Lecture 3

The Stack. Buffer Management

Computer and Network Security
October 14, 2019

Computer Science and Engineering Department

Runtime/Dynamic Analysis

- inspect processes
- inspect resources: file, sockets, IPC (lsof, netstat, ss)
- inspect memory: pmap, GDB
- inspect calls: strace, ltrace
- thorough inspection: in debuggers (GDB, Immunity, OllyDbg)

Runtime Application Security

- attack vulnerabilities in process address space and process flow
- attacker aims
  - get a shell
  - privilege escalation
  - information leak
  - denial of service
- defender: hardening process and runtime environment (libraries, permissions, sandboxing, monitoring)

Knowledge and Skills Required

- thread and process management
- (virtual) memory management
- intimate information on the process address space
- working with arrays and strings
- hex/binary
- assembly, dissasembling
- platform ISA
- good skills working with a debugger

Process Address Space

- memory address space of a process
- linear
- memory areas, responsibilities
- static/dynamic allocation
- memory mapping
- access rights

Text

- stores code
- read only and executable
- instruction pointer/program counter points to current instruction
- libraries posses code segment
- instruction pointer may jump to library code

Data

- stores data (global variables)
- `.data`, `.bss`, `.rodata`
- read-write, `.rodata` is read-only
- accessed through normal registers (eax, ebx, ecx, edx)

Heap

- dyanamic memory allocation
- `malloc` and friends
- linked list implementation in the backend
- pointer madness
- memory leaks
- read-write

Stack

- store function call frames
- function arguments and local variables
- stack pointer, frame pointer
- read-write

Generic Stack Operations

- push: push new element on stack
- pop: pop element on stack, return `null` if no element on stack
- top/peek: show last element on stack
- can only push to top and pop from top of the stack

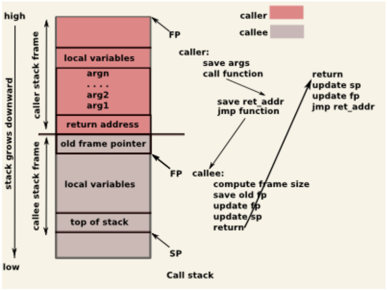
The Stack in the Process Address Space

- it's bottom up in x86 architecture
- base address points to bottom of the stack
- stack pointer points to top of the stack
- stack pointer <= base address
- stack size = base_address - stack pointer
- stack "grows down"
    - when stack grows, stack pointer decreases in value
    - when stack decreases, stack pointer increases in value

Push/Pop

- push operation adds data to stack: stack grows, stack pointer decreases
- push is equivalent to
    - `sub $4, %esp`
    - `mov value, (%esp)`
- pop operation removes data from stack: stack decreases, stack pointer increases
- push is equivalent to
    - `mov (%esp), value`
    - `add $4, %esp`

Stack Frame

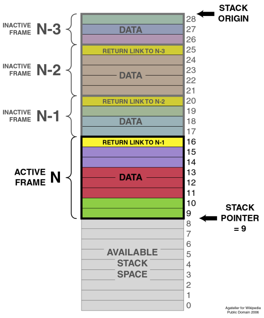


http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-04

Stack Frame (2)

- caller and callee
- stores current function call context
- stores return address
- identified by frame pointer
- What does the `-fomit-frame-pointer` option do?
- call stack
- stack (back)trace

Call Stack



http://en.wikipedia.org/wiki/Stack_(abstract_data_type)#Hardware_stacks

Calling a Function

- push function arguments, stack pointer decreases, the stack grows
- issue `call new-function-address`
    - save/push instruction pointer on stack (stack grows, stack pointer decreases
    - jump to `new-function-address`
- save/push old frame pointer
- save current stack pointer in frame pointer register
- save registers
- make room on stack (stack grows, stack pointer decreases)

Returning from a Function

- discard stack (stack decreases, stack pointer increases)
- restore/pop registers
- restore/pop old frame pointer
- issue `ret`
  - restore instruction pointer from top of the stack (stack decreases, stack pointer increases)
  - continue execution from previous point
- restore frame pointer
- discard stack in caller frame

What is a Buffer?

- an array of bytes for storing temporary data
- generally dynamic (its contents change during runtime)
- frequent access: read-write
- base address, data type, number of elements
- buffer size = number of elements * sizeof(data type)

Why Buffers?

- store data during runtime
- pass data between functions (arguments or return values)

On Memory Allocation

- static allocation: at compile time (in data or bss)
- dynamic allocation: at runtime (`malloc`, on heap)
- automatic allocation: on the stack, during runtime, usually fixed size
- in case of dynamic allocation, the pointer variable is stored on the stack and the actual buffer data is stored on the heap
- allocation granularity is the page at OS/hardware-level

Arrays vs. Pointers

- `int buffer[10];` – array
- `int *buffer;` – pointer
- array occupies sizeof(buffer)
- pointer occupies sizeof(int *) + size_of_buffer
- an array is like a label
- a pointer is a variable

Problems with Buffers

- you have to know their length
  - buffer overflow
- you have to be careful about the index
  - index out of bounds
  - buffer overflow
  - negative index

Buffer Overflow

- write data continuously in buffer (`strcpy`-like)
- pass buffer boundary and overwrite data
- may be exploited by writing function pointers, return address or function pointers
- allocations is page level, so overflow won't trigger exceptions
- may be stack-based or heap-based

What if?

- not enough arguments for a function call
- too many arguments for a function call
- overflow of local buffers

The Return Address

- stored on the stack to allow jump back
- may be overwritten and allow random jumps (the stack is read write)

Stack Overflow

- the stack overflows, goes into another memory zone
- may be the heap
- may be another stack in case of a multithreaded program

Stack Buffer Overflow

- overflow buffer on stack and rewrite something
- rewriting may be a local variable (number, function pointer) or return address of current stack frame
- if rewriting a function pointer jump to a conveniant address: address of buffer on stack, address of environment variable, address of function in libc

Rewrite the Return Address with Address on Stack

- the usual way to exploit a stack buffer overflow (needs non-executable stack)
- do a stack buffer overflow and overwrite the return address (ebp+4)
- ovewrite with start address of buffer on the stack
- when function returns, jump to start address of buffer
- carefully place instructions to execute desired code at the beginning of the buffer (also dubbed shellcode)

NOP Sled

- buffer may be placed at non-exact address
- one solution is guessing the address
- the other is placing a sufficient number of NOP operations and jump to an address in the middle of the NOPs
- the program executes a set of NOPs and then reaches the actual shellcode

Shellcode

- a sequence of instructions allowing the execution of an instruction similar to `system("/bin/sh");`
- usually provides a shell out of an average program
- may do some other actions (reading files, writing to files)
- the shell is a first step of an exploitation
- followed by an attempt to gain root access
- more on "Lecture 03: Exploiting"

Return-to-libc Attack

- jump to a function call in the C library (such as system or exec)
- may be used in heap or data segments
- useful when stack is non-executable

Demo

- the stack in shellcodes
- level 5 from io.smashthestack.org

Keywords

- address space
- stack
- push
- pop
- stack frame
- call stack
- stack trace
- `call`
- `ret`
- buffer
- allocation
- buffer overflow
- return address
- NOP sled
- shellcode

Useful Links

- Aleph One – Smashing the Stack for Fun and Profit: `http://insecure.org/stf/smashstack.html`
- `http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/stack.html`
- `http://www.cs.vu.nl/~herbertb/misc/bufferoverflow/`
- `http://www.win.tue.nl/~aeb/linux/hh/hh-10.html`

References

- Security Warrior
  - Chapter 5. Overflow Attacks
- The Ethical Hacker's Handbook, 3rd Edition
  - Chapter 11: Basic Linux Exploits
- The Art of Exploitation, 2nd Edition
  - Section 0x270. Memory Segmentation
  - Chapter 0x300. Exploitation