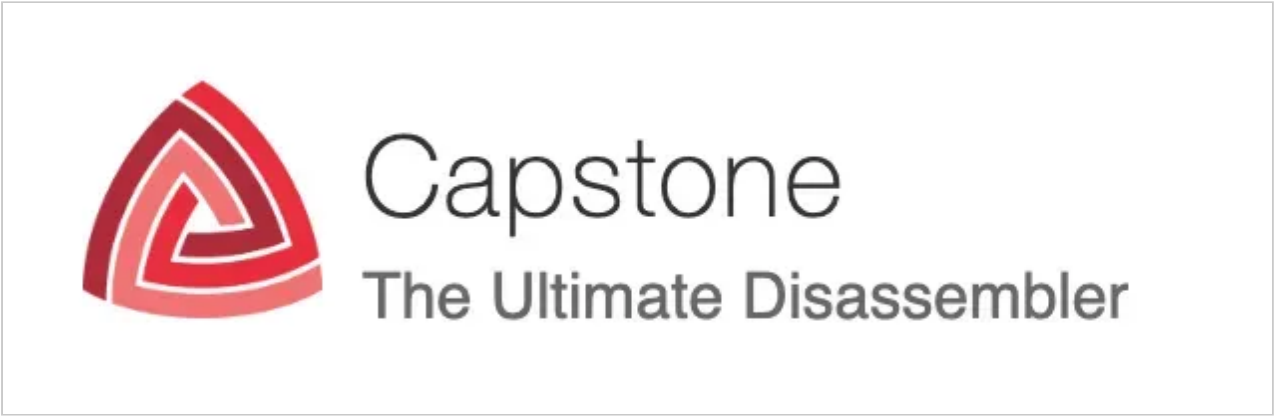
FIELD NOTE / MALWARE ANALYSIS

Capstone for Malware Analysis: A Beginner’s Guide

Petruknisme · 9 Sep, 2025 · 8 min to read

Malware analysis is all about understanding what a binary actually does: how it manipulates registers, memory, syscalls, and how it tries to hide its real intent. Honestly, having a fast, accurate disassembler like Capstone makes this so much easier.

Capstone is a lightweight, open-source multi-architecture disassembly engine that exposes a clean Python API (capstone). It can decode millions of instructions per second, supports Intel/AMD x86, x64, ARM, Thumb, MIPS, PowerPC, and more, and integrates seamlessly with other binary-analysis libraries. What sets it apart is its speed and scriptability, you can process hundreds of samples in minutes through Python scripts, making it ideal for bulk analysis and automation workflows that would be tedious with interactive tools like Ghidra or IDA.



Capstone

Capstone’s main advantages:

- Speed matters: While others GUI RE app is still loading, Capstone has already disassembled your sample
- Automation is king: Write Python scripts that analyze hundreds of samples automatically
- Works everywhere: Supports x86, x64, ARM, PowerPC, basically any architecture you’ll encounter
- No licensing headaches: It’s free and open-source
- Lightweight: Runs on potato hardware without breaking a sweat

Before we dive in, you should probably have at least:

- Basic understanding of assembly language concepts
- Familiarity with Python programming

- Knowledge of common malware types and behaviors

Why Capstone for Malware Analysis?
Malware authors are sneaky! They'll throw all sorts of tricks at you: obfuscated instructions, custom byte-code interpreters, anti-debug techniques (like that classic `int 0x2e`), or even self-modifying code. Capstone's accuracy and speed are a lifesaver when you're dealing with large binaries or memory dumps.

Setting Up Capstone

Getting Capstone up and running is pretty straightforward. The easiest method is through pip, Python's package manager. This will install both the core Capstone engine and Python bindings all at once:

Python Installation (Recommended):

```
❯ pip install capstone
Collecting capstone
  Downloading capstone-5.0.6-py3-none-macosx_11_0_arm64.whl.metadata (3.3
Downloading capstone-5.0.6-py3-none-macosx_11_0_arm64.whl (1.2 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.2/1.2 MB 10.8 MB/s eta 0:00:
Installing collected packages: capstone
Successfully installed capstone-5.0.6
```

Verification and Testing

Once you've got it installed, let's make sure everything's working. The most basic test is just checking that Python can find and import Capstone without any error.

check_capstone.py

```python
try:
    import capstone
    print("Capstone imported successfully!")
    print(f"Capstone version: {capstone.__version__}")
except ImportError as e:
    print(f"Import failed: {e}")
    print("Try: pip install capstone")
```

```
❯ python check_capstone.py
Capstone imported successfully!
Capstone version: 5.0.6
```

Next, let's see what architectures we've got available. Capstone supports a bunch of CPU architectures, and it's useful to understand the options available when analyzing malware samples:

check_capstone_arch.py

```python
import capstone
print("Discovering available architectures from Capstone library:")

# Get all CS_ARCH_* constants from capstone module
arch_constants = {}
for attr_name in dir(capstone):
    if attr_name.startswith('CS_ARCH_'):
        arch_value = getattr(capstone, attr_name)
        arch_name = attr_name.replace('CS_ARCH_', '')
        arch_constants[arch_name] = arch_value

print(f"Found {len(arch_constants)} architectures:")
for arch_name, arch_id in arch_constants.items():
    print(f"  {arch_name}: {arch_id}")
```

```
❯ python check_capstone_arch.py
Discovering available architectures from Capstone library:
Found 20 architectures:
  ALL: 65535
  ARM: 0
  ARM64: 1
  BPF: 14
  EVM: 11
  M680X: 10
  M68K: 8
  MAX: 18
  MIPS: 2
```

```
  MOS65XX: 12
  PPC: 4
  RISCV: 15
  SH: 16
  SPARC: 5
  SYSZ: 6
  TMS320C64X: 9
  TRICORE: 17
  WASM: 13
  X86: 3
  XCORE: 7
```

Phew, that's a lot of architectures! Don't worry about memorizing all of them. You'll mostly be working with x86, x64, and maybe ARM.

We can just check for common architectures using this script:

check_capstone_archsupport.py

```python
import capstone

print("Testing architecture support:")

# Test each architecture by trying to create a disassembler
architectures_to_test = [
    ("x86-32 (Windows malware)", capstone.CS_ARCH_X86, capstone.CS_MODE_32
    ("x86-64 (Modern Windows/Linux)", capstone.CS_ARCH_X86, capstone.CS_MO
    ("ARM (Mobile malware)", capstone.CS_ARCH_ARM, capstone.CS_MODE_ARM),
    ("MIPS (IoT malware)", capstone.CS_ARCH_MIPS, capstone.CS_MODE_MIPS32)
]

for name, arch, mode in architectures_to_test:
    try:
        md = capstone.Cs(arch, mode)
        print(f"  {name}: Working")
    except Exception as e:
        print(f"  {name}: Failed ({e})")

print("\nArchitecture test complete.")
```

```
❯ python check_capstone_archsupport.py
Testing architecture support:
  x86-32 (Windows malware): Working
  x86-64 (Modern Windows/Linux): Working
  ARM (Mobile malware): Working
  MIPS (IoT malware): Working

Architecture test complete.
```

From the test script above, it say that Capstone support for most common arch that we already familiar with it.

While Capstone supports many architectures, malware analysts typically focus on these:

- x86/x64 (Intel/AMD): The vast majority of Windows, Linux, and macOS malware. This is where you'll spend most of your time.
- ARM: Mobile malware targeting Android and iOS, plus some IoT devices. ARM64 is becoming more common with modern phones and Apple's M1/M2 chips.
- MIPS: Often found in router and IoT malware, especially targeting home network devices.

Capstone Fundamentals

In this case, we'll use Capstone to disassemble a classic x86 function prologue, a small sequence of machine code instructions used to set up a stack frame at the beginning of a function. These instructions are commonly found in binaries compiled from C/C++ and form a solid foundation for learning how Capstone works.

The following raw bytes represent our target machine code:

```
CODE = b"\x55\x89\xE5\x83\xEC\x10"
```

This sequence corresponds to:

```
push ebp
mov ebp, esp
sub esp, 0x10
```

If you're not familiar with these instructions, don't worry. We're just using a common pattern here to demonstrate Capstone. We'll walk through how to disassemble this with Capstone and understand each instruction.

```
import capstone

# A simple function prologue (setup code at the start of functions)
CODE = b"\x55\x89\xE5\x83\xEC\x10"

# Initialize Capstone for x86, 32-bit mode
md = capstone.Cs(capstone.CS_ARCH_X86, capstone.CS_MODE_32)
md.detail = True  # Enable detailed output

# Disassemble the machine code starting at address 0x401000
for insn in md.disasm(CODE, 0x401000):
    print(f"Address: 0x{insn.address:x}")        # Memory address of instr
    print(f"Bytes: {insn.bytes.hex()}")          # Raw bytes of the instru
    print(f"Mnemonic: {insn.mnemonic}")          # Opcode (instruction typ
    print(f"Operands: {insn.op_str}")            # Operands used by the in
    print(f"Size: {insn.size} bytes")            # Length of instruction i
    print("-" * 40)
```

What each part tells us:

- **Address**: Helps track control flow and identify jump targets
- **Bytes**: Useful for pattern matching and
- **Mnemonic**: The action being performed (move data, jump, call function, etc.)
- **Operands**: What's being acted upon (registers, memory addresses, immediate values)
- **Size**: Important for calculating offsets and understanding instruction density

Output:

```
Address: 0x401000
Bytes: 55
Mnemonic: push
Operands: ebp
Size: 1 bytes
----------------------------------------
Address: 0x401001
Bytes: 89e5
```

```
Mnemonic: mov
Operands: ebp, esp
Size: 2 bytes
----------------------------------------
Address: 0x401003
Bytes: 83ec10
Mnemonic: sub
Operands: esp, 0x10
Size: 3 bytes
----------------------------------------
```

As expected, the Capstone disassembler successfully decoded the raw byte sequence into the exact assembly instructions explained above.

To further validate, you can use `nasm` to assemble the same instruction sequence and `ndisasm` to disassemble it back:

practice.asm

```
❯ cat practice.asm
; prologue.asm
BITS 32
global _start

_start:
    push ebp
    mov ebp, esp
    sub esp, 0x10
    nop
    nop
    nop
```

Assemble using `nasm`

```
nasm -f bin practice.asm -o practice.bin
```

Disassemble with `ndisasm`:

```
❯ ndisasm -b 32 practice.bin

00000000  55                push ebp
00000001  89E5              mov ebp,esp
00000003  83EC10            sub esp,byte +0x10
00000006  90                nop
00000007  90                nop
00000008  90                nop
```

As you can see, this confirms that the `CODE = b"\x55\x89\xE5\x83\xEC\x10"` indeed maps directly to the same assembly instructions we disassembled using Capstone.

Memory Addresses and Offsets

Understanding how addresses work is crucial for malware analysis. When you disassemble code, you need to provide a starting address. In this chapter, we use fake address:

```python
# Example: Analyzing code from different parts of a malware sample
import capstone

CODE = b"\xE8\x05\x00\x00\x00\x58\x83\xC0\x0A"  # call/pop/add pattern
md = capstone.Cs(capstone.CS_ARCH_X86, capstone.CS_MODE_32)

# Analyze as if this code starts at the typical Windows entry point
print("Analysis starting at Windows entry point:")
for insn in md.disasm(CODE, 0x401000):
    print(f"0x{insn.address:x}: {insn.mnemonic} {insn.op_str}")

print("\nSame code, different base address:")
# Same code, but as if it's from a different section
for insn in md.disasm(CODE, 0x10001000):
    print(f"0x{insn.address:x}: {insn.mnemonic} {insn.op_str}")
```

Output:

```
Analysis starting at Windows entry point:
0x401000: call 0x40100a
0x401005: pop eax
0x401006: add eax, 0xa

Same code, different base address:
0x10001000: call 0x1000100a
0x10001005: pop eax
0x10001006: add eax, 0xa
```

Analyzing PE Files with Capstone

Now that we've got the basics of Capstone under our belt, let's jump into something you'll actually use every day in malware analysis. We're talking about PE (Portable Executable) files, Windows standard executable format.

Understanding PE Structure

Before we dive into disassembly, it's helpful to understand the basic structure of a PE file:

- DOS Header: Legacy compatibility header
- PE Header: Signature and basic file information
- Optional Header: More detailed information about the executable
- Section Headers: Define the various sections in the file
- Sections: Actual code and data

Let's write a simple script to extract and disassemble the code section of a PE file:

Installing Required Dependencies

First, we need to install the `pefile` library, which will help us parse PE files:

```
pip install pefile
```

Basic PE Analysis Script

Let's break this down step by step. First thing we need to do is load the PE file and find that code section:

```
import pefile
import capstone

def analyze_pe_file(filepath):
    # Load the PE file
    pe = pefile.PE(filepath)

    # Find the code section (.text)
    code_section = None
    for section in pe.sections:
        if section.Name.decode().strip('\x00') == '.text':
            code_section = section
            break

    if not code_section:
        print("No .text section found!")
        return
```

Here we're loading the PE file with `pefile.PE()` and hunting for the `.text` section, which usually has our executable code. PE section names are fixed-length and stuffed with null bytes, so we need that `strip('\x00')` to clean them up for comparison.

Next, we extract the code data and calculate the actual memory addresses:

```
# Get the raw data of the code section
    code_data = code_section.get_data()

    # Get the virtual address where this section is loaded
    code_rva = code_section.VirtualAddress

    # Calculate the actual address in memory
    image_base = pe.OPTIONAL_HEADER.ImageBase
    code_address = image_base + code_rva

    print(f"Code section found at RVA 0x{code_rva:x}")
    print(f"Section size: {len(code_data)} bytes")
```

```
13      print(f"Starting address: 0x{code_address:x}")
14      print("-" * 50)
```

Once we've found the code section, we grab the raw bytes and figure out the actual memory address. The Virtual Address (RVA) is relative to the image base, so we just add the image base to get where the code would live in memory.

Now we set up Capstone based on the architecture and disassemble the code:

```
1   # Initialize Capstone disassembler for x86-64 or x86-32
2       if pe.FILE_HEADER.Machine == 0x8664:  # AMD64
3           md = capstone.Cs(capstone.CS_ARCH_X86, capstone.CS_MODE_64)
4       else:  # Assume x86-32 (works for most cases)
5           md = capstone.Cs(capstone.CS_ARCH_X86, capstone.CS_MODE_32)
6
7       # Disassemble the first 100 bytes as an example
8       print("Disassembly of first 100 bytes:")
9       count = 0
10      for insn in md.disasm(code_data, code_address):
11          if count >= 20:  # Limit to 20 instructions
12              break
13          print(f"0x{insn.address:x}:\t{insn.mnemonic}\t{insn.op_str}")
14          count += 1
15
16  analyze_pe_file("sample.exe")
```

We configure Capstone based on whether the PE file is 64-bit or 32-bit by checking the Machine field in the PE header. Then we iterate through the disassembled instructions, printing the address, mnemonic (instruction name), and operands. We limit the output to 20 instructions to keep the output manageable.

Basic PE Analysis Script

Here's the complete script for analyzing basic PE structure:

analyze.py

```
1   import pefile
2   import capstone
3
4   def analyze_pe_file(filepath):
```

```
    # Load the PE file
    pe = pefile.PE(filepath)

    # Find the code section (.text)
    code_section = None
    for section in pe.sections:
        if section.Name.decode().strip('\x00') == '.text':
            code_section = section
            break

    if not code_section:
        print("No .text section found!")
        return

    # Get the raw data of the code section
    code_data = code_section.get_data()

    # Get the virtual address where this section is loaded
    code_rva = code_section.VirtualAddress

    # Calculate the actual address in memory
    image_base = pe.OPTIONAL_HEADER.ImageBase
    code_address = image_base + code_rva

    print(f"Code section found at RVA 0x{code_rva:x}")
    print(f"Section size: {len(code_data)} bytes")
    print(f"Starting address: 0x{code_address:x}")
    print("-" * 50)

    # Initialize Capstone disassembler for x86-64 or x86-32
    if pe.FILE_HEADER.Machine == 0x8664:  # AMD64
        md = capstone.Cs(capstone.CS_ARCH_X86, capstone.CS_MODE_64)
    else:  # Assume x86-32
        md = capstone.Cs(capstone.CS_ARCH_X86, capstone.CS_MODE_32)

    # Disassemble the first 100 bytes as an example
    print("Disassembly of first 100 bytes:")
    count = 0
    for insn in md.disasm(code_data, code_address):
        if count >= 20:  # Limit to 20 instructions
            break
        print(f"0x{insn.address:x}:\t{insn.mnemonic}\t{insn.op_str}")
```

```
47          count += 1
48
49  analyze_pe_file("sample.exe")
```

Understanding the Output

When you run this script on a PE file, you might see output like this:

```
1   ❯ python analyze.py
2   Code section found at RVA 0x1000
3   Section size: 4096 bytes
4   Starting address: 0x401000
5   ----------------------------------------------------
6   Disassembly of first 100 bytes:
7   0x401000:       push    ebp
8   0x401001:       mov     ebp, esp
9   0x401003:       push    esi
10  0x401004:       push    edi
11  0x401005:       push    8
12  0x401007:       xor     edi, edi
13  0x401009:       push    edi
14  0x40100a:       push    0x402108
15  0x40100f:       call    dword ptr [0x402008]
16  0x401015:       mov     esi, eax
17  0x401017:       test    esi, esi
18  0x401019:       je      0x40104a
19  0x40101b:       push    0x40212c
20  0x401020:       push    esi
21  0x401021:       call    dword ptr [0x402000]
22  0x401027:       test    eax, eax
23  0x401029:       je      0x40103b
24  0x40102b:       push    dword ptr [ebp + 0x14]
25  0x40102e:       push    dword ptr [ebp + 0x10]
26  0x401031:       push    dword ptr [ebp + 0xc]
```

This output tells us several important things:

1. Where the code section is located in the file (RVA)
2. The size of the code section (handy for spotting anomalies)

3. The virtual address where it would be loaded (crucial for proper disassembly)
4. The actual disassembled instructions (the good stuff!)

Identifying Entry Points

In malware analysis, finding the entry point of a PE file is pretty crucial, it's where execution actually starts. Let me walk you through how to find it:

First, we'll locate the entry point in the PE file:

```
import pefile
import capstone

def find_entry_point(filepath):
    pe = pefile.PE(filepath)

    # Get the entry point RVA
    entry_point_rva = pe.OPTIONAL_HEADER.AddressOfEntryPoint

    # Calculate the actual address
    image_base = pe.OPTIONAL_HEADER.ImageBase
    entry_point_address = image_base + entry_point_rva

    print(f"Entry point RVA: 0x{entry_point_rva:x}")
    print(f"Entry point address: 0x{entry_point_address:x}")
```

The entry point is stored as an RVA (Relative Virtual Address) in the Optional Header. We convert it to an absolute address by adding the image base, which gives us the actual memory location where execution begins.

Next, we'll identify which section contains the entry point and extract the relevant code:

```
# Find which section contains the entry point
    for section in pe.sections:
        section_start = section.VirtualAddress
        section_end = section_start + section.Misc_VirtualSize

        if section_start <= entry_point_rva < section_end:
            print(f"Entry point is in section: {section.Name.decode().stri

            # Calculate offset within the section
```

```
            offset_in_section = entry_point_rva - section_start
            code_data = section.get_data()

            # Get a chunk of code around the entry point
            start_offset = max(0, offset_in_section - 20)
            end_offset = min(len(code_data), offset_in_section + 50)
            entry_code = code_data[start_offset:end_offset]
```

We go through all the sections to find where the entry point lives, this is usually in the.text section but not always! Then we calculate the offset in that section and pull out a chunk of code around the entry point (20 bytes before and 50 after) so we can see what's happening around the start of execution. I like to grab a bit before and after because context matters.

Finally, we disassemble the code around the entry point:

```
# Disassemble the entry point code
            if pe.FILE_HEADER.Machine == 0x8664:  # AMD64
                md = capstone.Cs(capstone.CS_ARCH_X86, capstone.CS_MODE_64
            else:  # Assume x86-32
                md = capstone.Cs(capstone.CS_ARCH_X86, capstone.CS_MODE_32

            print("\nDisassembly around entry point:")
            for insn in md.disasm(entry_code, entry_point_address - 20):
                print(f"0x{insn.address:x}:\t{insn.mnemonic}\t{insn.op_str
            break

find_entry_point("sample.exe")
```

We set up Capstone based on the architecture and disassemble the code chunk we extracted, displaying the instructions around the entry point. This gives us insight into what the program does when it starts executing.

Complete Entry Point Script

Here's the complete script for identifying and analyzing PE file entry points:

find_entry.py

```
import pefile
import capstone

```

```python
def find_entry_point(filepath):
    pe = pefile.PE(filepath)

    # Get the entry point RVA
    entry_point_rva = pe.OPTIONAL_HEADER.AddressOfEntryPoint

    # Calculate the actual address
    image_base = pe.OPTIONAL_HEADER.ImageBase
    entry_point_address = image_base + entry_point_rva

    print(f"Entry point RVA: 0x{entry_point_rva:x}")
    print(f"Entry point address: 0x{entry_point_address:x}")

    # Find which section contains the entry point
    for section in pe.sections:
        section_start = section.VirtualAddress
        section_end = section_start + section.Misc_VirtualSize

        if section_start <= entry_point_rva < section_end:
            print(f"Entry point is in section: {section.Name.decode().stri

            # Calculate offset within the section
            offset_in_section = entry_point_rva - section_start
            code_data = section.get_data()

            # Get a chunk of code around the entry point
            start_offset = max(0, offset_in_section - 20)
            end_offset = min(len(code_data), offset_in_section + 50)
            entry_code = code_data[start_offset:end_offset]

            # Disassemble the entry point code
            if pe.FILE_HEADER.Machine == 0x8664:  # AMD64
                md = capstone.Cs(capstone.CS_ARCH_X86, capstone.CS_MODE_64
            else:  # Assume x86-32
                md = capstone.Cs(capstone.CS_ARCH_X86, capstone.CS_MODE_32

            print("\nDisassembly around entry point:")
            for insn in md.disasm(entry_code, entry_point_address - 20):
                print(f"0x{insn.address:x}:\t{insn.mnemonic}\t{insn.op_str
            break

find_entry_point("sample.exe")
```

Understanding the Output

When you run this script on a PE file, you might see output like this:

```
❯ python find_entry.py
Entry point RVA: 0x12b5
Entry point address: 0x4012b5
Entry point is in section: .text

Disassembly around entry point:
0x4012a1:       call    0x40174c
0x4012a6:       push    esi
0x4012a7:       call    0x401d95
0x4012ac:       push    dword ptr [ebp - 0x20]
0x4012af:       call    0x401d9b
0x4012b4:       int3
0x4012b5:       call    0x401680
0x4012ba:       jmp     0x401139
0x4012bf:       push    ebp
0x4012c0:       mov     ebp, esp
0x4012c2:       push    0
0x4012c4:       call    dword ptr [0x402034]
0x4012ca:       push    dword ptr [ebp + 8]
0x4012cd:       call    dword ptr [0x402038]
0x4012d3:       push    0xc0000409
0x4012d8:       call    dword ptr [0x402030]
0x4012de:       push    eax
0x4012df:       call    dword ptr [0x40202c]
0x4012e5:       pop     ebp
0x4012e6:       ret
```

This output tells us several important things:

1. Entry Point RVA — The entry point relative virtual address (0x12b5) shows where program execution begins, relative to the image base.
2. Entry Point Address — The absolute virtual memory address (0x4012b5) indicates the exact location in memory where execution starts when the binary is loaded.
3. Section Mapping — The entry point lies inside the.text section, which is typical because this section contains the executable code.

4. Disassembly Context — The surrounding instructions (calls, pushes, jumps, etc.) give insight into what happens at the program's startup.

Conclusion

Capstone is honestly a game-changer for static analysis on malware samples. Whether you're doing basic disassembly or deep PE file analysis, Capstone's Python API makes it easy to automate the boring stuff and focus on the interesting parts of malware analysis.

As you continue your journey in malware analysis, remember that tools like Capstone are just the beginning. The real art comes from understanding what the disassembled code is actually doing and how it fits into the bigger picture of the malware's behavior.

Start with simple examples like we've covered here, and don't rush into the deep end.Take your time, build your skills gradually, and before you know it, you'll be spotting patterns, detecting obfuscation techniques, and understanding exactly how those malware samples work under the hood.



(https://buymeacoffee.com/petruknisme)

Like the article and want to support me? you can support me here (https://buymeacoffee.com/petruknisme)

Originally published on Medium (https://medium.com/@petruknisme/capstone-for-malware-analysis-a-beginners-guide-fe5f01daae6d).

Malware Analysis # Security # Tutorial
