

# Exploiting Software Vulnerabilities

## Program Binary Analysis

© All wrongs reversed – under CC-BY-NC-SA 4.0 license



**Universidad**  
Zaragoza

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

Course 2020/2021

**Master's Degree in Informatics Engineering**

UNIVERSITY OF ZARAGOZA

Google Meet



# Outline

- 1** Introduction to Program Binary Analysis
- 2** Static Analysis Techniques
- 3** Dynamic Analysis Techniques

# Outline

- 1** Introduction to Program Binary Analysis
- 2 Static Analysis Techniques
- 3 Dynamic Analysis Techniques

# Introduction

```
#include <stdio.h>
```

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

```
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!
- **Sequences of characters (bytes)**
- Hard to work with (for humans, not for machines)
- We need some **structured representation**

# Introduction

## Program Analysis

**Automatically reasoning and deriving properties about the behavior of computer programs**

### Approaches

#### ■ Static Program Analysis

- Without executing the program
- Abstract model of the program is obtained and (symbolically) executed
- Analysis done through the abstract model
- **Examples:** CFA, DFA, concolic execution, ...

#### ■ Dynamic Program Analysis

- Executing the program on chosen inputs
- Traces are collected and then analyzed
- Analysis done through these concrete executions
- **Examples:** software testing, taint analysis, ...

# Introduction

## Input program formats for analysis

- **Abstract model:** all unnecessary information for the analysis have been removed. Only necessary information remains
- **Source code:** Keep track of high-level and human-readable information about the program (variables, types, functions, etc.)
- **Bytecode:** may vary depending on the bytecode considered, but keep track of fewhigh-level information about the program such as types and functions. Programs are unstructured, through
- **Binary file:** only keep track of the instructions in an unstructured way (no for-loop, no clear argument passing in procedures, etc). No type, no naming. Binary file may enclose meta-data that might be helpful for analysis (symbols, debug, etc.)
- **Memory dump:** Pure assembler instructions with a full memory state of the current execution. We do not have anymore 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 don't rely on getting the corresponding high-level source code**

# Introduction

## Motivations

### Why analyze binary programs?

- Lack of high-level source code
- Low-level assembly code built-in the source code
- Legacy code
- Commercial Off-the-shelf software (COTS)
- Application stores (for cell phones and tablets)
- Malware (or other “hostile” programs)
- Technology forecasting
- Mistrust in the Compilation Chain
- C compiler possibly buggy
- Checking low-level bugs (e.g., exploitability of a stack buffer-overflow)
- Bugs with a strong interconnection with hardware



# 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

■ Context-(in)sensitive

■ Inter-(intro)procedural

■ Path-(in)sensitive

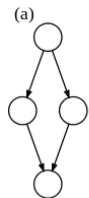
**Further reading:** *A few billion lines of code later: using static analysis to find bugs in the real world.* Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallett, Charles Henri-Gros, Asya Kamsky, Scott McPeak, Dawson Engler. *Communications of the ACM*, vol. 53, iss. 2, pp. 66-75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

# Outline

- 1 Introduction to Program Binary Analysis
- 2 Static Analysis Techniques**
- 3 Dynamic Analysis Techniques

# Static Analysis Techniques

## Control-Flow Graphs

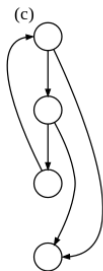


- **Flow of control inside a function**

- **Nodes: basic blocks**

- Sequence of consecutive program instructions having an entry point (first instruction executed) and one exit point (last instruction executed)
- Entry and exit blocks

- **Edge:** control flows from A to B



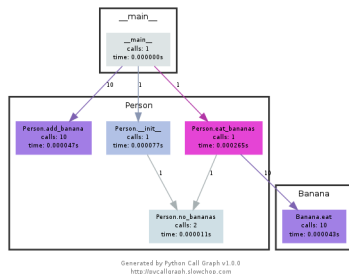
## Applications

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

Credits: [https://en.wikipedia.org/wiki/Control\\_flow\\_graph](https://en.wikipedia.org/wiki/Control_flow_graph)

# Static Analysis Techniques

## Call Graphs



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

Credits: [https://en.wikipedia.org/wiki/Call\\_graph](https://en.wikipedia.org/wiki/Call_graph)

# Static Analysis Techniques

## Disassembling

0040166B	..0F85 24010000	JNZ xconv.00401795	
00401671	. 6A 0E	PUSH 0E	
00401673	. 68 2D544000	PUSH xconv.0040542D	
00401678	. 68 30900000	PUSH 30	
0040167D	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	
00401680	. E8 C1040000	CALL <JMP.&user32.GetDlgItemTextA>	GetDlgItemTextA
00401685	. 83F8 0C	CMP EAX,0C	
00401688	..v7F 4B	JG SHORT xconv.004016D5	
0040168A	. 33F3 94	CMP EAX,4	
0040168D	..v7C 46	JL SHORT xconv.004016D5	
0040168F	. 68 2D544000	PUSH xconv.0040542D	
00401694	. 68 06334000	PUSH xconv.00403306	
00401699	. E8 26050000	CALL <JMP.&kernel32.lstrcpyA>	lstrcpyA
0040169E	. 6A 1B	PUSH 1B	
004016A0	. 68 62544000	PUSH xconv.00405462	
004016A5	. 68 81000000	PUSH 81	
004016AA	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	
004016AD	. E8 94040000	CALL <JMP.&user32.GetDlgItemTextA>	GetDlgItemTextA
004016B2	. E8 48020000	CALL xconv.00401501	
004016B7	. 83F8 01	CMP EAX,1	
004016BA	..v74 32	JE SHORT xconv.004016EE	
004016BC	. 8005 B8534000	ADD BYTE PTR DS:[4053B8],1	
004016C3	. 805D B8534000	CMP BYTE PTR DS:[4053B8],3	
004016C8	..0F34 9C000000	JE xconv.00401795	
004016D0	..vE9 81000000	JMP xconv.00401755	
004016D5	> 6A 10	PUSH 10	
004016D7	. 68 79304000	PUSH xconv.00403079	
004016DC	. 68 53324000	PUSH xconv.00403253	
004016E1	. 68 30900000	PUSH DWORD PTR SS:[EBP+8]	
004016E4	. E8 69040000	CALL <JMP.&user32.MessageBoxA>	MessageBoxA
004016E9	..vE9 D7000000	JMP xconv.004017C5	
004016EE	> E8 A0030000	CALL xconv.00401A98	
004016F3	. C605 72434000	MOV BYTE PTR DS:[404372],1	
004016F8	. 6A 40	PUSH 40	
004016FC	. 68 33304000	PUSH xconv.00403033	
00401701	. 68 3E304000	PUSH xconv.0040303E	
00401706	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	
00401709	. E8 44040000	CALL <JMP.&user32.MessageBoxA>	MessageBoxA
0040170E	. 6A 00	PUSH 0	Result = 0

- Roughly speaking, read PUSH EAX instead of 0x50
- **Lot of tools** see [https://en.wikibooks.org/wiki/X86\\_Disassembly/Disassemblers\\_and\\_Decompilers](https://en.wikibooks.org/wiki/X86_Disassembly/Disassemblers_and_Decompilers)
  - Win32Dasm
  - OllyDBG (also debugger)
  - IDA Pro (also debugger)
  - r2 (also debugger)

# Static Analysis Techniques

## Disassembling

### Main challenges

- **Variable length instruction sets:** overlapping instructions
- **Data and code mixed:** misclassify data as instructions
- **Indirect jumps:** Any location could be the start of an instruction!
- **Start of functions:** when calls are indirect
- **End of functions:** when no dedicated return instruction exists
  - Handwritten assembly code may not conform the standard calling conventions
- **Code compression:** code of two functions overlaps
- **Self-modifying code**

# Static Analysis Techniques

## Decompilation – example

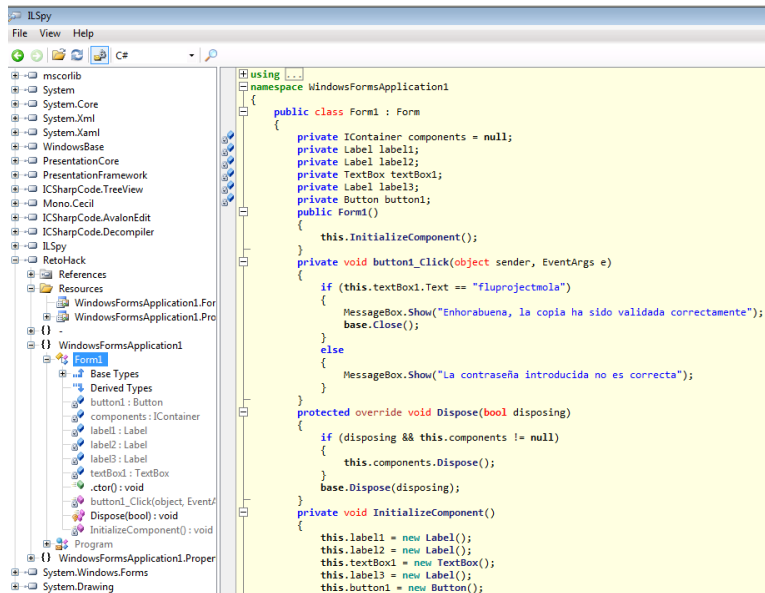
```
int __stdcall sub_40162C(HWND hDlg, int a2, int a3, int a4){
    HICON v4; // eax@2
    UINT v5; // eax@5

    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\r\n
Coded by\t: TDC and BoR0\r\n\r\nVersion\t\t: 1.0\r\n\r\nRelease
date\t: 18-08-2005\r\n\r\nX-Convertor converts up to 4KB
each convert.\r\n\r\n\r\nRegistered version. Thank you.\r\n\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



The screenshot displays the ILSpy decompilation tool interface. On the left, a tree view shows the project structure, including references to various .NET assemblies like System, System.Core, System.Xml, System.Xaml, WindowsBase, PresentationCore, PresentationFramework, ICSharpCode.TreeView, Mono.Cecil, ICSharpCode.AvalonEdit, ICSharpCode.Decompiler, ILSpy, and RetoHack. The 'Resources' folder is expanded, showing the decompiled code for 'WindowsFormsApplication1.Form1'. The main pane on the right shows the decompiled C# code for the 'Form1' class, which inherits from 'Form'. The code includes private fields for 'components', 'label1', 'label2', 'label3', 'textBox1', and 'button1', a constructor, and two event handlers: 'button1\_Click' and 'InitializeComponent'. The 'button1\_Click' method contains two conditional messages: one for a successful validation ('Enhorabuena, la copia ha sido validada correctamente') and another for an incorrect password ('La contraseña introducida no es correcta'). The 'InitializeComponent' method initializes the UI controls with 'new' statements.

```
using ...
namespace WindowsFormsApplication1
{
    public class Form1 : Form
    {
        private IContainer components = null;
        private Label label1;
        private Label label2;
        private TextBox textBox1;
        private Label label3;
        private Button button1;
        public Form1()
        {
            this.InitializeComponent();
        }
        private void button1_Click(object sender, EventArgs e)
        {
            if (this.textBox1.Text == "fluprojectmola")
            {
                MessageBox.Show("Enhorabuena, la copia ha sido validada correctamente");
                base.Close();
            }
            else
            {
                MessageBox.Show("La contraseña introducida no es correcta");
            }
        }
        protected override void Dispose(bool disposing)
        {
            if (disposing && this.components != null)
            {
                this.components.Dispose();
            }
            base.Dispose(disposing);
        }
        private void InitializeComponent()
        {
            this.label1 = new Label();
            this.label2 = new Label();
            this.textBox1 = new TextBox();
            this.label3 = new Label();
            this.button1 = new Button();
        }
    }
}
```



# Static Analysis Techniques

## Decompilation

### Main challenges

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

# Static Analysis Techniques

## Data Flow Analysis

- Analyze the **effect of each basic block**
- Compose effects of basic blocks to derive information at basic block boundaries
- Framework to **provide facts about programs**. Based on all paths through program (including also infeasible paths)
- **Derives information about the dynamic behavior of a program by only examining the static code**

### Useful for...

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

# Static Analysis Techniques

## Data Flow Analysis

Consider the statement  $a = b + c$

### Statement effects

- **Uses variables** ( $b, c$ )
- **“Kills” a previous definition** (previous value of  $a$ )
- **New definition** ( $a$ )

### ■ **Compose effect of statements** → **effect of a basic block**

- *Locally exposed use*: use of a data item which is not preceded in the basic block by a definition of the data item
- Any definition of a data item kills all definitions of the same data item reaching the basic block
- *Locally available definition*: last definition of data item in basic block

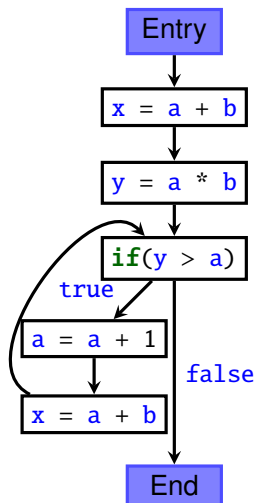
# Static Analysis Techniques

## Data Flow Analysis

### ■ Facts

- $a + b$  is available
- $a * b$  is available
- $a + 1$  is available

- Let's compute for each program point the facts that holds!

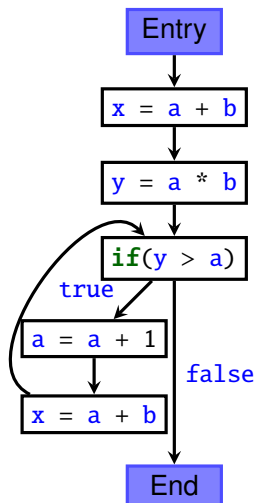


# Static Analysis Techniques

## Data Flow Analysis

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

---



# Static Analysis Techniques

## Data Flow Analysis

- *Forward vs. backward*: data flow from in to out (vs. out to in)
- *Must vs. may*: at joint points, only keep the 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 at analyzing local variables**
  - What about values stored in the heap?
  - Not modeled in traditional data flow
- In general, **it is hard to analyze pointers**. Suppose  $*x = p$ 
  - Assume all data flow facts are killed
  - Or assume that write through  $x$  may affect any variable whose address has been taken

# Static Analysis Techniques

## Symbolic Execution

- **Enables us to scale and model all possible program executions**
- Concrete vs. symbolic execution
  - **Testing works, but each test only explores one possible execution path**
- **Generalizes testing**
  - Allows unknown symbolic variables in evaluation
  - **Checks feasibility of program paths**

### Main challenges

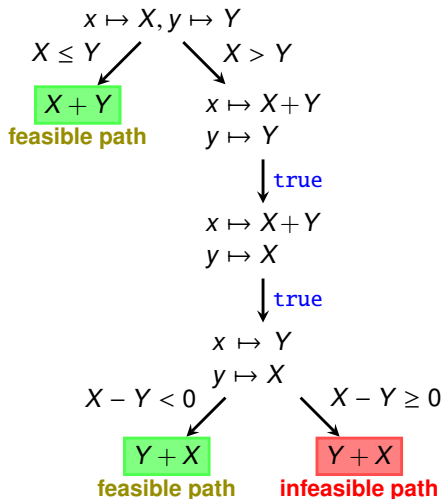
- **Path explosion**
- **Modeling statements and environments**
- **Constraint solving**

**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

```
1  int f(int x, int y)
2  {
3      if(x > y)
4      {
5          x = x + y;
6          y = x - y;
7          x = x - y;
8          if(x - y > 0)
9              perror("Error!");
10     }
11
12     return x + y;
13 }
```



How to decide which branches are feasible?

**Combine path condition with branch condition and ask a SMT solver!**



# Static Analysis Techniques

## Symbolic Execution – example: bug finding

**Spot the vulnerability! Which value triggers it?**

```
1  int bar(int i)
2  {
3      int j = 2*i;
4      i++;
5      i = i*j;
6      if (i < 1)
7          i = -i;
8
9      i = j/i;
10     return i;
11 }
```

False branch condition

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

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

True branch condition

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

$$(i_{in} + 1)2i_{in} < 1$$

# Static Analysis Techniques

## Symbolic Execution – example: bug finding

**Spot the vulnerability! Which value triggers it?**

```
1  int bar(int i)
2  {
3      int j = 2*i;
4      i++;
5      i = i*j;
6      if (i < 1)
7          i = -i;
8
9      i = j/i;
10     return i;
11 }
```

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 arises problems...

# Static Analysis Techniques

## Symbolic Execution – example: bug finding

**Spot the vulnerability! Which value triggers it?**

```
1  int bar(int i)
2  {
3      int j = 2*i;
4      i++;
5      i = i*j;
6      if (i < 1)
7          i = -i;
8
9      i = j/i;
10     return i;
11 }
```

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 arises problems...

False branch is always safe

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

What happens with the true branch?

# Static Analysis Techniques

## Symbolic Execution – example: bug finding

**Spot the vulnerability! Which value triggers it?**

```
1 int bar(int i)
2 {
3     int j = 2*i;
4     i++;
5     i = i*j;
6     if (i < 1)
7         i = -i;
8
9     i = j/i;
10    return i;
11 }
```

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 arises problems...

False branch is always safe

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

What happens with the true branch?

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

# Static Analysis Techniques

## Symbolic Execution – example: bug finding

Spot the vulnerability! Which value triggers it?

```
1 int bar(int i)
2 {
3     int j = 2*i;
4     i++;
5     i = i*j;
6     if (i < 1)
7         i = -i;
8
9     i = j/i;
10    return i;
11 }
```

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 arises problems...

False branch is always safe

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

What happens with the true branch?

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

# Outline

- 1 Introduction to Program Binary Analysis
- 2 Static Analysis Techniques
- 3 Dynamic Analysis Techniques**

# Dynamic Analysis Techniques

## Debugging

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

# Dynamic Analysis Techniques

## Debugging (example: OLLYDBG)

**OllyDbg - OLLYDBG.EXE - [CPU - main thread, module OLLYDBG]**

File View Debug Plugins Options Window Help

LEMTW H C / K B R ... S

00401000 \$->EB 10 JMP SHORT OLLYDBG.00401012

00401002	62	DB 62	CHAR '*'	EAX 00000000
00401003	3A	DB 3A	CHAR 'b'	EAX 0012FF80
00401004	3A	DB 3A	CHAR '*'	EDX 7C91E4F4 ntdll.KiFastSystemC
00401005	43	DB 43	CHAR 'C'	EBX 7FFDE000
00401006	2B	DB 2B	CHAR '+'	ESP 0012FFC4
00401007	2B	DB 2B	CHAR '+'	EBP 0012FF80
00401008	43	DB 43	CHAR 'H'	ESI FFFFFFFF
00401009	4F	DB 4F	CHAR '0'	EDI 7C920200 ntdll.7C920200
0040100A	4F	DB 4F	CHAR '0'	EIP 00401000 OLLYDBG.<ModuleEntry
0040100B	4B	DB 4B	CHAR 'K'	C 0 ES 0023 32bit 0 (FFFFFFFF)
0040100C	90	NOP		P 1 CS 001B 32bit 0 (FFFFFFFF)
0040100D	59	POP E9		A 0 SS 0023 32bit 0 (FFFFFFFF)
0040100E	> 2B014B00	DD OFFSET OLLYDBG.____CPPdebugHook		C 1 DS 0023 32bit 0 (FFFFFFFF)
00401012	> A1 1B014B00	MOV EAX, DWORD PTR DS:[4B011B]		S 0 GS 0000 NULL
00401017	> C1E0 02	SHL EAX, 2		T 0 D 0
0040101A	> A3 1F014B00	MOV DWORD PTR DS:[4B011F], EAX		O 0 LastErr ERROR_FILE_NOT_FOUND
0040101F	> 52	PUSH EDX	[Module = NULL GetModuleLeHandler	EFL 00000246 (NO, NB, E, BE, NS, PE, GI
00401020	> 6A 00	PUSH 0		ST0 empty -UNORM EDEC 01050104 0
00401022	> E8 4BE0A000	CALL <JMP, &KERNEL32.GetModuleLeHandler>		ST1 empty +UNORM 006F 005C0030 0
00401027	> 8B00	MOV EDX, EAX		ST2 empty +UNORM 0069 002E0067 0
00401029	> E8 DA240A00	CALL OLLYDBG.004A3508		ST3 empty 0,0
0040102E	> 5A	POP EDX		ST4 empty 0,0
0040102F	> E8 70100A00	CALL OLLYDBG.004A2804		ST5 empty 0,0
00401034	> E8 D3240A00	CALL OLLYDBG.004A350C		ST6 empty 1.00000000000000000000
00401039	> 6A 00	PUSH 0		ST7 empty 1.00000000000000000000
0040103B	> E8 F8390A00	CALL OLLYDBG.004A4438	[Arg1 = 00000000 GetModuleLeHandler	FST 4020 Cond 3 2 1 0 E S I
00401040	> 59	POP EAX		FCW 027F Prec 1 0 0 0 Err 0 0
00401041	> C4004B00	PUSH OLLYDBG.004B00C4		
00401044	> 00	PUSH 0		
00401048	> E8 25E00A00	CALL <JMP, &KERNEL32.GetModuleLeHandler>	[Module = NULL GetModuleLeHandler	

00401012=OLLYDBG.00401012

Address	Hex dump	ASCII	Comment
004B0000	00 00 60 43 4A 00 00 00	..*CJ..	
004B0008	F4 45 4A 00 00 05 F0 5C	MEJ..&-l	
004B0010	0A 03 00 04 03 53 00 00	.#SJ..	
004B0018	00 04 93 83 4A 00 00 0A	.#SJ..	
004B0020	AC 02 4D 00 00 0A 08 E1	#SJ..ll	
004B0028	4A 00 00 0A F0 AC 4A 00	.J..0%J.	
004B0030	00 0A CC E6 4A 00 00 01	.#AJ..0	
004B0038	53 FE 4D 00 00 01 5C C2	%SJ..0*	
004B0040	4A 00 00 03 53 C6 4A 00	.#SJ..	
004B0048	00 02 04 CB 4A 00 00 03	.#SfJ..#	
004B0050	9C CD 4A 00 00 00 FC D3	%-J..*E	
004B0058	4A 00 00 00 88 28 4A 00	.J...e(J.	
004B0060	00 01 C3 3C 4A 00 00 00	.0 CJ..	
004B0068	0C 46 4A 00 00 00 5D FJ..	.#m	
004B0070	4A 00 00 04 9C 83 4A 00	.#eSj..	
004B0078	00 03 E0 C6 4A 00 00 02	.#0Sj..0	

Registers (FPU)

EAX 00000000  
ECX 0012FF80  
EDX 7C91E4F4 ntdll.KiFastSystemC  
EBX 7FFDE000  
ESP 0012FFC4  
EBP 0012FF80  
ESI FFFFFFFF  
EDI 7C920200 ntdll.7C920200  
EIP 00401000 OLLYDBG.<ModuleEntry:

C 0 ES 0023 32bit 0 (FFFFFFFF)  
P 1 CS 001B 32bit 0 (FFFFFFFF)  
A 0 SS 0023 32bit 0 (FFFFFFFF)  
C 1 DS 0023 32bit 0 (FFFFFFFF)  
S 0 GS 0000 NULL  
T 0 D 0  
O 0 LastErr ERROR\_FILE\_NOT\_FOUND  
EFL 00000246 (NO, NB, E, BE, NS, PE, GI

ST0 empty -UNORM EDEC 01050104 0  
ST1 empty +UNORM 006F 005C0030 0  
ST2 empty +UNORM 0069 002E0067 0  
ST3 empty 0,0  
ST4 empty 0,0  
ST5 empty 0,0  
ST6 empty 1.00000000000000000000  
ST7 empty 1.00000000000000000000

FST 4020 Cond 3 2 1 0 E S I  
FCW 027F Prec 1 0 0 0 Err 0 0

0012FFC4 7C817067 RETURN to kernel32.7C817067  
7C920200 ntdll.7C920200  
0012FFC0 FFFFFFFF  
0012FFD0 7FFDE000  
0012FFD4 805448FD  
0012FFD8 0012FFC8  
0012FFDC 82D0B1D8  
0012FFE0 FFFFFFFF  
0012FFE4 7C8394C0 End of SEH chain  
0012FFE8 7C817070 SE handler  
00000000 kernel32.7C817070  
00000000  
00000000  
00401000 OLLYDBG.<ModuleEntryPc  
0012FFFC 00000000

Analysing OLLYDBG: 1017 heuristical procedures, 4602 calls to known, 7988 calls to guessed functions

Paused



# 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 previous knowledge of program internals
  - **Evolved to white-box approach**: state-of-the-art fuzzers “learn” from program behavior
- **Many anomalous (unexpected, invalid, or random data) inputs are given to the application**
- **Application is monitored for any sign of error**
  - Unexpected behavior
  - Crashes
    - Buffer overflow
    - Integer overflow
    - Memory corruption errors
    - Format string bugs

# Dynamic Analysis Techniques

## Fuzzing

### 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

### A simple example: HTTP GET requests

- A 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 /AAAAAAAAAAAAAA.html HTTP/1.1
  - GET /index.html HTTTTTTTTTTTTTTP/1.1
  - GET /index.html HTTP/1.1.1.1.1.1.1.1
  - etc.

### Types of fuzzers

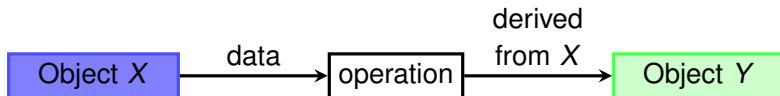
- Mutation-based fuzzing
- Generation-based fuzzing

# Dynamic Analysis Techniques

## Taint analysis

### Measure what is the influence of the input data into the application

- Data come from tainted sources (any external input) and end 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 **untrustworthy** (e.g., user-supplied string)



### Taint Propagation

- **Object  $X$  tainted the 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 properly handle some types of information flow

#### ■ Overtainting

- Deciding when to introduce taint is often easier than deciding when to remove taint

#### ■ Time of detection vs. time of attack

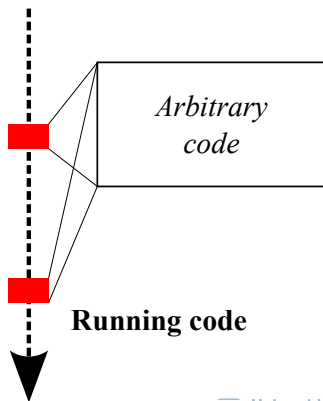
- When used for attack detection, dynamic taint analysis may raise an alert too late

# Dynamic Analysis Techniques

## Dynamic Binary Instrumentation

### Arbitrary code addition during the execution of a binary

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



# Dynamic Analysis Techniques

## Dynamic Binary Instrumentation

### Arbitrary code addition during the execution of a binary

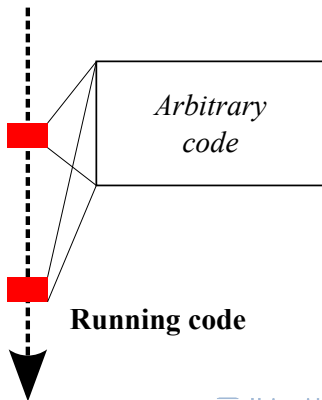
- What do I insert? → **instrumentation function**
- Where? → **addition places**

#### Advantages

- Independent of the programming language
- We can instrument proprietary software
- No need to recompile/relink each time
- Allows to instrument a process already in execution

#### Main disadvantage

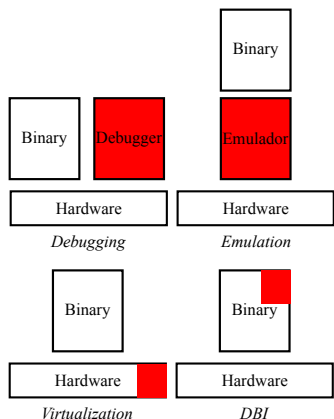
- **Overhead** ⇒ ↓ performance





# Dynamic Analysis Techniques

## Placing DBI in the context of dynamic analysis



- Executable transformation
- Total control over execution
- No need of architectural support

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

# Exploiting Software Vulnerabilities

## Program Binary Analysis

© All wrongs reversed – under CC-BY-NC-SA 4.0 license



**Universidad**  
Zaragoza

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

Course 2020/2021

**Master's Degree in Informatics Engineering**

UNIVERSITY OF ZARAGOZA

Google Meet

