CSE 410/510 Special Topics: Software Security

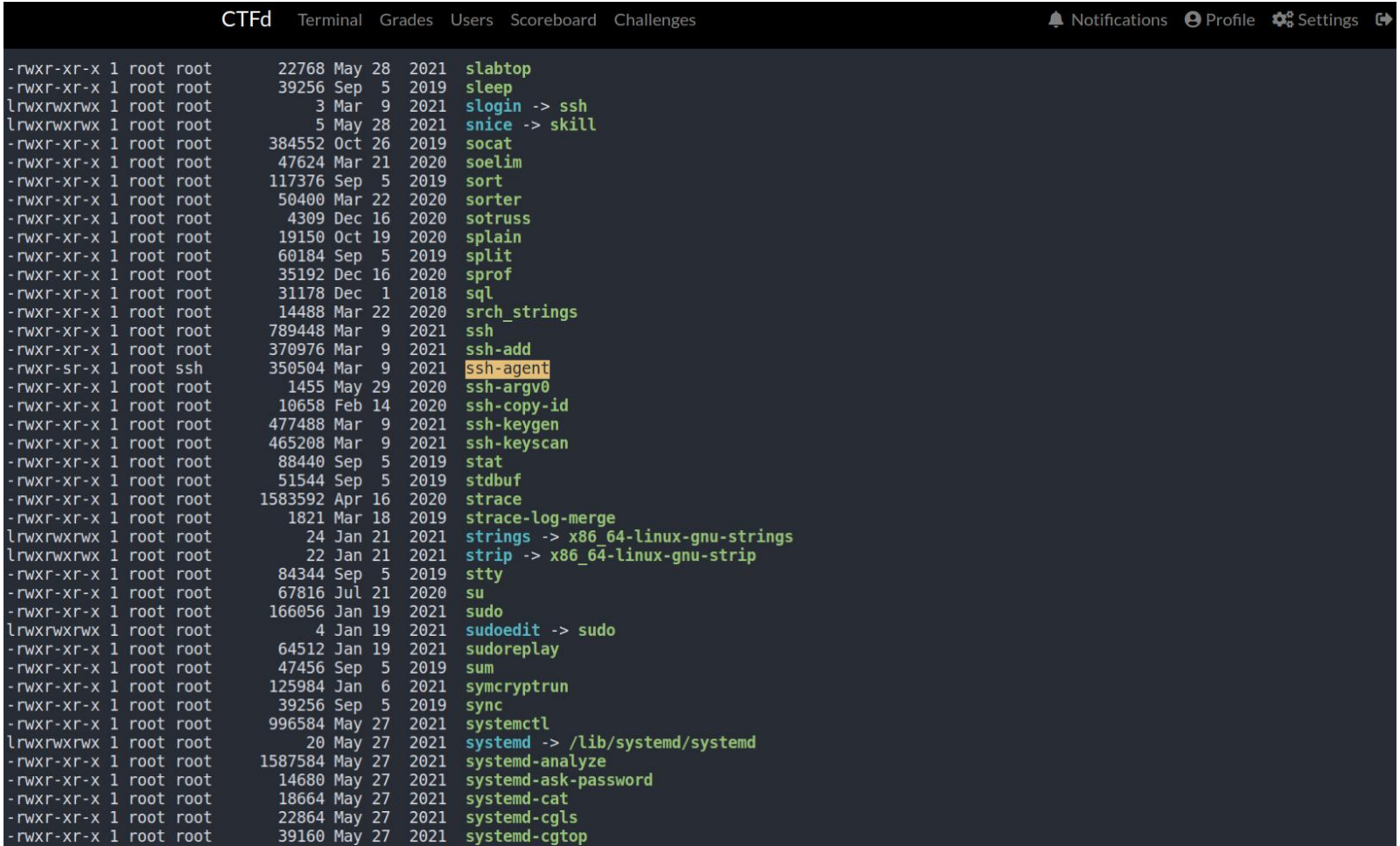
Instructor: Dr. Ziming Zhao

Location: Obrian 109
Time: Monday, Wednesday 5:00PM-6:20PM

This Class

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

SET-UID programs on our server



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 Assembly

On x86/x86-64, most system calls rely on the software interrupt (the **int 0x80** instruction).

A software interrupt is caused either by an exceptional condition in the processor itself, or a special 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)

%eax	Name	Source	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	kernel/exit.c	int	-	-	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t	-	-
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t	-	-
5	sys_open	fs/open.c	const char *	int	int	-	-
6	sys_close	fs/open.c	unsigned int	-	-	-	-
7	sys_waitpid	kernel/exit.c	pid_t	unsigned int *	int	-	-
8	sys_creat	fs/open.c	const char *	int	-	-	-
9	sys_link	fs/namei.c	const char *	const char *	-	-	-
10	sys_unlink	fs/namei.c	const char *	-	-	-	-
11	sys_execve	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
12	sys_chdir	fs/open.c	const char *	-	-	-	-
13	sys_time	kernel/time.c	int *	-	-	-	-
14	sys_mknod	fs/namei.c	const char *	int	dev_t	-	-
15	sys_chmod	fs/open.c	const char *	mode_t	-	-	-
16	sys_lchown	fs/open.c	const char *	uid_t	gid_t	-	-
18	sys_stat	fs/stat.c	char *	struct __old_kernel_stat *	-	-	-
19	sys_lseek	fs/read_write.c	unsigned int	off_t	unsigned int	-	-
20	sys_getpid	kernel/sched.c	-	-	-	-	-
21	sys_mount	fs/super.c	char *	char *	char *	-	-
22	sys_oldumount	fs/super.c	char *	-	-	-	-

https://www.informatik.htw-dresden.de/~beck/ASM/syscall_list.html

Making a System Call in x86 Assembly

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

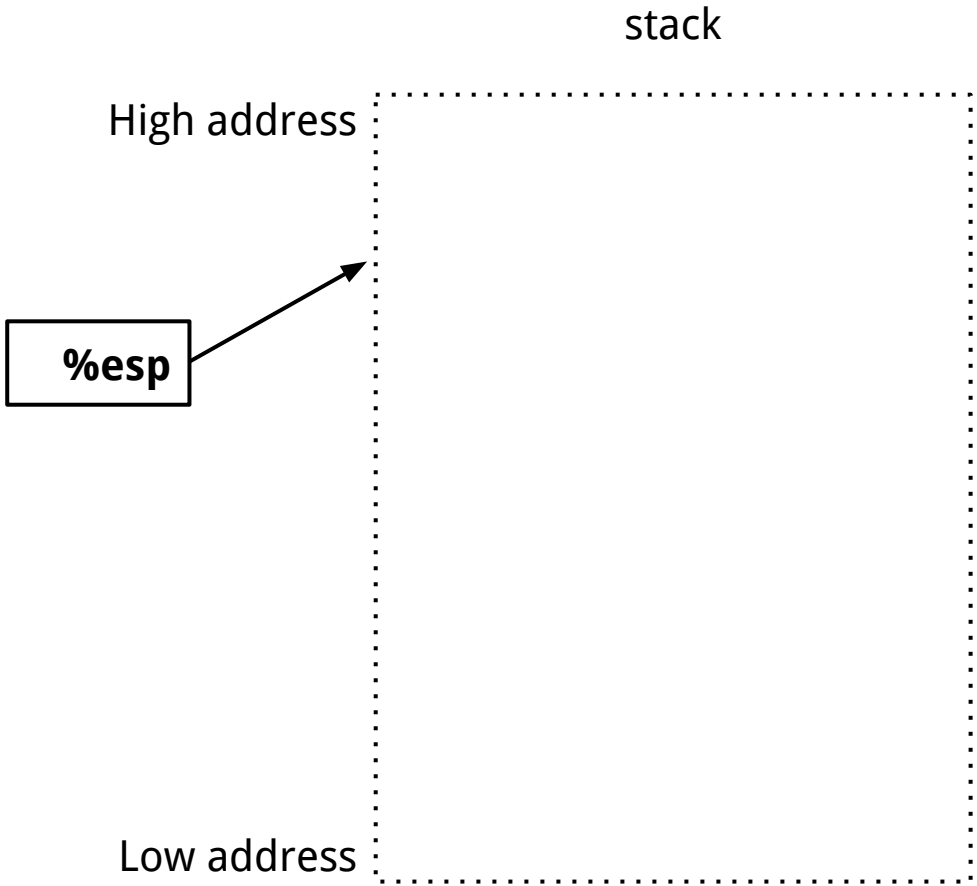
Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of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

Source: www.LookupTables.com

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

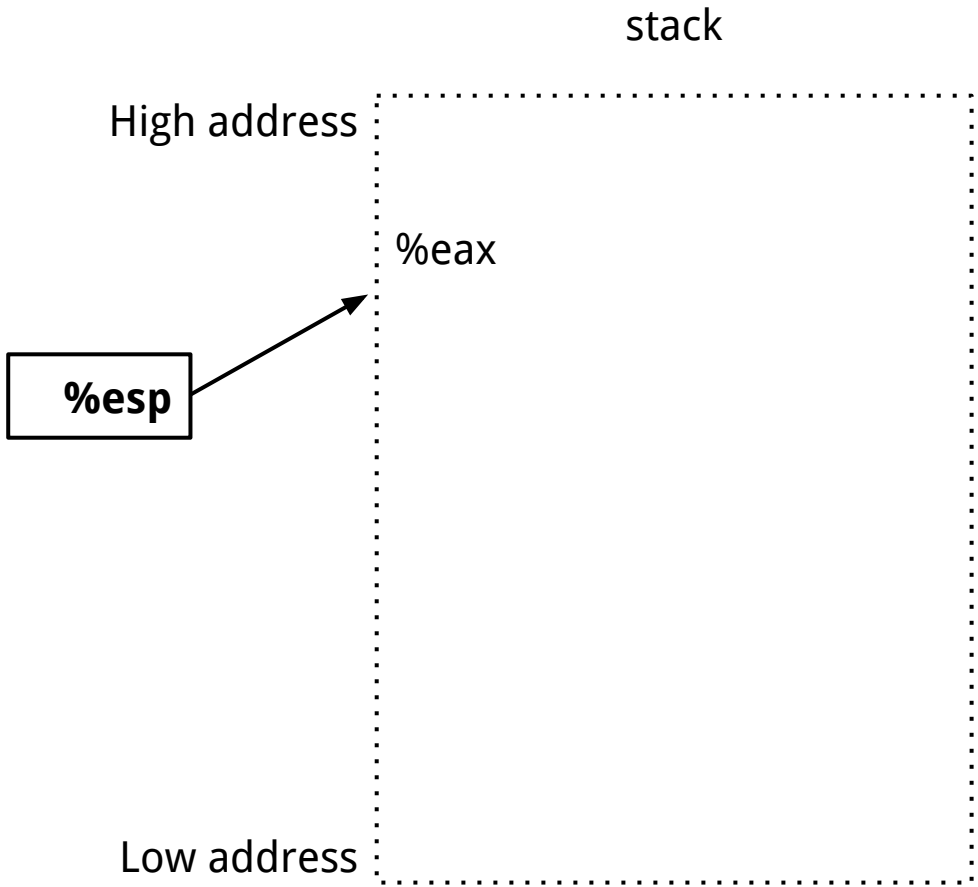
Making a System Call in x86 Assembly

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



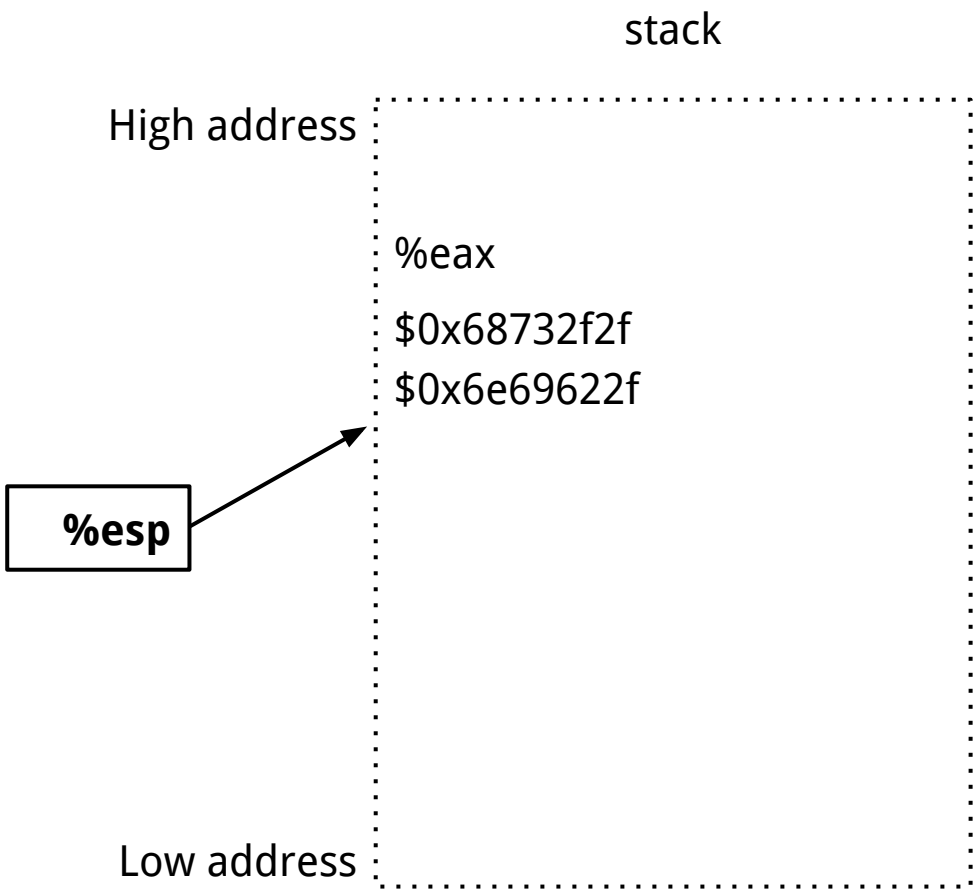
Making a System Call in x86 Assembly

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



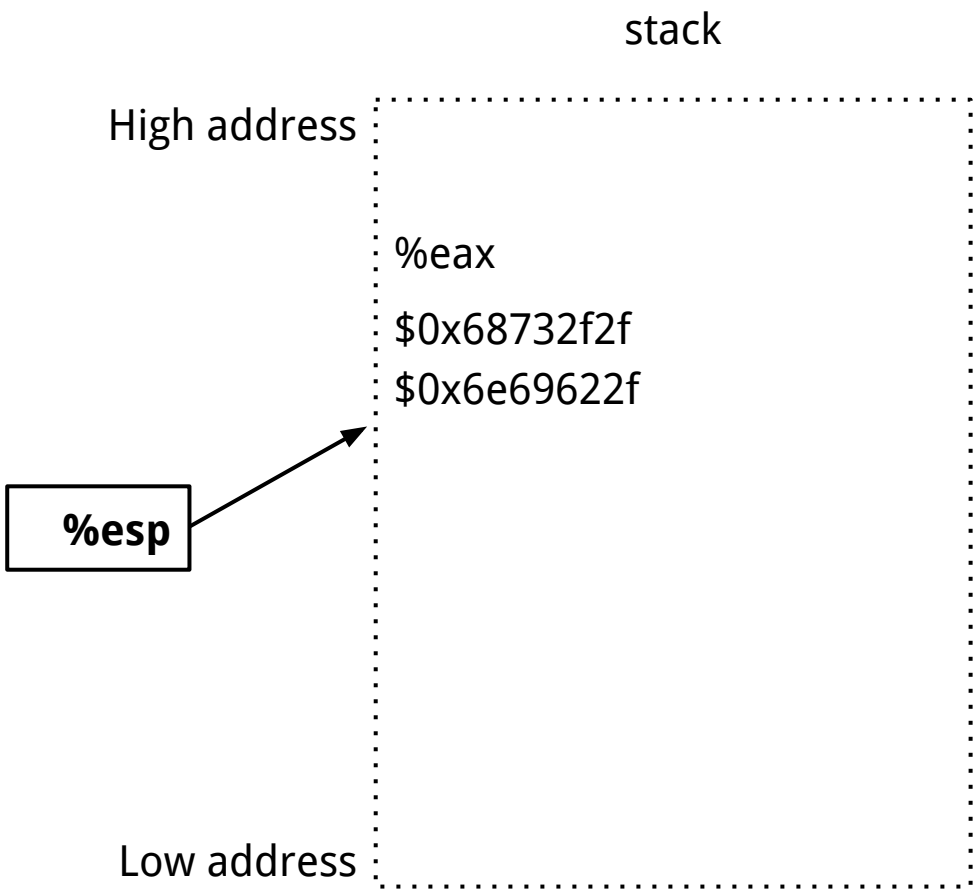
Making a System Call in x86 Assembly

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

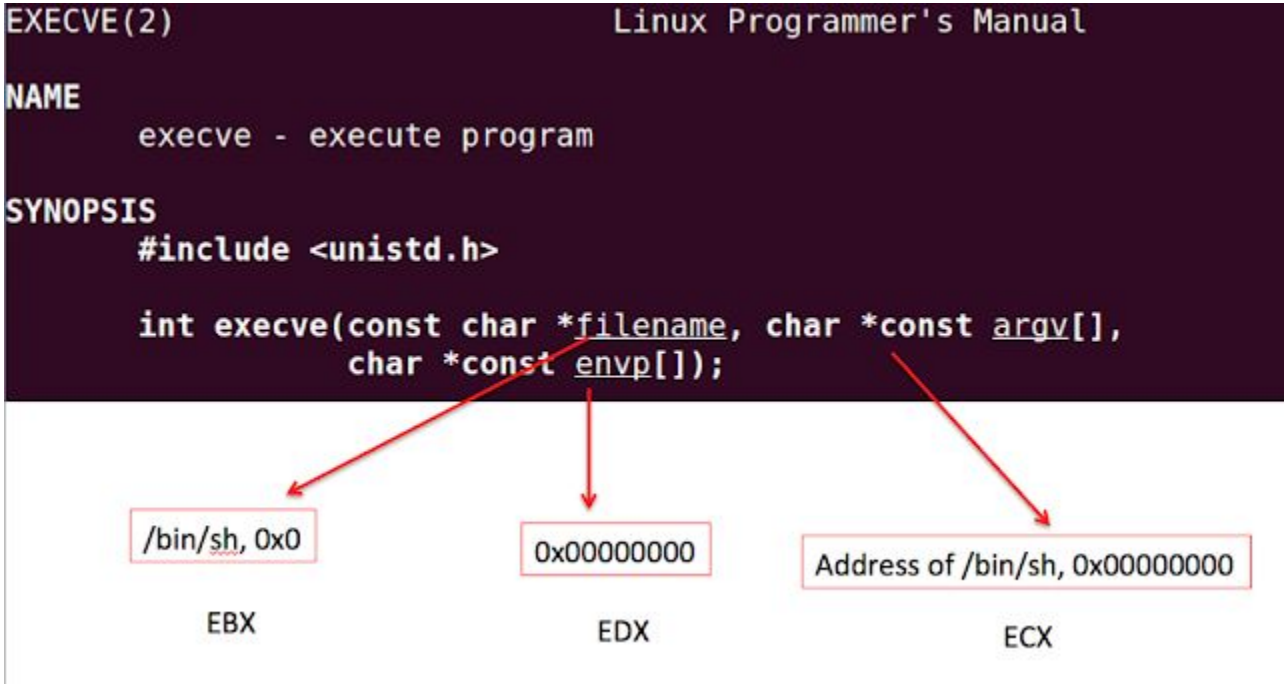


Making a System Call in x86 Assembly

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



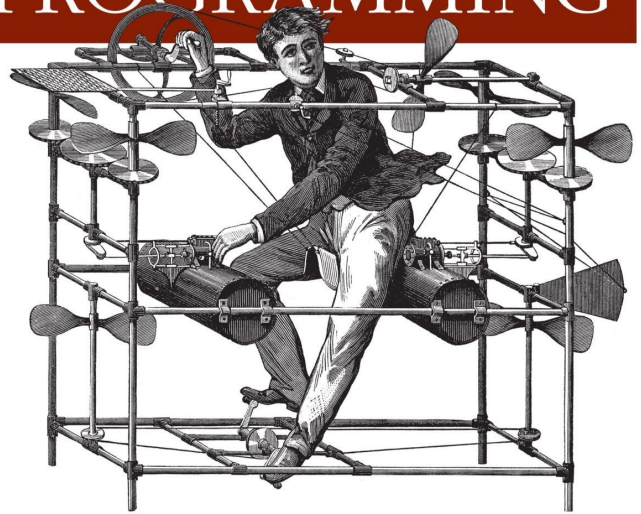
Making a System Call in x86 Assembly



execve(“/bin/sh”, address of string “/bin/sh”, 0)

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.

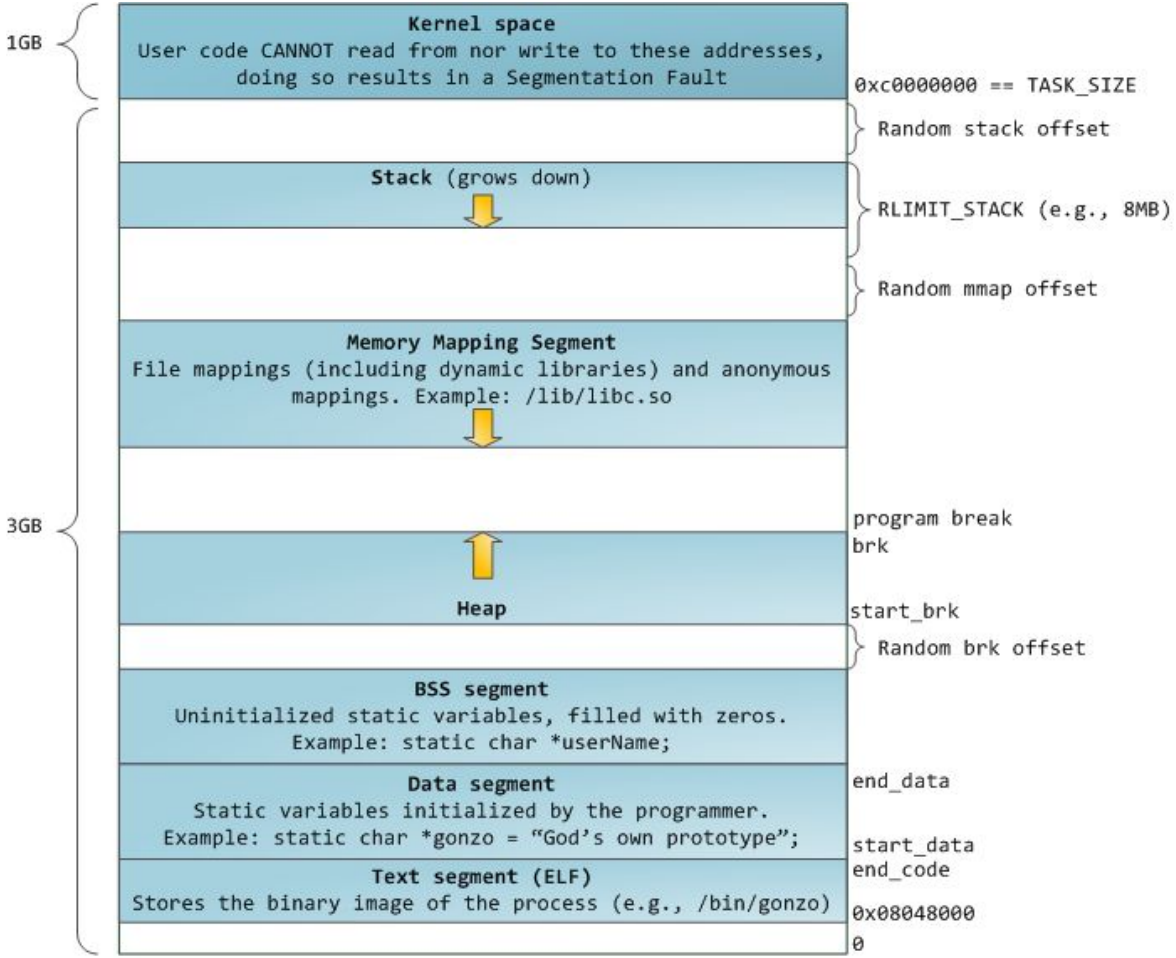
Common Environment Variables

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.

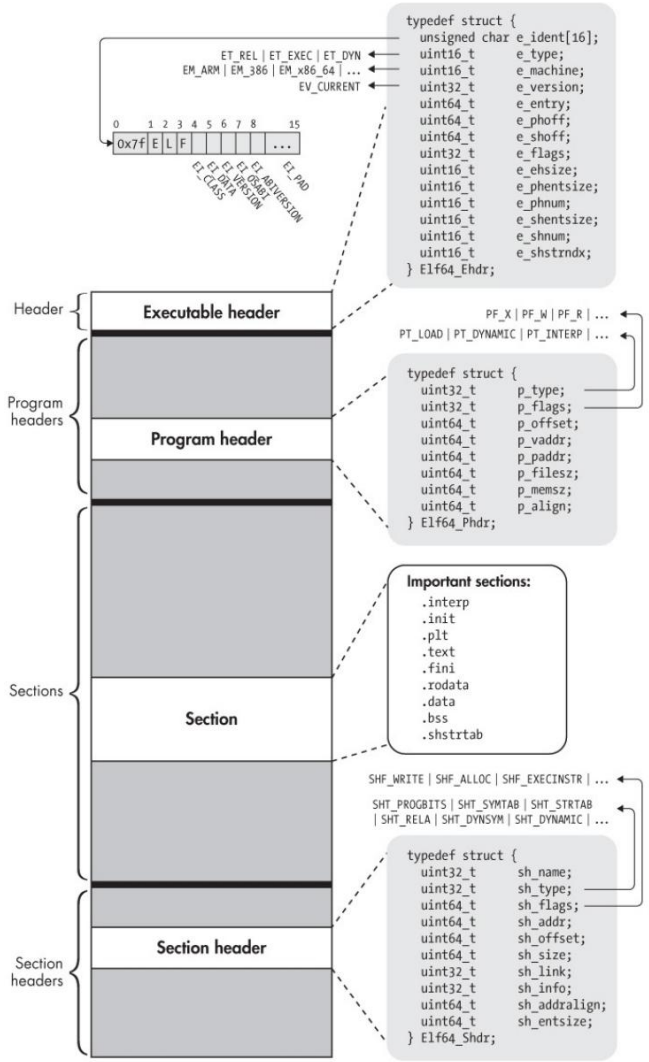


Executable and Linkable Format (ELF)

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).



- 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
```

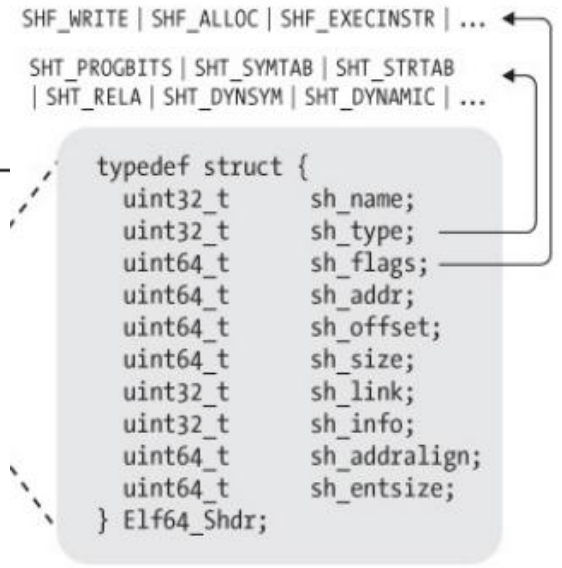
Sections

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.

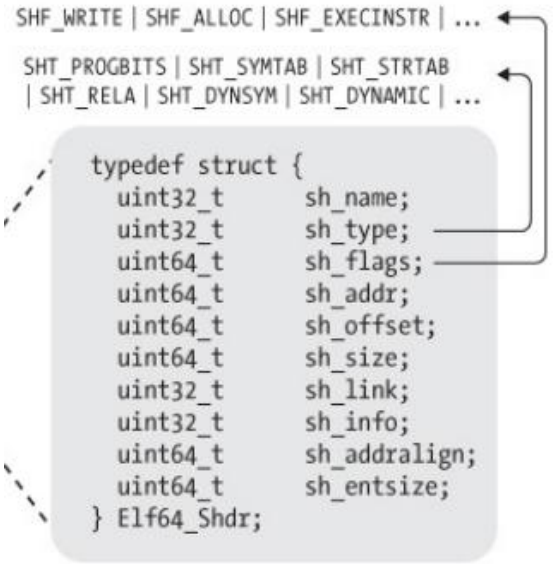
```
readelf -S a.out
```

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.



```
➜ 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.

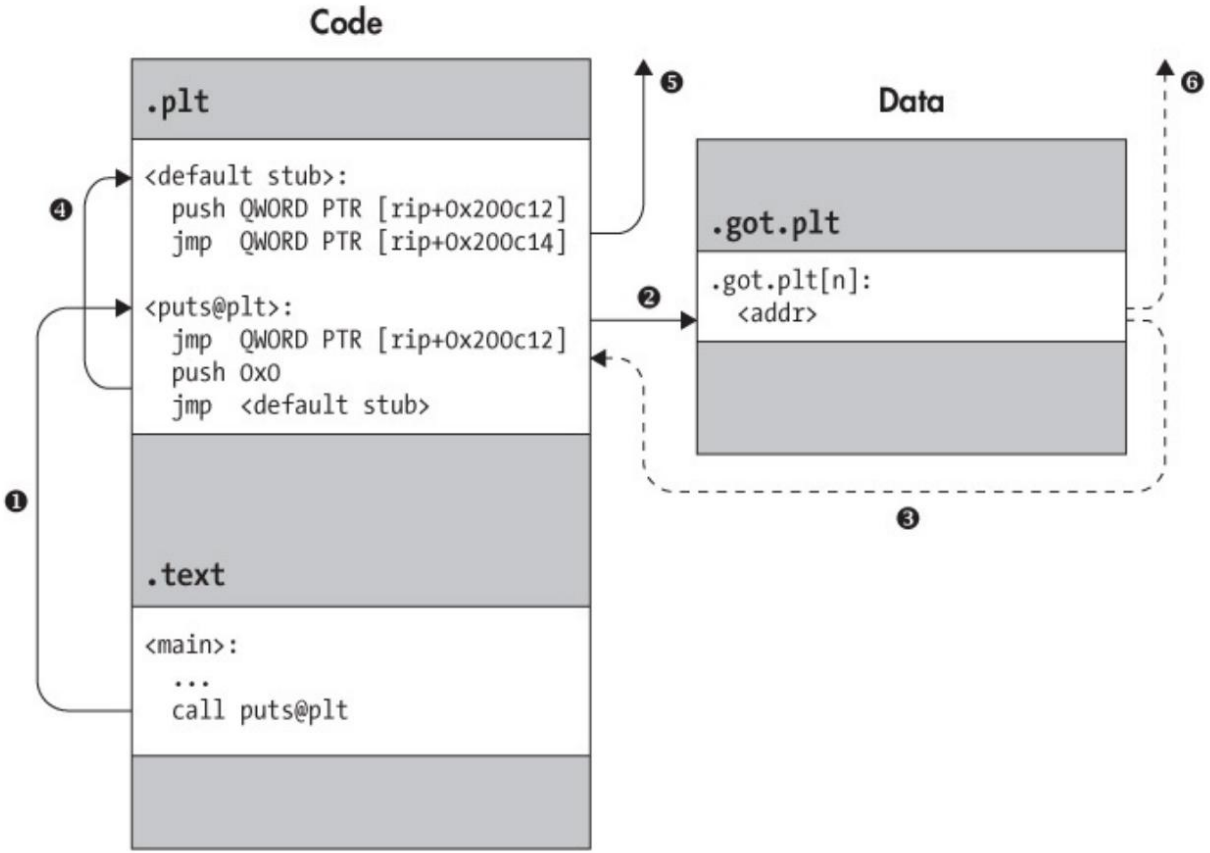
Lazy Binding (.plt, .got, .got.plt Sections)

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).

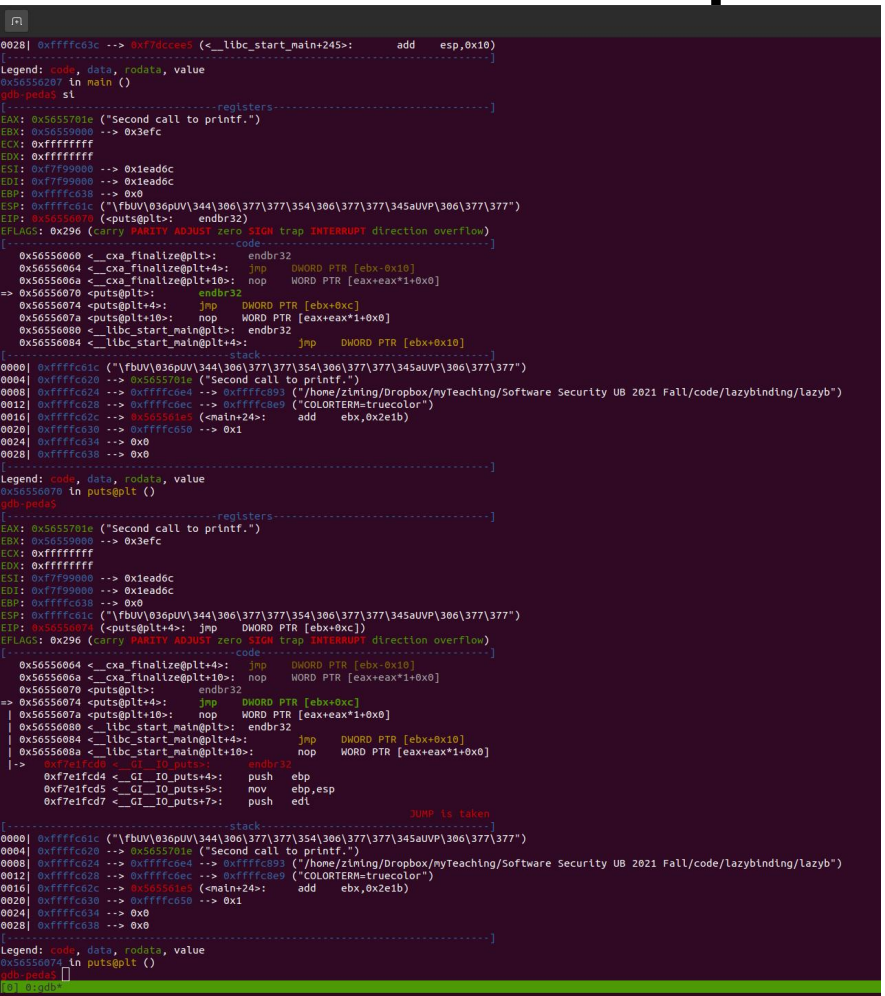
.plt is a code section that contains executable code. The PLT consists entirely of stubs of a well-defined format, dedicated to directing calls from the .text section to the appropriate library location.

.got.plt is a data section.

Dynamically Resolving a Library Function Using the PLT



Example: Debug code\lazyb



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.

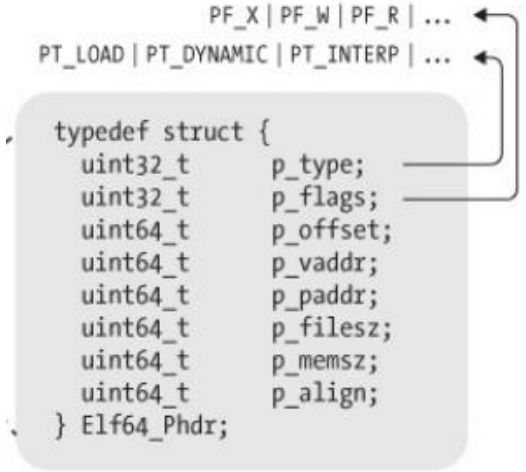
The section view of an ELF binary is meant for static linking purposes.

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
```

```
➜ 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")

Dues

1. Homework-1