远程调试附加协议

此协议使得外部工具能够附加到正在运行的 CPython 进程并远程执行 Python 代码。

大多数平台上需要提升的权限才能附加到另一个 Python 进程。

权限需求

在大多数平台上，需要提升的权限才能附加到一个正在运行的 Python 进程以远程调试。具体的需求和故障排除方法取决于您的操作系统：

Linux

追踪进程必须拥有 `CAP_SYS_PTRACE` 能力或等价的权限。你只能追踪你拥有且可发送信号的进程。如果该进程正被追踪或者在 set-user-ID 或 set-group-ID 下运行，追踪可能失败。Yama 等安全模块可能会进一步限制追踪。

若要暂时放松 ptrace 限制（直到重启），可以运行：

```
echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope
```

备注： 禁用 `ptrace_scope` 会降低系统安全强度，因而只应在受信任的环境中进行。

若在容器中运行，使用 `--cap-add=SYS_PTRACE` 或者 `--privileged`，并按需以 root 身份运行。

尝试用提升的权限重新运行命令：

```
sudo -E !!
```

macOS

要附加到另一个进程，您通常需要通过使用 `sudo` 或以 root 身份运行，从而以提升的权限运行调试工具。

即使您拥有要附加到的进程，在 macOS 上调试仍可能因系统安全限制被阻止，除非使用 root 权限运行调试器。

Windows

要附加到另一个进程，您通常需要以管理员权限运行调试工具：以管理员身份运行命令提示符或者终端。

使用管理员权限时，除非启用 `SeDebugPrivilege` 权限，否则有的进程仍可能无法被访问。

要解决文件或文件夹访问的问题，请调整安全权限：

1. 右键文件或文件夹并选择 **属性**。
2. 在 **安全** 选项卡中查看有访问权限的用户和用户组。
3. 点击 **编辑** 以调整权限。
4. 选择您的用户账户。
5. 在 **权限** 中，按需勾选 **读取** 或者 **完全控制**。
6. 在点击 **应用** 后点击 **确定**。

> **备注：** 在继续前，请确保您已满足所有 权限需求。

本节描述了低级协议，该协议使外部工具能够在运行的CPython进程中注入和执行Python脚本。

该机制构成了 `sys.remote_exec()` 函数的基础，该函数用于指示远程Python进程执行指定的 `.py` 文件。但本节并不记录该函数的具体用法，而是详细阐述其底层协议的工作原理——该协议以目标Python进程的 `pid` 和待执行的Python源文件路径作为输入。这些信息支持协议的独立重新实现，且不受编程语言限制。

> **警告：** 注入脚本的执行依赖于解释器到达安全的求值点。因此，实际执行时机可能会因目标进程的运行时状态而产生延迟。

一旦注入，脚本将在解释器下一次达到安全求值点时在目标进程中执行。这种方法能够在不修改运行中Python应用的行为或结构的情况下实现远程执行功能。

后续各节提供了该协议的逐步描述，包括定位内存中解释器结构的技术、安全访问内部字段以及触发代码执行的方法。适用的情况下会注明平台特定的变体，并包含示例实现以澄清每个操作。

定位PyRuntime结构

CPython将 `PyRuntime` 结构放置在一个专用的二进制节中，以帮助外部工具在运行时找到它。该节的名称和格式因平台而异。例如，在ELF系统上使用 `.PyRuntime`，在macOS上使用 `__DATA,__PyRuntime`。工具可以通过检查磁盘上的二进制文件来找到该结构的偏移量。

`PyRuntime` 结构包含 CPython 的全局解释器状态，并提供对其他内部数据的访问，包括解释器列表、线程状态和调试器支持字段。

要处理远程Python进程，调试器首先必须在目标进程中找到 `PyRuntime` 结构的内存地址。这个地址不能硬编码或通过符号名计算，因为它取决于操作系统加载二进制文件的位置。

查找 `PyRuntime` 的方法取决于平台，但一般步骤是相同的：

1. 找到Python二进制文件或共享库在目标进程中加载的基址。
2. 使用磁盘上的二进制文件定位 `.PyRuntime` 段的偏移。
3. 将段偏移加到基址上，计算出内存中的地址。

以下部分将说明在每个受支持平台上如何进行此操作，并包括示例代码。

Linux (ELF)

在Linux上查找 `PyRuntime` 结构：

1. 读取进程的内存映射（例如，`/proc/<pid>/maps`）以找到Python可执行文件或 `libpython` 加载的地址。
2. 解析二进制文件中的ELF段头，获取 `.PyRuntime` 段的偏移。
3. 将此偏移加到步骤1中的基址上，得到 `PyRuntime` 的内存地址。

以下是一个示例实现：

```
def find_py_runtime_linux(pid: int) -> int:
    # 步骤 1：尝试在内存中找到 Python 可执行文件
    binary_path, base_address = find_mapped_binary(
        pid, name_contains="python"
    )

    # 步骤2：如果找不到可执行文件，则回退到共享库
    if binary_path is None:
        binary_path, base_address = find_mapped_binary(
            pid, name_contains="libpython"
        )

    # 步骤3：解析ELF头以获取.PyRuntime节的偏移量
    section_offset = parse_elf_section_offset(
        binary_path, ".PyRuntime"
    )

    # 步骤4：计算内存中的PyRuntime地址
    return base_address + section_offset
```

在Linux系统上，有两种主要方法读取另一个进程的内存。第一种是通过 `/proc` 文件系统，具体来说是通过读取 `/proc/[pid]/mem` ，它提供了对进程内存的直接访问。这需要适当的权限——要么是与目标进程相同的用户，要么拥有root权限。第二种方法是使用 `process_vm_readv()` 系统调用，它提供了在进程间复制内存的更高效方式。虽然ptrace的 `PTRACE_PEEKTEXT` 操作也可以用来读取内存，但它显著较慢，因为它一次只读取一个字，并且需要在跟踪器和被跟踪进程之间进行多次上下文切换。

为了解析ELF节，过程包括从磁盘上的二进制文件中读取和解释ELF文件格式结构。ELF头部包含一个指向节头表的指针。每个节头包含有关节的元数据，包括其名称（存储在单独的字符串表中）、偏移量和大小。要查找特定节（如.PyRuntime），需要遍历这些头部并匹配节名称。节头然后提供该节在文件中存在的偏移量，这可以用来计算二进制文件加载到内存时的运行时地址。

你可以在`ELF规范 <https://en.wikipedia.org/wiki/Executable_and_Linkable_Format>`_中了解更多关于ELF文件格式的信息。

macOS (Mach-O)

在macOS上查找 `PyRuntime` 结构：

1. 调用 `task_for_pid()` 以获取目标进程的 `mach_port_t` 任务端口。此句柄用于通过 `mach_vm_read_overwrite` 和 `mach_vm_region` 等API读取内存。
2. 扫描内存区域，找到包含Python可执行文件或 `libpython` 的区域。
3. 从磁盘加载二进制文件并解析Mach-O头部，以在 `__DATA` 段中找到名为 `PyRuntime` 的节。在macOS上，符号名称自动以一个下划线为前缀，因此 `PyRuntime` 符号在符号表中显示为 ``_PyRuntime``，但节名称不受影响。

以下是一个示例实现：

```
def find_py_runtime_macos(pid: int) -> int:
    # 步骤 1：访问进程的内存
    handle = get_memory_access_handle(pid)

    # 步骤 2：尝试在内存中找到 Python 可执行文件
    binary_path, base_address = find_mapped_binary(
        handle, name_contains="python"
    )

    # 步骤3：如果找不到可执行文件，则回退到libpython
    if binary_path is None:
        binary_path, base_address = find_mapped_binary(
            handle, name_contains="libpython"
        )

    # 步骤4：解析Mach-O头以获取__DATA,__PyRuntime段的偏移量
    section_offset = parse_macho_section_offset(
        binary_path, "__DATA", "__PyRuntime"
    )

    # 步骤5：计算内存中的PyRuntime地址
    return base_address + section_offset
```

在macOS上，访问另一个进程的内存需要使用Mach-O特定的API和文件格式。第一步是通过 `task_for_pid()` 获取 `task_port` 句柄，这提供了对目标进程内存空间的访问。此句柄通过 `mach_vm_read_overwrite()` 等API启用内存操作。

可以使用 `mach_vm_region()` 检查进程内存，以扫描虚拟内存空间，而 `proc_regionfilename()` 帮助识别每个内存区域加载了哪些二进制文件。当找到 Python 二进制文件或库时，需要解析其 Mach-O 头部以定位 `PyRuntime` 结构。

Mach-O 格式将代码和数据组织到段和节中。`PyRuntime` 结构位于 `__DATA` 段中的名为 `__PyRuntime` 的节内。实际的运行时地址计算涉及找到作为二进制文件基址的 `__TEXT` 段，然后定位包含目标节的 `__DATA` 段。最终地址是通过将基址与 Mach-O 头部中的适当节偏移量组合来计算的。

请注意，在 macOS 上访问另一个进程的内存通常需要提升权限——要么是 root 访问权限，要么是授予调试进程的特殊安全权限。

Windows (PE)

在 Windows 上查找 `PyRuntime` 结构：

1. 使用 ToolHelp API 枚举目标进程中加载的所有模块。这通过使用如 CreateToolhelp32Snapshot, Module32First, 和 Module32Next 等函数来完成。
2. 识别对应于 `python.exe` 或 `pythonXY.dll` 的模块，其中 `X` 和 `Y` 是 Python 版本的主次版本号，并记录其基址。
3. 定位 `PyRuntim` 节。由于 PE 格式对节名称有 8 个字符的限制（定义为 `IMAGE_SIZEOF_SHORT_NAME`），原始名称 `PyRuntime` 被截断。此节包含 `PyRuntime` 结构。
4. 检索节的相对虚拟地址（RVA），并将其添加到模块的基址。

以下是一个示例实现：

```
def find_py_runtime_windows(pid: int) -> int:
    # 步骤 1：尝试在内存中找到 Python 可执行文件
    binary_path, base_address = find_loaded_module(
        pid, name_contains="python"
    )

    # 步骤2：如果可执行文件未找到，则回退到共享的pythonXY.dll
    #
    if binary_path is None:
        binary_path, base_address = find_loaded_module(
            pid, name_contains="python3"
        )

    # 步骤 3：解析 PE 节头以获取 PyRuntime 节的相对虚拟地址（RVA）。
    # 由于 PE 格式（IMAGE_SIZEOF_SHORT_NAME）规定的 8 字符限制，
    # 该节的名称显示为“PyRuntim”。.
    section_rva = parse_pe_section_offset(binary_path, "PyRuntim")

    # 步骤4：计算内存中的PyRuntime地址
    return base_address + section_rva
```

在Windows上，访问另一个进程的内存需要使用Windows API函数，如 `CreateToolhelp32Snapshot()` 和 `Module32First()/Module32Next()` 来枚举已加载的模块。`OpenProcess()` 函数提供了一个句柄，用于访问目标进程的内存空间，通过 `ReadProcessMemory()` 实现内存操作。

可以通过枚举已加载的模块来检查进程内存，以找到Python二进制文件或DLL。找到后，需要解析其PE头以定位 `PyRuntime` 结构。

PE 格式将代码和数据组织到节中。 `PyRuntime` 结构位于名为 "PyRuntim" 的节中（由于 PE 的 8 字符名称限制，从 "PyRuntime" 截断）。 实际的运行时地址计算涉及从模块入口找到模块的基址，然后在 PE 头中定位目标节。最终地址是通过将基址与PE节头中的节的虚拟地址组合来计算的。

请注意，在Windows上访问另一个进程的内存通常需要适当的权限——要么是管理员访问权限，要么是授予调试进程的 `SeDebugPrivilege` 权限。

读取_Py_DebugOffsets

一旦确定了 `PyRuntime` 结构的地址，下一步就是读取位于 `PyRuntime` 块开头的 `_Py_DebugOffsets` 结构。

该结构提供了特定版本的字段偏移量，这些偏移量用于安全地读取解释器和线程状态内存。这些偏移量在CPython版本之间有所变化，必须在使用前进行检查以确保它们是兼容的。

要读取和检查调试偏移量，请按照以下步骤操作：

1. 从目标进程的 `PyRuntime` 地址开始读取内存，覆盖的字节数与 `_Py_DebugOffsets` 结构相同。该结构位于 `PyRuntime` 内存块的起始位置。其布局在CPython的内部头文件中定义，并在给定的小版本中保持不变，但在大版本中可能会发生变化。
2. 检查该结构是否包含有效数据：
   - `cookie` 字段必须与预期的调试标记匹配。
   - `version` 字段必须与调试器使用的Python解释器版本匹配。
   - 如果调试器或目标进程使用的是预发布版本（例如，alpha、beta或发布候选版本），则版本必须完全匹配。
   - `free_threaded` 字段在调试器和目标进程中必须具有相同的值。
3. 如果结构体有效，其中包含的偏移量可以用于定位内存中的字段。如果任何检查失败，调试器应停止操作，以避免以错误格式读取内存。

以下是一个读取和检查 `_Py_DebugOffsets` 的示例实现：

```
def read_debug_offsets(pid: int, py_runtime_addr: int) -> DebugOffsets:
    # 步骤1：从目标进程中读取PyRuntime地址处的内存
    data = read_process_memory(
        pid, address=py_runtime_addr, size=DEBUG_OFFSETS_SIZE
    )

    # 第2步：将原始字节反序列化为_Py_DebugOffsets结构体
    debug_offsets = parse_debug_offsets(data)

    # 步骤3：验证结构体的内容
    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
```

警告： 建议挂起进程

为避免竞态条件并确保内存一致性，在执行任何读取或写入解释器内部状态的操作前，强烈建议先挂起目标进程。Python运行时可能在正常执行期间并发修改解释器数据结构（例如创建或销毁线程），这可能导致无效的内存读写操作。

调试器可以通过使用 `ptrace` 附加到进程或发送 `SIGSTOP` 信号来挂起执行。只有在调试器端的内存操作完成后，才应恢复执行。

备注： 一些工具，如性能分析器或基于采样的调试器，可以在不挂起运行进程的情况下操作。在这种情况下，工具必须明确设计以处理部分更新或不一致的内存。对于大多数调试器实现来说，挂起进程仍然是最安全、最稳健的方法。

定位解释器和线程状态

在远程Python进程中注入并执行代码前，调试器必须选定一个目标线程来调度执行。这是因为用于远程代码注入的控制字段位于 `_PyRemoteDebuggerSupport` 结构体中，而该结构体又嵌入在 `PyThreadState` 对象内。调试器通过修改这些字段来请求执行已注入的脚本。

`PyThreadState` 结构体表示在Python解释器内运行的线程。它维护线程的求值上下文，并包含调试器协调所需的字段。因此，定位一个有效的 `PyThreadState` 是触发远程执行的关键前提。

通常基于线程的角色或ID来选择线程。在大多数情况下，使用主线程，但一些工具可能通过其本地线程ID定位特定线程。一旦选择了目标线程，调试器必须在内存中定位解释器和相关的线程状态结构。

相关内部结构体定义如下：

- `PyInterpreterState` 表示一个隔离的Python解释器实例。每个解释器维护其自己的导入模块集、内置状态和线程状态列表。尽管大多数Python应用程序使用单个解释器，但CPython支持在同一进程中使用多个解释器。
- `PyThreadState` 表示在解释器内运行的线程。它包含执行状态和调试器使用的控制字段。

要定位一个线程：

1. 使用偏移量 `runtime_state.interpreters_head` 获取 `PyRuntime` 结构体中第一个解释器的地址。这是活动解释器链表的入口点。
2. 使用偏移量 `interpreter_state.threads_main` 访问与选定解释器相关联的主线程状态。这通常是目标的最可靠线程。
3. 可选地，使用偏移量 `interpreter_state.threads_head` 遍历所有线程状态的链表。每个 `PyThreadState` 结构体包含一个 `native_thread_id` 字段，可以将其与目标线程 ID 进行比较以找到特定线程。
4. 一旦找到有效的 `PyThreadState`，其地址可以在协议的后续步骤中使用，例如写入调试器控制字段和调度执行。

以下是一个定位主线程状态的示例实现:

```
def find_main_thread_state(
    pid: int, py_runtime_addr: int, debug_offsets: DebugOffsets,
) -> int:
    # 步骤 1：从 PyRuntime 中读取 interpreters_head
    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("在目标进程中没有找到解释器")

    # 步骤2：从解释器中读取threads_main指针
    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("主线程状态不可用")

    return thread_state_addr
```

以下示例演示了如何通过其本地线程 ID 定位线程：

```
def find_thread_by_id(
    pid: int,
    interp_addr: int,
    debug_offsets: DebugOffsets,
    target_tid: int,
) -> int:
    # 从 threads_head 开始遍历链表
    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("没有找到给定ID的线程")
```

一旦定位到有效的线程状态，调试器可以继续修改其控制字段并调度执行，如下一节所述。

写入控制信息

一旦识别出有效的 `PyThreadState` 结构体，调试器可以修改其中的控制字段以调度指定 Python 脚本的执行。这些控制字段由解释器定期检查，当正确设置时，它们会在求值循环的安全点触发远程代码的执行。

每个 `PyThreadState` 包含一个 `_PyRemoteDebuggerSupport` 结构体，用于调试器和解释器之间的通信。其字段的位置由 `_Py_DebugOffsets` 结构体定义，包括以下内容：

- `debugger_script_path`：一个固定大小的缓冲区，用于存储 Python 源文件（`.py`）的完整路径。当触发执行时，目标进程必须能够访问并读取该文件。

- `debugger_pending_call`：一个整数型旗标。将其设为 `1` 表示告知解释器已有脚本准备就绪等待执行。
- `eval_breaker`：解释器在执行过程中会检查的字段。设置该字段的第5位（`_PY_EVAL_PLEASE_STOP_BIT`，值为 `1U << 5`）将使解释器暂停并检查调试器活动。

要完成注入，调试器必须执行以下步骤：

1. 将完整脚本路径写入 `debugger_script_path` 缓冲区。
2. 将 `debugger_pending_call` 设置为 `1`。
3. 读取 `eval_breaker` 的当前值，设置位 5（`_PY_EVAL_PLEASE_STOP_BIT`），并将更新后的值写回。这会指示解释器检查调试器活动。

以下是一个示例实现：

```
def inject_script(
    pid: int,
    thread_state_addr: int,
    debug_offsets: DebugOffsets,
    script_path: str
) -> None:
    # 计算 _PyRemoteDebuggerSupport 的基准偏移量
    support_base = (
        thread_state_addr +
        debug_offsets.debugger_support.remote_debugger_support
    )

    # 步骤 1：将脚本路径写入 debugger_script_path
    script_path_ptr = (
        support_base +
        debug_offsets.debugger_support.debugger_script_path
    )
    write_string(pid, script_path_ptr, script_path)

    # 步骤 2：将 debugger_pending_call 设置为 1
    pending_ptr = (
        support_base +
        debug_offsets.debugger_support.debugger_pending_call
    )
    write_int(pid, pending_ptr, 1)

    # 步骤 3：在 eval_breaker 中设置 _PY_EVAL_PLEASE_STOP_BIT
    # （第 5 位，值为 1 << 5）
    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)
```

设置这些字段后，调试器可以恢复进程（如果它被挂起）。解释器将在下一个安全求值点处理请求，从磁盘加载脚本并执行它。

调试器有责任确保脚本文件在执行期间对目标进程保持存在和可访问。

> **备注：** 脚本执行是异步的。注入脚本后不能立即删除脚本文件。调试器应等待注入脚本产生可观察的效果后再删除文件。这个效果取决于脚本的设计目的。例如，调试器可能会等待远程进程连接回套接字后再删除脚本。一旦观察到此类效果，可以安全地假设文件不再需要。

总结

要在远程进程中注入并执行 Python 脚本：

1. 在目标进程的内存中定位 `PyRuntime` 结构体。
2. 读取并验证 `PyRuntime` 开头的 `_Py_DebugOffsets` 结构体。
3. 使用该偏移量来定位一个有效的 `PyThreadState`。
4. 将一个 Python 脚本的路径写入到 `debugger_script_path`。
5. 将 `debugger_pending_call` 旗标设为 `1`。
6. 设置 `eval_breaker` 字段中的 `_PY_EVAL_PLEASE_STOP_BIT`。
7. 恢复进程（如已挂起）。 脚本将在下一个安全求值点开始执行。