# Software Reverse Engineering

# SRE Background

- Before starting lets go overs some concepts of background knowledge for this
  - Operating Systems
    - Syscalls
    - Emulation
    - Kernel space interaction
  - Programming Language influence
    - C/C++
- Architectural Influenences
  - Drivers
  - Shared Libraries

## Software Reverse Engineering

- When someone typically thinks of reverse engineering in cybersecurity this is typically where they fall taking an existing executable and disassembling it for the expressed purpose of understanding it’s functionality
- For this Series’s purpose, We’ll be focusing on Linux Executable however much of the concepts covered apply to most executable file formats
- Linux(Broadly) uses ELFs(**E**xecutable **L**inking **F**ile) as a executable format in which executable code, resources, and linking information is stored in a defined file format.
- Depending on the CPU architecture that the file is compiled for some sections of the executable may be exclude or extra sections included

# Static Analysis

- Static Analysis is typically the first stage in any reverse engineering endeavor, It typically involves pulling as much information as possible out of an executable with having to execute it in it’s entirety, this is done for a couple of reasons
- Depending on the purpose of a project the approach may differ but typically can be placed into broad goals for the static analysis phase
  - Identification of
    - critical executable attributes
    - Contextual Resources
    - Strings, static datastructs, and OOP Resources(Typically not Human readable)
    - Tagging Areas of interest in the Assembly
    - OS environment requirements
- There can be more or less than these goals as these are more of starting point depending on your goals

## Dynamic Analysis

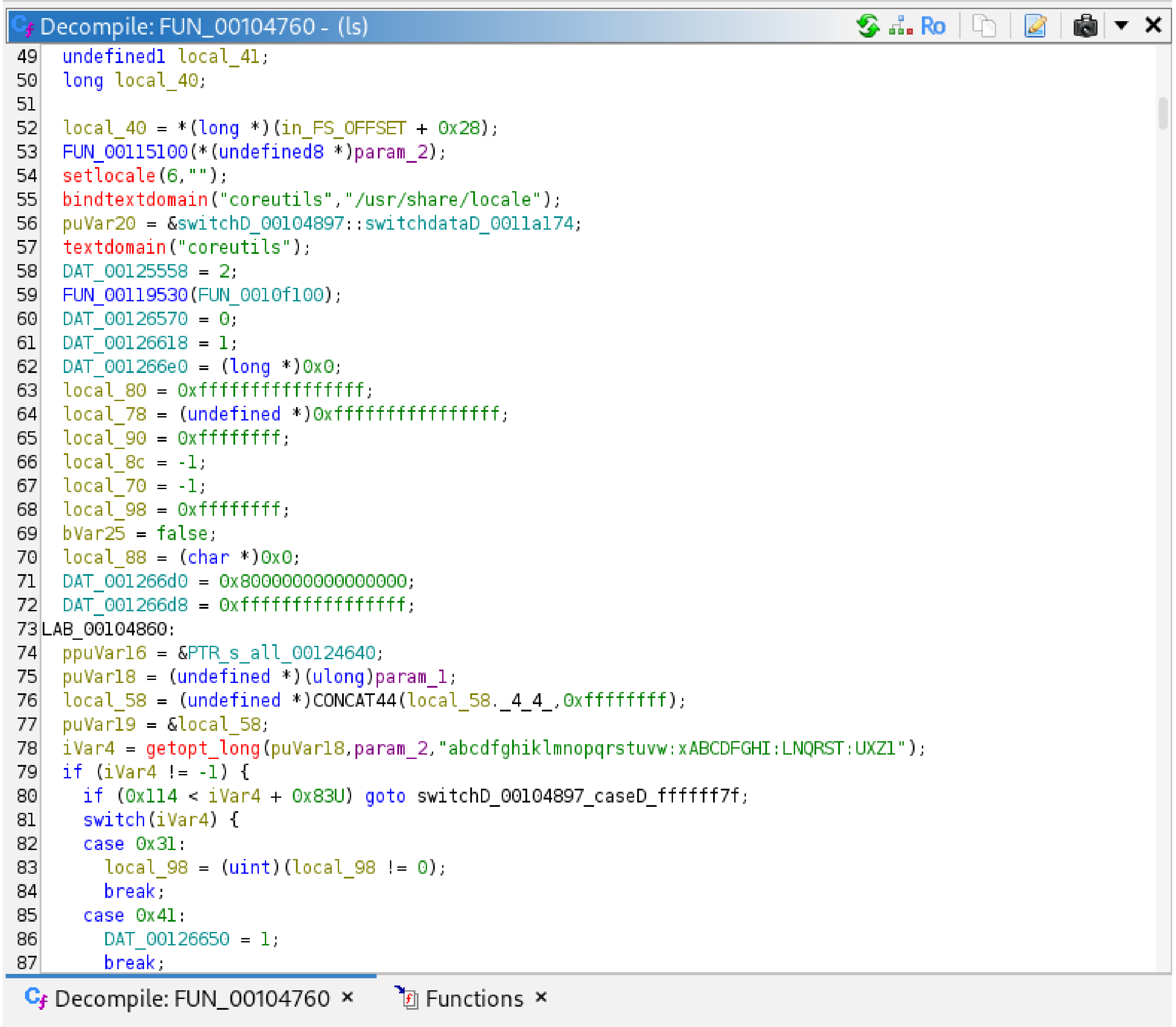
- Dynamic Analysis typically is a later phase in which actions such as debugging and fuzzing occur, often this is most time consuming portion
- Once again there are multiple valid approaches to this phase, often isolating the executable and execution environments is a good start
  - Docker Container(Same Arch)
  - Virtual Machine(Extensible but large)
  - Creating a cross compiled chroot
- This is done so that the executable can be observed in a type of ‘cleanroom’ for not only safety but also for logging purposes as having the executable directly use system resources can make things difficult for actions such as tracing and following the execution path

## Tool Usage

- For Our contexts we’ll be using a couple of tools
  - Objdump – Compiler/Arch Specific Disassembler
  - Capstone Tool – PIC Disassembler
  - ReadElf – ELF Header Parser
- As far as frameworks go, it’s a matter of choice but for ours it’s
  - Ghidra
  - Radare2
  - qemu

# Disassembly vs. Decompilation

- There are times where these two terms are used interchangeably however they are distinct concepts from each other
  - Disassembly
    - The action of taking compiled executable code and extracting the contextual information(Opcodes) being used as instructions while preserving the interpretation of instructions as sent to the CPU
  - Decompilation
    - The action of disassembling the executable code into it’s opcodes then “recompiling” them into an approximation of the source code used to generate the executable code



```
  undefined1 local_41;
  long local_40;

  local_40 = *(long *)(in_FS_OFFSET + 0x28);
  FUN_00115100(*(undefined8 *)param_2);
  setlocale(6,"");
  bindtextdomain("coreutils","/usr/share/locale");
  puVar20 = &switchD_00104897::switchdataD_0011a174;
  textdomain("coreutils");
  DAT_00125558 = 2;
  FUN_00119530(FUN_0010f100);
  DAT_00126570 = 0;
  DAT_00126618 = 1;
  DAT_001266e0 = (long *)0x0;
  local_80 = 0xffffffffffffffff;
  local_78 = (undefined *)0xffffffffffffffff;
  local_90 = 0xffffffff;
  local_8c = -1;
  local_70 = -1;
  local_98 = 0xffffffff;
  bVar25 = false;
  local_88 = (char *)0x0;
  DAT_001266d0 = 0x8000000000000000;
  DAT_001266d8 = 0xffffffffffffffff;
LAB_00104860:
  ppuVar16 = &PTR_s_all_00124640;
  puVar18 = (undefined *)(ulong)param_1;
  local_58 = (undefined *)CONCAT44(local_58._4_4_,0xffffffff);
  puVar19 = &local_58;
  iVar4 = getopt_long(puVar18,param_2,"abcdfghiklmnopqrstuvw:xABCDFGHI:LNQRST:UXZ1");
  if (iVar4 != -1) {
    if (0x114 < iVar4 + 0x83U) goto switchD_00104897_caseD_ffffff7f;
    switch(iVar4) {
    case 0x31:
      local_98 = (uint)(local_98 != 0);
      break;
    case 0x41:
      DAT_00126650 = 1;
      break;
```

# ELF Sections

- For most executable formats they separated into discrete sections which serve specific purposes
  - .text – Primary Executable Code
  - .(ro)data – Where many global and intializated local variables live
  - .bss – The dedicate section for the Heap and uninitialized local variables
  - .got – **G**lobal **O**ffset **T**able
  - .plt – **P**rocedural **L**inkage **T**able
- Specific  sections
  - .dtors & .ctors – C++ Destructor and Constructors
  - .symtab – Symbol table
  - .debug / .comment – Debugging and Version Info

# ELF Sections pt2

- Understanding the section purpose help in understanding how both the operating system and executable function for example being able to watch specific sections for changes is beneficial for debugging and analysis purposes
- Effectively since the file structure is replicated when loaded into memory, this also translates into the permissions for the program as well
- For Malware Analysis purposes, typically watching these permissions is crucial to locating memory only resources
- For More traditional Reverse Engineering purposes keeping track of these section permissions is a good start in understanding where to look for sensitive information

## Contextual Resources

- Contextual resources are used by a program to provide crucial functionality in order for the application to function
- This can include things like
  - Encryption Keys
  - Licensing Information
  - Anti-debugging  & Obfuscated resources
  - Intentional Program Opacity
  - Obfuscated and undocumented assembly
- Realistically, the importance of these resources can change but understanding and finding these resources can help in the dynamic phase

# Resource Discovery

- Many types of resources that are described in the data sections of executable, are useful in understanding the functionality of a program, but can also help regenerate custom datatypes and the OOP underpinnings
- Address Tables
  - Think of these as a meta-informational method of storing linked location inside of a program
- V(f)tables
  - These exist as specific type of Address table in which object primives are stored in a point array in order to express a more complex object
- By finding these structures it makes the whole program execution easier to follow and can even find previously referenced code!

# Actually running binaires

- Depending on the situation actually debugging the program is as complex or as simple as you want to make it
- Typically if your running on the same arch and target environment as the binary it’s as simple as running gdb
  - “gdb <target-program>”
- If remote?
  - Server
    - gdb-server <interface>:<port> <target-program>
  - Client
    - gdb -c “target remote <interface>:<port>”
- For much of gdb you can use the REPL cli for the debugging but if would like there are more User-friendly frontends for the

# Running on different arch

- When you need to run a program on a different arch,
  - If you have a device which can run the program properly use it(duh!)
  - If you have “the” device but can’t run anything on it(next week)
  - Virtualization
  - ESIL
- For virtualization
  - Create a Target arch QEMU VM
    - Typically debian is a good starting point has many archs available
  - Create a Static chroot using a qemu static user binary
    - Chroot . qemu-<arch> -g <binary to run>

## Framework Usage

- Reverse Engineering frameworks are useful in that they take multiple steps of the process and consolidate them for ease of use
- Many of them take time to learn how to use and unfortunately are not as user friendly as other programs but the time investment is well worth it
- Common Frameworks
  - Ghidra
  - Rardare2
  - IDA
- Realistically, no framework is worse or better than any other simply put they all have strengths and weaknesses