

Binary Analysis - 2

REVERSE ENGINEERING

João Paulo Barraca

deti universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

Binary Analysis Process

- Up to now we know how ELF and PE files are structured
- but the question remains: how do we analyse ELF/PE 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)

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?
 - What are its control structures

Disassembler basics with ghidra

- ghidra is a open source tool, doing Disassembly and Static Analysis
 - Has support for Dynamic Analysis (debug)
 - Works on Windows, Linux and macos
 - Java based
- Not the most important tool, but is gaining a huge traction
 - It's free and very powerful with a huge number of platforms and a fine decompiler
- Roles:
 - Disassembler
 - Decompiler
 - ... complete SRE framework



Program Trees

- authenticator
 - .bss
 - .data
 - .got
 - .dynamic
 - .fini_array
 - .init_array
 - .eh_frame
 - .eh_frame_hdr
 - .rodata

Symbol Tree

- Imports
- Exports
- Functions
 - __cxa_finalize
 - __do_global_dtors_aux
 - _gmon_start__
 - _libc_csu_fini
 - _libc_csu_init
 - _libc_start_main
 - _stack_chk_fail
 - _stack_chk_fail
 - fini

Data Type Manager

Data Types

- BuiltinTypes
- authenticator
- generic_cib
- generic_cib_64
- jni_all
- libCPython
- lua

Listing: authenticator

```
// segment_2.1
// Loadable segment [0x0 - 0xelf] (disabled execute bit)
// ram:00100000-ram:00100237
//
01 00 00 ...
00100000 7f          db          7Fh          e_ident_magi...
00100001 45 4c 46     ds          "ELF"         e_ident_magi...
00100004 02          db          2h          e_ident_class
00100005 01          db          1h          e_ident_data
00100006 01          db          1h          e_ident_vers...
00100007 00          db          0h          e_ident_osabi
00100008 00          db          0h          e_ident_abiv...
00100009 00 00 00 00 00  db[7]       e_ident_pad
00100010 03 00          dw          3h          e_type
00100012 3e 00          dw          3Eh         e_machine
00100014 01 00 00 00     ddw         1h          e_version
00100018 d0 07 00 00 00  dq          _start        e_entry
00100020 40 00 00 00 00  dq          Elf64_Phdr_ARRAY_00100... e_phoff =
00100028 00 2b 00 00 00  dq          Elf64_Shdr_ARRAY_elfS... e_shoff
00100030 00 00 00 00     ddw         0h          e_flags
00100034 40 00          dw          40h         e_ehsize
00100036 38 00          dw          38h         e_phentsize
00100038 09 00          dw          9h          e_phnum
0010003a 40 00          dw          40h         e_shentsize
0010003c 1d 00          dw          1Dh         e_shnum
0010003e 1c 00          dw          1Ch         e_shstrndx

Elf64_Phdr_ARRAY_00100040 XREF[2]: 00100020(*), 0010005
00100040 06 00 00     Elf64_Ph... PT_PHDR - Program
00 04 00
00 00 40 ...
```

Decompile: _start - (authenticator)

```
1 void _start(undefined8 param_1,undefined8 param_2,undefined8 param_3)
2
3
4 {
5     undefined8 in_stack_00000000;
6     undefined auStack8 [8];
7
8     __libc_start_main(main,in_stack_00000000,&stack0x00000008,__libc_csu_init,__libc_csu_fini,pe
9         auStack8);
10 do {
11     /* WARNING: Do nothing block with infinite loop */
12 } while( true );
13 }
14
```

Top menu and tools for quick access.

Function Call Trees - <No Function>

Incoming Calls

No Function

Outgoing Calls

No Function

Program Trees

- authenticator
 - .bss
 - .data
 - .got
 - .dynamic
 - .fini_array
 - .init_array
 - .eh_frame
 - .eh_frame_hdr
 - .rodata

Symbol Tree

- Imports
- Exports
- Functions
 - __cxa_finalize
 - __cxa_finalize
 - __do_global_dtors_aux
 - _gmon_start__
 - __libc_csu_fini
 - __libc_csu_init
 - __libc_start_main
 - __stack_chk_fail
 - __stack_chk_fail
 - fini

Data Type Manager

Data Types

- BuiltinTypes
- authenticator
- generic_clib
- generic_clib_64
- jni_all
- libCPython
- lua

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.

00100028	00 2b 00 00 00	dq	Elf64_Shdr_ARRAY_elfS... e_shoff		
	00 00 00				
00100030	00 00 00 00	ddw	0h	e_flags	
00100034	40 00	dw	40h	e_ehsize	
00100036	38 00	dw	38h	e_phentsize	
00100038	09 00	dw	9h	e_phnum	
0010003a	40 00	dw	40h	e_shentsize	
0010003c	1d 00	dw	1Dh	e_shnum	
0010003e	1c 00	dw	1Ch	e_shstrndx	
			Elf64_Phdr_ARRAY_00100040	XREF[2]:	00100020(*), 0010005
00100040	06 00 00		Elf64_Ph...		PT_PHDR - Program
	00 04 00				
	00 00 40 ...				

```
Decompile: _start - (authenticator)
1
2 void _start(undefined8 param_1,undefined8 param_2,undefined8 param_3)
3
4 {
5     undefined8 in_stack_00000000;
6     undefined auStack8 [8];
7
8     __libc_start_main(main,in_stack_00000000,&stack0x00000008,__libc_csu_init,__libc_csu_fini,pe
9         auStack8);
10
11     do {
12         /* WARNING: Do nothing block with infinite loop */
13     } while( true );
14 }
```

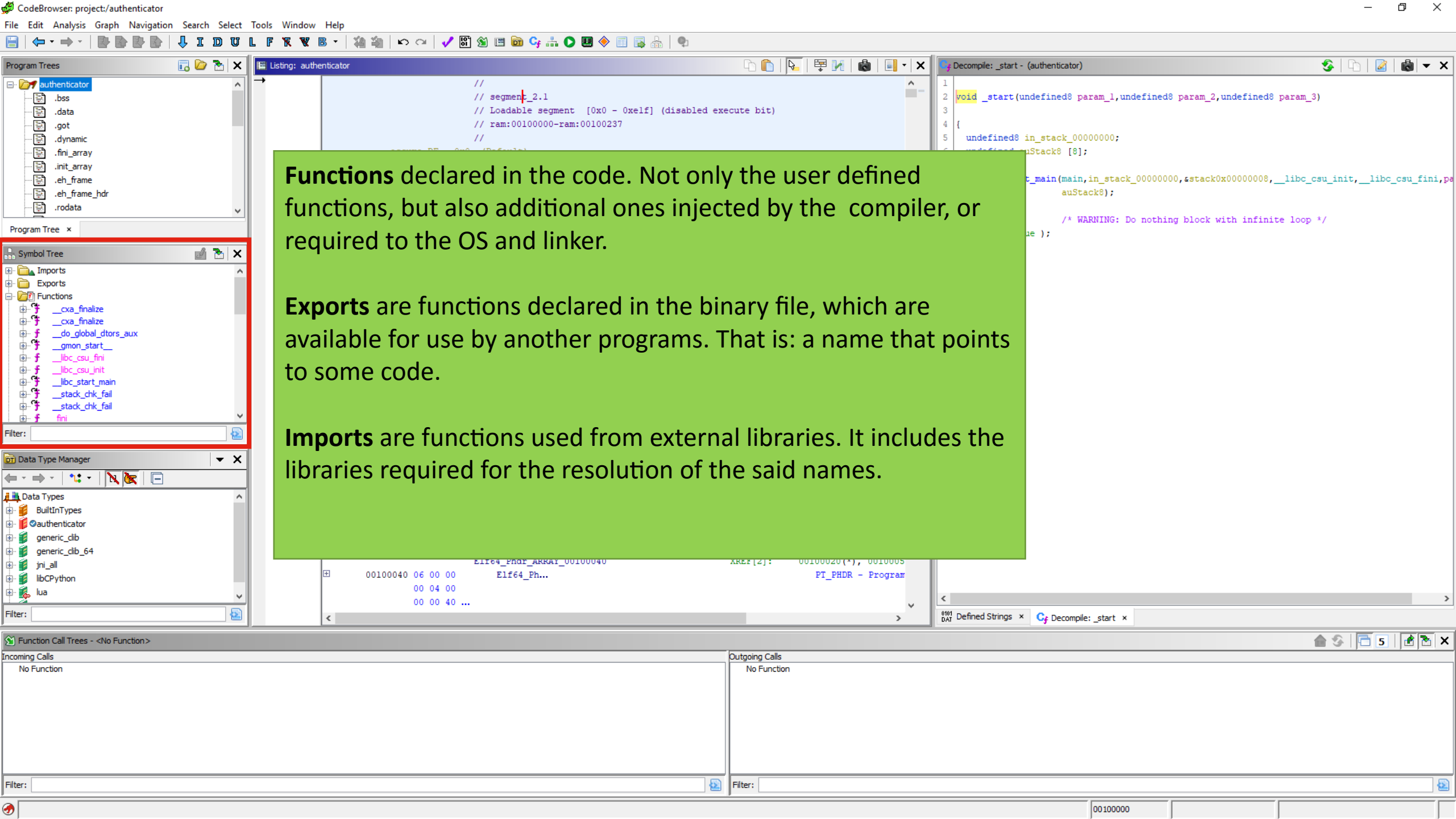
Function Call Trees - <No Function>

Incoming Calls

No Function

Outgoing Calls

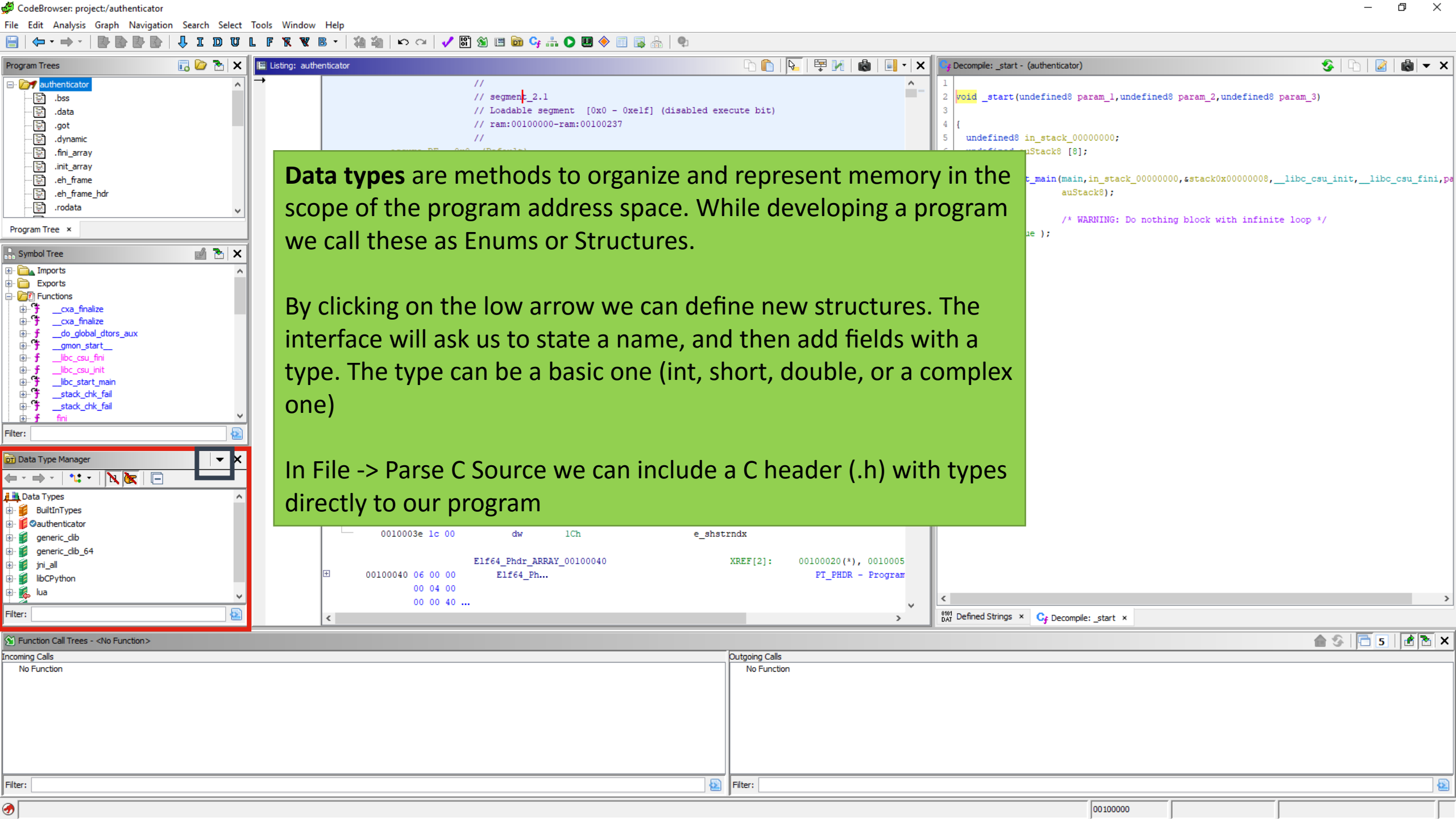
No Function



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.

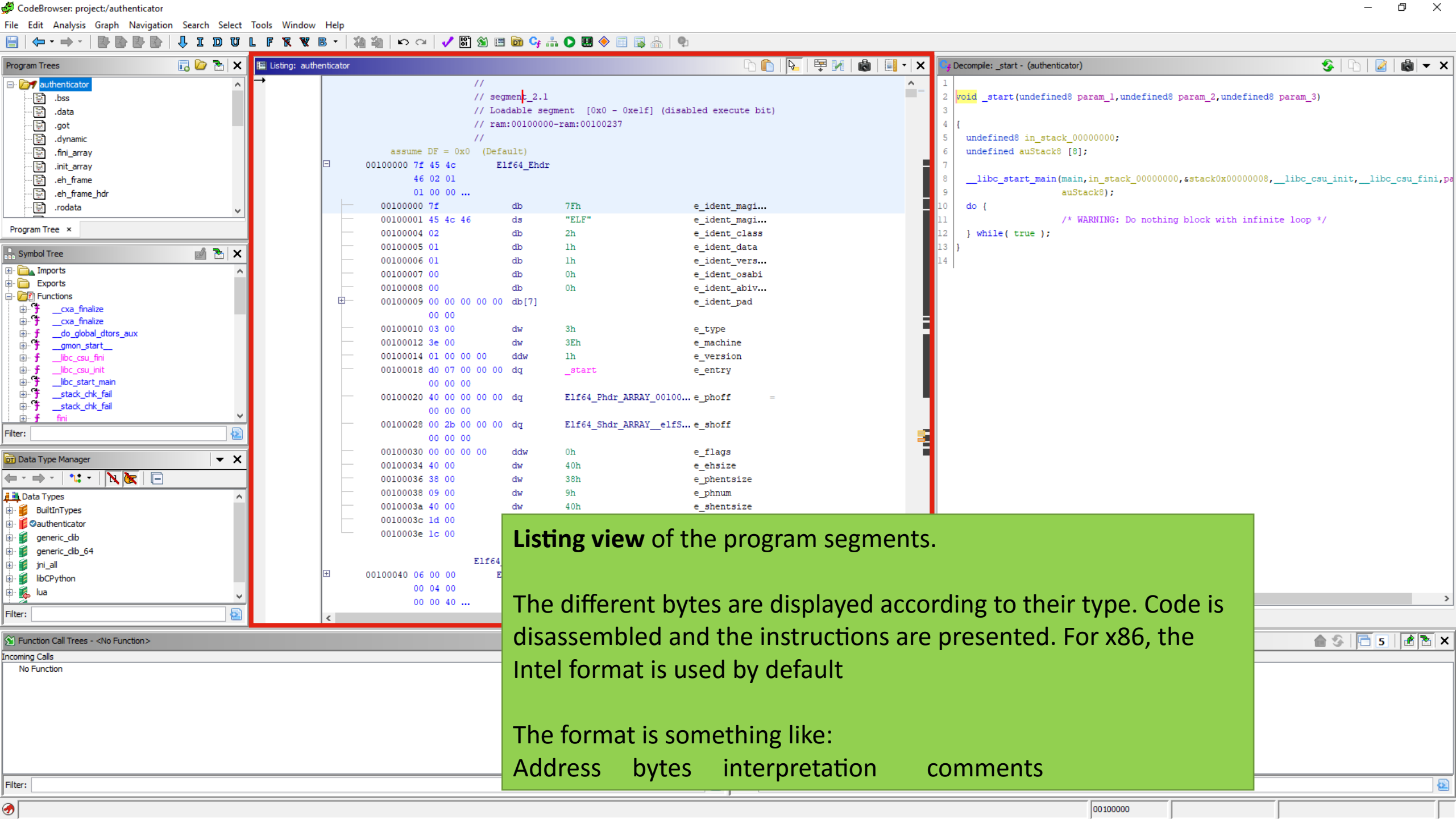
Imports are functions used from external libraries. It includes the libraries required for the resolution of the said names.



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)

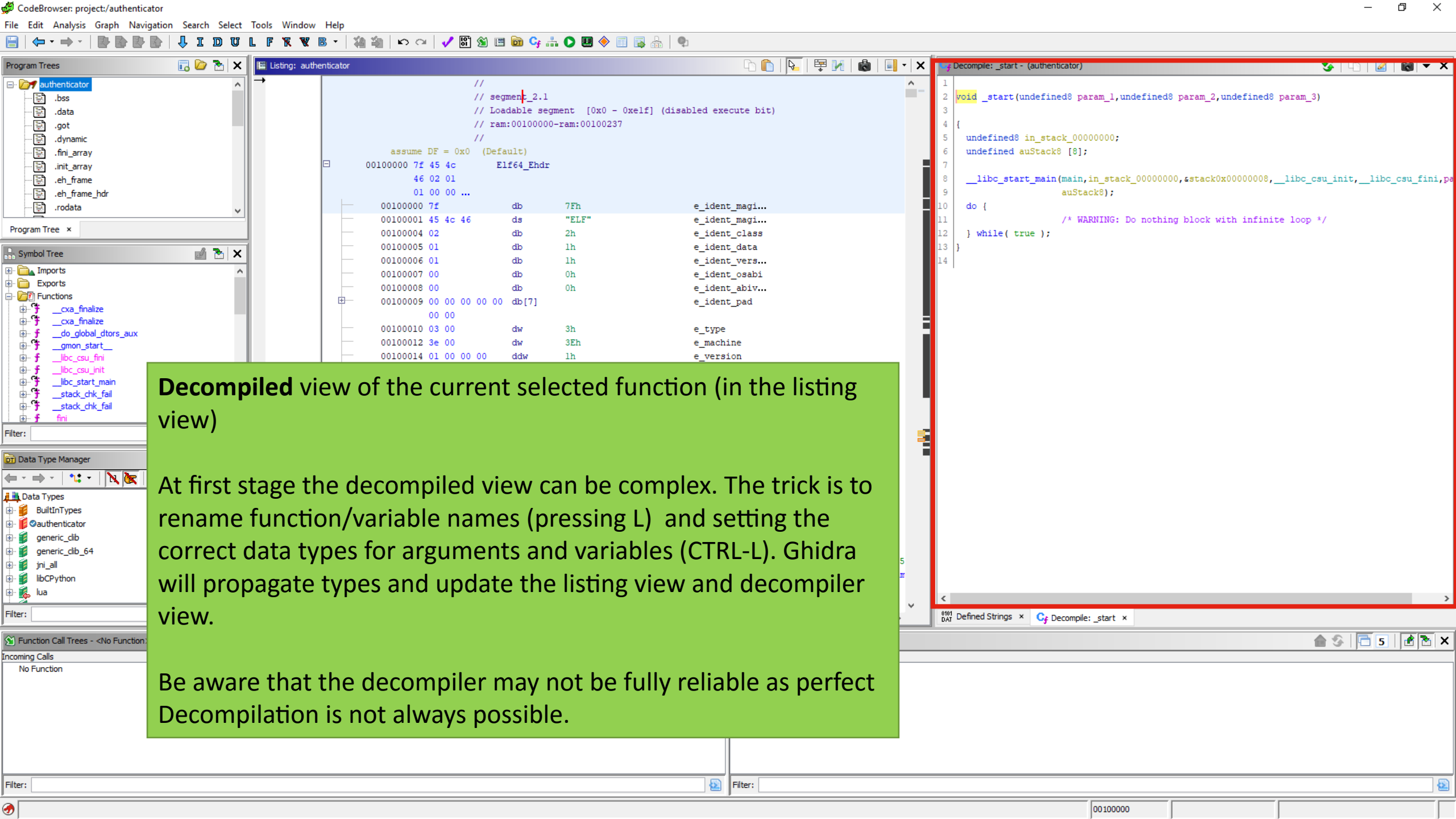
In File -> Parse C Source we can include a C header (.h) with types directly to our program



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.

Program Trees

- authenticator
 - .bss
 - .data
 - .got
 - .dynamic
 - .fini_array
 - .init_array
 - .eh_frame
 - .eh_frame_hdr
 - .rodata

Symbol Tree

- _init
 - _ITM_deregisterTMCloneTable
 - _ITM_registerTMCloneTable
 - _start
 - local_10
 - checkpin
 - deregister_tm_clones
 - fgets
 - fgets
 - frame_dummy
 - FUN_00100730
 - local_10

Data Type Manager

Data Types

- BuiltInTypes
- authenticator
- generic_clib
- generic_clib_64
- jni_all
- libCPython
- lua

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

main

001009db	55	PUSH	RBP
001009dc	48 89 e5	MOV	RBP, RSP
001009df	48 83 ec 50	SUB	RSP, 0x50
001009e3	64 48 8b	MOV	RAX, qword ptr FS:[0x28]

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

```

size_t strlen(char *)
XREF[1]: 001009ae(j)
*****
*****
XREF[2]: 001009ec(W)
XREF[2]: 00100ae7(R)
XREF[2]: 00100a98(*)
XREF[2]: 00100aa9(*)
XREF[2]: 00100a40(*)
XREF[2]: 00100a51(*)
XREF[4]: Entry Point(*),
_start:001007ed(*),
00100da4(*)
  
```

```

Decompile: main - (authenticator)
1  undefined8 main(void)
2
3
4  {
5      int iVar1;
6      long in_FS_OFFSET;
7      char local_58 [32];
8      char local_38 [40];
9      long local_10;
10
11     local_10 = *(long *) (in_FS_OFFSET + 0x28);
12     setbuf(stdout, (char *) 0x0);
13     printstr(&DAT_00100bc3, 0);
14     printstr("Please enter your credentials to continue.\n\n", 0);
15     printstr("Alien ID: ", 0);
16     fgets(local_58, 0x20, stdin);
17     iVar1 = strcmp(local_58, "11337\n");
18     if (iVar1 == 0) {
19         printstr("Pin: ", 0);
20         fgets(local_38, 0x20, stdin);
21         iVar1 = checkpin(local_38);
22     }
  
```

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

Function Call Trees: main - (authenticator)

Incoming Calls

- Incoming References - main
 - _start

Outgoing Calls

- Outgoing References - main
 - setbuf
 - fgets
 - strcmp
 - printstr
 - checkpin
 - _stack_chk_fail

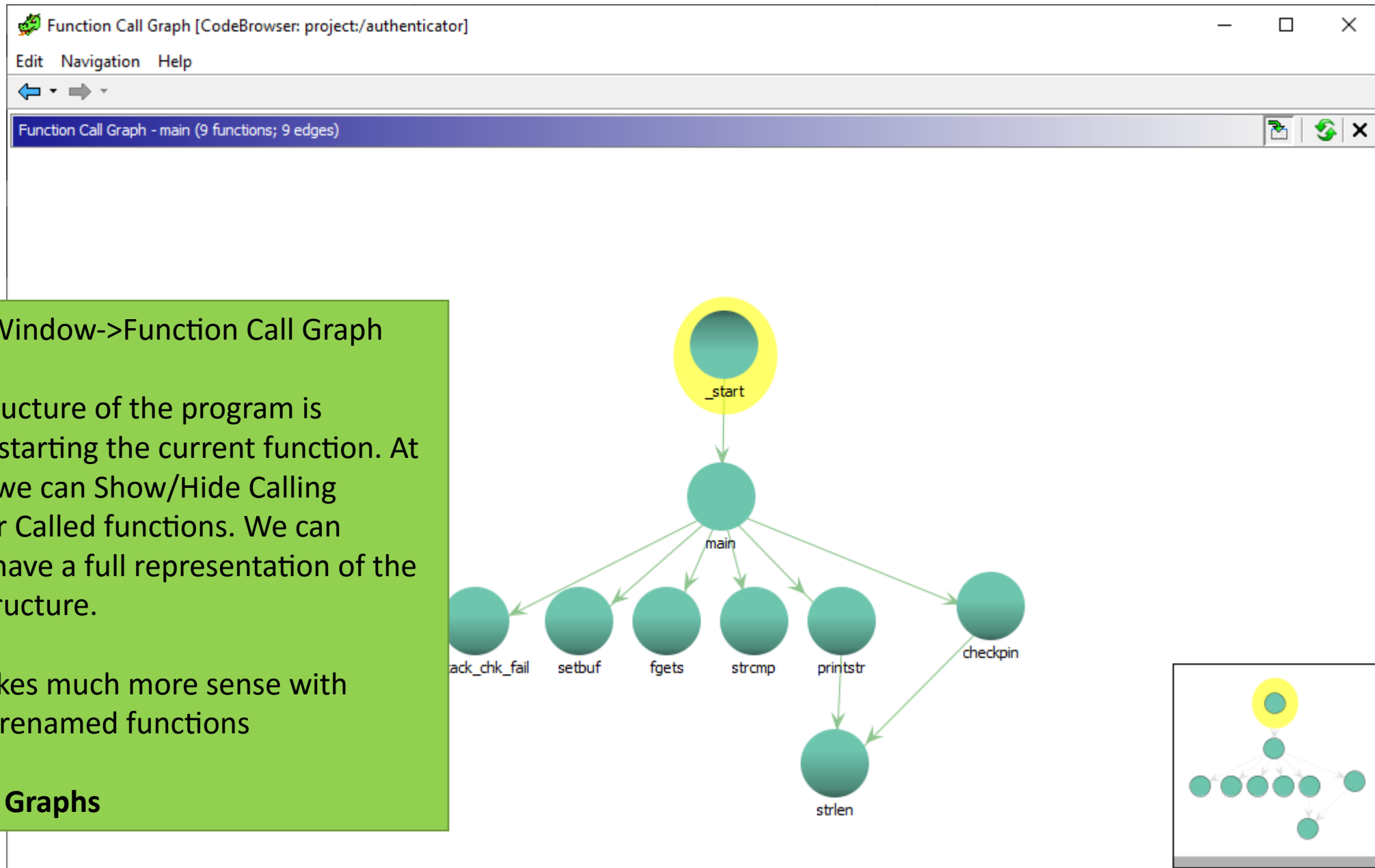
In Menu->Window->Function Graph

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

Called: Control-Flow Graphs

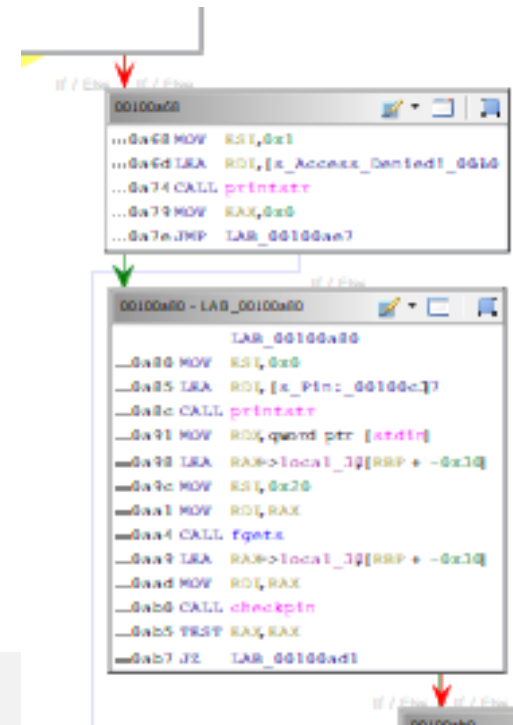
The screenshot displays the CodeBrowser interface for a function graph. The main window shows assembly code for the 'main' function, with a yellow oval highlighting the initial instructions. Below the code, a control flow graph (CFG) is generated, showing the logical structure of the function with nodes and edges representing branches. The nodes are labeled with addresses and instructions, such as '001000b - main', '0010040 - LAB_0010040', and '00100a7 - LAB_00100a7'. A smaller inset window on the right shows a simplified version of the CFG with a yellow oval highlighting the start node.

```
001000b - main
undefined:main
undefined:main ALLI <CRYSOR>
undefined:main Stack[-0x10]:local_10
undefined:main Stack[-0x30]:local_30
undefined:main Stack[-0x50]:local_50
main
...09db PUSH RBP
...09dc MOV RBP,RBP
...09dd SUB RSI,0x50
...09de MOV RAX,word ptr FS:[0x20]
...09df MOV word ptr [RBP + local_10],RAX
...09e0 XOR RAX,RAX
...09e1 MOV RAX,word ptr [rsi]
...09e2 MOV RAX,word ptr [rsi+4]
...09e3 MOV RSI,0x0
...09e4 MOV RDI,RAX
...09e5 CALL setbuf
...09e6 MOV RSI,0x0
...09e7 LRA RSI,[0AF_001000b]
...09e8 CALL printf
...09e9 MOV RSI,0x0
...09ea LRA RSI,[_Fltarea_enter_point]
...09eb CALL printf
...09ec MOV RSI,0x0
...09ed LRA RSI,[_Athen_ID: 00100c5]
...09ee CALL printf
...09ef MOV RDI,word ptr [rsi]
...09f0 LRA RAX>local_50[RBP + -0x50]
...09f1 MOV RDI,0x20
...09f2 MOV RDI,RAX
...09f3 CALL fgets
...09f4 LRA RAX>local_50[RBP + -0x50]
...09f5 LRA RSI,[_11337_00100c0]
...09f6 MOV RDI,RAX
...09f7 CALL strcmp
...09f8 PUSH RAX,RAX
...09f9 JE LAB_00100a0
```



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 $a \neq b$
 - Typical for a decision or preparation block

- Basic block b post-dominates a if every path through a also passes through b later
 - Typical for a wrap up or clean up block

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 section “beginning”)
 - objects which do not have data at the beginning
 - architecture uses variable length instructions (x86)

Linear Disassembly

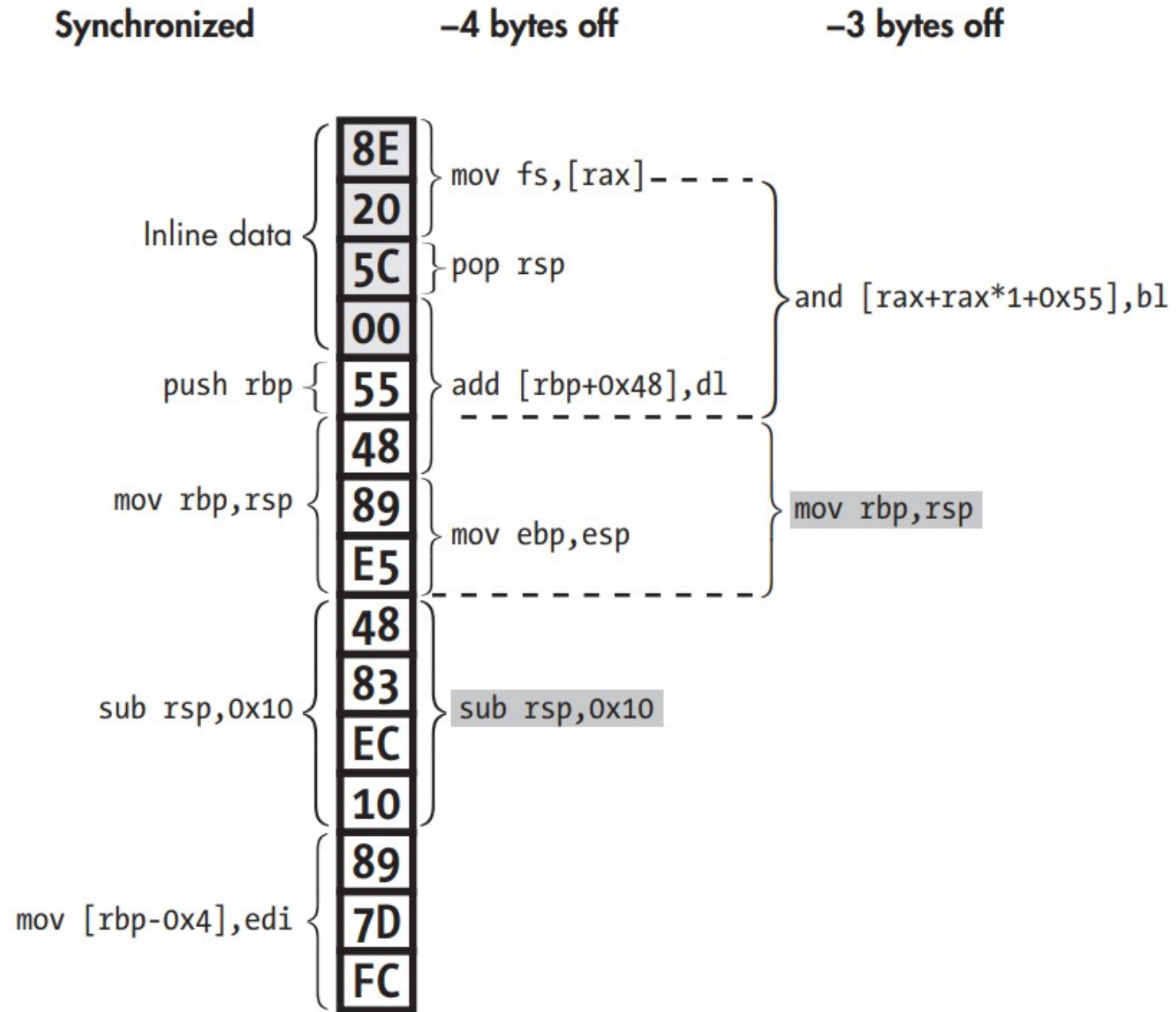
It is vital to define the initial address for decompiling.

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 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 inline data and Linear Disassembly is not adequate
 - Inline data: data (not code) together with code
 - 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

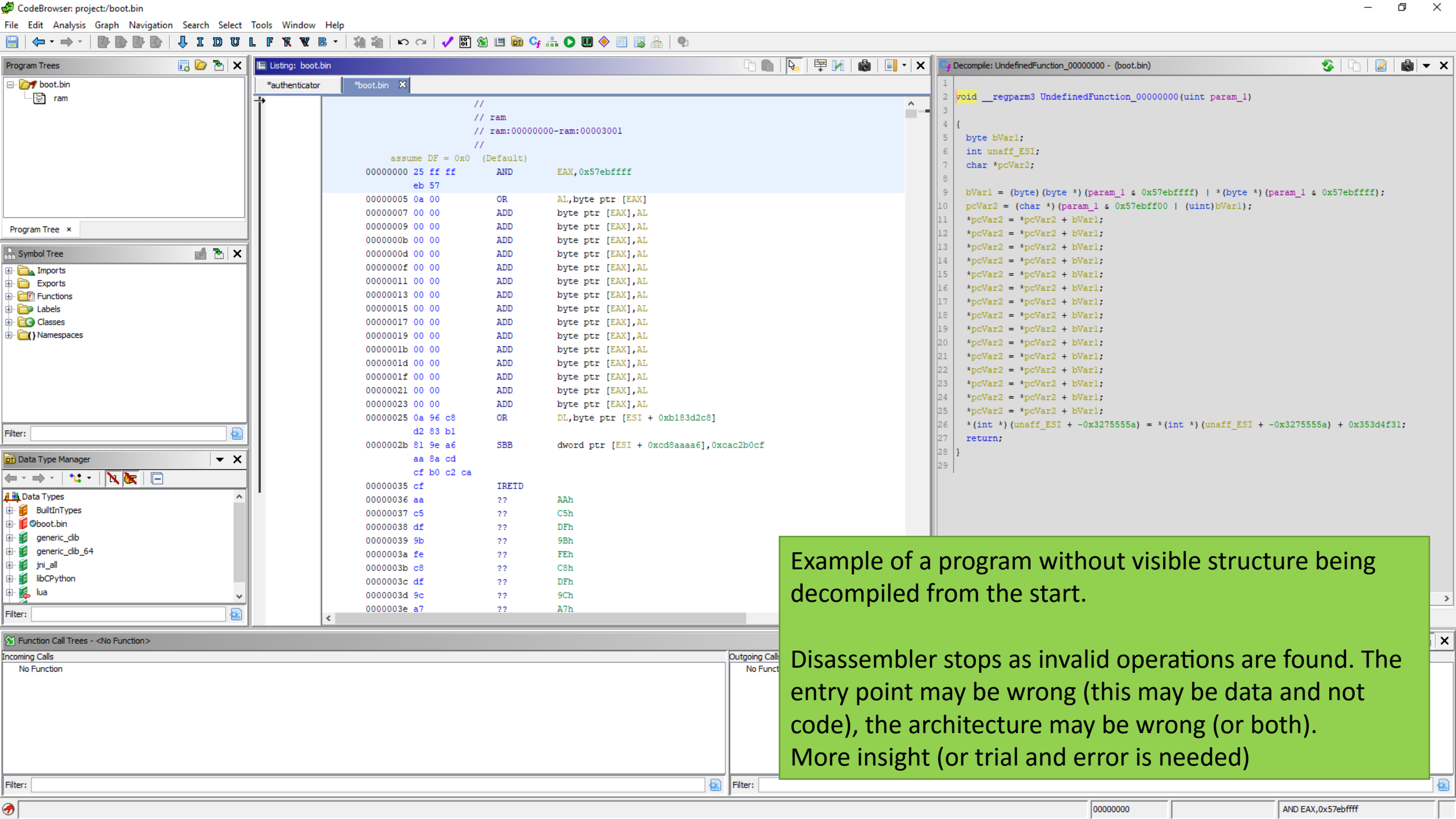
So why is it useful?

- 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

- Disassembles code since an initial point, while following the control flow
 - That is: follows jmp, call and ret
- IF 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*)) someaddress)()`



```
//
// ram
// ram:00000000-ram:00003001
//
assume DF = 0x0 (Default)
00000000 25 ff ff      AND     EAX,0x57ebffff
           eb 57
00000005 0a 00         OR     AL,byte ptr [EAX]
00000007 00 00         ADD   byte ptr [EAX],AL
00000009 00 00         ADD   byte ptr [EAX],AL
0000000b 00 00         ADD   byte ptr [EAX],AL
0000000d 00 00         ADD   byte ptr [EAX],AL
0000000f 00 00         ADD   byte ptr [EAX],AL
00000011 00 00         ADD   byte ptr [EAX],AL
00000013 00 00         ADD   byte ptr [EAX],AL
00000015 00 00         ADD   byte ptr [EAX],AL
00000017 00 00         ADD   byte ptr [EAX],AL
00000019 00 00         ADD   byte ptr [EAX],AL
0000001b 00 00         ADD   byte ptr [EAX],AL
0000001d 00 00         ADD   byte ptr [EAX],AL
0000001f 00 00         ADD   byte ptr [EAX],AL
00000021 00 00         ADD   byte ptr [EAX],AL
00000023 00 00         ADD   byte ptr [EAX],AL
00000025 0a 96 c8    OR     DL,byte ptr [ESI + 0xb183d2c8]
           d2 83 b1
0000002b 81 9e a6     SBB   dword ptr [ESI + 0xcd8aaaa6],0xcac2b0cf
           aa 8a cd
           cf b0 c2 ca
00000035 cf          IRETD
00000036 aa          ??    AAh
00000037 c5          ??    C5h
00000038 df          ??    DFh
00000039 9b          ??    9Bh
0000003a fe          ??    FEh
0000003b c8          ??    C8h
0000003c df          ??    DFh
0000003d 9c          ??    9Ch
0000003e a7          ??    A7h
```

```
Decompile: UndefinedFunction_00000000 - (boot.bin)
1
2 void __regparm3 UndefinedFunction_00000000(uint param_1)
3
4 {
5     byte bVar1;
6     int unaff_ESI;
7     char *pcVar2;
8
9     bVar1 = (byte)(byte *)(param_1 & 0x57ebffff) | *(byte *)(param_1 & 0x57ebffff);
10    pcVar2 = (char *) (param_1 & 0x57ebff00 | (uint)bVar1);
11    *pcVar2 = *pcVar2 + bVar1;
12    *pcVar2 = *pcVar2 + bVar1;
13    *pcVar2 = *pcVar2 + bVar1;
14    *pcVar2 = *pcVar2 + bVar1;
15    *pcVar2 = *pcVar2 + bVar1;
16    *pcVar2 = *pcVar2 + bVar1;
17    *pcVar2 = *pcVar2 + bVar1;
18    *pcVar2 = *pcVar2 + bVar1;
19    *pcVar2 = *pcVar2 + bVar1;
20    *pcVar2 = *pcVar2 + bVar1;
21    *pcVar2 = *pcVar2 + bVar1;
22    *pcVar2 = *pcVar2 + bVar1;
23    *pcVar2 = *pcVar2 + bVar1;
24    *pcVar2 = *pcVar2 + bVar1;
25    *pcVar2 = *pcVar2 + bVar1;
26    *(int *) (unaff_ESI + -0x3275555a) = *(int *) (unaff_ESI + -0x3275555a) + 0x353d4f31;
27    return;
28 }
29
```

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)))`
 - `__builtin_return_address(n)`

Function detection

Typical Prologue with Stack Guard

Stack allocation code

- Stores RBP
- Makes RBP = RSP
- Allocates 0x30 bytes

```
00400af7 55          PUSH    RBP
00400af8 48 89 e5    MOV     RBP,RSP
00400afb 48 83 ec 30  SUB     RSP,0x30
00400aff 89 7d dc    MOV     dword ptr [RBP + local_2c],EDI
00400b02 64 48 8b    MOV     RAX,qword ptr FS:[0x28]
           04 25 28
           00 00 00
00400b0b 48 89 45 f8  MOV     qword ptr [RBP + local_10],RAX
00400b0f 31 c0      XOR     EAX,EAX
```

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

```
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
```

Fetches the Canary

- 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

Function detection

Non Returning Functions

- Some functions, like `exit` or `abort`, are non-returning functions.
 - Such functions do not return to the caller after executing.
 - Instead, they do drastic things like halting the execution of the program.
- If the SRE tool does not know that a function is non-returning, it will assume that bytes after the function are instructions and attempt to disassemble them.
 - This should not happen as one of the instructions would interrupt execution

Function detection

Non Returning Functions

```
00000000000003cf <printInputThenLoop1>:
3cf:  55                push   %rbp
3d0:  48 89 e5          mov    %rsp,%rbp
a
3d7:  48 89 7d f8       mov    %rdi,-0x8(%rbp)
3db:  48 89 75 f0       mov    %rsi,-0x10(%rbp)
3df:  48 8b 55 f0       mov    -0x10(%rbp),%rdx
3e3:  48 8b 45 f8       mov    -0x8(%rbp),%rax
3e7:  48 8d 0d 39 0f 00 00 lea   0xf39(%rip),%rcx
3ee:  48 89 c6          mov    %rax,%rsi
3f1:  48 89 cf          mov    %rcx,%rdi
3f4:  b8 00 00 00 00    mov    $0x0,%eax
3f9:  e8 c2 fe ff ff    call   2c0 <printf@plt>a
3fe:  bf 01 00 00 00    mov    $0x1,%edi
403:  e8 c8 fe ff ff    call   2d0 <loopForever@plt>
408:  ff                (bad)
409:  ff                .byte 0xff
```

```
00000000000003a9 <loopForever>:
3a9:  55                push   %rbp
3aa:  48 89 e5          mov    %rsp,%rbp
3ad:  48 83 ec 10       sub    $0x10,%rsp
3b1:  89 7d fc          mov    %edi,-0x4(%rbp)
3b4:  8b 45 fc          mov    -0x4(%rbp),%eax
3b7:  48 8d 15 62 0f 00 00 lea   0xf62(%rip),%rdx
3be:  89 c6             mov    %eax,%esi
3c0:  48 89 d7          mov    %rdx,%rdi
3c3:  b8 00 00 00 00    mov    $0x0,%eax
3c8:  e8 f3 fe ff ff    call   2c0 <printf@plt>
3cd:  eb e5             jmp    3b4
<loopForever+0xb>
```

Calling Conventions

- Compilers handle the function calling processes differently
 - They use 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 from caller to the callee
 - How return codes are passed to the caller
 - Who allocates the stack, and how
 - Who stores important registers such as the Program Counter

Calling Conventions

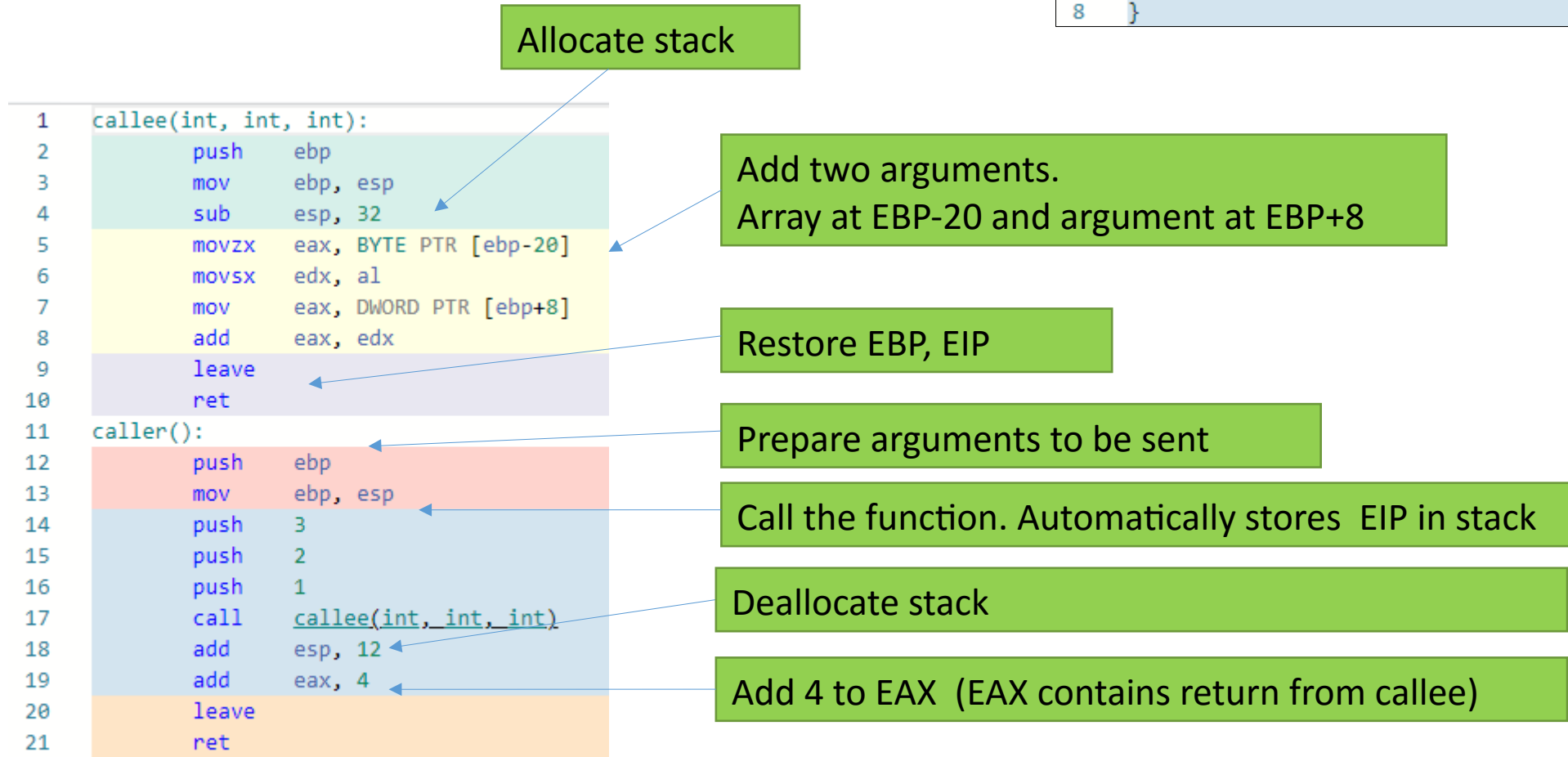
cdecl

- 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 IP, A, C, D
 - Callee saves BP, and others and restores RIP

Calling Conventions

cdecl

```
1 int callee(int a, int b, int c) {  
2     char d[20];  
3     return a + d[0];  
4 }  
5  
6 int caller(void) {  
7     return callee(1, 2, 3) + 4;  
8 }
```



Calling Conventions

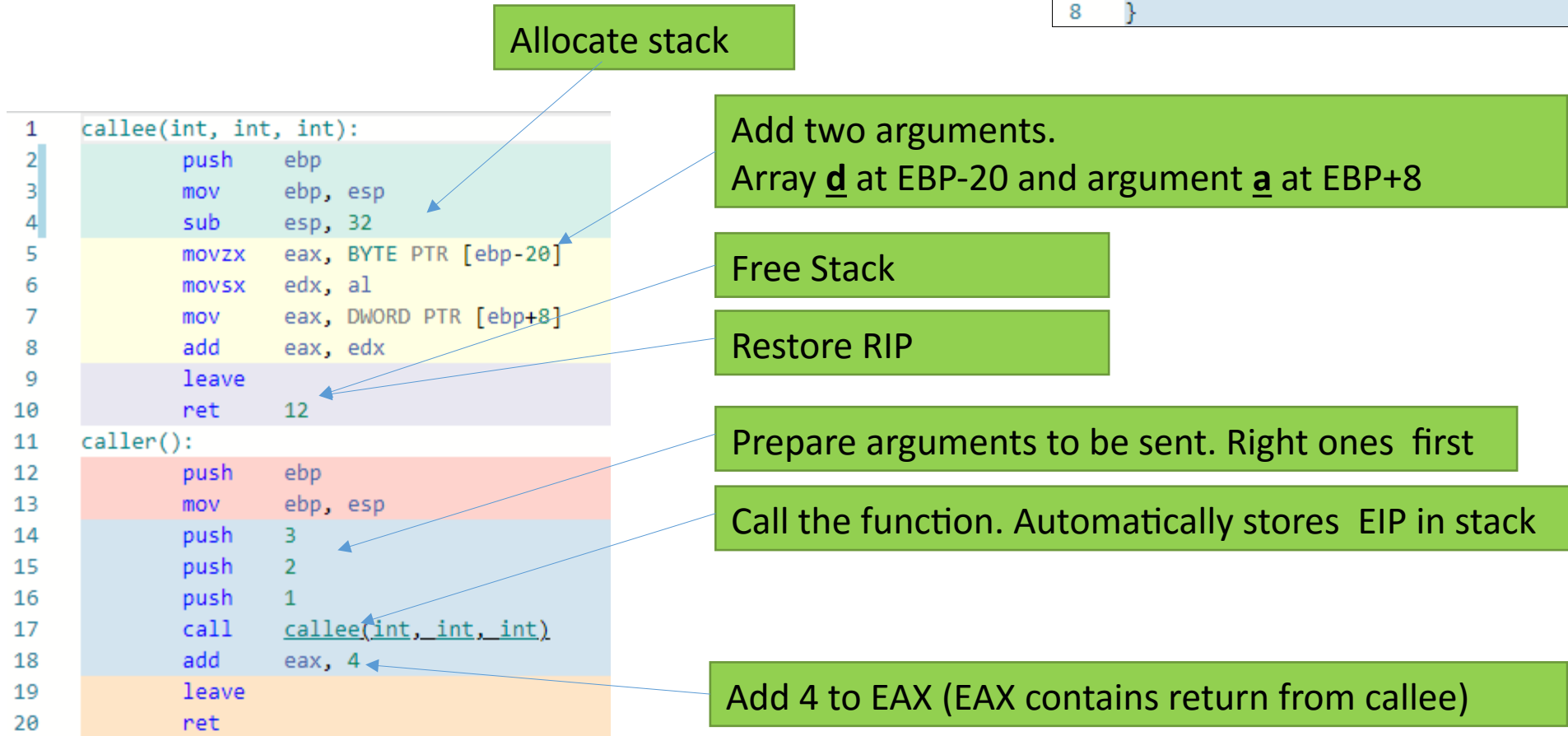
stdcall

- 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 by leaf. Allocated by callee
 - 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

```
1 int callee(int a, int b, int c) {  
2     char d[20];  
3     return a + d[0];  
4 }  
5  
6 int caller(void) {  
7     return callee(1, 2, 3) + 4;  
8 }
```



Calling Conventions

fastcall

- Official call convention for Win32API 64bits
- **Arguments: left to right, first arguments 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)

```
1 int callee(int a, int b, int c) {  
2     char d[20];  
3     return a + d[0];  
4 }  
5  
6 int caller(void) {  
7     return callee(1, 2, 3) + 4;  
8 }
```

```
1 callee(int, int, int):  
2     push    ebp  
3     mov     ebp, esp  
4     sub     esp, 40  
5     mov     DWORD PTR [ebp-36], ecx  
6     mov     DWORD PTR [ebp-40], edx  
7     movzx  eax, BYTE PTR [ebp-20]  
8     movsx  edx, al  
9     mov     eax, DWORD PTR [ebp-36]  
10    add     eax, edx  
11    leave  
12    ret     4  
13 caller():  
14    push    ebp  
15    mov     ebp, esp  
16    push    3  
17    mov     edx, 2  
18    mov     ecx, 1  
19    call   callee(int, int, int)  
20    add     eax, 4  
21    leave  
22    ret
```

Allocate stack

Stores arguments to stack

Adds (char) d[0] with a at EBP-36
d[0] at EBP-20

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

```
1 callee(int, int, int):
2     push    ebp
3     mov     ebp, esp
4     sub     esp, 32
5     movzx   eax, BYTE PTR [ebp-20]
6     movsx   edx, al
7     mov     eax, DWORD PTR [ebp+8]
8     add     eax, edx
9     leave
10    ret
11 caller():
12    push    ebp
13    mov     ebp, esp
14    push    3
15    push    2
16    push    1
17    call    callee(int, int, int)
18    add     esp, 12
19    add     eax, 4
20    leave
21    ret
```

cdecl

```
1 callee(int, int, int):
2     push    ebp
3     mov     ebp, esp
4     sub     esp, 32
5     movzx   eax, BYTE PTR [ebp-20]
6     movsx   edx, al
7     mov     eax, DWORD PTR [ebp+8]
8     add     eax, edx
9     leave
10    ret    12
11 caller():
12    push    ebp
13    mov     ebp, esp
14    push    3
15    push    2
16    push    1
17    call    callee(int, int, int)
18    add     eax, 4
19    leave
20    ret
```

stdcall

```
1 1 callee(int, int, int):
2     push    ebp
3     mov     ebp, esp
4     sub     esp, 40
5     mov     DWORD PTR [ebp-36], ecx
6     mov     DWORD PTR [ebp-40], edx
7     movzx   eax, BYTE PTR [ebp-20]
8     movsx   edx, al
9     mov     eax, DWORD PTR [ebp-36]
10    add     eax, edx
11    leave
12    ret     4
13 2 caller():
14    push    ebp
15    mov     ebp, esp
16    push    3
17    mov     edx, 2
18    mov     ecx, 1
19    call    callee(int, int, int)
20    add     eax, 4
21    leave
22    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)

```
1 int callee(int a, int b, int c) {  
2     char d[20];  
3     return a + d[0];  
4 }  
5  
6 int caller(void) {  
7     return callee(1, 2, 3) + 4;  
8 }
```

```
5 int callee(int,int,int) PROC  
6 $LN3:  
7     mov     DWORD PTR [rsp+24], r8d  
8     mov     DWORD PTR [rsp+16], edx  
9     mov     DWORD PTR [rsp+8], ecx  
10    sub     rsp, 40  
11    mov     eax, 1  
12    imul   rax, rax, 0  
13    movsx  eax, BYTE PTR d$[rsp+rax]  
14    mov     ecx, DWORD PTR a$[rsp]  
15    add     ecx, eax  
16    mov     eax, ecx  
17    add     rsp, 40  
18    ret     0  
19 int callee(int,int,int) ENDP  
20  
21 int caller(void) PROC  
22 $LN3:  
23    sub     rsp, 40  
24    mov     r8d, 3  
25    mov     edx, 2  
26    mov     ecx, 1  
27    call   int callee(int,int,int)  
28    add     eax, 4  
29    add     rsp, 40  
30    ret     0  
31 int caller(void) ENDP
```

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)

d\$ = 0
__\$ArrayPad\$ = 24
a\$ = 48
b\$ = 56
c\$ = 64

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

```
1 int callee(int a, int b, int c) {  
2     char d[20];  
3     return a + d[0];  
4 }  
5  
6 int caller(void) {  
7     return callee(1, 2, 3) + 4;  
8 }
```

Leaf function uses stack directly (Shadow)

```
1 callee(int, int, int):  
2     push    rbp  
3     mov     rbp, rsp  
4     mov     DWORD PTR [rbp-36], edi  
5     mov     DWORD PTR [rbp-40], esi  
6     mov     DWORD PTR [rbp-44], edx  
7     movzx  eax, BYTE PTR [rbp-32]  
8     movsx  edx, al  
9     mov     eax, DWORD PTR [rbp-36]  
10    add     eax, edx  
11    pop     rbp  
12    ret  
13 caller():  
14    push   rbp  
15    mov   rbp, rsp  
16    mov   edx, 3  
17    mov   esi, 2  
18    mov   edi, 1  
19    call callee(int, int, int)  
20    add   eax, 4  
21    pop   rbp  
22    ret
```

Adds (char) d[0] with a
d[0] is at RBP-32. d[1] at RBP-31...
a is at RBP-36

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

```
1  callee(int, int, int):
2      push    rbp
3      mov     rbp, rsp
4      mov     DWORD PTR [rbp-36], edi
5      mov     DWORD PTR [rbp-40], esi
6      mov     DWORD PTR [rbp-44], edx
7      movzx   eax, BYTE PTR [rbp-32]
8      movsx   edx, al
9      mov     eax, DWORD PTR [rbp-36]
10     add     eax, edx
11     pop     rbp
12     ret
13  caller():
14     push    rbp
15     mov     rbp, rsp
16     mov     edx, 3
17     mov     esi, 2
18     mov     edi, 1
19     call   callee(int, int, int)
20     add     eax, 4
21     pop     rbp
22     ret
```

System V AMD64 ABI

```
5  int callee(int,int,int) PROC
6  $LN3:
7      mov     DWORD PTR [rsp+24], r8d
8      mov     DWORD PTR [rsp+16], edx
9      mov     DWORD PTR [rsp+8], ecx
10     sub     rsp, 40
11     mov     eax, 1
12     imul   rax, rax, 0
13     movsx  eax, BYTE PTR d$[rsp+rax]
14     mov     ecx, DWORD PTR a$[rsp]
15     add     ecx, eax
16     mov     eax, ecx
17     add     rsp, 40
18     ret     0
19  int callee(int,int,int) ENDP
20
21  int caller(void) PROC
22  $LN3:
23     sub     rsp, 40
24     mov     r8d, 3
25     mov     edx, 2
26     mov     ecx, 1
27     call   int callee(int,int,int)
28     add     eax, 4
29     add     rsp, 40
30     ret     0
31  int caller(void) ENDP
```

fastcall

Common Logic Structures

- It's important to recognize basic flow control structures
 - Remember that the decompiler may be unreliable
- Basic structures (besides functions):
 - 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 structured **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]:local_20
    undefined4         Stack[-0x24]:local_24
    undefined1         Stack[-0x25]:local_25
    undefined1         Stack[-0x26]:local_26
    undefined8         Stack[-0x30]: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_24],...
...096a LEA RAX,[s_}a:Vh|}a:g|8j=}89gV
...0971 MOV qword ptr [RBP + local_20]...
...0975 MOV dword ptr [RBP + local_24]...
...097c JMP LAB_001009b4
```

```
001009b0 - LAB_001009b0
LAB_001009b0
...09b0 ADD dword ptr [RBP + local_24]...
```

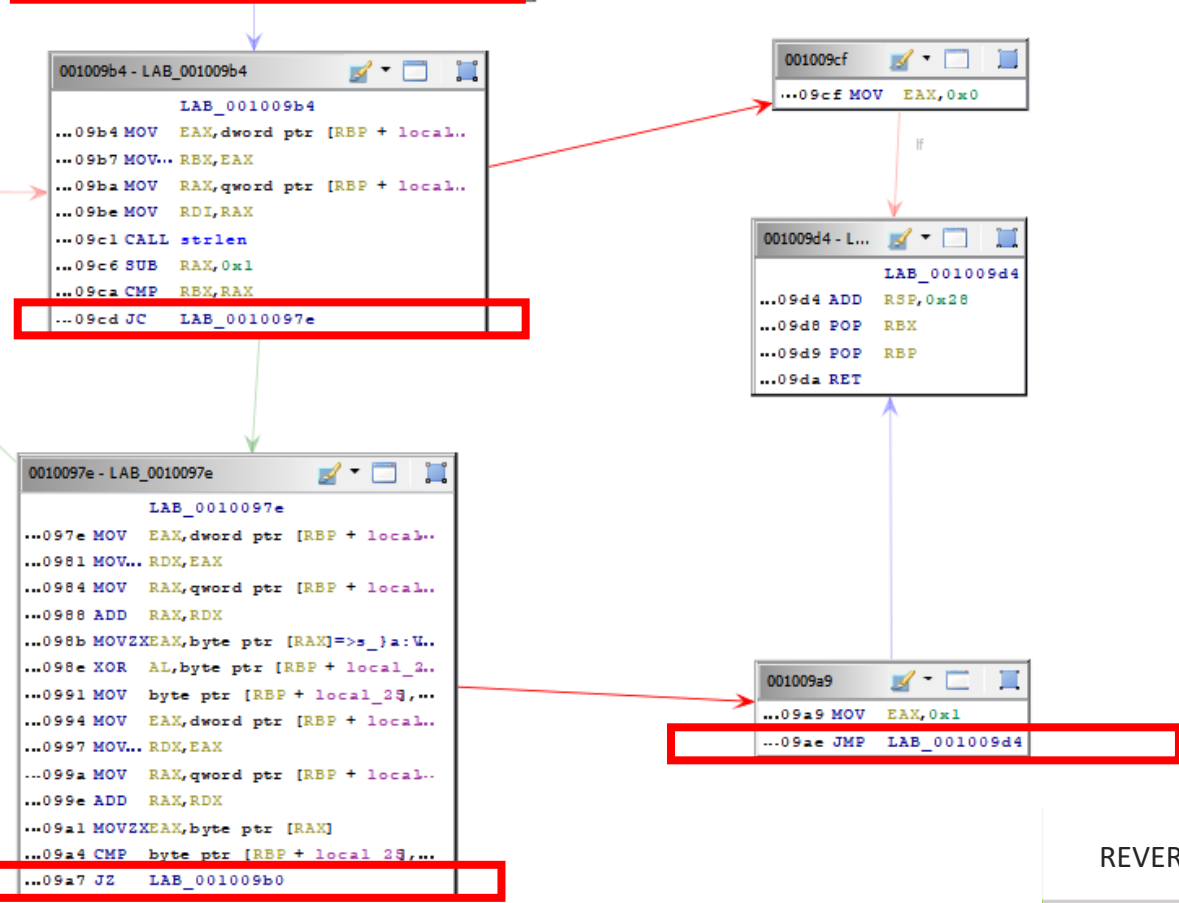
```
001009b4 - LAB_001009b4
LAB_001009b4
...09b4 MOV EAX,dword ptr [RBP + local_..
...09b7 MOV... RBX,EAX
...09ba MOV RAX,qword ptr [RBP + local_..
...09be MOV RDI,RAX
...09c1 CALL strlen
...09c6 SUB RAX,0x1
...09ca CMP RBX,RAX
...09cd JC LAB_0010097e
```

```
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
...098b MOVZX EAX,byte ptr [RAX]=>s_}a:U..
...098e XOR AL,byte ptr [RBP + local_2..
...0991 MOV byte ptr [RBP + local_23],...
...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_23],...
...09a7 JZ LAB_001009b0
```

```
001009a9
...09a9 MOV EAX,0x1
...09ae JMP LAB_001009d4
```



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...

```
1 int bar(int b) {  
2     return b * b;  
3 }  
4  
5 int foo(int a) {  
6     if(a == 0){  
7         return bar(a) * 1;  
8     }  
9     else  
10        if(a < 0){  
11            return bar(a) - 1;  
12        }  
13        else{  
14            return bar(a) + 1;  
15        }  
16    }  
17 }
```

```
1  bar:  
2      imul    edi, edi  
3      mov     eax, edi  
4      ret  
5  foo:  
6      test    edi, edi  
7      je     .L6  
8      js     .L7  
9      call   bar  
10     add    eax, 1  
11     ret  
12  .L6:  
13     call   bar  
14     ret  
15  .L7:  
16     call   bar  
17     sub    eax, 1  
18     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 may be dangerous if breaks are missing
- **test**: compare two registers. Set 3 flags:
 - PF: Even number of bits
 - ZF: Zero
 - SF: Signed value

```
1 int bar(int b) {
2     return b * b;
3 }
4
5 int foo(int a) {
6     switch(a){
7         case 0:
8             a = bar(1) + 1;
9             break;
10        case 1:
11            a = bar(2+ a) + 2;
12            break;
13        case 3:
14            a = bar(3) + 3;
15        default:
16            a = bar(4) + 4;
17        }
18
19    return a;
20 }
21
```

```
1 bar:
2     imul    edi, edi
3     mov     eax, edi
4     ret
5
6 foo:
7     test    edi, edi
8     je     .L3
9     cmp    edi, 1
10    je     .L4
11    mov     edi, 4
12    call   bar
13    add     eax, 4
14    ret
15 .L3:
16    mov     edi, 1
17    call   bar
18    add     eax, 1
19    ret
20 .L4:
21    add     edi, 2
22    call   bar
23    add     eax, 2
24    ret
```

Common Logic Structures

loops

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

```
1 int bar(int b) {
2     return b * b;
3 }
4
5 int foo(int a) {
6     int b = 0;
7     for(int i = 0; i < a; i++){
8         b += bar(i);
9     }
10
11     return b;
12 }
13
14 int caller(void) {
15     return callee(1, 2, 3) + 4;
16 }
```

```
1  bar:
2      imul    edi, edi
3      mov     eax, edi
4      ret
5  foo:
6      push   r12
7      push   rbp
8      push   rbx
9      mov    r12d, edi
10     mov    ebx, 0
11     mov    ebp, 0
12  .L3:
13     cmp    ebx, r12d
14     jge   .L6
15     mov    edi, ebx
16     call  bar
17     add    ebp, eax
18     add    ebx, 1
19     jmp   .L3
20  .L6:
21     mov    eax, ebp
22     pop    rbx
23     pop    rbp
24     pop    r12
25     ret
```

Prepares stack

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

- Loop body

Jump to top of loop

Defining Data Types

- One of the best ways to clean up the decompiled code is to apply data types.
- You can define types manually through the Data Type Manager.
- You can also have Ghidra help you by right-clicking on a variable in the decompiler view and selecting
 - Auto Create (Class) Structure, or
 - Auto Fill in (Class) Structure.
- You can parse data types from it by selecting File → Parse C Source... from the Code Browser, or load gdt files with definitions (e.g. the **jni_all.gdt** file)

Defining Data Types

After Parsing a .h header
Or Defining the HardwareDevice Structure

```
void print_device_status(uint *dev)
{
    if (dev != (uint *)0x0) {
        puts("--- Device Status Report ---");
        printf("ID: 0x%08X\n", (ulong)*dev);
        printf("Name: %s\n", dev + 6);
        if (dev[1] == 1) {
            puts("Status: ONLINE");
            printf("Temperature: %.2fC\n", (double)(float)dev[2]);
            printf("IPv4: 0x%08X\n", (ulong)dev[4]);
        }
        else {
            printf("Status: OFFLINE (%d)\n", (ulong)dev[1]);
        }
        puts("-----");
    }
    return;
}
```

```
void print_device_status(HardwareDevice *dev)
{
    if (dev != (HardwareDevice *)0x0) {
        puts("--- Device Status Report ---");
        printf("ID: 0x%08X\n", (ulong)dev->device_id);
        printf("Name: %s\n", dev->device_name);
        if (dev->current_state == STATE_ONLINE) {
            puts("Status: ONLINE");
            printf("Temperature: %.2fC\n", (double)dev->temperature);
            printf("IPv4: 0x%08X\n", (ulong)(dev->net_id).ipv4_addr);
        }
        else {
            printf("Status: OFFLINE (%d)\n", (ulong)dev->current_state);
        }
        puts("-----");
    }
    return;
}
```

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
 - Constructors, 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

```
1  #include <iostream>
2  #include <string>
3
4  class A {
5      std::string text1;
6
7      public:
8      A(std::string text1) {
9          this->text1 = text1;
10     }
11     void print() {
12         std::cout << this->text1 << std::endl;
13     }
14 };
15
16 int main(int argc, char** argv) {
17     A a(std::string("Hello World"));
18     a.print();
19
20 }
```

hello1.cpp

```
1 $ readelf --dyn-sym hello1
2
3 Symbol table '.dynsym' contains 21 entries:
4   Num:      Value              Size Type   Bind   Vis      Ndx Name
5     0: 0000000000000000          0 NOTYPE LOCAL DEFAULT UND
6     1: 0000000000000000          0 FUNC   GLOBAL DEFAULT UND _ZNSt7__cxx1112basic_stri@GLIBCXX_3.4.21 (2)
7     2: 0000000000000000          0 FUNC   GLOBAL DEFAULT UND _ZSt4endlIcSt11char_trait@GLIBCXX_3.4 (4)
8     3: 0000000000000000          0 FUNC   GLOBAL DEFAULT UND _ZNSt7__cxx1112basic_stri@GLIBCXX_3.4.21 (2)
9     4: 0000000000000000          0 FUNC   GLOBAL DEFAULT UND __cxa_atexit@GLIBC_2.2.5 (3)
10    5: 0000000000000000          0 FUNC   GLOBAL DEFAULT UND _ZSt1sIcSt11char_traitsIc@GLIBCXX_3.4.21 (2)
11    6: 0000000000000000          0 FUNC   GLOBAL DEFAULT UND _ZNSolsEPFRSoS_E@GLIBCXX_3.4 (4)
12    7: 0000000000000000          0 FUNC   GLOBAL DEFAULT UND _ZNSaIcED1Ev@GLIBCXX_3.4 (4)
13    8: 0000000000000000          0 FUNC   GLOBAL DEFAULT UND _ZNSt7__cxx1112basic_stri@GLIBCXX_3.4.21 (2)
14    9: 0000000000000000          0 FUNC   GLOBAL DEFAULT UND _ZNSt7__cxx1112basic_stri@GLIBCXX_3.4.21 (2)
15   10: 0000000000000000          0 FUNC   GLOBAL DEFAULT UND _ZNSt8ios_base4InitC1Ev@GLIBCXX_3.4 (4)
16   11: 0000000000000000          0 FUNC   GLOBAL DEFAULT UND __gxx_personality_v0@CXXABI_1.3 (5)
17   12: 0000000000000000          0 NOTYPE WEAK   DEFAULT UND _ITM_deregisterTMCloneTab
18   13: 0000000000000000          0 FUNC   GLOBAL DEFAULT UND _Unwind_Resume@GCC_3.0 (6)
19   14: 0000000000000000          0 FUNC   GLOBAL DEFAULT UND _ZNSaIcEC1Ev@GLIBCXX_3.4 (4)
20   15: 0000000000000000          0 FUNC   GLOBAL DEFAULT UND __libc_start_main@GLIBC_2.2.5 (3)
21   16: 0000000000000000          0 NOTYPE WEAK   DEFAULT UND __gmon_start__
22   17: 0000000000000000          0 NOTYPE WEAK   DEFAULT UND _ITM_registerTMCloneTable
23   18: 0000000000000000          0 FUNC   GLOBAL DEFAULT UND _ZNSt8ios_base4InitD1Ev@GLIBCXX_3.4 (4)
24   19: 0000000000000000          0 FUNC   WEAK   DEFAULT UND __cxa_finalize@GLIBC_2.2.5 (3)
25   20: 00000000000040a0        272 OBJECT GLOBAL DEFAULT 26 _ZSt4cout@GLIBCXX_3.4 (4)
```

C++ code

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

- Constructors
- Methods
- Destructors

```
1
2 /* WARNING: Unknown calling convention yet parameter storage is locked */
3
4 int main(void)
5
6 {
7     A local_68 [32];
8     basic_string<char, std::char_traits<char>, std::allocator<char>> local_48 [47];
9     allocator<char> local_19 [9];
10
11     std::allocator<char>::allocator();
12         /* try { // try from 00101203 to 00101207 has its CatchHandler @ 00101263 */
13     std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>::basic_string
14         ((char *)local_48, (allocator *)"Hello World");
15         /* try { // try from 00101216 to 0010121a has its CatchHandler @ 00101252 */
16     A::A(local_68, (basic_string)0xb8);
17     std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>::~~basic_string
18         (local_48);
19     std::allocator<char>::~~allocator(local_19);
20         /* try { // try from 0010123a to 0010123e has its CatchHandler @ 0010127d */
21     A::print(local_68);
22     A::~~A(local_68);
23     return 0;
24 }
25
```

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
0010123a e8 11 01      CALL   A::print
              00 00
              } // end try from 0010123a to 0010123e
              XREF[1]: 001
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
First argument of all non static class members

```
*****  
* 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)  
                XREF[4]:    Entry Point(*), main:0010123a(c),  
                00102058, 00102150 (*)  
_ZN1A5printEv  
A::print  
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
```