A PROGRAMMER'S PERSPECTIVE

MINSEOK SONG

Code Security

- In the beginning of the chapter, B & H introduces two scenarios regarding vulnerability.
- 1) Unlike Java, C doesn't feature a garbage collector. However, when declaring an array such as char kbuf[KSIZE];, the array kbuf can be initialized with garbage values. In two's complement representation, there's a significant difference between -MSIZE and MSIZE due to the most significant bit being set to 1 for negative numbers. This discrepancy can introduce vulnerabilities and potentially lead to memory leaks.
- 2)If we do malloc(ele_cnt * ele_size) but if each argumnent is very large, the program might not allocate the desired memory size due to integer overflow.

Two's Complement

arithmetic and Underflow/Overflow When adding two numbers in two's complement representation, there's a potential for underflow and overflow.
Let's denote the result of addition as $TAdd_w(u, v)$
where w is the bit width of the numbers, and u and v are the numbers being added. Then, we can define $TAdd_w(u, v)$ as:

$$TAdd_w(u,v) = \begin{cases} u+v+2^w & \text{if } u+v < TMin_w \\ u+v & \text{if } TMin_w \leqslant u+v \leqslant TMax_w \\ u+v-2^w & \text{if } u+v > TMax_w \end{cases}$$

Here, $TMin_w$ and $TMax_w$ are the minimum and maximum values representable with w bits in two's complement, respectively.

For instance:

- When we add two values (i.e., $0\ldots$ and $0\ldots$) and get a result starting with $01\ldots$, it indicates an overflow (desired: $a-2^{w-1}+b-2^{w-1} = a+b-2^w$, but what we get: $a+b+2^w-2^w = a+b$, where a and b are bits after the most significant bit, so subtract 2^w).
- When we add two values (i.e., $1\ldots$ and $1\ldots$), and the result is $1\ldots$, it indicates an overflow (desired: y, but what we get: $-(2^w - y) = y - 2^w$, so add back 2^w).

Remark 1.
- Multiplication and division are a bit more involved; but essentially, we use addition and bit-shifting operations to accomplish tasks (we need to be careful when dividend is negative number and introduce the concept of "bias").
- When sign-extending an integer using the >> (right shift) operator, the most significant bit (often called the sign bit) is replicated to fill in the shifted positions.

Assembly Language

Register for x86-64

- Extensive list is on `https://wiki.osdev.org/CPU_Registers_x86-64`
- RIP: Instruction pointer(used for count)

- RAX, RBX, RCX, RDX: general Purpose Registers, with RAX often used for return values.
- R8 to R15: extra general purpose register
- RSP: Stack pointer, RBP: Base Pointer
- RDI, RSI, RDX, RCX, R8, R9: used for function arguments

Basics, Control Flow

- The type of processor (specifically its architectore or ISA - instruction Set Architecture) dictates the set of instructions that are available for use.
- The example of processor includes
  1. Intel: x86(widely-used 32-bit architecture), IA32(Intel's 32-bit architecture), Itanium(64-bit architecture developed by Intel, didn't see wide adoption compared to x86-64 pioneered by AMD), x86-64(later cross-licensed, enabling both companies to introduce enhancements since)
  2. ARM: Used in almost all smartphones
- There are three ways to get assembly language for the code.
  1. Use 'gcc' with the '-S' flag. For example, put gcc -S source.c
  2. Reverse engineering for the compiled binary. For example, put objdump -d binary_name
  3. Using 'gdb' debugger. For example, go to gdb and put disassemble function_name
- The last two options are appropriate when you do not want to go into the source code directly.
- Address computation
  - 0x8 (%rdx) means to add 0x8 to the address %rdx.
  - (%rdx, %rcx) means to add the address %rdx with %rcx.
  - (%rdx,%rcx,4) means to add the address of %rdx with four times the address of %rcx.
- Some instructions for x86-64 architecture
  1. movq: simply just moving; q stands for quadword (64bits, versus 8 bits for byte, 16 bits for word, and so on), meaning that this operates on a 64-bit quadword operand.

     Example 2. When we do movq (%rsi), %rdx, this means that we are assigning the value at the address %rsi to the register %rdx.
  2. leaq: this stands for "load effective address quadword."

     Example 3. when we do leaq (%rbx, %rcx, 4), %rax, we'd calculate %rbx+4*%rcx and put this address into %rax. This is different when using movq, we would access the value at that address.
     Note that %rbx and %rcx can be address or register.
  3. addq: add, imulq: multiplication, salq: shift, ret: return, etc
  4. when we do cmp, add, sub, etc, some flags (which belong to FLAGS register) are implicitly set, so we can use them for control flow.
  5. when we do cmp Src2, Src1, we effectly do Src1-Src2 but does not store result (as opposed to sub).
     (a) CF(carry flag) set: if carry out from most significant bit
     (b) ZF(zero flag) set: if a == b
     (c) SF(sign flag) set: if (a-b) ¡ 0
     (d) OF(overflow flag) set: if two's complement overflow: $(a > 0\&b < 0\&(a-b) < 0)||(a < 0\&b > 0\&(a-b) > 0)$
  6. some notable registers when we use control flow: temporary data, location of runtime stack , location of current code control point(points to the instruction being executed), status of recent tests.
  7. when we say %al, %bl, %cl, and so on, it refers to the lower end of %rax, %rbx, %rcx, and so on.

(8) movzbl: stands for move. zero-extend, byte, long. Extend the rest 32 bits to zero.
(9) store the return value on %rax.

- Conditional move vs. branch move
  (1) Conditional moves eliminate the need for branching by selecting a result based on a condition without branching.
  (2) If the prediction of branch move is wrong, it can lead to a pipeline stall due to mispredicted branches.
  (3) Even if we write if-else statement in C code, the compiler may use conditional move for optimization.

  Example 4. pitfalls: $val = x > 0 ? x* = 7 : x+ = 3;$

- GCC has some compiler optimization levels, such as -O1, -O2, -Og (the latter being high order, hence abstracting out),etc. -Og is suited the most for illustration purposes.

Example 5.
gcc -Og -o p p1.c p2.c
This does mean that we optimize using -Og flag, and then save it as p. We compiled the source files p1.c and p2.c into assembly language and linked them together.

- As a result of this optimization, two different source codes can generate the same assembly code.
- This is why reverse engineering can be challenging.
- Control flow in assembly language can be largely summarized by the following three principles.
  (1) Conditional jump

  Example 6.
  CMP AX, BX; Compare the contents of registers AX and BX
  JE Label; Jump to 'Label' if AX equals BX (JE stands for "Jump if Equal")

  (2) Conditional move

  Example 7.
  CMP AX, BX; Compare the contents of registers AX and BX
  CMOVNE CX, DX; Move the value from DX to CX only if AX is not equal to BX

  (3) Indirect jump via jump tables

  Example 8.
  movq %rdx, %rcx
  cmpq $6, %rdi # x:6
  ja .L8
  jmp .L4(,%rdi, 8)

Machine Procedures

- ABI: a set of conventions about the register.
- There are three aspects: passing control, passing data, and memory management.
- Passing control
  (1) 'call' instruction is used to initiate a subroutine; addresses control.
  (2) %rsp stores the address of the top of the call stack.
  (3) When a function is called, the return address (the address of the instruction immediately following the function call) is pushed onto the stack.
- Passing data
  (1) We pass the first six data onto register %rdi, %rsi, .., %r9, (in this order, by convention) and move over to stack after sixth argument.

   (2) Return value is stored in %rax.
- Memory management
   (1) The usage of %rbp is optional - we usually know how much we "step back" by the amount of memories we need to deallocate.
- There are two register saving conventions
   (1) Caller-Saved Registers: Before making a function call, the caller is responsible for saving the current values of these registers if they need to be preserved. After the function call returns, the caller can restore the values if necessary. Examples of caller-saved registers include: %rax, %rdi, %rsi, ..., %r11.
   (2) Callee-Saved Registers: When a function is called (i.e., the callee), it's the function's responsibility to save the current values of these registers if it intends to use them. Before returning, the function restores their original values. Examples of callee-saved registers are: %rbx, %r12, %r13, %r14, %rbp, %rsp.