Programming Languages

Reverse Engineering

CSCI.GA-2110-001
Fall 2013

What Is Reverse Engineering?

Means different things to different people. Could refer to anything, not just software (hardware, pharmaceuticals, etc.) We will assume software.

One reasonable definition: "the act of analyzing something to determine its properties or how it works."

This may, but does not necessarily include translating software from executable form to source code.

Why Reverse Engineer?

Many uses for reverse engineering, including:

- Education
- Dealing with lost source code
- Inspect for viruses/malware
- Analyze malware to see what it does
- Security assessment
- Compiler validation
- To modify the reversed engineered code
- Gather evidence (e.g., patent infringement)
- Malicious purposes: compromising security, data theft, etc.

Is This Legal?

Disclaimer: consult an attorney.

Numerous overlapping laws (e.g. copyright, fair use, DMCA, patent). Not clear in many cases.

It's generally legal in the US to reverse-engineer software in order to:

1. Recover lost source code
2. Detect viruses
3. Inspect a program for purposes of research and teaching
4. Translate code from obsolete languages
5. Make software with unpublished APIs inter operable
6. Correct errors, when the author is not available

Is This Legal? Part 2

It's generally not legal to:

1. Reverse engineer a program you agreed not to reverse engineer.
2. Copy (copyrighted) source code or ideas contained within.
3. Circumvent security to avoid licensing/fees.
4. Practice a patented invention.

Fun fact! — bad people and bad companies don't care about laws.

Can it Be Prevented?

No, but it can be made difficult.

Two main strategies:

1. **Obfuscation**: modify the target language such that it is confusing to humans, but functionally equivalent to the original.
2. **Packing**: encrypt the executable. Must be decrypted prior to execution. Makes reflection difficult.

Things that won't prevent reverse engineering:

- Code signing: assures the recipient of the author's identity and that the code hasn't been modified since compilation. Does not prevent inspection or modification.
- Public/private key encryption. Ensures that license keys can only be generated by the author. Prevents *runtime* access to a protected area. (Useless if authentication can be bypassed.)

Can it Be Prevented?

- Watermarking: making every assembly unique in some way, such that the registered owner of a ripped off copy can be identified.

Overview

In this course, we've studied high level languages.

There are 2 varieties: *compiled* or *interpreted*:

- **Compiled**: translated from high level source code to some lower level format prior to execution (C++, Java, Ada). Can be statically inspected.
- **Interpreted**: source read and executed at runtime by an *interpreter* (Javascript, PHP, ML)

Here, we consider *compiled* code, where program instructions are either in:

1. Native executable format (".exe")
2. Hardware-independent format.
   - Java: bytecode (".class"), Dalvik (".dex")
   - .NET: Common Intermediate Language (CIL).

Java Reverse Engineering

Java is compiled into *bytecode* on desktops. Android devices use a Google-proprietary optimized format called *Dalvik* (".dex").

Java programs come in 2 forms:

- Applets : Runs locally in your web browser. Browser downloads the program as part of a web page. (Look for APPLET tag)
- Applications : Standalone applications.

Applets have more security restrictions. Applications behave more like traditional native software.

Java compilation is *lossless*: Java is translated to bytecode in a manner than can be completely reversed:

- Control structures preserved
- Identifiers preserved (class names, methods, data members).
- Package dependencies preserved

Java Part 2

Reverse engineering Java applets and applications is embarrassingly easy.

Options:

1. Disassemble into bytecode (e.g., using `javap`)
2. Even better: Decompile into original source code (e.g. using `jd`)

Terminology:

Disassembly is the process of converting binary executables to assembly code (including bytecode). Applies to compiled languages.

Decompilation is the process of converting disassembled code to high level source code.

Java Virtual Machine

Java bytecode is interpreted by the *Java Virtual Machine* (JVM).

Bytecode is hardware-independent pseudo-assembly. The JVM is responsible for transforming bytecode into executable instructions on a specific microprocessor. Basic components:

- Local method parameters
  - addressable by $0, 1, \ldots$, where 0 is always "this" (for non-static calls)
  - move locals to stack using instructions like `aload`
  - move stack contents to locals using instructions like `astore`
- Stack
  - method parameters are pushed here
  - method invocations consume the parameters
  - invocation instructions: `invokestatic`, `invokespecial`, `invokevirtual`, `invokeinterface`.

Java source

```
public void stop()
  {
    Debug.println("Inside stop()");

    this._stopThread = new Thread() { // anon inner class
      public void run() {
        try {
         sleep(1000L); } catch (InterruptedException le)) {
        }
        MainTAFS.this.realStop();
      }
    };
    this._stopThread.start();

    Debug.println("Leaving stop()");
  }
```

Corresponding Bytecode

```
public void stop();
   0:   ldc_w   #325; //String Inside stop()
   3:   invokestatic    #36; //Method Debug.println
   6:   aload_0
   7:   new     #326; //class MainTAFS$1
   10:  dup
   11:  aload_0
   12:  invokespecial   #327; //Method MainTAFS$1."<init>"
   15:  putfield        #303; //Field _stopThread;
   18:  aload_0
   19:  getfield        #303; //Field _stopThread;
   22:  invokevirtual   #328; //Method Thread.start:()V
   25:  ldc_w   #329; //String Leaving stop()
   28:  invokestatic    #36; //Method Debug.println
   31:  return
```

Steps

1. Obtain the Java class files (or Jar file)
2. To disassemble:

   ```
   javap -classpath Hello.jar -c full.path.to.Classname
   ```
3. To decompile: Use a decompiler like *Java Decompiler* (`jd`). There's also a GUI: `jd-gui`

.NET

.NET (pronounced "dot net") is a language framework developed by Microsoft.

Dozens of languages are built on top of this framework including C#, VB.NET, F# (OCaml/ML port), C++/CLI, J# (Java port), P# (Prolog port), A# (Ada port).

In common: high level code translated from each respective source language to a common pseudo-assembly format called *Common Intermediate Language* (CIL). Executed by an interpreter/JIT compiler known as the *Common Language Runtime* (CLR). CIL is embedded within an executable file (.EXE) in a format known as a *Portable Executable* (PE) file.

CIL: analagous to Java bytecode
PE file: analogous to Java class file, except also contains a CLR bootstrapper

.NET Part 2

- Disassembly: Microsoft Intermediate Language Disassembler `ildasm.exe` launches a code viewing GUI. Other tools also exist.
- Decompilation: .NET Reflector (commercial)

Demo: VS proj, ildasm, reflector

Native Applications

A *native application* is one which is fully compiled into executable machine-specific binary format. Unlike applications written in intermediary format,

- compilation of native applications is *lossy*. After compilation:
  - No high level control structures
  - No explicit type information
  - No variable names
  - Function names may not be present
- many-to-many: source to assembly, assembly to source.
- machine dependent:
  - every microprocessor has different assembly
  - structure of the binary executable (ELF, PE, etc.)
- not easy to distinguish between code and data

Native Applications Part 2

- sometimes cannot be disassembled with available information
- sometimes can be disassembled, but not decompiled

Useful tools for native applications: *nix Utilities:

1. `nm` : dump external symbols (carried over from object files).
2. `ldd` : will list the library dependencies.
3. `file` : will guess the type of file (good for determining executable format).
4. `objdump` : output general information about object files.
5. `dumpbin` : output general information about Windows PE files.
6. `c++filt` : determine name mangling for overloaded functions.
7. `strings` : list embedded strings.

Steps to Disassemble

1. Detect or specify executable format (ELF, PE, etc.)
2. Identify what to disassemble and where to start.
3. Read an instruction and map it to an assembly opcode.
   - fixed vs. variable width instructions
4. Identify the operand(s) of the instruction.
   - immediate?
   - memory address?
5. Write to assembly file.
6. Next:
   - Linear sweep: go to next instruction.
   - Recursive descent: follow branches.

Disassembly Algorithms

- Linear Sweep
  - No attempt to understand control flow.
  - **Pro**: complete coverage of the code.
  - **Con**: code and data can be intermingled. Difficult to distinguish. Chance of misclassification.
- Recursive Descent
  - When functions are encountered, locate & disassemble the function.
  - When branches are encountered, follow the branches.
  - If conditional, also proceed linearly.
  - **Pro**: more reliable code/data classification. Anything not covered assumed to be data.
  - **Con**: sometimes branch address is dynamically determined. No way to follow—dead end. Code can be misclassified as data.

Name Mangling

Languages encode subprogram details (e.g. signatures) into seemingly cryptic names.

For example, `??0B@@IAE@XZ` in GNU C++ means:

```
protected: ___thiscall B::B(void).
```

The main purpose of this is to make overloaded subprogram names short and unique.

Different compilers use different schemes for name mangling. Ability to de-mangle requires knowledge of the compiler (and possibly version) used to build the program.

x86 Assembly Primer

- 4 general purpose 32-bit registers: EAX, EBX, ECX, EDX
- 16-bit "registers within registers": AX, BX, CX, DX
- Example: AX is the lower 16 bits of EAX
- 8-bit "registers within registers": AL, AH, BL, BH, etc.
- Example: AL and AH are the lower/higher 8 bits of AX
- Other registers:
  - ESI, EDI (32-bit registers for moving large blocks of data)
  - ESP: stack pointer
  - EBP: base pointer (the "frame pointer" on x86)
  - FLAGS: flags register. Includes carry, overflow, sign, zero.

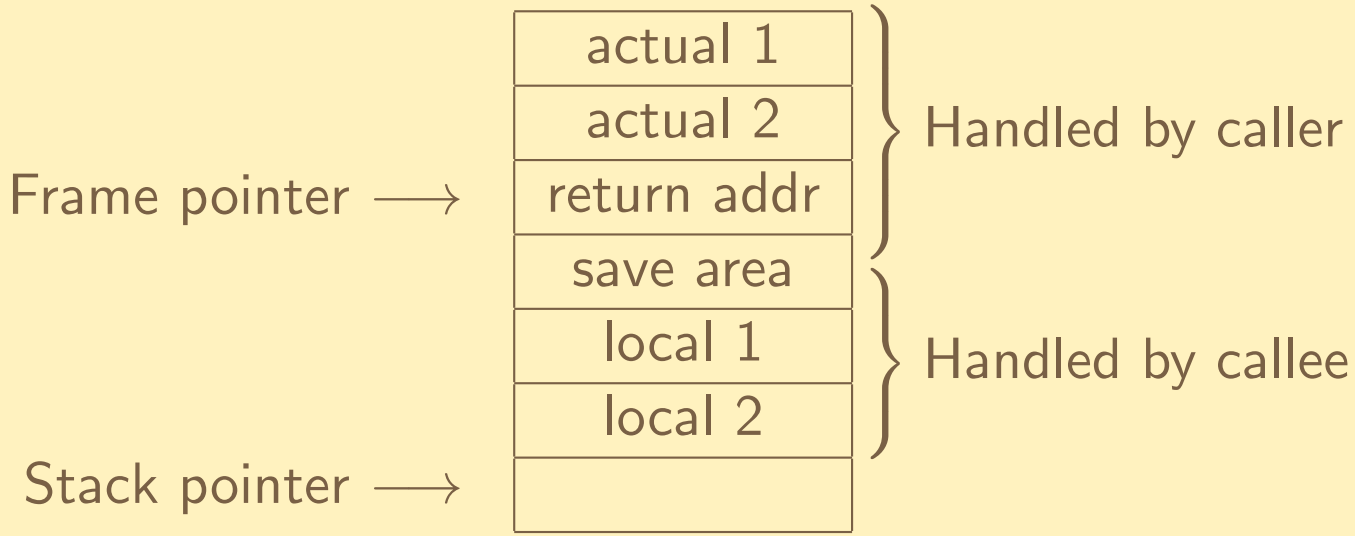
x86 Assembly Primer

Typical instructions (Intel syntax):

- `mov eax, ebx` : move contents of ebx to eax
- `mov eax, [ebx]` : move contents at address in ebx to eax
- `call near ptr addr` : push return address & jump to addr
- `leave` : equivalent to `mov esp, ebp` and then `pop ebp`
- `ret` : pop return address and jump
- `retn` : pop return address, reduce stack by another n bytes and jump
- `push eax`, `pop eax` : push eax to stack, pop top into eax
- `jmp eax` : unconditional jump to address in eax
- `jz addr`, `jnz` : jump if zero/if not zero to addr
- `sub eax, ebx` : subtract ebx from eax, store in eax
- `xor eax, ebx` : exclusive-or eax and ebx and store in eax
- `lea eax, [eax+ebx]` : add eax and ebx, store in eax (add optimization)

Activation Records

Recall the following from the **Subprograms** lecture:



Subprogram callers and callees must *completely agree* on who does what, how, and when. Where do the parameters go? What order? Who changes the stack pointer? These details are encompassed in a protocol known as the *calling convention*.

Calling Conventions

Why do we need to care about calling conventions?

- Calling any function (e.g. `malloc`) requires both the caller and callee to follow the same calling convention.
- The compiler usually worries about the calling conventions. But...
- Reverse engineers need to know about these conventions to understand the assembly code.

The most popular conventions:

- C (`cdecl`): Parameters placed on the stack in right-to-left order. Caller required to clear the stack parameters.
- Microsoft Standard (`stdcall`): called function required to clear the stack parameters.
- Fast Call (`fastcall`): up to two parameters placed in hardware registers. Rest on the stack.
- Microsoft C++ (`thiscall`): the "this" pointer passed through the CX register (x86)

More Calling Conventions

Programmers can choose the calling convention if desired. C++:

```
void _cdecl foo(int x, int y) { return x+y; }
```

Foo disassembled:

```
__Z3fooii proc near

arg_0= dword ptr  8
arg_4= dword ptr  0Ch

push    ebp                     ; save context
mov     ebp, esp                ; set frame pointer
mov     eax, [ebp+arg_4]        ; y
mov     edx, [ebp+arg_0]        ; x
lea     eax, [edx+eax]          ; add and store return value in ax
pop     ebp                     ; restore context
retn                            ; caller clears params
Z3fooii endp
```

Vtable Lookup

```
class A
{
 public:
  virtual int foo(int x) { return x+x; }
};

class B : public A
{
  public:
  virtual int foo(int x)  { return x*x; }
};

int main ()
{
 A* a = new B();
 return a->foo(4);
}
```

Vtable Lookup 2

```
_main proc near
thisparam       = dword ptr -30h
formal_x        = dword ptr -2Ch
var_a           = dword ptr -14h
                  .......
call    ___main
mov     [esp+30h+thisparam], 4
call    ___wrap__Znwj
mov     ebx, eax
mov     eax, ebx
mov     dword ptr [eax], 0
mov     [esp+30h+thisparam], eax
call    __ZN1BC1Ev      ; B::B(void)
mov     eax, ebx
mov     [esp+30h+var_a], eax
mov     eax, [esp+30h+var_a] ; Load & of B data
mov     eax, [eax]      ; The vtable ptr
mov     edx, [eax]      ; The vtable lookup
mov     [esp+30h+formal_x], 4
mov     eax, [esp+30h+var_a]
mov     [esp+30h+thisparam], eax
call    edx             ; Virtual call
                  .......
retn
_main           endp
```

```
class A
{
 public:
  virtual int foo(int x) { return x+x; }
};

class B : public A
{
  public:
  virtual int foo(int x)  { return x*x; }
};

int main ()
{
 A* a = new B();
 return a->foo(4);
}
```

Demo: simple3.cc

Obfuscation

How do you prevent your code from being ripped off? Answer: *Obfuscation*.

To *obfuscate* code is to make it confusing and perplexing to one who may attempt to understand what it does.

Why obfuscate?

- Protect proprietary ideas or algorithms.
- Protect sensitive data.
- Make security measures difficult to circumvent.
- Prevent hacking & malware injections.

- Numerous obfuscation tools available (e.g., Shiva).
- Also numerous tools for undoing the obfuscation (e.g., IDA Pro, QuickUnpack).

Wise Words

"A skilled reverse engineer can fire up IDA-Pro and slice through your application like butter no matter what you do. A packed application can be unpacked and obfuscation only prevents it from [being] a walk in the park."
—Anonymous

Obfuscation Techniques

Obfuscation is a battle between those who want to protect their code and those who want to understand it.

Anti-obfuscation techniques aim to understand obfuscated code. 2 main varieties:

- Anti-static analysis techniques — prevent an analyst from understanding static code.
- Anti-dynamic analysis techniques — prevent an analyst from understanding running code.

Anti-Static Obfuscation Techniques

Anti-static analysis:

- Obscure the executable's starting address.
- Obscure the distinction between code and data.
- Upon returning from a function, return to a different address than the location where the function was called.
- Manipulate control flow at runtime: function pointers, threads, etc.
- Install a bogus exception handler and trap an exception.
- Encrypt the executable file itself (at least some may be decrypted).
- Obscure runtime library dependencies (render `dumpbin`, `ldd`, `objdump` useless).
- Self-modifying code.
- Exploit weaknesses of specific RE tools (e.g. IDA Pro).

Anti-Static Obfuscation (Desynchronization)

```
.text:00401000     xor  eax, eax
.text:00401002     jz short near ptr loc_401009+1 ; ??
.text:00401004     mov ebx, [eax]
.text:00401006     mov [ecx-4], ebx
.text:00401009
.text:00401009  loc_401009:
.text:00401009     call near ptr 0ADFEFFC6h
.text:0040100E     ficom word ptr [eax+59h]
```

```
.text:00401000     xor eax, eax
.text:00401002     jz short loc_40100A
.text:00401004     mov ebx, [eax]
.text:00401006     mov [ecx-4], ebx
.text:00401009     db  0E8h
.text:0040100A
.text:0040100A  loc_40100A:  ; the actual jump target
.text:0040100A     mov  eax, 0DEADBEEFh
.text:0040100F     push eax
.text:00401010     pop ecx
```

Dynamic Obfuscation Techniques

Anti-dynamic analysis:

- Detecting virtualization software (e.g., VMWare). Execute useless/confusing instructions when detected.
  - VM sandboxes useful for observing software execution.
  - Viruses/malware often written in VM sandboxes for easy rollback.
- Detecting instrumentation software (e.g., Process Monitor, Wireshark).
- Detect debuggers & prevent debugging
  - Introduce spurious breakpoints.
  - Clear hardware breakpoints.
  - Raise bogus exceptions, obliterate the stack.
  - Preventing process attachment. (programs fork & attach to themselves).

Advice & Recommended Reading

Advice #1: There is no single magic tool that will reverse engineer everything in the world, and there never will be. Sooner or later, you'll need to know your stuff.

Advice #2: you cannot truly learn to *reverse* engineer until you know how to *forward* engineer. Therefore, focus first on learning to write assembly, CIL, bytecode, etc.

A good starter book:

The Art of Assembly Language 2nd Ed. by Randall Hyde. No Starch Press. 2010.

The IDA Pro Book by Chris Eagle. No Starch Press. 2008.

See also: `http://www.woodmann.com/crackz/`