Exploiting Software Vulnerabilities

Program Binary Analysis

Universidad Zaragoza
1542

Dept. of Computer Science and Systems Engineering
University of Zaragoza, Spain

Course 2022/2023

Master's Degree in Informatics Engineering

University of Zaragoza

Seminar A.22, Ada Byron building

Outline

1. Introduction to Program Binary Analysis

2. Static Analysis Techniques

3. Dynamic Analysis Techniques

Outline

Universidad Zaragoza

Introduction

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("hello world!\n");
    return 0;
}
```

```asm
push    ebp
mov     ebp, esp
and     esp, -16
sub     esp, 16
call    ___main
mov     DWORD PTR [esp], OFFSET FLAT:LC0
call    _puts
mov     eax, 0
leave
ret
```

- **Programs are written in text**
  - Both source code and assembly!
  - **Character sequences (bytes)**
  - Difficult to work with (for humans, not for machines)
  - We need some **structured representation**

Universidad Zaragoza

Introduction

Program Analysis

Automatically reason and derive properties about the behavior of computer programs

Approaches

- **Static Program Analysis**
  - Without running the program
  - The abstract model of the program is obtained and (symbolically) executed
  - Analysis performed through the abstract model
  - **Examples**: CFA, DFA, symbolic execution, . . .
- **Dynamic Program Analysis**
  - Running the program on some chosen inputs
  - Traces are collected and then analyzed
  - Analysis performed through these concrete executions
  - **Examples**: software testing, taint analysis, concolic execution. . .

Universidad Zaragoza

Introduction

Input program formats for analysis

- **Abstract model**: all unnecessary information for analysis have been removed. Only the necessary information remains
- **Source code**: Keep track of high-level, human-readable information about the program (variables, types, functions, etc.)
- **Bytecode**: may vary depending on the bytecode considered, but keep a record of little high-level information about the program, such as types and functions. The programs are unstructured
- **Binary file**: just keep track of statements in an unstructured way (no for-loop, no clear argument passing in procedures, etc). No type, no names. The binary file can include meta-data that can be useful for analysis (symbols, debug, etc.)
- **Memory dump**: Pure assembler instructions with a full memory state of the current execution. We no longer have the meta-data of the executable file

Binary code is the closest format of what will be executed!

Introduction

Binary code vs. source code

What you code is not what you execute!

We want to analyze binary code. It can come as:

- an executable file,
- an object file,
- a dynamic library,
- a firmware,
- a memory dump,
- …

We do not trust to obtain the corresponding high-level source code

Universidad Zaragoza

Why should we analyze binary programs?

- Lack of high-level source code
- Low-level assembly code embedded in source code
- Legacy code
- Commercial Off-The-Shelf software (COTS)
- App stores (for mobile phones and tablets)
- Malware (or other “hostile” programs)
- Technology forecast
- Mistrust in the compilation chain
- C compiler possibly buggy
- Checking for low-level bugs (e.g., exploiting a stack buffer overflow)
- Errors with strong hardware interconnection

Introduction

Understanding papers on Program Analysis

> *For those who keep track of such things, checkers in the research system typically traverse program paths (flow-sensitive) in a forward direction, going across function calls (inter-procedural) while keeping track of call-site-specific information (context-sensitive) and toward the end of the effort had some of the support needed to detect when a path was infeasible (path-sensitive).*

Note these terms

- **Flow-(in)sensitive**
- **Inter-(intro)procedural**
- **Context-(in)sensitive**
- **Path-(in)sensitive**

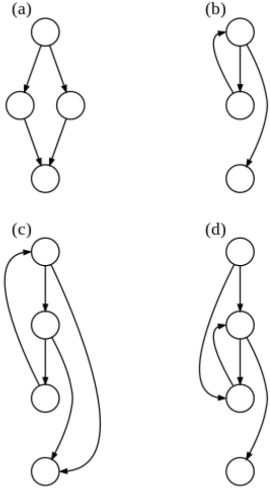
Further reading: *A few billion lines of code later: using static analysis to find bugs in the real world*. Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, Dawson Engler. *Communications of the ACM*, vol. 53, iss. 2, pp. 66-75 (February 2010). doi: 10.1145/1646353.1646374

Universidad Zaragoza

Outline

Static Analysis Techniques

Control-Flow Graphs



- **Control flow within a function**
- **Nodes: basic blocks**
  - Sequence of consecutive program instructions that have an entry point (first instruction executed) and an exit point (last instruction executed)
  - Entry and exit blocks
- **Edge**: control flows from A to B

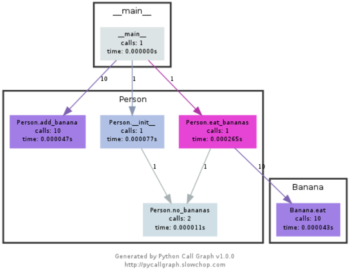
Applications

- Compiler optimizations
- Data-flow analysis (taint analysis)
- Behavioral-based monitors

Credits: https://en.wikipedia.org/wiki/Control_flow_graph

Static Analysis Techniques

Call Graphs

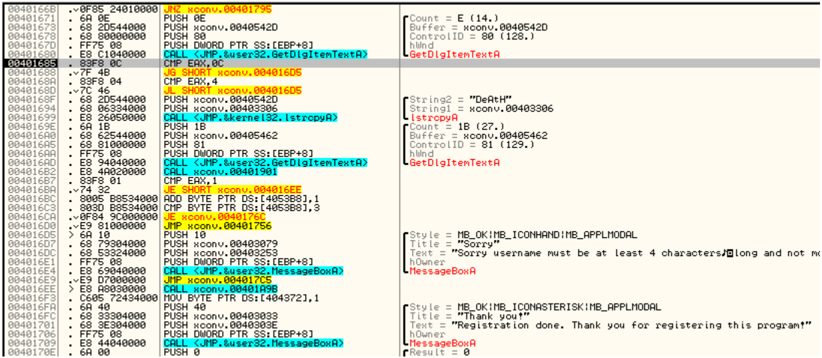


- **Interprocedural CFG**. **Information flow between functions**
- **Nodes**: functions
- **Edge**: A could call B
- Types: static, dynamic (record of a program execution)
- Application: find procedures never called
- **Available tools for automatic generation of call-graphs**

Credits: `https://en.wikipedia.org/wiki/Call_graph`

Static Analysis Techniques

Disassembling



- Generally speaking, **read `PUSH EAX` instead of `0x50`**
- **Lots of tools** see `https://en.wikibooks.org/wiki/X86_Disassembly/Disassemblers_and_Decompilers`
  - Win32Dasm
  - OllyDBG (also debugger)
  - IDA Pro (also debugger)
  - r2 (also debugger)

Universidad Zaragoza

Static Analysis Techniques
Disassembling

Main challenges

- **Variable-length instruction sets**: overlapping instructions
- **Mixed data and code**: misclassifying data as instructions
- **Indirect jumps**: Any location can be the start of an instruction!
- **Start of functions**: when the calls are indirect
- **End of functions**: when there is no dedicated return instruction exists
  - Handwritten assembly code may not conform to standard call conventions
- **Code compression**: the code of two functions overlaps
- **Self-modifying code**

Static Analysis Techniques

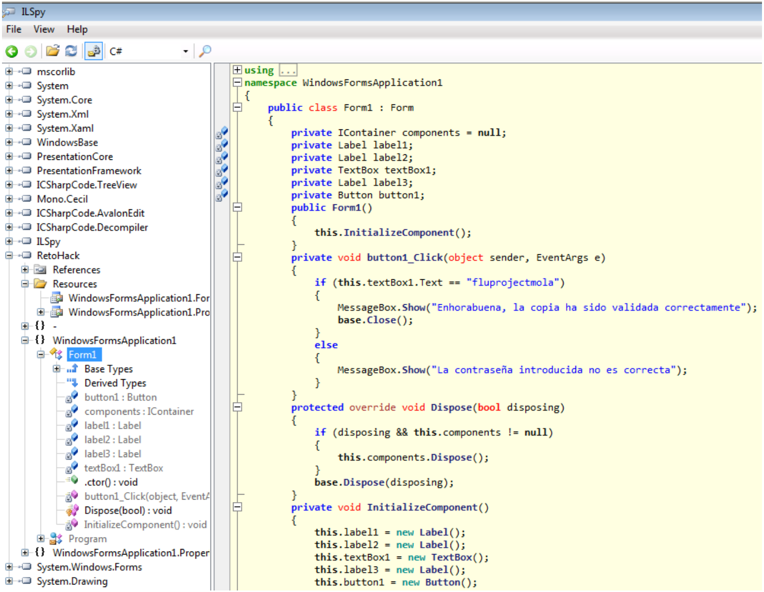
Decompilation – example

```
int __stdcall sub_40162C(HWND hDlg, int a2, int a3, int a4){
  HICON v4; // eax@2
  UINT v5; // eax@5

  switch ( a2 )  {
    case 272:
      v4 = LoadIconA(hInstance, (LPCSTR)0x64);
      SendMessageA(hDlg, 0x80u, 1u, (LPARAM)v4);
      break;
    case 273:
      if ( a3 == 126 ) {
        v5 = GetDlgItemTextA(hDlg, 128, dword_40542D, 14);
        if ( (signed int)v5 > 12 || (signed int)v5 < 4 ) {
          MessageBoxA(hDlg,"Sorry username must be at least 4
characters\r\nlong
and not more than 12 characters.", "Sorry", 0x10u);
        } else {
          lstrcpyA(dword_403306, dword_40542D);
          GetDlgItemTextA(hDlg, 129, byte_405462, 27);
          if ( sub_401901() == 1 ) {
            sub_401A9B();
            byte_404372 = 1;
            MessageBoxA(hDlg, "Registration done. Thank you for registering
this
program!", "Thank you!", 0x40u);
            EndDialog(hDlg, 0);
            EnableWindow(dword_403363, 0);
            SetWindowTextA(
              dword_4054A7,
              "X-Convertor v1.0  2005 by TDC and BoR0\r\n\n
Coded by\t: TDC and BoR0\r\nVersion\t\t: 1.0\r\nRelease
date\t: 18-08-2005\r\n \r\nX-Convertor converts up to 4KB
each convert.\r\n \r\nRegistered version. Thank you.\r\n");
            lstrcatA(byte_403330, dword_403306);
            SetWindowTextA(dword_4054AB, byte_403330);
          }
          else {
           ++byte_4053B8;
           if ( byte_4053B8 == 3 ) {
             MessageBoxA(hDlg, "Your serial is not correct",
                               "Sorry", 0x10u);
             byte_4053B8 = 0;
             EndDialog(hDlg, 0);
           } else {
             MessageBoxA(hDlg, "Your serial is not correct",
                                "Sorry", 0x10u);
           }
          }
        }
      } else {
        if ( a3 == 127 ) {
          byte_4053B8 = 0;
          EndDialog(hDlg, 0);
        }
      }
      break;
    case 16:
      byte_4053B8 = 0;
      EndDialog(hDlg, 0);
      break;
  }
  return 0;
}
```

Static Analysis Techniques

Decompilation



Static Analysis Techniques

Decompilation

Main challenges

- **Disassembly**: **first step of any decompiler!**
- **Target language**: assembly code may not correspond to any (correct) source code
- **Library functions**
- **Compiler-dependent instruction equivalences**
  - `int a= 0` → `mov eax, [a]; xor eax, eax`
- **Target architecture artifacts**: unnecessary jumps-to-jumps
- **Structured control flow**
- **Compiler optimizations**: loop unrolling, shifts, adds, ...
- **Loads/stores**: operations on arrays, records, pointers, and objects
- **Self-modifying code**: normally, the segment code will be unchanged, although there are programs that modify themselves!

Static Analysis Techniques

Data Flow Analysis

- Analyze the **effect of each instruction**
- Compose instruction effects to derive information at the basic block boundaries
- Framework for **providing facts about programs.** Based on all paths through program (including also infeasible paths)
- **Derive information about the dynamic behavior of a program by examining the code statically**

Useful for. . .

- **Program debugging**: what definitions (of variables) can reach a program point?
- **Program optimizations**: constant folding, copy propagation, elimination of common sub-expressions, etc.

Consider the statement $a = b + c$

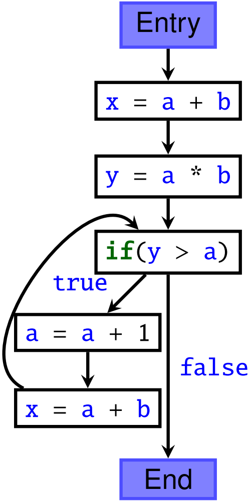
Statement effects

- **Uses variables** (b, c)
- **"Kills" a previous definition** (old value of a)
- **New definition** (a)

- **Compose effect of statements → effect of a basic block**
  - *Locally exposed usage*: usage of a data item that is not preceded in the basic block by a data item definition
  - Any definition of a data item *kills* all definitions of the same data item that reach the basic block
  - *Locally available definition*: last definition of the data item in the basic block

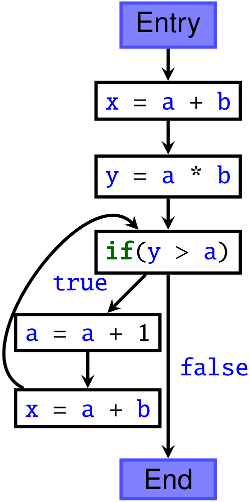
Static Analysis Techniques
Data Flow Analysis

- **Facts**
  - $a + b$ is available
  - $a * b$ is available
  - $a + 1$ is available
- Let's calculate the facts that hold for each program point!



Static Analysis Techniques
Data Flow Analysis

Statement	Gen	Kill
`x = a + b`	$a + b$	
`y = a * b`	$a * b$	
`y > a`		
`a = a + 1`		$a + b$
		$a * b$
		$a + 1$



- *Forward* vs. *backward*: data flow from in to out (vs. from out to in)
- *Must* vs. *may*: at joint points, just keep facts that hold on all paths (vs. any path) that are joined

	Must	**May**
Forward	Available expressions	Reaching definitions
Backward	Very busy expressions	Live variables

Limitations

- **Data-Flow Analysis is good for analyzing local variables**
  - **What happens to values stored in the heap?**
  - Not modeled on traditional data flow
- In general, **it is difficult to analyze pointers**. Suppose `*x = p`
  - *Assume all data flow facts are killed*
  - Or assume writing via `x` can affect any variable whose address has been taken

Static Analysis Techniques
Symbolic Execution

- **Allows us to scale and model all possible executions of a program**
- Concrete vs. symbolic execution
  - **Tests work, but each test only explores one possible execution path**
- **Symbolic execution generalizes testing**
  - Allows unknown *symbolic* variables in evaluation
  - **Checks the feasibility of the program paths**

Challenges

- **Path explosion**
- **Modeling statements and environments**
- **Constraint resolution**

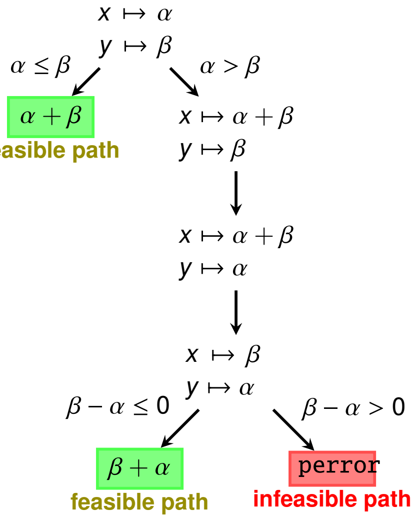
Further reading: Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. *A Survey of Symbolic Execution Techniques*. ACM Comput. Surv. 51, 3, Article 50 (July 2018), 39 pages. doi: 10.1145/3182657

Static Analysis Techniques

Symbolic Execution

```c
int f(int x, int y)
{
    if(x > y)
    {
        x = x + y;
        y = x - y;
        x = x - y;
        if(x - y > 0)
            perror("Error!");
    }

    return x + y;
}
```



How to decide which branches are feasible?

Combine the path condition with the branch condition and ask an SMT solver!

Static Analysis Techniques

Symbolic Execution – example: bug finding

Catch the error! What value (or values) triggers it?

```c
int bar(int i)
{
    int j = 2*i;
    i++;
    i = i*j;
    if (i < 1)
        i = -i;

    i = j/i;
    return i;
}
```

False branch condition

$$i = (i_{in} + 1)2i_{in}$$
$$(i_{in} + 1)2i_{in} \geq 1$$

True branch condition

$$i = -(i_{in} + 1)2i_{in}$$
$$(i_{in} + 1)2i_{in} < 1$$

Universidad Zaragoza

Symbolic Execution – example: bug finding

Catch the error! What value (or values) triggers it?

```
int bar(int i)
{
    int j = 2*i;
    i++;
    i = i*j;
    if (i < 1)
        i = -i;

    i = j/i;
    return i;
}
```

False branch condition

$i = (i_{in} + 1)2i_{in}$
$(i_{in} + 1)2i_{in} \geq 1$

True branch condition

$i = -(i_{in} + 1)2i_{in}$
$(i_{in} + 1)2i_{in} < 1$

Division by zero creates problems...

Universidad Zaragoza

Static Analysis Techniques

Symbolic Execution – example: bug finding

Catch the error! What value (or values) triggers it?

```c
int bar(int i)
{
    int j = 2*i;
    i++;
    i = i*j;
    if (i < 1)
        i = -i;

    i = j/i;
    return i;
}
```

False branch condition	$i = (i_{in} + 1)2i_{in}$ $(i_{in} + 1)2i_{in} \geq 1$
True branch condition	$i = -(i_{in} + 1)2i_{in}$ $(i_{in} + 1)2i_{in} < 1$

Division by zero creates problems. . .

False branch is always safe

$(i > 0, \forall i_{in} | (i_{in} + 1)2i_{in} \geq 1)$

What about the true branch?

Catch the error! What value (or values) triggers it?

```c
int bar(int i)
{
    int j = 2*i;
    i++;
    i = i*j;
    if (i < 1)
        i = -i;

    i = j/i;
    return i;
}
```

False branch condition	$i = (i_{in} + 1)2i_{in}$ $(i_{in} + 1)2i_{in} \geq 1$
True branch condition	$i = -(i_{in} + 1)2i_{in}$ $(i_{in} + 1)2i_{in} < 1$

Division by zero creates problems. . .

False branch is always safe

$(i > 0, \forall i_{in} | (i_{in} + 1)2i_{in} \geq 1)$

What about the true branch?

$-(i_{in} + 1)2i_{in} = 0$

Static Analysis Techniques

Symbolic Execution – example: bug finding

Catch the error! What value (or values) triggers it?

```c
int bar(int i)
{
    int j = 2*i;
    i++;
    i = i*j;
    if (i < 1)
        i = -i;

    i = j/i;
    return i;
}
```

False branch condition	$i = (i_{in} + 1)2i_{in}$ $(i_{in} + 1)2i_{in} \geq 1$
True branch condition	$i = -(i_{in} + 1)2i_{in}$ $(i_{in} + 1)2i_{in} < 1$

Division by zero creates problems...
False branch is always safe
$(i > 0, \forall i_{in} | (i_{in} + 1)2i_{in} \geq 1)$
What about the true branch?
$-(i_{in} + 1)2i_{in} = 0 \rightarrow i_{in} = -1, i_{in} = 0$

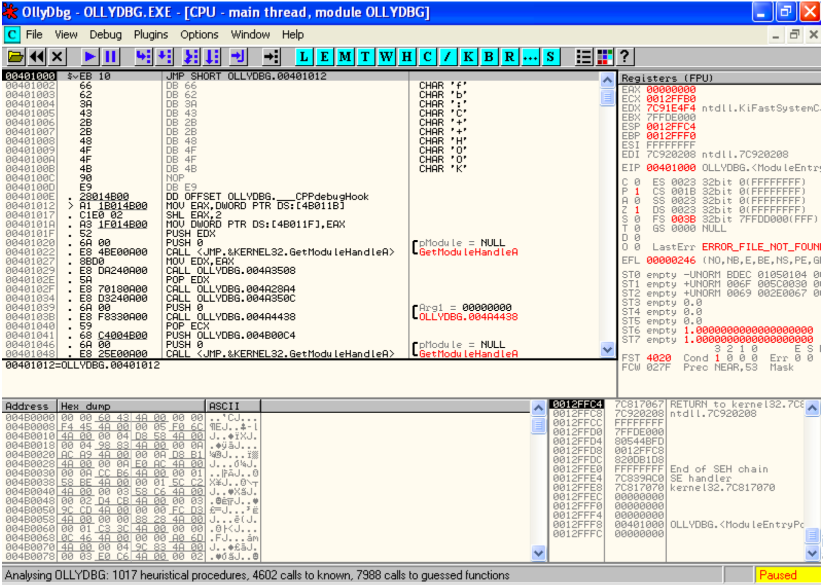
Outline

Dynamic Analysis Techniques
Debugging

- **Execute program instructions with special software**: *debuggers*
  - We can see the values of each register of the CPU, the stack, the memory, etc.
- Source code vs. binary debugging
- **Breakpoints**: stops execution when reached
  - Software breakpoints (memory)
  - Hardware breakpoints
  - On execute, read, or write operations
- Step into / step onto

Dynamic Analysis Techniques

Debugging (example: `OllyDBG`)



Dynamic Analysis Techniques
Fuzzing

Roughly speaking, “fuzzing means…” (quoting Iñaki Rodríguez-Gastón)

Dynamic Analysis Techniques
Fuzzing

Roughly speaking, “fuzzing means…” (quoting Iñaki Rodríguez-Gastón)

- **Black-box approach** (at the beginning): no prior knowledge of the internal aspects of the program
  - **Evolved to a white-box approach**: state-of-the-art fuzzers “learn” from program behavior
- **Many abnormal inputs (unexpected, invalid, or random data) are given to the application**
- **The application is monitored for any signs of error**
  - Unexpected behavior
  - Crashes
    - Buffer overflow
    - Integer overflow
    - Memory corruption errors
    - Format string bugs

Universidad Zaragoza

Charlie Miller's "five lines of Python" dumb fuzzer

- **Found vulnerabilities in PDF readers and MS Powerpoint**

```
numwrites = random.randrange(math.ceil((float(len (buf)) / FuzzFactor))) + 1
for j in range (numwrites):
     rbyte = random.randrange(256)
     rn = random.randrange(len(buf))
     buf[rn] = "%c"%(rbyte);
```


Dynamic Analysis Techniques

Fuzz Testing

An example: HTTP GET requests

- Standard HTTP GET request: `GET /index.html HTTP/1.1`
- **Anomalous requests**
  - `AAAAAA...AAAA /index.html HTTP/1.1`
  - `GET ///////index.html HTTP/1.1`
  - `GET %n%n%n%n%n%n.html HTTP/1.1`
  - `GET /AAAAAAAAAAAAA.html HTTP/1.1`
  - `GET /index.html HTTTTTTTTTTTTTP/1.1`
  - `GET /index.html HTTP/1.1.1.1.1.1.1.1`
  - etc.

Types of fuzzers

- **Mutation-based fuzzing**
- **Generational-based fuzzing**

Dynamic Analysis Techniques

Taint analysis

Measure what is the influence of the input data in the application

- Data comes from tainted sources (any external input) and ends up in tainted sinks
- *Flow* from X to Y: **an operation that uses** X **to derive a value** Y
- **Tainted value: if the source of the value** X **is not trusted** (e.g., user-supplied string)

Taint Propagation

- **Object** X **tainted object** Y
- **Taint operator** t: $X \mapsto t(Y)$
- **A taint operator is transitive**: $X \mapsto t(Y)$ and $Y \mapsto t(Z)$, then $X \mapsto t(Z)$

Dynamic Analysis Techniques

Taint analysis

Main challenges

- **Tainted addresses**
  - Distinguishing between memory addresses and cells is not always appropriate
  - Taint granularity is important (bit, byte, word, etc.)
- **Undertainting**
  - Dynamic taint analysis does not handle some types of information flow correctly
- **Overtainting**
  - Deciding when to introduce taint is often easier than deciding when to remove it
- **Time of detection vs. time of attack**
  - When used for attack detection, dynamic taint analysis may generate an alert too late

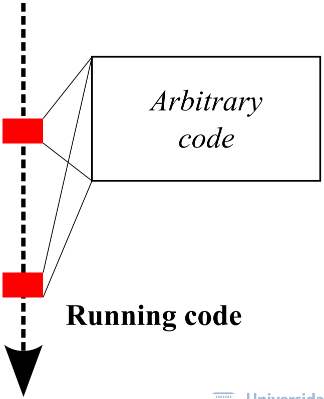
Universidad Zaragoza

Dynamic Analysis Techniques

Dynamic Binary Instrumentation

Adding arbitrary code during binary execution

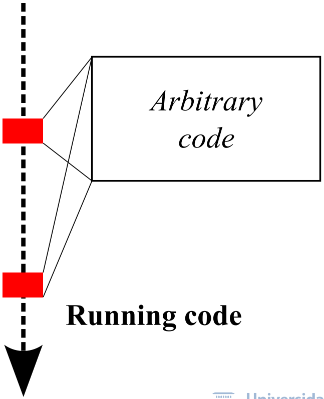
- What insert? → **instrumentation function**
- Where? → **addition places**



Dynamic Analysis Techniques
Dynamic Binary Instrumentation

Adding arbitrary code during binary execution

- What insert? → **instrumentation function**
- Where? → **addition places**



Advantages

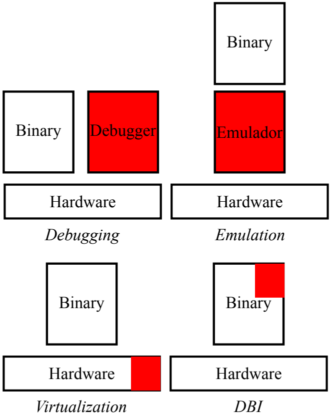
- Programming language independent
- We can instrument proprietary software
- No need to recompile/relink each time
- Allows you to instrument a process already running

Main disadvantage

- **Overhead ⇒⇓ performance**

Dynamic Analysis Techniques

Placing DBI in the context of dynamic analysis



- **No transformation of the program file**
- **Full control over execution**
- **No need for architectural support**

Credits: J-Y. Marion, D. Reynaud *Dynamic Binary Instrumentation for Deobfuscation and Unpacking*. *DeepSec*, 2009

Exploiting Software Vulnerabilities

Program Binary Analysis

Universidad Zaragoza
1542

Dept. of Computer Science and Systems Engineering
University of Zaragoza, Spain

Course 2022/2023

Master's Degree in Informatics Engineering

UNIVERSITY OF ZARAGOZA

Seminar A.22, Ada Byron building