Binary Analysis - 2

REVERSE ENGINEERING

deti universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

João Paulo Barraca

Binary Analysis Process

- Up to now we know how ELF files are structure, but the question remains: how do we analyse ELF files?
  - Or any other binary

- A possible flow can be:
  - File analysis (file, nm, ldd, content visualization, foremost, binwalk)
  - Static Analysis (disassemblers and decompilers
  - Behavioral Analysis (strace, LD_PRELOAD)
  - Dynamic Analysis (debuggers)

Identifying a file

- Files should be seen as containers (this includes ELF files)
  - May have the expected content type
    - But it may have an unexpected behavior (e.g. bug or malware)
  - May have unexpected, additional content (e.g. polyglots)
    - More common in DRM schemes and malware in order to hide binary blobs

- Files should not be trusted
  - Both the expected and additional content may be malicious
  - Static analysis is safe (as long as nothing is executed)
  - Dynamic analysis is not safe. Sandboxes and VMs must be used

Questions to answer

- What type of file we have?
  - Are there hidden contents?
- What is the architecture?
- Is it 64/32 or ARM7/ARM9/ARM9E/ARM10?
- Where is the starting address?
- What the main function does?
- What will the program will actually do?

Questions to answer

Some basic tools go a long way

- **`file`**: (try to identify) the type of file
  - Only applies to a top container. File is not able to look into enclosed binary blobs
  - Alternatives that complement **`file`** are **`binwalk`** and **`foremost`**

- **`xxd`**: hexdump the file, allowing to rapidly detect patterns
  - less also helps to hold the content in the terminal

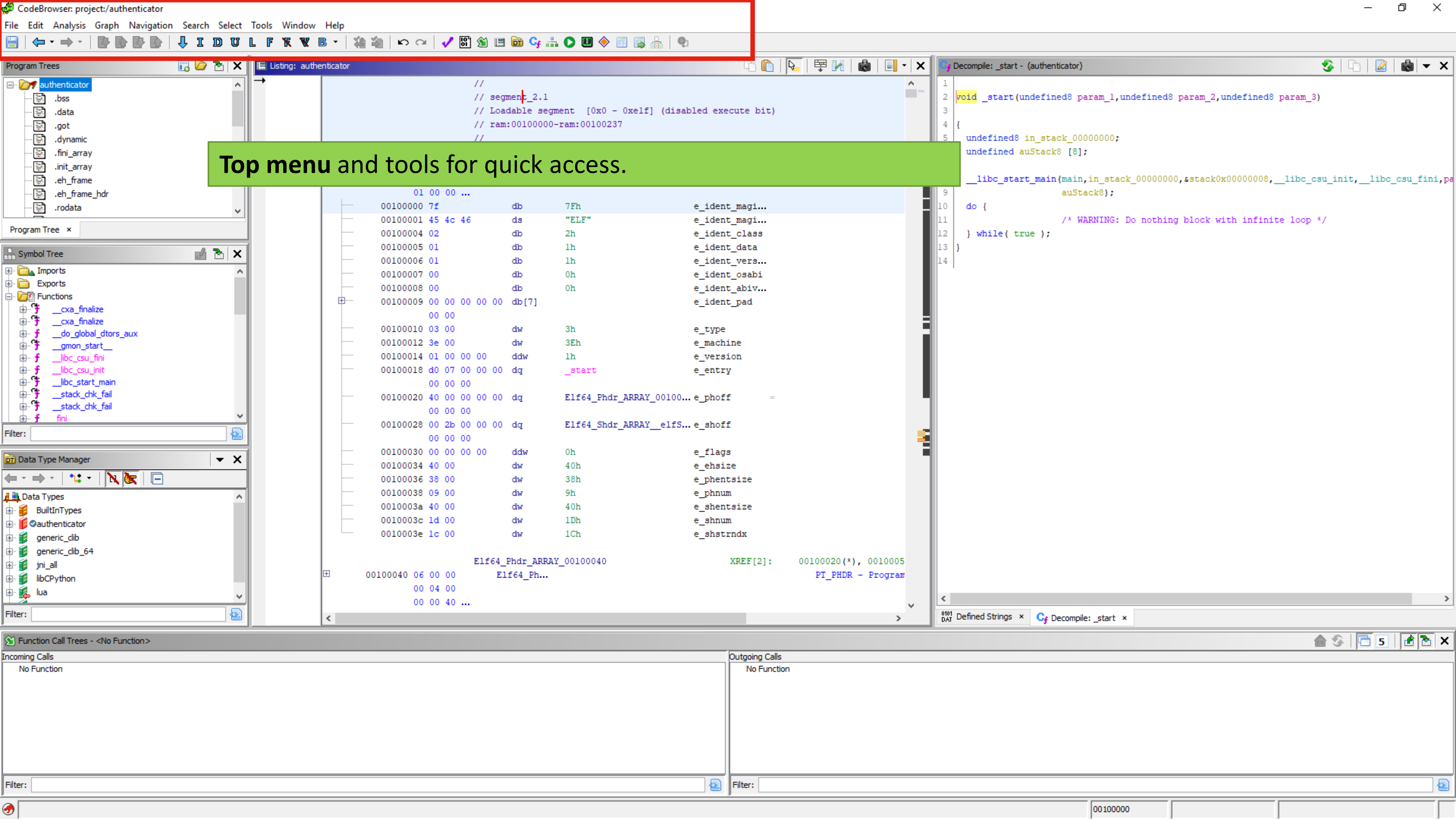
- **`strings`**: prints null terminated sequence chars
  - By default, with more than 4 characters (-n setting)

- **`ldd`**: print shared object dependencies
  - Libraries registered in the ELF that are required (typically for dynamically relocate symbols)

- **nm**: dumps symbols from `.symtab` (or `.dyntab` with –D)

Disassembler basics with ghidra

- ghidra is a open source tool developed by NSA and released to the public doing Disassembly and Static Analysis
  - Development branch has support for Dynamic Analysis (should be released "soon")
- Works on Windows, Linux and macos
  - Java based
- Not the most important tool (IDA is), but is gaining a huge traction
  - It's free and very powerful with a huge number of platforms and a **fine decompiler**

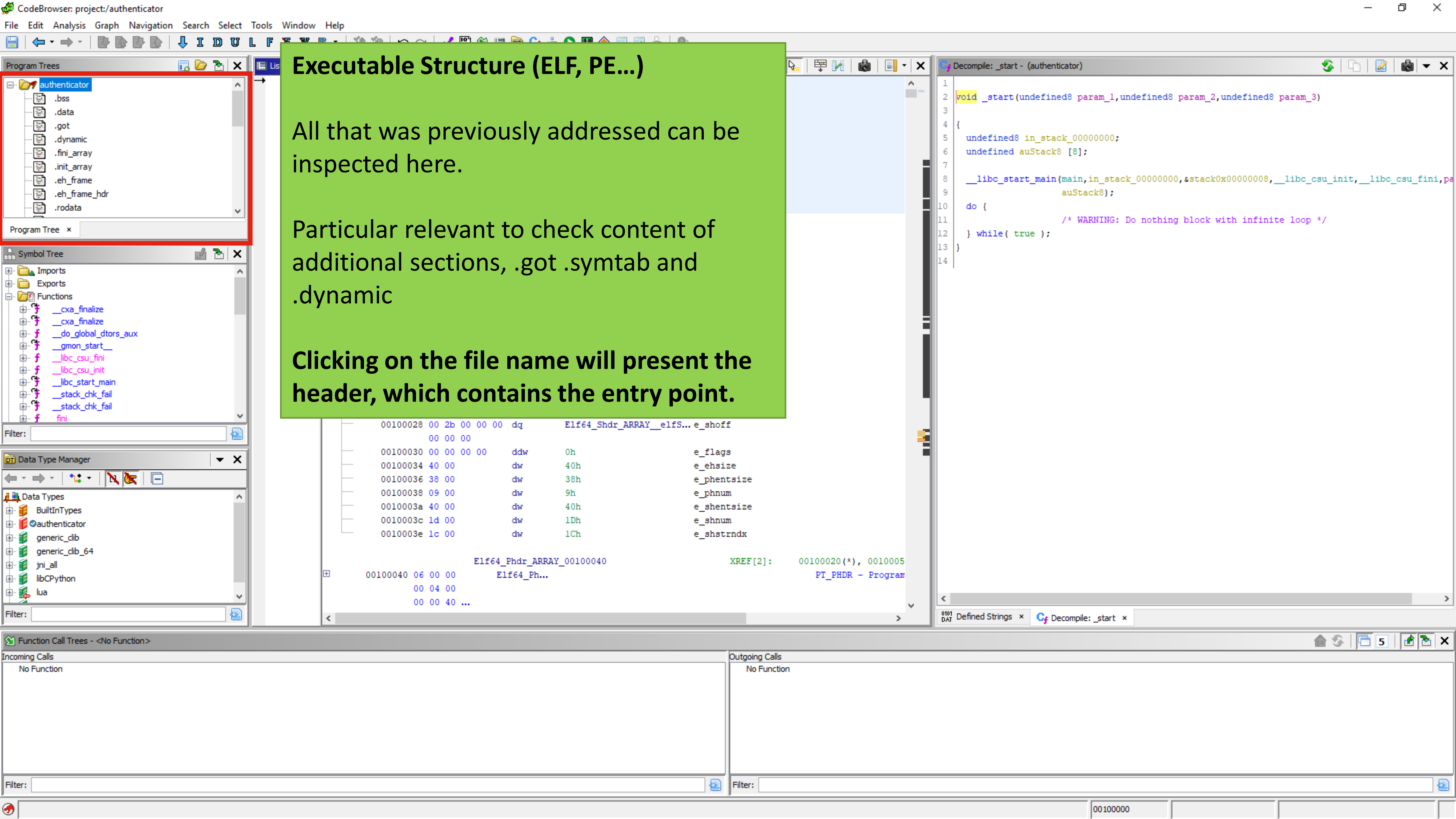

Top menu and tools for quick access.
CodeBrowser: project:/authenticator
File Edit Analysis Graph Navigation Search Select Tools Window Help
Program Trees
Symbol Tree
Data Type Manager
Listing: authenticator
Decompile: _start - (authenticator)
Function Call Trees - <No Function>
Incoming Calls
Outgoing Calls
No Function

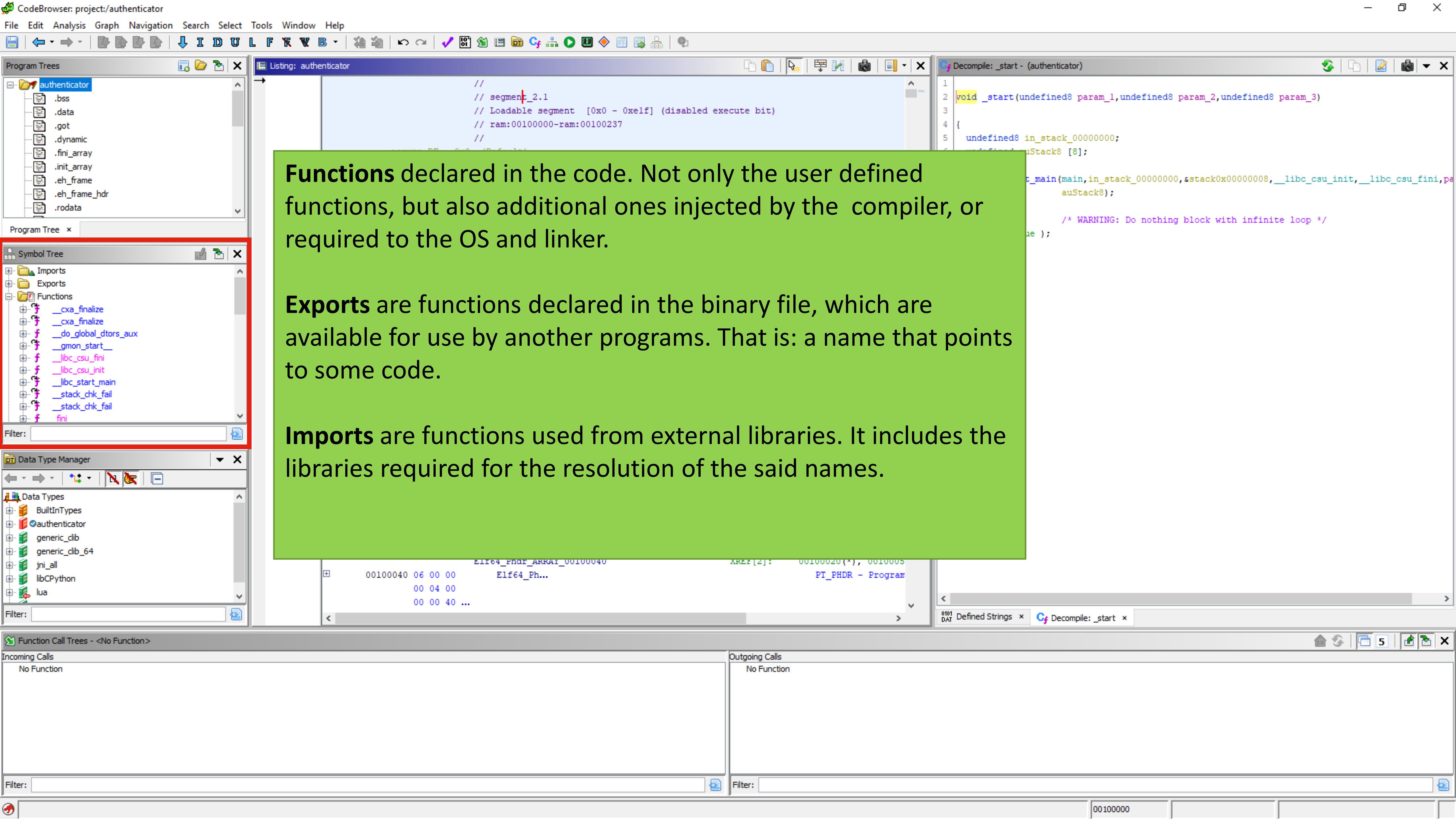
Executable Structure (ELF, PE…)

All that was previously addressed can be inspected here.

Particular relevant to check content of additional sections, .got .symtab and .dynamic

Clicking on the file name will present the header, which contains the entry point.

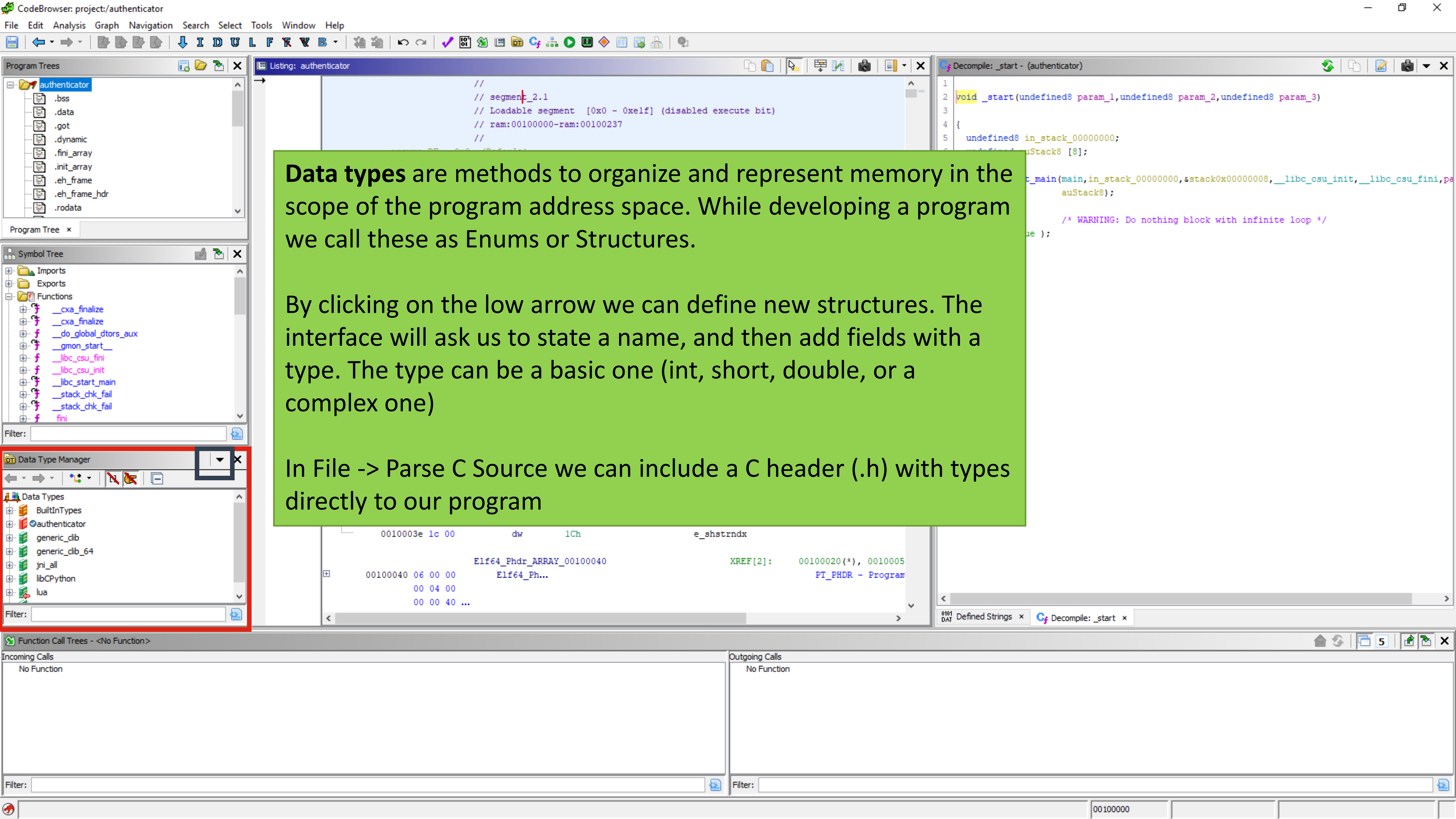




Functions declared in the code. Not only the user defined functions, but also additional ones injected by the compiler, or required to the OS and linker.

Exports are functions declared in the binary file, which are available for use by another programs. That is: a name that points to some code.

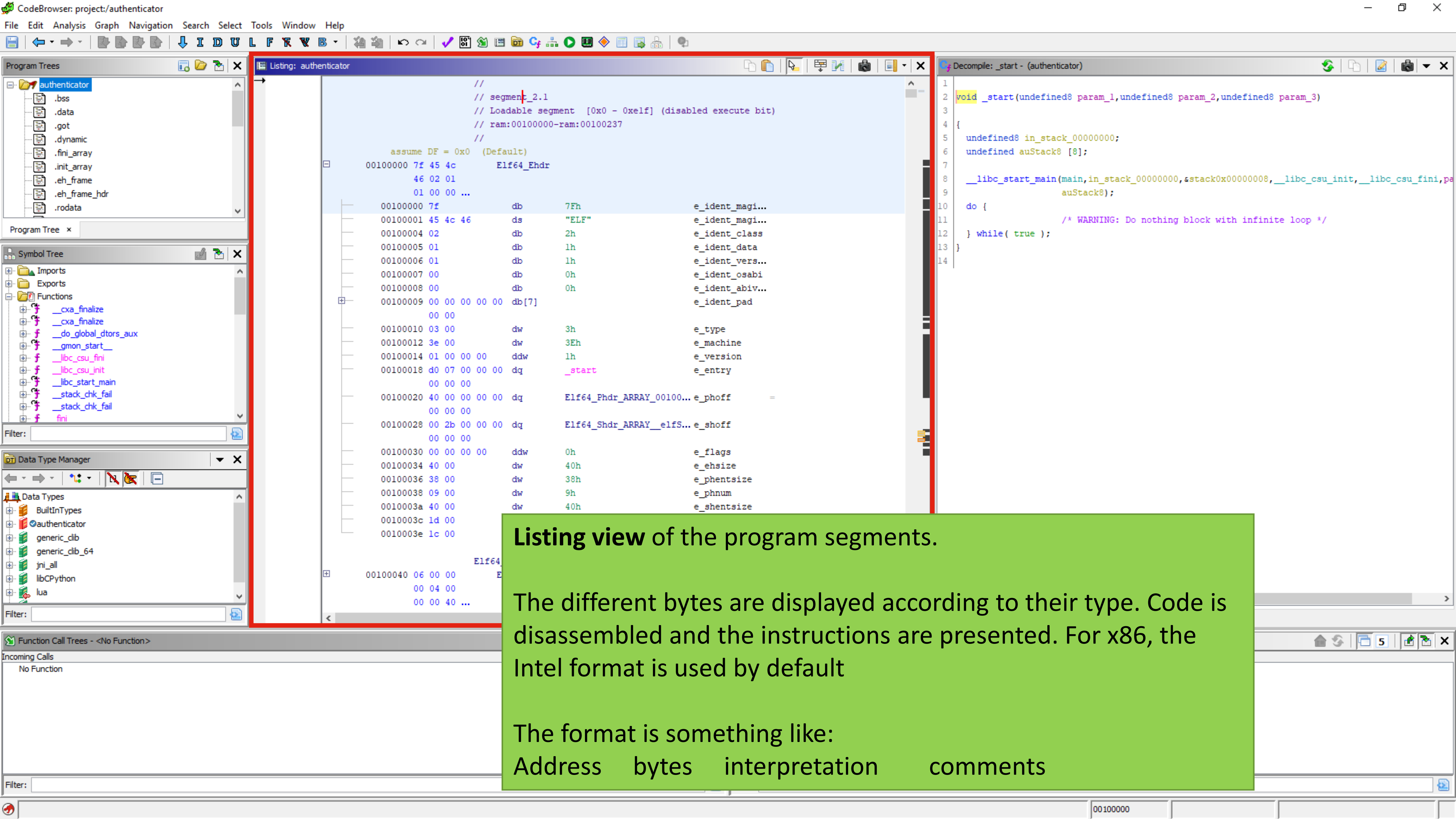
Imports are functions used from external libraries. It includes the libraries required for the resolution of the said names.



Data types are methods to organize and represent memory in the scope of the program address space. While developing a program we call these as Enums or Structures.

By clicking on the low arrow we can define new structures. The interface will ask us to state a name, and then add fields with a type. The type can be a basic one (int, short, double, or a complex one)

In File -> Parse C Source we can include a C header (.h) with types directly to our program

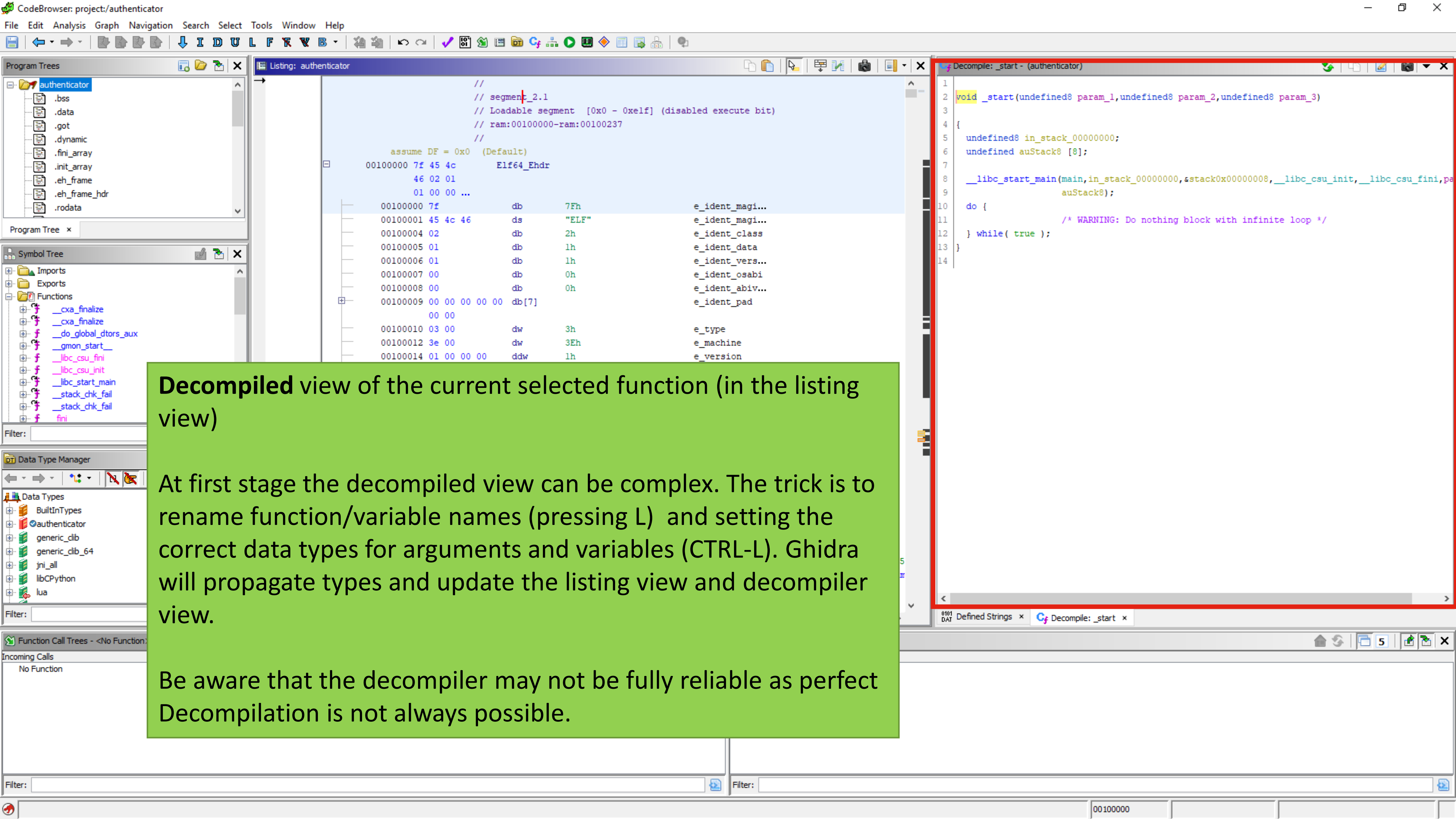


Listing view of the program segments.

The different bytes are displayed according to their type. Code is disassembled and the instructions are presented. For x86, the Intel format is used by default

The format is something like:

Address bytes interpretation comments



Decompiled view of the current selected function (in the listing view)

At first stage the decompiled view can be complex. The trick is to rename function/variable names (pressing L) and setting the correct data types for arguments and variables (CTRL-L). Ghidra will propagate types and update the listing view and decompiler view.

Be aware that the decompiler may not be fully reliable as perfect Decompilation is not always possible.

Listing view presents functions with name, if the name is in the .dynamic or .symtab.

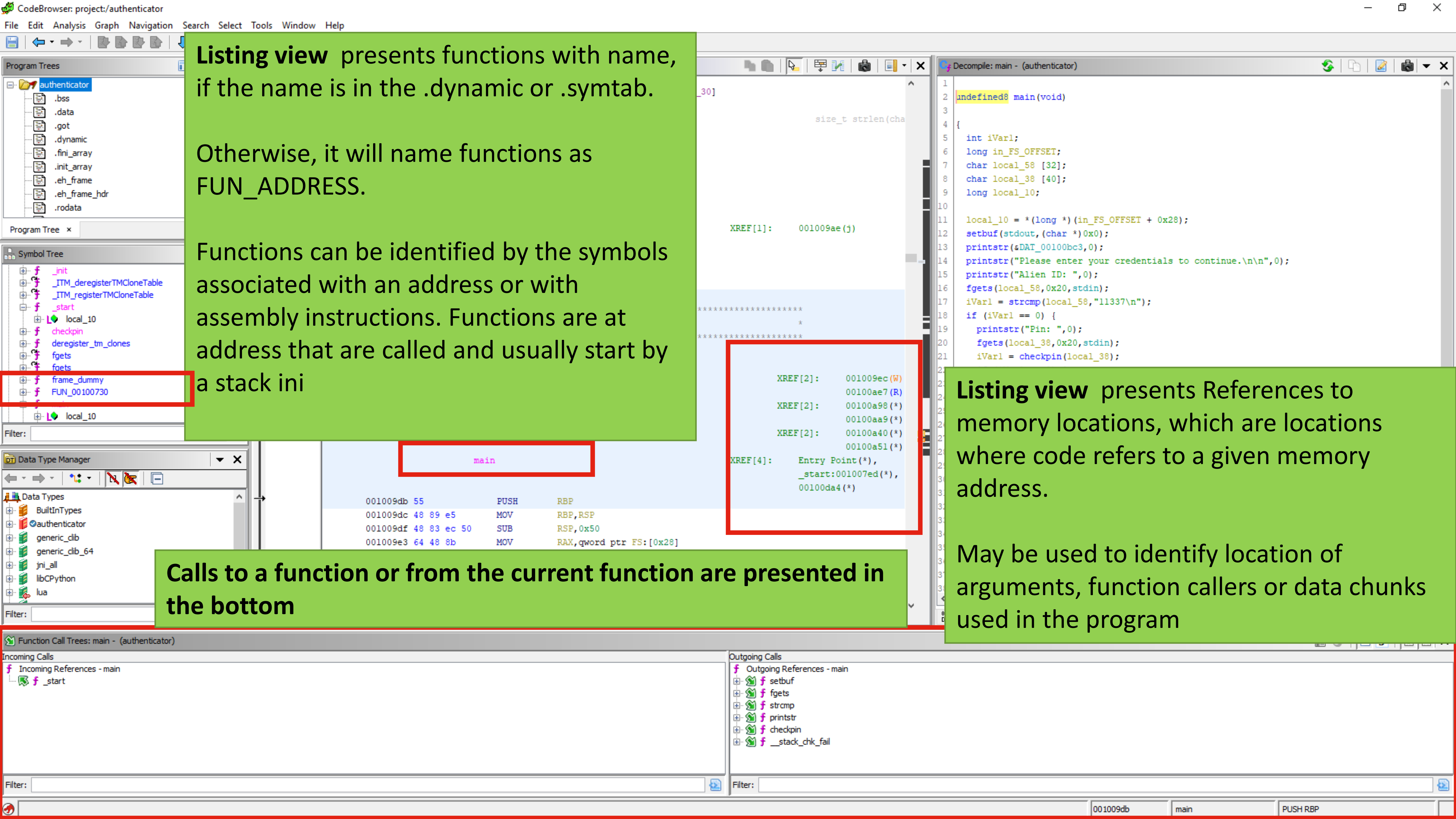
Otherwise, it will name functions as FUN_ADDRESS.

Functions can be identified by the symbols associated with an address or with assembly instructions. Functions are at address that are called and usually start by a stack ini

Listing view presents References to memory locations, which are locations where code refers to a given memory address.

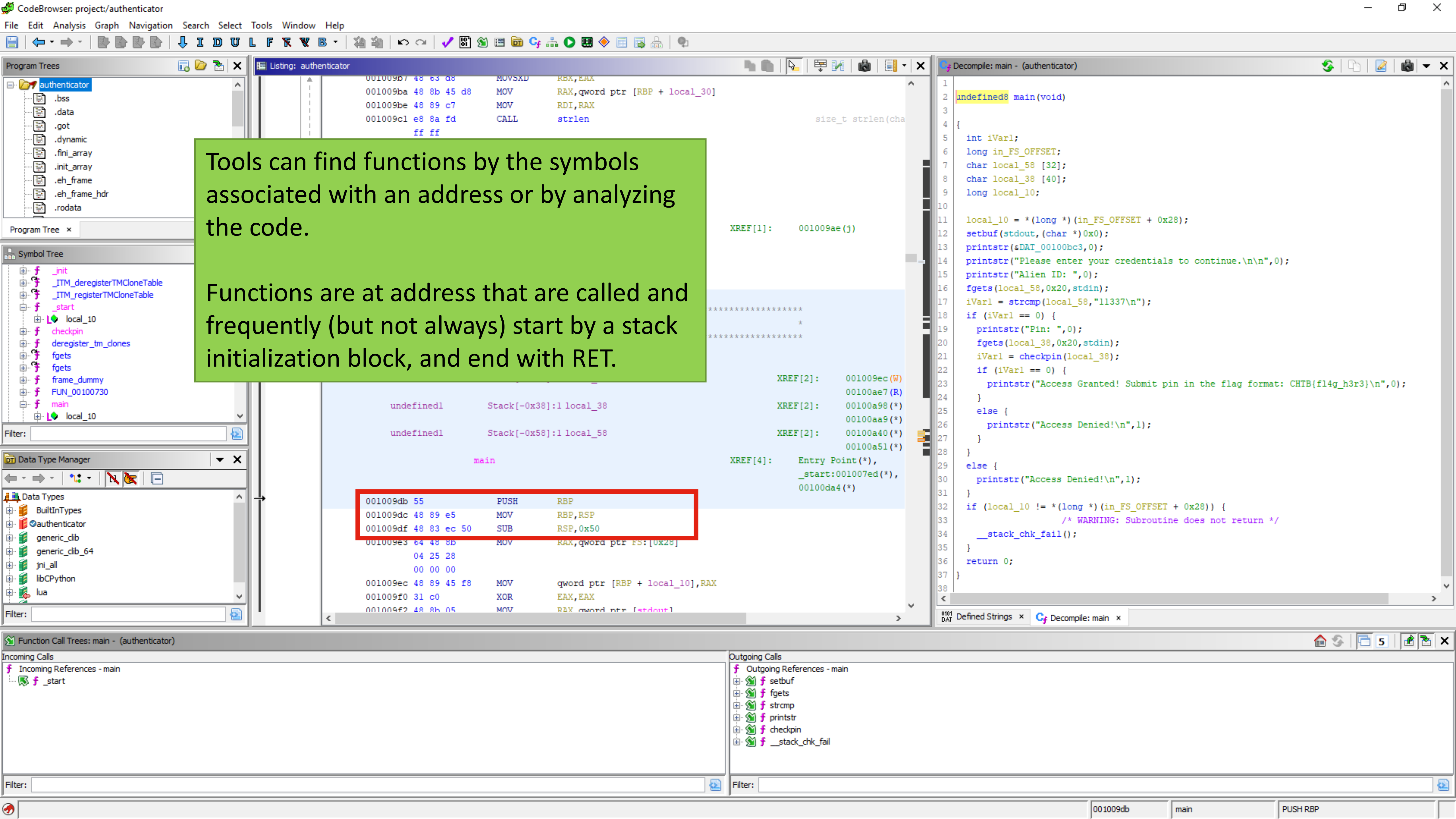
May be used to identify location of arguments, function callers or data chunks used in the program

Calls to a function or from the current function are presented in the bottom



Tools can find functions by the symbols associated with an address or by analyzing the code.

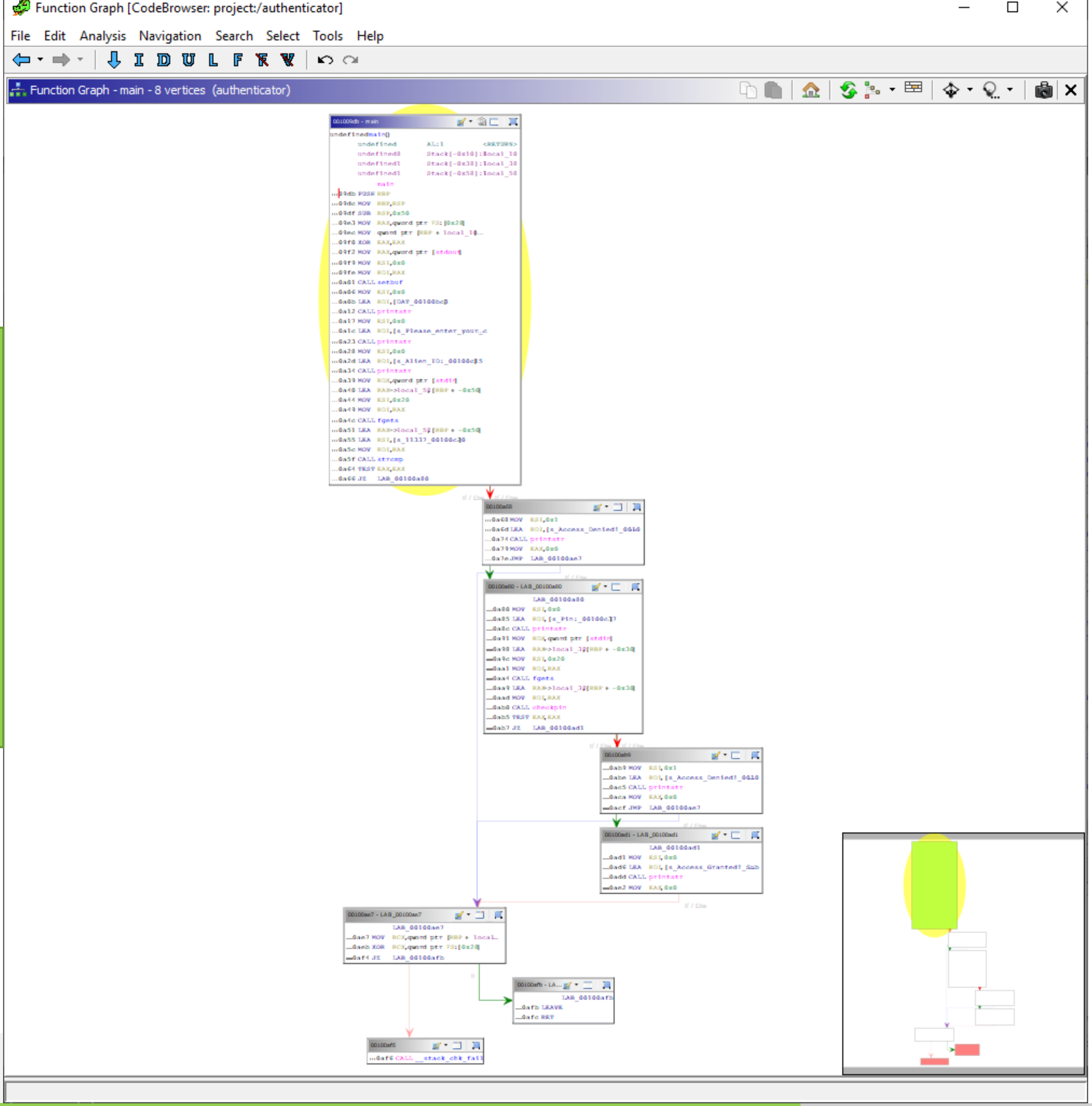
Functions are at address that are called and frequently (but not always) start by a stack initialization block, and end with RET.



In Menu->Window->Function Graph

A logical structure of the function is presented. This is generated by interpreting the branches that segment the function code.

Called: Control-Flow Graphs

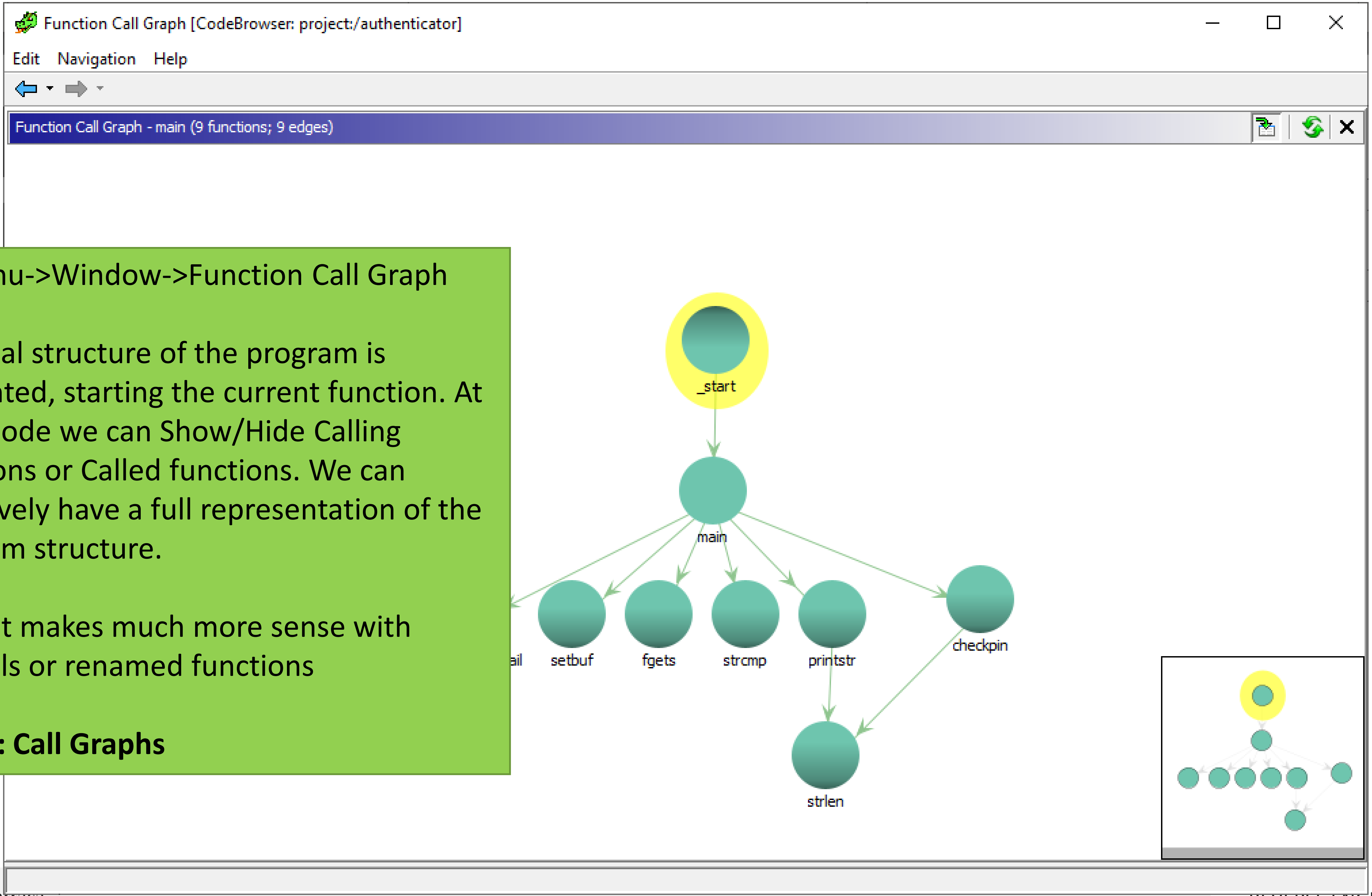


In Menu->Window->Function Call Graph

A logical structure of the program is presented, starting the current function. At each node we can Show/Hide Calling functions or Called functions. We can effectively have a full representation of the program structure.

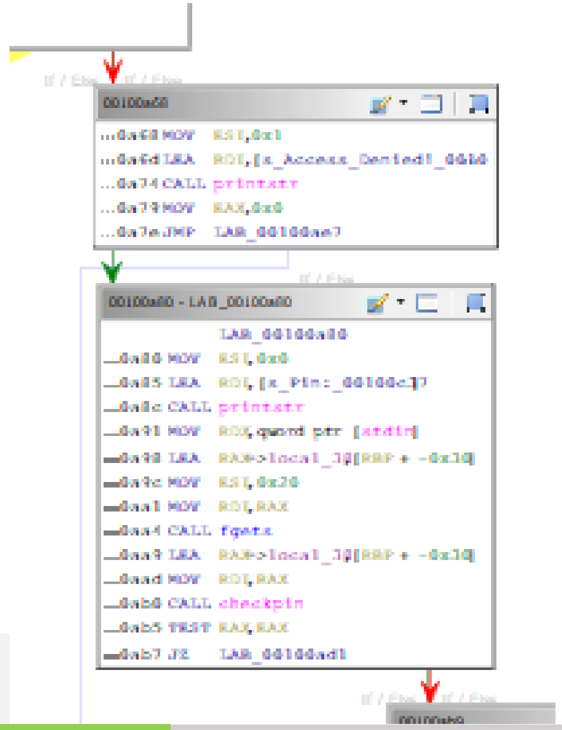
HINT: It makes much more sense with symbols or renamed functions

Called: Call Graphs



CFGs

- It is useful to think of machine code in a graph structure, called a control-flow graph
- A node in a CFG is a group of adjacent instructions called a **basic block**:
  - The only jumps into a basic block are to the first instruction
  - The only jumps out of a basic block are from the last instruction
  - I.e., a basic block always executes as a unit



- Edges between blocks represent possible jumps

CFGs

- Basic block ***a*** dominates basic block ***b*** if every path to ***b*** passes through ***a*** first
  - strictly dominates if ***a != b***

- Basic block ***b*** post-dominates ***a*** if every path through ***a*** also passes through ***b*** later

Disassembly

- The disassembly process involves analyzing the binary, converting binary code to assembly
  - But “binary” is just a sequence of bytes, that must be mapped in the scope of a given architecture
  - Conversion depends on many factors, including compiler and flags

- Process is not perfect and may induce RE Analysts in error
  - Present instructions that actually do not exist
  - Ignore instructions that are in the binary code

- Main approaches:
  - Linear Disassembly
  - Recursive Disassembly

Linear Disassembly

- Simplest approach towards analyzing a program: **Iterate over all code segments, disassembling the binary code as opcodes are found**

- Start at some address and follow the binary
  - Entry point or other point in the binary file
  - Entry point may not be known

- Works best with:
  - binary blobs such as from firmwares (start at the beginning)
  - objects which do not have data at the beginning
  - architecture uses variable length instructions (x86)

Linear Disassembly

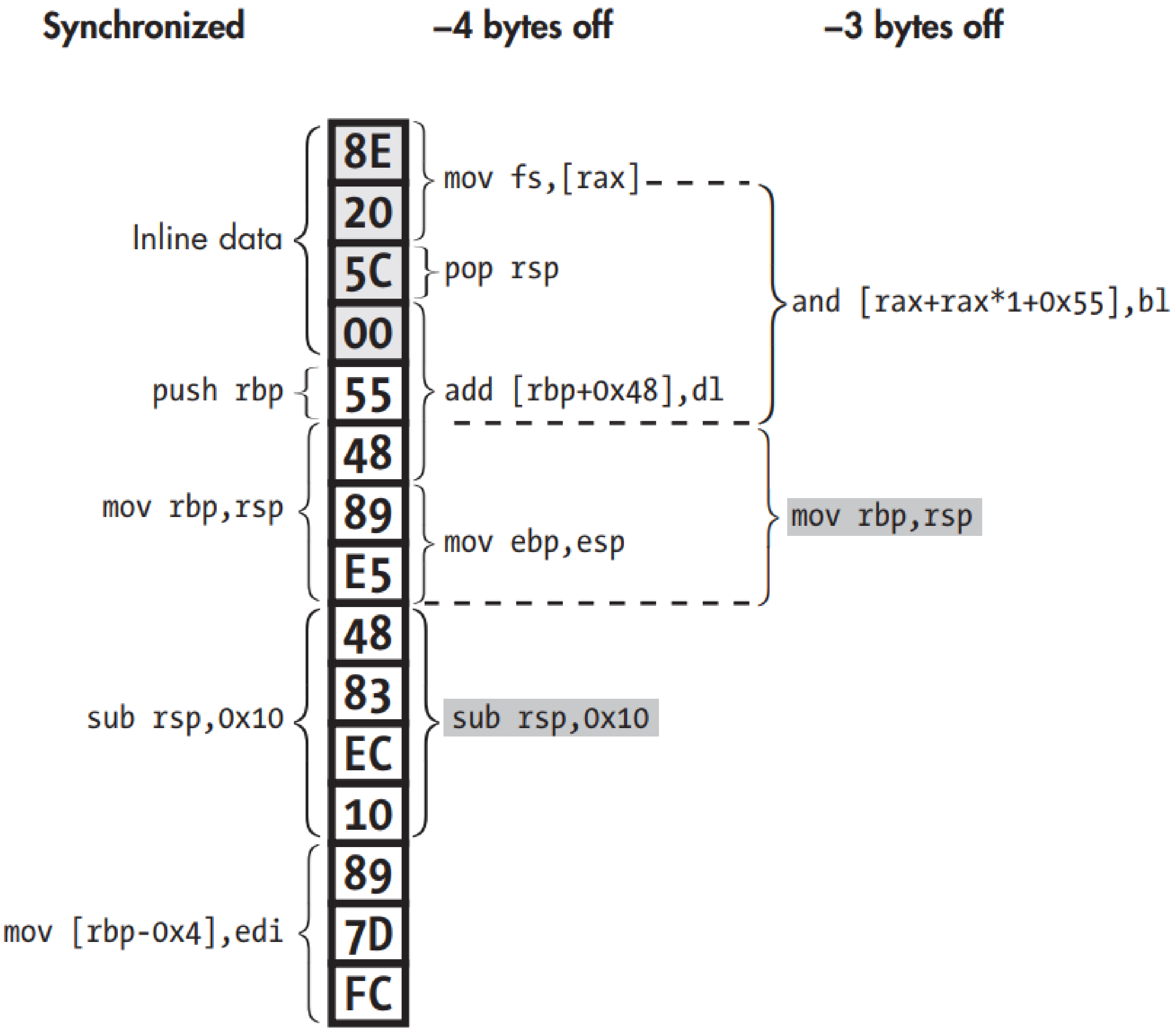
It is vital to define the initial address for decompiling.

An offset error will result in invalid or wrong instructions being decoded.

Linear disassembly will also try to disassemble data from the binary as if it was actual code.

Linear Disassembly is oblivious to the actual Program Flow.

With x86, because it each opcode has a variable length, the code tends to auto synchronize, but the first instructions will be missed



Linear Disassembly

Issues

- With ELF files in x86, linear disassembly tends to be useful
  - Compilers do not emit inline data and the process rapidly synchronizes
  - Still, padding and alignment efforts may create some wrong instructions
- With PE files, compilers may emit in inline data and Linear Disassembly is not adequate
  - Every time data is found, disassembly becomes desynchronized
- Other architectures (ARM) and binary objects usually are not suited for Linear Disassembly
  - Obfuscation may include code as data, which is loaded dynamically
  - Fixed length instruction sets will not easily synchronize

Linear Disassembly

So why is it useful?

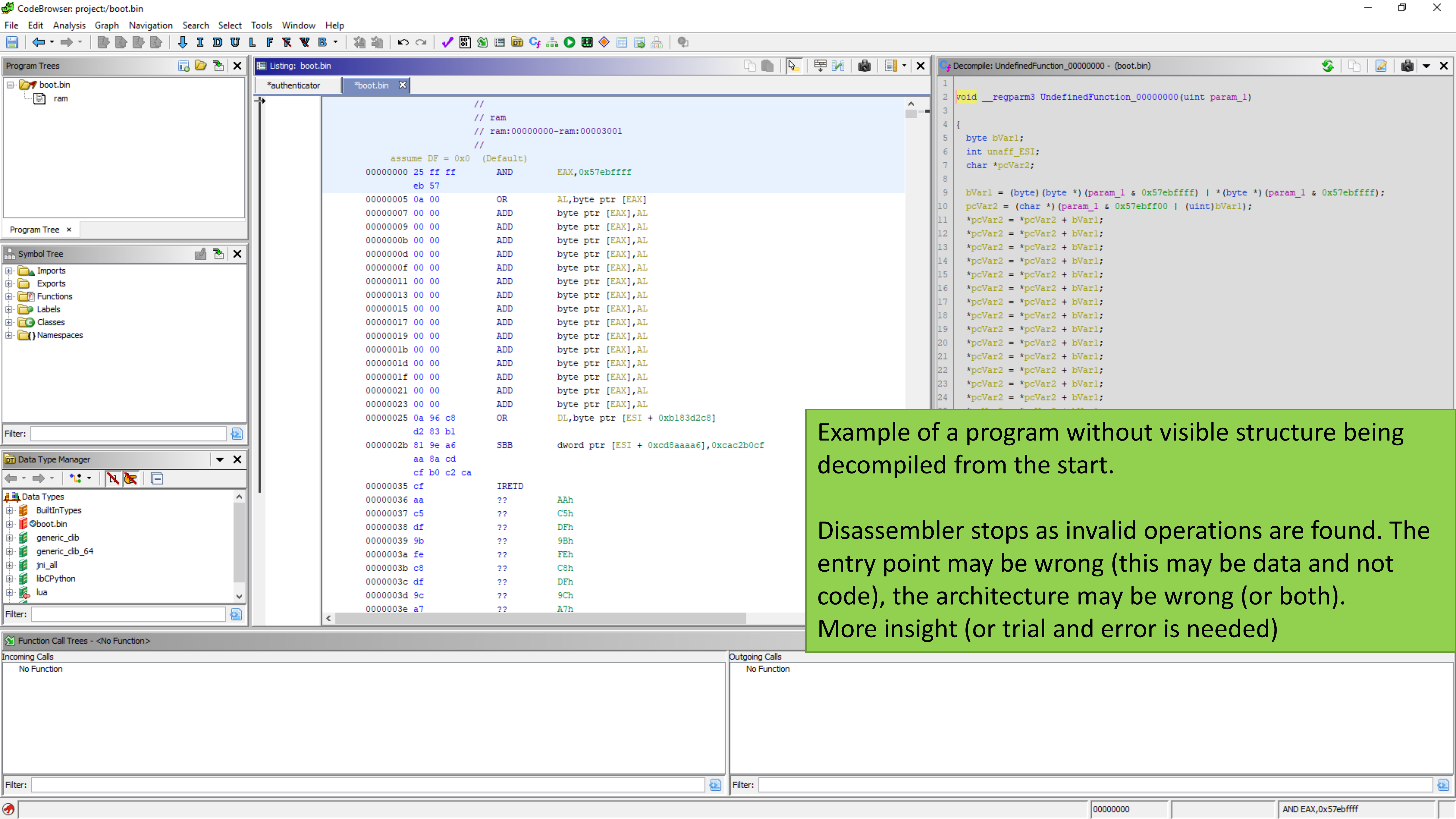
- Code in the binary blob may be executed with a dynamic call
  - Some JMP/CALL with an address computed dynamically and unknown to the static analyzer
- Linear Disassembly will decompile everything:
  - whether or not it is called - May be useful to uncover hidden program code
  - even if the binary blob is not a structured executable – Boot sector, firmware
- Readily available with simple tools: objdump and gdb
  - Gdb memory dump (x/i) will also use Linear Disassembly

Recursive Disassembly

- More complex approach that disassembles code since an initial point, **while following the control flow**.
  - That is: follows **`jmp`**, **`call`** and **`ret`**

- **As long as the start point is correct**, or it synchronizes rapidly, flow can be fully recovered
  - This is the standard process for more complex tools such as **`ghidra`** and **`IDA`**

- Goes around inline data as no instruction will exist that will make the program execute at such address
  - Well… control flow can easily be forged with **`((void (*)(int, char*)) ptr)()`**



Example of a program without visible structure being decompiled from the start.

Disassembler stops as invalid operations are found. The entry point may be wrong (this may be data and not code), the architecture may be wrong (or both). More insight (or trial and error is needed)

Function detection

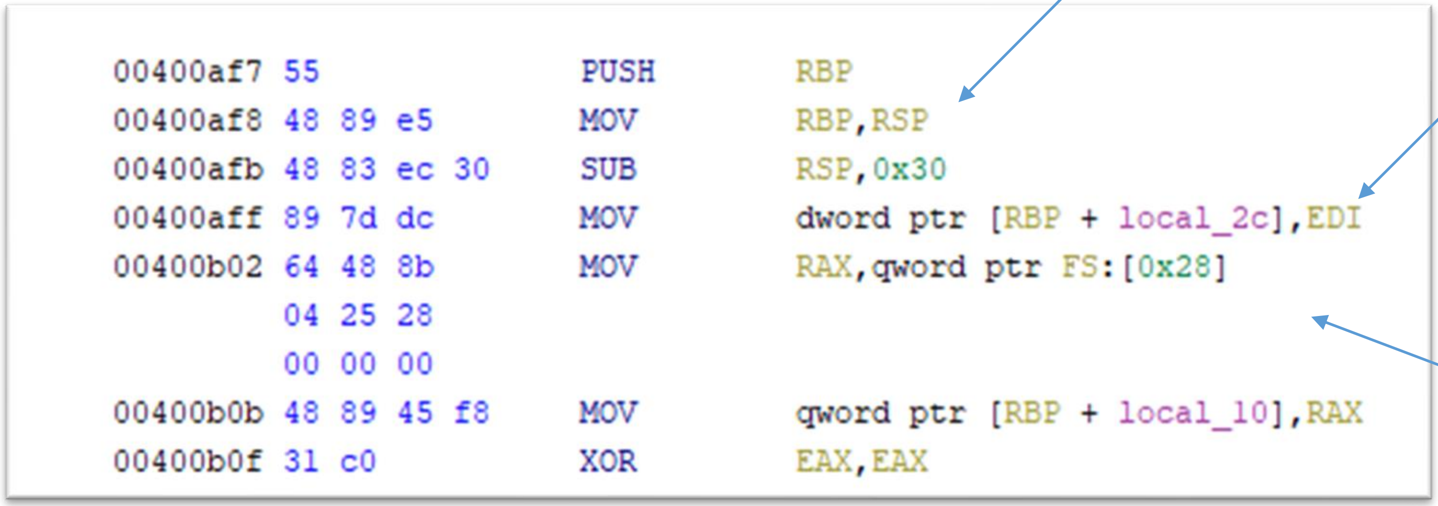
- Functions frequently include known **prolog** and **epilogues**
  - Prolog: setup the stack and optionally setup Stack Guard Canaries
  - Epilog: optionally check the canaries and release stack
- This information may be used to determine function boundaries
  - But it is architecture and compiler dependent
- Alternatives:
  - Pattern matching (automatic, done by disassemblers) can also recover functions
  - Exception handling code in the .eh_frame section
  - gcc intrinsics to cleanup stacks with exceptions __attribute__((__cleanup__(f))) and __builtin_return_address(n)

Function detection

Typical Prologue with Stack Guard

Stack allocation code
- Stores RBP
- Makes RBP = RSP
- Allocates 0x30 bytes

Stores register in stack



Canary setup
- Fetches value from FS:[0x28] to RAX
- Stores value at RBP+local_10 (top of the local stack)
- Erase RAX

Function detection

Typical Epilogue with Stack Guard

Fetches the Canary

```
00400b5a 48 8b 45 f8       MOV        RAX,qword ptr [RBP + local_10]
00400b5e 64 48 33          XOR        RAX,qword ptr FS:[0x28]
         04 25 28
         00 00 00
00400b67 74 05             JZ         LAB_00400b6e
00400b69 e8 b2 fb          CALL       __stack_chk_fail
         ff ff
                     -- Flow Override: CALL_RETURN (CALL_TERMINATOR)

                     LAB_00400b6e
00400b6e c9                LEAVE
00400b6f c3                RET
```

- XORs the Canary with reference value
- This sets the Zero flag if they are equal (No corruption)
- Jumps to end of program, or crashes the program with `__stack_chk_fail`

Deallocate stack and return to caller

Calling Conventions

- Compilers handle the function calling processes differently, and we have several conventions
  - Adapted to how programmers use the languages (number of arguments)
  - Adapted to number of registers and other architecture details

- These dictate:
  - How arguments are passed to the callee
  - How return codes are passed to the caller
  - Who allocates the stack
  - Who stores important registers such as the Program Counter

Calling Conventions

cdecl

- Originally created by Microsoft compilers, widely used in x86, including GCC
  - Standard method for most code in x86 environments
- **Arguments: passed in the stack, in inverted order (right to left)**
  - First argument is pushed last
- Registers: Mixed
  - Caller saves RIP, A, C, D
  - Callee saves BP, and others and restores RIP

Calling Conventions

cdecl

```c
int callee(int a, int b, int c) {
    char d[20];
    return a + d[0];
}

int caller(void) {
    return callee(1, 2, 3) + 4;
}
```

```asm
callee(int, int, int):
        push    ebp
        mov     ebp, esp
        sub     esp, 32
        movzx   eax, BYTE PTR [ebp-20]
        movsx   edx, al
        mov     eax, DWORD PTR [ebp+8]
        add     eax, edx
        leave
        ret
caller():
        push    ebp
        mov     ebp, esp
        push    3
        push    2
        push    1
        call    callee(int, int, int)
        add     esp, 12
        add     eax, 4
        leave
        ret
```

Allocate stack

Add two arguments.
Array at EBP-20 and argument at EBP+8

Restore EBP, EIP

Prepare arguments to be sent

Call the function. Automatically stores EIP in stack

Deallocate stack

Add 4 to EAX (EAX contains return from callee)

Calling Conventions

stdcall

- Official call convention for the Win32API (32 bits)
- **Arguments: passed in the stack from right to left**
  - Additional arguments are passed in the stack
- Registers: Callee saves
  - Except EAX, ECX and EDX which can be freely used
- Stack Red Zone: Leaf functions have a 128 byte area kept safe which doesn't need to be allocated
  - Can be used for local variables, and avoids the use of two operations (sub rsp, add rsp)
  - Leaf functions are functions that do not call others

Calling Conventions

stdcall

```c
int callee(int a, int b, int c) {
    char d[20];
    return a + d[0];
}

int caller(void) {
    return callee(1, 2, 3) + 4;
}
```

```
callee(int, int, int):
        push    ebp
        mov     ebp, esp
        sub     esp, 32
        movzx   eax, BYTE PTR [ebp-20]
        movsx   edx, al
        mov     eax, DWORD PTR [ebp+8]
        add     eax, edx
        leave
        ret     12
caller():
        push    ebp
        mov     ebp, esp
        push    3
        push    2
        push    1
        call    callee(int, int, int)
        add     eax, 4
        leave
        ret
```

Allocate stack

Add two arguments.
Array at EBP-20 and argument at EBP+8

Free Stack

Restore RIP

Prepare arguments to be sent

Call the function. Automatically stores EIP in stack

Add 4 to EAX (EAX contains return from callee)

Calling Conventions

fastcall

- Official call convention for Win32API 64bits
- **Arguments: left to right, first as registers**
  - Additional arguments are passed in the stack
- Registers: Caller saves
- Stack Shadow Zone: Leaf functions have a 32 byte area kept safe which doesn’t need to be allocated
  - Can be used for local variables, and avoids the use of two operations (sub rsp, add rsp)
  - Leaf functions are functions that do not call others

Calling Conventions

fastcall (32bits)

```
callee(int, int, int):
        push    ebp
        mov     ebp, esp
        sub     esp, 40
        mov     DWORD PTR [ebp-36], ecx
        mov     DWORD PTR [ebp-40], edx
        movzx   eax, BYTE PTR [ebp-20]
        movsx   edx, al
        mov     eax, DWORD PTR [ebp-36]
        add     eax, edx
        leave
        ret     4
caller():
        push    ebp
        mov     ebp, esp
        push    3
        mov     edx, 2
        mov     ecx, 1
        call    callee(int, int, int)
        add     eax, 4
        leave
        ret
```

Allocate stack

Stores arguments to stack

Add two arguments

Restore EIP

Prepare arguments to be sent

Call the function. Automatically stores IP in stack

Add 4 to EAX (EAX contains return from callee)

Calling Conventions

```
callee(int, int, int):
        push    ebp
        mov     ebp, esp
        sub     esp, 32
        movzx   eax, BYTE PTR [ebp-20]
        movsx   edx, al
        mov     eax, DWORD PTR [ebp+8]
        add     eax, edx
        leave
        ret
caller():
        push    ebp
        mov     ebp, esp
        push    3
        push    2
        push    1
        call    callee(int, int, int)
        add     esp, 12
        add     eax, 4
        leave
        ret
```

cdecl

```
callee(int, int, int):
        push    ebp
        mov     ebp, esp
        sub     esp, 32
        movzx   eax, BYTE PTR [ebp-20]
        movsx   edx, al
        mov     eax, DWORD PTR [ebp+8]
        add     eax, edx
        leave
        ret     12
caller():
        push    ebp
        mov     ebp, esp
        push    3
        push    2
        push    1
        call    callee(int, int, int)
        add     eax, 4
        leave
        ret
```

stdcall

```
callee(int, int, int):
        push    ebp
        mov     ebp, esp
        sub     esp, 40
        mov     DWORD PTR [ebp-36], ecx
        mov     DWORD PTR [ebp-40], edx
        movzx   eax, BYTE PTR [ebp-20]
        movsx   edx, al
        mov     eax, DWORD PTR [ebp-36]
        add     eax, edx
        leave
        ret     4
caller():
        push    ebp
        mov     ebp, esp
        push    3
        mov     edx, 2
        mov     ecx, 1
        call    callee(int, int, int)
        add     eax, 4
        leave
        ret
```

fastcall

Calling Conventions

Fastcall for 64bits (Windows)

- Official convention for x86_64 architectures with MSVC (Windows)
  - Mandatory if compiling for x86_64 in Windows

- Arguments: passed as RDX, RCX, R8, R9
  - Additional arguments are passed in the stack (right to left)

- Registers: Mixed
  - Caller save: RAX, RCX, RDX, R8, R9, R10, R11
  - Callee save: RBX, RBP, RDI, RSI, RSP, R12, R13, R14, and R15

- Stack Red Zone: Leaf functions have a 32 byte area kept safe, allocated by the callee
  - Can be used to store RDX, RCX, R8, R9
  - (Leaf functions are functions that do not call others)

Calling Conventions

fastcall (64bits)

```
int callee(int a, int b, int c) {
    char d[20];
    return a + d[0];
}

int caller(void) {
    return callee(1, 2, 3) + 4;
}
```

```
int callee(int,int,int) PROC                          ; callee
$LN3:
        mov     DWORD PTR [rsp+24], r8d
        mov     DWORD PTR [rsp+16], edx
        mov     DWORD PTR [rsp+8], ecx
        sub     rsp, 40                           ; 00000028H
        mov     eax, 1
        imul    rax, rax, 0
        movsx   eax, BYTE PTR d$[rsp+rax]
        mov     ecx, DWORD PTR a$[rsp]
        add     ecx, eax
        mov     eax, ecx
        add     rsp, 40                           ; 00000
        ret     0
int callee(int,int,int) ENDP                          ; callee

int caller(void) PROC                       ; caller
$LN3:
        sub     rsp, 40                           ; 0000
        mov     r8d, 3
        mov     edx, 2
        mov     ecx, 1
        call    int callee(int,int,int)               ; callee
        add     eax, 4
        add     rsp, 40                           ; 00000028H
        ret     0
int caller(void) ENDP                       ; caller
```

Stores arguments to shadow

Add two arguments

Free Stack

Restore RIP

Prepare arguments to be sent

Call the function. Automatically stores IP in stack

Add 4 to EAX (EAX contains return from callee)

Calling Conventions

System V AMD64 ABI

- Official convention for x64 architectures using Linux, BSD, Unix, Windows

- Arguments: passed as RDI, RSI, RDX, RCX, R8, R9
  - Additional arguments are passed in the stack

- Registers: Caller saves
  - Except RBX, RSP, RBP, R12-R15 which callee must save if they are used

- Stack Red Zone: Leaf functions have a 128 byte area kept safe which doesn’t need to be allocated
  - Can be used for local variables, and avoids the use of two operations (sub rsp, add rsp)
  - Leaf functions are functions that do not call others

Calling Conventions

System V AMD64 ABI

```c
int callee(int a, int b, int c) {
    char d[20];
    return a + d[0];
}

int caller(void) {
    return callee(1, 2, 3) + 4;
}
```

```asm
callee(int, int, int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-36], edi
        mov     DWORD PTR [rbp-40], esi
        mov     DWORD PTR [rbp-44], edx
        movzx   eax, BYTE PTR [rbp-32]
        movsx   edx, al
        mov     eax, DWORD PTR [rbp-36]
        add     eax, edx
        pop     rbp
        ret
caller():
        push    rbp
        mov     rbp, rsp
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    callee(int, int, int)
        add     eax, 4
        pop     rbp
        ret
```

Leaf function uses stack directly

Adds value with argument

Restore RIP

Prepare arguments to be sent

Call the function. Automatically stores RIP in stack

Add values. EAX will contain the result

Calling Conventions

64bits

```
callee(int, int, int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-36], edi
        mov     DWORD PTR [rbp-40], esi
        mov     DWORD PTR [rbp-44], edx
        movzx   eax, BYTE PTR [rbp-32]
        movsx   edx, al
        mov     eax, DWORD PTR [rbp-36]
        add     eax, edx
        pop     rbp
        ret
caller():
        push    rbp
        mov     rbp, rsp
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    callee(int, int, int)
        add     eax, 4
        pop     rbp
        ret
```

System V AMD64 ABI

```
int callee(int,int,int) PROC
$LN3:
        mov     DWORD PTR [rsp+24], r8d
        mov     DWORD PTR [rsp+16], edx
        mov     DWORD PTR [rsp+8], ecx
        sub     rsp, 40
        mov     eax, 1
        imul    rax, rax, 0
        movsx   eax, BYTE PTR d$[rsp+rax]
        mov     ecx, DWORD PTR a$[rsp]
        add     ecx, eax
        mov     eax, ecx
        add     rsp, 40
        ret     0
int callee(int,int,int) ENDP

int caller(void) PROC
$LN3:
        sub     rsp, 40
        mov     r8d, 3
        mov     edx, 2
        mov     ecx, 1
        call    int callee(int,int,int)
        add     eax, 4
        add     rsp, 40
        ret     0
int caller(void) ENDP
```

fastcall

Common Logic Structures

- When analyzing code, it’s important to recognize basic flow control structures
  - Remember that the decompiler may be unreliable

- Basic structures:
  - If else
  - Switch case
  - For

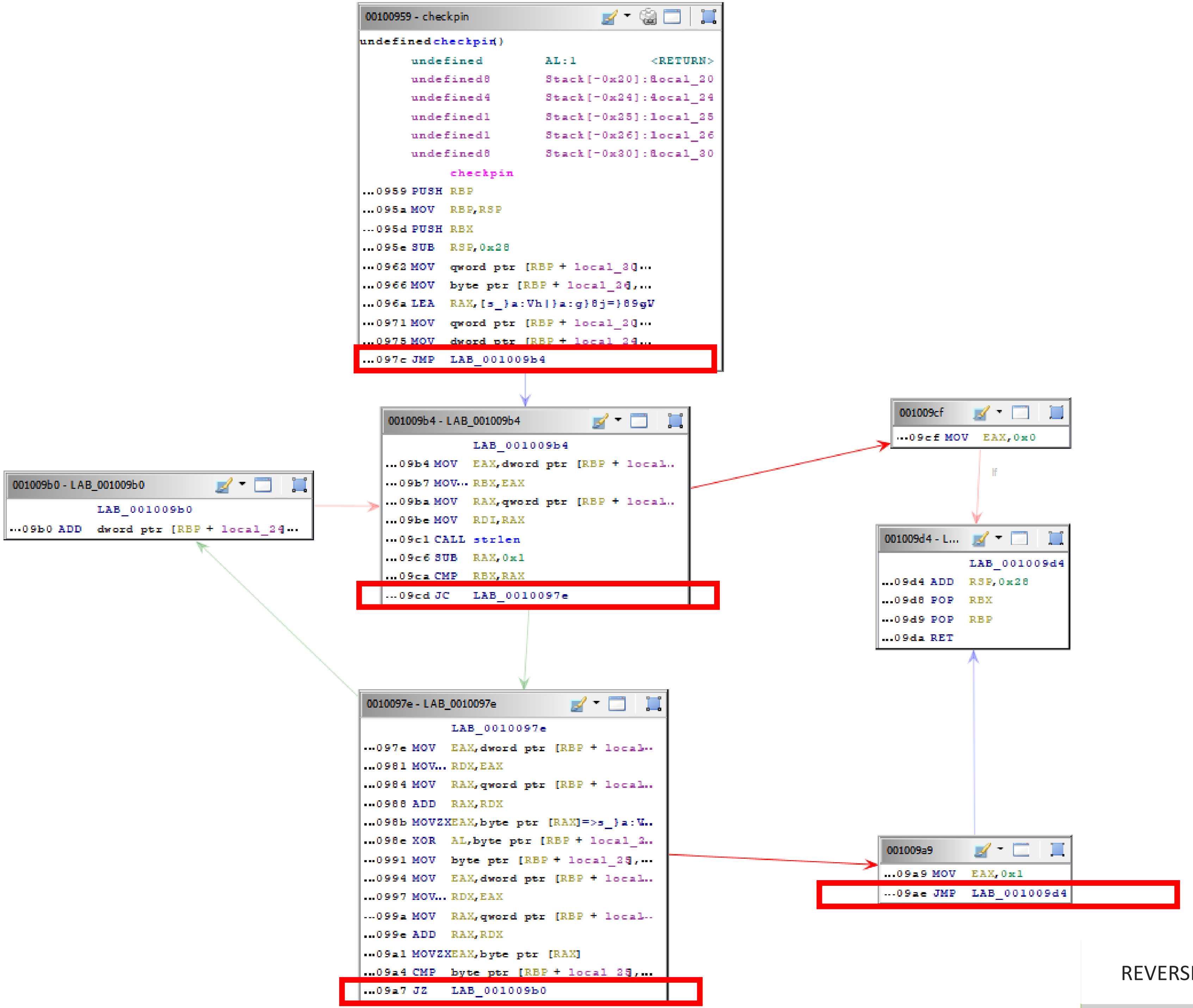
Common Logic Structures

Conditional Branches (if else)

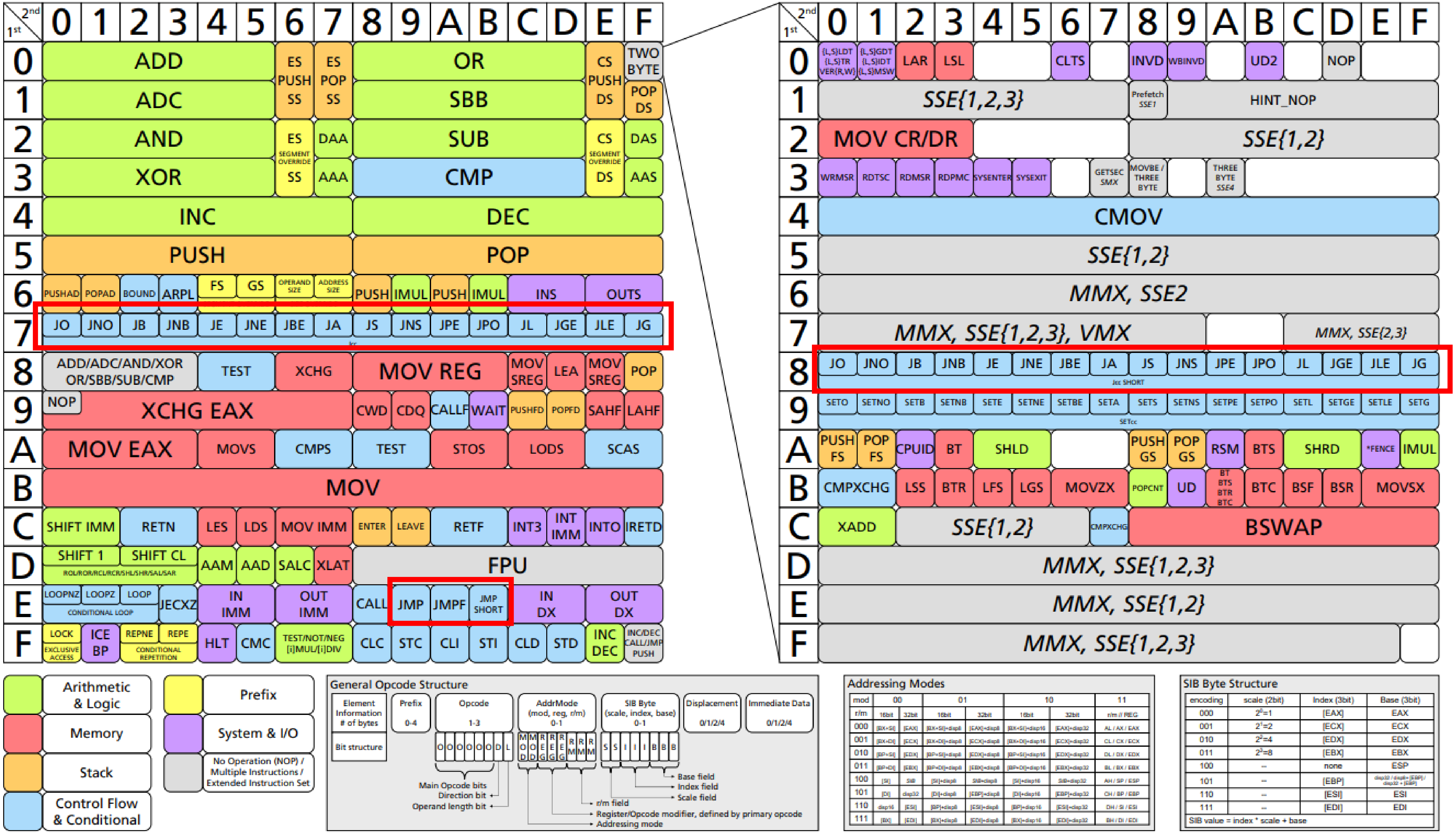
- Basic control-flow instructions: move execution to a defined address if a condition is true
  - Usually, one condition tested at a time. Complex If/else must be broken

- Assembly code is structed **as a graph with tests and execution** statements (the conditions body)

- x86 and most architectures have inherent support for many types of comparisons.
  - In x86 this is the jXX family of instructions.


00100959 - checkpin
undefined checkpin()
undefined AL:1 <RETURN>
undefined8 Stack[-0x20]:8 local_20
undefined4 Stack[-0x24]:4 local_24
undefined1 Stack[-0x25]:1 local_25
undefined1 Stack[-0x26]:1 local_26
undefined8 Stack[-0x30]:8 local_30
checkpin
...0959 PUSH RBP
...095a MOV RBP,RSP
...095d PUSH RBX
...095e SUB RSP,0x28
...0962 MOV qword ptr [RBP + local_30]...
...0966 MOV byte ptr [RBP + local_26],...
...0971 MOV qword ptr [RBP + local_20]...
...097c JMP LAB_001009b4
001009b4 - LAB_001009b4
LAB_001009b4
...09b4 MOV EAX,dword ptr [RBP + local...
...09b7 MOV... RBX,EAX
...09ba MOV RAX,qword ptr [RBP + local...
...09be MOV RDI,RAX
...09c1 CALL strlen
...09c6 SUB RAX,0x1
...09ca CMP RBX,RAX
...09cd JC LAB_0010097e
001009b0 - LAB_001009b0
LAB_001009b0
...09b0 ADD dword ptr [RBP + local_24]...
001009cf
...09cf MOV EAX,0x0
001009d4 - L...
LAB_001009d4
...09d4 ADD RSP,0x28
...09d8 POP RBX
...09d9 POP RBP
...09da RET
0010097e - LAB_0010097e
LAB_0010097e
...097e MOV EAX,dword ptr [RBP + local...
...0981 MOV... RDX,EAX
...0984 MOV RAX,qword ptr [RBP + local...
...0988 ADD RAX,RDX
...098e XOR AL,byte ptr [RBP + local_2...
...0991 MOV byte ptr [RBP + local_25],...
...0994 MOV EAX,dword ptr [RBP + local...
...0997 MOV... RDX,EAX
...099a MOV RAX,qword ptr [RBP + local...
...099e ADD RAX,RDX
...09a1 MOVZX EAX,byte ptr [RAX]
...09a4 CMP byte ptr [RBP + local_25],...
...09a7 JZ LAB_001009b0
001009a9
...09a9 MOV EAX,0x1
...09ae JMP LAB_001009d4

x86 Opcode Structure and Instruction Overview



v1.0 – 30.08.2011
Contact: Daniel Plohmann – +49 228 73 54 228 – daniel.plohmann@fkie.fraunhofer.de

Source: Intel x86 Instruction Set Reference
Opcode table presentation inspired by work of Ange Albertini

Common Logic Structures

Conditional Branches (if else)

- Signed comparison: **`l < , le <=, g >, ge >=`**
- Unsigned comparison: **`b <, be <=, a >=, ae >=`**
  - Below and Above
- Equality **e**
- Every condition can be negated with **n**

Common Logic Structures

Conditional Branches (if else)

- `z, s, c, o,` and **p** for **`ZF, SF, CF OF,`** and **PF**
  - **ZF: Zero Flag**, 1 if last operation was 0
  - **CF: Carry Flag**. Last operation required an additional bit (e.g. 255 + 1, which has 9 bits)
  - **OF: Overflow Flag**. Last operation had an arithmetic overflow (127 + 127 in a signed variable results in overflow)
  - **PF: Parity Flag**. 1 if last operation resulted in a value with even number of 1
  - **SF: Sign Flag**. 1 if last operation resulted in a signed value (MSB bit = 1)

- **s** means negative, **ns** non-negative
  - Signal or not signal

- **p** and **np** are also pe "parity even" and **po** "parity odd"

Common Logic Structures

Conditional Branches (if else)

- `and`, `or`, and `xor` clear **OF** and **CF**, and set **ZF**, **SF**, and **PF** based on the result

- `test` is like **and** but only sets the flags discarding the result

- Checking **nz** after `test` is like `if (x & mask) in C`

- `test` a register against itself is the fastest way to check for zero or negative

Common Logic Structures

Conditional Branches (if else)

- **Direct jump**: target(s) specified in code (harcoded)

- **Indirect jump**: target selected from runtime data like register or memory contents

- **Conditional jump**: target differs based on a condition

Common Logic Structures

Conditional Branches (If else)

- Structure can be recognized by one or more conditional branches, without loops

- je: jump equal

- js: jump is sign

- …etc…

```c
int bar(int b) {
    return b * b;
}

int foo(int a) {
    if(a == 0){
        return bar(a) * 1;


    }
    else
        if(a < 0){
            return bar(a) - 1;
        }
        else{
            return bar(a) + 1;
        }
}
```

```asm
bar:
        imul    edi, edi
        mov     eax, edi
        ret
foo:
        test    edi, edi
        je      .L6
        js      .L7
        call    bar
        add     eax, 1
        ret
.L6:
        call    bar
        ret
.L7:
        call    bar
        sub     eax, 1
        ret
```

Common Logic Structures

Switch case

- Structure can be recognized by several comparisons and jumps or **jump table**

- Observe the difference between what a programmer writes and what is produced
  - Switch is written as an atomic instruction, but it isn't
  - Also, it is dangerous because of missing breaks;

- Test: compare two registers. Set 3 flags:
  - PF: Even number of bits
  - ZF: Zero
  - SF: Signed value

```c
int bar(int b) {
    return b * b;
}

int foo(int a) {
   switch(a){
       case 0:
         a = bar(1) + 1;
         break;
       case 1:
         a = bar(2+ a) + 2;
         break;
      case 3:
         a = bar(3) + 3;
      default:
         a= bar(4) + 4;
   }

   return a;
}
```

```asm
bar:
        imul    edi, edi
        mov     eax, edi
        ret
foo:
        test    edi, edi
        je      .L3
        cmp     edi, 1
        je      .L4
        mov     edi, 4
        call    bar
        add     eax, 4
        ret
.L3:
        mov     edi, 1
        call    bar
        add     eax, 1
        ret
.L4:
        add     edi, 2
        call    bar
        add     eax, 2
        ret
```

Common Logic Structures

loops

- For, while and do while are generally the same
- Identified by:
  - an index
  - an increment
  - a comparison
  - two jumps

```c
int bar(int b) {
    return b * b;
}

int foo(int a) {
    int b = 0;
    for(int i = 0; i< a; i++){
        b += bar(i);
    }

   return b;
}

int caller(void) {
    return callee(1, 2, 3) + 4;
}
```

```asm
bar:
        imul    edi, edi
        mov     eax, edi
        ret
foo:
        push    r12
        push    rbp
        push    rbx
        mov     r12d, edi
        mov     ebx, 0
        mov     ebp, 0
.L3:
        cmp     ebx, r12d
        jge     .L6
        mov     edi, ebx
        call    bar
        add     ebp, eax
        add     ebx, 1
        jmp     .L3
.L6:
        mov     eax, ebp
        pop     rbx
        pop     rbp
        pop     r12
        ret
```

Prepares stack

- r12d will contain the number of iterations
- ebx will be the counter

- Loop body

Jump to top of loop

C++ code

- C++ is very popular, and adds an additional layer of complexity
  - A program doesn’t have functions, has methods
  - Methods have a shared context (the object)
  - Methods can be overridden due to inheritance
  - The `this` pointer commonly allows access to data outside the function stack
  - Contructors, new…?
  - Strings are complex objects

C++ code

- this pointer
  - The “this” pointer plays a crucial role in the identification of C++ sections in the assembly code. It is initialized to point to the object used, to invoke the function, when it is available in non-static C++ functions.

- Vtables
  - Eases runtime resolution of calls to virtual functions.
  - The compiler generates a vtable containing pointers to each virtual function for the classes which contain virtual functions.

- Constructors and destructors
  - A member function which initializes objects of a class and it can be identified in assembly by studying the objects in which it’s created.

C++ code

- Runtime Type Information (RTTI)
  - Mechanism to identify the object type at run.
  - These keywords pass information, such as class name and hierarchy, to the class.

- Structured exception handling (SEH)
  - Irregularities in source code that unexpectedly strike during runtime, terminating the program.
  - SEH is the mechanism that controls the flow of execution and handles errors by isolating the code section where the unexpected condition originates. Inheritance

- Inheritance
  - allows new objects to take on existing object properties.
  - Observing RTTI relationships can reveal inheritance hierarchy

hello1.cpp

A simple hello world

```cpp
#include <iostream>
#include <string>

class A {
    std::string text1;

    public:
    A(std::string text1) {
        this->text1 = text1;
    }
    void print() {
        std::cout << this->text1 << std::endl;
    }
};

int main(int argc, char** argv) {
    A a(std::string("Hello World"));
    a.print();

}
```

hello1.cpp

```
$ readelf --dyn-sym hello1

Symbol table '.dynsym' contains 21 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _ZNSt7__cxx1112basic_stri@GLIBCXX_3.4.21 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _ZSt4endlIcSt11char_trait@GLIBCXX_3.4 (4)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _ZNSt7__cxx1112basic_stri@GLIBCXX_3.4.21 (2)
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __cxa_atexit@GLIBC_2.2.5 (3)
     5: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _ZStlsIcSt11char_traitsIc@GLIBCXX_3.4.21 (2)
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _ZNSolsEPFRSoS_E@GLIBCXX_3.4 (4)
     7: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _ZNSaIcED1Ev@GLIBCXX_3.4 (4)
     8: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _ZNSt7__cxx1112basic_stri@GLIBCXX_3.4.21 (2)
     9: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _ZNSt7__cxx1112basic_stri@GLIBCXX_3.4.21 (2)
    10: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _ZNSt8ios_base4InitC1Ev@GLIBCXX_3.4 (4)
    11: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __gxx_personality_v0@CXXABI_1.3 (5)
    12: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
    13: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _Unwind_Resume@GCC_3.0 (6)
    14: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _ZNSaIcEC1Ev@GLIBCXX_3.4 (4)
    15: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (3)
    16: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    17: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
    18: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _ZNSt8ios_base4InitD1Ev@GLIBCXX_3.4 (4)
    19: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.2.5 (3)
    20: 00000000000040a0   272 OBJECT  GLOBAL DEFAULT   26 _ZSt4cout@GLIBCXX_3.4 (4)
```

C++ code

```
/* WARNING: Unknown calling convention yet parameter storage is locked */

int main(void)

{
  A local_68 [32];
  basic_string<char,std::char_traits<char>,std::allocator<char>> local_48 [47];
  allocator<char> local_19 [9];

  std::allocator<char>::allocator();
                    /* try { // try from 00101203 to 00101207 has its CatchHandler @ 00101263 */
  std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::basic_string
            ((char *)local_48,(allocator *)"Hello World");
                    /* try { // try from 00101216 to 0010121a has its CatchHandler @ 00101252 */
  A::A(local_68,(basic_string)0xb8);
  std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string
            (local_48);
  std::allocator<char>::~allocator(local_19);
                    /* try { // try from 0010123a to 0010123e has its CatchHandler @ 0010127d */
  A::print(local_68);
  A::~A(local_68);
  return 0;
}
```

No C++ class declarations, but C++ class use.

- Constructors
- Methods
- Destructors

C++ code

```
                                  _
00101216 e8 e1 00        CALL       A::A
         00 00
                              } // end try from 00101216 to 0010121a
0010121b 48 8d 45 c0     LEA        RAX=>local_48,[RBP + -0x40]
0010121f 48 89 c7        MOV        RDI,RAX
00101222 e8 19 fe        CALL       ~basic_string
         ff ff
00101227 48 8d 45 ef     LEA        RAX=>local_19,[RBP + -0x11]
0010122b 48 89 c7        MOV        RDI,RAX
0010122e e8 4d fe        CALL       ~allocator
         ff ff
00101233 48 8d 45 a0     LEA        RAX=>local_68,[RBP + -0x60]
00101237 48 89 c7        MOV        RDI,RAX

                              try { // try from 0010123a to 0010123e has its CatchHandler @...
                              LAB_0010123a                                  XREF[1]:     00]
0010123a e8 11 01        CALL       A::print
         00 00
                              } // end try from 0010123a to 0010123e
0010123f 48 8d 45 a0     LEA        RAX=>local_68,[RBP + -0x60]
00101243 48 89 c7        MOV        RDI,RAX
00101246 e8 3d 01        CALL       A::~A
         00 00
0010124b b8 00 00        MOV        EAX,0x0
         00 00
00101250 eb 45           JMP        LAB_00101297
```

Additional Hints related to exception handling

Standard ASM code with function invocation, using arguments in registers and values stored in the stack

C++ code

.eh_frame ELF section contains information about the multiple methods.

Required for unwinding frames, when iterating over the function frames. Contains language specific information, organized in Call Frame Information records

```
                    ****************************************************************
                    * Frame Descriptor Entry                                       *
                    ****************************************************************
                    fde_00102148                                  XREF[1]:     0010205c(*)
00102148 1c 00 00 00     ddw         1Ch                                         (FDE) Length
0010214c a4 00 00 00     ddw         cie_001020a8                                (FDE) CIE Reference Pointer
00102150 00 f2 ff ff     ddw         A::print                                    (FDE) PcBegin
00102154 37 00 00 00     ddw         37h                                         (FDE) PcRange
00102158 00              uleb128     0h                                          (FDE) Augmentation Data Length
00102159 41 0e 10        db[15]                                                  (FDE) Call Frame Instructions
         86 02 43
         0d 06 72 ...
```

C++ code

this is passed as an additional, hidden argument
In this case, in RDI as the method has no arguments

```
                   **************************************************************
                   * A::print()                                                 *
                   **************************************************************
                   undefined __thiscall print(A * this)
     undefined         AL:1           <RETURN>
     A *               RDI:8 (auto)   this
     undefined8        Stack[-0x10]:8 local_10                        XREF[2]:     00101358(W),
                                                                                   0010135c(R)
                   _ZN1A5printEv                                   XREF[4]:     Entry Point(*), main:0010123a(c),
                   A::print                                                     00102058, 00102150(*)
00101350 55              PUSH       RBP
00101351 48 89 e5        MOV        RBP,RSP
00101354 48 83 ec 10     SUB        RSP,0x10
00101358 48 89 7d f8     MOV        qword ptr [RBP + local_10],this
0010135c 48 8b 45 f8     MOV        RAX,qword ptr [RBP + local_10]
00101360 48 89 c6        MOV        RSI,RAX
00101363 48 8d 3d        LEA        this,[std::cout]                                  =
         36 2d 00 00
0010136a e8 f1 fc        CALL       operator<<                                        basic_ostream * operator<<(basic...
         ff ff
0010136f 48 89 c2        MOV        RDX,RAX
00101372 48 8b 05        MOV        RAX,qword ptr [->endl<char,std::char_traits<ch... = 00105008
         57 2c 00 00
00101379 48 89 c6        MOV        RSI=>endl<char,std::char_traits<char>>,RAX        = ??
0010137c 48 89 d7        MOV        this,RDX
0010137f e8 ec fc        CALL       operator<<                                        undefined operator<<(basic_ostre...
         ff ff
```