Ghidra
An Open Source Reverse Engineering Tool

Lars A. Wallenborn

FrOSCon 2019, 10th August

How the NSA open-sourced all software in 2019

Intro

whoami

Lars A. Wallenborn
lars@wallenborn.net
@larsborn

whoami

Lars A. Wallenborn
lars@wallenborn.net
@larsborn

Since 2004 IT Freelancer
2013 Diploma in Mathematics @ Uni Bonn
2014 - 2015 Software Developer in Bonn
Since 2015: Security Researcher at CrowdStrike

Overview

1. What is Reverse Engineering?
2. Why should I do it?
3. How do I do it?

What is Reverse Engineering

What is Reverse Engineering

- *RE* or *reversing* for short

What is Reverse Engineering

- *RE* or *reversing* for short
- very general term:

What is Reverse Engineering

- *RE* or *reversing* for short
- very general term: process of ”reversing” the production process of an artificial object

What is Reverse Engineering

- *RE* or *reversing* for short
- very general term: process of ”reversing” the production process of an artificial object
- with the aim to reveal its designs, architecture, or – generally – to extract knowledge

What is Reverse Engineering

- *RE* or *reversing* for short
- very general term: process of ”reversing” the production process of an artificial object
- with the aim to reveal its designs, architecture, or – generally – to extract knowledge

This Presenation

What is Reverse Engineering

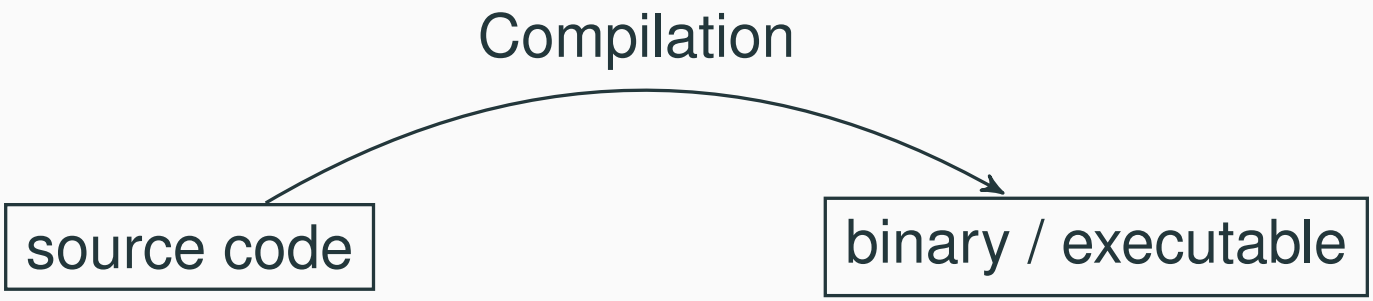
- *RE* or *reversing* for short
- very general term: process of ”reversing” the production process of an artificial object
- with the aim to reveal its designs, architecture, or – generally – to extract knowledge

This Presenation

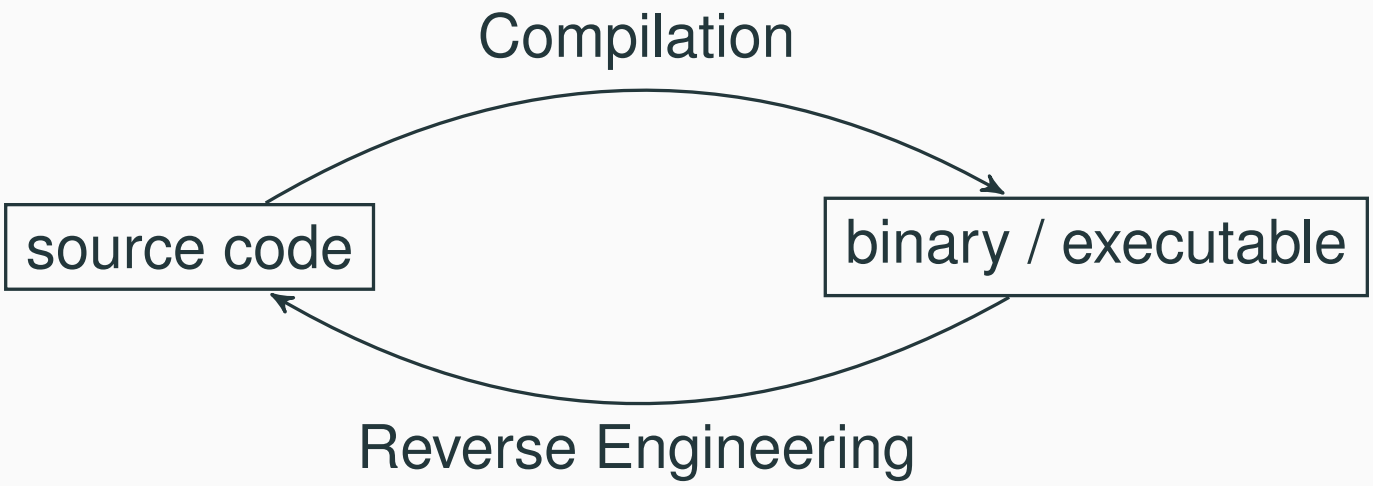
We will focus on a very specific kind of Reverse Engineering:

Binary Software Reverse Engineering

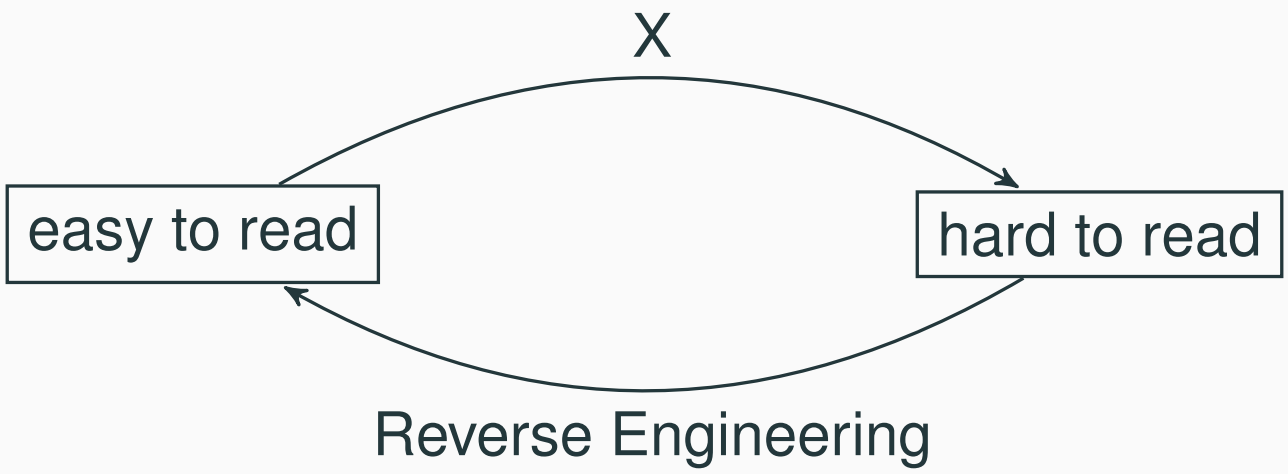
Binary Software Reverse Engineering



Binary Software Reverse Engineering



Binary Software Reverse Engineering – More General



Why should I do it?

Motivation

- Quality Assurance

Motivation

- Quality Assurance: Does it do what it is supposed to do?

Motivation

- Quality Assurance: Does it do what it is supposed to do?
- Interoperatibility

Motivation

- Quality Assurance: Does it do what it is supposed to do?
- Interoperatibility: A wild undocumented binary blog appears.

Motivation

- Quality Assurance: Does it do what it is supposed to do?
- Interoperatibility: A wild undocumented binary blog appears.
- Educational Purposes

Motivation

- Quality Assurance: Does it do what it is supposed to do?
- Interoperatibility: A wild undocumented binary blog appears.
- Educational Purposes: Excuse to hack.

Motivation

- Quality Assurance: Does it do what it is supposed to do?
- Interoperatibility: A wild undocumented binary blog appears.
- Educational Purposes: Excuse to hack.
- Malware Analysis

Motivation

- Quality Assurance: Does it do what it is supposed to do?
- Interoperatibility: A wild undocumented binary blog appears.
- Educational Purposes: Excuse to hack.
- Malware Analysis: Understand The Bad Guys^{TM}.

Motivation

- Quality Assurance: Does it do what it is supposed to do?
- Interoperatibility: A wild undocumented binary blog appears.
- Educational Purposes: Excuse to hack.
- Malware Analysis: Understand The Bad Guys^{TM}.
- Exploit Development

Motivation

- Quality Assurance: Does it do what it is supposed to do?
- Interoperatibility: A wild undocumented binary blog appears.
- Educational Purposes: Excuse to hack.
- Malware Analysis: Understand The Bad Guys^{TM}.
- Exploit Development: Are there bugs? Can I exploit them to make it behave in a way it was not intended?

Motivation

- Quality Assurance: Does it do what it is supposed to do?
- Interoperatibility: A wild undocumented binary blog appears.
- Educational Purposes: Excuse to hack.
- Malware Analysis: Understand The Bad Guys™.
- Exploit Development: Are there bugs? Can I exploit them to make it behave in a way it was not intended?
- Cracking

Motivation

- Quality Assurance: Does it do what it is supposed to do?
- Interoperatibility: A wild undocumented binary blog appears.
- Educational Purposes: Excuse to hack.
- Malware Analysis: Understand The Bad Guys^{TM}.
- Exploit Development: Are there bugs? Can I exploit them to make it behave in a way it was not intended?
- Cracking: How to circumvent copy right protection?

Motivation

- Quality Assurance: Does it do what it is supposed to do?
- Interoperatibility: A wild undocumented binary blog appears.
- Educational Purposes: Excuse to hack.
- Malware Analysis: Understand The Bad Guys^{TM}.
- Exploit Development: Are there bugs? Can I exploit them to make it behave in a way it was not intended?
- Cracking: How to circumvent copy right protection?
- Economic Espionage

Motivation

- Quality Assurance: Does it do what it is supposed to do?
- Interoperatibility: A wild undocumented binary blog appears.
- Educational Purposes: Excuse to hack.
- Malware Analysis: Understand The Bad Guys^{TM}.
- Exploit Development: Are there bugs? Can I exploit them to make it behave in a way it was not intended?
- Cracking: How to circumvent copy right protection?
- Economic Espionage: How does it work with the goal to reimplement it and then sell it.

Motivation

- Quality Assurance: Does it do what it is supposed to do?
- Interoperatibility: A wild undocumented binary blog appears.
- Educational Purposes: Excuse to hack.
- Malware Analysis: Understand The Bad Guys^{TM}.
- Exploit Development: Are there bugs? Can I exploit them to make it behave in a way it was not intended?
- Cracking: How to circumvent copy right protection?
- Economic Espionage: How does it work with the goal to reimplement it and then sell it.

I am not a lawyer

Motivation

- Quality Assurance: Does it do what it is supposed to do?
- Interoperatibility: A wild undocumented binary blog appears.
- Educational Purposes: Excuse to hack.
- Malware Analysis: Understand The Bad Guys™.
- Exploit Development: Are there bugs? Can I exploit them to make it behave in a way it was not intended?
- Cracking: How to circumvent copy right protection?
- Economic Espionage: How does it work with the goal to reimplement it and then sell it.

I am not a lawyer

But this is roughly sorted by how legal I think it is.

Motivation

- Quality Assurance: Does it do what it is supposed to do?
- Interoperatibility: A wild undocumented binary blog appears.
- Educational Purposes: Excuse to hack.
- Malware Analysis: Understand The Bad Guys™.
- Exploit Development: Are there bugs? Can I exploit them to make it behave in a way it was not intended?
- Cracking: How to circumvent copy right protection?
- Economic Espionage: How does it work with the goal to reimplement it and then sell it.

I am not a lawyer

But this is roughly sorted by how legal I think it is. In Germany.

Motivation

- Quality Assurance: Does it do what it is supposed to do?
- Interoperatibility: A wild undocumented binary blog appears.
- Educational Purposes: Excuse to hack.
- Malware Analysis: Understand The Bad Guys™.
- Exploit Development: Are there bugs? Can I exploit them to make it behave in a way it was not intended?
- Cracking: How to circumvent copy right protection?
- Economic Espionage: How does it work with the goal to reimplement it and then sell it.

I am not a lawyer

But this is roughly sorted by how legal I think it is. In Germany. On a sunny day.

How do I do it?

Show me an Example!

Show me an Example!

```c
#include <stdio.h>

int main() {
    printf("Hallo␣FrOSCon!");

    return 0;
}
```

Compile it!

Compile C program to a binary.

```
gcc main.c
```

Compile it!

Compile C program to a binary.

```
gcc main.c
```

```
strip a.exe
```

strings Based Reversing

Demo

What can we deduce already?

What can we deduce already?

- `!This program cannot be run in DOS mode.`

What can we deduce already?

- `!This program cannot be run in DOS mode.`
  $\Rightarrow$ It probably is a Windows executable.

What can we deduce already?

- `!This program cannot be run in DOS mode.`
  $\Rightarrow$ It probably is a Windows executable.
- `Hallo FrOSCon!`

What can we deduce already?

- `!This program cannot be run in DOS mode.`
  $\Rightarrow$ It probably is a Windows executable.
- `Hallo FrOSCon!`
  $\Rightarrow$ suggests that this is a ”hello world program”.

What can we deduce already?

- `!This program cannot be run in DOS mode.`
  $\Rightarrow$ It probably is a Windows executable.
- `Hallo FrOSCon!`
  $\Rightarrow$ suggests that this is a ”hello world program”.
- `Mingw-w64 runtime failure:`

What can we deduce already?

- `!This program cannot be run in DOS mode.`
  $\Rightarrow$ It probably is a Windows executable.
- `Hallo FrOSCon!`
  $\Rightarrow$ suggests that this is a ”hello world program”.
- `Mingw-w64 runtime failure:`
  $\Rightarrow$ probably compiled with MinGW (Minimalist GNU for Windows).

What can we deduce already?

- `!This program cannot be run in DOS mode.`
  $\Rightarrow$ It probably is a Windows executable.
- `Hallo FrOSCon!`
  $\Rightarrow$ suggests that this is a ”hello world program”.
- `Mingw-w64 runtime failure:`
  $\Rightarrow$ probably compiled with MinGW (Minimalist GNU for Windows).

Too many “probably”s and “suggests”s?

What can we deduce already?

- `!This program cannot be run in DOS mode.`
  $\Rightarrow$ It probably is a Windows executable.
- `Hallo FrOSCon!`
  $\Rightarrow$ suggests that this is a ”hello world program”.
- `Mingw-w64 runtime failure:`
  $\Rightarrow$ probably compiled with MinGW (Minimalist GNU for Windows).

Too many “probably”s and “suggests”s?

Due to time constraints while reversing, you often have to find a balance between speed and confidence.

Overview

1. What is Reverse Engineering?
2. Why should I do it?
3. How do I do it?

Overview

Static vs. Dynamic Reverse Engineering

Static vs. Dynamic Reverse Engineering

Static vs. Dynamic Reverse Engineering

Definition: Dynamic Reversing

Dynamic software reverse engineering is the analysis of computer software that is performed by executing programs on a real or virtual processor.

Static vs. Dynamic Reverse Engineering

Definition: Dynamic Reversing
Dynamic software reverse engineering is the analysis of computer software that is performed by executing programs on a real or virtual processor.

Definition: Static Reversing
Static software reverse engineering is the analysis of computer software that is performed without actually executing the target program.

Executable Formats

Executable Formats

- Depends on operating system

Executable Formats

- Depends on operating system. We will focus on Windows.

Executable Formats

- Depends on operating system. We will focus on Windows.
- Windows uses the Portable Executable (PE) format.

Executable Formats

- Depends on operating system. We will focus on Windows.
- Windows uses the Portable Executable (PE) format.
- RE techniques heavily depend on the used programming language. C, C++, Delphi, Go, .NET ...

Executable Formats

- Depends on operating system. We will focus on Windows.
- Windows uses the Portable Executable (PE) format.
- RE techniques heavily depend on the used programming language. C, C++, Delphi, Go, .NET ...
- Focus on ”native” PE files, i.e. files that are ”normal” Windows executables.

PE files

PE files

- PE files contain so-called *sections*.

PE files

- PE files contain so-called *sections*.
- Named like `.text`, `.data`, `.rdata` or `.bss`.

PE files

- PE files contain so-called *sections*.
- Named like `.text`, `.data`, `.rdata` or `.bss`.
- When the program is executed, the so-called *PE loader* copies the content of these sections to different regions in memory.

PE files

- PE files contain so-called *sections*.
- Named like `.text`, `.data`, `.rdata` or `.bss`.
- When the program is executed, the so-called *PE loader* copies the content of these sections to different regions in memory.
- Then, execution is handed over to the so-called *entry point* within the `.text` section.

Assembly

Assembly

Assembly

- Low-Level language executed by the CPU.

Assembly

- Low-Level language executed by the CPU.
- Only able to do very basic things.

Assembly

- Low-Level language executed by the CPU.
- Only able to do very basic things.
- Central concepts: Registers, Stack, Functions.

Assembly

- Low-Level language executed by the CPU.
- Only able to do very basic things.
- Central concepts: Registers, Stack, Functions.
- Often shown in disassembled state

Assembly

- Low-Level language executed by the CPU.
- Only able to do very basic things.
- Central concepts: Registers, Stack, Functions.
- Often shown in disassembled state: Instead of

```
4881ec98000000
```

Assembly

- Low-Level language executed by the CPU.
- Only able to do very basic things.
- Central concepts: Registers, Stack, Functions.
- Often shown in disassembled state: Instead of

```
4881ec98000000
```

we see

```
SUB RSP, 0x98
```

Tools

Tools

Tools

- IDA

Tools

- IDA (Interactive Disassembler)

Tools

- IDA (Interactive Disassembler) + HexRays Decompiler

Tools

- IDA (Interactive Disassembler) + HexRays Decompiler
- Binary Ninja

Tools

- IDA (Interactive Disassembler) + HexRays Decompiler
- Binary Ninja
- RetDec

Tools

- IDA (Interactive Disassembler) + HexRays Decompiler
- Binary Ninja
- RetDec (retargetable decompiler)

Tools

- IDA (Interactive Disassembler) + HexRays Decompiler
- Binary Ninja
- RetDec (retargetable decompiler)
- Ghidra

Ghidra

What is Ghidra?

- Existence is publicly known since the Vault7 leaks in 2017.

What is Ghidra?

- Existence is publicly known since the Vault7 leaks in 2017.
- At the RSA security conference 2019, the NSA announced to release it as open source software.

What is Ghidra?

- Existence is publicly known since the Vault7 leaks in 2017.
- At the RSA security conference 2019, the NSA announced to release it as open source software.
- Really did so in the following months.

What is Ghidra?

- Existence is publicly known since the Vault7 leaks in 2017.
- At the RSA security conference 2019, the NSA announced to release it as open source software.
- Really did so in the following months.
- JAVA-based GUI, backend written in C.

What is Ghidra?

- Existence is publicly known since the Vault7 leaks in 2017.
- At the RSA security conference 2019, the NSA announced to release it as open source software.
- Really did so in the following months.
- JAVA-based GUI, backend written in C.
- Capable of

What is Ghidra?

- Existence is publicly known since the Vault7 leaks in 2017.
- At the RSA security conference 2019, the NSA announced to release it as open source software.
- Really did so in the following months.
- JAVA-based GUI, backend written in C.
- Capable of *decompiling* native PEs

What is Ghidra?

- Existence is publicly known since the Vault7 leaks in 2017.
- At the RSA security conference 2019, the NSA announced to release it as open source software.
- Really did so in the following months.
- JAVA-based GUI, backend written in C.
- Capable of *decompiling* native PEs
- (und many other formats)

What can Ghidra do?

What can Ghidra do?

- Import executables and disassemble them

What can Ghidra do?

- Import executables and disassemble them
- Decompile the assembly and display pseudo code (C-like)

What can Ghidra do?

- Import executables and disassemble them
- Decompile the assembly and display pseudo code (C-like)
- Guess variable and function names when possible

What can Ghidra do?

- Import executables and disassemble them
- Decompile the assembly and display pseudo code (C-like)
- Guess variable and function names when possible
- Allow some basic refactoring similar to an integrated development environment (IDE)

How do I use it?

Demo

Thank You

Lars A. Wallenborn
lars@wallenborn.net
@larsborn

Some advertisement: I will give Reverse Engineering classes soon. If you are interested, talk to me or sent me an email!

Thank You

Lars A. Wallenborn
lars@wallenborn.net
@larsborn

Some advertisement: I will give Reverse Engineering classes soon. If you are interested, talk to me or sent me an email!

Questions?

Appendix: Static vs. Dynamic RE Comparison

Static vs. Dynamic Comparison Comparison

Static vs. Dynamic Comparison Comparison

Static Analyis	Dynamic Analyis

Static vs. Dynamic Comparison Comparison

Static Analyis	Dynamic Analyis
timeconsuming	

Static vs. Dynamic Comparison Comparison

Static Analyis	Dynamic Analyis
timeconsuming resource intensive (humans)	

Static vs. Dynamic Comparison Comparison

Static Analyis	Dynamic Analyis
timeconsuming	
resource intensive (humans)	
not fool-prove	

Static vs. Dynamic Comparison Comparison

Static Analyis	Dynamic Analyis
timeconsuming	evasion techniques (arms race)
resource intensive (humans)	
not fool-prove	

Static vs. Dynamic Comparison Comparison

Static Analyis	Dynamic Analyis
timeconsuming	evasion techniques (arms race)
resource intensive (humans)	resource intensive (computers)
not fool-prove	

Static vs. Dynamic Comparison Comparison

Static Analyis	Dynamic Analyis
timeconsuming	evasion techniques (arms race)
resource intensive (humans)	resource intensive (computers)
not fool-prove	not fool-prove

Static vs. Dynamic Comparison Comparison

Static Analyis	Dynamic Analyis
timeconsuming	evasion techniques (arms race)
resource intensive (humans)	resource intensive (computers)
not fool-prove	not fool-prove

Conclusion

A combined approach is the best of course.

We will focus on static analysis here.

Appendix: Assembly

Assembly: Registers

Assembly: Registers

- There are around 16 registers in a 64-bit CPU.

Assembly: Registers

- There are around 16 registers in a 64-bit CPU.
- Named like `RAX`, `RBP` or `R8`.

Assembly: Registers

- There are around 16 registers in a 64-bit CPU.
- Named like `RAX`, `RBP` or `R8`.
- Each register can store 64 bit of data.

Assembly: Registers

- There are around 16 registers in a 64-bit CPU.
- Named like `RAX`, `RBP` or `R8`.
- Each register can store 64 bit of data.
- They are extremly fast (even compared to RAM).

Assembly: Registers

- There are around 16 registers in a 64-bit CPU.
- Named like `RAX`, `RBP` or `R8`.
- Each register can store 64 bit of data.
- They are extremly fast (even compared to RAM).
- Example:

Assembly: Registers

- There are around 16 registers in a 64-bit CPU.
- Named like `RAX`, `RBP` or `R8`.
- Each register can store 64 bit of data.
- They are extremly fast (even compared to RAM).
- Example:

```
MOV RAX, 0x12
SUB RAX, 0x8
ADD RAX, 0x4
```

Assembly: Registers

- There are around 16 registers in a 64-bit CPU.
- Named like `RAX`, `RBP` or `R8`.
- Each register can store 64 bit of data.
- They are extremly fast (even compared to RAM).
- Example:

```
MOV RAX, 0x12
SUB RAX, 0x8
ADD RAX, 0x4
```

- Each instructions is made up of a *mnemonic* and (optionally) *arguments*.

Assembly: Registers

- There are around 16 registers in a 64-bit CPU.
- Named like `RAX`, `RBP` or `R8`.
- Each register can store 64 bit of data.
- They are extremly fast (even compared to RAM).
- Example:

```
MOV RAX, 0x12
SUB RAX, 0x8
ADD RAX, 0x4
```

- Each instructions is made up of a *mnemonic* and (optionally) *arguments*.
- Depending on how you count there are between 1000 and 4000 assembly instructions.

Assembly: Instruction Pointer

Assembly: Instruction Pointer

- There is a *very* special register: `RIP`.

Assembly: Instruction Pointer

- There is a *very* special register: `RIP`.
- It stores the address of the next assembly command that should be executed by the CPU.

Assembly: Instruction Pointer

- There is a *very* special register: `RIP`.
- It stores the address of the next assembly command that should be executed by the CPU.
- Yes, the program lives in the same space as the data.

Assembly: Instruction Pointer

- There is a *very* special register: `RIP`.
- It stores the address of the next assembly command that should be executed by the CPU.
- Yes, the program lives in the same space as the data.
- This is the cause of *many* problems we have with computers nowadays.

Assembly: Instruction Pointer

- There is a *very* special register: `RIP`.
- It stores the address of the next assembly command that should be executed by the CPU.
- Yes, the program lives in the same space as the data.
- This is the cause of *many* problems we have with computers nowadays.

Assembly: Stack

Assembly: Stack

- Central datastructure.

Assembly: Stack

- Central datastructure.
- Resides in RAM.

Assembly: Stack

- Central datastructure.
- Resides in RAM.
- Literally a *stack*.

Assembly: Stack

- Central datastructure.
- Resides in RAM.
- Literally a *stack*.
- `PUSH` and `POP` can be used to manipulate it.

Assembly: Stack

- Central datastructure.
- Resides in RAM.
- Literally a *stack*.
- `PUSH` and `POP` can be used to manipulate it.
- Registers `RSP` and `RBP` store its location.

Assembly: Stack

- Central datastructure.
- Resides in RAM.
- Literally a *stack*.
- `PUSH` and `POP` can be used to manipulate it.
- Registers `RSP` and `RBP` store its location.
- Example:

Assembly: Stack

- Central datastructure.
- Resides in RAM.
- Literally a *stack*.
- `PUSH` and `POP` can be used to manipulate it.
- Registers `RSP` and `RBP` store its location.
- Example:

```
PUSH RBX
PUSH 0x98
POP RBX
```

Assembly: Functions

Assembly: Functions

- Often functions in C are compiled to functions in assembly.

Assembly: Functions

- Often functions in C are compiled to functions in assembly.
- `CALL` and `RET` are the responsible mnemonics.

Assembly: Functions

- Often functions in C are compiled to functions in assembly.
- `CALL` and `RET` are the responsible mnemonics.
- Example:

Assembly: Functions

- Often functions in C are compiled to functions in assembly.
- `CALL` and `RET` are the responsible mnemonics.
- Example:

```
CALL f5 15 00 00
CALL printf
RET
```

Assembly: Functions

- Often functions in C are compiled to functions in assembly.
- `CALL` and `RET` are the responsible mnemonics.
- Example:

```
CALL f5 15 00 00
CALL printf
RET
```

- `CALL` pushes `EIP` to the stack

Assembly: Functions

- Often functions in C are compiled to functions in assembly.
- `CALL` and `RET` are the responsible mnemonics.
- Example:

```
CALL f5 15 00 00
CALL printf
RET
```

- `CALL` pushes `EIP` to the stack
- And then sets its value to the given argument

Assembly: Functions

- Often functions in C are compiled to functions in assembly.
- `CALL` and `RET` are the responsible mnemonics.
- Example:

```
CALL f5 15 00 00
CALL printf
RET
```

- `CALL` pushes `EIP` to the stack
- And then sets its value to the given argument
- Effectively continuing execution in the function

Assembly: Functions

- Often functions in C are compiled to functions in assembly.
- `CALL` and `RET` are the responsible mnemonics.
- Example:

```
CALL f5 15 00 00
CALL printf
RET
```

- `CALL` pushes `EIP` to the stack
- And then sets its value to the given argument
- Effectively continuing execution in the function
- `RET` does the inverse.

Assembly: Functions

- Often functions in C are compiled to functions in assembly.
- `CALL` and `RET` are the responsible mnemonics.
- Example:

```
CALL f5 15 00 00
CALL printf
RET
```

- `CALL` pushes `EIP` to the stack
- And then sets its value to the given argument
- Effectively continuing execution in the function
- `RET` does the inverse.
- This allow arbitrarily deeply nested calls.