Lecture 2
Program Analysis

Computer and Network Security
October 7, 2019

Computer Science and Engineering Department

Program Analysis

- automatic analysis of programs
- property verification
- optimization (performance) or correctness
- static analysis or dynamic analysis

Program Model

- automaton
- control flow graph (CFG) (set of states and transitions)
- coverage: how much of the CFG can the analysis cover to ensure property validation

- do not execute or execute the program
- static analysis on source code or on binary program (executable)
- dynamic analysis on resource usage and behavior (process)
- symbolic execution is static analysis
- fuzzing is dynamic analysis
- static analysis: broad, may go into path explosion
- dynamic analysis: depth, may miss certain cases

Source Code vs Executable

- extensive analysis on source code but ...
- we don't know what the compiler / linker does to it, what optimizations happen, how it links to other components
- it may not be available
- we focus most on static binary analysis

Challenges of Static Binary Analysis

- more difficult to understand: requires reverse engineering
- may be subject to obfuscation, encryption, packing
- typically doubled by dynamic analysis

Process as a Goal

- provide functionality
- dynamic / run time
- allocate and use memory and other resources

Steps from Source Code to Process

1. compile and assemble source code into object files
2. link object files into executable
3. load executable (disk image file) into process (memory + CPU)

Object File

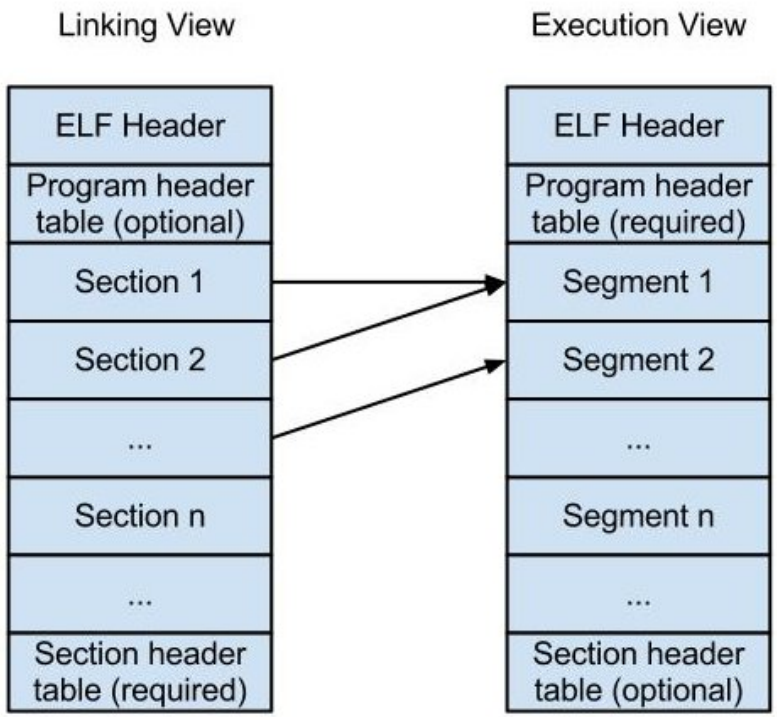
- binary files
- headers and binary code
- may be disassembled
- data and code
- sections

Library Files

- archive/collection of object files
- modularity
- static-linking and dynamic linking libraries
  - linking happens at link time
  - linking happens at load time

Executable Files

- binary files
- similar to object files, consist of object code
- may be disassembled
- created from object files
- static and dynamic executables
  - static: all object code is part of the executable
  - dynamic: library stubs to library functions



http://www.roman10.net/2012/11/28/an-intro-to-elf-file-formatpart-1-file-types-and-dual-views/

Object File Format

- format of a file that contains object code: object file, executable files, dynamic-linking library files
- headers, sections
- data and code
- may be disassembled
- PE (Portable Executable) on Windows
- COFF (Common Object File Format) on Unix
- ELF (Executable and Linking Format) on Linux

Common Information in Executabile Files

- entry point
- program addresses (section addresses)
- section sizes
- symbols (names and addresses)
- permissions

ELF Format

- header
- program headers
- sections
- segments
- symbols
- `readelf`, `objdump`, `nm`

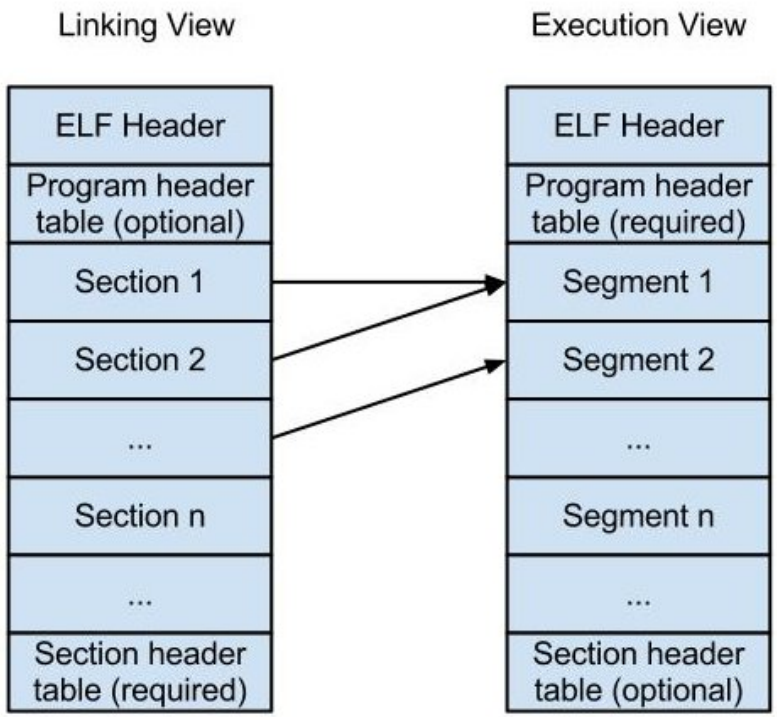
Sections

- storing data or code
- `readelf -S program`
- `.text`, `.data`, `.bss`
- `.symtab`, `.strtab`

Sections vs. Segments

- segments contain 0 ore more sections
- sections are used by linker, some sections may be ditched at runtime
- segments are used by the operating system (loaded into memory)

View of ELF File



http://www.roman10.net/2012/11/28/an-intro-to-elf-file-formatpart-1-file-types-and-dual-views/

Symbols

- `readelf -s program`
- `.dynsym` and `.symtab`
- name, value, type, bind, size

Debugging Symbols

- Map Assembly instructions to variable, function or line in the source code
- Help mapping stack values with function parameters
- Optimize data flow analysis
- Optimize static and dynamic analysis
- On Linux, symbol table is embedded in the ELF file. PE files use an external symbols file

Stripping

- Removing symbol table from program executable
- Complicates reverse engineering
- Less space used by original binary

Overview of Linking

- All object files are linked together to produce an executable file
- Input: Object files, static libraries, dynamic libraries
- Output: Executable image
- The linker resolved external references from each object file

Using `ld`

- Command used in the last compiling phase
- Libraries are specified using `-l` option
- PIE option enables ASLR support

Static Linking

- Linker copies library routines directly into executables image
- Executable is more portable because all data needed to execute resides in the file
- Faster execution because imports are not resolved at runtime
- Uses more space

Tools of Trade

- building machine code files
- inspecting machine code files
- disassembling machine code files

Building Executables

- gcc, gas, nasm, ar, ld

ELF Inspection

- `strings`
- `xxd`
- `readelf`
- `nm`

- **IDA**
- objdump
- radare2

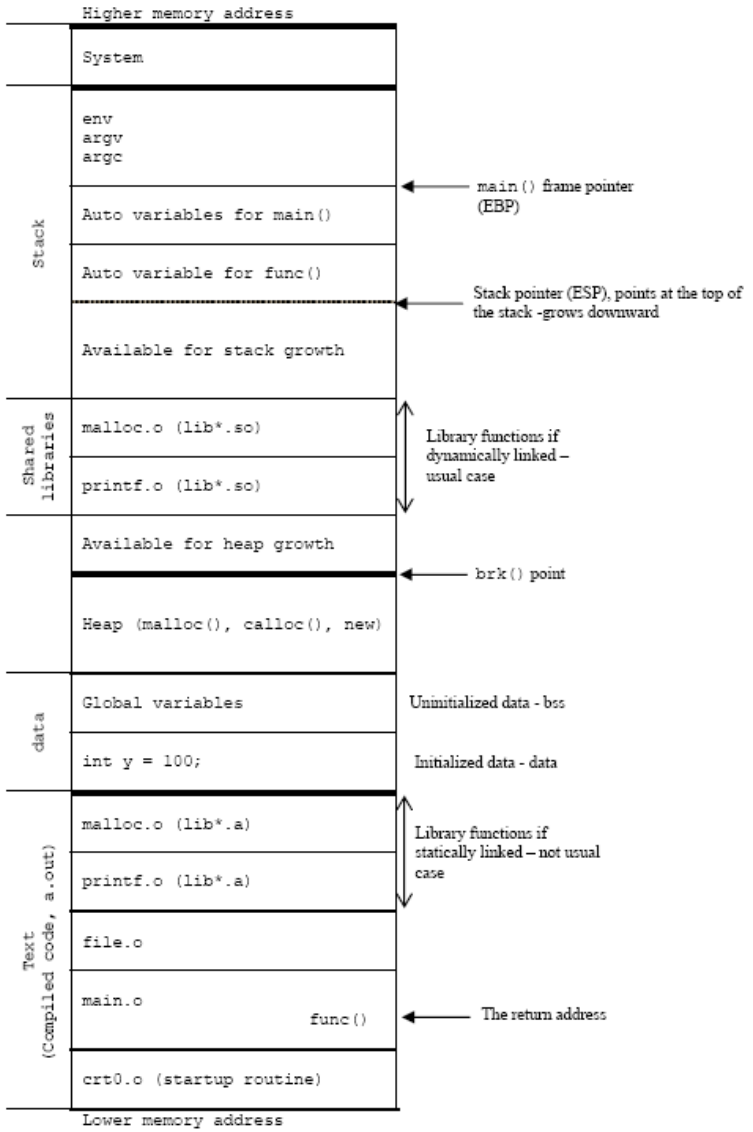
Not for Static Analysis

- `pmap`
- `lsof`
- `ltrace`
- `strace`
- GDB

- starts from executable files
- investigate processes
- requires process to run
- runtime analysis
- blackbox analysis

- unit of work in the operating system
- virtual memory address space, threads, resources
- isolated from each other
- at **load time** the executable gives birth to a process

Process Memory Layout



http://www.tenouk.com/Bufferoverflowc/Bufferoverflow1_files/image022.png

Interesting Process Information

- the process memory map (virtual memory areas)
- memory addresses: code, variables
- memory region access rights
- machine code (to be disassembled)
- process state: registers, (call) stack, code

- get output for input (blackbox)
- glimpse into the internals
- monitor/inspect resource usage
- debug execution and test attacks (step by step)

What Do We Investigate?

- code: system calls, library calls, function calls, step-by-step code
- state: thread information, process maps, open files, resources
- data: registers, variables, raw memory data

- function call tracing
- disassembling
- step by step instructions
- look into code where required in the process virtual address space

- variables: global (data) and local (stack)
- runtime metadata: return addresses, function arguments, command line arguments, GOT and PLT (to be discussed later)
- registers
- raw memory data: heap, stack, random address

Inspecting State

- process memory map
- thread state
- open file descriptors

Types of Tools

- blackbox inspection: function call tracers (strace, ltrace, dtrace/dtruss), fuzzers
- profilers: most often for performance: perf, callgrind, vTune
- debugging: GDB, LLDB, valgrind

Fuzzing

- generate “random” input and detect program flaws
- program is run
- smart fuzzer try to direct
- AFL, libfuzzer

- `strace ./a.out`
- `strace -e read,write ./a.out`
- `strace -e file ./a.out`
- `strace -e file -f ./a.out`
- `strace -e file -s 512 -f ./a.out`
- similar options for `ltrace`

lsof/pmap

- PID as argument
- `lsof -p 12345`
- `pmap 12345`

- default profiler on Linux
- sampling profiler, doesn't instrument the code
- uses events sampling
- `perf stat -e cache-misses -a ./mem-walk`
- `sudo perf list`
- some actions and events may require privileged access

GDB

- default debugger on GNU/Linux distributions
- command line; there are some GUI front-ends
- incorporated in Linux-based IDEs
- debugging, dynamic analysis / process investigation
- `gdb ./a.out`
- `gdb -q ./a.out`

LLDB

- LLVM Debugger
- used on Mac OS X
- similar features to GDB
- command line; most commands are equivalent to GDB
- `http://lldb.llvm.org/lldb-gdb.html`

- useful for debugging embedded devices
- JTAG: Joint Test Action Group
  - uses dedicated debug port
- Lauterbach Trace32: in circuit debugger (device using JTAG)

- not just for debugging
- follow what a process does (step instructions)
- inspect data (memory, registers)

- process state inspection
- register inspection
- (machine) code inspection
- memory inspection
- memory alteration
- function call tracing

GDB Basics

- starting a process
- stepping instructions
- breakpoints
- disassemble
- show registers
- display data
- trace function calls
- alter data

Starting a Process

- `run`
- `run < input file`
- `run arg1 arg2 arg3`
- `set args arg1 arg2 arg3` and then issue `run`
- `start`: breakpoint at main / starting point

Stepping Instructions

- `si` and `ni`
- `ni` doesn't go into nested functions
- very useful for understanding programs and validating attacks

Breakpoints

- `b symbol-name`
- `b *address`: `b *0x80123456`
- `continue`: continue until the next breakpoint
- `help breakpoints`

Disassembling

- during runtime
- `disass symbol-name`: `disass printf`
- `help disassemble`

Displaying Data

- show memory data or registers
- `info registers`
- `p $eax`
- `p *0x80123456`
- `x/10x 0x12345678`: examine memory and display in hex
- `x/10s 0x12345678`: examine memory and display in string
- `x/10i 0x12345678`: examine memory and display in instructions
- `help p`
- `help x`

Find Data in Memory

- `find "sh"`
- `find 0x01020304`
- `find 0x400000, 100000, "sh"`

Trace Function Calls

- `backtrace`: show function trace
- `up`, `down`: update current call stack
- `http://web.mit.edu/gnu/doc/html/gdb_8.html`

Alter Data

- `set variable num = 10`
- `set {int}0x8038290 = 10`
- `set $eax = 0x12345678`

PEDA

- *Python Exploit Development Assistance*
- enhancement for GDB
- create cyclic patterns
- Return Oriented Programming features
- custom view: code, registers, stack
- shellcode features
- telescope an address (follow pointers)

Times in Getting from Source Code to Process

- compile time: when translating source code to object code in object files (using gcc, gas, nasm)
- link time: when aggregating multiple object files into an executable file (using gcc, ld)
- load time: when executable is loaded in memory and a process is created (using ./program)
- run time: while the process is running (using strace -p, lsof -p)

- linking is getting object files together into an executable or dynamic-linking file
- for the linker, object files are input and executables are output
- loading is getting an executable into memory and starting a process
- for the loader, executable file is input, process is output

Static linking

- all symols are solved at link time
- all code is part of the executable
- static executables
- large executable files, but with no dependencies, highly portable

Load Time Dynamic Linking

- symbols are marked as stubs inside the executable file
- symbols are solved at load time, the moment the process is created
- symbols are picked from dynamic-linking library files
- provides reduced size executable files but requires dependencies to be satisfied

Run Time Dynamic Linking

- linking (and loading) is done at runtime
- it may be implicit (lazy binding) or explicit
- dlopen, dlsym for the explicit case: explicitly load a library and locate a symbol

Lazy Binding

- postpone linking of a symbol until it is called
- usually done for functions through the use of a trampoline section (PLT for ELF)
- the first time a function is called, the dynamic linker also does the binding

Locating Libraries

- for stating linking, use the -L argument to gcc
- for dynamic linking, the dynamic linker/loader is used: ld-linux.so
- `man ld-linux.so`
- searches for
  1. values in `LD_LIBRARY_PATH`
  2. the `/etc/ld.so.cache` file; populated by `ldconfig`
  3. the default `/lib` and `/usr/lib` library folders

PLT

- used for external library function calls
- generic trampoline code to jump to initially jump to per-function binder (`.plt` in ELF)
- writable data area storing function pointers (`.got.plt`)
  - initially store pointers to binder code (symbol solver)
  - after the first call store actual pointer to function call

GOT

- Global Offset Table
- `.got` in ELF for global variables
- `.got.plt` in ELF for external library function pointers
- local uses of external library symbol point to GOT
- GOT if filled by the dynamic linker at the beginning

Keywords

- static analysis
- dynamic analysis
- executable
- ELF
- readelf
- section
- segment
- disassembling
- objdump
- symbols
- linker
- process
- strace / ltrace
- lsof / pmap
- perf
- GDB
- breakpoint
- `info`
- `examine`
- `ni`, `si`
- `backtrace`, `up`, `down`
- `write`
- `searchmem`
- dynamic linking
- dynamic loading
- lazy binding
- trampoline

Useful Links

- http://www.skyfree.org/linux/references/ELF_Format.pdf
- ftp://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html
- https://msdn.microsoft.com/en-us/library/windows/desktop/ee416588(v=vs.85).aspx
- https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html