Deriving Memory Shape

Timothy Daly

August 9, 2007

Contents

Abstract

We need to understand the shape of data in memory. This will also give us information related to the stride of a program while it walks memory. This pamphlet examines various data structures, their memory layouts, their method of access, and the code laid down by GCC.

1 Overview

2 Deriving memory shape from int80 args

If we look at the int80 system call table for linux we see:

1	exit	kernel/exit.c
	ebx	int
2	fork	arch/i386/kernel/process.c
	ebx	struct pt_regs
3	read	fs/read_write.c
	ebx	unsigned int
	ecx	char *
	edx	size_t
4	write	fs/read_write.c
	ebx	unsigned int
	ecx	const char *
	edx	size_t
5	open	fs/open.c
	ebx	const char *
	ecx	int
	edx	int
6	close	fs/open.c
	ebx	unsigned int
7	waitpid	kernel/exit.c
	ebx	pid_t
	ecx	unsigned int *
	edx	int
8	creat	fs/open.c
	ebx	const char *
	ecx	int
9	link	fs/namei.c
	ebx	const char *
	ecx	const char *
10	unlink	fs/namei.c
	ebx	const char *

11	execve	arch/i386/kernel/process.c
	ebx	struct pt_regs
12	chdir	fs/open.c
	ebx	const char *
13	time	kernel/time.c
	ebx	int *
14	mknod	fs/nameei.c
	ebx	const char *
	ecx	int
	edx	dev_t
15	chmod	fs/open.c
	ebx	const char *
	ecx	mode_t
16	lchown	fs/open.c
	ebx	const char *
	ecx	uid_t
	edx	gid_t
18	stat	fs/stat.c
	ebx	char *
	ecx	struct old_kernel_stat *
19	lseek	fs/read_write.c
	ebx	unsigned int
	ecx	off_t
	edx	unsigned int
20	getpid	kernel/sched.c

21	mount	fs/super.c
	ebx	char *
	ecx	char *
	edx	char *
22	oldumount	fs/super.c
	ebx	char *
23	setuid	kernel/sys.c
	ebx	uid_t
24	getuid	kernel/sched.c
25	stime	kernel/time.c
	ebx	int *
26	ptrace	arch/i386/kernel/ptrace.c
	ebx	long
	ecx	long
	edx	long
	esx	long
27	alarm	kernel/sched.c
	ebx	unsigned int
28	fstat	fs/stat.c
	ebx	unsigned int
	ecx	struct old_kernel_stat *
29	pause	arch/i386/ernel/sys_i386.c
30	utime	fs/open.c
	ebx	char *
	ecx	struct utimbuf *
33	access	fs/open.c
	ebx	const char *
	ecx	int
34	nice	kernel/sched.c
	ebx	int
36	sync	fs/buffer.c
37	kill	kernel/signal.c
	ebx	int
	ecx	int
38	rename	fs/namei.c
	ebx	const char *
	ecx	const char *
39	mkdir	fs/namei.c
	ebx	const char *
	ecx	int

40	rmdir	fs/namei.c
	ebx	const char *
41	dup	fs/fcntl.c
	ebx	unsigned int
42	pipe	arch/i386/kernel/sys_i386.c
	ebx	unsigned long *
43	times	kernel/sys.c
	ebx	struct tms *
45	brk	mm/mmap.c
	ebx	unsigned long
46	setgid	kernel/sys.c
	ebx	get_t
47	getgid	kernel/sched.c
48	signal	kernel/signal.c
	ebx	int
	ecx	sighandler_t
49	geteuid	kernel/sched.c
50	getegid	kernel/sched.c
51	acct	kernel/acct.c
	ebx	const char *
52	umount	fs/super.c
	ebx	char *
	ecx	int
54	ioctl	fs/ioctl.c
	ebx	unsigned int
	ecx	unsigned int
	edx	unsigned long
55	fcntl	fs/fcntl.c
	ebx	unsigned int
	ecx	unsigned int
	edx	unsigned long
57	setpgid	kernel/sys.c
	ebx	pid_t
	ecx	pid_t
59	olduname	arch/i386/kernel/sys_i386.c
	ebx	struct oldold_utsname *

60	umask	ernel/sys.c
	ebx	int
61	chroot	fs/open.c
	ebx	const char *
62	ustat	fs/super.c
	ebx	dev_t
	ecx	struct ustat *
63	dup2	fs/fcntl.c
	ebx	unsigned int
	ecx	unsigned int
64	getppid	kernel/sched.c
65	getpgrp	kernel/sys.c
66	setsid	kernel/sys.c
67	sigaction	arch/i386/kernel/signal.c
	ebx	int
	ecx	const struct old_sigaction *
	edx	struct old_sigaction *
68	sgetmask	kernel/signal.c
69	ssetmask	kernel/signal.c
	ebx	int

70	setreuid	kernel/sys.c
	ebx	uid_t
	ecx	uid_t
71	setregid	kernel/sys.c
	ebx	gid_t
	ecx	gid_t
72	sigsuspend	arch/i386/kernel/signal.c
	ebx	int
	ecx	int
	edx	old_sigset_t
73	sigpending	kernel/signal.c
	ebx	old_sigset_t *
74	sethostname	kernel/sys.c
	ebx	char *
	ecx	int
75	setrlimit	kernel/sys.c
	ebx	unsigned int
	ecx	struct rlimit *
76	getrlimit	kernel/sys.c
	ebx	unsigned int
	ecx	struct rlimit *
77	getrusage	kernel/sys.c
	ebx	int
	ecx	struct rusage *
78	gettimeofday	kernel/time.c
	ebx	struct timeeval *
	ecx	struct timezone *
79	settimeofday	kernel/time.c
	ebx	struct timeeval *
	ecx	struct timezone *

80	getgroups	kernel/sys.c
	ebx	int
	ecx	get_t *
81	setgroups	kernel/sys.c
	ebx	int
	ecx	get_t *
82	old_select	arch/i386/kernel/sys_i386.c
	ebx	struct sel_arg_struct *
83	symlink	fs/namei.c
	ebx	const char *
	ecx	const char *
84	lstat	fs/stat.c
	ebx	char *
	ecx	struct old_kernel_stat *
85	readlink	fs/stat.c
	ebx	const char *
	ecx	char *
	edx	int
86	uselib	fs/exec.c
	ebx	const char *
87	swapon	mm/swapfile.c
	ebx	const char *
	ecx	int
88	reboot	kernel/sys.c
	ebx	int
	ecx	int
	edx	int
	esx	void *
89	old_readdir	fs/readdir.c
	ebx	unsigned int
	ecx	void *
	edx	unsigned int

90	old_mmap	arch/i386/kernel/sys_i386.c
	ebx	struct mmap_arg_struct *
91	munmap	mm/mmap.c
	ebx	unsigned long
	ecx	size_t
92	truncate	fs/open.c
	ebx	const char *
	ecx	unsigned long
93	ftruncate	fs/open.c
	ebx	unsigned long
	ecx	unsigned long
94	fchmod	fs/open.c
	ebx	unsigned int
	ecx	mode_t
95	fchown	fs/open.c
	ebx	unsigned int
	ecx	uid_t
	edx	gid_t
96	getpriority	kernel/sys.c
	ebx	int
	ecx	int
97	setpriority	kernel/sys.c
	ebx	int
	ecx	int
	edx	int
99	statfs	fs/open.c
	ebx	const char *
	ecx	struct statfs *

100	fstatfs	fs/open.c
	ebx	unsigned int
	ecx	struct statfs *
101	ioperm	arch/i386/kernel/ioport.c
	ebx	unsigned long
	ecx	unsigned long
	edx	int
102	socketcall	net/socket.c
	ebx	int
	ecx	unsigned long *
103	syslog	kernel/printk.c
	ebx	int
	ecx	char *
	edx	int
104	setitimer	kernel/itimer.c
	ebx	int
	ecx	struct itimerval *
	edx	struct itimerval *
105	getitimer	kernek/itimer.c
	ebx	int
	ecx	struct itimerval *
106	newstat	fs/stat.c
	ebx	char *
	ecx	struct stat *
107	newlstat	fs/stat.c
	ebx	char *
	ecx	struct stat *
108	newfstat	fs/stat.c
	ebx	unsigned int
	ecx	struct stat *
109	uname	arch/i386/kernel/sys_i386.c
	ebx	struct old_utsname *

110	iopl	arch/i386/kernel/ioport.c
	ebx	unsigned long
111	vhangup	fs/open.c
112	idle	arch/i386/kernel/process.c
113	vm86old	arch/i386/kernel/vm86.c
	ebx	unsigned long
	ecx	struct vm86plus_struct *
114	wait4	kernel/exit.c
	ebx	pid_t
	ecx	unsigned long *
	edx	int options
	esx	struct rusage *
115	swapoff	mm/swapfile.c
	ebx	const char *
116	sysinfo	kernel/info.c
	ebx	struct sysinfo *
117	ipc(*NOTE)	arch/i386/kernel/sys_i386.c
	ebx	uint
	ecx	int
	edx	int
	esx	int
	edi	void *
		long
118	fsync	fs/buffer.c
	ebx	unsigned int
119	sigreturn	arch/i386/kernel/signal.c
	ebx	unsigned long

120	clone	arch/i386/kernel/process.c
	ebx	struct pt_regs
121	setdomainname	kernel/sys.c
	ebx	char *
	ecx	int
122	newuname	kernel/sys.c
	ebx	struct new_utsname *
123	modify_ldt	arch/i386/kernel/ldt.c
	ebx	int
	ecx	void *
	edx	unsigned long
124	adjtimex	kernel/time.c
	ebx	struct timex *
125	mprotect	mm/mprotect.c
	ebx	unsigned long
	ecx	size_t
	edx	unsigned long
126	sigprocmask	kernel/signal.c
	ebx	int
	ecx	old_sigset_t *
	edx	old_sigset_t *
127	create_module	kernel/module.c
	ebx	const char *
	ecx	size_t
128	init_module	kernel/module.c
	ebx	const char *
	ecx	struct module *
129	delete_module	kernel/module.c
	ebx	const char *

130	get_kernel_syms	kernel/module.c
	ebx	struct kernel_sym *
131	quotactl	fs/dquot.c
	ebx	int
	ecx	const char *
	edx	int
	esx	caddr_t
132	getpgid	kernel/sys.c
	ebx	pid_t
133	fchdir	fs/open.c
	ebx	unsigned in
134	bdflush	fs/buffer.c
	ebx	int
	ecx	long
135	sysfs	fs/super.c
	ebx	int
	ecx	unsigned long
	edx	unsigned long
136	personality	kernel/exec_domain.c
	ebx	unsigned long
138	setfsuid	kernel/sys.c
	ebx	uid_t
139	setfsgid	kernel/sys.c
	ebx	gid_t

140	llseek	fs/read_write.c
	ebx	unsigned int
	ecx	unsigned long
	edx	unsigned long
	esx	loff_t *
	edi	unsigned int
141	getdents	fs/readdir.c
	ebx	unsigned int
	ecx	void *
	edx	unsigned int
142	select	fs/select.c
	ebx	int
	ecx	fd_set *
	edx	fd_set *
	esx	fs_set *
	edi	sttruct timeval *
143	flock	fs/locks.c
	ebx	unsigned int
	ecx	unsigned int
144	msync	mm/filemap.c
	ebx	unsigned long
	ecx	size_t
	edx	int
145	readv	fs/read_write.c
	ebx	unsigned long
	ecx	const struct iovec *
	edx	unsigned long
146	writev	fs/read_write.c
	ebx	unsigned long
	ecx	const struct iovec *
	edx	unsigned long
147	getsid	kernel/sys.c
	ebx	pid_t
148	fdatasync	fs/buffer.c
	ebx	unsigned int
149	sysctl	kernel/sysctl.c
	ebx	struct sysctl_args *

150	mlock ebx ecx	mm/mlock.c unsigned long size_t
151	munlock ebx ecx	mm/mlock.c unsigned long size_t
152	mlockall ebx	mm/mlock.c int
153	munlockall	mm/mlock.c
154	sched_setparam ebx ecx	kernel/sched.c pid_t struct sched_param *
155	sched_getparam ebx ecx	kernel/sched.c pid_t struct sched_param *
156	sched_setscheduler ebx ecx edx	kernel/sched.c pid_t int struct sched_param *
157	sched_getscheduler ebx	kernel/sched.c pid_t
158	sched_yield	kernel/sched.c
159	sched_get_priority_max ebx	kernel/sched.c int

160	sched_get_priority_min	kernel/sched.c
	ebx	int
161	sched_rr_get_interval	kernel/sched.c
	ebx	pid_t
	ecx	struct timespec *
162	nanosleep	kernel/sched.c
	ebx	struct timespec *
	ecx	struct timespec *
163	mremap	mm/mremap.c
	ebx	unsigned long
	ecx	unsigned long
	edx	unsigned long
	esx	unsigned long
164	setresuid	kernel/sys.c
	ebx	uid_t
	ecx	uid_t
	edx	uid_t
165	getresuid	kernel/sys.c
	ebx	uid_t *
	ecx	uid_t *
	edx	uid_t *
166	vm86	arch/i386/kernel/vm86.c
	ebx	struct vm86_struct *
167	query_module	kernel/module.c
	ebx	const char *
	ecx	int
	edx	char *
	esx	size_t
	edi	size_t *
168	poll	fs/select.c
	ebx	struct pollfd *
	ecx	unsgned int
	edx	long
169	nfsservctl	fs/filesystems.c
	ebx	int
	ecx	void *
	edx	void *

170	setresgid	kernel/sys.c
	ebx	gid_t
	ecx	gid_t
	edx	gid_t
171	getresgid	kernel/sys.c
	ebx	gid_t *
	ecx	gid_t *
	edx	gid_t *
172	prctl	kernel/sys.c
	ebx	int
	ecx	unsigned long
	edx	unsigned long
	esx	unsigned long
	edi	unsigned long
173	rt_sigreturn	arch/i386/kernel/signal.c
	ebx	unsigned long
174	rt_sigaction	kernel/signal.c
	ebx	int
	ecx	const struct sigaction *
	edx	struct sigaction *
	esx	size_t
175	rt_sigprocmask	kernel/signal.c
	ebx	int
	ecx	sigset_t *
	edx	sigset_t *
	esx	size_t
176	rt_sigpending	kernel/signal.c
	ebx	sigset_t *
	ecx	size_t
177	rt_sigtimedwait	kernel/signal.c
	ebx	const sigset_t *
	ecx	siginfo_t *
	edx	const struct timespec *
	esx	size_t
178	rt_sigqueueinfo	kernel/signal.c
	ebx	int
	ecx	int
	edx	siginfo_t *
179	rt_sigsuspend	arch/i386/kernel/signal.c
	ebx	sigset_t *
	ecx	size_t

180	pread	fs/read_write.c
	ebx	unsigned int
	ecx	char *
	edx	size_t
	esx	loff_t
181	pwrite	fs/read_write.c
	ebx	unsigned int
	ecx	const char *
	edx	size_t
	esx	loff_t
182	chown	fs/open.c
	ebx	const char *
	ecx	uid_t
	edx	gid_t
183	getcwd	fs/dcache.c
	ebx	char *
	ecx	unsigned long
184	capget	kernel/capability.c
	ebx	cap_user_header_t
	ecx	cap_user_data_t
185	capset	kernel/capability.c
	ebx	cap_user_header_t
	ecx	const cap_user_data_t
186	sigaltstack	arch/i386/kernel/signal.c
	ebx	const stack_t *
	ecx	stack_t *
187	sendfile	mm/filemap.c
	ebx	int
	ecx	int
	edx	off_t *
	esx	size_t
190	vfork	arch/i386/kernel/process.c
	ebx	struct pt_regs

Note: for sys_ipc(117) this syscall takes six arguments, so it can't fit into the five registers (ebx-edi); the last parameter is of type long. This syscall requires a special call method where a pointer is put in ebx which points to an array containing the six arguments.

For the numbers of the syscalls, look in `arch/i386/kernel/entry.S` for the `sys_call_table`. The syscall numbers are offsets into that table. Several spots in the table are occupied by the syscall `sys_ni_syscall`. This is a placeholder that either replaces an obsolete syscall or reserves a spot for future syscalls.

The system calls are calld from the function `system_call` in the same file; in particular, they are called with the assembly instruction call

```
call *SYMBOL_NAME(sys_call_table)(,%eax,4)
```

The `SYMBOL_NAME(sys_call_table)` gets replaced by a symbol name in `sys_call_table`.

The `SYMBOL_NAME` is a macro defined in `include/linux/linkage.h` and it just replaces itself with its argument.

2.1 Typedefs

atomic_t include/asm/atomic.h:
```
#ifdef__SMP__
typedef struct { volatile int counter; } atomic_t
#else
typedef struct { int counter; } atomic_t;
#endif
```

---

caddr_t include/asm/posix_types.h:
```
typedef char* __kernel_caddr_t;
```
include/linux/types.h:
```
typedef __kernel_caddr_t caddr_t;
```

---

cap_user_header_t include/linux/capability.h:
```
typedef struct __user_cap_header_struct {
   __u32 version;
   int pid;
} *cap_user_data_t;
```

---

cap_user_data_t include/linux/capability.h:
```
typedef struct __user_cap_data_struct {
   __u32 effective;
   __u32 permitted;
   __u32 inheritable;
} *cap_user_data_t;
```

---

clock_t include/asm/posix_types.h:
```
typedef long __kernel_clock_t;
```
include/linux/types.h
```
typedef __kernel_clock_t clock_t;
```

---

dev_t include/asm/posix_types.h:
```
typedef unsigned short __kernel_dev_t;
```
include/linux/types.h:
```
typedef __kernel_dev_t dev_t;
```

---

fdset include/linux/posix_types.h:
```
#define __FD_SETSIZE 1024
#define __NFDBITS (8 * sizeof(unsigned long))
#define __FDSET_LONGS (__FD_SETSIZE/__NFDBITS)
```
(==> __FDSET_LONGS=32)

gid_t include/asm/posix_types.h
typedef unsigned short __kernel_gid_t;
include/linux/types.h
typedef __kernel_gid_t gid_t;

__kernel_daddr_t include/asm/posix_types.h:
typedef int __kernel_daddr_t;

__kernel_fsid_t include/asm/posix_types.h:
typedef struct {
int __val[2];
} __kernel_fsid_t;

__kernel_ino_t include/asm/posix_types.h:
typedef unsigned long __kernel_ino_t;

__kernel_size_t include/asm/posix_types.h:
typedef unsigned int __kernel_size_t;

loff_t include/asm/posix_types.h:
typedef long long __kernel_loff_t;
include/linux/types.h:
typedef __kernel_loff_t loff_t;

mode_t include/asm/posix_types.h:
typedef unsigned short __kernel_mode_t;
include/linux/types.h:
typedef __kernel_mode_t mode_t;

off_t include/asm/posix_types.h:
typedef long __kernel_off_t;
include/linux/types.h:
typedef __kernel_off_t off_t;

old_sigset_t include/asm/signal.h:
typedef unsigned long old_sigset_t;

pid_t include/asm/posix_types.h:
typedef int __kernel_pid_t;
include/linux/types.h:
typedef __kernel_pid_t pid_t;

__sighandler_t	include/asm/signal.h: typedef void (* __sighandler_t)(int);

siginfo_t include/asm/siginfo.h:

```
#define SI_MAX_SIZE 128
#define SI_PAD_SIZE ((SI_MAX_SIZE/sizeof(int))-3)
(==> SI_PAD_SIZE == 29)

typedef struct siginfo {
  int si_signo;
  int si_errno;
  int si_code;

  union {
    int _pad[SI_PAD_SIZE];

    /* kill() */
    struct {
      pid_t _pid; /* sender's pid */
      uid_t _uid; /* sender's uid */
    } _kill;

    /* POSIX.2b timers */
    struct {
      unsigned int _timer1;
      unsigned int _timer2;
    } _timer;

    /* POSIX.1b signals */
    struct {
      pid_t _pid; /* sender's pid */
      uid_t _uid; /* sender's uid */
      sigval_t _sigval;
    } _rt;

    /* SIGCHLD */
    struct {
      pid_t _pid; /* which child */
      uid_t _uid; /* sender's uid */
      int _status; /* exit code */
      clock_t _utime;
      clock_t _stime;
    } _sigchld;

    /* SIGILL, SIGFPE, SIGSEGV, SIGBUS */
    struct {
      void *_addr; /* faulting insn/memory ref */
    } _sigfault;

    /* SIGPOLL */
    struct {
      int _band; /* POLL_IN, POLL_OUT, POLL_MSG */
      int _fd;
    } _sigpoll;
  } _sifields;
}siginfo_t;
```

sigset_t include/asm/signal.h:
typedef unsigned long sigset_t;

size_t include/asm/posix_types.h:
typedef unsigned int __kernel_size_t;
include/linux/types.h:
typedef __kernel_size_t size_t;

ssize_t include/asm/posix_types.h:
typedef int __kernel_ssize_t;
include/linux/types.h:
typedef __kernel_ssize_t ssize_t;

stack_t include/asm/signal.h:
typedef struct sigaltstack {
void *ss_sp;
int ss_flags;
size_t ss_size;
} stack_t;

suseconds_t include/asm/posix_types.h:
typedef long __kernel_suseconds_t;
include/linux/types.h:
typedef __kernel-suseconds_t suseconds_t;

time_t include/asm/posix_types.h:
typedef long __kernel_time_t;
include/linux/types.h:
typedef __kernel_time_t time_t;

bf uid_t include/asm/posix_types.h:
typedef unsigned short __kernel_uid_t;
include/linux/types.h:
typedef __kernel_uid_t uid_t;

uint include/linux/types.h:
typedef unsigned int uint;

__u32 include/asm/types.h:
typedef unsigned int __u32;

3 C Calling Conventions

One of the "big picture" issues in looking at compiled C code is the function calling conventions. These are the methods that a calling function and a called function agree on how parameters and return values should be passed between them, and how the stack is used by the function itself. The layout of the stack constitutes the "stack frame", and knowing how this works can go a long way to decoding how something works.

There are two conventions in general use, the "__cdecl" convention usually used on unix and the "__stdcall" convention usually used on windows.

3.1 The __cdecl convention

This convention is the most common because it supports semantics required by the C language. The C language supports variadic functions (variable argument lists, such as in printf calls). This means that the *caller* must clean up the stack after the function call: the called function has no way to know how to do this. It's not terribly optimal, but the C language semantics demand it.

3.2 The __stdcall convention

Also known as "__pascal", this requires that each function take a fixed number of parameters, and this means that the *called* function can do argument cleanup in one place rahter than have this be scattered throughout the program in every place that calls it. The Win32 API primarily uses __stdcall.

3.3 Register use in the stack frame

In both __cdecl and __stdcall conventions, the same set of three registers is involved in a function call frame.

- ESP - Stack Pointer

  This 32-bit register is implicitly manipulated by several CPU instructions (PUSH, POP, CALL, RET among others). It always points to the last element used on the stack, not the first free element. This means that the PUSH and POP operations would be specified in pseudo-C as:

  ```
  *--ESP = value;  //push
  value = *ESP++   //pop
  ```

  The "Top of the Stack" is an occupied location, not a free one, and is at the lowest memory address.

- EBP - Base Pointer

  This 32 bit register is used to reference all the function parameters and local variable in the current stack frame. Unlike the ESP register, the base

pointer is manipulated only explicitly. This is also known as the frame pointer.

- EIP - Instruction Pointer

  This holds the address of the next CPU instruction to be executed, and it's saved onto the stack as part of the CALL instruction. Any of the "jump" instructions modify the EIP directly.

3.4 Calling a __cdecl function

The best way to understand the stack organization is to see each step in calling a function with the __cdecl conventions. These steps are taken automatically by the compiler, and though not all of them are used in every case (sometimes no parameters, sometimes no local variables, sometimes no saved registers), this shows the overall mechanism employed.

1. Push parameters onto the stack, from right to left

   Parameters are pushed onto the stack, one at a time, from right to left. Whenther the parameters are evaluated from right to left is a different matter, and in the case where this is unspecified by the language, the code should never rely on this. The calling code must keep track of how many bytes of parameters have been pushed onto the stack so it can clean it up later.

   ```
   function parameter #n
   ...
   function parameter #2
   function parameter #1
   ```

2. Call the function

   Here, the processor pushes contents of the EIP (instruction pointer) onto the stack, and it points to the first byte after the CALL instruction. After this finishes, the caller has lost control, and the callee is in charge. This step does not change the EBP register

```
function parameter #n
...
function parameter #2
function parameter #1
EIP of caller (the instruction to return to)
```

3. Save and update the EBP

   Now that we're in the new function, we need a new local stack frame pointed to by EBP, so this is done by saving the current EBP, which belongs to the previous funtions's frame, and making it point to the top of the stack:

```
push ebp       ; save the caller's frame pointer
move ebp, esp  ; frame pointer is stack bottom
```

   Once EBP has been changed, it can now refer directly to the function's arguments using the new frame pointer (EBP register) as a base.

```
function parameter #n
...
function parameter #2
function parameter #1
EIP of caller (the instruction to return to)
EBP of caller (caller's frame pointer)
```

4. Square up the stack

   The stack expects to be aligned on a 8 byte boundary. On the pentium and subsequent x86 processors there is a substantial penalty if double-precision variables are not stored 8-byte aligned. However if the caller used a far jump call it also saved the 16-bit CS segment register on the stack. To protect against this problem we make sure the lower bits of the stack pointer are aligned by making them all zeros. This is done by:

```
sub esp,0x8
and esp,0xfffffff0
```

5. Allocate local variables

   This function may choose to use local stack based variables, and they are allocated here simply by decrementing the stack pointer by the amount of space required. This is always done in four-byte chunks.

   The local variables are located on the stack between the EBP and ESP registers, and though it would be possible to refer to them as offsets from either one, by convention the EBP register is used. That means that [EBP-4] refers to the first local variable.

```
function parameter #n
...
function parameter #2   [EBP+12]
function parameter #1   [EBP+8]
EIP of caller (the instruction to return to)
EBP of caller (caller's frame pointer)
local variable #1       [EBP-4]
local variable #2       [EBP-8]
...
local variable #n       [EBP-4*n]
```

6. Save CPU registers used for temporaries If this function will use any CPU register, it has to save the old vaues first lest it walk on data used by the calling functions. Each register to be used is pushed onto the stack one at a time, and the compiler must remember what it did so it can unwind it later.

```
function parameter #n
...
function parameter #2   [EBP+12]
function parameter #1   [EBP+8]
EIP of caller (the instruction to return to)
EBP of caller (caller's frame pointer)
local variable #1       [EBP-4]
local variable #2       [EBP-8]
...
local variable #n       [EBP-4*n]
saved register
saved register
...
saved register          [ESP] stack top of subroutine
```

7. Perform the function's purpose At this point, the stack frame is set up correctly, and this is represented by the diagram to the right. All the parameters and locals are offsets from the EBP register.

   The function is free to use any of the registers that have been saved onto the stack upon entry, but must not change the stack pointer.

8. Release local storage

   When the function allocates local, temporary space, it does so by subtracting from the stack by space needed, and this process must be reversed to reclaim it. It's usually done by adding to the stack pointer the same amount which was subtracted previously, though a series of POP instructions could achieve the same thing. The addition method means we don't need a spare register as the target of the POPs.

9. Restore saved registers

   For each register saved onto the stack upon entry, it must be restored from the stack in reverse order.

10. Restore the old base pointer

    The first thing this function did upon entry was save the caller's EBP base pointer, and by restoring it now (popping the top item off the stack), we effectively discard the entire local stack frame and put the caller's frame back in play. This is done by the LEAVE instruction

    ```
    LEAVE
    ```

11. Return from the function

    This is the last step of the called function and the RET instruction pops the old EIP from the stack, which causes a jump to that location. This gives control back to the calling function. Only the stack pointer and instruction pointers are modified by a subroutine return.

12. Clean up pushed parameters

    In the _cdecl convention, the caller must clean up the paramters pushed onto the stack, and this is done either by popping the stack into don't care registers (for a few parameters) or by adding the parameter block size to the stack pointer directly.

4 Initial tests

4.1 nothing.c

```
int main() {
  return(0);
}
```

```
00000000 <main>:
   0: 55                    push   ebp      ; save frame
   1: 89 e5                 mov    ebp,esp  ; set frame
   3: 83 ec 08              sub    esp,0x8  ; align stack
   6: 83 e4 f0              and    esp,0xfffffff0
   9: b8 00 00 00 00        mov    eax,0x0  ; alloc no vars
   e: 29 c4                 sub    esp,eax

return(0);
  10: b8 00 00 00 00        mov    eax,0x0  ; set returncode
  15: c9                    leave
  16: c3                    ret
```

4.2 nonzeroret.c

```
int main() {
 return(3);
}
```

```
00000000 <main>:
   0: 55                    push   ebp
   1: 89 e5                 mov    ebp,esp
   3: 83 ec 08              sub    esp,0x8
   6: 83 e4 f0              and    esp,0xfffffff0
   9: b8 00 00 00 00        mov    eax,0x0
   e: 29 c4                 sub    esp,eax

return(3);
  10: b8 03 00 00 00        mov    eax,0x3
  15: c9                    leave
  16: c3                    ret
```

4.3 integer.c

Allocate an integer and initialize it. Note that the integer i does not get allocated explicitly but appears to use one of the two available stack words.

```
int main() {
  int i=3;
  return(0);
}
```

```
00000000 <main>:
   0: 55                    push   ebp
   1: 89 e5                 mov    ebp,esp
   3: 83 ec 08              sub    esp,0x8
   6: 83 e4 f0              and    esp,0xfffffff0
   9: b8 00 00 00 00        mov    eax,0x0
   e: 29 c4                 sub    esp,eax

int i=3;
  10: c7 45 fc 03 00 00 00  mov    DWORD PTR [ebp-4],0x3

return(0);
  17: b8 00 00 00 00        mov    eax,0x0
  1c: c9                    leave
  1d: c3                    ret
```

4.4 return3.c

```
int main() {
```

```
  int i=3;
  return(i);
}

00000000 <main>:
   0: 55                    push   ebp
   1: 89 e5                 mov    ebp,esp
   3: 83 ec 08              sub    esp,0x8
   6: 83 e4 f0              and    esp,0xfffffff0
   9: b8 00 00 00 00        mov    eax,0x0
   e: 29 c4                 sub    esp,eax

int i=3;
  10: c7 45 fc 03 00 00 00  mov    DWORD PTR [ebp-4],0x3

return(i);
  17: 8b 45 fc              mov    eax,DWORD PTR [ebp-4]
  1a: c9                    leave
  1b: c3                    ret
```

Unfortunately the local variable name, i has disappeared.

Notice the pattern:

```
mov stackoffset, thing       ; initialize the variable
mov register, stackoffset    ; access the variable
```

5 array handling

From the examples below we investigate some behaviors of array handling.

- array1.c – array initialization
- array2.c – array assignment
- array3.c – array assignment out of bounds
- array4.c – array access out of bounds
- array5.c – two array initialization

The arrays seem to be stack allocated.

We can see that the compiler generates code without doing bounds checking. GCC will happily let you access any location available off the stack segment.

If we had the idea of a real stack rather than the stack simulation that GCC is using we can flag the errors in array3.c and array4.c.

5.1 array1.c

Next we test array initialization.

```
int main() {
  int a[] = {1,2,3};
  return(0);
}
```

The array is initialized on the stack.

Note that the

Note that array offsets are calculated so the array is effectively pushed onto the stack in order. That is, the stack looks like:

```
   [ebp-24]   ; a[0]
   [ebp-20]   ; a[1]
   [ebp-16]   ; a[2]

00000000 <main>:
   0: 55                    push   ebp
   1: 89 e5                 mov    ebp,esp
   3: 83 ec 18              sub    esp,0x18
   6: 83 e4 f0              and    esp,0xfffffff0
   9: b8 00 00 00 00        mov    eax,0x0
   e: 29 c4                 sub    esp,eax

int a[] = {1,2,3};
  10: c7 45 e8 01 00 00 00  mov    DWORD PTR [ebp-24],0x1
  17: c7 45 ec 02 00 00 00  mov    DWORD PTR [ebp-20],0x2
  1e: c7 45 f0 03 00 00 00  mov    DWORD PTR [ebp-16],0x3

return(0);
  25: b8 00 00 00 00        mov    eax,0x0
  2a: c9                    leave
  2b: c3                    ret
```

5.2 array2.c

Now we look at assignment. As expected the operations are all stack oriented.

```
int main() {
  int a[] = {1,2,3};
  a[2] = a[0];
  return(0);
}

00000000 <main>:
   0: 55                    push   ebp
```

```
   1: 89 e5                 mov    ebp,esp
   3: 83 ec 18              sub    esp,0x18
   6: 83 e4 f0              and    esp,0xfffffff0
   9: b8 00 00 00 00        mov    eax,0x0
   e: 29 c4                 sub    esp,eax

int a[] = {1,2,3};
  10: c7 45 e8 01 00 00 00  mov    DWORD PTR [ebp-24],0x1
  17: c7 45 ec 02 00 00 00  mov    DWORD PTR [ebp-20],0x2
  1e: c7 45 f0 03 00 00 00  mov    DWORD PTR [ebp-16],0x3

a[2] = a[0];
  25: 8b 45 e8              mov    eax,DWORD PTR [ebp-24]
  28: 89 45 f0              mov    DWORD PTR [ebp-16],eax

return(0);
  2b: b8 00 00 00 00        mov    eax,0x0
  30: c9                    leave
  31: c3                    ret
```

5.3 array3.c

Here we try to assign data outside the stack area. Note that the compiler quite mistakenly allows assignments outside of the array bounds.

```
int main() {
  int a[] = {1,2,3};
  a[16] = a[0];
  return(0);
}
```

```
00000000 <main>:
   0: 55                    push   ebp
   1: 89 e5                 mov    ebp,esp
   3: 83 ec 18              sub    esp,0x18
   6: 83 e4 f0              and    esp,0xfffffff0
   9: b8 00 00 00 00        mov    eax,0x0
   e: 29 c4                 sub    esp,eax

int a[] = {1,2,3};
  10: c7 45 e8 01 00 00 00  mov    DWORD PTR [ebp-24],0x1
  17: c7 45 ec 02 00 00 00  mov    DWORD PTR [ebp-20],0x2
  1e: c7 45 f0 03 00 00 00  mov    DWORD PTR [ebp-16],0x3

a[16] = a[0];
  25: 8b 45 e8              mov    eax,DWORD PTR [ebp-24]
  28: 89 45 28              mov    DWORD PTR [ebp+40],eax
```

```
return(0);
  2b: b8 00 00 00 00        mov    eax,0x0
  30: c9                    leave
  31: c3                    ret
```

5.4 array4.o

Here we access an element outside the array and assign it to the array location.

```
int main() {
  int a[] = {1,2,3};
  a[0] = a[16];
  return(0);
}

00000000 <main>:
   0: 55                    push   ebp
   1: 89 e5                 mov    ebp,esp
   3: 83 ec 18              sub    esp,0x18
   6: 83 e4 f0              and    esp,0xfffffff0
   9: b8 00 00 00 00        mov    eax,0x0
   e: 29 c4                 sub    esp,eax

int a[] = {1,2,3};
  10: c7 45 e8 01 00 00 00  mov    DWORD PTR [ebp-24],0x1
  17: c7 45 ec 02 00 00 00  mov    DWORD PTR [ebp-20],0x2
  1e: c7 45 f0 03 00 00 00  mov    DWORD PTR [ebp-16],0x3

a[0] = a[16];
  25: 8b 45 28              mov    eax,DWORD PTR [ebp+40]
  28: 89 45 e8              mov    DWORD PTR [ebp-24],eax

return(0);
  2b: b8 00 00 00 00        mov    eax,0x0
  30: c9                    leave
  31: c3                    ret
```

5.5 array5.c

Here we initialize two arrays. From this example we can see that the two arrays are stack allocated next to each other. There is no way to distinguish this example from a single array of length 6.

```
int main() {
  int a[] = {1,2,3};
  int b[] = {4,5,6};
```

```
  return(0);
}

00000000 <main>:
   0: 55                    push   ebp
   1: 89 e5                 mov    ebp,esp
   3: 83 ec 28              sub    esp,0x28
   6: 83 e4 f0              and    esp,0xfffffff0
   9: b8 00 00 00 00        mov    eax,0x0
   e: 29 c4                 sub    esp,eax

int a[] = {1,2,3};
  10: c7 45 e8 01 00 00 00  mov    DWORD PTR [ebp-24],0x1
  17: c7 45 ec 02 00 00 00  mov    DWORD PTR [ebp-20],0x2
  1e: c7 45 f0 03 00 00 00  mov    DWORD PTR [ebp-16],0x3

int b[] = {4,5,6};
  25: c7 45 d8 04 00 00 00  mov    DWORD PTR [ebp-40],0x4
  2c: c7 45 dc 05 00 00 00  mov    DWORD PTR [ebp-36],0x5
  33: c7 45 e0 06 00 00 00  mov    DWORD PTR [ebp-32],0x6

return(0);
  3a: b8 00 00 00 00        mov    eax,0x0
  3f: c9                    leave
  40: c3                    ret
```

6 Argument passing

6.1 arg1.c

Notice that declaring the arguments has no effect on the code. The argument values are copied from the previous frame.

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

00000000 <main>:
   0: 55                    push   ebp
   1: 89 e5                 mov    ebp,esp
   3: 83 ec 08              sub    esp,0x8
   6: 83 e4 f0              and    esp,0xfffffff0
   9: b8 00 00 00 00        mov    eax,0x0
   e: 29 c4                 sub    esp,eax
return(argc);
  10: 8b 45 08              mov    eax,DWORD PTR [ebp+8]
```

```
  13: c9                    leave
  14: c3                    ret
```

7 Linked Lists

7.1 linkedlist.c

```
typedef struct llnode {
  int data;
  struct llnode *next;
} list;

int main(int argc, char *argv[]) {
  list *head = (void *)0;
}
```

```
00000000 <main>:
   0: 55                    push   ebp
   1: 89 e5                 mov    ebp,esp
   3: 83 ec 08              sub    esp,0x8
   6: 83 e4 f0              and    esp,0xfffffff0
   9: b8 00 00 00 00        mov    eax,0x0
   e: 29 c4                 sub    esp,eax
list *head = (void *)0;
  10: c7 45 fc 00 00 00 00  mov    DWORD PTR [ebp-4],0x0
  17: c9                    leave
  18: c3                    ret
```

7.2 linkedlist1.c

Here we just look at the code for allocating a new node so we can understand how the malloc subroutine call is handled. We can derive some information about the size of the structure from the size argument to malloc.

```
typedef struct llnode {    /* a linked list node contains */
  int data;                /* a place to point to data    */
  struct llnode *next;     /* and a pointer to  next node */
} list;

int main(int argc, char *argv[]) {
  list *head = (void *)0;  /* head points to nothing    */
  list *tmp;               /* a place to put a new node */
  tmp = (list *)malloc(sizeof(*tmp));  /* allocate node */
}
```

```
00000000 <main>:
   0: 55                    push   ebp
   1: 89 e5                 mov    ebp,esp
   3: 83 ec 08              sub    esp,0x8
   6: 83 e4 f0              and    esp,0xfffffff0
   9: b8 00 00 00 00        mov    eax,0x0
   e: 29 c4                 sub    esp,eax
list *head = (void *)0;
  10: c7 45 fc 00 00 00 00  mov    DWORD PTR [ebp-4],0x0
tmp = (list *)malloc(sizeof(*tmp));
;;; allocate 3 words
  17: 83 ec 0c              sub    esp,0xc
;;; the struct is 8 bytes long, push that value
  1a: 6a 08                 push   0x8
;;; the stack looks like:
;;;     0x08             <===== size of the new memory block
;;;     0xyy
;;;     0xyy
;;;     0xyy
  1c: e8 fc ff ff ff        call   1d <main+0x1d>
;;; clean up the 4 words on the stack
  21: 83 c4 10              add    esp,0x10
;;; set the tmp variable
  24: 89 45 f8              mov    DWORD PTR [ebp-8],eax
  27: c9                    leave
  28: c3                    ret
```

In this case, since we are calling an external routine, we get a second entry in the symbol table:

```
00000000 T main
         U malloc
```

7.3

Here we continue building the linked list. We set the data into the new node, point to the previous list head, and make this node the new head node. This is a stack order linked list.

```
typedef struct llnode {  /* a linked list node contains */
  int data;              /* a place to point to data    */
  struct llnode *next;   /* a pointer to the next node  */
} list;

int main(int argc, char *argv[]) {
  list *head = (void *)0;  /* head points to nothing     */
  list *tmp;               /* a place to put a new node  */
```

```
  tmp = (list *)malloc(sizeof(*tmp));  /* allocate node  */
  tmp -> data = 3;         /* set the data memory        */
  tmp -> next = head;      /* point to the existing head */
  head = tmp;              /* make the new node be head  */
}


00000000 <main>:
   0: 55                    push   ebp
   1: 89 e5                 mov    ebp,esp
   3: 83 ec 08              sub    esp,0x8
   6: 83 e4 f0              and    esp,0xfffffff0
   9: b8 00 00 00 00        mov    eax,0x0
   e: 29 c4                 sub    esp,eax

list *head = (void *)0;
  10: c7 45 fc 00 00 00 00  mov    DWORD PTR [ebp-4],0x0

tmp = (list *)malloc(sizeof(*tmp));
  17: 83 ec 0c              sub    esp,0xc
  1a: 6a 08                 push   0x8
  1c: e8 fc ff ff ff        call   1d <main+0x1d>
  21: 83 c4 10              add    esp,0x10
  24: 89 45 f8              mov    DWORD PTR [ebp-8],eax

tmp -> data = 3;
  27: 8b 45 f8              mov    eax,DWORD PTR [ebp-8]
  2a: c7 00 03 00 00 00     mov    DWORD PTR [eax],0x3

tmp -> next = head;
;;; get tmp pointer
  30: 8b 55 f8              mov    edx,DWORD PTR [ebp-8]
;;; get head pointer
  33: 8b 45 fc              mov    eax,DWORD PTR [ebp-4]
;;; store tmp+4 = head
  36: 89 42 04              mov    DWORD PTR [edx+4],eax

head = tmp;
  39: 8b 45 f8              mov    eax,DWORD PTR [ebp-8]
  3c: 89 45 fc              mov    DWORD PTR [ebp-4],eax
  3f: c9                    leave
  40: c3                    ret
```

7.4 linkedlist3.c

Here we write a loop to free the elements of the linked list.

```
typedef struct llnode {  /* a linked list node contains */
  int data;              /* a place to point to data    */
  struct llnode *next;   /* a pointer to the next node  */
} list;

int main(int argc, char *argv[]) {
  list *head = (void *)0;     /* head points to nothing    */
  list *tmp;                  /* a place to put a new node */
  tmp = (list *)malloc(sizeof(*tmp));  /* allocate node    */
  tmp -> data = 3;           /* set the data memory        */
  tmp -> next = head;        /* point to the existing head */
  head = tmp;                /* make the new node be head  */

  while (0 != head) {        /* free the whole list         */
    free(head->data);        /* recover data memory         */
    tmp = head->next;        /* remember the next node      */
    free(head);              /* recover the node memory     */
    head=tmp;                /* set up to process next node */
  }
}
```

;;; note that in the following

```
;;;   head = DWORD PTR [ebp-4]
;;;   tmp  = DWORD PTR [ebp-8]

00000000 <main>:
   0: 55                push   ebp
   1: 89 e5             mov    ebp,esp
   3: 83 ec 08          sub    esp,0x8
   6: 83 e4 f0          and    esp,0xfffffff0
   9: b8 00 00 00 00    mov    eax,0x0
   e: 29 c4             sub    esp,eax

list *head = (void *)0;  /* head points to nothing */
  10: c7 45 fc 00 00 00 00  mov    DWORD PTR [ebp-4],0x0 (head)

tmp = (list *)malloc(sizeof(*tmp));  /* allocate node */
  17: 83 ec 0c          sub    esp,0xc
  1a: 6a 08             push   0x8
  1c: e8 fc ff ff ff    call   1d <main+0x1d>
  21: 83 c4 10          add    esp,0x10
  24: 89 45 f8          mov    DWORD PTR [ebp-8],eax (tmp)

tmp -> data = 3;         /* set the data memory */
  27: 8b 45 f8          mov    eax,DWORD PTR [ebp-8] (tmp)
```

```
  2a: c7 00 03 00 00 00 mov    DWORD PTR [eax],0x3   (tmp->data)

tmp -> next = head;      /* point to the existing head */
  30: 8b 55 f8          mov    edx,DWORD PTR [ebp-8] (tmp)
  33: 8b 45 fc          mov    eax,DWORD PTR [ebp-4] (head)
  36: 89 42 04          mov    DWORD PTR [edx+4],eax (tmp->next)

head = tmp;              /* and make the new node be head */
  39: 8b 45 f8          mov    eax,DWORD PTR [ebp-8] (tmp)
  3c: 89 45 fc          mov    DWORD PTR [ebp-4],eax (head)

while (0 != head) {      /* free the whole list */
  3f: 83 7d fc 00       cmp    DWORD PTR [ebp-4],0x0 (head)
  43: 75 02             jne    47 <main+0x47>        (loop body)
  45: eb 2f             jmp    76 <main+0x76>        (loop exit)

free(head->data);        /* recover data memory */
  47: 83 ec 0c          sub    esp,0xc
  4a: 8b 45 fc          mov    eax,DWORD PTR [ebp-4] (head)
  4d: ff 30             push   DWORD PTR [eax]       (head->data)
  4f: e8 fc ff ff ff    call   50 <main+0x50>        (free)
  54: 83 c4 10          add    esp,0x10

tmp = head->next;        /* remember the next node */
  57: 8b 45 fc          mov    eax,DWORD PTR [ebp-4] (head)
  5a: 8b 40 04          mov    eax,DWORD PTR [eax+4] (head->next)
  5d: 89 45 f8          mov    DWORD PTR [ebp-8],eax (tmp)

free(head);              /* recover the node memory */
  60: 83 ec 0c          sub    esp,0xc
  63: ff 75 fc          push   DWORD PTR [ebp-4]     (head)
  66: e8 fc ff ff ff    call   67 <main+0x67>        (free)
  6b: 83 c4 10          add    esp,0x10

head=tmp;                /* set up to process next node */
  6e: 8b 45 f8          mov    eax,DWORD PTR [ebp-8] (tmp)
  71: 89 45 fc          mov    DWORD PTR [ebp-4],eax (head)
  74: eb c9             jmp    3f <main+0x3f>        (loop top)
  76: c9                leave
  77: c3                ret
```

Now the symbol table contains a reference to the free routine.

```
         U free
00000000 T main
         U malloc
```

7.5 linkedlist4.c

Here we walk the list summing up the value of a single node.

```
typedef struct llnode {     /* a linked list node contains */
  int data;                 /* a place to point to data    */
  struct llnode *next;      /* a pointer to the next node  */
} list;

int main(int argc, char *argv[]) {
  list *head = (void *)0;   /* head points to nothing     */
  list *tmp;                /* a place to put a new node  */
  int count = 0;            /* a counter                  */
  tmp = (list *)malloc(sizeof(*tmp));  /* allocate node   */
  tmp -> data = 3;          /* set the data memory        */
  tmp -> next = head;       /* point to the existing head */
  head = tmp;               /* make the new node be head  */

  while (0 != head) {       /* count the whole list        */
    count = count + head->data; /* sum the next node       */
    head=head->next;        /* set up to process next node */
  }

  while (0 != head) {       /* free the whole list         */
    free(head->data);       /* recover data memory         */
    tmp = head->next;       /* remember the next node      */
    free(head);             /* recover the node memory     */
    head=tmp;               /* set up to process next node */
  }
}
```

Note that now we have

```
   DWORD PTR [ebp-4]  = head
   DWORD PTR [ebp-8]  = tmp
   DWORD PTR [ebp-12] = count

00000000 <main>:
   0: 55                  push  ebp
   1: 89 e5               mov   ebp,esp
   3: 83 ec 18            sub   esp,0x18
   6: 83 e4 f0            and   esp,0xfffffff0
   9: b8 00 00 00 00      mov   eax,0x0
   e: 29 c4               sub   esp,eax

list *head = (void *)0;  /* head points to nothing    */
  10: c7 45 fc 00 00 00 00  mov   DWORD PTR [ebp-4],0x0 (head)
```

```
int count = 0;          /* a counter                    */
  17: c7 45 f4 00 00 00 00  mov   DWORD PTR [ebp-12],0x0 (count)

tmp = (list *)malloc(sizeof(*tmp));  /* allocate node */
  1e: 83 ec 0c           sub   esp,0xc
  21: 6a 08              push  0x8
  23: e8 fc ff ff ff     call  24 <main+0x24>
  28: 83 c4 10           add   esp,0x10
  2b: 89 45 f8           mov   DWORD PTR [ebp-8],eax (tmp)

tmp -> data = 3;        /* set the data memory */
  2e: 8b 45 f8           mov   eax,DWORD PTR [ebp-8] (tmp)
  31: c7 00 03 00 00 00  mov   DWORD PTR [eax],0x3

tmp -> next = head;     /* point to the existing head */
  37: 8b 55 f8           mov   edx,DWORD PTR [ebp-8] (tmp)
  3a: 8b 45 fc           mov   eax,DWORD PTR [ebp-4] (head)
  3d: 89 42 04           mov   DWORD PTR [edx+4],eax

head = tmp;             /* and make the new node be head */
  40: 8b 45 f8           mov   eax,DWORD PTR [ebp-8] (tmp)
  43: 89 45 fc           mov   DWORD PTR [ebp-4],eax (head)

while (0 != head) {     /* count the whole list */
  46: 83 7d fc 00        cmp   DWORD PTR [ebp-4],0x0 (head)
  4a: 75 02              jne   4e <main+0x4e>        (loop body)
  4c: eb 15              jmp   63 <main+0x63>        (loop exit)

count = count + head->data; /* sum the next node */
  4e: 8b 45 fc           mov   eax,DWORD PTR [ebp-4] (head)
  51: 8b 10              mov   edx,DWORD PTR [eax]   (head->data)
  53: 8d 45 f4           lea   eax,[ebp-12]          (count)
  56: 01 10              add   DWORD PTR [eax],edx

head=head->next;        /* set up to process next node */
  58: 8b 45 fc           mov   eax,DWORD PTR [ebp-4] (head)
  5b: 8b 40 04           mov   eax,DWORD PTR [eax+4] (head->next)
  5e: 89 45 fc           mov   DWORD PTR [ebp-4],eax (head)
  61: eb e3              jmp   46 <main+0x46>        (loop top)

  63: 90                 nop                         (loop exit)

while (0 != head) {     /* free the whole list  */
  64: 83 7d fc 00        cmp   DWORD PTR [ebp-4],0x0 (head)
  68: 75 02              jne   6c <main+0x6c>        (loop body)
```

```
  6a: eb 2f              jmp   9b <main+0x9b>        (loop exit)

free(head->data);       /* recover data memory    */
  6c: 83 ec 0c           sub   esp,0xc
  6f: 8b 45 fc           mov   eax,DWORD PTR [ebp-4] (head)
  72: ff 30              push  DWORD PTR [eax]       (head->data)
  74: e8 fc ff ff ff     call  75 <main+0x75>
  79: 83 c4 10           add   esp,0x10

tmp = head->next;       /* remember the next node  */
  7c: 8b 45 fc           mov   eax,DWORD PTR [ebp-4] (head)
  7f: 8b 40 04           mov   eax,DWORD PTR [eax+4] (head->next)
  82: 89 45 f8           mov   DWORD PTR [ebp-8],eax (tmp)

free(head);             /* recover the node memory */
  85: 83 ec 0c           sub   esp,0xc
  88: ff 75 fc           push  DWORD PTR [ebp-4]     (head)
  8b: e8 fc ff ff ff     call  8c <main+0x8c>
  90: 83 c4 10           add   esp,0x10

head=tmp;               /* set up to process next node */
  93: 8b 45 f8           mov   eax,DWORD PTR [ebp-8] (tmp)
  96: 89 45 fc           mov   DWORD PTR [ebp-4],eax (head)
  99: eb c9              jmp   64 <main+0x64>        (loop top)
  9b: c9                 leave                        (loop exit)
  9c: c3                 ret
```

8 Binary Trees

8.1 binarytree.c

```
typedef struct btnode {      /* a binary tree node contains */
  int data;                  /* a place to point to data    */
  struct btnode *left;       /* a pointer to the left node  */
  struct btnode *right;      /* a pointer to the right node */
} tree;

int main(int argc, char *argv[]) {
  tree *node = (void *)0;    /* node points to nothing    */
  tree *tmp;                 /* a place to put a new node */

  tmp = (tree *)malloc(sizeof(*tmp));  /* allocate node */
  tmp -> data = 3;           /* set the data        */
  tmp -> left = (void *)0;   /* unset left subtree  */
  tmp -> right = (void *)0;  /* unset right subtree */
```

```
}
```

Note that here we have allocated a single tree node.

```
   DWORD PTR [ebp-4]   node
   DWORD PTR [ebp-8]   tmp

00000000 <main>:
   0: 55                    push   ebp
   1: 89 e5                 mov    ebp,esp
   3: 83 ec 08              sub    esp,0x8
   6: 83 e4 f0              and    esp,0xfffffff0
   9: b8 00 00 00 00        mov    eax,0x0
   e: 29 c4                 sub    esp,eax

tree *node = (void *)0;    /* node points to nothing */
  10: c7 45 fc 00 00 00 00  mov    DWORD PTR [ebp-4],0x0

tmp = (tree *)malloc(sizeof(*tmp));  /* allocate node */
  17: 83 ec 0c              sub    esp,0xc
  1a: 6a 0c                 push   0xc
  1c: e8 fc ff ff ff        call   1d <main+0x1d>
  21: 83 c4 10              add    esp,0x10

tmp -> data = 3;           /* set the data */
  24: 89 45 f8              mov    DWORD PTR [ebp-8],eax
  27: 8b 45 f8              mov    eax,DWORD PTR [ebp-8]
  2a: c7 00 03 00 00 00     mov    DWORD PTR [eax],0x3

tmp -> left = (void *)0;   /* unset the left subtree */
  30: 8b 45 f8              mov    eax,DWORD PTR [ebp-8]
  33: c7 40 04 00 00 00 00  mov    DWORD PTR [eax+4],0x0

tmp -> right = (void *)0;  /* unset the right subtree */
  3a: 8b 45 f8              mov    eax,DWORD PTR [ebp-8]
  3d: c7 40 08 00 00 00 00  mov    DWORD PTR [eax+8],0x0
  44: c9                    leave
  45: c3                    ret
```

8.2 binarytree1.c

```
typedef struct btnode { /* a binary tree node contains     */
  int data;             /* a place to point to data        */
  struct btnode *left;  /* and a pointer to the left node  */
  struct btnode *right; /* and a pointer to the right node */
} tree;
```

```
int main(int argc, char *argv[]) {
  tree *node = (void *)0; /* node points to nothing     */
  tree *tmp;              /* a place to put a new node  */
  int count = 0;          /* a counter                  */

  tmp = (tree *)malloc(sizeof(*tmp));  /* allocate node */
  tmp -> data = 3;          /* set the data             */
  tmp -> left = (void *)0;  /* unset the left subtree   */
  tmp -> right = (void *)0; /* unset the right subtree  */

  if (0 == node)            /* if there is no tree           */
    node = tmp;             /* make the new node the tree    */
  else if (tmp->data < node->data) /* if the tmp is smaller  */
    node->left = tmp;       /* tack it onto the left         */
  else node->right = tmp;   /* nope. tack it onto the right  */

}


00000000 <main>:
   0: 55                   push ebp
   1: 89 e5                mov  ebp,esp
   3: 83 ec 18             sub  esp,0x18
   6: 83 e4 f0             and  esp,0xfffffff0
   9: b8 00 00 00 00       mov  eax,0x0
   e: 29 c4                sub  esp,eax

tree *node = (void *)0; /* node points to nothing     */
  10: c7 45 fc 00 00 00 00 mov  DWORD PTR [ebp-4],0x0 (node)

int count = 0;          /* a counter                  */
  17: c7 45 f4 00 00 00 00 mov  DWORD PTR [ebp-12],0x0 (count)

tmp = (tree *)malloc(sizeof(*tmp));  /* allocate node */
  1e: 83 ec 0c             sub  esp,0xc
  21: 6a 0c                push 0xc
  23: e8 fc ff ff ff       call 24 <main+0x24>
  28: 83 c4 10             add  esp,0x10

tmp -> data = 3;          /* set the data             */
  2b: 89 45 f8             mov  DWORD PTR [ebp-8],eax (tmp)
  2e: 8b 45 f8             mov  eax,DWORD PTR [ebp-8] (tmp-data)
  31: c7 00 03 00 00 00    mov  DWORD PTR [eax],0x3

tmp -> left = (void *)0;  /* unset the left subtree   */
```

```
  37: 8b 45 f8              mov  eax,DWORD PTR [ebp-8] (tmp)
  3a: c7 40 04 00 00 00 00 mov  DWORD PTR [eax+4],0x0 (tmp-left)

tmp -> right = (void *)0; /* unset the right subtree  */
  41: 8b 45 f8              mov  eax,DWORD PTR [ebp-8] (tmp)
  44: c7 40 08 00 00 00 00 mov  DWORD PTR [eax+8],0x0 (tmp-right)

if (0 == node)            /* if there is no tree           */
  4b: 83 7d fc 00           cmp  DWORD PTR [ebp-4],0x0 (node)
  4f: 75 08                 jne  59 <main+0x59>

node = tmp;              /* make the new node the tree    */
  51: 8b 45 f8              mov  eax,DWORD PTR [ebp-8] (tmp)
  54: 89 45 fc              mov  DWORD PTR [ebp-4],eax (node)
  57: eb 20                 jmp  79 <main+0x79>        (jmp exit)

else if (tmp->data < node->data) /* if the tmp is smaller  */
  59: 8b 45 f8              mov  eax,DWORD PTR [ebp-8] (tmp)
  5c: 8b 55 fc              mov  edx,DWORD PTR [ebp-4] (node)
  5f: 8b 00                 mov  eax,DWORD PTR [eax]   (tmp-data)
  61: 3b 02                 cmp  eax,DWORD PTR [edx]   (node-data)
  63: 7d 0b                 jge  70 <main+0x70>        (jmp else)

node->left = tmp;        /* tack it onto the left         */
  65: 8b 55 fc              mov  edx,DWORD PTR [ebp-4] (node)
  68: 8b 45 f8              mov  eax,DWORD PTR [ebp-8] (tmp)
  6b: 89 42 04              mov  DWORD PTR [edx+4],eax (node-left)
  6e: eb 09                 jmp  79 <main+0x79>        (jmp exit)

else node->right = tmp;   /* nope. tack it onto the right  */
  70: 8b 55 fc              mov edx,DWORD PTR [ebp-4] (node)
  73: 8b 45 f8              mov eax,DWORD PTR [ebp-8] (tmp)
  76: 89 42 08              mov DWORD PTR [edx+8],eax (node-right)
  79: c9                    leave                     (exit)
  7a: c3                    ret
```

9 Strings

9.1 string.c

```
int main() {
  char *a = "asdf";
  return(0);
}
```

Oddly enough I see no initialization of the variable “a”.

```
   0: 55                    push   ebp
   1: 89 e5                 mov    ebp,esp
   3: 83 ec 08              sub    esp,0x8
   6: 83 e4 f0              and    esp,0xfffffff0
   9: b8 00 00 00 00        mov    eax,0x0
   e: 29 c4                 sub    esp,eax

return(0);
  10: c7 45 fc 00 00 00 00  mov    DWORD PTR [ebp-4],0x0
  17: b8 00 00 00 00        mov    eax,0x0
  1c: c9                    leave
  1d: c3                    ret
```

9.2 string1.c

```
int main() {
  char *a = "asdf";
  unsigned char b = '\0';
  int i = 0;
  while (a[i] != b)
    i++;
  printf("%d\n",i);
  return(i);
}
```

The stack appears to be:

```
   DWORD PTR [ebp-4]    a
   BYTE PTR [ebp-5]     b
   DWORD PTR [ebp-12]   i
```

In this case we look at the fully resolved elf file because the string is held in data storage (the heap). A reference to the string is not resolved until final link time. The presence of the pattern:

```
    movsx edx,BYTE PTR [eax]     <== sign extended char load
    movzx eax,BYTE PTR [ebp-5]   <== zero extended char load
    cmp  edx,eax                 <== compare (implied byte)
```

The fact that the compare is a full register compare but the data loaded into the register can only be byte makes the pattern a bit harder to detect.

```
080482f4 <main>:
 80482f4: 55              push ebp
 80482f5: 89 e5           mov  ebp,esp
 80482f7: 83 ec 18        sub  esp,0x18
 80482fa: 83 e4 f0        and  esp,0xfffffff0
 80482fd: b8 00 00 00 00 mov  eax,0x0
```

```
 8048302: 29 c4          sub  esp,eax

char *a = "asdf";
 8048304: c7 45 fc e4 83 04 08
                         mov DWORD PTR [ebp-4],0x80483e4 (*a)

unsigned char b = '\0';
 804830b: c6 45 fb 00    mov  BYTE PTR [ebp-5],0x0 (b)

int i = 0;
 804830f: c7 45 f4 00 00 00 00  mov  DWORD PTR [ebp-12],0x0 (i)

while (a[i] != b)
 8048316: 8b 45 f4       mov  eax,DWORD PTR [ebp-12]     (i)
 8048319: 03 45 fc       add  eax,DWORD PTR [ebp-4]      (a)
 804831c: 0f be 10       movsx edx,BYTE PTR [eax]        (a[i])
 804831f: 0f b6 45 fb    movzx eax,BYTE PTR [ebp-5]      (b)
 8048323: 39 c2          cmp  edx,eax                    (!=)
 8048325: 75 02          jne  8048329 <main+0x35> (loop body)
 8048327: eb 07          jmp  8048330 <main+0x3c> (loop end)

i++
 8048329: 8d 45 f4       lea  eax,[ebp-12]              (i)
 804832c: ff 00          inc  DWORD PTR [eax]           (i++)
 804832e: eb e6          jmp  8048316 <main+0x22> (loop head)

return(i);
 8048330: 8b 45 f4       mov  eax,DWORD PTR [ebp-12] (i)
 8048333: c9             leave
 8048334: c3             ret
```

9.3 string2.c

The only change we've made here is that the end test for the string does not use a variable but explicitly uses null. The presence of the

```
  cmp  BYTE PTR [eax],0x0
```

byte compare to zero could be a telltale sign of string walking.

```
int main() {
  char *a = "asdf";
  int i = 0;
  while (a[i] != '\0')
    i++;
  return(i);
}
```

```
080482f4 <main>:
 80482f4: 55             push ebp
 80482f5: 89 e5          mov  ebp,esp
 80482f7: 83 ec 08       sub  esp,0x8
 80482fa: 83 e4 f0       and  esp,0xfffffff0
 80482fd: b8 00 00 00 00 mov  eax,0x0
 8048302: 29 c4          sub  esp,eax

char *a = "asdf";
 8048304: c7 45 fc d8 83 04 08
                         mov  DWORD PTR [ebp-4],0x80483d8 (*a)

int i = 0;
 804830b: c7 45 f8 00 00 00 00
                         mov  DWORD PTR [ebp-8],0x0 (i)

while (a[i] != '\0')
 8048312: 8b 45 f8       mov  eax,DWORD PTR [ebp-8] (a)
 8048315: 03 45 fc       add  eax,DWORD PTR [ebp-4] (a[i])
 8048318: 80 38 00       cmp  BYTE PTR [eax],0x0    (!=0)
 804831b: 75 02          jne  804831f <main+0x2b> (loop body)
 804831d: eb 07          jmp  8048326 <main+0x32> (loop end)

i++;
 804831f: 8d 45 f8       lea  eax,[ebp-8]         (i)
 8048322: ff 00          inc  DWORD PTR [eax]     (i++)
 8048324: eb ec          jmp  8048312 <main+0x1e> (loop head)

return(i);
 8048326: 8b 45 f8       mov  eax,DWORD PTR [ebp-8] (i)
 8048329: c9             leave
 804832a: c3             ret
```

9.4 string3.c

Here we use the "unsigned" attribute of the character string to suggest that the compiler generate byte-explicit compare code.

```
int main() {
  unsigned char *a = "asdf";
  unsigned char b = '\0';
  int i = 0;
  while (a[i] != b)
    i++;
  return(i);
}
```

```
 80482f4: 55             push ebp
 80482f5: 89 e5          mov  ebp,esp
 80482f7: 83 ec 18       sub  esp,0x18
 80482fa: 83 e4 f0       and  esp,0xfffffff0
 80482fd: b8 00 00 00 00 mov  eax,0x0
 8048302: 29 c4          sub  esp,eax

unsigned char *a = "asdf";
 8048304: c7 45 fc e0 83 04 08
                         mov  DWORD PTR [ebp-4],0x80483e0

unsigned char b = '\0';
 804830b: c6 45 fb 00    mov  BYTE PTR [ebp-5],0x0

int i = 0;
 804830f: c7 45 f4 00 00 00 00
                         mov  DWORD PTR [ebp-12],0x0

while (a[i] != b)
 8048316: 8b 45 f4       mov  eax,DWORD PTR [ebp-12] (i)
 8048319: 03 45 fc       add  eax,DWORD PTR [ebp-4]  (a+i)
 804831c: 8a 00          mov  al,BYTE PTR [eax]      (a[i])
 804831e: 3a 45 fb       cmp  al,BYTE PTR [ebp-5]    (b)
 8048321: 75 02          jne  8048325 <main+0x31> (loop body)
 8048323: eb 07          jmp  804832c <main+0x38> (loop end)

i++;
 8048325: 8d 45 f4       lea  eax,[ebp-12]         (i)
 8048328: ff 00          inc  DWORD PTR [eax]      (i++)
 804832a: eb ea          jmp  8048316 <main+0x22> (loop head)

return(i);
 804832c: 8b 45 f4       mov  eax,DWORD PTR [ebp-12] (i)
 804832f: c9             leave
 8048330: c3             ret
```

References

[1] IA-32 Intel Architecture Software Developer's Manual Volume 2A: Instruction Set Reference, A-M Order Number 253666-016 June 2005,

[2] IA-32 Intel Architecture Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z Order Number 253667-016 June 2005

[3] http://www.unixwiz.net/techtips/win32-callconv-asm.html