   83 ec 08            sub    esp,0x8
    12d3:   ff 75 0c            push   DWORD PTR [ebp+0xc]
    12d6:   8d 45 ee            lea    eax,[ebp-0x12]
    12d9:   50                  push   eax
    12da:   e8 fc ff ff ff      call   12db <vulfoo+0x17>
    12df:   83 c4 10            add    esp,0x10
    12e2:   83 7d f4 00         cmp    DWORD PTR [ebp-0xc],0x0
    12e6:   74 07               je     12ef <vulfoo+0x2b>
    12e8:   e8 10 ff ff ff      call   11fd <print_flag>
    12ed:   eb 10               jmp    12ff <vulfoo+0x3b>
    12ef:   83 ec 0c            sub    esp,0xc
    12f2:   68 45 20 00 00      push   0x2045
    12f7:   e8 fc ff ff ff      call   12f8 <vulfoo+0x34>
    12fc:   83 c4 10            add    esp,0x10
    12ff:   b8 00 00 00 00      mov    eax,0x0
    1304:   c9                  leave
    1305:   c3                  ret
```



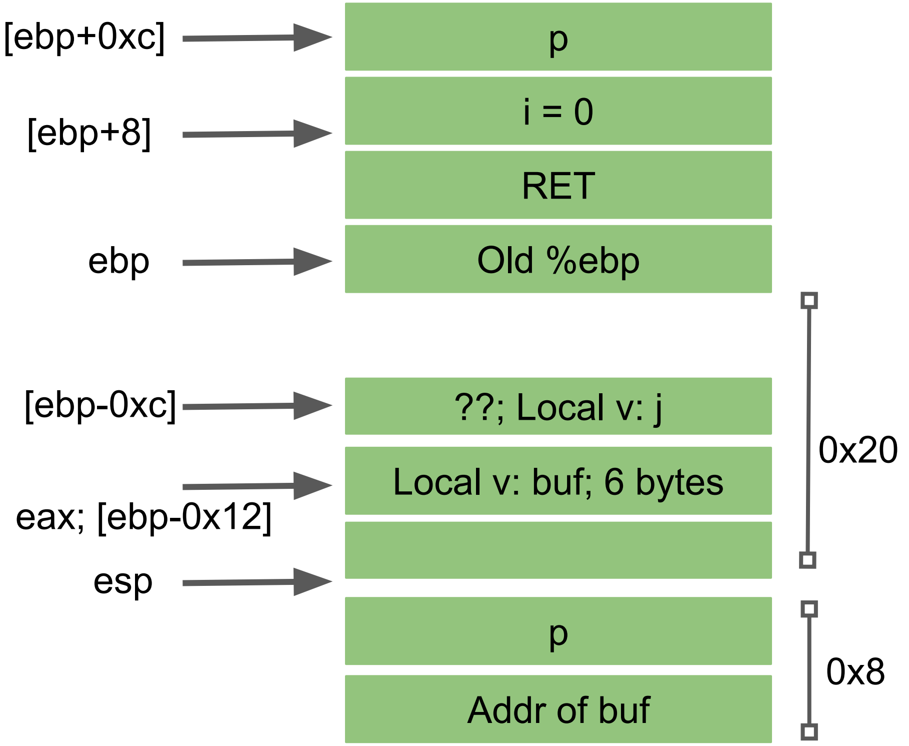
Buffer Overflow Example: code/overflowlocal

```
000012c4 <vulfoo>:
    12c4:      55                   push   ebp
    12c5:      89 e5                mov    ebp,esp
    12c7:      83 ec 18             sub    esp,0x18
    12ca:      8b 45 08             mov    eax,DWORD PTR [ebp+0x8]
    12cd:      89 45 f4             mov    DWORD PTR [ebp-0xc],eax
    12d0:      83 ec 08             sub    esp,0x8
    12d3:      ff 75 0c             push   DWORD PTR [ebp+0xc]
    12d6:      8d 45 ee             lea    eax,[ebp-0x12]
    12d9:      50                   push   eax
    12da:      e8 fc ff ff ff       call   12db <vulfoo+0x17>
    12df:      83 c4 10             add    esp,0x10
    12e2:      83 7d f4 00          cmp    DWORD PTR [ebp-0xc],0x0
    12e6:      74 07                je     12ef <vulfoo+0x2b>
    12e8:      e8 10 ff ff ff       call   11fd <print_flag>
    12ed:      eb 10                jmp    12ff <vulfoo+0x3b>
    12ef:      83 ec 0c             sub    esp,0xc
    12f2:      68 45 20 00 00       push   0x2045
    12f7:      e8 fc ff ff ff       call   12f8 <vulfoo+0x34>
    12fc:      83 c4 10             add    esp,0x10
    12ff:      b8 00 00 00 00       mov    eax,0x0
    1304:      c9                   leave
    1305:      c3                   ret
```



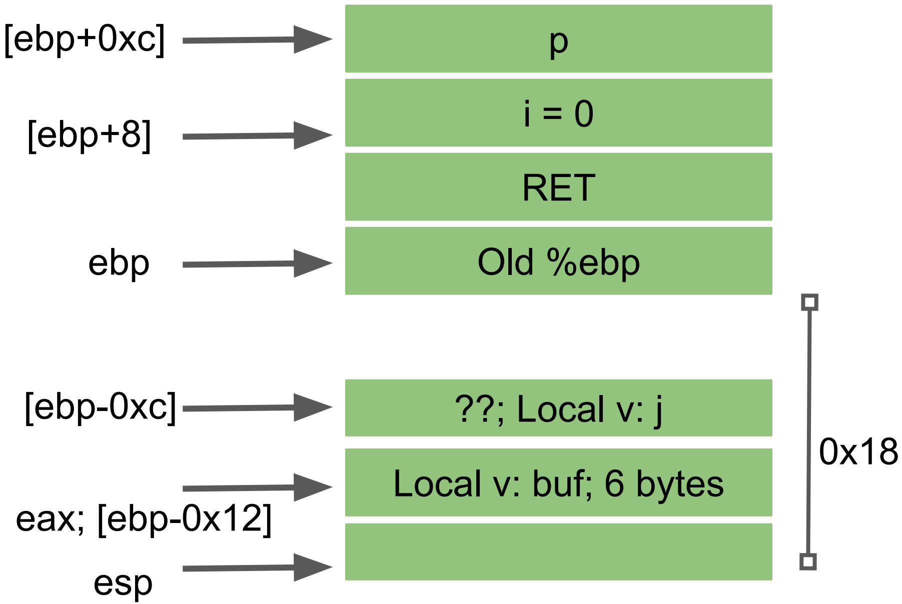
Buffer Overflow Example: code/overflowlocal

```
000012c4 <vulfoo>:
    12c4:       55                      push   ebp
    12c5:       89 e5                   mov    ebp,esp
    12c7:       83 ec 18                sub    esp,0x18
    12ca:       8b 45 08                mov    eax,DWORD PTR [ebp+0x8]
    12cd:       89 45 f4                mov    DWORD PTR [ebp-0xc],eax
    12d0:       83 ec 08                sub    esp,0x8
    12d3:       ff 75 0c                push   DWORD PTR [ebp+0xc]
    12d6:       8d 45 ee                lea    eax,[ebp-0x12]
    12d9:       50                      push   eax
    12da:       e8 fc ff ff ff          call   12db <vulfoo+0x17>
    12df:       83 c4 10                add    esp,0x10
    12e2:       83 7d f4 00             cmp    DWORD PTR [ebp-0xc],0x0
    12e6:       74 07                   je     12ef <vulfoo+0x2b>
    12e8:       e8 10 ff ff ff          call   11fd <print_flag>
    12ed:       eb 10                   jmp    12ff <vulfoo+0x3b>
    12ef:       83 ec 0c                sub    esp,0xc
    12f2:       68 45 20 00 00          push   0x2045
    12f7:       e8 fc ff ff ff          call   12f8 <vulfoo+0x34>
    12fc:       83 c4 10                add    esp,0x10
    12ff:       b8 00 00 00 00          mov    eax,0x0
    1304:       c9                      leave
    1305:       c3                      ret
```



Buffer Overflow Example: code/overflowlocal

```
000012c4 <vulfoo>:
    12c4:   55                  push   ebp
    12c5:   89 e5               mov    ebp,esp
    12c7:   83 ec 18            sub    esp,0x18
    12ca:   8b 45 08            mov    eax,DWORD PTR [ebp+0x8]
    12cd:   89 45 f4            mov    DWORD PTR [ebp-0xc],eax
    12d0:   83 ec 08            sub    esp,0x8
    12d3:   ff 75 0c            push   DWORD PTR [ebp+0xc]
    12d6:   8d 45 ee            lea    eax,[ebp-0x12]
    12d9:   50                  push   eax
    12da:   e8 fc ff ff ff      call   12db <vulfoo+0x17>
    12df:   83 c4 10            add    esp,0x10
    12e2:   83 7d f4 00         cmp    DWORD PTR [ebp-0xc],0x0
    12e6:   74 07               je     12ef <vulfoo+0x2b>
    12e8:   e8 10 ff ff ff      call   11fd <print_flag>
    12ed:   eb 10               jmp    12ff <vulfoo+0x3b>
    12ef:   83 ec 0c            sub    esp,0xc
    12f2:   68 45 20 00 00      push   0x2045
    12f7:   e8 fc ff ff ff      call   12f8 <vulfoo+0x34>
    12fc:   83 c4 10            add    esp,0x10
    12ff:   b8 00 00 00 00      mov    eax,0x0
    1304:   c9                  leave
    1305:   c3                  ret
```



Buffer Overflow Example: code/overflowlocal 64-bit

```
int vulfoo(int i, char* p)
{
  int j = i;
  char buf[6];

  strcpy(buf, p);

  if (j)
    print_flag();
  else
    printf("I pity the fool!\n");

  return 0;
}

int main(int argc, char *argv[])
{
  if (argc == 2)
    vulfoo(0, argv[1]);
}
```

```
000000000000125e <vulfoo>:
    125e:      55                      push   rbp
    125f:      48 89 e5                mov    rbp,rsp
    1262:      48 83 ec 20             sub    rsp,0x20
    1266:      89 7d ec                mov    DWORD PTR [rbp-0x14],edi
    1269:      48 89 75 e0             mov    QWORD PTR [rbp-0x20],rsi
    126d:      8b 45 ec                mov    eax,DWORD PTR [rbp-0x14]
    1270:      89 45 fc                mov    DWORD PTR [rbp-0x4],eax
    1273:      48 8b 55 e0             mov    rdx,QWORD PTR [rbp-0x20]
    1277:      48 8d 45 f6             lea    rax,[rbp-0xa]
    127b:      48 89 d6                mov    rsi,rdx
    127e:      48 89 c7                mov    rdi,rax
    1281:      e8 aa fd ff ff          call   1030 <strcpy@plt>
    1286:      83 7d fc 00             cmp    DWORD PTR [rbp-0x4],0x0
    128a:      74 0c                   je     1298 <vulfoo+0x3a>
    128c:      b8 00 00 00 00          mov    eax,0x0
    1291:      e8 f3 fe ff ff          call   1189 <print_flag>
    1296:      eb 0c                   jmp    12a4 <vulfoo+0x46>
    1298:      48 8d 3d a6 0d 00 00    lea    rdi,[rip+0xda6]        # 2045
<_IO_stdin_used+0x45>
    129f:      e8 9c fd ff ff          call   1040 <puts@plt>
    12a4:      b8 00 00 00 00          mov    eax,0x0
    12a9:      c9                      leave
    12aa:      c3                      ret
```

Exercise: code/overflowlocal2

```
int vulfoo(int i, char* p)
{
  int j = i;
  char buf[6];

  strcpy(buf, p);

  if (j == 0x12345678)
    print_flag();
  else
    printf("I pity the fool!\n");

  return 0;
}

int main(int argc, char *argv[])
{
  vulfoo(argc, argv[1]);
}
```

Shell Command

Run a program and use another program’s output as a parameter

```
./program $(python -c "print '\x12\x34'*5")
```

Shell Command

Compute some data and redirect the output to another program’s stdin

```
python2 -c "print 'A'*18+'\x2d\x62\x55\x56' + 'A'*4 + '\x78\x56\x34\x12'" |
./program
```