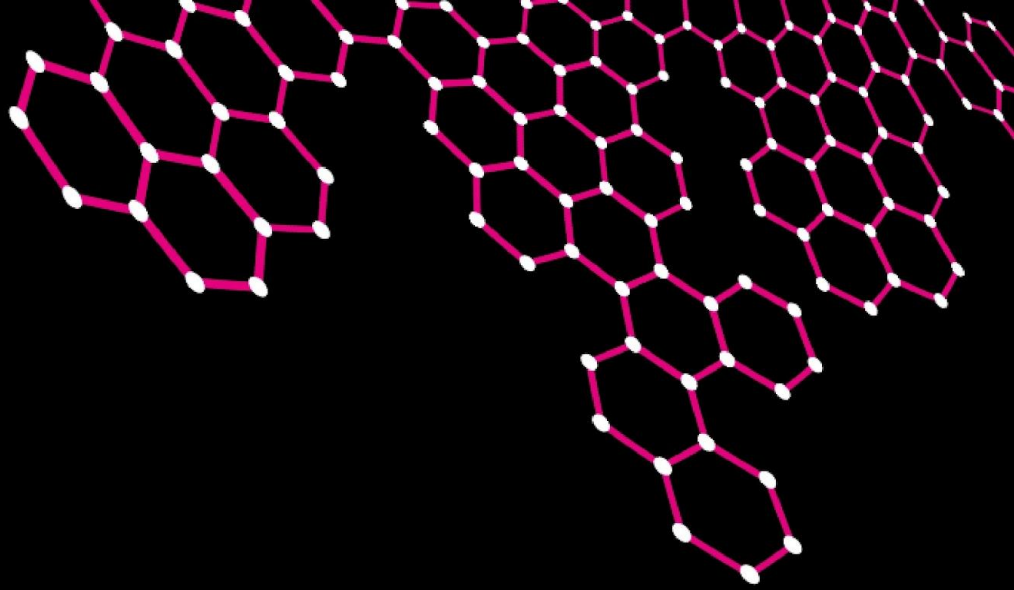
UNIVERSITY OF TWENTE.



A short introduction to: Binary Reverse Engineering

Part 1: Static Analysis

Yoep Kortekaas (y.a.m.kortekaas@utwente.nl)
THS Workshop 11-10-2021

Reverse Engineering - Definition

“to disassemble and examine or analyze in detail (a product or device) to discover the concepts involved in manufacture usually in order to produce something similar” [1]

[1] https://www.merriam-webster.com/dictionary/reverse%20engineer

Reverse Engineering - Definition

“to disassemble and examine or analyze in detail (a product or device) to discover the concepts involved in manufacture usually in order to produce something similar” [1]

To deconstruct a binary executable in order to figure out how the program behaves, reveal it’s design and *extract knowledge*, without having access to the source code of the executable.

[1] https://www.merriam-webster.com/dictionary/reverse%20engineer

Reverse Engineering - Techniques & Tools

Static Analysis

- Find out as much as you can about an executable by looking at the (binary) code
- Usually by means of decompilation
- Hard to perform when code is obfuscated and/or encrypted

Tools:

- Strings
- Radare2
- Ghidra
- ...

Dynamic Analysis

- Find out as much as you can about an executable by interacting with the program in a controlled environment
- Hard to get a `full picture' of the executable under examination

Tools:

- GDB + pwndbg
- pin
- Angr
- ...

Reverse Engineering - Techniques & Tools

Static Analysis

- Find out as much as you can about an executable by looking at the (binary) code
- Usually by means of decompilation
- Hard to perform when code is obfuscated and/or encrypted

Tools:

- Strings
- Radare2
- Ghidra
- ...

Dynamic Analysis

- Find out as much as you can about an executable by interacting with the program in a controlled environment
- Hard to get a `full picture' of the executable under examination

Tools:

- GDB + pwndbg
- pin
- Angr
- ...

Reverse Engineering - Techniques & Tools

Static Analysis

- Find out as much as you can about an executable by looking at the (binary) code
- Usually by means of decompilation
- Hard to perform when code is obfuscated and/or encrypted

Tools:

- Strings
- Radare2
- Ghidra
- ...

Dynamic Analysis

- Find out as much as you can about an executable by interacting with the program in a controlled environment
- Hard to get a `full picture' of the executable under examination

Tools:

- GDB + pwndbg
- pin
- Angr
- ...

Reverse Engineering - Techniques & Tools

Static Analysis

- Find out as much as you can about an executable by looking at the (binary) code
- Usually by means of decompilation
- Hard to perform when code is obfuscated and/or encrypted

Tools:

- Strings
- Radare2
- Ghidra
- ...

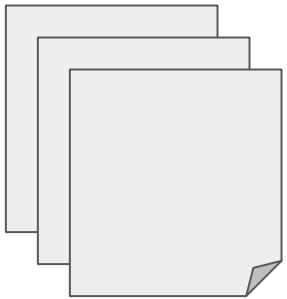
Dynamic Analysis

- Find out as much as you can about an executable by interacting with the program in a controlled environment
- Hard to get a `full picture' of the executable under examination

Tools:

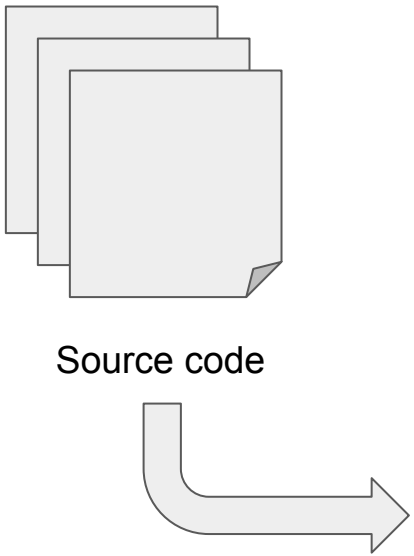
- GDB + pwndbg
- pin
- Angr
- ...

Reverse Engineering - Compilation

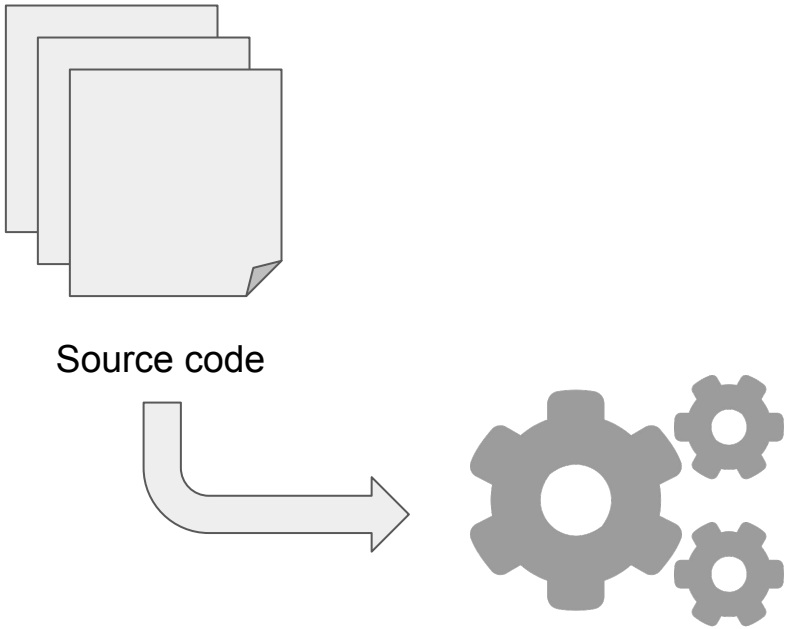


Source code

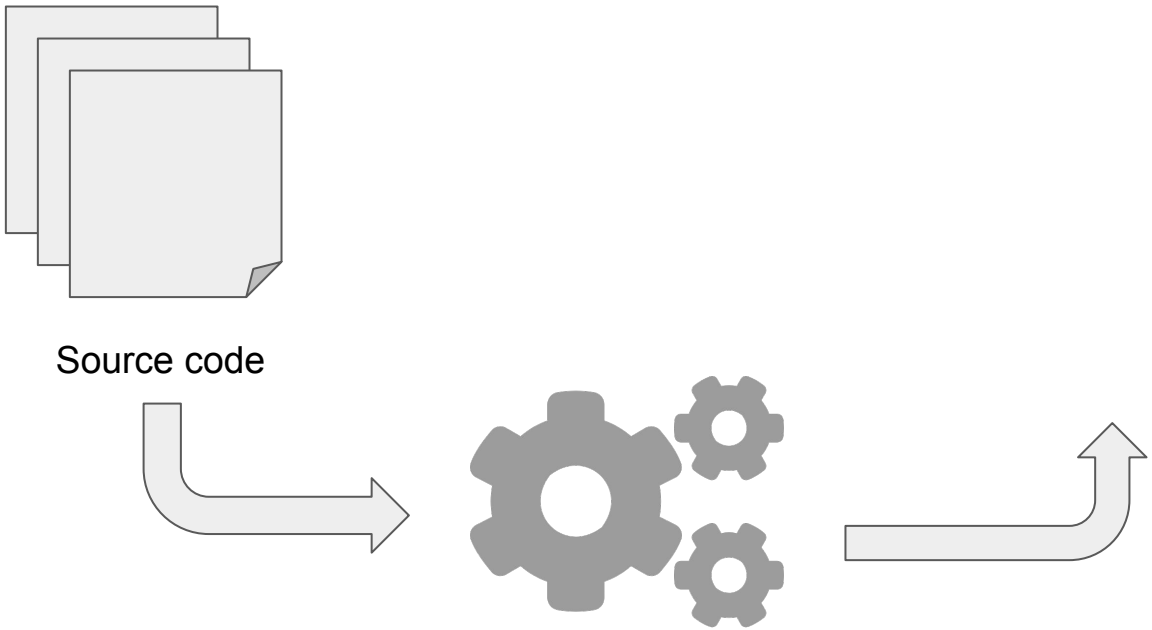
Reverse Engineering - Compilation



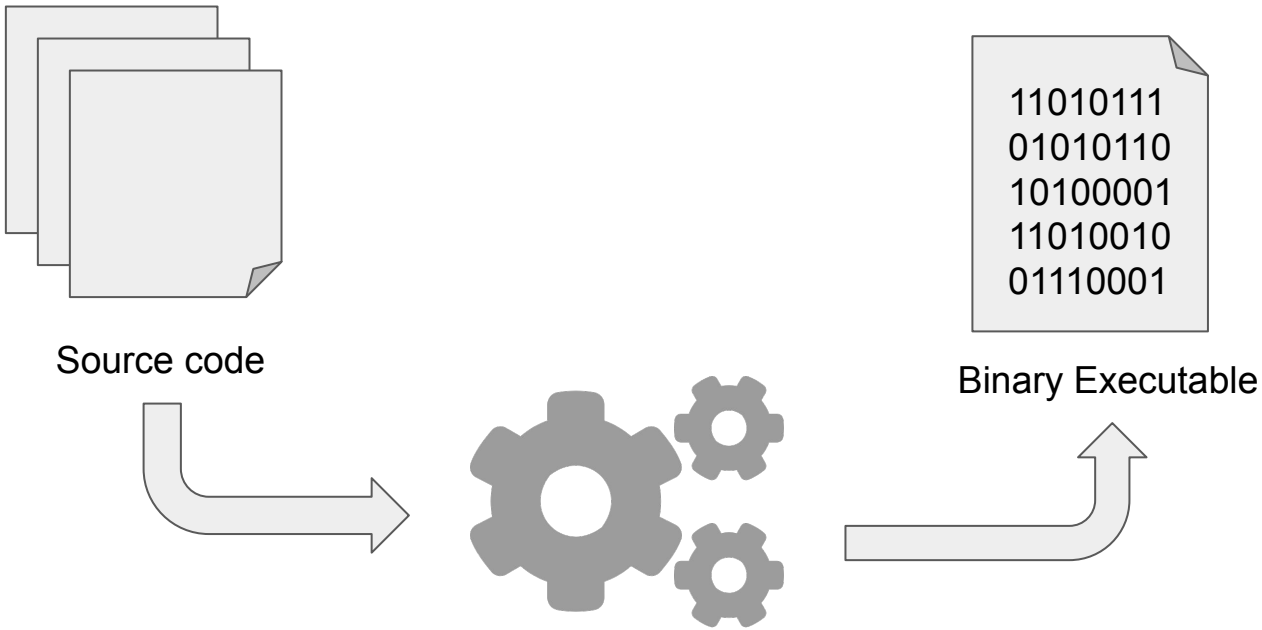
Reverse Engineering - Compilation



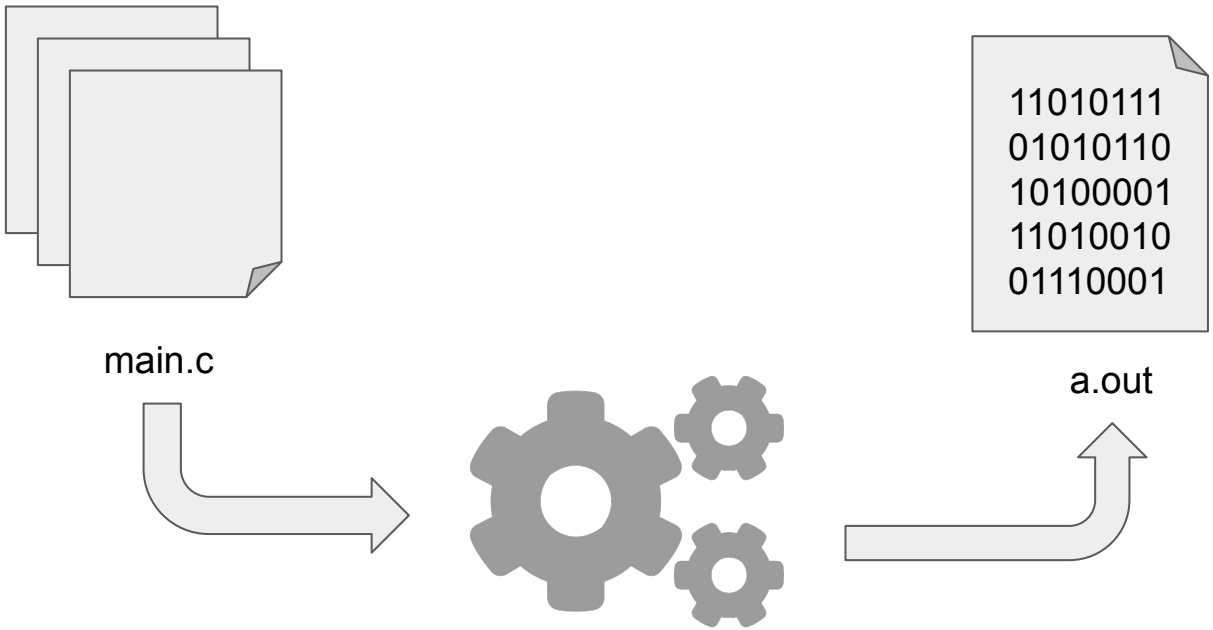
Reverse Engineering - Compilation



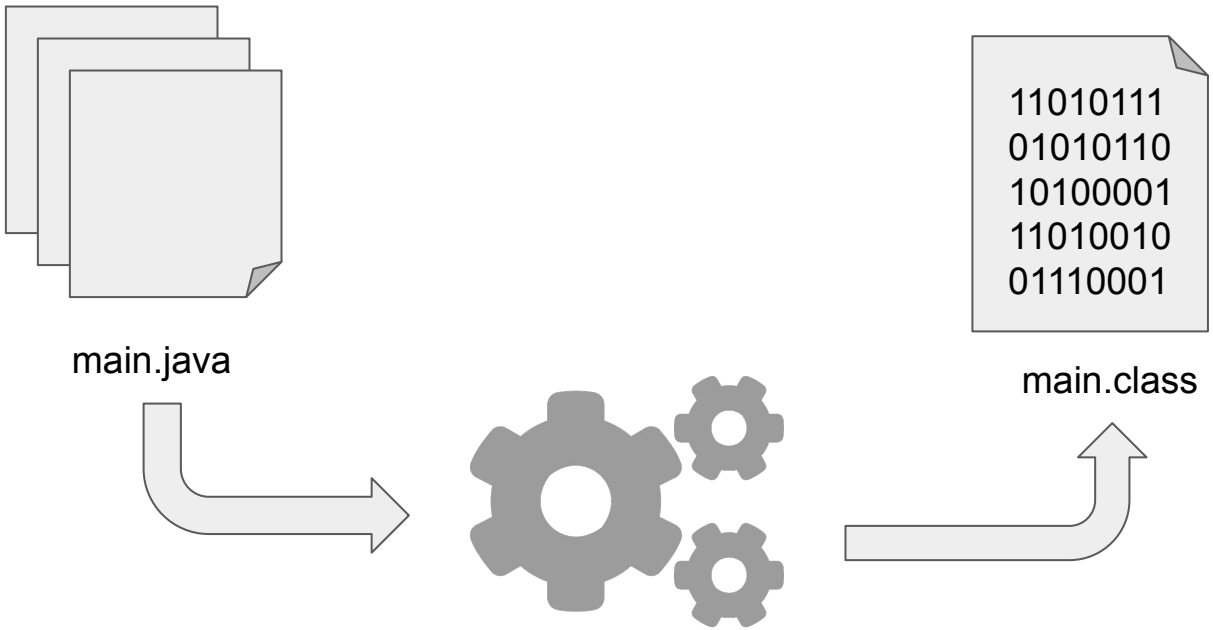
Reverse Engineering - Compilation



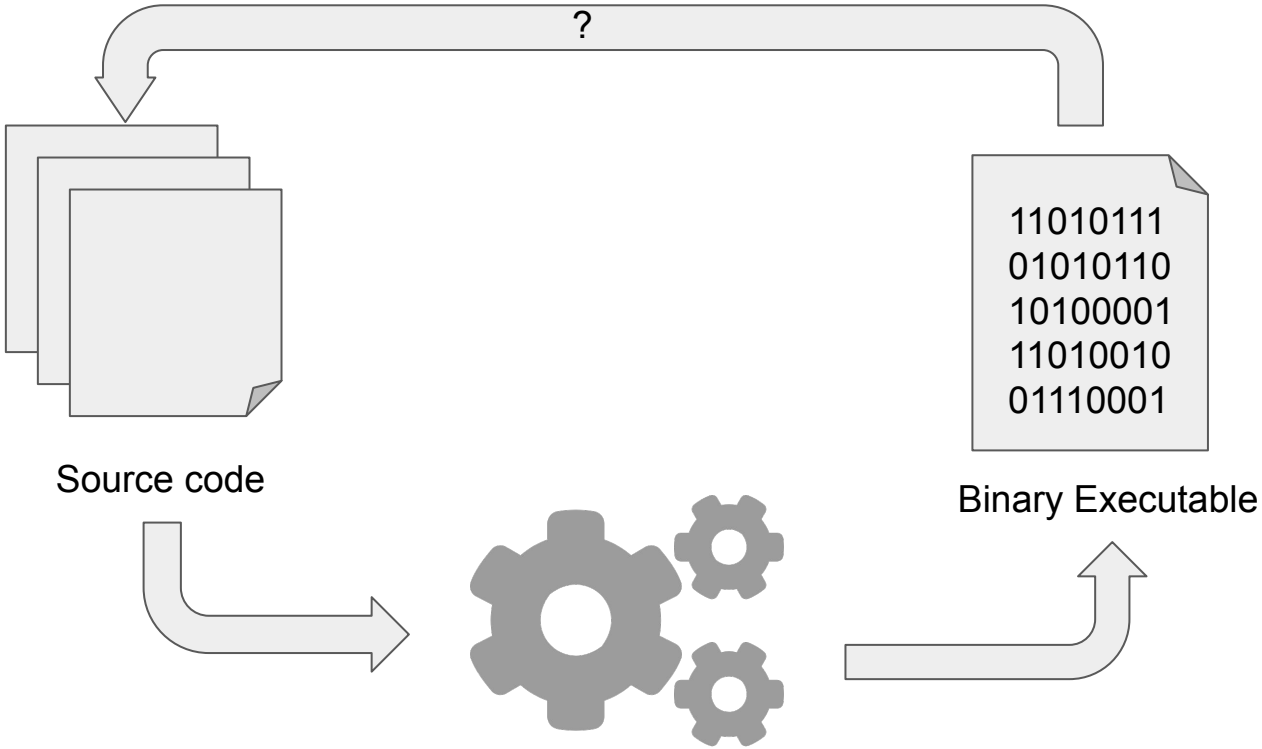
Reverse Engineering - Compilation



Reverse Engineering - Compilation



Reverse Engineering - Decompilation?



Binary Formats & System Architecture

UNIVERSITY OF TWENTE.

Assumptions

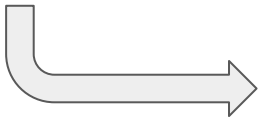
The concepts of reverse engineering apply to any operating system, architecture, programming language, and executable. However, for simplicity we assume:

- Executables running on Linux (kernel >= 2.6)
- Stored in the ELF format (Executable and Linkable Format)
- On a machine running the x86(-64) architecture
- Written in the C language

Assumptions

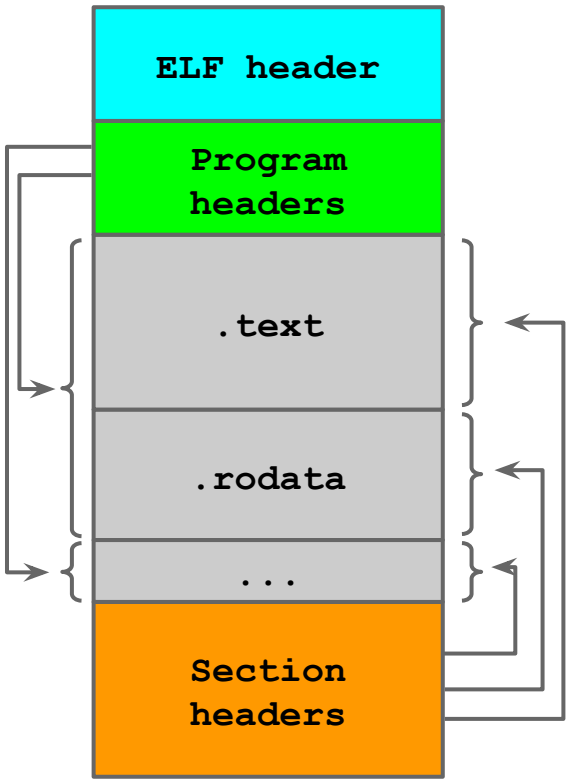
The concepts of reverse engineering apply to any operating system, architecture, programming language, and executable. However, for simplicity we assume:

- Executables running on Linux (kernel >= 2.6)
- Stored in the ELF format (Executable and Linkable Format)
- On a machine running the x86(-64) architecture
- Written in the C language



For challenges in other languages check out https://decompetition.io/

ELF structure



ELF on disk

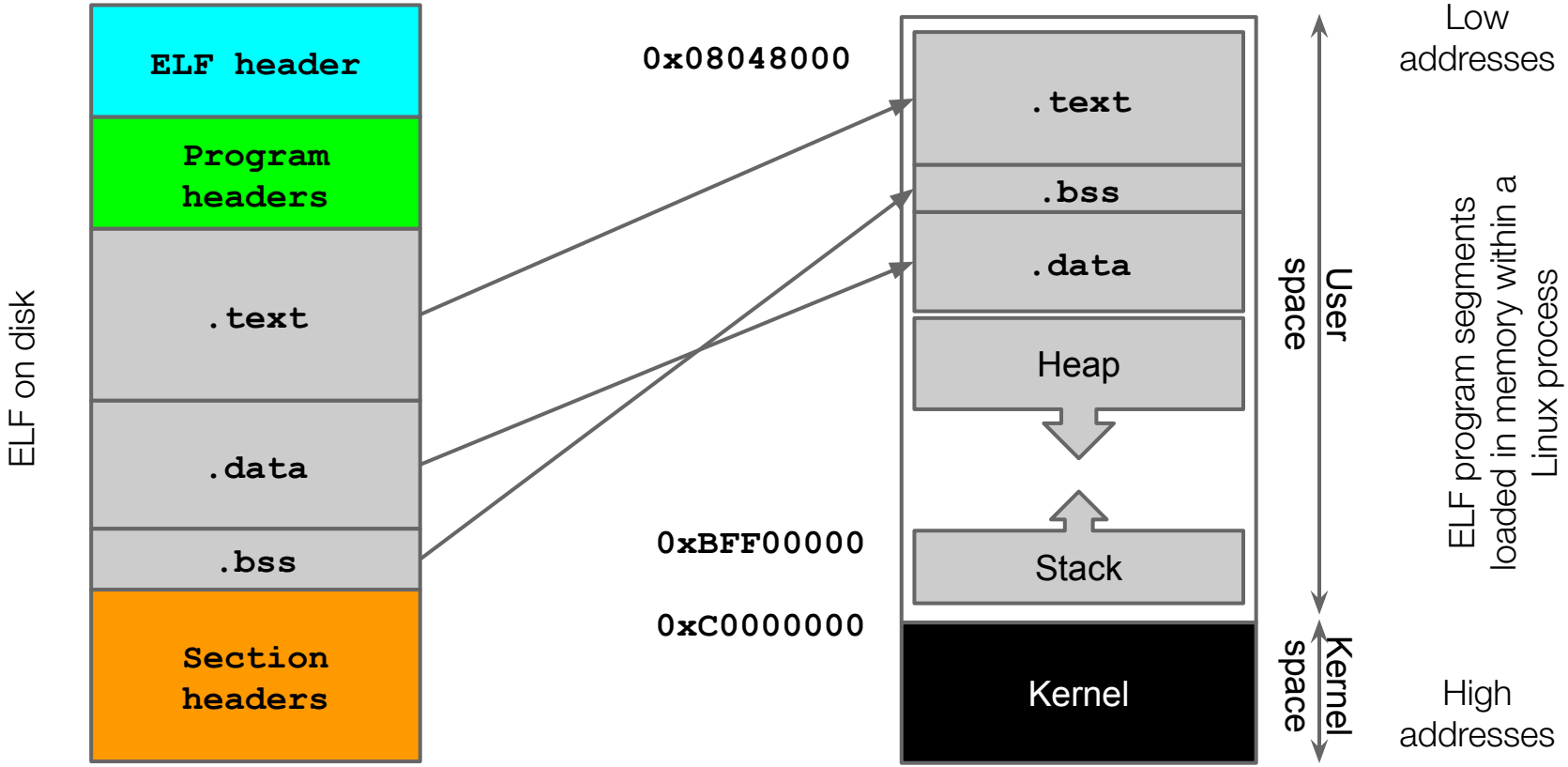
ELF header: Describes the file type and layout, e.g. where the program and section headers start + their size

Program header table: Describes how the executable should be loaded into memory and gives the system the information needed to prepare the program for execution.

Section header table: Describes how the binary is stored on disk.

.bss: holds uninitialized data of the program
.(ro)data: holds the initialized data of the program
.init: holds initialization instructions of the program
.text: Holds the executable instructions of the program

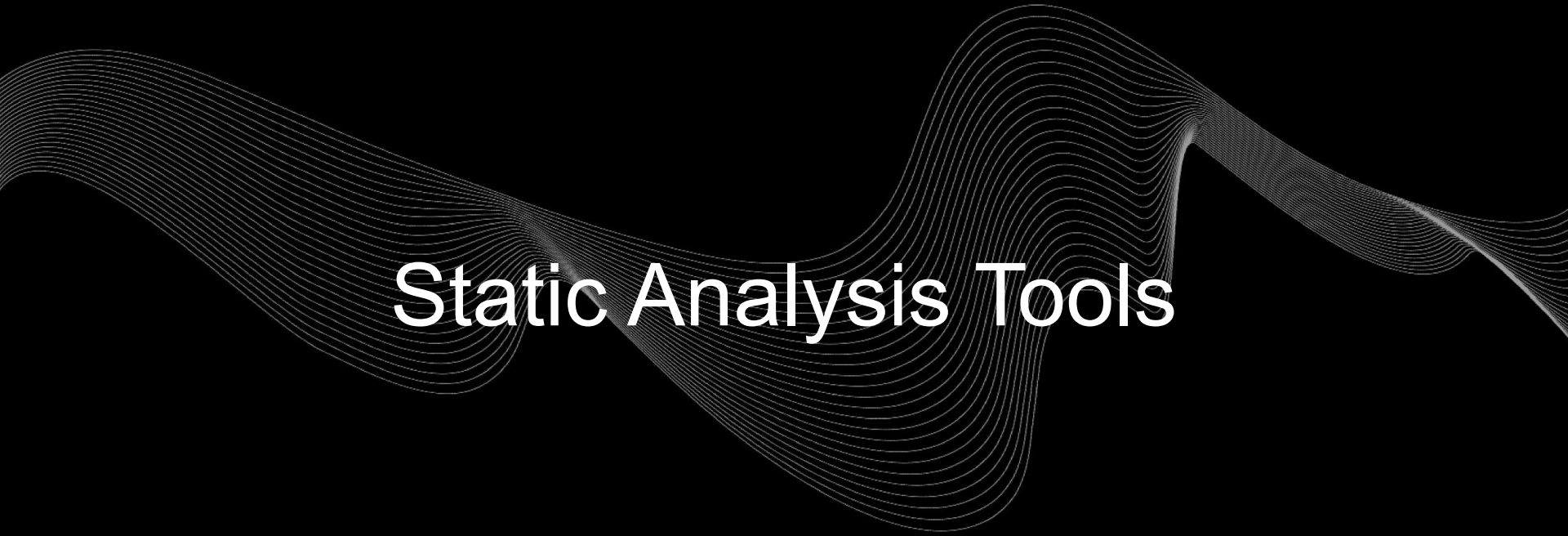
ELF -> Process


ELF on disk
ELF header
Program headers
.text
.data
.bss
Section headers
0x08048000
0xBFF00000
0xC0000000
.text
.bss
.data
Heap
Stack
Kernel
User space
Kernel space
Low addresses
High addresses
ELF program segments loaded in memory within a Linux process

x86-64 Registers

- **General Purpose:** Common mathematical operations. They store data and addresses (EAX, EBX, ECX)
- **ESP**: address of the last stack operation, the **top of the stack**
- **EBP**: address of the **base of the current function frame** (i.e., activation record)
  - relative addressing
- **Control**: Control the function of the processor (execution)
  - **EIP**: address of the next machine instruction to be executed

Static Analysis Tools



UNIVERSITY
OF TWENTE.

Static Analysis Tools

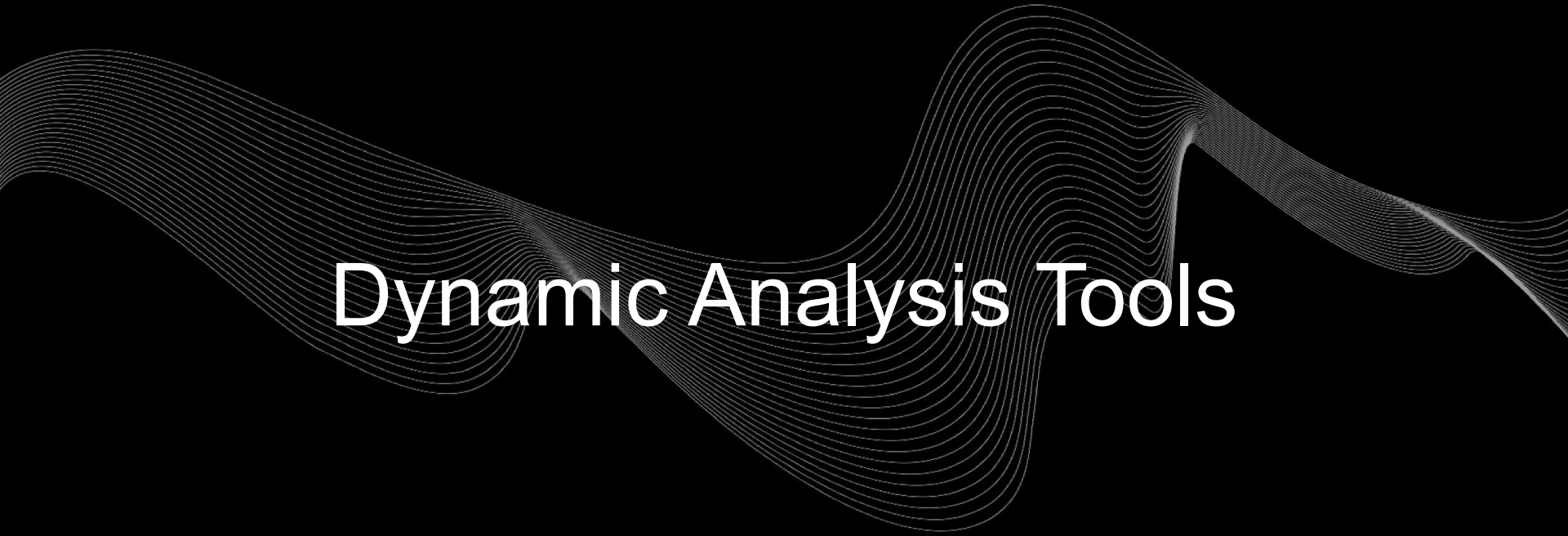
- `strings`

- `radare2`
- `Ghidra`
- `IDA pro`

Challenges

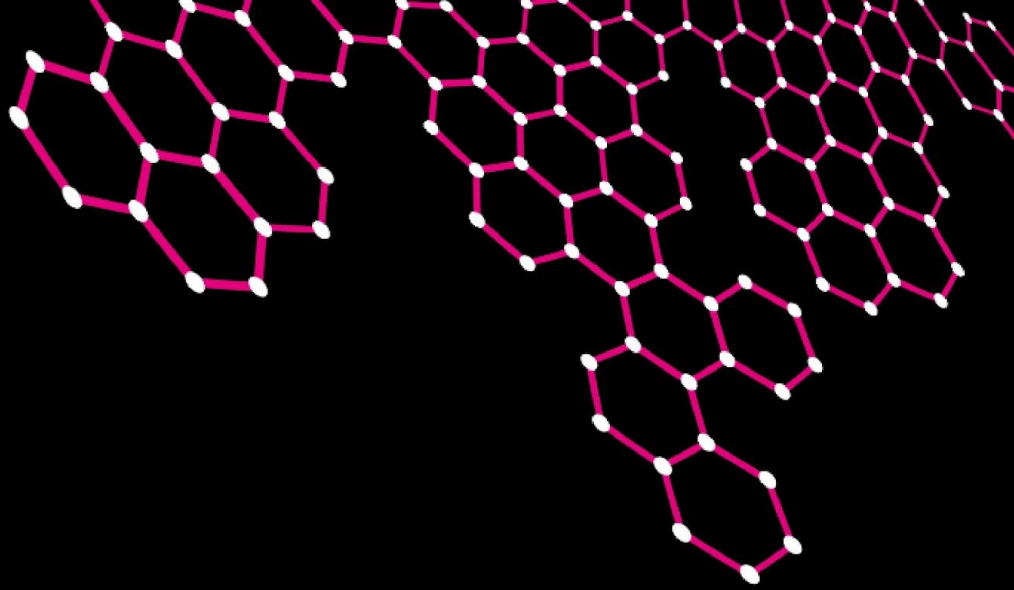
- Hackthebox
  - Impossible Password
  - Exatlon
- Crackmes (https://crackmes.one/)
- Challenge 0 - 4 (https://ths.eemcs.utwente.nl/resources/)

Dynamic Analysis Tools



UNIVERSITY
OF TWENTE.

UNIVERSITY OF TWENTE.



A short introduction to: Binary Reverse Engineering

Part 2: Dynamic Analysis

Yoep Kortekaas (y.a.m.kortekaas@utwente.nl)
THS Workshop 18-10-2021

Reverse Engineering - Static Analysis Example

Reverse Engineering - Techniques & Tools

Static Analysis

- Find out as much as you can about an executable by looking at the (binary) code
- Usually by means of decompilation
- Hard to perform when code is obfuscated and/or encrypted

Tools:

- Strings
- Radare2
- Ghidra
- ...

Dynamic Analysis

- Find out as much as you can about an executable by interacting with the program in a controlled environment
- Hard to get a `full picture' of the executable under examination

Tools:

- GDB + pwndbg
- pin
- Angr
- ...

Reverse Engineering - Techniques & Tools

Static Analysis

- Find out as much as you can about an executable by looking at the (binary) code
- Usually by means of decompilation
- Hard to perform when code is obfuscated and/or encrypted

Tools:

- Strings
- Radare2
- Ghidra
- ...

Dynamic Analysis

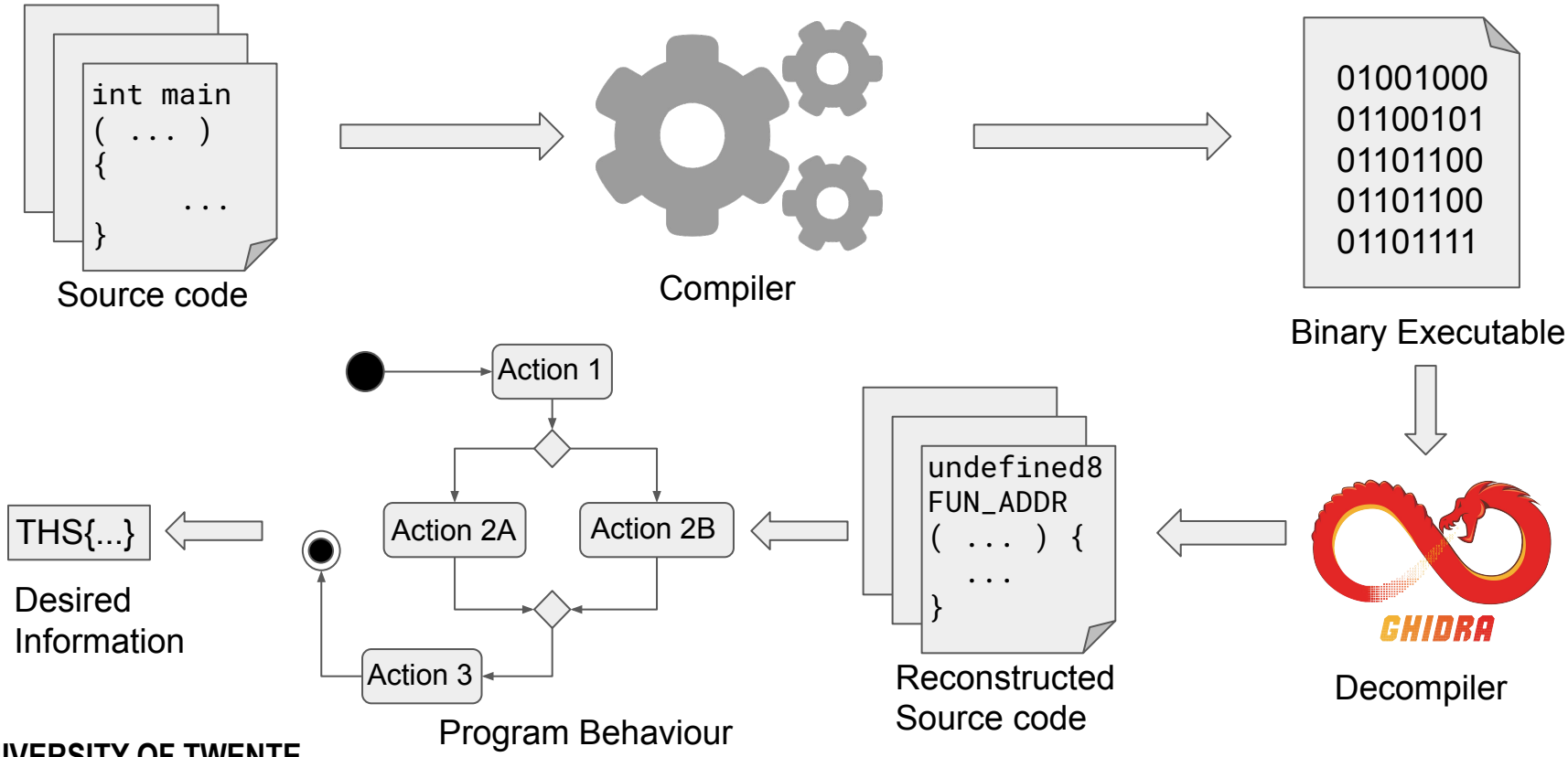
- Find out as much as you can about an executable by interacting with the program in a controlled environment
- Hard to get a `full picture' of the executable under examination

Tools:

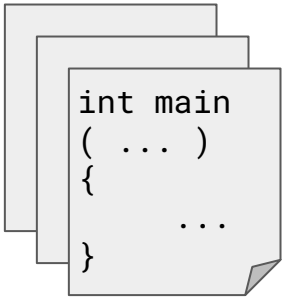
- GDB + pwndbg
- pin
- Angr
- ...

Dynamic Analysis: Concept

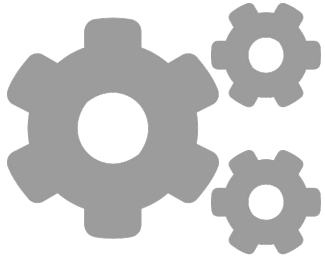
Reverse Engineering - Static Approach



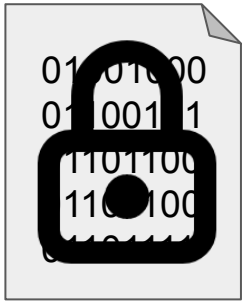
Reverse Engineering - Static Approach



Source code

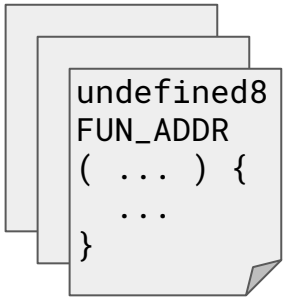


Compiler

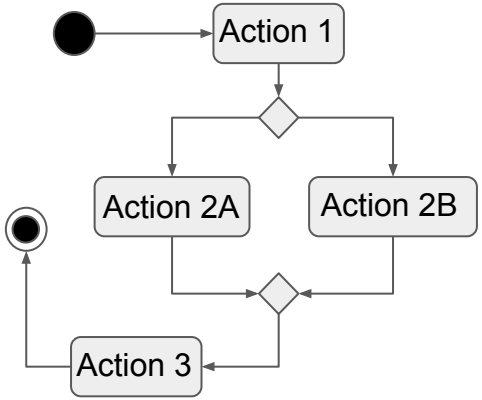


Binary Executable

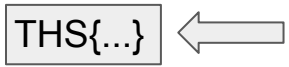
Decompiler



Reconstructed
Source code

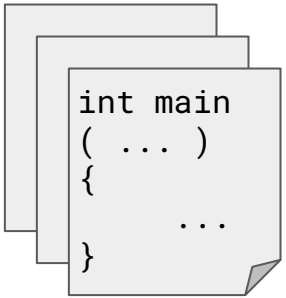


Program Behaviour

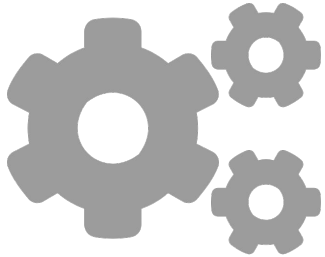


Desired
Information

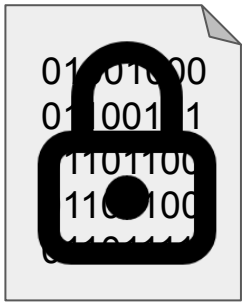
Reverse Engineering - Dynamic Approach



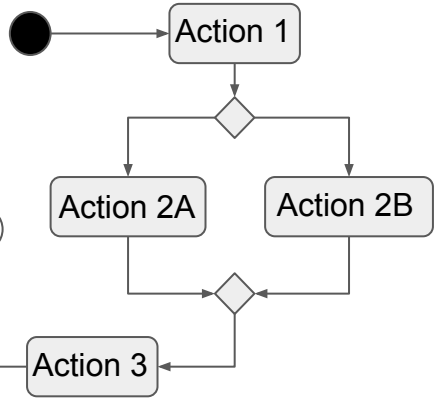
Source code



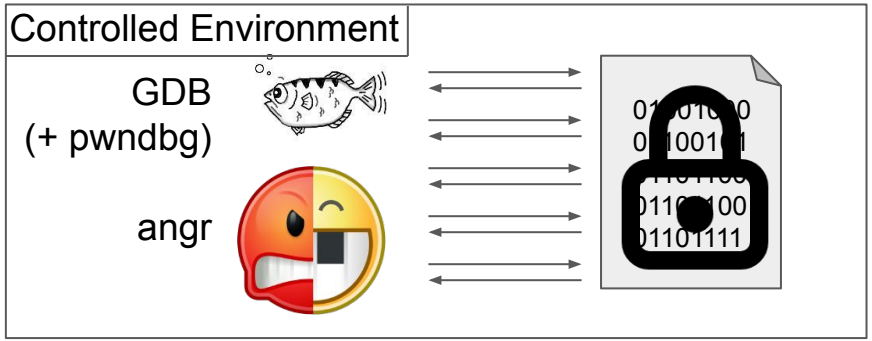
Compiler

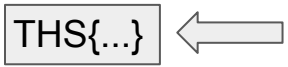


Binary Executable



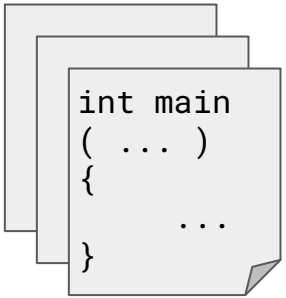
Program Behaviour



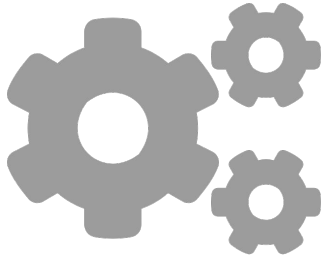


Desired
Information

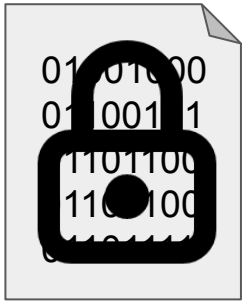
Reverse Engineering - Dynamic Approach



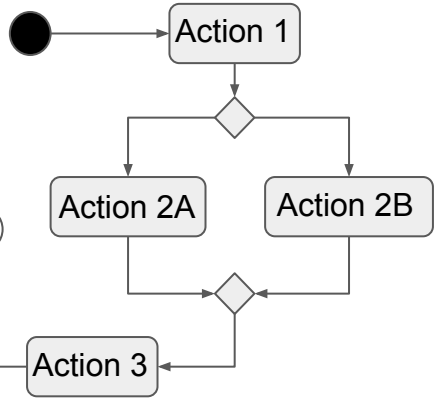
Source code



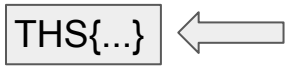
Compiler



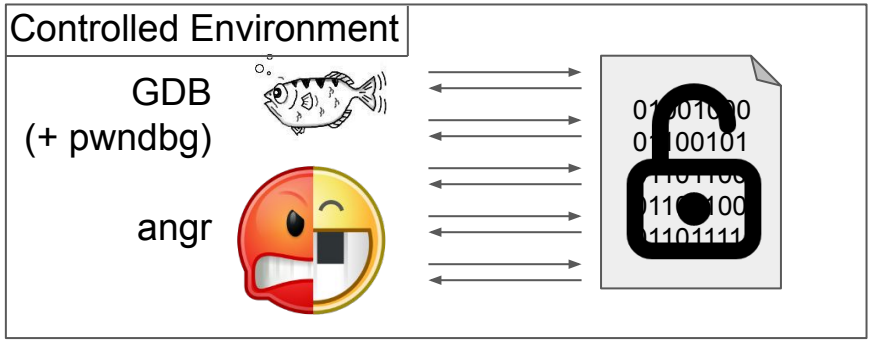
Binary Executable

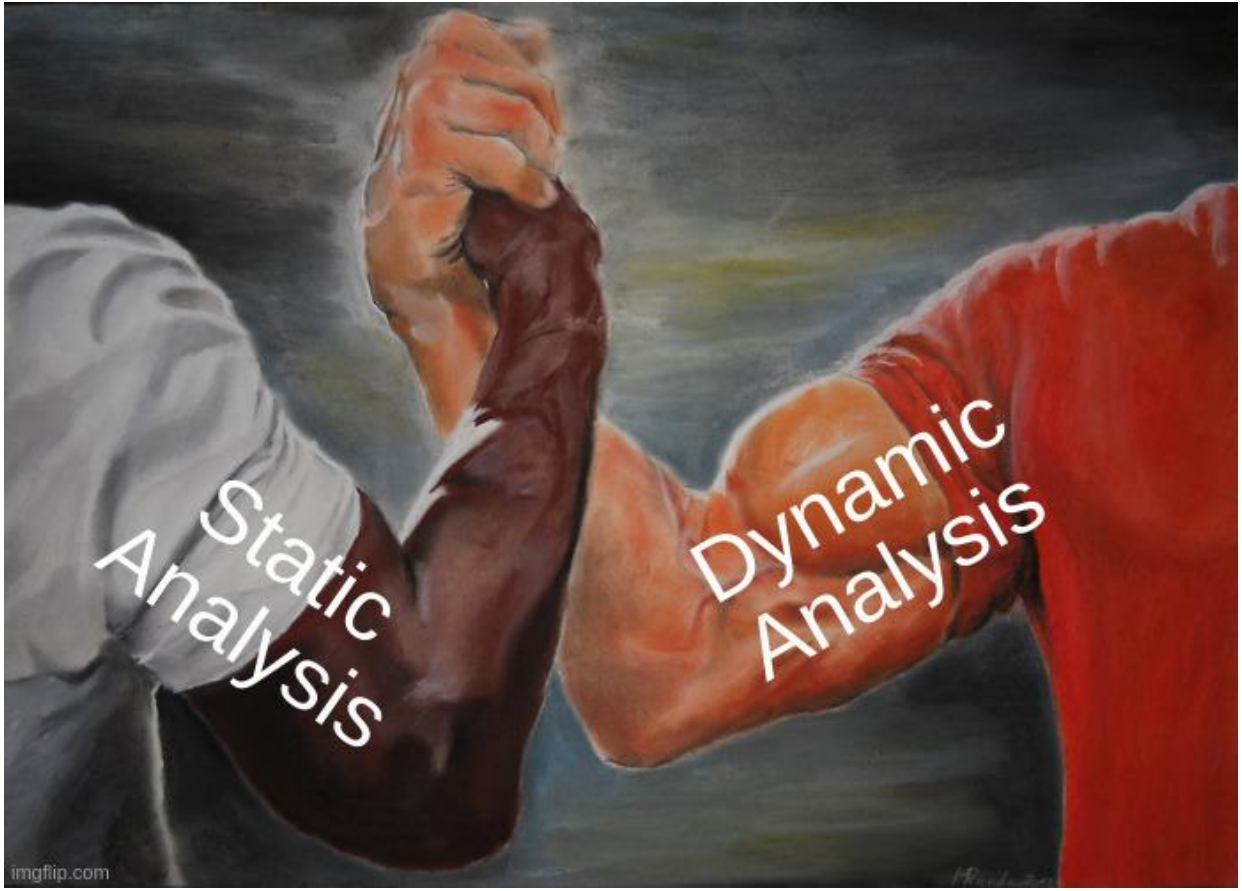


Program Behaviour

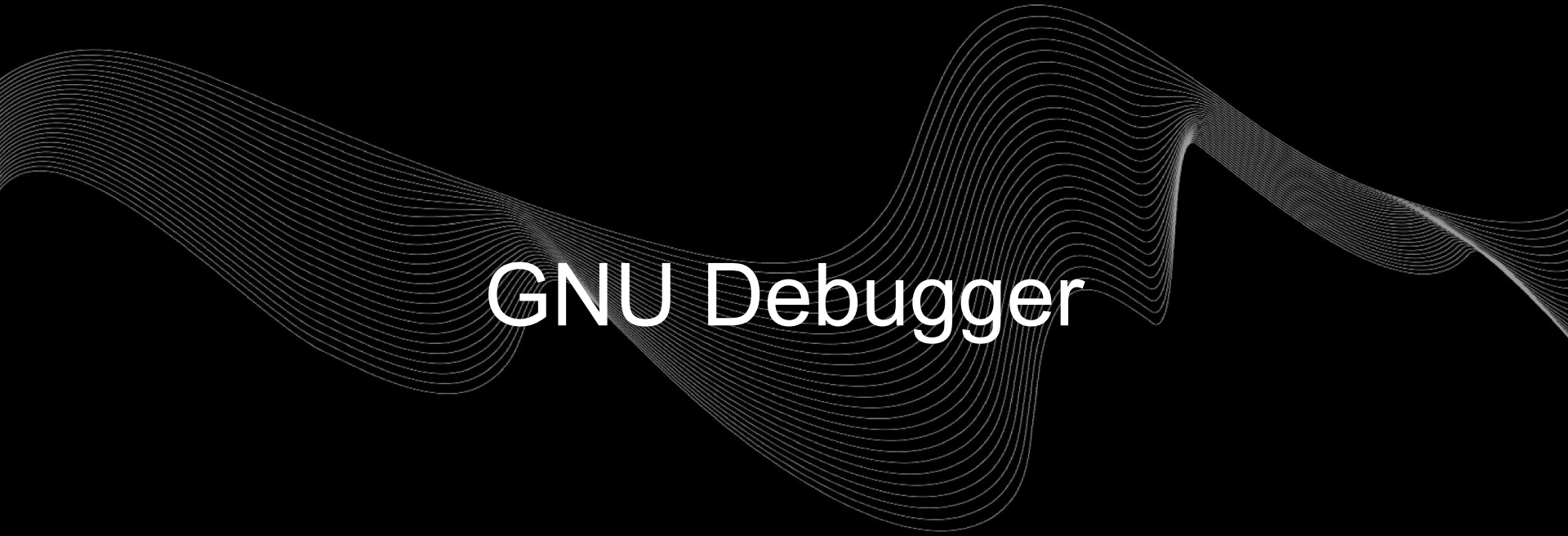


Desired Information



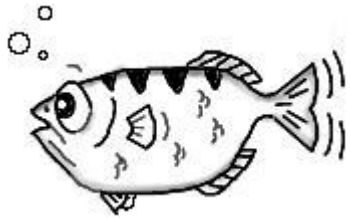

Static Analysis
Dynamic Analysis
imgflip.com

GNU Debugger



UNIVERSITY
OF TWENTE.

GNU Debugger



- Software debugger for most Unix-like systems
- Official support for 12 languages, such as C(++), Go and Rust
- Integrated in multiple IDE’s, such as:
  - CLion
  - Eclipse
  - Visual Studio Code
- Interfaces very nicely with Python

```
GNU gdb (GDB) 11.1
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb)
```

Pwndbg

- “Vanilla GDB is terrible to use for reverse engineering and exploit development.”
- Integrates with Ghidra/ Radare2 decompilation

Pwndbg

- “Vanilla GDB is terrible to use for reverse engineering and exploit development.”
- Integrates with Ghidra/ Radare2 decompilation

Pwndbg

- “Vanilla GDB is terrible to use for reverse engineering and exploit development.”
- Integrates with Ghidra/ Radare2 decompilation

```
► 0x5555555551a1 <main+62>          call   printf@plt <printf@plt>
      format: 0x555555556011 ◂— '3 * 2 + 6 = %d \n'
      vararg: 0xc
```

```
[ STACK ]
00:0000 rsp 0x7fffffffdc00 —▸ 0x7fffffffdd08 —▸ 0x7fffffffe0e4 ◂— '/home/yoep/Workspace/THS/thstest/hello_world'
01:0008     0x7fffffffdc08 ◂— 0x100000000
02:0010 rbp 0x7fffffffdc10 ◂— 0x0
03:0018     0x7fffffffdc18 —▸ 0x7ffff7df4b25 (__libc_start_main+213) ◂— mov    edi, eax
04:0020     0x7fffffffdc20 —▸ 0x7fffffffdd08 —▸ 0x7fffffffe0e4 ◂— '/home/yoep/Workspace/THS/thstest/hello_world'
05:0028     0x7fffffffdc28 ◂— 0x100000064 /* 'd' */
06:0030     0x7fffffffdc30 —▸ 0x555555555163 (main) ◂— push   rbp
07:0038     0x7fffffffdc38 ◂— 0x1000
```

Pwndbg

- “Vanilla GDB is terrible to use for reverse engineering and exploit development.”
- Integrates with Ghidra/ Radare2 decompilation

```
► 0x5555555551a1 <main+62>            call   printf@plt <printf@plt>
      format: 0x555555556011 ◂— '3 * 2 + 6 = %d \n'
      vararg: 0xc
```

```
[ STACK ]
00:0000 rsp 0x7fffffffdc00 —▸ 0x7fffffffdd08 —▸ 0x7fffffffe0e4 ◂— '/home/yoep/Workspace/THS/thstest/hello_world'
01:0008     0x7fffffffdc08 ◂— 0x100000000
02:0010 rbp 0x7fffffffdc10 ◂— 0x0
03:0018     0x7fffffffdc18 —▸ 0x7ffff7df4b25 (__libc_start_main+213) ◂— mov    edi, eax
04:0020     0x7fffffffdc20 —▸ 0x7fffffffdd08 —▸ 0x7fffffffe0e4 ◂— '/home/yoep/Workspace/THS/thstest/hello_world'
05:0028     0x7fffffffdc28 ◂— 0x100000064 /* 'd' */
06:0030     0x7fffffffdc30 —▸ 0x555555555163 (main) ◂— push   rbp
07:0038     0x7fffffffdc38 ◂— 0x1000
```

```
[ BACKTRACE ]
► f 0   0x7ffff7e25230 printf
  f 1   0x5555555551a6 main+67
  f 2   0x7ffff7df4b25 __libc_start_main+213
  f 3   0x55555555507e _start+46
```

GDB + pwndbg - Cheatsheet (1)

- **file [file]** load file [file] into gdb
- **set args [args]** set arguments of program
- **(r)un** run program until breakpoint
- **(k)ill** kill current program
- **(b)reak [where]** set breakpoint at
  - **function_name** known function
  - ***** **address** memory address
- **(i)nfo break** display breakpoints
- **delete/enable/disable [breakpoint]** modify existing breakpoint
- **(s)tep, (n)ext** advance program by 1 instruction
- **(c)ontinue** advance program to next break
- **finish** advance program to end of function call
- **(i)nfo** give info about program being debugged
- **(h)elp [command]** print info on command

GDB + pwndbg - Cheatsheet (2)

- **`(p)rint [expression]`** Print value of expression
  - **`variables`**
  - **`memory addresses`**
  - **`registers`**
  - **`arithmetic operations`**
  - **`casting / dereferencing`**
- **`x/(num)(format)(unit_size) [address]`** Inspect memory @ address
  - **`num`** number of units to print
  - **`format`** format character
  - **`unit_size`** size of the unit (b/h/w/g)
- **`dump (binary/ihex) memory [filename] [start_addr] [end_addr]`**
  Dump memory in range [start_addr, end_addr] in binary/ihex format to *filename*
- **`shell [command] [string]`** Execute shell command in gdb

Resources

- Sources
  - GDB (https://www.gnu.org/software/gdb/)
  - Pwndbg (https://github.com/pwndbg/pwndbg)
- Documentation
  - GDB (https://www.gnu.org/software/gdb/documentation/)
  - Pwndbg (https://browserpwndbg.readthedocs.io/en/docs/)
- Cheatsheets
  - Pwndbg features (https://github.com/pwndbg/pwndbg/blob/dev/FEATURES.md)
  - Darkdust cheatsheet (https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf)
  - Brown University cheatsheet (https://cs.brown.edu/courses/cs033/docs/guides/gdb.pdf)

Symbolic Execution

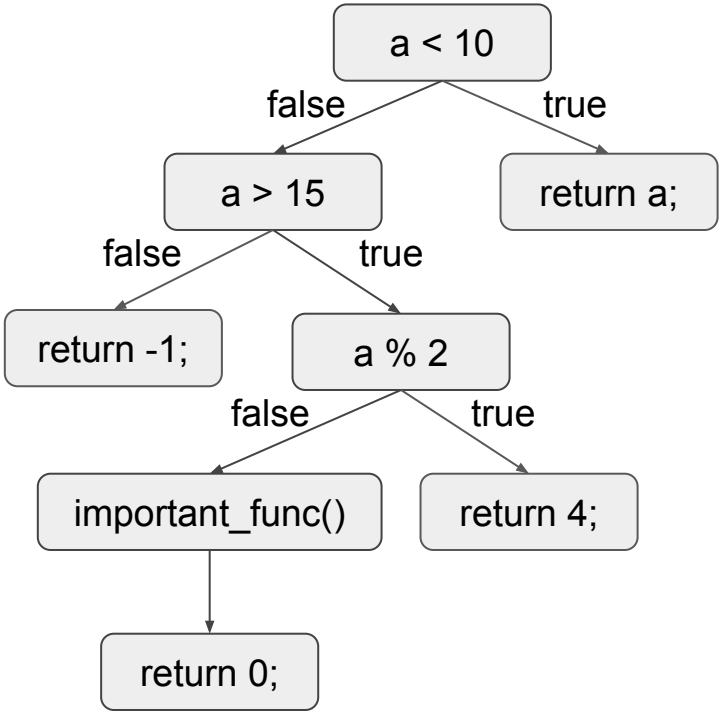
UNIVERSITY
OF TWENTE.

Symbolic Execution - Concept

```
int foo(int a) {
  if (a < 10) {
    return a;
  if (a > 15) {
    if (a % 2) {
      return 4;
    } else {
      important_func();
      return 0;
    }
  }
  return -1;
}
```

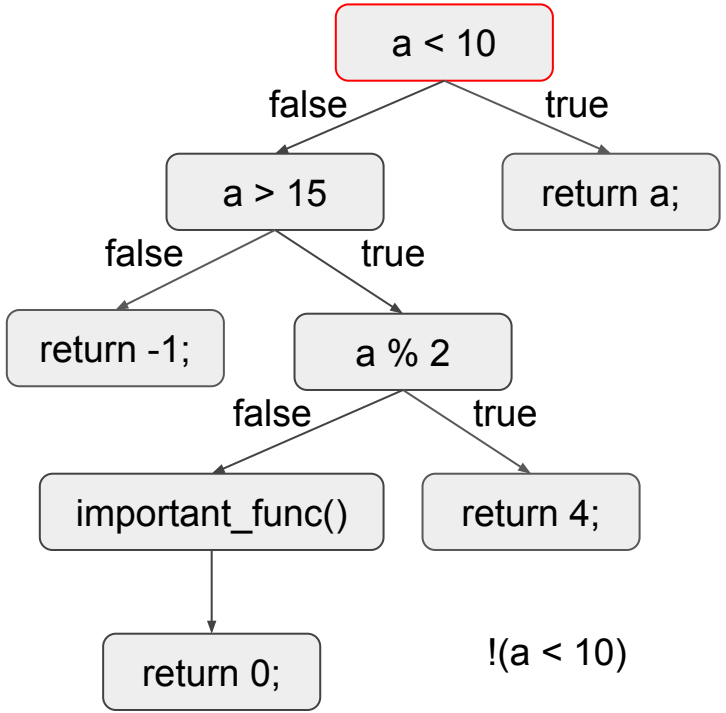
Symbolic Execution - Concept

```
int foo(int a) {
  if (a < 10) {
    return a;
  if (a > 15) {
    if (a % 2) {
      return 4;
    } else {
      important_func();
      return 0;
    }
  }
  return -1;
}
```



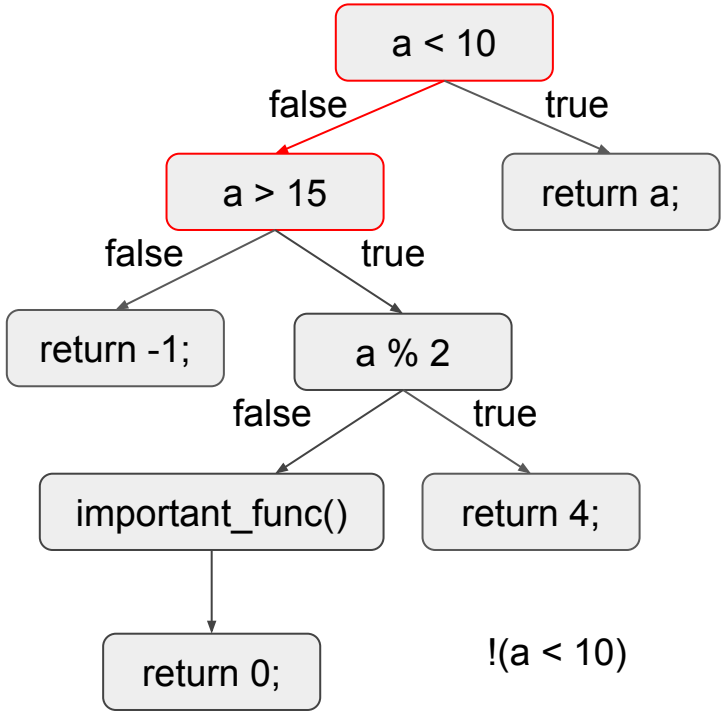
Symbolic Execution - Concept

```
int foo(int a) {
  if (a < 10) {
    return a;
  if (a > 15) {
    if (a % 2) {
      return 4;
    } else {
      important_func();
      return 0;
    }
  }
  return -1;
}
```



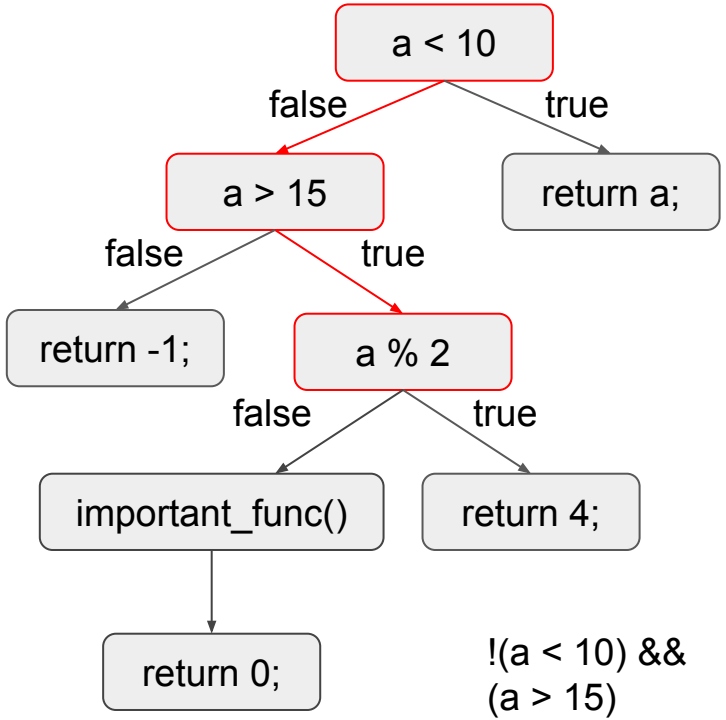
Symbolic Execution - Concept

```
int foo(int a) {
  if (a < 10) {
    return a;
  if (a > 15) {
    if (a % 2) {
      return 4;
    } else {
      important_func();
      return 0;
    }
  }
  return -1;
}
```



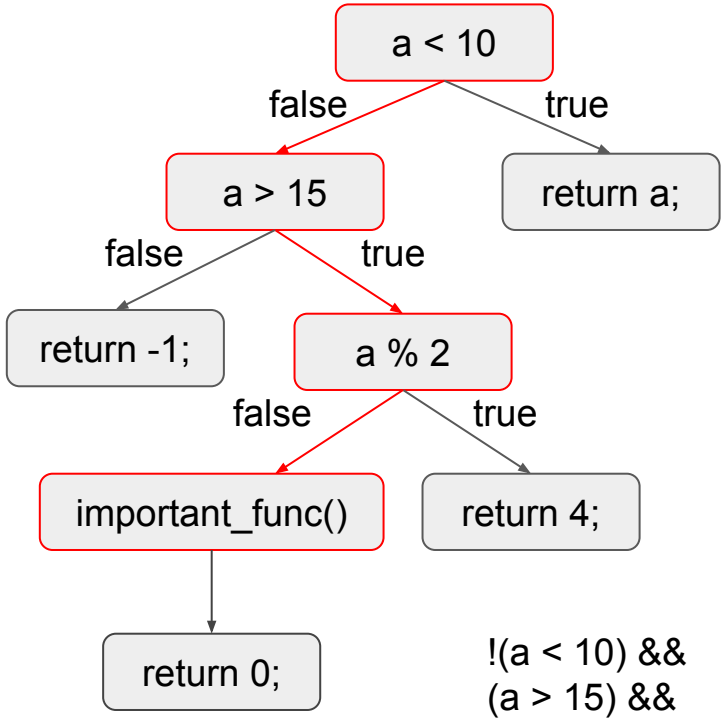
Symbolic Execution - Concept

```
int foo(int a) {
  if (a < 10) {
    return a;
  if (a > 15) {
    if (a % 2) {
      return 4;
    } else {
      important_func();
      return 0;
    }
  }
  return -1;
}
```



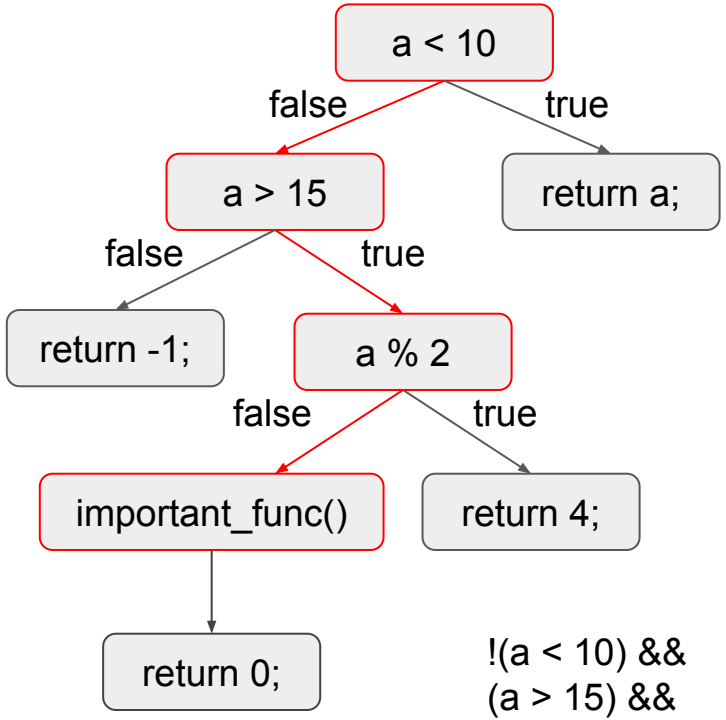
Symbolic Execution - Concept

```
int foo(int a) {
  if (a < 10) {
    return a;
  if (a > 15) {
    if (a % 2) {
      return 4;
    } else {
      important_func();
      return 0;
    }
  }
  return -1;
}
```



Symbolic Execution - Concept

```
int foo(int a) {
  if (a < 10) {
    return a;
  if (a > 15) {
    if (a % 2) {
      return 4;
    } else {
      important_func();
      return 0;
    }
  }
  return -1;
}
```



!(a < 10) && (a > 15) && !(a % 2)

a = 17

a = 19

Symbolic Execution - angr

- Python-based tool for:
  - Creating control-flow graphs
  - Performing symbolic execution
  - Automatically creating ROP chains (more on ROP chains will be explained during the powning tutorial)

- Powerful tool, however, if not initialized properly, it takes ages to run
  - “State explosion”

```
if (cond1) {
  if (cond2) {
    ...
      if (cond25021) {
        ...
} else {
  if (cond2_2) {
    ...
```

Resources

- angr website (https://angr.io/)
- angr documentation (https://docs.angr.io/)
- angr examples (https://github.com/angr/angr-doc/blob/master/docs/examples.md)
- angr API reference (https://angr.io/api-doc/)