# x86 Disassembler Internals

## 22c3: Private Investigations
## December 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

- Disassember Core Architecture
  - Instruction Decoder
    - IA-32
  - Executable Binary Format Parser
    - Executable and Linkable Format (ELF)
    - Portable Executable (PE)
  - Disassembly Analyzer

- Basic Disassembly Analysis
  - Data Associations
    - Function Recognition
    - Cross-Referencing
  - Hinting
    - System Calls
    - Function Calls
    - Assembly Syntax
  - Demo (codis)

# Agenda

- Advanced Disassembly Analysis
  - Path Analysis
    - Loop Detection
  - Data Analysis
    - Static data flow analysis
    - Emulation
    - Data Structure Recognition
    - Demo (ida-x86emu + idastruct)

- 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 of reverse engineering is to gain a higher understanding of the machine readable 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 / argument detection
  - Code and data cross-referencing
  - Structure recognition

# Disassembler Core Architecture

# Disassembler Core Architecture

- The core function of a disassembler is to interpret executable files and decode their instructions.

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

- The executable file parsers are each designed to extract useful information from various executable binary formats.

# IA-32 Instruction Decoding

- The IA-32 processor is considered to be a CISC architecture. The instruction set includes many operands which do similar things or combine multiple operations into one instruction.
- RISC architectures have far fewer opcodes and simpler opcode lookup algorithms
- IA-32 has variable length opcodes and opcode extensions, which results in a larger set of tables for opcode and operand decoding.

# IA-32 Instruction Decoding

- IA-32 Opcode Table and Flags

```
// 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 },
....
// Operand Addressing Methods, from the Intel manual
#define MASK_AM(x) ((x) & 0x00ff0000)
#define AM_A 0x00010000         // Direct address with segment prefix
#define AM_C 0x00020000         // MODRM reg field defines control register
#define AM_D 0x00030000         // MODRM reg field defines debug register
#define AM_E 0x00040000         // MODRM byte defines reg/memory address
#define AM_G 0x00050000         // MODRM byte defines general-purpose reg
....
// Operand Types, from the intel manual
#define MASK_OT(x) ((x) & 0xff000000)
#define OT_a  0x01000000
#define OT_b  0x02000000        // always 1 byte
#define OT_c  0x03000000        // byte or word, depending on operand
#define OT_d  0x04000000        // double-word
#define OT_q  0x05000000        // quad-word
#define OT_dq 0x06000000        // double quad-word
```

(example taken from from libdasm.h)

# IA-32 Instruction Decoding

- IA-32 Opcode Decoding
  - Parse opcode prefixes
    - First byte of opcode
    - Indicate multi-byte opcodes or opcode extensions
    - Determine lookup table
  - Perform lookup in opcode table by current index value

- IA-32 Operand Decoding
  - Index opcode table to get operand types and flags
    - Addressing method
      - Register
      - Immediate
      - Displacement
    - Operand type
      - Word
      - Double-word
      - Float

# 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
  - Official 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)

- 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;
```

# Executable and Linkable Format (ELF)

- ELF Symbols
  - Sections of type SHT_SYMTAB or SHT_DYNSYM contain symbol tables. which are identical and 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 Structure**

```
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 Structure**

```
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
  - Offical reference:

    *Microsoft Portable Executable and Common Object File Format Specification*

    *Microsoft Corporation Revision 6.0 - February 1999*

# Portable Executable Format (PE/COFF)

- 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)

```
typedef struct _OPTHEADERS{
        WORD    Magic;
        BYTE    MajorLinkerVersion;
        BYTE    MinorLinkerVersion;
        DWORD   SizeOfCode;                    // code segment size
        DWORD   SizeOfInitializedData;         // data segment size
        DWORD   SizeofUninitializedData;       // data segment size
        DWORD   AddressOfEntryPoint;           // entry point
        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;           // data directories
}OPTHEADERS, *POPTHEADERS;
```

# Portable Executable Format (PE/COFF)

- COFF Section Tables
  - Located by adding:

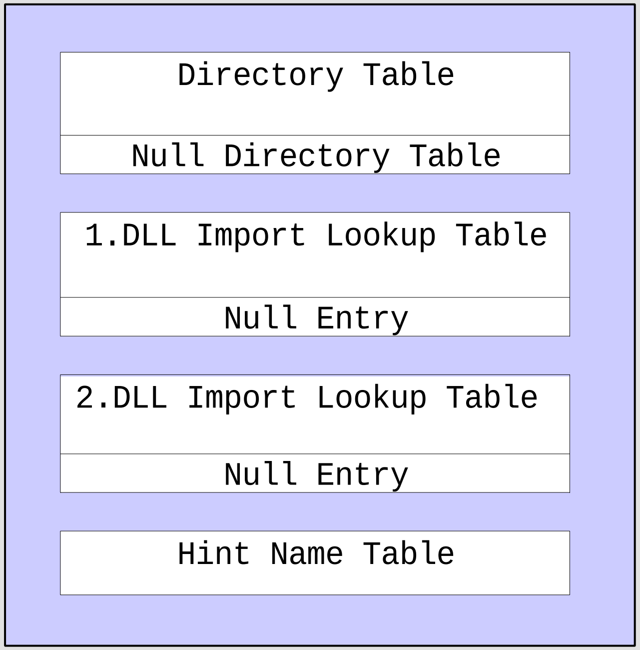
```
base_addr + *(uint32)(base_addr + 0x3c)
+ sizeof(COFF) + PCOFF->SizeOfOptionalHeader
```

    - 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



Directory Table

Null Directory Table

1.DLL Import Lookup Table

Null Entry

2.DLL Import Lookup Table

Null Entry

Hint Name Table

```
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;
```

# Portable Executable Format (PE/COFF)

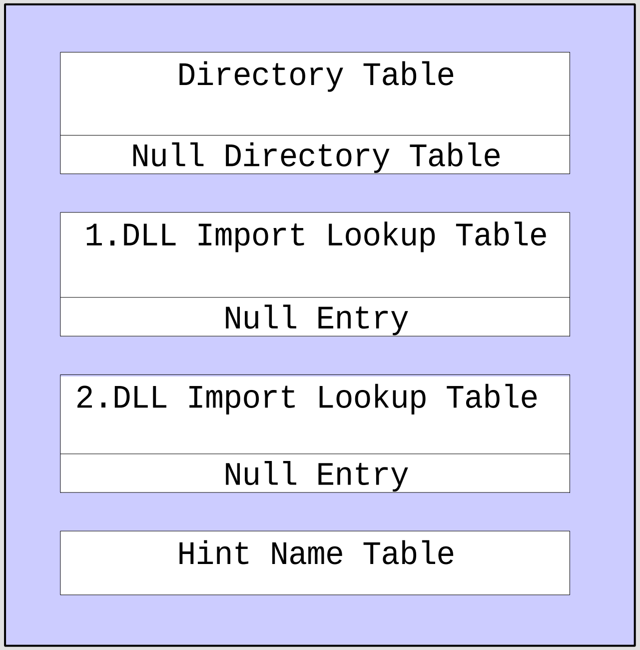
- PE Symbol Parsing
  - Locate and loop Import Directory Table
  - Get the pointer to the FirstThunk

```
IDD->FirstThunk
```

  - Loop Thunks for symbol import data

```
struct IMAGE_IMPORT_BY_NAME[]
```

```
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;
```



Directory Table

Null Directory Table

1.DLL Import Lookup Table

Null Entry

2.DLL Import Lookup Table

Null Entry

Hint Name Table

# Disassembly Analyzer

- The final component of a useful disassembler for reverse engineering is the disassembly analyzer.

- The analyzer builds a database of associations from the binary and can perform additional specialized disassembly analysis tasks.

- Disassembly analyzers attempt 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

# Disassembly Analysis

# Disassembly Analysis

- A wealth of information can be generated using very simple analysis logic.

- Data associations including function detection, static data references, string references, and execution branch references can be performed through simple opcode and operand parsing.

- Assembly hinting or commenting can aid the reverse-engineer by eliminating guesswork.
  - System call detection and argument labelling
  - Function call calling convention and argument detection
  - Assembly syntax hints

# Data Associations

- Function detection
  - Standard function detection is done by pattern matching for function prologues.
  - Prologues are generated during compilation and typically perform tasks including frame initialization and stack canary generation.

```
Standard function prologue

| 55            |   push %ebp             ; push old frame pointer
| 89 e5         |   mov  %esp, %ebp       ; store current stack pointer as new frame
```

```
Microsoft Visual Studio “hotfix” function prologue for system libraries and drivers

| 90            |   nop                   ; five nops make space for a long relative jmp
| 90            |   nop                   ;
| 90            |   nop                   ;
| 90            |   nop                   ;
| 90            |   nop                   ;
                                          ; Begin Function Prologue
| 8b ff         |   mov  %edi, %edi       ; 2-byte nop (space for short relative jmp)
| 55            |   push %ebp             ; push old frame pointer
| 89 e5         |   mov  %esp, %ebp       ; store current stack pointer as new frame
```

# Data Associations

- Function detection without prologue
  - 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                  ;
```

  - Symbolic function names are created for code locations that do not have a pre-defined symbol associated with them.

# Data Associations

- Cross Referencing
  - Disassembly analyzers create a database of cross-references which describe the relationships between code and data in the binary.
  - Cross-references are determined by examining immediate operand values or by tracing register exchanges to watch references to a known value.
  - The use of an instruction decoder core which implements operand permissions flags is required.
    - libdasm – available at nologin.org by jt
    - libdisasm – available at bastard.sf.net by _mammon
  - For each instruction, analyze the operands for internal relationships
    - Check for operand types: IMMEDIATE, MEMORY, REGISTER
    - Check operand permission flags
  - Cross-references are stored in data-structures for later use.

# Data Associations

- Code execution flow can be determined by detecting code branches which are indicated by the RET, IRET, INT, CALL and the various JMP opcodes for IA-32.

- Flow Control Instructions
  - Call
    - Indicates a new function
    - Needs to be checked against symbol tables when displaying disassembly
    - Pushes calling address before transferring execution control
  - Branch
    - Any opcode of the JMP variety
    - Indicates new code 'block'
    - Code blocks can be analyzed for functionality
    - Used for loops, signal handlers, etc
  - Return
    - Used to divert flow control by popping a pointer from the stack

# Data Associations

- Symbols, strings, and pointers within pre-initialized data sections in the binary are examples of data cross-references that can be determined through simple disassembly analysis.

```
00401050 |
........ |  ;;;;;;;;;;;;;;;;;;;;;;;;;;;
........ |  ;;; S U B R O U T I N E ;;;
........ |  ;;;;;;;;;;;;;;;;;;;;;;;;;;;
........ |  sub_00401050:                   ;  xrefs:  0x004010a7
........ |    push    %ebp                  ;
00401051 |    mov     %esp, %ebp            ;
00401053 |    sub     $0x8, %esp            ;
00401056 |    movl    $0x402000, (%esp)     ; "this string is a pre-initialized variable\n"
0040105d |    call    <printf>              ; imported shared library symbol
00401062 |    leave                         ;
00401063 |    ret                           ;
........
004010a7 |    call    <sub_00401050>        ; symbolic function names cross-referenced
004010ac |    mov     $0x0, %eax            ;
004010b1 |    leave                         ;
```

# Hinting

- Disassemblers should use a database of information regarding system calls and standard system library calls to aid in disassembly hinting.

- System Call hinting can help a reverse engineer determine what system services a function utilizes.
  - Syscall Numbers are stored in /usr/src/linux/include/asm/unistd.h
  - 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.

- Function argument types and high level data-structures can be parsed from header files.
  - Every platform has a set of default libraries and headers.
  - The more the disassembler knows about variable types, the better it can understand how the data is being used.

# Hinting

- 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

# Disassembly Analysis

- The features described in this section should be standard fare.
- IDA Pro, HTE, the bastard, and Codis are currently the only disassemblers available which implement most of the features.
- Required development time: 2 - 3 weeks

**Codis Demo**

# Advanced Disassembly Analysis

# Advanced Disassembly Analysis

- The flexibility offered by DataRescue's IDA Pro SDK has allowed for recent advancements in disassembly analysis capabilities.

- IDA Pro plug-ins have access to the program's internal database which allows for rapid development of concept ideas.
  - Path Analysis
    - Peter Silberman's loop detection plugin
  - Data Analysis
    - idastruct - data structure enumeration

# Path Analysis

- Path analyzers recursively follow execution flow to build a control flow graph.

- 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

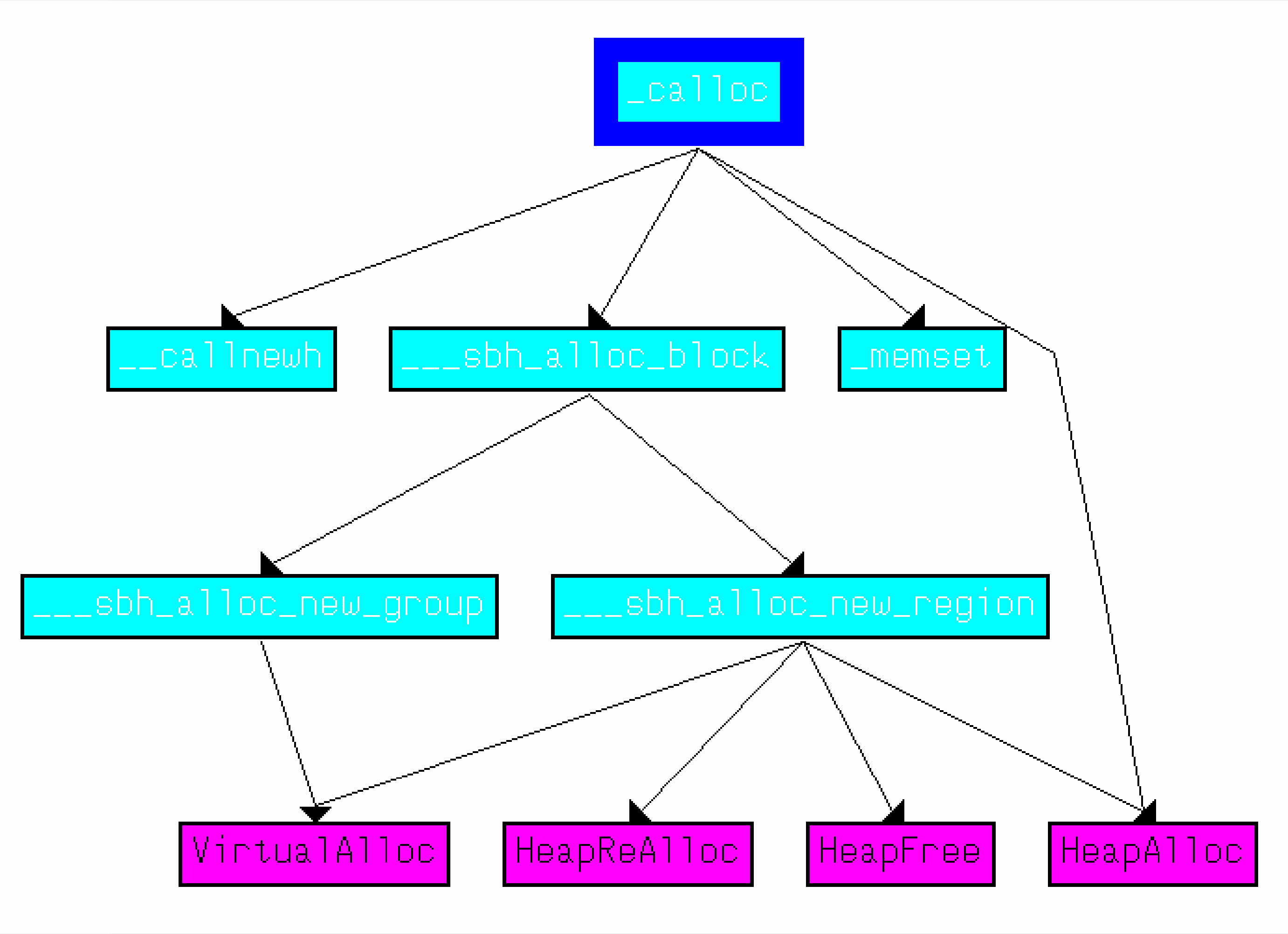
# Path Analysis

- Hand written assembly code can cause disassemblers to generate code listings that are completely incorrect:

```
(gdb) disas loc
Dump of assembler code for function loc:
0x0040107a <loc+0>:     pushw  $0xfeeb
0x0040107e <loc+4>:     jmp    0x40107c <loc+2>
0x00401080 <loc+6>:     pushl  0xaabbccdd
0x00401086 <loc+12>:    mov    $0x0,%eax
0x0040108b <loc+17>:    leave
0x0040108c <loc+18>:    ret

------------------------------------------------

Breakpoint 1, 0x0040107a in loc ()
1: x/i $pc  0x40107a <loc>:     pushw  $0xfeeb
(gdb) si
0x0040107e in loc ()
1: x/i $pc  0x40107e <loc+4>:   jmp    0x40107c <loc+2>
(gdb) si
0x0040107c in loc ()
1: x/i $pc  0x40107c <loc+2>:   jmp    0x40107c <loc+2>
…
```

# Path Analysis

- Once a control flow graph has been built programmatically, it can easily be represented using data visualization software.



# Loop Detection

- Loop detection is an advanced application of control flow analysis.

- Loop detection can be applied to recognize program structure as well as specific types of vulnerabilities.

- Recognizing loops can aid other disassembly analysis tasks and eliminate heavy analysis of code multiple times.

- Example: Peter Silberman’s Loop Detection Plugin
  - Designed to help reverse-engineers locate code loops that may lead to exploitable scenarios.

# Loop Detection

- Reducible loops have one entry point and can be reduced to a Natural Loop.
- Natural Loop structure is found by determining node dominance in the control flow graph.
- If node C is unreachable other than through node B, then B dominates node C.

In this diagram, there is a small loop between B and D.

The *Natural Loop* can be determined by locating the path between the two nodes that are under dominance of B.

The secondary loop between B and D can be ignored when determining the *Natural Loop*

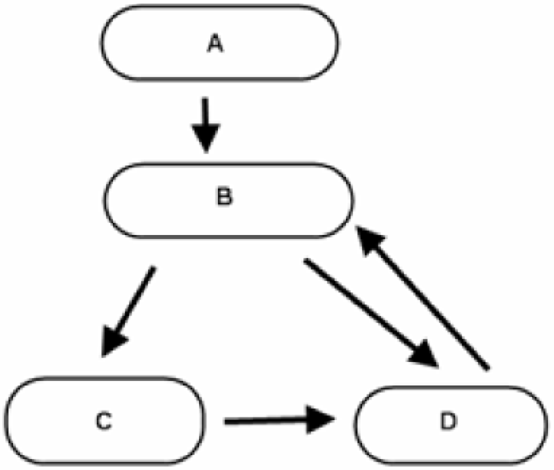


Figure 3.1: An example of a reducible loop

# Loop Detection

- Traditional loop detection algorithms are known to have trouble detecting loops with more than one potential entry point (non-reducible loops).
- Using IDA’s powerful cross-referencing and flow control graphing algorithms, Peter has developed a method for identifying irreducible loops.
- Peter’s work can be found on www.uninformed.org

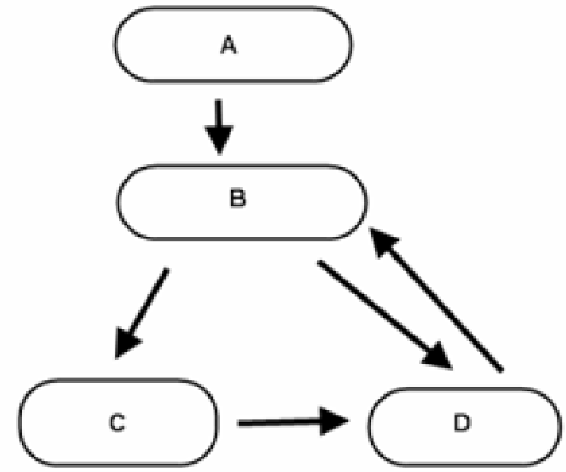


Figure 3.1: An example of a reducible loop

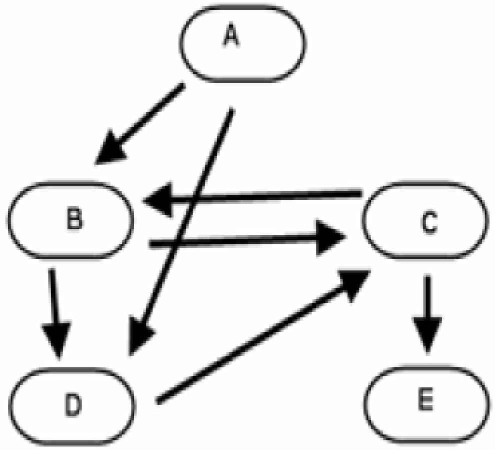
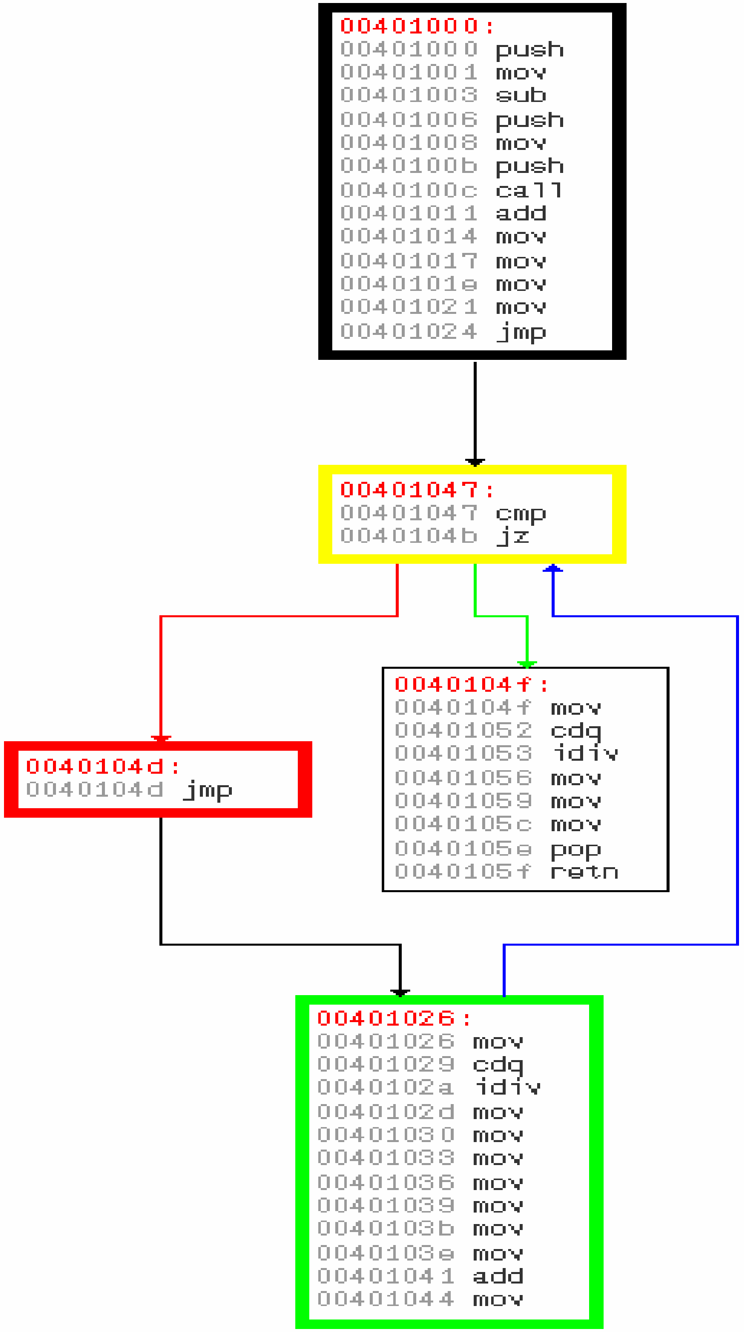


Figure 3.2: An example of an irreducible loop

# Loop Detection



```
00401000:
00401000 push
00401001 mov
00401003 sub
00401006 push
00401008 mov
0040100b push
0040100c call
00401011 add
00401014 mov
00401017 mov
0040101e mov
00401021 mov
00401024 jmp
```

```
00401047:
00401047 cmp
0040104b jz
```

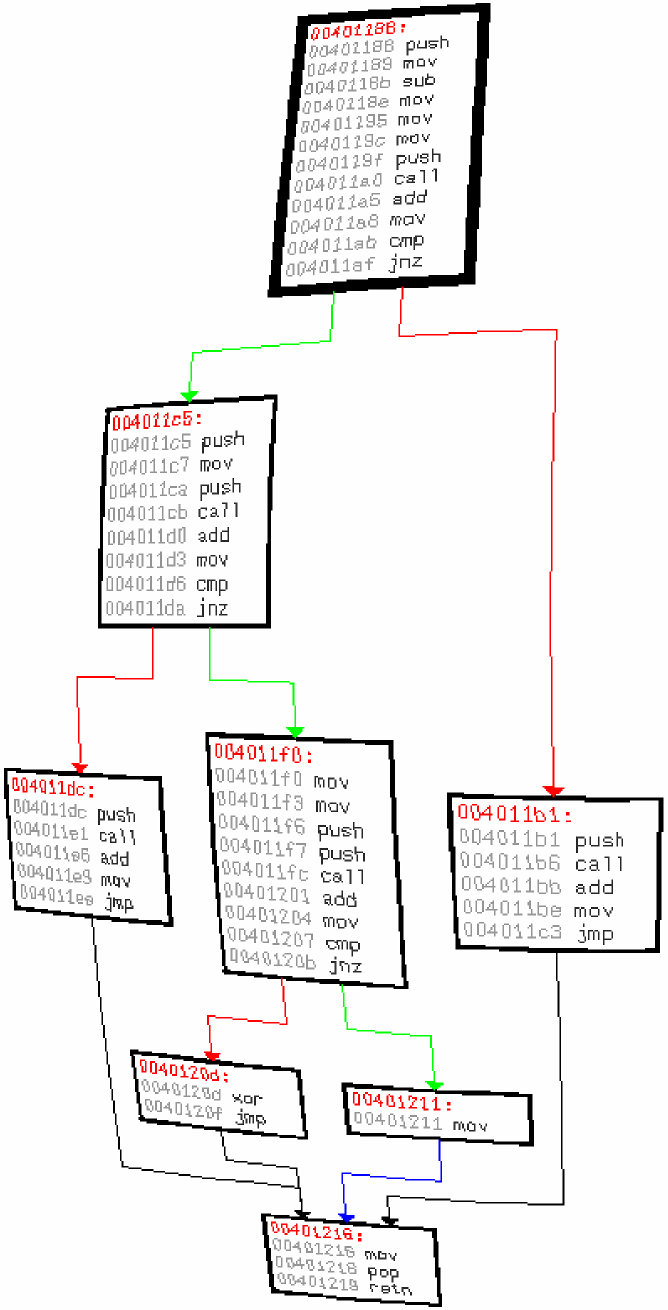
```
0040104f:
0040104f mov
00401052 cdq
00401053 idiv
00401056 mov
00401059 mov
0040105c mov
0040105e pop
0040105f retn
```

```
0040104d:
0040104d jmp
```

```
00401026:
00401026 mov
00401029 cdq
0040102a idiv
0040102d mov
00401030 mov
00401033 mov
00401036 mov
00401039 mov
0040103b mov
0040103e mov
00401041 add
00401044 mov
```

<- Loop

No Loop ->



```
00401186:
00401186 push
00401189 mov
0040118b sub
0040118e mov
00401195 mov
0040119c mov
0040119f push
004011a0 call
004011a5 add
004011a8 mov
004011ab cmp
004011af jnz
```

```
004011c5:
004011c5 push
004011c7 mov
004011ca push
004011cb call
004011d0 add
004011d3 mov
004011d6 cmp
004011da jnz
```

```
004011dc:
004011dc push
004011e1 call
004011e6 add
004011e9 mov
004011ee jmp
```

```
004011f0:
004011f0 mov
004011f3 mov
004011f6 push
004011f7 push
004011fc call
00401201 add
00401204 mov
00401207 cmp
0040120b jnz
```

```
004011b1:
004011b1 push
004011b6 call
004011bb add
004011be mov
004011c3 jmp
```

```
0040120d:
0040120d xor
0040120f jmp
```

```
00401211:
00401211 mov
```

```
00401216:
00401216 mov
00401218 pop
00401219 retn
```

# Data Analysis

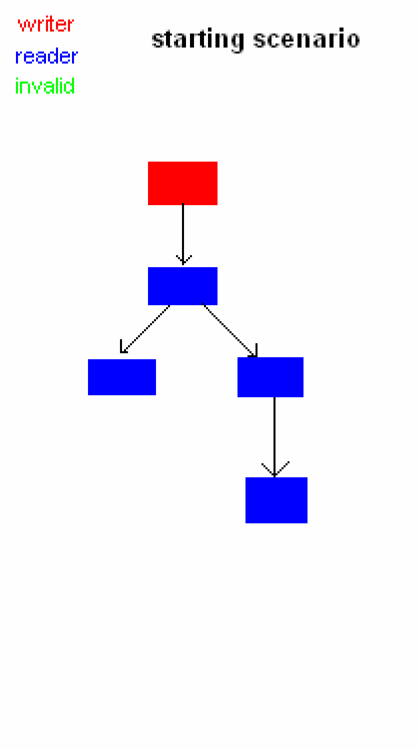
- Unlike code paths, analyzing data relationships is a non-trivial exercise.
- Data references are occasionally supplied as immediate values, but are more often passed around in registers to perform operations.
- There are numerous obstacles to overcome when tracing assembly for the purpose of data reference tracking – it has yet to be implemented successfully.
- To follow data paths, a variable tracing algorithm must be developed... or does it?

# Data Analysis

- Variable Tracing

```
// init trace
add_xref(ea, dst);

// simplified variable tracing loop
while(ea = ea.next)
{
   while(op = operand.next)
   {
      mask = SIZE_MASKS[opsize];
      switch(op->type)
      {
      case o_imm:
         val = op->addr & mask;
         break;
      case o_displ:
         val = (registers[op->reg] + op->addr)& mask;
         break;
      case o_phrase:
         val = registers[op->phrase] & mask;
         break;
    }
}

if(search_itrace_list(val))
   remove_xref(ea, dst);
else if search_itrace_list(src)
   add_xref(ea, dst);
```

```
unsigned long registers[8];
unsigned long eip;
unsigned long eflags;

typedef struct _itrace {
        struct _itrace *next, *prev;
        ea_t addr; // address of reference
        ea_t xref; // address referenced
        unsigned char  reftype; // RWX
} itrace_t;
```

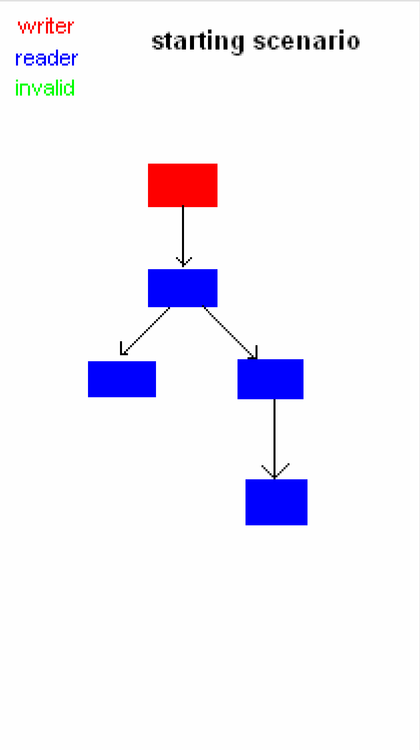


# Data Analysis

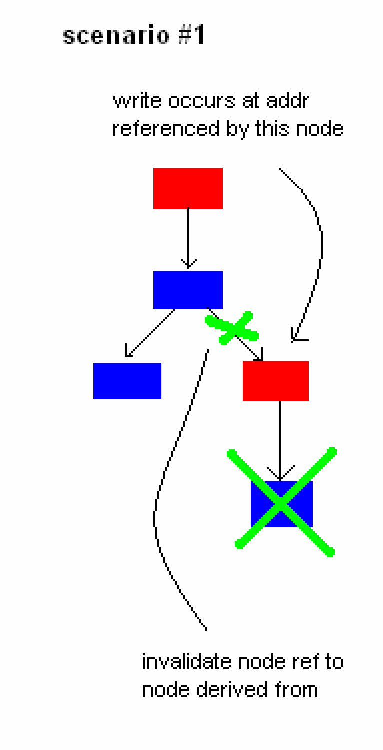
- Variable Tracing

```
// init trace
add_xref(ea, dst);

// simplified variable tracing loop
while(ea = ea.next)
{
   while(op = operand.next)
   {
      mask = SIZE_MASKS[opsize];
      switch(op->type)
      {
      case o_imm:
         val = op->addr & mask;
         break;
      case o_displ:
         val = (registers[op->reg] + op->addr)& mask;
         break;
      case o_phrase:
         val = registers[op->phrase] & mask;
         break;
    }
}

if(search_itrace_list(val))
   remove_xref(ea, dst);
else if search_itrace_list(src)
   add_xref(ea, dst);
```

```
unsigned long registers[8];
unsigned long eip;
unsigned long eflags;

typedef struct _itrace {
        struct _itrace *next, *prev;
        ea_t addr; // address of reference
        ea_t xref; // address referenced
        unsigned char  reftype; // RWX
} itrace_t;
```



starting scenario



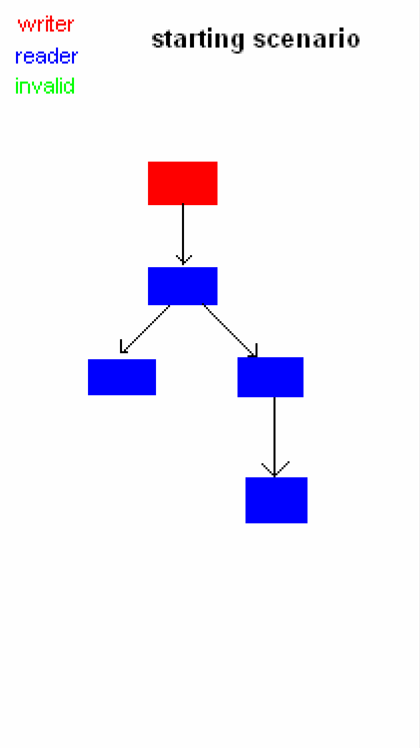
scenario #1

# Data Analysis

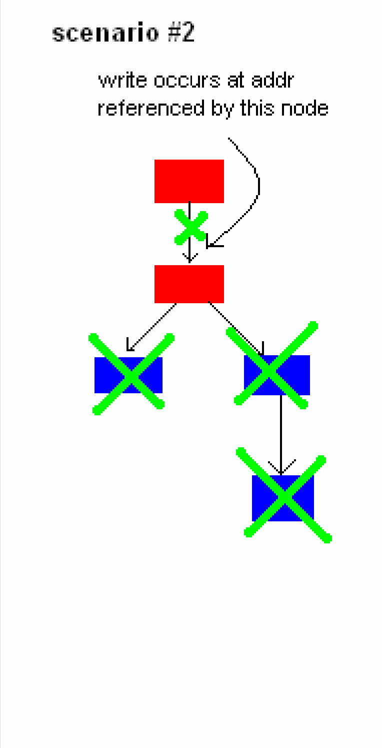
- Variable Tracing

```
// init trace
add_xref(ea, dst);

// simplified variable tracing loop
while(ea = ea.next)
{
   while(op = op.next)
   {
      mask = SIZE_MASKS[opsize];
      switch(op->type)
      {
      case o_imm:
         val = op->addr & mask;
         break;
      case o_displ:
         val = (registers[op->reg] + op->addr)& mask;
         break;
      case o_phrase:
         val = registers[op->phrase] & mask;
         break;
    }
}

if(search_itrace_list(val))
   remove_xref(ea, dst);
else if search_itrace_list(src)
   add_xref(ea, dst);
```

```
unsigned long registers[8];
unsigned long eip;
unsigned long eflags;

typedef struct _itrace {
        struct _itrace *next, *prev;
        ea_t addr; // address of reference
        ea_t xref; // address referenced
        unsigned char  reftype; // RWX
} itrace_t;
```



starting scenario



scenario #2

write occurs at addr referenced by this node

# Data Analysis

- *Combinatorial Explosion*
  - Occurs when a huge number of possible combinations are created by increasing the number of entities which can be combined--forcing us to consider a constrained set of possibilities when we consider related problems.

- Variable tracing is susceptible to combinatorial explosion and infinite recursion if bounds are not set on the depth of the search.

- In theory static data reference tracing is possible but has yet to be successfully implemented beyond proof of concept.

# Emulation

- CPU emulation can be used as a powerful resource when analyzing static code.

- CPU Emulation involves the execution of instructions in a virtual CPU.

- Virtual CPUs emulate the core components of a hardware CPU in software.
  - Instruction decoding/evaluation
  - Registers
  - Memory

- Emulation is “safe”
  - depending on implementation of course

# ida-x86emu

- ida-x86emu is an opensource emulator written by Chris Eagle and Jeremey Cooper
  - Emulator codebase can be easily hooked for special analysis purposes.
  - Undergoing development
  - Some features are missing but code is easily hackable
  - Use the CVS version!
- Evaluates complex instruction sequences
- Emulates dynamic memory allocator functionality
- Can hook PE imports and load required libraries
  - Sometimes has some hiccups – currently looking into this

# idastruct

- idastruct is data structure reference tracing code built on top of ida-x86emu.

- Arbitrary bounds within the emulated memory space can be traced using simple logic.

- As operands are evaluated for each instruction, a check is made to determine if that operand is referencing memory that is being traced.

- IDA database is updated with structure information and member data as references are detected and types are applied to the reference.

# idastruct

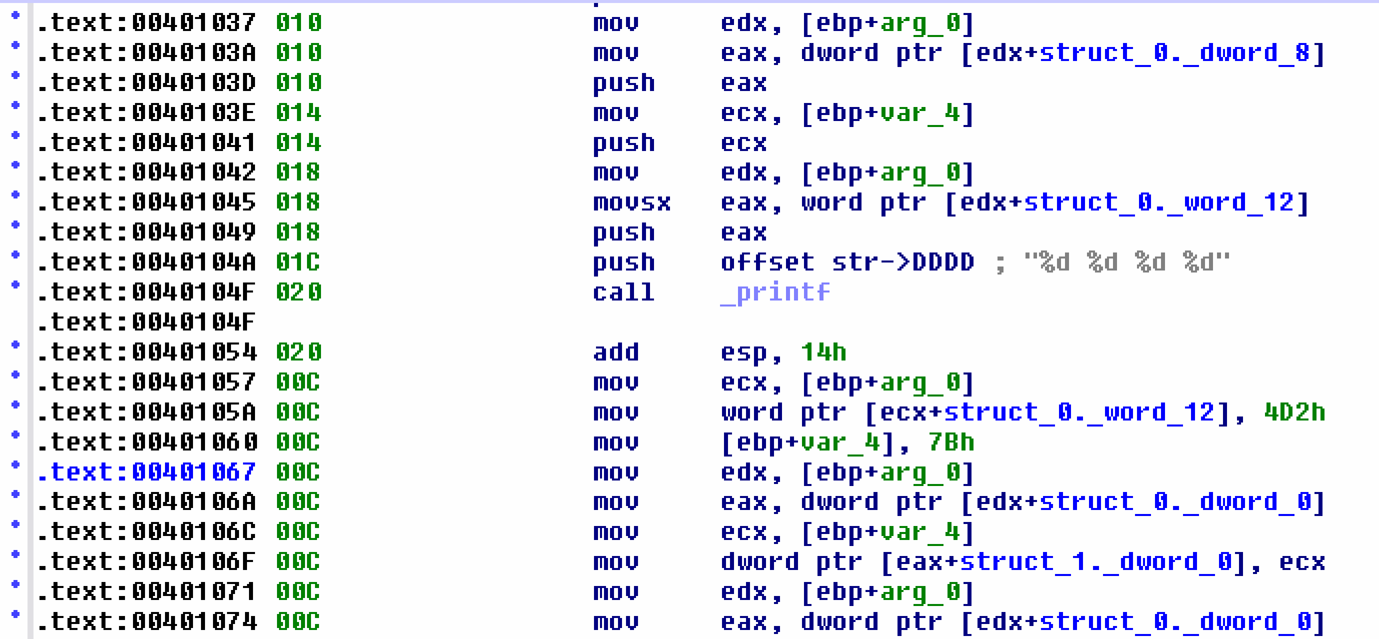
- Structure reference tracing

```
void struct_trace(ea_t addr)
{
   strace_t *trace;
   ua_ana0(addr);

   for(int opnum = 0; cmd.Operands[opnum].type != o_void; opnum++)
   {
      op_t *op = &cmd.Operands[opnum];
      …
      // evaluate operand value
      …
      for(trace = strace; trace; trace = trace->next)
      {
         // determine if operand value points within trace bounds
         if(val >= trace->base && val <= trace->base + trace->size)
         {
            …
            struc_t *sptr = trace->sptr;
            member_t *mptr = get_member(sptr, val - trace->base);

            if(!mptr)
            {
               char *mtype;
               switch(get_dtyp_size(op->dtyp))
               …
               // assign a name to the new member that indicates type size
               …
```

# idastruct

- Structure reference tracing

```
void struct_trace(ea_t addr)
{
        …
        // create ida structure member
        if(struct_member_add(sptr, name, val - trace->base, 0, NULL,
                             get_dtyp_size(op->dtyp)) < 0)
        {
            trace = trace->next;
            continue;
        }
        mptr = get_member(sptr, val - trace->base);
    }
    …
    // update member reference
    tid_t path[2];
    path[0] = sptr->id;
    path[1] = mptr->id;
    op_stroff(addr, opnum,
              path, 2, 0);
}
```



```
.text:00401037 010                 mov     edx, [ebp+arg_0]
.text:0040103A 010                 mov     eax, dword ptr [edx+struct_0._dword_8]
.text:0040103D 010                 push    eax
.text:0040103E 014                 mov     ecx, [ebp+var_4]
.text:00401041 014                 push    ecx
.text:00401042 018                 mov     edx, [ebp+arg_0]
.text:00401045 018                 movsx   eax, word ptr [edx+struct_0._word_12]
.text:00401049 018                 push    eax
.text:0040104A 01C                 push    offset str->DDDD ; "%d %d %d %d"
.text:0040104F 020                 call    _printf
.text:0040104F
.text:00401054 020                 add     esp, 14h
.text:00401057 00C                 mov     ecx, [ebp+arg_0]
.text:0040105A 00C                 mov     word ptr [ecx+struct_0._word_12], 4D2h
.text:00401060 00C                 mov     [ebp+var_4], 7Bh
.text:00401067 00C                 mov     edx, [ebp+arg_0]
.text:0040106A 00C                 mov     eax, dword ptr [edx+struct_0._dword_0]
.text:0040106C 00C                 mov     ecx, [ebp+var_4]
.text:0040106F 00C                 mov     dword ptr [eax+struct_1._dword_0], ecx
.text:00401071 00C                 mov     edx, [ebp+arg_0]
.text:00401074 00C                 mov     eax, dword ptr [edx+struct_0._dword_0]
```

# Conclusion

- The ability to identify arbitrary structures via binary analysis should speed software reversing in all areas.

- Directly applies to vulnerability discovery through automation of fuzz template generation.

- Further analysis may be performed on the structure relationships within execution paths to tie complete structure hierarchies together.

# Questions?