x86 Disassembler Internals

Toorcon 7
September 2005

Richard Johnson | rjohnson@idefense.com

Welcome

Who am I?

- Richard Johnson
- Senior Security Engineer, iDEFENSE Labs
- Other Research: nologin.org / uninformed.org

What is iDEFENSE?

What is the purpose of this talk?

- Introduce the core components of a disassembler
- Refresh binary format parsing concepts
- Explore programmatic disassembly analysis methods
- Inspire the audience to take development of binary analysis tools a little further and explore the potential for automated disassembly analysis programs.

Agenda

- Introduction
- Disassembler Core Architecture
  - IA-32 instruction decoder
- Binary Format Parsing
  - Executable and Linkable Format (ELF)
  - Portable Executable (PE)
- Disassembly Analysis
  - Data Associations
    - Function Recognition
    - Cross References
  - Path Analysis
  - Auto-commenting
- Demo & Conclusion

Introduction

- Disassemblers decode machine language into human-readable mnemonics

- Reverse-engineering in the software world makes use of a disassembler to understand an unknown or closed system.

- Reverse-engineering has many applications
  - Interoperability
  - Copyright evasion
  - Technology theft
  - Software security

Introduction

The goal is to gain a higher understanding of the pure binary code that is available.

The low-level disassembler is powerful, yet limited. Manual reverse-engineering is tedious.

Advanced disassemblers are capable of recognizing structures and relationships within binary code.

- Executable binary format handling
- Function recognition
- String recognition
- Code references
- Data references

Disassembler Core Architecture

Disassembler Core Architecture

The core of any disassembler is the instruction decoder.

The instruction decoder translates compiled binary instructions back into mnemonics as defined by the architecture's reference manuals.

The decoding process is performed by doing lookups in an opcode table, which defines flags and operand values for the specified opcode.

IA-32 Instruction Decoding

IA-32 instruction set is CISC and includes many operands which do similar things or combine multiple operations into one instruction.

RISC architectures have far fewer opcodes and have much simpler lookup tables

IA-32 has variable length opcodes, which results in very kludgy set of tables for each set of opcodes of differing lengths

IA-32 Opcode Table

```
// 1-byte opcodes
INST inst_table1[256] = {
        { INSTRUCTION_TYPE_ADD,    "add",      AM_E|OT_b|P_w,              AM_G|OT_b|P_r,
   FLAGS_NONE,   1 },
        { INSTRUCTION_TYPE_ADD,    "add",      AM_E|OT_v|P_w,              AM_G|OT_v|P_r,
   FLAGS_NONE,   1 },
        { INSTRUCTION_TYPE_ADD,    "add",      AM_G|OT_b|P_w,              AM_E|OT_b|P_r,
   FLAGS_NONE,   1 },
        { INSTRUCTION_TYPE_ADD,    "add",      AM_G|OT_v|P_w,              AM_E|OT_v|P_r,
   FLAGS_NONE,   1 },
        { INSTRUCTION_TYPE_ADD,    "add",      AM_REG|REG_EAX|OT_b|P_w,    AM_I|OT_b|P_r,
   FLAGS_NONE,   0 },
        { INSTRUCTION_TYPE_ADD,    "add",      AM_REG|REG_EAX|OT_v|P_w,    AM_I|OT_v|P_r,
   FLAGS_NONE,   0 },
        { INSTRUCTION_TYPE_PUSH,   "push",     AM_REG|REG_ES|F_r|P_r,      FLAGS_NONE,
FLAGS_NONE,   0 },
        { INSTRUCTION_TYPE_POP,    "pop",      AM_REG|REG_ES|F_r|P_w,      FLAGS_NONE,
FLAGS_NONE,   0 },
        { INSTRUCTION_TYPE_OR,     "or",       AM_E|OT_b|P_w,              AM_G|OT_b|P_r,
   FLAGS_NONE,   1 },
        { INSTRUCTION_TYPE_OR,     "or",       AM_E|OT_v|P_w,              AM_G|OT_v|P_r,
   FLAGS_NONE,   1 },
```

Executable Binary Formats

Executable Binary Formats

Executable binary formats instruct an operating system how to initialize the required environment for an executable and how to place the binary in memory for execution.

The kernel is responsible for:

- Creating a new task
- Loading a binary into memory
- Loading a binary's interpreter
- Transferring control to the new task

The kernel understands the binary as a series of memory segments.

Executable Binary Formats

Most binaries are dynamically linked

Execution control is transferred to the linker rather than the executable's entry point.

The linker is responsible for:

- Library loading
- Symbol relocation
- Symbol resolution

The linker interprets the binary as a series of sections with special run-time purposes.

Executable and Linkable Format (ELF)

Executable and Linkable Format

Originally introduced in UNIX SVR4 in 1989 and is now used in Linux and most System V derivatives like Solaris, IRIX, FreeBSD and HP-UX

Reference:

ELF Portable Formats Specification, Version 1.1

Tool Interface Standards (TIS)

Contains important information for binary analysis including section headers, symbol tables, string tables, dynamic linking information.

Executable and Linkable Format (ELF)

ELF Objects

- Header info
  - ELF Header
    - Details how to access headers within the object and identifies the executable's properties
  - Section Header Table
    - Details how to access various sections in the file (linker)
  - Program Header Table
    - Details how to load the executable into memory (kernel)
- Object Code
- Relocation info
- Symbols
  - .symtab – Contains information about all symbols being defined or imported (not present if binary is stripped)
  - .dynsym – Contains information about external symbols that need to be resolved or dynamic symbols that are exported by the binary

Executable and Linkable Format (ELF)

ELF Header

- Located at the beginning of every ELF binary
- Identifies properties of the ELF binary
- Details how to access section and program header tables

```
#define EI_NIDENT (16)

typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* Magic number and other info */
  Elf32_Half    e_type;                 /* Object file type */
  Elf32_Half    e_machine;              /* Architecture */
  Elf32_Word    e_version;              /* Object file version */
  Elf32_Addr    e_entry;                /* Entry point virtual address */
  Elf32_Off     e_phoff;                /* Program header table file offset */
  Elf32_Off     e_shoff;                /* Section header table file offset */
  Elf32_Word    e_flags;                /* Processor-specific flags */
  Elf32_Half    e_ehsize;               /* ELF header size in bytes */
  Elf32_Half    e_phentsize;            /* Program header table entry size */
  Elf32_Half    e_phnum;                /* Program header table entry count */
  Elf32_Half    e_shentsize;            /* Section header table entry size */
  Elf32_Half    e_shnum;                /* Section header table entry count */
  Elf32_Half    e_shstrndx;             /* Section header string table index */
} Elf32_Ehdr;
```

Executable and Linkable Format (ELF)

iDEFENSE

ELF Section Header Table

- Located by adding:
  - `base_addr + Elf32_Ehdr->e_shoff`
- Describes sections in the binary
  - Contains flags that describe memory permissions and type of data contained in the section
  - Can describe relationships between two sections in an ELF file.
- Disassembler should take note of special sections
  - .dynamic, .plt, .got, .symtab, .dynsym, .text

```
typedef struct
{
  Elf32_Word    sh_name;                /* Section name (string tbl index) */
  Elf32_Word    sh_type;                /* Section type */
  Elf32_Word    sh_flags;               /* Section flags */
  Elf32_Addr    sh_addr;                /* Section virtual addr at execution */
  Elf32_Off     sh_offset;              /* Section file offset */
  Elf32_Word    sh_size;                /* Section size in bytes */
  Elf32_Word    sh_link;                /* Link to another section */
  Elf32_Word    sh_info;                /* Additional section information */
  Elf32_Word    sh_addralign;           /* Section alignment */
  Elf32_Word    sh_entsize;             /* Entry size if section holds table */
} Elf32_Shdr;
```

ELF Symbols

Sections of type SHT_SYMTAB or SHT_DYNSYM contain symbol tables. which are identical can can be parsed the same way.

The st_info member describes symbol type, for example whether the symbol is a code or data object.

Symbols will be associated with code locations once disassembly is performed.

```
typedef struct
{
  Elf32_Word    st_name;                /* Symbol name (string tbl index) */
  Elf32_Addr    st_value;               /* Symbol value */
  Elf32_Word    st_size;                /* Symbol size */
  unsigned char st_info;                /* Symbol type and binding */
  unsigned char st_other;               /* Symbol visibility */
  Elf32_Section st_shndx;               /* Section index */
} Elf32_Sym;
```

Executable and Linkable Format (ELF)

ELF Symbol Parsing

Enumerate section headers:

```
for (shdr = (base + ehdr->e_shoff), count = 0;
     count < ehdr->e_shnum;
     shdr++, count++)
{
     if(shdr->sh_type == SHT_DYNSYM ||
        shdr->sh_type == SHT_SYMTAB)
          // parse symbol table
}
```

Enumerate the symbol table:

```
for (sym = (base + shdr->sh_offset), symidx = 0;
     symidx < (shdr->sh_size / shdr->sh_entsize);
     sym++, symidx++)
{
     // store symbol information
}
```

String table lookup:

```
Elf32_Shdr *strtab = base + ehdr->e_shoff
                   + (shdr->sh_link * ehdr->e_shentsize);
char *string = base + strtab->sh_offset + sym->st_name;
```

Section Header Struct

```
typedef struct
{
  Elf32_Word    sh_name;
  Elf32_Word    sh_type;
  Elf32_Word    sh_flags;
  Elf32_Addr    sh_addr;
  Elf32_Off     sh_offset;
  Elf32_Word    sh_size;
  Elf32_Word    sh_link;
  Elf32_Word    sh_info;
  Elf32_Word    sh_addralign;
  Elf32_Word    sh_entsize;
} Elf32_Shdr;
```

Symbol Table Struct

```
typedef struct
{
  Elf32_Word    st_name;
  Elf32_Addr    st_value;
  Elf32_Word    st_size;
  unsigned char st_info;
  unsigned char st_other;
  Elf32_Section st_shndx;
} Elf32_Sym;
```

Portable Executable Format (PE/COFF)

Portable Executable and Common Object File Format

- Originally introduced as part of the Win32 specification
- Derived from DEC's Common Object File Format (COFF)
- Object files are generated as COFF and later linked as PE binaries
- Reference:
  - Microsoft Portable Executable and Common Object File Format Specification Microsoft Corporation Revision 6.0 - February 1999

Portable Executable Format (PE/COFF)

iDEFENSE

PECOFF Structure

- DOS Stub + Signature
  - Pointer to PE Sig at offset 0x3c
  - Executable MS-DOS program
- IMAGE_NT_SIGNATURE (0x00004550)
- File Header (COFF)
- Optional Header (PE Header)
- Data Directories
  - Located at static offsets in the binary
  - Point to specific data structures
    - Imports, Exports, IAT, etc
- Section Headers
- Sections

```
+-------------------+
| DOS-stub          |
+-------------------+
| File-Header       |
+-------------------+
| optional header   |
|- - - - - - - - - -|
|                   |----------------+
| data directories  |                |
|                   |                |
|(RVAs to direc-    |-------------+  |
|tories in sections)|             |  |
|                   |---------+   |  |
|                   |         |   |  |
+-------------------+         |   |  |
|                   |-----+   |   |  |
| section headers   |     |   |   |  |
| (RVAs to section  |--+  |   |   |  |
|  borders)         |  |  |   |   |  |
+-------------------+<-+  |   |   |  |
|                   | <---|---+   |  |
| section data 1    |     |       |  |
|                   | <---|-------+  |
+-------------------+<----+          |
|                   |                |
| section data 2    |                |
|                   | <--------------+
+-------------------+
```

Portable Executable Format (PE/COFF)

COFF File Header

- Locate by adding the value at offset 0x3c to the base address
- Number of sections
- COFF Symbol table information
- Optional header size
- Characteristic flags
  - Byte ordering
  - Word size

```
typedef struct _COFF {
        WORD    Machine;
        WORD    NumberOfSections;
        DWORD   TimeDateStamp;
        DWORD   PointerToSymbolTable;
        DWORD   NumberOfSymbols;
        WORD    SizeOfOptionalHeader;
        WORD    Characteristics;
}COFF, *PCOFF;
```

```
+-------------------+
| DOS-stub          |
+-------------------+
| File-Header       |
+-------------------+
| optional header   |
|- - - - - - - - - -|
|                   |----------------+
| data directories  |                |
|                   |                |
|(RVAs to direc-    |-------------+  |
|tories in sections)|             |  |
|                   |---------+   |  |
|                   |         |   |  |
+-------------------+         |   |  |
|                   |-----+   |   |  |
| section headers   |     |   |   |  |
| (RVAs to section  |--+  |   |   |  |
|  borders)         |  |  |   |   |  |
+-------------------+<-+  |   |   |  |
|                   |  <--|---+   |  |
| section data 1    |     |       |  |
|                   |  <--|-------+  |
+-------------------+<----+          |
|                   |                |
| section data 2    |                |
|                   |  <-------------+
+-------------------+
```

Portable Executable Format (PE/COFF)

Optional Header (PE Hdr)

- Entry point
- Stack size
- Code segment size
- Data segment size
- Number of RVAs (Data Directories)

```
typedef struct _OPTHEADERS{
        WORD    magic;
        WORD    majmin;
        DWORD   SizeOfCode;
        DWORD   SizeOfInitializedData;
        DWORD   SizeofUninitializedData;
        DWORD   AddressOfEntryPoint;
        DWORD   BaseOfCode;
        DWORD   BaseOfData;
        DWORD   ImageBase;
        DWORD   SectionAlignment;
        DWORD   FileAlignment;
        WORD    MajorOperatingSystemVersion;
        WORD    MinorOperatingSystemVersion;
        WORD    MajorSubsystemVersion;
        WORD    MinorSubsystemVersion;
        DWORD   Reserved;
        DWORD   SizeOfImage;
        DWORD   SizeOfHeaders;
        DWORD   CheckSum;
        WORD    Subsystem;
        DWORD   DllCharacteristics;
        DWORD   SizeOfStackReserve;
        DWORD   SizeOfStackCommit;
        DWORD   SizeOfHeapReserve;
        DWORD   SizeOfHeapCommit;
        DWORD   LoaderFlags;
        DWORD   NumberOfRvaAndSizes;
}OPTHEADERS, *POPTHEADERS;
```

Portable Executable Format (PE/COFF)

COFF Section Tables

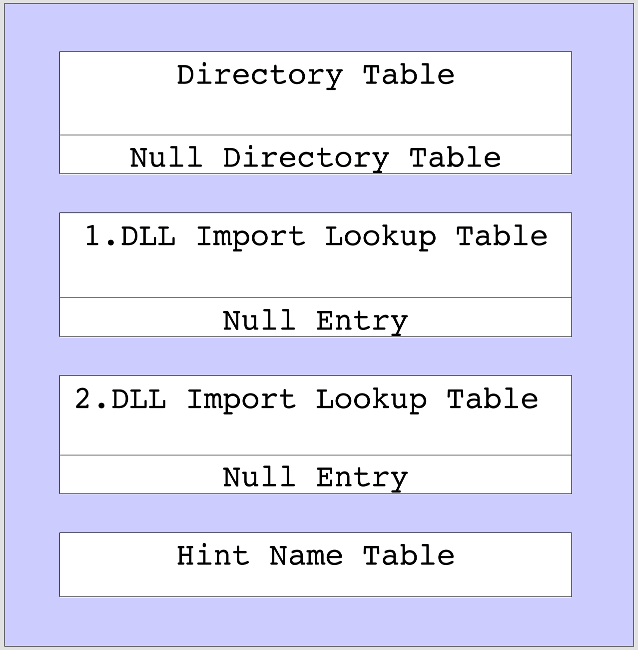
- Located by adding:
  ```
  base_addr + *(uint32)(base_addr + 0x3c)
  + sizeof(COFF) + PCOFF->SizeOfOptionalHeader
  ```
  - Then enumerate the data directories until you hit the section tables
- Relocation entries are only present in object files
- Line-number entries associate code with line numbers in source files
- Characteristic flags indicate section types, memory permissions, and alignment information

```
typedef struct _SECTIONTABLES {
        BYTE    Name[8];                 /* Section name */
        DWORD   VirtualSize;             /* Size of section in memory */
        DWORD   VirtualAddress;          /* Address of mapped section */
        DWORD   SizeOfRawData;           /* Size of section on disk */
        DWORD   PointerToRawData;        /* Section file offset */
        DWORD   PointerToRelocations;    /* Relocation entries file offset */
        DWORD   PointerToLineNumbers;    /* Line-number entries file offset */
        WORD    NumberOfRelocations;     /* Number of relocation entries */
        WORD    NumberOfLineNumbers;     /* Number of line-number entries */
        DWORD   Characteristics;         /* Characteristics flags */
}SECTIONTABLES, *PSECTIONTABLES;
```

Portable Executable Format (PE/COFF)

PECOFF Symbols

- Data_Directory[1] – Import Directory
  - .idata section
- Import Directory entries describe DLLs
  - DLL Name
  - RVA of Import Lookup Table
  - RVA of Import Address Table
- Image Thunk Data
  - Table of structures describing functions to be imported from the module




```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
        union {
                DWORD   Characteristics;
                PIMAGE_THUNK_DATA OriginalFirstThunk;
        } DUMMYUNIONNAME;
        DWORD   TimeDateStamp;
        DWORD   ForwarderChain;
        DWORD   Name;
        PIMAGE_THUNK_DATA FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR,*PIMAGE_IMPORT_DESCRIPTOR;
```

PE Symbol Parsing

- Locate and loop Import Directory Table
- Get the pointer to the FirstThunk
  - IID->FirstThunk
- Loop Thunks for symbol import data
  - struct IMAGE_IMPORT_BY_NAME array

```
Import name entry

typedef struct _IMAGE_IMPORT_BY_NAME {
        WORD    Hint;
        BYTE    Name[1];
} IMAGE_IMPORT_BY_NAME,*PIMAGE_IMPORT_BY_NAME;

Import Thunk

typedef struct _IMAGE_THUNK_DATA {
        union {
                LPBYTE    ForwarderString;
                PDWORD    Function;
                DWORD     Ordinal;
                PIMAGE_IMPORT_BY_NAME   AddressOfData;
        } u1;
} IMAGE_THUNK_DATA,*PIMAGE_THUNK_DATA;
```

Disassembly Analysis

Disassembly Analysis

Disassembly analysis attempts to aid the reverse engineer by automating some of the manual processes used when looking at assembly code dead listings

Programmatic disassembly analysis is an imperfect science. The more powerful the analyzer becomes, the closer it becomes to truly emulating the disassembled code

Disassembler analyzers lend themselves to interactive disassembly tools more than non-interactive

Analyzers can also power the logic of debugging tools built on top of the disassembler core

Function Recognition

Frame initialization from function prologue:

```
push %ebp          ; push old frame pointer
mov  %esp, %ebp    ; store current stack pointer as new frame
```

Cross reference calls in case of -fomit-frame-pointer

```
080483b4 |
........ |  ;;;;;;;;;;;;;;;;;;;;;;;;;;;
........ |  ;;; S U B R O U T I N E ;;;
........ |  ;;;;;;;;;;;;;;;;;;;;;;;;;;;
........ |  sub_080483b4:                        ;  xrefs:  0x08048403
........ |    sub     $0x3c, %esp                ;
080483b7 |    mov     0x40(%esp), %eax           ;
080483bb |    mov     %eax, 0x4(%esp)            ;
080483bf |    lea     0x10(%esp), %eax           ;
```

```
080483f4 |    shr     $0x4, %eax                 ;
080483f7 |    shl     $0x4, %eax                 ;
080483fa |    sub     %eax, %esp                 ;
080483fc |    movl    $0x804851e, (%esp)         ;
08048403 |    call    0x80483b4                  ;
```

Cross Referencing

- Cross referencing allows immediate recognition of relationships within the disassembly
- Data cross references allow tracking variables through various code blocks to determine where user input originates
- Code cross references allow for flow control graphing and loop detection
- Aid in code navigation when interactive disassemblers are used

Cross Referencing

- Use an instruction decoder library which incorporates operand permissions
  - Libdasm – available at www.nologin.org by jt
  - Libdisasm – available at bastard.sf.net by mammon
- For each instruction, analyze the operands for relationships
  - Check for operand types: IMMEDIATE, MEMORY, REGISTER
  - Check operand permission flags
- Classify certain operands depending on relationship
  - Flow control (call, branch, return)
  - Data manipulation (arithmetic)
- Stored both in a link list and a splay tree indexed by address for quicker searching

Flow Control

- Call
  - Indicates a new function
  - Needs to be checked against symbol tables
- Branch
  - jmp, longjmp, etc
  - Indicates new code 'block'
  - Code blocks can be analyzed for functionality
  - Used for loops, signal handlers, etc
- Return
  - Can be used to divert flow control by pushing a pointer to the stack

Disassembly Analysis

Path Analysis

Recursive disassembly analyzers follow flow control

By recording branches, graphs can be generated for visual recognition of related functions

When reverse engineering, entire code paths can be quickly grouped for functionality to speed the code recognition process

Linear disassemblers can not determine the relationships of code blocks, and may disassemble instructions incorrectly if data is injected in-between compiled code

Hand written assembly code can cause disassemblers to generate code that is completely incorrect

Auto-commenting

System Calls

- System calls allow access to system resources via the kernel
- Libc wrappers most system calls, however in case of hand-written assembly or library disassembly, recognition of system calls can speed the recognition of functionality within code blocks
- Syscall Numbers are stored in /usr/src/linux/include/asm/unistd.h
- Scripts can easily generate system call definitions to be applied to the disassembly analysis.
- Arguments are typically passed in registers, so once data xrefs are applied we can tell if user-supplied data is being used in a system call

Inline System Call prototypes

```
#define _syscall1(type,name,type1,arg1)
#define _syscall2(type,name,type1,arg1,type2,arg2)
#define _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3)
#define _syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4)
#define _syscall5(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,arg5)
#define _syscall6(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,arg5,type6,arg6)
```

Auto-commenting

Function call argument detection

- Function prologues swap the the current stack pointer into %ebp to represent the base of the stack for the local function
- Function arguments can be determined by internal references to offsets of %ebp
- In the case of code compiled without frame pointers, offsets to esp will be used.
- Arguments can be determined as local variables vs passed arguments depending on their offset to %ebp
- Depending on calling convention, arguments to functions are typically passed via the stack
  - Stdcall – push args in reverse order to the stack (last to first)
  - Fastcall – uses registers when possible to hold args
- Argument types can be determined via basic heuristics or by prototype parsing
  - Heuristics can determine if passed values are pointers to memory, string references or integer values

More Auto-commenting

- Assembly hinting
- Inline function recognition
- Loop detection

Graph Generation

Common graphing tools take input in the form of simple flat-file text markup.

Graphs can be generated to allow quick visualization of various properties of compiled code

Check out process stalker for good application of graphing applied to binary analysis

Debugger Integration

Combine powerful disassembly analyzers with runtime tracing

Determining bounds of allowed input data via disassembly analysis and code coverage analysis of the run-time tracing increases effectiveness of fuzzing

Demo & Conclusion

aterm

```
;----------------------------------------------------------------------
; Section 13  <.text>
; virtual address: 08048c90  file offset:   00000c90
; section size:    00001fc0  loadable:      YES
; section type:    CODE      permissions:   READ EXECUTE
;----------------------------------------------------------------------

08048c90 |    xor     %ebp, %ebp                    ;
08048c92 |    pop     %esi                          ;
08048c93 |    mov     %esp, %ecx                    ;
08048c95 |    and     $0xfffffff0, %esp             ;
08048c98 |    push    %eax                          ;
08048c99 |    push    %esp                          ;
08048c9a |    push    %edx                          ;
08048c9b |    push    $0x804ab80                    ;
08048ca0 |    push    $0x804ab20                    ;
08048ca5 |    push    %ecx                          ;
08048ca6 |    push    %esi                          ;
08048ca7 |    push    $0x8048ee0                    ;
08048cac |    call    0x8048b04                     ;
08048cb1 |    hlt                                   ;
08048cb2 |    nop                                   ;
08048cb3 |    nop                                   ;
08048cb4 |
........ |  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
........ |  ;;; S U B R O U T I N E ;;;
........ |  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
........ |  sub_08048cb4:                           ;  xrefs:   0x080489c2
........ |    push    %ebp                          ;
08048cb5 |    mov     %esp, %ebp                    ;
08048cb7 |    push    %ebx                          ;
08048cb8 |    call    0x8048cbd                     ;
08048cbd |
........ |  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
........ |  ;;; S U B R O U T I N E ;;;
........ |  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
........ |  sub_08048cbd:                           ;  xrefs:   0x08048cb8
........ |    pop     %ebx                          ;
08048cbe |    add     $0x3c77, %ebx                 ;
08048cc4 |    push    %eax                          ;
08048cc5 |    mov     0xb8(%ebx), %eax              ;
08048ccb |    test    %eax, %eax                    ;
08048ccd |    jz      0x8048cd1                     ;
08048ccf |    call    %eax                          ;
08048cd1 |
........ |  loc_08048cd1:                           ;  xrefs:   0x08048ccd
........ |    mov     0xfffffffc(%ebp), %ebx        ;
08048cd4 |    leave                                 ;
08048cd5 |    ret                                   ;
:
```

aterm

```
;----------------------------------------------------------------------
; Section 15  <.rodata>
; virtual address: 0804ac80  file offset:   00002c80
; section size:    00000bad  loadable:      YES
; section type:    DATA      permissions:   READ
;----------------------------------------------------------------------

00000000 |  03 00 00 00 01 00 02 00 67 72 6F 75 70 00 67 72  | ........group.gr
00000010 |  6F 75 70 73 00 6E 61 6D 65 00 72 65 61 6C 00 68  | oups.name.real.h
00000020 |  65 6C 70 00 76 65 72 73 69 6F 6E 00 0A 52 65 70  | elp.version..Rep
00000030 |  6F 72 74 20 62 75 67 73 20 74 6F 20 3C 25 73 3E  | ort bugs to <%s>
00000040 |  2E 0A 00 62 75 67 2D 63 6F 72 65 75 74 69 6C 73  | ...bug-coreutils
00000050 |  40 67 6E 75 2E 6F 72 67 00 2F 75 73 72 2F 73 68  | @gnu.org./usr/sh
00000060 |  61 72 65 2F 6C 6F 63 61 6C 65 00 00 63 6F 72 65  | are/locale..core
00000070 |  75 74 69 6C 73 00 61 67 6E 72 75 47 00 44 61 76  | utils.agnruG.Dav
00000080 |  69 64 20 4D 61 63 4B 65 6E 7A 69 65 00 41 72 6E  | id MacKenzie.Arn
00000090 |  6F 6C 64 20 52 6F 62 62 69 6E 73 00 35 2E 32 2E  | old Robbins.5.2.
000000a0 |  31 00 69 64 00 25 73 3A 20 4E 6F 20 73 75 63 68  | 1.id.%s: No such
000000b0 |  20 75 73 65 72 00 28 25 73 29 00 20 67 69 64 3D  |  user.(%s). gid=
000000c0 |  25 75 00 20 65 75 69 64 3D 25 75 00 20 65 67 69  | %u. euid=%u. egi
000000d0 |  64 3D 25 75 00 20 67 72 6F 75 70 73 3D 00 00 00  | d=%u. groups=...
000000e0 |  88 AC 04 08 00 00 00 00 00 00 00 00 67 00 00 00  | ............g...
000000f0 |  8E AC 04 08 00 00 00 00 00 00 00 00 47 00 00 00  | ............G...
00000100 |  95 AC 04 08 00 00 00 00 00 00 00 00 6E 00 00 00  | ............n...
00000110 |  9A AC 04 08 00 00 00 00 00 00 00 00 72 00 00 00  | ............r...
00000120 |  31 AD 04 08 00 00 00 00 00 00 00 00 75 00 00 00  | 1...........u...
00000130 |  9F AC 04 08 00 00 00 00 00 00 00 00 7E FF FF FF  | ............~...
00000140 |  A4 AC 04 08 00 00 00 00 00 00 00 00 7D FF FF FF  | ............}...
00000150 |  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
00000160 |  54 72 79 20 60 25 73 20 2D 2D 68 65 6C 70 27 20  | Try `%s --help'
00000170 |  66 6F 72 20 6D 6F 72 65 20 69 6E 66 6F 72 6D 61  | for more informa
00000180 |  74 69 6F 6E 2E 0A 00 00 00 00 00 00 00 00 00 00  | tion............
00000190 |  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
000001a0 |  55 73 61 67 65 3A 20 25 73 20 5B 4F 50 54 49 4F  | Usage: %s [OPTIO
000001b0 |  4E 5D 2E 2E 2E 20 5B 55 53 45 52 4E 41 4D 45 5D  | N]... [USERNAME]
000001c0 |  0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
000001d0 |  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  | ................
000001e0 |  50 72 69 6E 74 20 69 6E 66 6F 72 6D 61 74 69 6F  | Print informatio
000001f0 |  6E 20 66 6F 72 20 55 53 45 52 4E 41 4D 45 2C 20  | n for USERNAME,
00000200 |  6F 72 20 74 68 65 20 63 75 72 72 65 6E 74 20 75  | or the current u
00000210 |  73 65 72 2E 0A 0A 20 20 2D 61 20 20 20 20 20 20  | ser...  -a
00000220 |  20 20 20 20 20 20 20 20 69 67 6E 6F 72 65 2C 20  |         ignore,
00000230 |  66 6F 72 20 63 6F 6D 70 61 74 69 62 69 6C 69 74  | for compatibilit
00000240 |  79 20 77 69 74 68 20 6F 74 68 65 72 20 76 65 72  | y with other ver
00000250 |  73 69 6F 6E 73 0A 20 20 2D 67 2C 20 2D 2D 67 72  | sions.  -g, --gr
00000260 |  6F 75 70 20 20 20 20 20 70 72 69 6E 74 20 6F 6E  | oup     print on
00000270 |  6C 79 20 74 68 65 20 65 66 66 65 63 74 69 76 65  | ly the effective
00000280 |  20 67 72 6F 75 70 20 49 44 0A 20 20 2D 47 2C 20  |  group ID.  -G,
00000290 |  2D 2D 67 72 6F 75 70 73 20 20 20 20 70 72 69 6E  | --groups    prin
:
```

Questions?