Automated Debugging and Process Analysis

Interz0ne 4
March 2005

Richard Johnson | rjohnson@idefense.com

Introduction

Who am I?

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

What is iDEFENSE?

What is the purpose of this talk?

- Explore the current and potential uses of debug tools
- Discover the methods by which one can develop their own debugging tools
- Show off my prototype trace tool – dltrace
- Inspire the audience to look beyond the traditional use of tracers and debuggers

Agenda

Process Analysis 101

The Anatomy of a Debug Tool

dltrace – Prototyping a Portable Shared Library Call Tracer

Demo & Conclusion

Process Analysis 101

Debuggers vs Tracers

- Debug tools are designed to allow for the control and inspection of another process's execution
  - Register state
  - Virtual memory
  - Signals

Debuggers vs Tracers

Debuggers

- Interactive
- Powerful and flexible
- Dumb
- Sometimes scriptable / programmable

Tracers

- Non-interactive
- Typically single-purpose

Current Uses of Tracing

System Call Tracing

- System Calls (syscalls) provide the interface between userspace and the kernel.
- Effective for narrowing the cause of a program crash
- *strace, truss, tusc, par, ktrace (kernel mode)*

Shared Library Call Tracing

- Allows higher level program execution analysis
- Helpful when analyzing the function of large areas of code
- *ltrace* (Linux only), *truss/sotruss* (SUN only)

Performance Profiling

- Locate areas of inefficient code
- Locate code ideal for optimization due to heavy utilization

Potential Uses of Tracing

Deep process state analysis

- Complex state machines keep track of program execution
- Real-time, actionable process analysis

Detection and recovery from faults

- Process monitor intercepts signals and corrects faults
- Injection of fault handler code into traced process

Software vulnerability analysis

- Many Windows debug tools have a community which develops scripts or plugins for software analysis tools such as windbg, IDA Pro and Ollydbg

Dtrace: a kernel mode approach

- SUN released with Solaris 10
- Kernel Resident
- Works towards hybrid approach
  - Scriptable
  - Flexible

- Disadvantages
  - Not portable
  - Difficult to load libraries into process space or interact with run-time linker

The Anatomy of a Debug Tool

The Anatomy of A Debug Tool

- Target binfmt handling
  - Binary format structures
    - Headers
    - Dynamic Table
    - Symbol Tables
  - Linking and Loading
    - Reference: Linkers and Loaders - Devine
- Process analysis and control interface
  - Kernel Exported
  - The ptrace interface
  - The proc virtual file system
- Event Handling

ELF Binfmt

What is 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 useful information for debugging including symbol tables, string tables, library dependencies, and debugging information

ELF Binfmt

A program written in a high level language must be compiled and linked before it becomes an executable ELF binary

ELF Object Types

- Relocatable Objects
- Executable Objects
- Shared Objects

ELF Binfmt

Relocatable Objects

- Header info
  - ELF Header
    - Details how to access sections within the object
  - Section Header Table
    - Details how to access various sections in the file
- 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

May contain unresolved references to symbols in other objects or libraries

ELF Dynamic Linking

ELF Object Linking

- The linking process involves:
  - Merging of object code into *Executable* or *Shared Objects*
  - Resolving symbol references across objects
  - Replacing labels with resolved addresses
  - Creation of the Program Header Table

The program header table

- Gives the Linux kernel's ELF loader information about how to create a process image for the binary.
- Segments define the separation of memory when mapping the file into the process's address space and contain one or more sections.

Symbol Resolution

Symbols are resolved by enumerating section tables until a .dynsym or .symtab section is found:

```
for (shdr = melf_sectionGetEnum(melf); shdr;
     shdr = melf_sectionEnumNext(melf, shdr))
{
     if ((shdr->spec.section.sh_type == SHT_DYNSYM) ||
         (shdr->spec.section.sh_type == SHT_SYMTAB))
     {
          enum_symtab(melf, shdr);
     }
}
```

Symbol Resolution

Enumerate the symbol table with a Elf32_Sym pointer to determine symbol name and load address:

```
void enum_symtab(MELF *melf, ELF_SPEC_HEADER *shdr)
{
     Elf32_Sym *sym;
     unsigned long index = 0;

     while ((sym = melf_symbolTableEnum(melf, shdr, index++)))
          printf("%s\n", melf_symbolGetName(melf, shdr, sym));
}
```

Process Analysis and Control Interface

Most modern operating systems expose a debugging interface that allows a user process to monitor the execution of other processes.

For UNIX, the interface is exposed by the kernel through a system call or virtual device which provides the following functionality:

- Process Control (attaching, stepping, etc)
- Register Access
- Memory Access

ptrace Debug Interface

The ptrace debug interface

Exposed by the kernel via a system call

```
#include <sys/ptrace.h>
long ptrace(enum __ptrace_request request, pid_t pid, void
  *addr, void *data)
```

Supported by

- Linux
- FreeBSD
- Solaris
- HP-UX

ptrace Debug Interface

Process Control

PTRACE_ATTACH	Attach to the specified process id
PTRACE_SINGLESTEP	Execute one instruction and return to debugger
PTRACE_CONT	Continue execution. Will not return to debugger until a signal is received.

Data Access

PTRACE_GETREGS	Copy array of general purpose registers to data
PTRACE_GETFREGS	Copy array of floating point registers to data
PTRACE_PEEKDATA	Read a word of memory from addr into data

Platform Dependant

PTRACE_PEEKUSR	Read from process's USER area (platform dependent)
PTRACE_SYSCALL	Execute until next system call
PTRACE_TRACEME	If executed before exec() process will return control to the debugger once entry point has been reached

proc Debug Interface

The proc debug interface

- Exposed by the kernel via the proc virtual file system as device files.

- Supported by
  - Linux
  - BSD
  - Solaris

proc Debug Interface

Process control is accomplished by writing commands as strings to the /proc/<pid>/ctl file as shown:

attach	stops the target process and allows the sending process to become the debug control process
detach	continue execution of the target process and remove it from control by the debug process
run	continue running the target process until a signal is delivered, a breakpoint is hit, or the target process exits.
step	single step the target process, with no signal delivery.
wait	wait for the target process to come to a steady state ready for debugging. The target process must be in this state before any of the other commands are allowed

Signals may also be sent by writing the name of the signal lowercase and without the SIG prefix.

- dltrace -
Prototyping a Shared Library Call Tracer

Target Acquisition

Process initialization

- Execute specified binary if necessary
- Sending a SIGKILL after the fork() and before the execve() call will allow the debugger to attach before rtld has been executed

Attaching to a process

- Utilize the api call exposed by the debug interface to notify the kernel your process id has become the debugger of the traced process

Load Target Binary

Load symbols

- Load the .symtab if present
- Load the .dynsym

Load interpreter

- The interpreter is a typically a shared object and is loaded in a similar manner to shared libraries, however special note should be taken of interpreter symbols

Binary Loading

Load shared libraries

- Iterate the dynamic section of the target binary for DT_NEEDED flags
- Search library paths for required libraries
  - /lib:/usr/lib
  - /etc/ld.so.conf
  - LD_LIBRARY_PATH
- Store hash of ELF file
- Enumerate symbols
- Enumerate dynamic section

Trace Initialization

Shared library trace initialization

- Allow runtime linker to map library dependancies into memory
- Walk target process's address space by pages, looking for ELF signature
- Compute hash of ELF file in memory
- Iterate loaded library list for matching hash
  - Iterate library's linked list of symbols and add library load offset to symbols
  - Insert into splay tree
  - Insert breakpoint

Trace Initialization

System call trace initialization

- Enumerate loadable segments in each binary
- Disassemble each segment to locate system call interrupt or trap
- Insert breakpoint

Trace Execution

The debug program will gain control of the traced process when any breakpoint is reached or a signal is received for that process by the kernel

Handle events

- Signals
- System calls
- Shared library calls
- Shared library returns

Handling Events

Shared Library Calls

- Lookup current EIP register value in tree of shared library call addresses
- If call parameters have not been determined, analyze parent function
  - Analyze function's assembly code to determine argc
    - push
    - calls
    - jmps
    - mov's where the destination operand is an offset from esp
  - Store arg count
- Analyze arguments to determine type
  - If value is not within mapped memory space, display as integer
  - Dereference pointers and check for string values
  - Display pointer if not determined to be integer or string
    - Probably a struct pointer

Handling Events

Shared Library Call Returns

- Add a breakpoint at the calling address (return address) whenever a shared library call is executed
- Add to callstack

- On event, check address of the breakpoint on the top of the callstack
- Return values are typically stored in eax on x86 processors

Demo & Conclusion

Conclusion

The Linux/UNIX world is still lacking an adequate set of debugging tools

Cross-platform development is easy due to similar debug interfaces and necessary due to the lack of appropriate tools across the board

A hybrid of debugger and trace tool should increase both the power and speed of automated process analysis

Questions?