Script:
Program Analysis and Binary Exploitation

Paul Scherer

University of Bonn
Institute of Computer Science
Lecturer: Dr. Elmar Padilla, Martin Clauß
Winter term 2019/20

Contents

Liability exclusion

1 Basics

- binary representation
  - big endian: begin with highest-value byte, network byte order
  - little endian: begin with lowest-value byte
  - two's complement
  - word sizes
- positional notation
- base conversion

Structures

- lists
  - queue: first in first out (FIFO)
  - stack: last in first out (LIFO)
  - set: unsorted
- structs
- heaps
- processes: instance of a computer program
  - tools: ps
- threads: lightweight processes
- virtual memory
- ELF (Executable and Linkable Format) under UNIX
  - ELF header: magic byte 0x7f454c46 (0x7f, ELF)
  - program headers
  - section headers
  - .text segment: code
  - .(ro)data segment: (read only) data
  - .dynstr / .dynsym segment
  - .bss segment
  - .got: global offset table
  - .plt: (lazy) procedure linkage table
- PE (Portable Executable) under Windows

1.1 Analysis

Static analysis

- concept: parsing the binary directly

  source code overwhelmingly not available

- disassembly: conversion binary code → assembler instructions

  (almost) full reconstruction possible

  linear sweep disassembly: instruction after instruction, no logic checks, easily confusable

  recursive disassembly: next "known" instruction, following jumps/calls, logic analysis

  tools: objdump, gdb, IDA

- decompilation: conversion assembler instructions → program code

- dataflow analysis: programme as graph

- symbolic execution/evaluation: abstract interpretation as case differentiation (in a graph)

  input: symbolic values

  techniques: loop unrolling

  problem: path explosion (number of paths = $2^{\text{number of branches}}$)

Dynamic analysis

- concept: program as black box / unknown function of the input

  input → black box → output

  focus on concrete values for one execution path

- sandbox: execution in an isolated and controlled environment on the real processor

  implemented in virtual machines (Oracle Virtual Box, KVM, Xen, Qemu)

- emulation: execution on a simulated processor

- debugging: monitoring of the running process

Virtual Memory

- not enough physical RAM available for all processes

- mapping: program/virtual address → address translation → RAM/physical address

- outsourcing of pages to the hard disk if not enough space free

- organised in a page table

  hit: page table entry (PTE) is valid, data retrieved

  miss: page fault triggered → missing page loaded by the page fault handler

- address space separation: own physical pages for each process
  - shared pages possible
- virtual memory protection: permission bits for each virtual page
  - supervisor/kernel mode access
  - read, write
  - execution
- segments in process virtual memory
  - stack
  - heap
  - `.bss`: uninitialized static variables
  - `.data`: initialized local static and global variables
  - `.text`: code with executable instructions
- stack
  - direction of growth: downwards, lower addresses
  - used as temporary workspace
  - stack frame: coherent data in a block
  - backtrace: list of currently active functions
- heap
  - direction of growth: upwards, higher addresses
  - dynamically allocated
  - allocation via `malloc(), calloc(), realloc()`
  - unblocking manually via `free()`

Debugger

- components
  - interrupts handler: hardware-/softwareinterrupts, debug exceptions
  - system information extractor: memory inspection
  - communication protocols
- debug symbols: additional information in the symbol table
- hybrid analysis: combination of methods for static and dynamic analysis

POSIX signals

- purpose: notification to a process about the occured event
- properties:
  - asynchronous
  - can be sent by any process to another process at any time
- 31 signals defined

DBI (Dynamic binary instrumentation)

- purpose: insertion of code into a process without modification of the original binary
  extract runtime information
  change of behaviour of the program
- applications: e.g. Valgrind
  vulnerability research/bugs hunting, product hacking

1.2 Tools

- `file`: determine file type
  three sets of tests: filesystem tests, magic tests, language tests
  result of the first successful test returned
- `readelf`: extract ELF header
- `objdump`: extract file headers, sym(bol)s; disassemble
- `ltrace`: library calls of the program to the GNU C Library, received signals
- `strace`: system calls of the GNU C Library to the Linux kernel, received signals
- `strings`: ascii strings in the binary (consecutive printable values + `\0`)
- `nm`: extraction of the symbol table

gdb commands

- running a programme: `run`
  `run args`
  `run < file_input`
- breakpoints: stop at specific addresses
  `b[reak] function_name`
  `b[reak] *address`
  conditional: `b[reak] position if condition`
- navigation
  `next/nexti`: next high/low-level instruction
  `step/stepi`: step into high/low-level
  `continue`: execution until the next breakpoint
  `until function/line`: execution until the function/line
- information
  `print x`: variable/register
  `x/7bx$sp, x/1i $rcx`: examination of stack/memory/register
  `info registers`: register values

- watchpoints: stop if the value of an expression changes
- attaching a process

    `gdb program PID`

    `gdb; attach PID`

2 Fuzzing

- aim: finding bugs and vulnerabilities in software
- process: input generation/mutation → binary execution → binary misbehaviour? → crash analysis, bug and exploit

Approaches

- dumb fuzzing

    brute-force over the input space

    random generation/mutations of the input
- coverage guided fuzzing

    iteration over an initial input queue with random mutations

    detection of new branches during the execution

    → improved performance

Target selection

- concrete target given or projects from Github chosen
- preferable C/C++ projects and standalone binaries or libraries
- harder:

    non-compiled languages with native modules

    server applications
- target preparation: usually special compilation required

    e.g. modification of makefiles, etc.

    all dependencies must be included

Fuzzer selection

- dump fuzzing: `radamsa`
- coverage guided fuzzing: `AFL, libfuzzer, Hongfuzz`
- black-box fuzzing: limited knowledge

    given: e.g. closed-source binary

    only the output available

    special tracing techniques necessary for internal information

- white-box fuzzing: full knowledge about target
  given: e.g. source code
  target modification possible
  current state of the binary known
- decision criteria
  suitability
  performance
  personal preference

AddressSanitizer (ASAN)

- purpose: bug detection during runtime
  no protection provided
- parameter: `-fsanitze=address`
- process: compilation → binary execution → bug triggered? → crash
  performance significantly effected
- mechanism: memory access check for validity
  check of every memory access with additional code
  red zones around allocated memory reserved
  check against the shadow memory
  special implementation for `malloc(), free()`
- limitations
  no false positives, but false negatives
  overwriting of allocated variables/objects not detectable

+ advantage: earlier bug recognition

- disadvantage:
  higher memory consumption
  slower execution

Corpus

- selected example inputs instead of random inputs
- purpose: high coverage
  diverse inputs within a limited total amount of samples
  maximal coverage with minimal number of examples

Crash analysis

- first hints in the ASAN output
  - category: e.g. stack/heap overflow
  - exploitability estimation
- further steps
  - debugging with crash input
  - exploit development for Prove of Concept (PoC)
- crash deduplication by
  - same execution path
  - same stack trace, compared by hash value
  - "ground truth", very precise but expensive
- problem: some constraints too specific for more or less random inputs

→ symbolic execution possibly more helpful

- fuzzer evaluation as complicated task depending on
  - applied metric
  - usable results

3 Vulnerability Research

3.1 Definitions

- a **bug** exposing a **vulnerability**, that can be **exploited**
- vulnerability: flaw/weakness in
  - system's design
  - implementation
  - operation and management
- control flow: blocks of instructions connected with arrows
- data flow: semantics of blocks connected with arrows

Attack objectives

- **C**onfidentiality
- **I**ntegrity
- **A**vailability

3.2 Process

Scoping

- often not possible to analyse the complete program
- limitations by
  certain functionality
  concrete elements
  time constraints

⇒ definition of goals

- salt: triangle of
  termination
  soundness/unsoundness: all/some facts understood
  completeness/incompleteness: actual/additional facts

Classification of variables

- forward: reaching definitions
  valid assignments not yet overwritten
  compression by constant propagation: equivalent replacement for code segments
- backward: live variable analysis
  variable with a value that may be needed
  compression by dead code elimination

Intermediate representations

- abstraction from assembler instructions
  computation in different steps
  improved readability
- existing representations
  vex, reil, llvm, esil, bil/bnil

3.3 Attack surface

- Program input/output
  environment variables
  command line arguments
  stdin
  files
  network security
  signals/exceptions

- symbols
- execution

C language issues

- assumption: deep knowledge about the language given
- not implemented
  - initialisation of data structures
  - prevention of out of bounds reading/writing
  - avoiding double-free, use-after-free, freeing unused memory
  - invalidation of dangling pointers

⇒ critical data possibly overwritten
  - depending on the location of the buffer

- casting of every data type to every other data type possible
  - unforeseen consequences
- manual error checking necessary
  - overflows possible if not done
- truncation: information loss through
  - casting a bigger type to a smaller one
  - shifting a value out of its range
- signed/unsigned variables as the other format
- partially detectable with ASan (AddressSanitizer)

4 Binary Exploitation

- stack frame for functions
  - parameters (right to left)
  - return address: pushed by call
  - old base pointer: saved by the callee
  - local variables (order dependent on the compiler)

4.1 Buffer overflows

- requirement: copy functions without length checking
  - `strcpy()`, `scanf("%s")`, `std::cin`, ...
  - function calls with incorrect/manipulated copy size
- more data written to a buffer than it can hold

Return Address Manipulation

- overwriting the return address
    - filling buffer and base pointer
    - setting new return address to an arbitrary address
- target addresses
    - functions given in the program's code
    - shellcode (NOP slide + malicious code)
    - libc functions
    - beginning of a ROP chain
- demands for shellcode
    - fitting in the available size
    - bad characters free (`\0`, `\n`, ...)
    - position independent
    - environment dependencies
    - stored in
        - buffer
        - environment variables
- ret2libc: calling functions in the standard C library (libc)
    - useful: system() to execute an arbitrary command
    - preparation of a stack frame for system()
    - 64-bit: pop parameter in register rdi (parameter on top of the stack, "pop rdi; ret" gadget)
- ROP (Return Oriented Programming)
    - requirements for gadgets
        - stored in program's memory (code segment, libc, etc.)
        - executable (R-X permission)
        - ending with `ret`
    - stack pointer as new instruction pointer
    - ROP-chain: addresses of gadgets written on the stack
    - gadgets: 3-4 instructions in average
        - x86 instructions interpretable from any given offset
        - tools for automatic gadget search
        - side effects: undesirable instructions possibly included in gadgets
            - compensation necessary
    - alternative: execution of mprotect for the stack, execution of the shellcode directly

Shellcode

- historically: starting a shell locally
- remote shell
  - TCP bind shell: connection attacker → target, likely blocked by the firewall
  - TCP reverse shell: connection target → attacker, outgoing connection usually allowed
- crafting
  - given: compiled C program with the desired functionality
  - tracing system calls
  - determining syscall numbers and parameters
  - rebuild in assembly
  - transformation to a hexstring
  - de-nullifying, removal of bad characters (`\0, \n`): replacement of single instructions
  - testing
  - alternative: automation tools (`pwn.shellcraft`)
- constraints
  - functions stopping at `\0,\n`, partially only alphanumeric characters allowed
  - signature matching in the target's firewall: detecting typically two parts (vulnerability trigger, payload)
  - shellcode obfuscating used to bypass signature matching (NOP instructions, jumping, encryption)

4.2 Heap Overflow

- growth and writing direction: towards higher addresses
- several implementations used
- components
  - arena: references to at least one heap
  - chunks: memory for `meta-data ; user data`, multiple of 8 bytes as size
  - free chunks: `prev size;size, AMP;forward pointer;backward pointer;data`
  - in-use chunks: `prev size ; size,AMP ; user data`
  - AMP: PREV_INUSE flag, indicator for usage of the previous chunk
  - bins: management of free chunks
    - fast bins: singly linked list
      - holding recently freed small chunks
      - LIFO list
      - inuse bit of entries still set

small/unsorted/large bins: doubly linked list

FIFO list

consolidation: combination of small and large bin chunks to larger chunks in the unsorted bin

condition: freed and to be freed chunks bordering a free chunk

unlinking from the doubly linked free list

increasing the size of the combined chunk

- GOT (Global Offset Table): jump targets for functions

  interesting objective for overwriting

Heap Overflows

- Unlink exploit: overwriting meta-data with user data

  arbitrary code execution possible

  `meta data;user data;meta data;user data`, second meta-data manipulated

  Write-What-Where condition when unlinking a free chunk

- Write-What-Where condition: *write* an (almost arbitrary) *value* to an (almost) arbitrary *location*

- controlled FD and BK ⇒ arbitrary write possible

  substitutions of function addresses

- steps

  allocation of two memory chunks

  writing the first chunk with a vulnerable function

  setting up a freed fake chunk:

  `dummy dummy;jmp+10;dummy dummy;sc + padding;-4 -4;&free;& sc`

  calling `free(first)` : overwriting the GOT entry of free with the shellcode

  calling `free(second)` : executing the shellcode

  dummy bytes overwritten by the unlink algorithm

Use After Free (UAF)

- freed data on the heap used as valid memory with a leftover reference / dangling pointer

- dangling/stale/wild pointer: reusable pointer/reference to freed data

  current content unknown

  requirement for UAF vulnerability

- several attack vectors available

- code execution
  memory chunk allocated for a structure with fields function pointer, char*
  freeing of the chunk and allocation for structure with fields int, char*
  user data written in int, char* of the new structure (&system, "/bin/sh")
  a pointer to the old structure used to call the function pointer
  ⇒ shell with system() and `/bin/sh`
- write condition / GOT overwrite
  memory chunk allocated for a structure with fields function pointer, char*
  freeing of the chunk and allocation for a structure with fields int1, int2, char*
  user data written in the new structure, int2 (desired target address)
  user data written in the old structure, char* (desired value)
  ⇒ write condition
- read condition
  memory chunk allocated for a structure with fields function pointer, char*1, char*2
  freeing of the chunk and allocation for a structure with fields int, char*1, char*2
  user data written in the new structure, char*1
  data read from the old structure, char*1
  ⇒ possibly secret data read
- exploitations
  VTable hacking (Virtual function Table): list of pointers to virtual functions

Heap Feng Shui

- influencing the heap layout
- concept
  deterministic heap allocator
  control of the heap layout with a specific sequence of allocation / free
  ⇒ determined address of a new object
- approach
  closing all holes
  adding a big consecutive memory block
  poking holes by deallocation
  address of the next allocation known
  data in the allocated bytes set by the attacker

4.3 Format String Attacks

- format string: conversion of different datatypes to string representations

  conversion specification: %i, %s, . . .

  ordinary characters copied

  parameters pushed on the stack

- vulnerability:

  input string from user interpreted as command for format functions

- mapping out the stack content: %p, %#.8x

  direct parameter access at #-th argument: %#$p (possibly $ to be escaped)

- reading arbitrary memory: %s

  reading until \0

  crash at invalid addresses → DoS attack

  problem: null bytes in the address

- write-what-where with %n

  %n: number of bytes written so far stored in the supplied pointer

  identification of the direct parameter access

  desired address in the first 4/8 byte (32/64 bit) of the format string

  application of %n for the identified direct parameter access

  format string:
  desired address;width field, direct parameter access; AAAA%008x%<#>$n, address size + width field = target value

  problem: addresses as big integers → length modifier (%hn: 2 byte value, %hhn: 1 byte value)

  ⇒ 4 bytes written to an address of our choice

- GOT overwrite

  replacing the address of a function in the GOT

  determining format string offset for %hn: pwnlib.fmtstr, BASH-Fu, try & error

  address of the GOT entry for the desired function: static/dynamic analysis tools (objdump/gdb)

  address for redirection of the program flow

- GOT alternatives

  DTORs: destructor in object oriented languages

  FINI_ARRAY: (optional) segment with instructions for process termination

  C library hooks: functions modifying the behaviour of malloc(), realloc(), free()

  __atexit structures: function called while execution of exit()

  function pointers

5 Protection Techniques

Stack

- stack canary: additional value between saved base pointer and return address
  static canary: fixed bytes
  random canary: generated at every call
  terminator canary: containing null byte(s) (`\0`)
  canary check against a save backup before returning
  problems:
  - canary brute forced or guessed
  - reading the canary by an information leak
  - setting the master canary to a known value
  - probably not all functions protected
- non-executable stack (NX)
  data on the stack marked as non-executable
  ⇒ ret2libc or ROP applicable
- checking tool: `checksec`

ASLR (Address Space Layout Randomization)

- randomized addresses of process segments
  ⇒ no fixed addresses in exploits working
- circumvention possible
  libraries, executables not as position independent code (PIC, PIE)
  investigation of addresses with brute-force
  information leak

RELRO (Relocation Read-only)

- headers in the binary marked as read-only when linker finished
- partial RELRO: .ctors, .dtors, .jcr read-only
  CTOR: constructor in object-oriented languages
  .jcr: section for registering compiled Java classes
- full RELRO: additional GOT read-only

6 Exim RCE

- Exim: mail transfer agent (MTA) for Linux systems
  mail relay over SMTP

Development environment

- virtual machines

  automated setup with Vagrant

- containers using

  LXC, Container Linux

  Docker, Singularity

Vulnerability

- exploit for the Base64 decoding function

  $3n + 1$ bytes allocated, but $3n + 2$ required

  ⇒ one byte heap overwrite

- own memory management implemented based on *libc*

  blocks in a singly linked list

Exploit

- Access Control List (ACL) checks can be pre-defined in the configuration file by the administrator

- predefined check string for ACL check overwritten with the exploit in the main memory

  `AUTH PLAIN` as malformed Base64 string

  → size field of the next chunk overwritten

  using heap feng shui

- extended chunk used to overwrite the following next pointer

- `free()` called in `smtp_reset`-step

- manipulated ACL storebook is reallocated and the `acl_smtp_rcpt` is overwritten with e.g. "${/bin/bash -c {'touch /tmp/pwnd'}}\0"

⇒ shell available for further exploitation

Limitations

- attack only working with deactivated ASLR

  partial overwrite possible to bypass ASLR