Reverse Engineering

bottom line: extracting knowledge or design information from anything man-made usually done to obtain missing information when it's not available in our context, this usually means that man-made == software usually an executable (since source code would mean just learning) either we grab the source and chug through it or we reverse engineer it

but sometimes it can also mean hardware

e.g., a power supply, a missile control system, etc

it's an old concept

we usually think of it as taking finished products and trying to figure out how they were made i.e., figure out their secrets

like examining things under a microscope to see what something does or how each part works we used to do this a lot (e.g., taking apart radios, electronics, etc) just to see how they work this is not as easy to do nowadays

software reverse engineering

software is actually quite complex

reverse engineering software means trying to figure out how software works it's a virtual process – only needs a CPU and your mind requires: code breaking, puzzle solving, programming, and logical analysis

why do people reverse engineer software?

to figure out how something works in order to develop a competing product but this actually doesn't make sense financially (just too hard and costly) two main (real) reasons:

security related software development related

security related reverse engineering

evaluating the level of security some encryption scheme has

e.g., developing and analyzing malware (for something like an "antidote")

e.g., analyze software to defeat copy protection schemes

the bad guys do this to find vulnerabilities so that they can create exploits exploits are used to gain access or obtain information good guys do this to analyze malware to see how it works to develop countermeasures

reverse engineering in software development

perhaps we have software that just isn't properly documented or we want to determine the quality of third party code (e.g., packaged APIs) maybe we want to achieve interoperability

mostly about dealing with improperly documented APIs

to ensure that your software (that makes use of them) works well

or developing competing software

why not?

or evaluating software quality and robustness

although looking at source code is much better than reverse engineering

but it's not always possible to obtain source code open source!

low-level software

aka, system software

encompasses compilers, linkers, debuggers, OSs, system utilities, assembly language, etc basically the stuff that isolates software developers (applications) from hardware we no longer need to deal with low-level stuff (since it's all available and has been developed) but years ago, we had to deal with low-level stuff

but to become good at reverse engineering, we need to know this low-level stuff in the end, reverse engineering tools just provide information (usually at a low-level) we must be able to extract meaningful information from this

the reverse engineering process

two main types of reverse engineering (or reversing):

system level reversing

determines the general structure of a program may locate areas of interest within it

code level reversing

provides detailed information on some chunk of code

system level reversing

involves running various tools on some program goal is to obtain information, inspect program executables, track program I/O, etc this requires close inspection of the OS since everything that goes to hardware requires the OS

code level reversing

sort of an art form

takes a lot of time and experience

no "manual"

involves extracting design concepts and algorithms from a program binary much harder to go from machine language to high level language than the other way around much is lost in the compilation process

in addition, the compiler automates many things which may not be evident to programmers

tools

reversing is all about the tools

system monitoring tools

sniff, monitor, explore, and expose the program being reversed displays information gathered by the OS about the application

disassemblers

translates machine language to assembly

a simple 1-to-1 conversion process

debuggers

allows software developers to observe their programs while they are running provide useful things like setting breakpoints and tracing through code other features include stepping through code, one line at a time (and much more) most of you have probably used debuggers before (usually part of an IDE)

```
decompilers
```

a step up from disassemblers translates a binary executable into a high level language a reversal of the compilation process (at least a try at it) not actually technically possible since things are lost in the compilation process

legality

this is an ongoing debate

usually revolves around: what social and economic impact reverse engineering has on society largely depends on what it's used for

going to the low-level

program structure makes large, complex software manageable

a program is broken down into chunks (e.g., libraries, modules, units, functions, statements) reverse engineers try to reconstruct this

machines don't need the organization that we do

in fact, they have a need not to have these (as they often bloat and slow things down) they remove redundancy, combine things so as not to worry about dependencies, etc

developers usually view separate chunks as black boxes

each black box must be well implemented and reliably perform its duties

each black box must have a well defined interface for communicating with the outside world reverse engineering means to first determine the component structure of an application

and then determine the responsibilities of each component

then to pick components of interest and find the details

tools used to "organize" code

modules

static libraries: compiled within the program

dynamic libraries: linked to separate files within the program

common code constructs

procedures: most fundamental unit

objects: logically associated data and code (state and behavior)

powerful paradigm (inheritance, etc)

data management

variables: the key to storing and managing data (fundamentally)

user-defined data structures: groups of data fields that are somehow related (think struct)

lists: items that are related in terms of their type (think arrays, linked lists, trees)

control flow

conditional blocks: allow for branching on multiple conditions (think if-statements) switch blocks: n-way conditionals with a bit more flexibility than if-statements loops: repeatedly perform the same thing a number of times

all have stopping or continuing conditionals

more on data management

```
int multiply(int x, int y)
{
```

```
int z;
              z = x * y;
              return z;
       }
       simple function; however, it can't be directly translated to low-level (i.e., line-by-line)
              low-level instructions (i.e., assembly/machine code) are too primitive
       but generally:
              store machine state prior to executing function code
              allocate memory for z
              load parameters x and y from memory into registers
              multiply x by y and store in a register
              optionally copy the result back into memory area previously allocated to z
              restore machine state stored earlier
              return to caller and send back z as the return value
       more complex, mostly from low-level data management
       why is this so different than a high-level language?
              it's all about speed and how the CPU works
              also how a bus is required to go to RAM
back to the low-level
       registers
              basically exist to avoid having to use the limited bus to go to RAM
              almost no performance penalty
              in the CPU
              but there are very few of them (32-bit processors have 8 usable ones)
              but long-term storage is still in RAM
       the stack
              values in a program are either placed in registers or on the stack
              maybe there are no available registers – go on the stack
              maybe there's a reason why some value needs to be placed in RAM – go on the stack
              the stack is an area in program memory (RAM) used for short-term storage
              think of it as secondary area for short-term storage
              registers: the most immediate data
              stack: slightly longer-term data
              usually used to temporarily store:
                     register values, local variables, function parameters and return addresses
assembly language (for IA-32 – Intel 32-bit architecture)
       it is the language for reversing
       often the only available way to look at executable code and to reverse it
```

IA-32 assembly (optional)

registers

8 generic ones:

EAX, EBX, EDX	used for any integer, boolean, logical, memory operation
ECX	sometimes used as a counter by repetitive instructions
ESI, EDI	frequently used as source/destination pointers in instructions that copy memory (SI=source index; DI=destination index)
EBP	mostly used as the stack base pointer (breaking it into frames)
ESP	stack pointer (points to the current position in the stack – anything pushed to the stack is placed beneath the pointer which is then updated)

flags

```
special registers (called EFLAGS)
contain status and system flags (notices)
we care about status flags (used to record the CPU's logical state)
some instructions operate based on these flags (e.g., conditional branching)
e.g., JCC (conditional jump) tests for certain flag values and jumps based on them
if (!blah)
return 0;
```

blah would be put into some register flags would be set to record whether it is 0 (false) or not a conditional jump instruction would branch based on this flag

instruction format

```
opcode and one or two operands opcode is an instruction operands are parameters for the instruction represent data (like parameters for a function) comes in 3 forms

register name: EAX, EBX, etc
immediate: a constant (e.g., 0x30004040)
memory address: somewhere in RAM (address enclosed in brackets)
e.g., [0x400394e]
general format: opcode dest operand, src operand
```

basic instructions

```
moving data: moves data from source to destination mov dest, src mov edi, [ecx+0x5b0]
```

arithmetic: basic arithmetic operations

ADD op1, op2	add two signed or unsigned integers (result stored in op1)
SUB op1, op2	subtract signed or unsigned op2 from signed or unsigned op1 (result stored in op1)
MUL op	multiply unsigned op by EAX (result stored in a 64-bit value EDX:EAX)
DIV op	divide a 64-bit unsigned op stored in EDX:EAX (quotient stored in EAX and remainder stored in EDX)
IMUL op	multiply signed op by EAX (result stored in EDX:EAX)
IDIV op	divide signed op stored in EDX:EAX (quotient stored in EAX and remainder stored in EDX)

```
edi, [ecx+0x5b0]
       mov
              ebx, [ecx+0xb54]
       mov
              edi, ebx
       imul
comparison: compare two operands and records the result in flags
       basically: op2 – op1 (result discarded, flags set appropriately)
       if the result is 0, the zero flag (ZF) is set (the two operands are equal)
       cmp
              op1, op2
              ebx, 0xf020
       cmp
conditional branches: branch to some address based on some conditions
       either branch to target address or go to the next instruction
       many variants (e.g., jnz – branch if ZF is not set)
       icc
              target address
              ebx, 0xf020
       cmp
              10026509
       jnz
```

function calls: implemented using two instructions (call and ret)
call pushes the current instruction pointer on the stack
and jumps to the specified address
ret returns to the caller (pops the address pushed on the stack during the call)

call function address call 0x10026eeb

a typical IA-32 function call sequence:

push eax
push edi
push ebx
push esi
push dword ptr [esp+0x24]
call 0x10026eeb

first four are the values of registers fifth is memory address at esp+0x24 dword ptr means a 32-bit int a stack address indicating a function parameter or local variable

tutorials

using Java reflection modifying the z-bit patching an executable