Software Security

First Aid Kit for Binary Analysis

Mohamed Sabt

Univ Rennes, CNRS, IRISA

2022 / 2023

Part I

Shared Libraries

Refresher – Shared vs Static

- Static Library
  - Code linked into application at compile time (by the static linker ‘ld’).
  - Bigger binaries, faster startup time.
  - Created by ar.

- Shared Library
  - Code linked into application at runtime (by the dynamic linker ‘ld.so’)
  - Smaller binaries, slower startup time.
  - Created by the ‘-shared’ option of the compiler/static linker.

Why Shared Libraries

- Can update without recompiling.
- Can load dynamically with dlopen, and friends.
- More efficient memory usage.

Shared Library Names

- Every shared library has a special name called the ‘’soname’’.
  - The soname has the prefix ‘’lib’’, the name of the library, the ‘’.so’’ extension, followed by a period and a version number.

- Every shared library has a real name.
  - The real name adds to the soname a period, a minor number, another period and the release number.

- The linker name is the name that the compiler uses when requesting a library.
  - The soname without any version number.

How Libraries are Used

- Runtime:
  - Starting up an ELF executable automatically causes all the shared libraries used by the program to be loaded.
  - The list of directories to be searched is stored in the file /etc/ld.so.conf.

- Compilation:
  - When you compile your program, you need to tell the linker about static and shared libraries that you are using.
  - Use the –L option if the library is not installed in a standard place.
  - The current directory is not a standard place.
  - You can see the list of the shared libraries by using ldd.

Environment Variables

- LD_LIBRARY_PATH
  - Directories where libraries should be searched for first, before the standard set of directories.
  - Should only be used for debugging and testing.
  - You do need to bother with this variable when setting ld’s ‘’rpath’’ option, which specifies the runtime library search path of that particular program being compiled.

- LD_DEBUG
  - Trigger verbose information during loading.
  - Setting LD_DEBUG to ‘’help’’ will list all the possible options.

Incompatible Libraries

- When a new version of a library is binary-incompatible with the old one the soname needs to change.
- In C, there are four basic reasons that a library would cease to be binary compatible:
  - The behavior of a function changes so that it no longer meets its original specification,
  - Exported data items change (exception: adding optional items to the ends of structures is okay, as long as those structures are only allocated within the library).
  - An exported function is removed.
  - The interface of an exported function changes

Part II

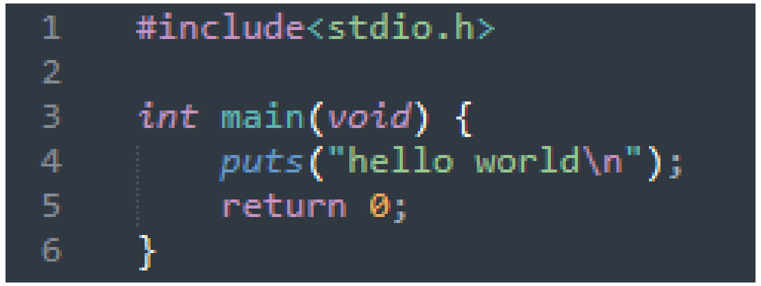
Sections Related to Shared Libraries

ELF Symbols

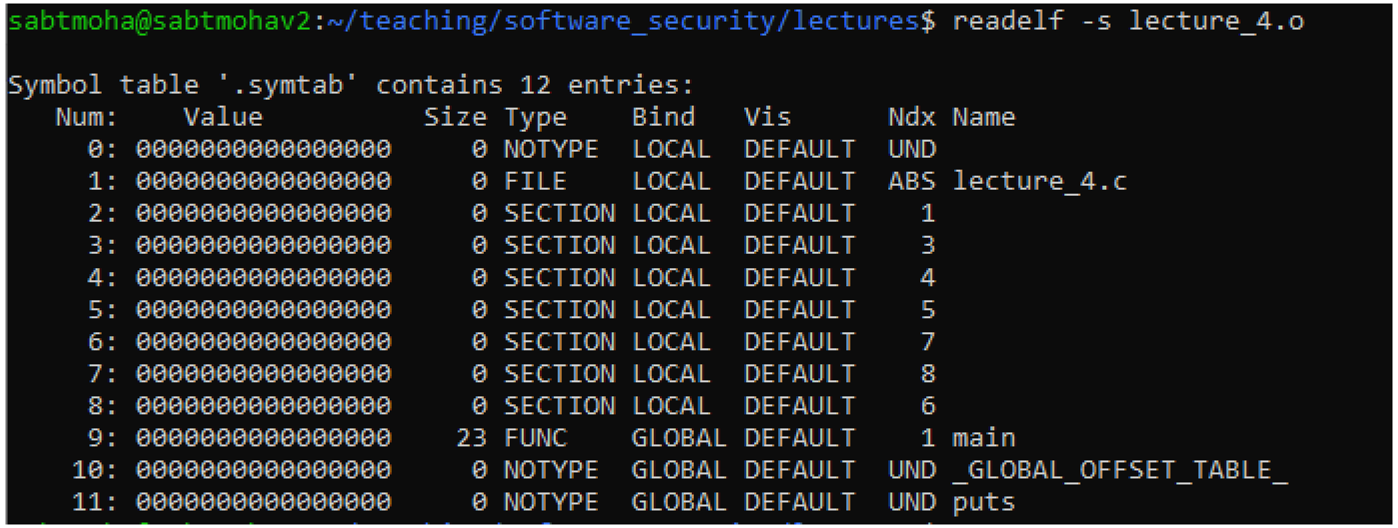
- Symbols are a reference to some type of data or code such as a global variable or function.
  - For instance, the printf() function is going to have a symbol entry that points to it in the dynamic symbol table .dynsym.

- In most ELF files, there exist two symbol tables:
  - .symtab for the static linker symbols: file-scope objects and functions.
  - .dynsym for the dynamic linker symbols: the exported and the imported.
- When a binary is linked against a library, the library needed is stored in a DT_NEEDED entry in the .dynamic section, and the needed functions needed are registered in .dynsym with the following attributes:
  - Value set to 0.
  - Ndx set to UND.

.symtab vs .dynsym 1/2

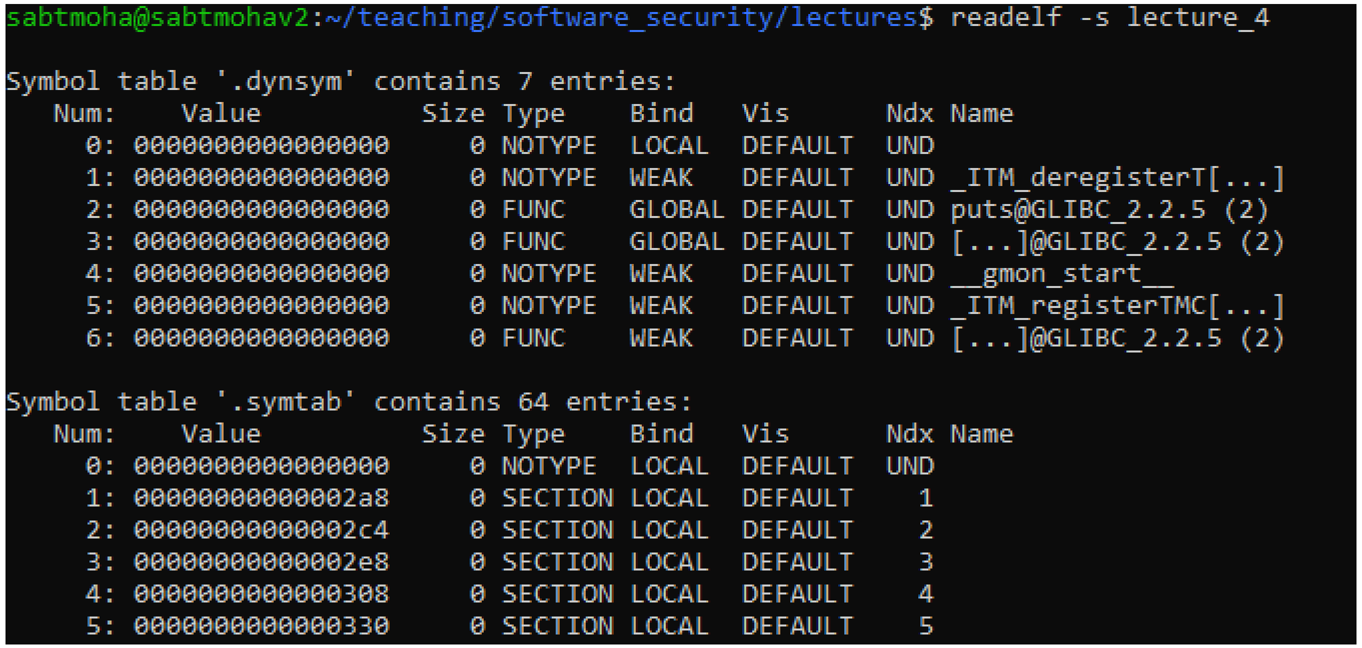


```c
#include<stdio.h>

int main(void) {
    puts("hello world\n");
    return 0;
}
```

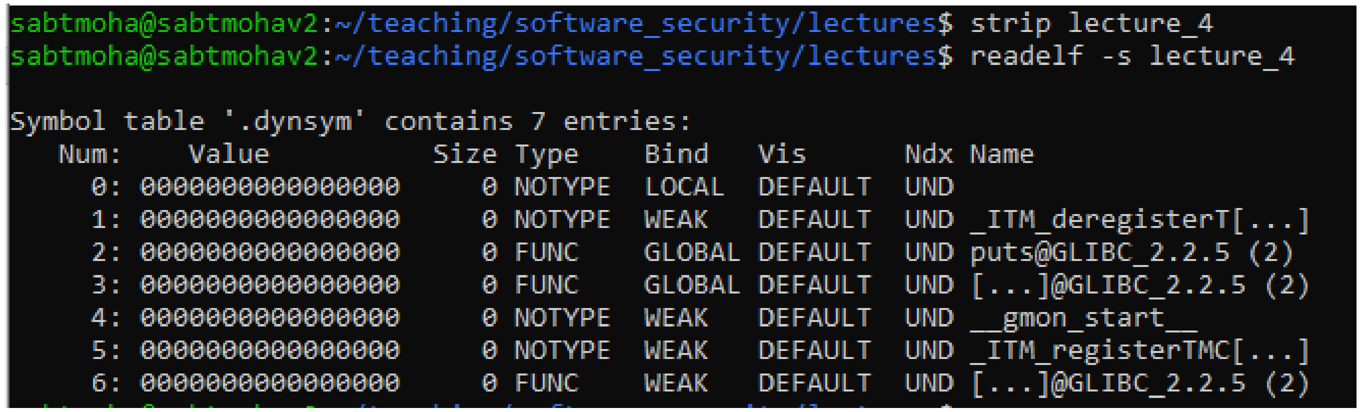


```
sabtmoha@sabtmohav2:~/teaching/software_security/lectures$ readelf -s lecture_4.o

Symbol table '.symtab' contains 12 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS lecture_4.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     9: 0000000000000000    23 FUNC    GLOBAL DEFAULT    1 main
    10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSET_TABLE_
    11: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
```

.symtab vs .dynsym 2/2



```
sabtmoha@sabtmohav2:~/teaching/software_security/lectures$ readelf -s lecture_4

Symbol table '.dynsym' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterT[...]
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.2.5 (2)
     4: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMC[...]
     6: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND [...]@GLIBC_2.2.5 (2)

Symbol table '.symtab' contains 64 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000000002a8     0 SECTION LOCAL  DEFAULT    1
     2: 00000000000002c4     0 SECTION LOCAL  DEFAULT    2
     3: 00000000000002e8     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000308     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000330     0 SECTION LOCAL  DEFAULT    5
```



```
sabtmoha@sabtmohav2:~/teaching/software_security/lectures$ strip lecture_4
sabtmoha@sabtmohav2:~/teaching/software_security/lectures$ readelf -s lecture_4

Symbol table '.dynsym' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterT[...]
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND [...]@GLIBC_2.2.5 (2)
     4: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMC[...]
     6: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND [...]@GLIBC_2.2.5 (2)
```

The .dynamic Section

- The .dynamic section functions as a "road map" for the dynamic linker when loading and setting up an ELF binary for execution.

- The .dynamic section consists of different tags:
  - For instance, tags of type DT_NEEDED inform the dynamic linker about dependencies of the executable.

- The .dynamic section also contains pointers to other important information required by the dynamic linker: .dynsym, .got.plt.

Lazy Binding

- When a binary is loaded into a process for execution, the dynamic linker performs last-minute relocations.
- It resolves references to functions located in shared libraries, where the load address is not yet known at compile time.
- Many of the relocations are typically not done right away when the binary is loaded but are deferred until the first reference to the unresolved location is actually made.
- Lazy binding ensures that the dynamic linker never needlessly wastes time on relocations; it only performs those relocations that are truly needed at runtime.

.plt and .got

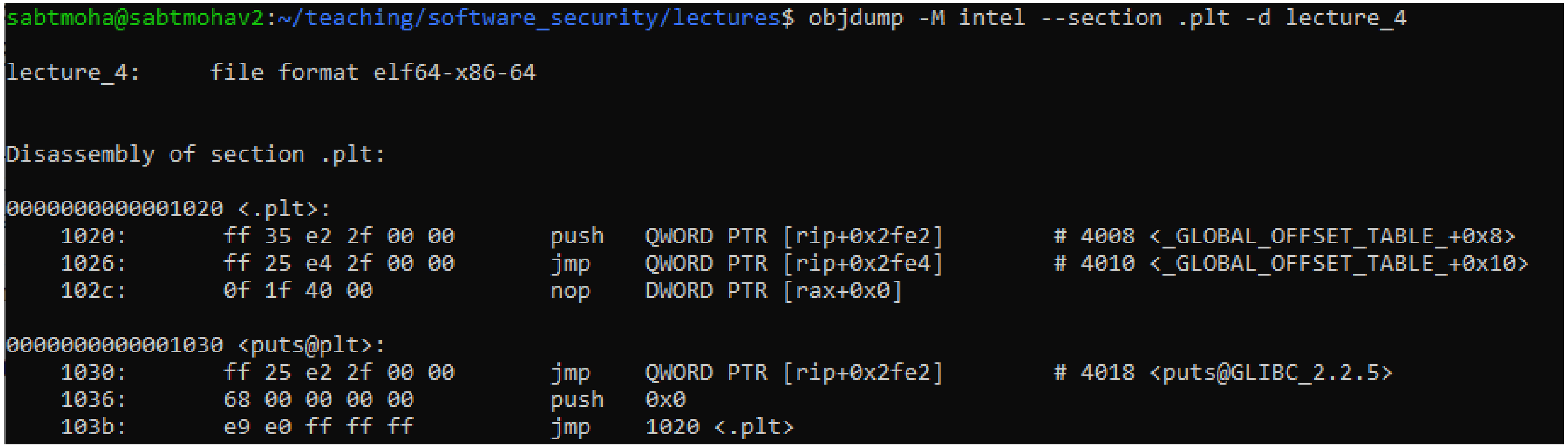
- Lazy binding is implemented with the help of two special sections:
  - *Procedure Linkage Table (.plt)*
  - *Global Offset Table (.got)*

- ELF binaries often contain a separate GOT section called .got.plt for use in conjunction with .plt in the lazy binding process.
  - The .got.plt section is analogous to the regular .got. Historically, they were the same.

- .plt is a code section that contains executable code, just like .text.
- .got.plt is a data section

PLT

- The PLT consists entirely of stubs of a well-defined format, dedicated to directing calls from the .text section to the library location.
- The format of the PLT is as follows:
  - There is a default stub.
  - Function stubs, one per library function, all following the same pattern.



```
sabtmoha@sabtmohav2:~/teaching/software_security/lectures$ objdump -M intel --section .plt -d lecture_4

lecture_4:     file format elf64-x86-64


Disassembly of section .plt:

0000000000001020 <.plt>:
    1020:       ff 35 e2 2f 00 00       push   QWORD PTR [rip+0x2fe2]        # 4008 <_GLOBAL_OFFSET_TABLE_+0x8>
    1026:       ff 25 e4 2f 00 00       jmp    QWORD PTR [rip+0x2fe4]        # 4010 <_GLOBAL_OFFSET_TABLE_+0x10>
    102c:       0f 1f 40 00             nop    DWORD PTR [rax+0x0]

0000000000001030 <puts@plt>:
    1030:       ff 25 e2 2f 00 00       jmp    QWORD PTR [rip+0x2fe2]        # 4018 <puts@GLIBC_2.2.5>
    1036:       68 00 00 00 00          push   0x0
    103b:       e9 e0 ff ff ff          jmp    1020 <.plt>
```

Resolving a Library Function Using the PLT 1/2

- Let’s say you want to call the ‘’puts’’ function, from the libc.
  - Instead of calling it directly (which isn’t possible), you can make a call to the corresponding PLT stub, puts@plt.

- The PLT stub begins with an indirect jump instruction, which jumps to an address stored in the .got.plt section.
  - Initially, before the lazy binding has happened, this address is simply the address of the next instruction in the function stub.
  - Subsequently, the next instruction jumps to the common default stub shared among all PLT function stubs.

- The default stub pushes an identifier (taken from the GOT), identifying the executable itself, and then jumps (indirectly, again through the GOT) to the dynamic linker.

Resolving a Library Function Using the PLT 2/2

- Using the identifiers pushed by the PLT stubs, the dynamic linker figures out that it should resolve the address of puts.

- The dynamic linker then looks up the address at which the puts function is located and plugs the address of that function into the GOT entry associated with puts@plt.

- Thus, the GOT entry no longer points back into the PLT stub, as it did initially, but now points to the actual address of puts.

- For any subsequent calls to puts@plt, the GOT entry already contains the appropriate (patched) address of puts, so that the jump at the start of the PLT stub goes directly to puts without involving the dynamic linker.

Why GOT

- Wouldn’t it be simpler to just patch the resolved library address directly into the code of the PLT stubs?

- .plt section is NOT writable (Principle: W^X).
  - In other words, this extra layer of indirection allows you to avoid creating writable executable sections.
  - Indeed, GOT is a data section.

- Data references go directly through the GOT, without the intermediate step of the PLT.
  - .got is for references to data items, while .got.plt is dedicated to storing resolved addresses for library functions accessed via the PLT.

Part III

Disassembly

Basic Binary Analysis Tools

- ldd: find out on which shared objects a binary depends and where (if anywhere) these dependencies are on your system.
- xxd: a hex-dumping program.

- readelf: parse ELF files.
- nm: display symbols.
- objdump: examine instruction-level behavior.

- strings: display any printable character strings in a file.
- strace & ltrace: show the system and library calls.

- gdb: dynamic analysis.

Static vs Dynamic

- When people say “disassembly,” they usually mean *static disassembly*, which involves extracting the instructions from a binary without executing it.

- *Dynamic disassembly*, more commonly known as *execution tracing*, logs each executed instruction as the binary runs.

Static Analysis Steps

- The goal of every static disassembler is to translate *all* code in a binary into a form that a human can read or a machine can process

- To achieve this goal, static disassemblers need to perform the following steps:
    1. Load a binary for processing, using a binary loader like libelf.
    2. Find all the machine instructions in the binary.
    3. Disassemble these instructions into a human- or machine-readable form.

- Unfortunately, step 2 is often very difficult in practice, resulting in disassembly errors.

- There are two major approaches to static disassembly, each of which tries to avoid disassembly errors in its own way:
    - *linear disassembly.*
    - *recursive disassembly*.

Linear Disassembly

- It iterates through all code segments in a binary, decoding all bytes consecutively and parsing them into a list of instructions.
  - Many simple disassemblers, including objdump, use this approach.

- The risk of using linear disassembly is that not all bytes may be instructions.
  - If disassemblers accidentally parse this *inline data* as code, they may encounter invalid opcodes, or even valid opcodes.

- In practice, linear disassemblers such as objdump are safe to use for disassembling ELF binaries compiled with recent versions of compilers such as gcc or LLVM’s clang.
  - The x86 versions of these compilers don’t typically emit inline data.

Recursive Disassembly

- Recursive disassembly starts from known entry points into the binary (such as the main entry point and exported function symbols)
  - From there recursively follows control flow (such as jumps and calls) to discover code.

- This allows recursive disassembly to work around data bytes
  - The downside of this approach is that not all control flow is so easy to follow.
  - For instance, it’s often difficult, if not impossible, to statically figure out the possible targets of indirect jumps or calls.

- Recursive disassembly is the de facto standard in many reverse-engineering applications, such as IDA Pro and Ghidra.

Linear vs Recursive

- For benign x86 ELF binaries, linear disassembly is a good choice because it will yield both a complete and accurate disassembly:
  - Such binaries typically don’t contain inline data that will throw the disassembler off.
  - The linear approach won’t miss code because of unresolved indirect control flow.
- In cases where disassembly correctness is paramount, even at the expense of completeness, you can use *dynamic disassembly*.
- If inline data or malicious code is involved, it’s probably a better idea to use a recursive disassembler that’s not as easily fooled into producing bogus output as a linear disassembler is.

Dynamic Analysis

- Dynamic disassemblers simply dump instructions (and possibly memory/register contents) as the program executes.
    - It is also known as *execution tracers* or *instruction tracers*

- When execution reaches a particular address, you can be absolutely sure there’s an instruction there, so dynamic disassembly doesn’t suffer from inaccuracy.

- The main downside of this approach is the *code coverage problem*:
    - The fact that dynamic disassemblers don’t see all instructions but only those they execute.

Code Coverage Problem

- The main disadvantage of dynamic disassembly is code coverage:
  - The analysis only ever sees the instructions that are actually executed during the analysis run.
- If any crucial information is hidden in other instructions, the analysis will never know about it.
- It is difficult and time-consuming to find the correct inputs to cover every possible program path.
  - Dynamic disassembly will almost never reveal all possible program behavior.
- There are several methods you can use to improve the coverage of dynamic analysis tools, though in general none of them achieves the level of completeness provided by static analysis.

Strategies 1 : Test Suites

- Running the analyzed binary with manually-chosen test inputs.
- The downside of this approach is that a ready-made test suite isn’t always available, for instance, for proprietary software or malware.

Strategies 2 : Fuzzing

- Fuzzing is to try to automatically generate inputs to cover new code paths in a given binary.

- Broadly speaking, fuzzers fall into two categories based on the way they generate inputs.
    - Generation-based fuzzers: These generate inputs from scratch (possibly with knowledge of the expected input format).
    - Mutation-based fuzzers: These fuzzers generate new inputs by mutating known valid inputs in some way, for instance, starting from an existing test suite.

- The success and performance of fuzzers depend greatly on the information available to the fuzzer.
    - For instance, it helps if source information is available or if the program's expected input format is known.

Strategies 3 : Symbolic Execution

- Symbolic execution allows you to execute an application not with *concrete values* but with *symbolic values*.

- Symbolic execution is essentially an emulation of a program, where all or some of the variables (or register and memory states) are represented using such symbols.

- The symbolic execution also computes *path constraints*, which are just restrictions on the concrete values that the symbols could take.

- The key point is that *given the list of path constraints, you can check whether there's any concrete input that would satisfy all these constraints.*
  - There are special programs, called *constraint solvers*, that check, given a list of constraints, whether there's any way to satisfy these constraints.

- Solving a set of path constraints is computationally intensive; if you don't take care, your symbolic execution approach can easily become unscalable.

Part IV

Binary Analysis

Structuring Code

- Large unstructured heaps of disassembled instructions are nearly impossible to analyze.
  - Most disassemblers structure the disassembled code in some way that's easier to analyze.

- Techniques:
  - Compartmentalizing: By breaking the code into logically connected chunks, it becomes easier to analyze what each chunk does.
  - Revealing control flow. How the chunks are related to each other.

Functions Detection

- Functions are the fundamental building blocks used to group logically connected pieces of code.

- Most disassemblers make some effort to recover the original program’s function structure and use it to group disassembled instructions.

- For binaries with symbolic information, function detection is trivial;
  - The symbol table specifies the set of functions, along with their names, start addresses, and sizes.

- Source-level functions have no real meaning at the binary level, so their boundaries may become blurred during compilation.
  - Bits and pieces of the function might be scattered throughout the code section,
  - Chunks of code may even be shared between functions (known as *overlapping code blocks*).

Functions Signatures

- *Function signatures:* patterns of instructions often used at the start or end of a function.
- Typically, signature-based function detection algorithms start with a pass over the disassembled binary to locate functions that are directly addressed by a call instruction.
- Function signature patterns include well-known *function prologues* (instructions used to set up the function's stack frame) and *function epilogues* (used to tear down the stack frame).
- Optimized functions may not have well-known function prologues or epilogues at all, making them impossible to recognize using a signature-based approach

Control-Flow Graphs

- To organize the internals of each function, disassemblers and binary analysis frameworks use another code structure, called a *control-flow graph (CFG).*

- CFGs represent the code inside a function as a set of code blocks, called *basic blocks*, connected by *branch edges.*

Call Graphs

- Call edges are not part of a CFG because they target code outside of the function.

- *Call graphs* are similar to CFGs, except they show the relationship between call sites and functions rather than basic blocks.

- CFGs show you how control may flow within a function, while call graphs show you which functions may call each other.

Structuring Data

- Disassemblers automatically identify various types of code structures to help your binary analysis efforts.

- Automatic data structure detection in stripped binaries is a notoriously difficult
  - Aside from some research work, disassemblers generally don’t even attempt it.

- Primitive types can sometimes be inferred by the registers they’re kept in or the instructions used to manipulate the data.
  - For instance, if you see a floating-point register or instruction being used, you know the data in question is a floating-point number.
  - If you see lodsb (*load string byte*) or stosb (*store string byte*), it’s likely manipulating a string.

- To facilitate structuring data manually, IDA Pro (or Ghidra) allows you to define your own composite types (which you have to infer by reversing the code) and assign these to data items.

Decompilation

- *Decompilers* are tools that attempt to “reverse the compilation process.”
  - They typically start from disassembled code and translate it into a higher-level language, usually a form of C-like pseudocode.
- Decompilers are useful when reversing large programs because decompiled code is easier to read than lots of assembly instructions.
- Decompilers are limited to manual reversing because the decompilation process is too error-prone to serve as a reliable basis for any automated analysis.

Optimization 1/2

- 📀 Unfortunately, optimized code is usually significantly harder to accurately disassemble (and thus analyze) than unoptimized code.

- Inlining: compilers often merge small functions into the larger functions calling them, to avoid the cost of the call instruction.

- 📀 Optimized calling conventions make function detection significantly less accurate.

- 📀 At higher optimization levels, compilers often emit padding bytes between functions and basic blocks to align them at memory addresses where they can be most efficiently accessed.

Optimization 2/2

- 💿 Compilers may “unroll” loops to avoid the overhead of jumping to the next iteration.

- Optimized code may use the same base register to index different arrays at the same time, making it difficult to recognize them as separate data structures.