Remote debugging attachment protocol

Release 3.14.0rc1

Guido van Rossum and the Python development team

July 22, 2025

Python Software Foundation
Email: docs@python.org

Contents

1 Locating the PyRuntime structure 2

2 Reading _Py_DebugOffsets 4

3 Locating the interpreter and thread state 6

4 Writing control information 7

5 Summary 8

This section describes the low-level protocol that enables external tools to inject and execute a Python script within a running CPython process.

This mechanism forms the basis of the `sys.remote_exec()` function, which instructs a remote Python process to execute a `.py` file. However, this section does not document the usage of that function. Instead, it provides a detailed explanation of the underlying protocol, which takes as input the `pid` of a target Python process and the path to a Python source file to be executed. This information supports independent reimplementation of the protocol, regardless of programming language.

> **Warning**
>
> The execution of the injected script depends on the interpreter reaching a safe evaluation point. As a result, execution may be delayed depending on the runtime state of the target process.

Once injected, the script is executed by the interpreter within the target process the next time a safe evaluation point is reached. This approach enables remote execution capabilities without modifying the behavior or structure of the running Python application.

Subsequent sections provide a step-by-step description of the protocol, including techniques for locating interpreter structures in memory, safely accessing internal fields, and triggering code execution. Platform-specific variations are noted where applicable, and example implementations are included to clarify each operation.

1 Locating the PyRuntime structure

CPython places the `PyRuntime` structure in a dedicated binary section to help external tools find it at runtime. The name and format of this section vary by platform. For example, `.PyRuntime` is used on ELF systems, and `__DATA,__PyRuntime` is used on macOS. Tools can find the offset of this structure by examining the binary on disk.

The `PyRuntime` structure contains CPython's global interpreter state and provides access to other internal data, including the list of interpreters, thread states, and debugger support fields.

To work with a remote Python process, a debugger must first find the memory address of the `PyRuntime` structure in the target process. This address can't be hardcoded or calculated from a symbol name, because it depends on where the operating system loaded the binary.

The method for finding `PyRuntime` depends on the platform, but the steps are the same in general:

1. Find the base address where the Python binary or shared library was loaded in the target process.
2. Use the on-disk binary to locate the offset of the `.PyRuntime` section.
3. Add the section offset to the base address to compute the address in memory.

The sections below explain how to do this on each supported platform and include example code.

Linux (ELF)

To find the `PyRuntime` structure on Linux:

1. Read the process's memory map (for example, `/proc/<pid>/maps`) to find the address where the Python executable or `libpython` was loaded.
2. Parse the ELF section headers in the binary to get the offset of the `.PyRuntime` section.
3. Add that offset to the base address from step 1 to get the memory address of `PyRuntime`.

The following is an example implementation:

```
def find_py_runtime_linux(pid: int) -> int:
    # Step 1: Try to find the Python executable in memory
    binary_path, base_address = find_mapped_binary(
        pid, name_contains="python"
    )

    # Step 2: Fallback to shared library if executable is not found
    if binary_path is None:
        binary_path, base_address = find_mapped_binary(
            pid, name_contains="libpython"
        )

    # Step 3: Parse ELF headers to get .PyRuntime section offset
    section_offset = parse_elf_section_offset(
        binary_path, ".PyRuntime"
    )

    # Step 4: Compute PyRuntime address in memory
    return base_address + section_offset
```

On Linux systems, there are two main approaches to read memory from another process. The first is through the `/proc` filesystem, specifically by reading from `/proc/[pid]/mem` which provides direct access to the process's memory. This requires appropriate permissions - either being the same user as the target process or having root access. The second approach is using the `process_vm_readv()` system call which provides a more efficient way to copy memory between processes. While ptrace's `PTRACE_PEEKTEXT` operation can also be used to read memory, it is significantly slower as it only reads one word at a time and requires multiple context switches between the tracer and tracee processes.

For parsing ELF sections, the process involves reading and interpreting the ELF file format structures from the binary file on disk. The ELF header contains a pointer to the section header table. Each section header contains metadata about a section including its name (stored in a separate string table), offset, and size. To find a specific section like .PyRuntime, you need to walk through these headers and match the section name. The section header then provides the offset where that section exists in the file, which can be used to calculate its runtime address when the binary is loaded into memory.

You can read more about the ELF file format in the ELF specification.

macOS (Mach-O)

To find the `PyRuntime` structure on macOS:

1. Call `task_for_pid()` to get the `mach_port_t` task port for the target process. This handle is needed to read memory using APIs like `mach_vm_read_overwrite` and `mach_vm_region`.
2. Scan the memory regions to find the one containing the Python executable or `libpython`.
3. Load the binary file from disk and parse the Mach-O headers to find the section named `PyRuntime` in the `__DATA` segment. On macOS, symbol names are automatically prefixed with an underscore, so the `PyRuntime` symbol appears as `_PyRuntime` in the symbol table, but the section name is not affected.

The following is an example implementation:

```
def find_py_runtime_macos(pid: int) -> int:
    # Step 1: Get access to the process's memory
    handle = get_memory_access_handle(pid)

    # Step 2: Try to find the Python executable in memory
    binary_path, base_address = find_mapped_binary(
        handle, name_contains="python"
    )

    # Step 3: Fallback to libpython if the executable is not found
    if binary_path is None:
        binary_path, base_address = find_mapped_binary(
            handle, name_contains="libpython"
        )

    # Step 4: Parse Mach-O headers to get __DATA,__PyRuntime section offset
    section_offset = parse_macho_section_offset(
        binary_path, "__DATA", "__PyRuntime"
    )

    # Step 5: Compute the PyRuntime address in memory
    return base_address + section_offset
```

On macOS, accessing another process's memory requires using Mach-O specific APIs and file formats. The first step is obtaining a `task_port` handle via `task_for_pid()`, which provides access to the target process's memory space. This handle enables memory operations through APIs like `mach_vm_read_overwrite()`.

The process memory can be examined using `mach_vm_region()` to scan through the virtual memory space, while `proc_regionfilename()` helps identify which binary files are loaded at each memory region. When the Python binary or library is found, its Mach-O headers need to be parsed to locate the `PyRuntime` structure.

The Mach-O format organizes code and data into segments and sections. The `PyRuntime` structure lives in a section named `__PyRuntime` within the `__DATA` segment. The actual runtime address calculation involves finding the `__TEXT` segment which serves as the binary's base address, then locating the `__DATA` segment containing our target section. The final address is computed by combining the base address with the appropriate section offsets from the Mach-O headers.

Note that accessing another process's memory on macOS typically requires elevated privileges - either root access or special security entitlements granted to the debugging process.

Windows (PE)

To find the `PyRuntime` structure on Windows:

1. Use the ToolHelp API to enumerate all modules loaded in the target process. This is done using functions such as CreateToolhelp32Snapshot, Module32First, and Module32Next.
2. Identify the module corresponding to `python.exe` or `python`*`XY`*`.dll`, where `X` and `Y` are the major and minor version numbers of the Python version, and record its base address.
3. Locate the `PyRuntim` section. Due to the PE format's 8-character limit on section names (defined as `IMAGE_SIZEOF_SHORT_NAME`), the original name `PyRuntime` is truncated. This section contains the `PyRuntime` structure.
4. Retrieve the section's relative virtual address (RVA) and add it to the base address of the module.

The following is an example implementation:

```
def find_py_runtime_windows(pid: int) -> int:
    # Step 1: Try to find the Python executable in memory
    binary_path, base_address = find_loaded_module(
        pid, name_contains="python"
    )

    # Step 2: Fallback to shared pythonXY.dll if the executable is not
    # found
    if binary_path is None:
        binary_path, base_address = find_loaded_module(
            pid, name_contains="python3"
        )

    # Step 3: Parse PE section headers to get the RVA of the PyRuntime
    # section. The section name appears as "PyRuntim" due to the
    # 8-character limit defined by the PE format (IMAGE_SIZEOF_SHORT_NAME).
    section_rva = parse_pe_section_offset(binary_path, "PyRuntim")

    # Step 4: Compute PyRuntime address in memory
    return base_address + section_rva
```

On Windows, accessing another process's memory requires using the Windows API functions like `CreateToolhelp32Snapshot()` and `Module32First()`/`Module32Next()` to enumerate loaded modules. The `OpenProcess()` function provides a handle to access the target process's memory space, enabling memory operations through `ReadProcessMemory()`.

The process memory can be examined by enumerating loaded modules to find the Python binary or DLL. When found, its PE headers need to be parsed to locate the `PyRuntime` structure.

The PE format organizes code and data into sections. The `PyRuntime` structure lives in a section named "PyRuntim" (truncated from "PyRuntime" due to PE's 8-character name limit). The actual runtime address calculation involves finding the module's base address from the module entry, then locating our target section in the PE headers. The final address is computed by combining the base address with the section's virtual address from the PE section headers.

Note that accessing another process's memory on Windows typically requires appropriate privileges - either administrative access or the `SeDebugPrivilege` privilege granted to the debugging process.

2 Reading _Py_DebugOffsets

Once the address of the `PyRuntime` structure has been determined, the next step is to read the `_Py_DebugOffsets` structure located at the beginning of the `PyRuntime` block.

This structure provides version-specific field offsets that are needed to safely read interpreter and thread state memory. These offsets vary between CPython versions and must be checked before use to ensure they are compatible.

To read and check the debug offsets, follow these steps:

1. Read memory from the target process starting at the `PyRuntime` address, covering the same number of bytes as the `_Py_DebugOffsets` structure. This structure is located at the very start of the `PyRuntime` memory block. Its layout is defined in CPython's internal headers and stays the same within a given minor version, but may change in major versions.
2. Check that the structure contains valid data:
   - The `cookie` field must match the expected debug marker.
   - The `version` field must match the version of the Python interpreter used by the debugger.
   - If either the debugger or the target process is using a pre-release version (for example, an alpha, beta, or release candidate), the versions must match exactly.
   - The `free_threaded` field must have the same value in both the debugger and the target process.
3. If the structure is valid, the offsets it contains can be used to locate fields in memory. If any check fails, the debugger should stop the operation to avoid reading memory in the wrong format.

The following is an example implementation that reads and checks `_Py_DebugOffsets`:

```
def read_debug_offsets(pid: int, py_runtime_addr: int) -> DebugOffsets:
    # Step 1: Read memory from the target process at the PyRuntime address
    data = read_process_memory(
        pid, address=py_runtime_addr, size=DEBUG_OFFSETS_SIZE
    )

    # Step 2: Deserialize the raw bytes into a _Py_DebugOffsets structure
    debug_offsets = parse_debug_offsets(data)

    # Step 3: Validate the contents of the structure
    if debug_offsets.cookie != EXPECTED_COOKIE:
        raise RuntimeError("Invalid or missing debug cookie")
    if debug_offsets.version != LOCAL_PYTHON_VERSION:
        raise RuntimeError(
            "Mismatch between caller and target Python versions"
        )
    if debug_offsets.free_threaded != LOCAL_FREE_THREADED:
        raise RuntimeError("Mismatch in free-threaded configuration")

    return debug_offsets
```

Warning

Process suspension recommended

To avoid race conditions and ensure memory consistency, it is strongly recommended that the target process be suspended before performing any operations that read or write internal interpreter state. The Python runtime may concurrently mutate interpreter data structures—such as creating or destroying threads—during normal execution. This can result in invalid memory reads or writes.

A debugger may suspend execution by attaching to the process with `ptrace` or by sending a `SIGSTOP` signal. Execution should only be resumed after debugger-side memory operations are complete.

Note

Some tools, such as profilers or sampling-based debuggers, may operate on a running process without suspension. In such cases, tools must be explicitly designed to handle partially updated or inconsistent memory. For most debugger implementations, suspending the process remains the safest and most robust approach.

3 Locating the interpreter and thread state

Before code can be injected and executed in a remote Python process, the debugger must choose a thread in which to schedule execution. This is necessary because the control fields used to perform remote code injection are located in the `_PyRemoteDebuggerSupport` structure, which is embedded in a `PyThreadState` object. These fields are modified by the debugger to request execution of injected scripts.

The `PyThreadState` structure represents a thread running inside a Python interpreter. It maintains the thread's evaluation context and contains the fields required for debugger coordination. Locating a valid `PyThreadState` is therefore a key prerequisite for triggering execution remotely.

A thread is typically selected based on its role or ID. In most cases, the main thread is used, but some tools may target a specific thread by its native thread ID. Once the target thread is chosen, the debugger must locate both the interpreter and the associated thread state structures in memory.

The relevant internal structures are defined as follows:

- `PyInterpreterState` represents an isolated Python interpreter instance. Each interpreter maintains its own set of imported modules, built-in state, and thread state list. Although most Python applications use a single interpreter, CPython supports multiple interpreters in the same process.
- `PyThreadState` represents a thread running within an interpreter. It contains execution state and the control fields used by the debugger.

To locate a thread:

1. Use the offset `runtime_state.interpreters_head` to obtain the address of the first interpreter in the `PyRuntime` structure. This is the entry point to the linked list of active interpreters.
2. Use the offset `interpreter_state.threads_main` to access the main thread state associated with the selected interpreter. This is typically the most reliable thread to target.

3. Optionally, use the offset `interpreter_state.threads_head` to iterate through the linked list of all thread states. Each `PyThreadState` structure contains a `native_thread_id` field, which may be compared to a target thread ID to find a specific thread.

1. Once a valid `PyThreadState` has been found, its address can be used in later steps of the protocol, such as writing debugger control fields and scheduling execution.

The following is an example implementation that locates the main thread state:

```
def find_main_thread_state(
    pid: int, py_runtime_addr: int, debug_offsets: DebugOffsets,
) -> int:
    # Step 1: Read interpreters_head from PyRuntime
    interp_head_ptr = (
        py_runtime_addr + debug_offsets.runtime_state.interpreters_head
    )
    interp_addr = read_pointer(pid, interp_head_ptr)
    if interp_addr == 0:
        raise RuntimeError("No interpreter found in the target process")

    # Step 2: Read the threads_main pointer from the interpreter
    threads_main_ptr = (
        interp_addr + debug_offsets.interpreter_state.threads_main
    )
    thread_state_addr = read_pointer(pid, threads_main_ptr)
    if thread_state_addr == 0:
        raise RuntimeError("Main thread state is not available")

    return thread_state_addr
```

The following example demonstrates how to locate a thread by its native thread ID:

```
def find_thread_by_id(
    pid: int,
    interp_addr: int,
    debug_offsets: DebugOffsets,
    target_tid: int,
) -> int:
    # Start at threads_head and walk the linked list
    thread_ptr = read_pointer(
        pid,
        interp_addr + debug_offsets.interpreter_state.threads_head
    )

    while thread_ptr:
        native_tid_ptr = (
            thread_ptr + debug_offsets.thread_state.native_thread_id
        )
        native_tid = read_int(pid, native_tid_ptr)
        if native_tid == target_tid:
            return thread_ptr
        thread_ptr = read_pointer(
            pid,
            thread_ptr + debug_offsets.thread_state.next
        )

    raise RuntimeError("Thread with the given ID was not found")
```

Once a valid thread state has been located, the debugger can proceed with modifying its control fields and scheduling execution, as described in the next section.

4 Writing control information

Once a valid `PyThreadState` structure has been identified, the debugger may modify control fields within it to schedule the execution of a specified Python script. These control fields are checked periodically by the interpreter, and when set correctly, they trigger the execution of remote code at a safe point in the evaluation loop.

Each `PyThreadState` contains a `_PyRemoteDebuggerSupport` structure used for communication between the debugger and the interpreter. The locations of its fields are defined by the `_Py_DebugOffsets` structure and include the following:

- **`debugger_script_path`: A fixed-size buffer that holds the full path to a** Python source file (`.py`). This file must be accessible and readable by the target process when execution is triggered.
- **`debugger_pending_call`: An integer flag. Setting this to `1` tells the** interpreter that a script is ready to be executed.
- **`eval_breaker`: A field checked by the interpreter during execution.** Setting bit 5 (`_PY_EVAL_PLEASE_STOP_BIT`, value `1U << 5`) in this field causes the interpreter to pause and check for debugger activity.

To complete the injection, the debugger must perform the following steps:

1. Write the full script path into the `debugger_script_path` buffer.
2. Set `debugger_pending_call` to `1`.
3. Read the current value of `eval_breaker`, set bit 5 (`_PY_EVAL_PLEASE_STOP_BIT`), and write the updated value back. This signals the interpreter to check for debugger activity.

The following is an example implementation:

```
def inject_script(
    pid: int,
    thread_state_addr: int,
    debug_offsets: DebugOffsets,
    script_path: str
) -> None:
    # Compute the base offset of _PyRemoteDebuggerSupport
    support_base = (
        thread_state_addr +
        debug_offsets.debugger_support.remote_debugger_support
    )

    # Step 1: Write the script path into debugger_script_path
    script_path_ptr = (
        support_base +
        debug_offsets.debugger_support.debugger_script_path
    )
    write_string(pid, script_path_ptr, script_path)

    # Step 2: Set debugger_pending_call to 1
    pending_ptr = (
        support_base +
        debug_offsets.debugger_support.debugger_pending_call
    )
    write_int(pid, pending_ptr, 1)

    # Step 3: Set _PY_EVAL_PLEASE_STOP_BIT (bit 5, value 1 << 5) in
    # eval_breaker
    eval_breaker_ptr = (
        thread_state_addr +
        debug_offsets.debugger_support.eval_breaker
    )
    breaker = read_int(pid, eval_breaker_ptr)
    breaker |= (1 << 5)
    write_int(pid, eval_breaker_ptr, breaker)
```

Once these fields are set, the debugger may resume the process (if it was suspended). The interpreter will process the request at the next safe evaluation point, load the script from disk, and execute it.

It is the responsibility of the debugger to ensure that the script file remains present and accessible to the target process during execution.

> **Note**
>
> Script execution is asynchronous. The script file cannot be deleted immediately after injection. The debugger should wait until the injected script has produced an observable effect before removing the file. This effect depends on what the script is designed to do. For example, a debugger might wait until the remote process connects back to a socket before removing the script. Once such an effect is observed, it is safe to assume the file is no longer needed.

5 Summary

To inject and execute a Python script in a remote process:

1. Locate the `PyRuntime` structure in the target process's memory.
2. Read and validate the `_Py_DebugOffsets` structure at the beginning of `PyRuntime`.

3. Use the offsets to locate a valid `PyThreadState`.
4. Write the path to a Python script into `debugger_script_path`.
5. Set the `debugger_pending_call` flag to `1`.
6. Set `_PY_EVAL_PLEASE_STOP_BIT` in the `eval_breaker` field.
7. Resume the process (if suspended). The script will execute at the next safe evaluation point.