

第 25 章 用于自动化逆向的二进制插桩

上一章中介绍了如何使用恶意软件沙盒来动态分析样本文件,记录样本文件的行为并分析其恶意性,以及如何自动化整个沙盒分析过程。但大多数基于行为与 API 记录的沙盒都存在一个较为明显的缺点,那就是很容易受到恶意软件采用的反分析技术的影响。除非分析时在尽可能低的层级(即机器指令级)进行处理,否则难以对抗反分析技术。具体来说,分析人员可以通过一种名为二进制插桩的技术来达到该目的,从而在机器指令级分析、处理甚至修改正在运行的程序。本章解释了二进制插桩的含义以及内部实现,阐述了二进制插桩为什么能够在机器指令级别监视程序运行。通过本章的各种分析示例,读者能够编写简单的检测工具来辅助分析恶意样本,甚至对其进行自动化的逆向工程。

什么是二进制插桩?

插桩是一种测量、监控程序或进程性能,跟踪其执行情况的方法。本章中将会对其功能进行介绍,甚至包括修改程序的行为。插桩与前述介绍的程序分析方法并没有什么不同,主要是插桩比之前使用的分析技术粒度更细。具体来说,本章中主要讨论的是被称为动态二进制插桩(DBI)的分析技术。如图 25-1 所示,动态二进制插桩是一种程序分析技术,主要针对二进制可执行文件进行分析。该技术属于动态分析的技术手段,会在运行时分析、处理二进制文件。动态二进制插桩也被称为动态二进制翻译,但这只是名称上的差异其实并没有本质区别。

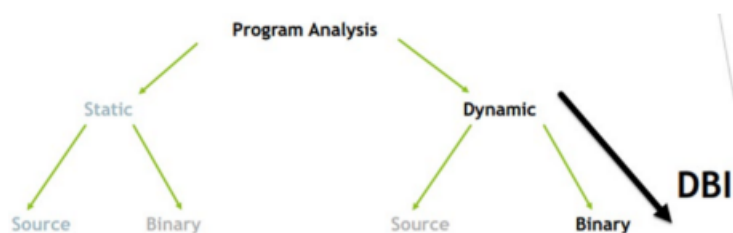


图 25-1

现在有一个想要检测的、正在运行的进程。分析人员可以通过各种分析工具使用另一个进程被动地检测该进程，例如对与该进程相关的事件进行采样，对进程与操作系统调用的交互进行采样等。但被动检测确实存在一些缺点，例如记录受监视进程的数据粒度与细节，以机器码指令的粒度监视进程的能力等。为了弥补这些缺点，进一步获取受监视进程细粒度数据，分析人员可以采用主动检测技术。主动检测技术涉及修改程序或者进程中的代码，提供进程执行细粒度的记录。

现在，我们已经知道了有一种通过修改程序或进程代码进行主动检测的技术。如果可以得到程序的源代码，用户就可以修改源代码来添加各种插桩代码。在源代码可用时，这种方式被称为源代码插桩。但大多数情况，包括恶意软件在内的各种应用程序都不提供源代码。分析人员只能通过修改其机器码来插桩二进制程序，这种方式就是二进制插桩。

与恶意软件分析领域中分为静态分析与动态分析类似，二进制插桩可以根据使用方式进一步分为静态二进制插桩与动态二进制插桩（DBI）。在静态二进制插桩中，插桩代码在程序运行前就被插入到程序文件中，如图 25- 2 所示。

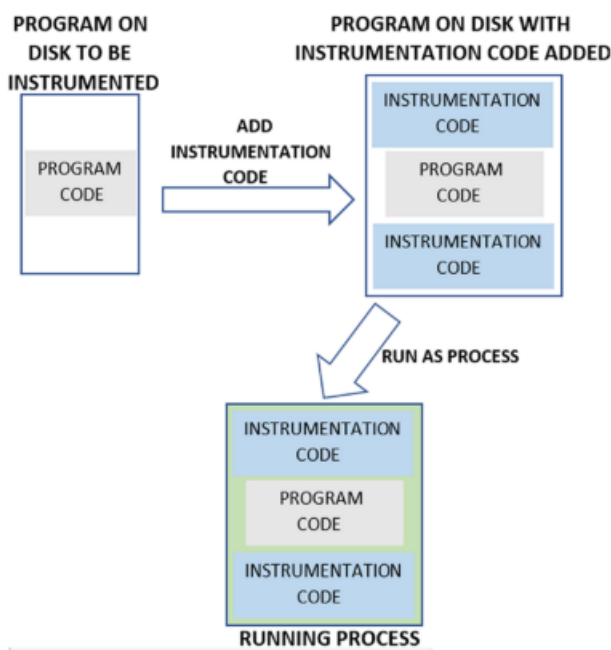


图 25- 2

如图所示，使用静态二进制插桩需要在各种插桩框架的帮助下，修改要分析的程序文件并生

成新的程序文件。新生成的程序文件包含原始的程序代码与插入的插桩代码，再运行该文件时会执行原始的程序代码与插入的插桩代码。

如图 25- 3 所示，使用动态二进制插桩会将插桩代码在运行时动态地插入到正在运行的进程中。

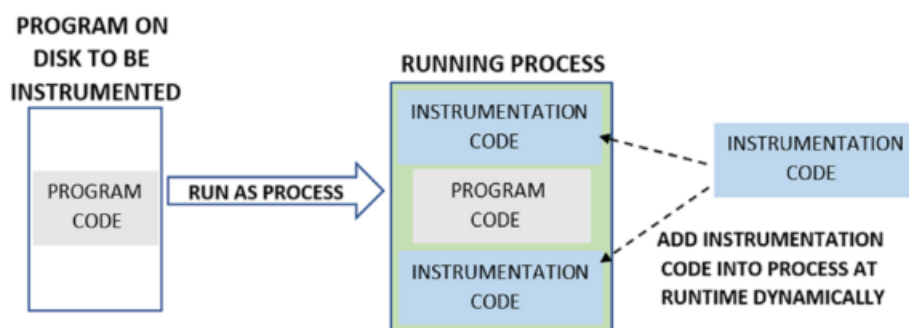


图 25- 3

到这里已经介绍了动态二进制插桩的基本概念，接下来的几节将会进一步解释动态二进制插桩的技术实现与相关术语，帮助读者深入理解其内部工作原理。

动态二进制插桩：相关术语与内部结构

动态二进制插桩技术要将插桩代码插入到正在运行的程序中，多个动态二进制插桩框架都可以帮助实现这一点，例如 PIN、Dynamo RIO、Frida 与 Valgrind。这些动态二进制插桩框架支持用户添加自定义的插桩代码来修改正在运行的程序中的代码，可以利用该框架监控正在运行的进程。

事实上，需要插桩的程序作为一个进程由动态二进制插桩框架来运行，由框架进行完全控制。动态二进制插桩框架会读取程序中的指令，从逻辑上将其分解为许多指令块以及块与块之间的结构关系，并且在这些指令块中插入插桩代码。如图 25- 4 所示，大多数动态二进制插桩框架使用两种结构来拆分程序中的指令以进行插桩：基本块（block）与 Trace。

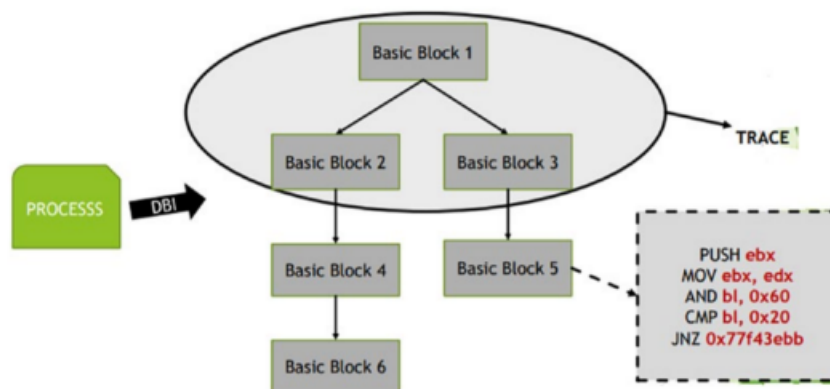


图 25- 4

基本块是一组指令，这组指令只有一个入口点与一个出口点。无论出口点的退出指令是有条件的还是无条件的，该基本块都会在此处结束。另一方面，Trace 由一组基本块组成，这组基本块有一个入口点，但只会在无条件退出指令（如 CALLS 与 RETURNS）处结束。如图 25- 5 所示，Trace 能够保证只有一个入口点，但由于包含多个基本块，Trace 是可以有多个出口点的。

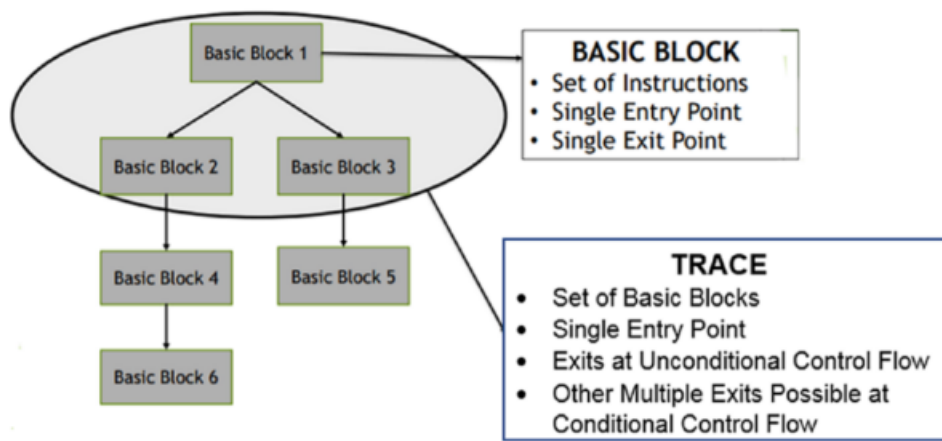


图 25- 5

以图 25- 6 为例，它由进程中的一组指令组成，该进程已被动态二进制插桩框架拆分为两个基本块。每个基本块都由一个入口点，每个基本块都在第一条退出指令处结束。对于基本块 1 来说，退出指令或者说条件分支指令是 JNZ。对于基本块 2 来说，退出指令是无条件退出指令 CALL。这两个基本块组成一个 Trace，其拥有一个入口点并且在第一条无条件退出指令（基本块 2 中的 CALL 指令）处退出。

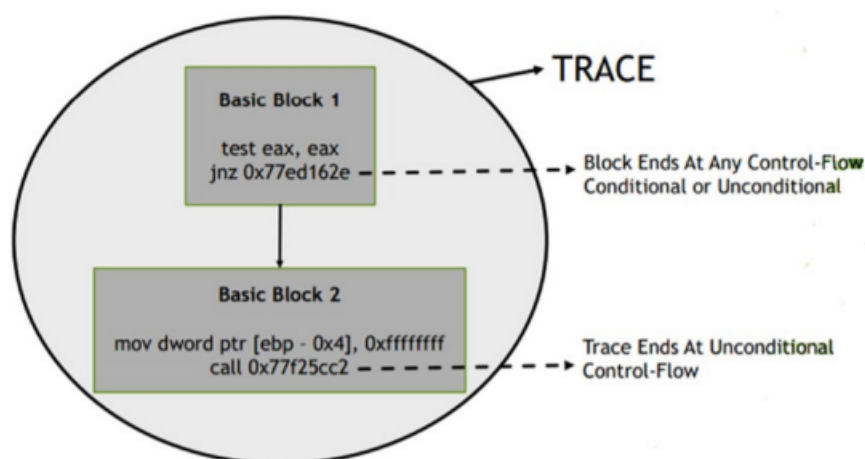


图 25- 6

插入插桩代码

如前所述，动态二进制插桩框架会将代码中的指令拆分为基本结构，如基本块与 Trace。基于此，动态二进制插桩框架就可以在这些基本块与 Trace 中插入基础插桩代码与用户插桩代码来对其进行监控与插桩，如图 25- 7 所示。

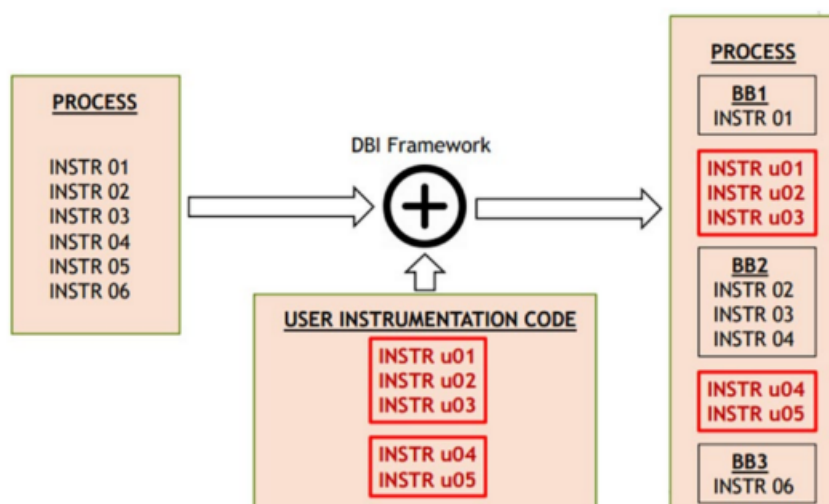


图 25- 7

大多数动态二进制插桩框架都支持通过回调 API 插入用户插桩代码，然后在被插桩进程触发各种事件或执行阶段调用这些回调函数。借助这种特性，用户可以在动态二进制插桩框架的帮助下开发分析程序。通过在动态二进制插桩框架的 API 中注册自定义的用户插桩回调函数，控制动态二进制插桩框架在进程执行时的各个阶段调用回调函数。例如，用户可以向

动态二进制插桩框架注册一个回调函数，要求其在基本块或 Trace 执行前调用回调函数。类似的，也可以注册要求动态二进制插桩框架在基本块或 Trace 执行后调用的回调函数。大多数动态二进制插桩框架甚至支持在逐指令级与 subroutine 级注册用户插桩代码的回调函数。接下来的几节中，将会深入探讨动态二进制插桩框架在恶意软件分析时的各种应用。提供的各种练习，可以帮助读者学习如何使用 Intel PIN 等动态二进制插桩框架，包括基于此辅助自动化恶意软件分析。

用于恶意软件分析的动态二进制插桩

动态二进制插桩技术用途广泛，下面列出了其中的一些：

- 代码性能分析
- 错误诊断
- 代码流分析
- 污点分析
- 内存分配与内存泄漏跟踪
- 漏洞检测
- 程序调试
- 恶意软件逆向工程
- 修复漏洞
- 漏洞利用开发
- 错误诊断

动态二进制插桩的所有应用场景远非以上列表能够穷举，该技术在恶意软件分析与样本文件自动化逆向工程中也有着广泛的应用。业界使用动态二进制插桩技术开发了各种用于分析恶

意样本的工具，例如 Trishool。该工具可以通过 <https://github.com/Juniper/trishool> 获取，读者阅读完本章内容后就可以尝试使用该工具。以下是动态二进制插桩在自动化恶意软件分析和逆向工程方面的一些应用：

- Win32 API 日志记录
- 脱壳
- 修改代码与进程状态绕过分析对抗技术
- 内存签名扫描
- 路径模糊测试
- 应用程序内存分配跟踪
- 恶意代码段回溯
- 与 IDA Pro 相似的代码块流程图

在下一节中将会尝试使用动态二进制插桩技术来编写简单的分析工具，这也是迈向自动化逆向分析恶意软件样本文件的良好开端。

缺点

前文介绍了动态二进制插桩在自动化恶意软件分析与逆向工程领域的各种优点，但该技术也有缺点，并不能直接替代沙盒中的 API 日志记录工具。

大多数恶意软件分析沙盒都是模拟运行，这可能会很慢。即使对 APIMiner 这样的 API 日志记录工具来说也会受限于性能，所以分析人员不能或者最好不要在沙盒内执行 CPU 密集型任务。通常都是将获取的分析日志与各种数据从沙盒内部传输到外部，在宿主机上进行 CPU 密集型的日志分析等任务。

而动态二进制插桩也是 CPU 密集型的，特别是与 APIMiner 这样的 API 日志记录工具相比，

对资源的要求更高。因此，使用动态二进制插桩分析每个样本文件可能并不切实际。如果检测产品中使用了恶意软件沙盒时，分析人员仍然希望使用 CPU 密集度较低的分析工具，如 APIMiner 等 API 日志记录工具。这些分析工具使用 CPU 资源消耗较少的、基于 Hook 的技术，获取有关样本文件的第一手 API 日志与其他数据。当所获取的分析日志不足以支撑分析时，再使用其他复杂的分析工具与分析技术（如动态二进制插桩）介入，重新对样本文件进行分析。这样就可以将动态二进制插桩的使用场景限制在常规分析工具失效的情况下，从而节约宝贵的计算资源。

使用动态二进制插桩框架编写分析工具

接下来介绍如何使用动态二进制插桩框架编写简单的分析工具。本书选择使用 Intel PIN 作为动态二进制插桩框架的示例，但使用 DynamoRIO、Frida 与其他动态二进制插桩框架也能够实现同样的效果。使用其他动态二进制插桩框架编写分析工具的任务，当作练习留给各位读者进行实践。

设置 PIN

在第 21 章已经配置了 Cygwin 与 Microsoft Visual Studio，本节要对使用的 Intel PIN 框架进行设置。首先，要从 Intel 网站上下载 Intel PIN 框架。本书使用 3.6 版本的 Intel PIN，读者可以自行选用最新版本的 PIN。由于版本带来的差异，读者可能需要进行一些细微的调整。

如图 25- 8 所示，读者可以首先将示例样本文件库中的 chapter_25_samples.zip 复制到 Windows 开发虚拟机中的 Documents 文件夹并解压缩。

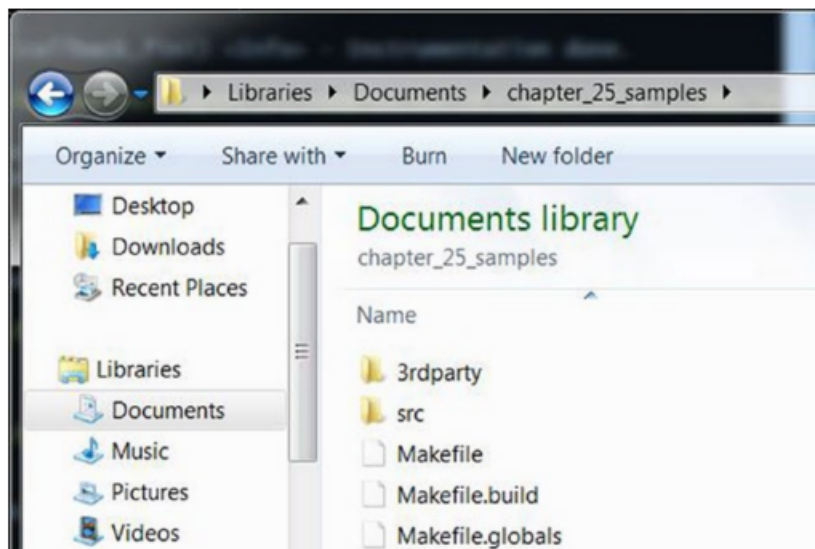


图 25- 8

现在，要将前面下载的 Intel PIN 框架复制到文件夹 `chapter_25_samples/3rdparty/pin` 中。并且，需要手动编辑文件夹中的 `Makefile` 文件以更新 `PIN_VERSION` 变量的值。本书使用的是 `pin-3.6-97554-g31f0a167d-msvc-windows.zip`，在文件名中删掉前缀 (`pin-`) 与后缀 (`.zip`) 就剩下了版本字符串 (`3.6-97554-g31f0a167d-msvc-windows`)。如图 25- 9 所示，将该版本字符串设置为 `Makefile` 文件中 `PIN_VERSION` 变量的值。

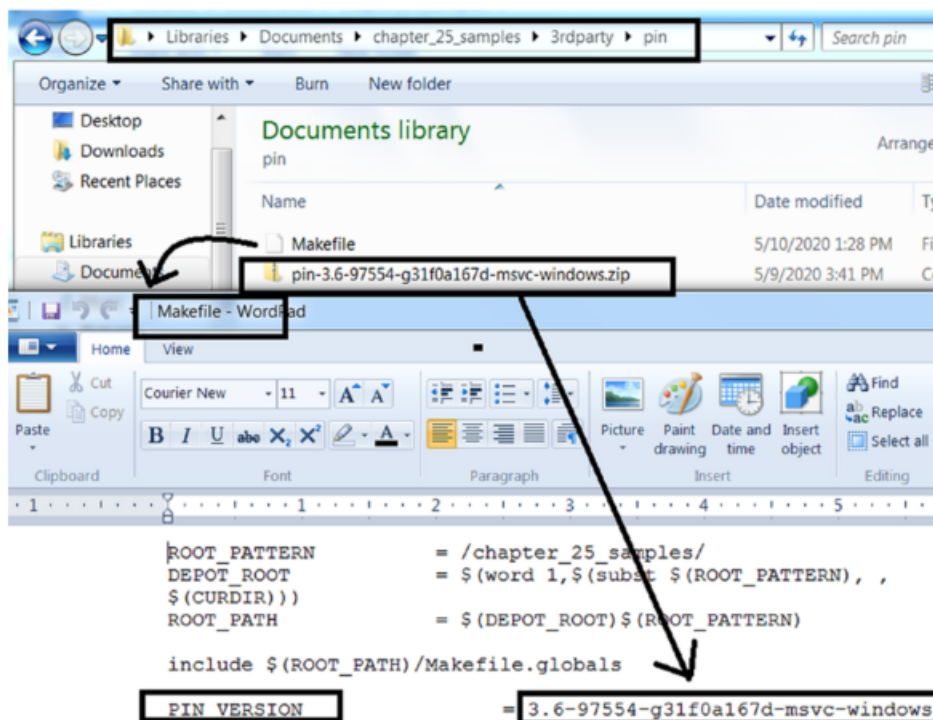


图 25- 9

本书提供了三个练习文件（Sample-25-03-pin.c、Sample-25-04-pin.c 与 Sample-25-05-pin.c）都位于 chapter_25_samples/src/samples 文件夹下，后续几节将会使用它们来插桩两个应用程序（Sample-25-01 和 Sample-25-02），这两个程序也在示例样本文件库中。

要构建 Intel PIN 练习文件，需要使用在第 21 章中介绍的 Cygwin.bat 文件打开 Cygwin。首先执行 cd 命令将目录切换到 Documents/chapter_25_samples 下，再执行如图 25- 10 所示的命令。

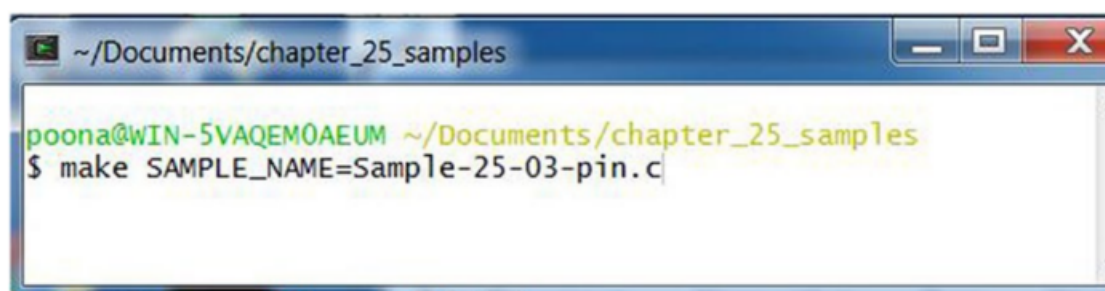


图 25- 10

命令执行后会基于 Sample-25-03-pin.c 构建分析工具，并将输出文件生成到 chapter_25_samples/文件夹下的 build-*目录中，如图 25- 11 所示。

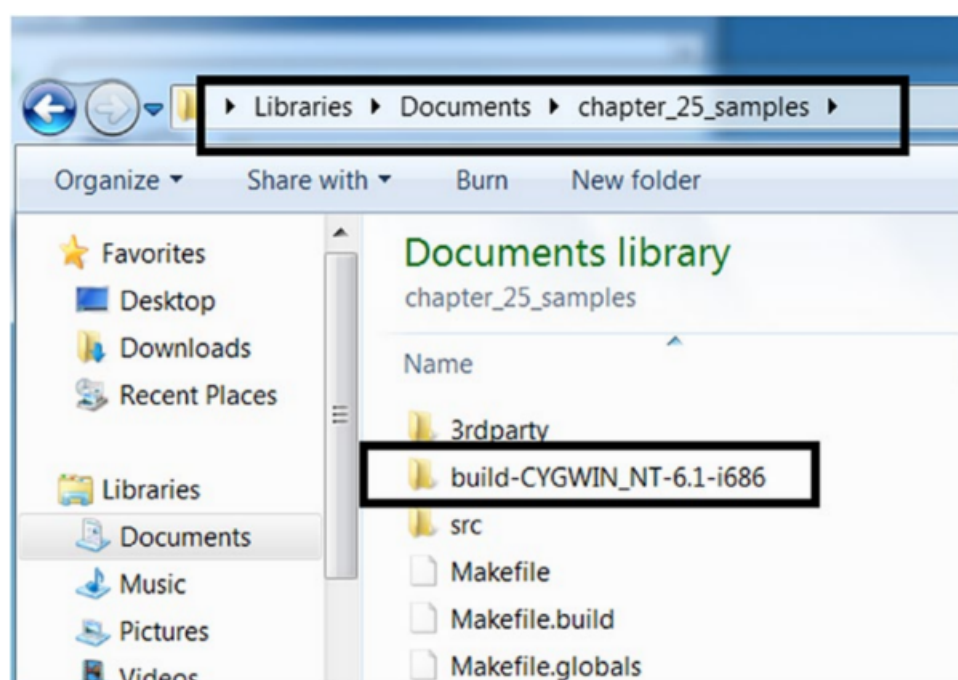


图 25- 11

如图 25- 12 所示，基于 Sample-25-03-pin.c 构建的分析工具会输出到 chapter_25_samples/build-CYGWIN_NT-6.1-i686/lib/Sample-pin-dll.dll。

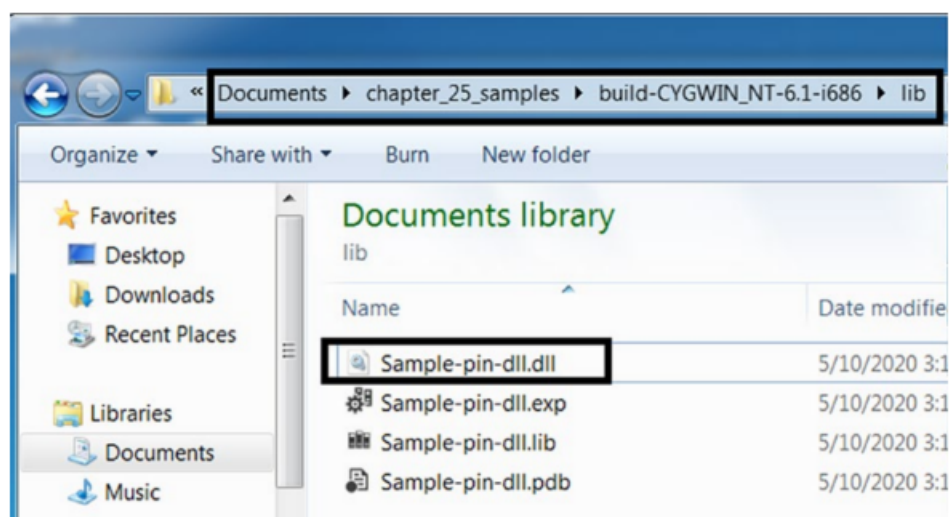


图 25- 12

首先，将构建好的分析工具 Sample-pin-dll.dll 复制到 chapter_25_samples/build-CYGWIN_NT-6.1-i686/3rdparty/pin/ 文件夹中。如图 25- 13 所示，再从示例样本文件库中复制 Sample-25-01 与 Sample-25-02 到相同文件夹，并为其增加.exe 文件扩展名。

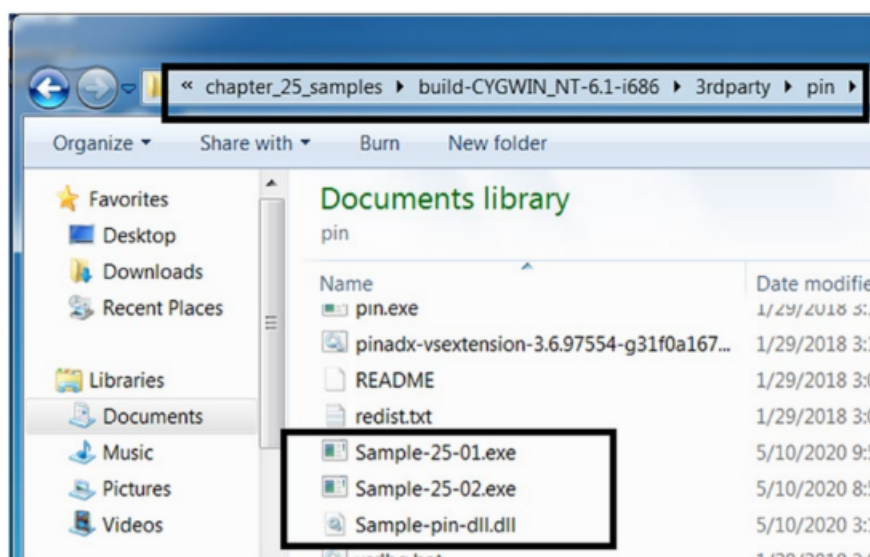


图 25- 13

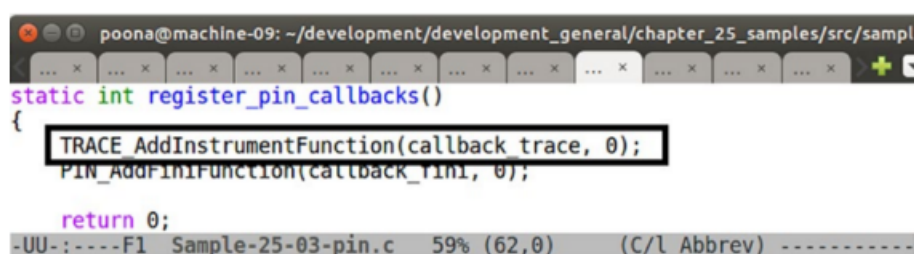
这样就足够了。后续想要构建其他两个分析工具（Sample-25-04-pin.c 和 Sample-25-04-pin.c）时，只需要删除 chapter_25_samples/build-*文件夹并重复以上构

建步骤即可。

工具 1：记录所有指令

基于 chapter_25_samples/src/samples/Sample-25-03-pin.c 构建的第一个分析工具, 旨在记录要监控的应用程序的所有指令。正如在分析工具中看到的那样, 当分析工具根据正在监控的进程的指令生成 Trace 时, 分析工具会向 PIN 框架注册生成的任何事件。

如图 25- 14 所示, 分析工具调用名为 TRACE_AddInstrumentFunction 的 PIN API, 控制其注册用户自定义的回调函数 callback_trace。这样, 即为要求 PIN 为正在监控的样本文件生成的每个 Trace 调用该回调函数。

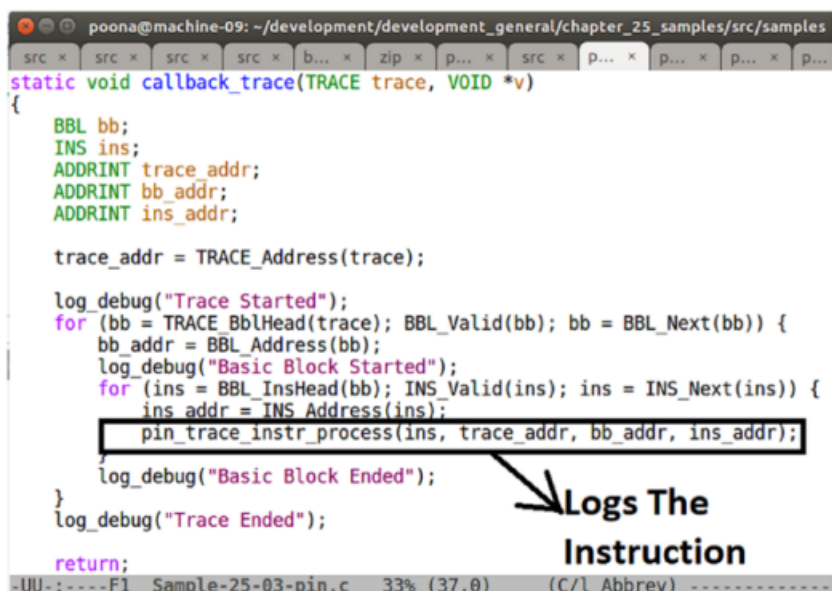


```
poona@machine-09: ~/development/development_general/chapter_25_samples/src/sampl
static int register_pin_callbacks()
{
    TRACE_AddInstrumentFunction(callback_trace, 0);
    PIN_AddFiniFunction(callback_fini, 0);

    return 0;
}
-UU-:----F1 Sample-25-03-pin.c 59% (62,0) (C/l Abbrev) -----
```

图 25- 14

与预期相符, PIN 会调用分析工具注册的回调函数, 向该函数提供根据正在监视的应用程序的指令生成的每个 Trace。如图 25- 15 所示, 回调函数 callback_trace 获取 Trace 后, 会查看 Trace 中的所有基本块, 依次遍历每个基本块中的每条指令。



```
poona@machine-09: ~/development/development_general/chapter_25_samples/src/samples
static void callback_trace	TRACE trace, VOID *v)
{
    BBL bb;
    INS ins;
    ADDRINT trace_addr;
    ADDRINT bb_addr;
    ADDRINT ins_addr;

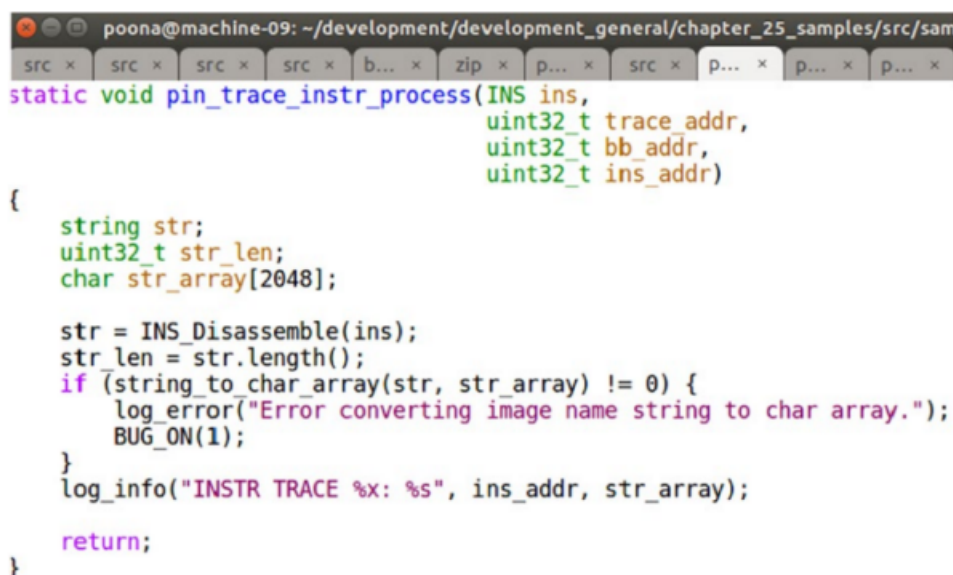
    trace_addr = TRACE_Address(trace);

    log_debug("Trace Started");
    for (bb = TRACE_BblHead(trace); BBL_Valid(bb); bb = BBL_Next(bb)) {
        bb_addr = BBL_Address(bb);
        log_debug("Basic Block Started");
        for (ins = BBL_InsHead(bb); INS_Valid(ins); ins = INS_Next(ins)) {
            ins_addr = INS_Address(ins);
            pin_trace_instr_process(ins, trace_addr, bb_addr, ins_addr);
        }
        log_debug("Basic Block Ended");
    }
    log_debug("Trace Ended");

    return;
}
-UU-:----F1 Sample-25-03-pin.c 33% (37,0) (C/l Abbrev) -----
```

图 25- 15

每条指令都会调用 pin_trace_instr_process 函数，这会将指令记录到指定的日志文件 poona_log.txt 中，如图 25- 16 所示。



```
poona@machine-09: ~/development/development_general/chapter_25_samples/src/sam
static void pin_trace_instr_process(INS ins,
                                   uint32_t trace_addr,
                                   uint32_t bb_addr,
                                   uint32_t ins_addr)
{
    string str;
    uint32_t str_len;
    char str_array[2048];

    str = INS_Disassemble(ins);
    str_len = str.length();
    if (string_to_char_array(str, str_array) != 0) {
        log_error("Error converting image name string to char array.");
        BUG_ON(1);
    }
    log_info("INSTR TRACE %x: %s", ins_addr, str_array);

    return;
}
```

图 25- 16

想要构建该分析工具，请先删除 chapter_25_samples 文件夹中的 build-*文件夹，随后重新运行如图 25- 10 所示的命令。如图 25- 12 所示，执行构建命令后，该工具的可执行文件会输出到 Sample-pin-dll.dll。如图 25- 13 所示，将该 DLL 文件复制到 build-CYGWIN_NT-6.1-i686/3rdparty/pin/文件夹下。如前所述，也要将示例样本库中 Sample-25-01 与 Sample-25-02 复制到相同文件夹下，并为其增加.exe 的文件扩展名。

完成前述配置后, 通过 cd 命令进入 build-CYGWIN_NT-6.1-i686/3rdparty/pin/目录。如图 25- 17 所示, 使用构建好的分析工具通过 PIN 运行应用程序 Sample-25-01.exe。



图 25- 17

命令执行后, 就可以利用基于 Sample-25-03-pin.c 构建的分析工具将所有指令记录到相同路径下名为 poona_log.txt 的日志文件中。如所示, 打开该文件查看即可发现其中包含被插桩的应用程序 Sample-25-01.exe 中的所有指令。

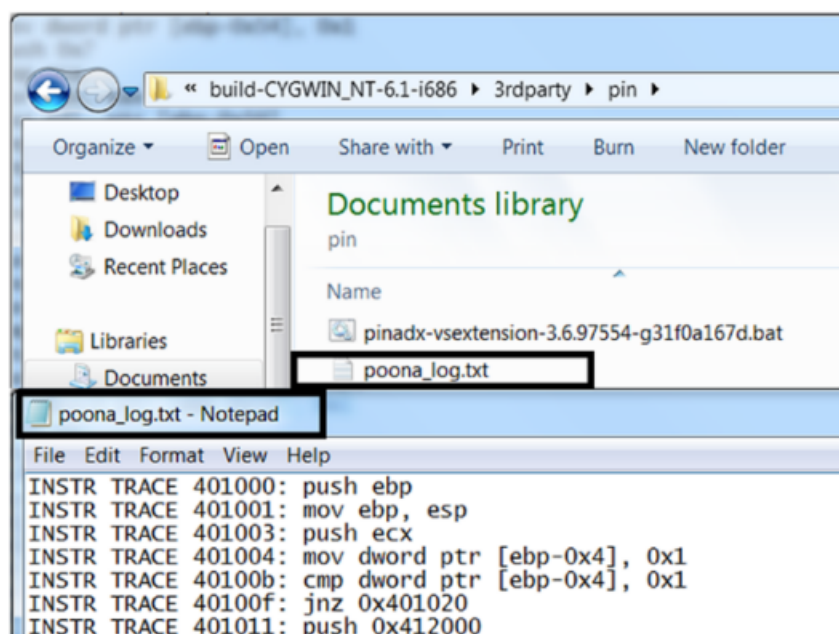


图 25- 18

工具 2: Win32 API 日志记录

本书前面的章节中介绍了使用 APIMiner 与 Cuckoo 沙盒等分析工具来记录恶意软件样本文件调用的 Win32 API 日志。事实上, 使用动态二进制插桩框架也可以实现相同的目标。基于 chapter_25_samples/src/samples/Sample-25-04-pin.c 构建的第二个分析工具就是完成这一功能的。查看 Sample-25-04-pin.c 中的代码, 与图 25- 14 类似, 也是注册一个

回调函数 `callback_trace` 从被插桩的程序接收 `Trace`。PIN 调用 `callback_trace` 时会传送被插桩程序的 `Trace`，如图 25- 15 所示循环遍历所有指令。相同的，对于每条指令都要调用 `pin_trace_instr_process`。基于 `Sample-25-04-pin.c` 构建的分析工具与基于 `Sample-25-03-pin.c` 构建的分析工具的主要区别在于 `pin_trace_instr_process` 函数的实现，如图 25- 19 所示。

```
static void pin_trace_instr_process(INS ins,
                                   uint32_t trace_addr,
                                   uint32_t bb_addr,
                                   uint32_t ins_addr)
{
    string str;
    uint32_t str_len;
    char str_array[2048];

    if (INS_IsCall(ins)) {
        INS_InsertCall(ins,
                       IPOINTEBEFORE,
                       (AFUNPTR)pin_callback_call_instr,
                       IARG_INST_PTR,
                       IARG_BRANCH_TARGET_ADDR,
                       IARG_END);
    }
}
```

图 25- 19

与 `APIMiner` 类似，该分析工具的目标是记录应用程序使用的 API。如前所述，API 调用不过是函数调用，在机器码或者汇编代码中通过 `CALL` 命令实现。该分析工具针对每条指令，都会利用名为 `INS_IsCall()` 的 PIN API 检查该指令是否为 `CALL` 指令。分析工具已经针对该指令向 PIN 框架注册一个新的回调函数 `pin_callback_call_instr`，如果确认为 `CALL` 指令，则会在执行该指令前调用此回调函数。所谓的“该指令”，并不是之所有的 `CALL` 指令，而是指位于应用程序该地址的这条特定的 `CALL` 指令。通常来说，最终会为被插桩的应用程序的每条 `CALL` 指令注册该回调函数。

对应于图 25- 19 中的 `IPOINTEBEFORE`，在执行特定 `CALL` 指令前，PIN 会调用回调函数 `pin_callback_call_instr`。如图 25- 20 所示，该函数会利用 PIN 提供的 API 获取 API 的名称，并将其记录下来。进一步跟踪 `platform_rtn_name_from_addr` 的 API 调用可以发现，PIN 最终是通过 `RTN_FindNameByAddress` 来获取 Win32 API 名称的。

```
static void pin_callback_call_instr(ADDRINT pc, ADDRINT target_addr)
{
    char rtn_name[2048];
    uint32_t rtn_name_len;

    if (image_address_is_win32(target_addr)) {
        rtn_name_len = sizeof(rtn_name);
        if (platform_rtn_name_from_addr(target_addr,
                                        rtn_name,
                                        &rtn_name_len) == 0)
        {
            log_info("Routine Called: %s", rtn_name);
        }
    }

    return;
}
```

图 25- 20

想要基于 Sample-25-04-pin.c 构建分析工具,请先删除 chapter_25_samples 文件夹中的 build-*文件夹,随后重新运行如图 25- 10 所示的命令。请注意,要将 make 命令对应的文件修改为 Sample-25-04-pin.c。如图 25- 12 所示,执行构建命令后,该工具的可执行文件会输出到 Sample-pin-dll.dll。如图 25- 13 所示,将该 DLL 文件复制到 build-CYGWIN_NT-6.1-i686/3rdparty/pin/文件夹下。如前所述,也要将示例样本库中 Sample-25-01 与 Sample-25-02 复制到相同文件夹下,并为其增加.exe 的文件扩展名。完成前述配置后,通过 cd 命令进入 build-CYGWIN_NT-6.1-i686/3rdparty/pin/目录。如图 25- 21 所示,使用构建好的分析工具通过 PIN 运行应用程序 Sample-25-02.exe。

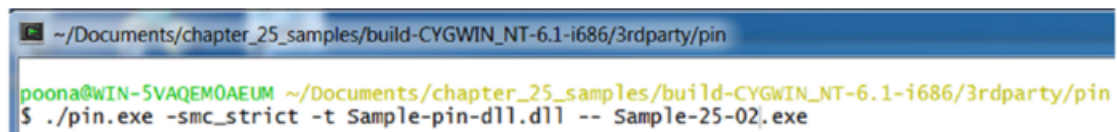


图 25- 21

如图 25- 22 所示,构建的分析工具会将 Sample-25-02.exe 使用的 API 输出到 poona_log.txt。右侧为 Sample-25-02.exe 对应的源代码 Sample-25-02.c,从代码可知其按 Sleep、VirtualAlloc 与 Sleep 的顺序调用了 Win32 API。左侧也能看到,分析工具将这一顺序的 Win32 API 调用记录了下来。

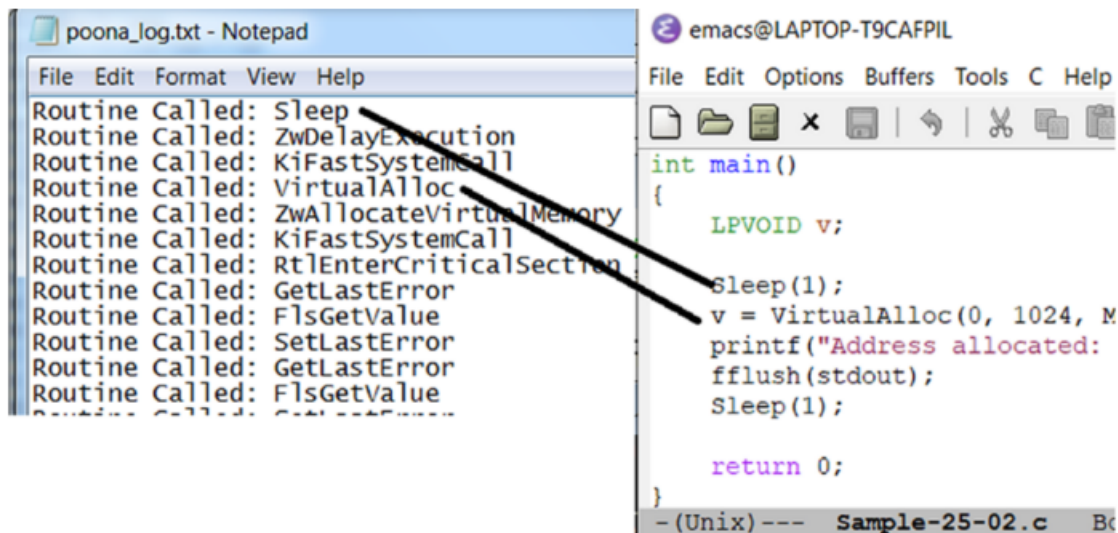


图 25- 22

通过代码可以查看分析工具实现其功能所使用的 PIN API。另外也可以通过查看 PIN 官方提供的 API 文档，了解这些 API 的具体含义，从而进一步调整并使用新的 API。读者可以将针对分析工具的改造作为练习，使分析工具像 APIMiner 一样支持记录这些 API 的参数并输出。

工具 3：代码修改和分支旁路

除了前述内容，通过动态二进制插桩还可以在插桩进程运行时实时修改指令与进程状态。这对于自动化恶意软件逆向分析时特别有用，尤其是以下两个场景：

- 对恶意软件的所有代码路径进行模糊测试，使用 APIMiner 等基于 Hook 的 API 记录工具对恶意软件进行分析时是不可能实现的，后者往往执行固定的代码路径。
- 绕过检测对抗技术。

以示例样本文件库中的 Sample-25-01.c 为例。如图 25- 23 所示，代码中包含一个 if-else 结构且被编程为始终执行 if 分支。

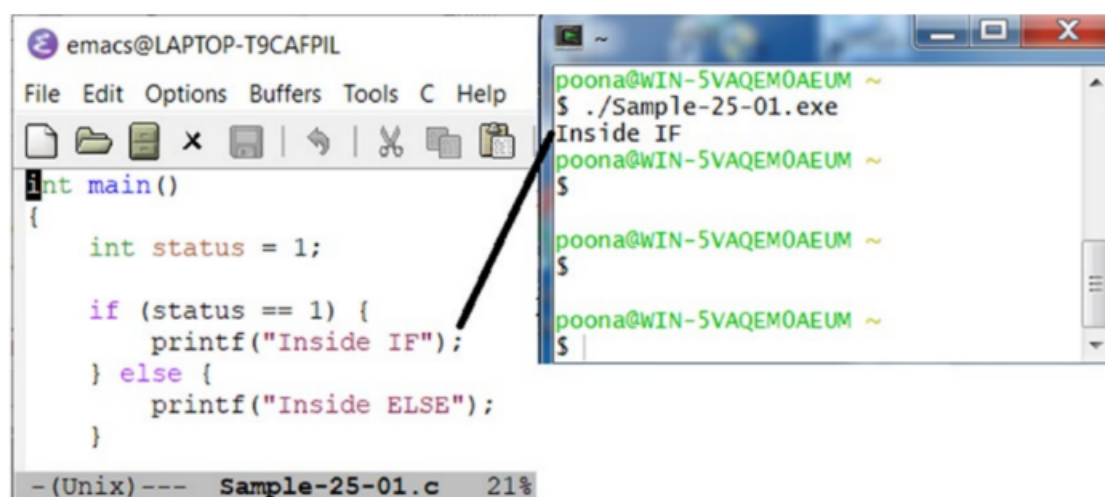


图 25- 23

可以利用动态二进制插桩实时修改这个程序的状态，控制其进入 else 分支吗？当然可以！

这也是基于 chapter_25_samples/src/samples/Sample-25-05-pin.c 构建的分析工具所要实现的目标。

如果将要插桩的程序 Sample-25-01.exe 进行反汇编，如图 25- 24 所示，可以看到程序使用 JNE 指令跳转 if-else 分支。JNE 指令会根据 FLAGS 寄存器的值决定向哪个分支跳转，而该寄存器会被位于 0x40100B 的 CMP 指令刷新。如果能在 JNE 指令执行前修改 FLAGS 寄存器的内容，就可以控制程序执行 else 分支。

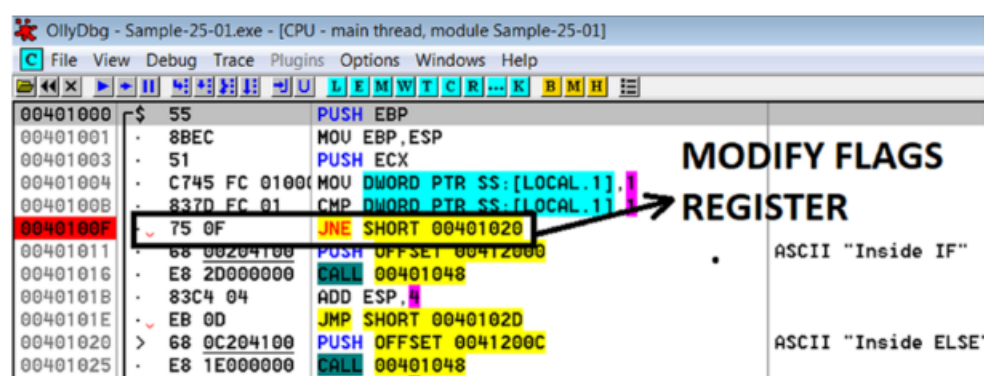


图 25- 24

这正是基于 Sample-25-05-pin.c 构建的分析工具要做的。如图 25- 25 所示，除了回调函数注册以外的其余代码与前面的分析工具非常相似。

```

poona@machine-09: ~/development/development_general/chapter_25_samples/sr
VOID modify_flag_for_branch(ADDRINT *eflags)
{
    *eflags = 0x00000297;
    return;
}

static void pin_trace_instr_process(INS ins,
                                   uint32_t trace_addr,
                                   uint32_t bb_addr,
                                   uint32_t ins_addr)
{
    string str;
    uint32_t str_len;
    char str_array[2048];

    if (ins_addr == 0x40100F) {
        INS_InsertPredicatedCall(ins, IPOINT_BEFORE,
                                (AFUNPTR)modify_flag_for_branch,
                                IARG_REG_REFERENCE, REG_EFLAGS,
                                IARG_UINT32, 0x00000297);
    }
}
-UU-:----F1 Sample-25-05-pin.c      6% (29,64) (C/l Abbrev) -----

```

图 25- 25

执行到图 25- 24 中位于 0x40100F 的 JNE 指令，PIN 会在执行该指令前调用注册的回调函数 `modify_flag_for_branch`。如果在源码文件 `Sample-25-05-pin.c` 中查看 `modify_flag_for_branch` 该函数的实现，可以看到其修改了 `FLAGS` 寄存器的值以改变进程的执行流。要构建该分析工具，请按照前两节介绍的步骤进行操作，此处不再赘述。如图 25- 26 所示，利用该分析工具再次运行 `Sample-25-01.exe` 即可发现已经执行 `else` 分支。

```

~/Documents/chapter_25_samples/build-CYGWIN_NT-6.1-i686/3rdparty/pin
poona@WIN-5VAQEM0AEUM ~/Documents/chapter_25_samples/build-CYGWIN_NT-6.1-i686/3rdparty/pin
$ ./pin.exe -smc_strict -t Sample-pin-dll.dll -- Sample-25-01.exe
Inside ELSE instrumentation done.

```

图 25- 26

通过动态二进制插桩可以实现更多更复杂的分析工具，以上三个只是作为示例让读者对其有个简单的了解。例如 `Trishool` (<https://github.com/Juniper/trishool>) 具有更多功能，包括确定恶意软件样本文件脱壳代码位置、扫描内存中的字符串等。通过查看 `Trishool` 如何实现这些功能，对学习如何自动化逆向工程有所帮助。

此外还可以查看 Intel 提供的 PIN API 官方文档，了解各种 API 与其细节。也可以浏览各种使用动态二进制插桩进行自动化程序分析的开源项目，都可以学到东西。不要局限于 PIN，还有很多其他动态二进制插桩框架也很不错。作者最喜欢的是 `DynamoRIO` 和 `Frida`，它们

都具有与 PIN 不同的功能。使用动态二进制插桩需要更多的实践练习，才能更熟练的使用各种 API。在大量实践的加持下，分析人员很快就可以利用动态二进制插桩来自动执行恶意软件逆向分析中的各种任务。

总结

动态二进制插桩 (DBI) 是一项非常有用的技术，利用该技术能够跨领域实现自动插桩、分析各种良性样本与恶意样本。本章介绍了插桩的含义以及各种子技术，包括动态二进制插桩 (DBI)。又进一步介绍了 DBI 的工作原理、了解了其内部概念以及大多数 DBI 框架常用的各种术语，包括 trace 与基本块。在第 21 章中配置的分析环境的基础上，也为分析环境更新了 PIN 分析工具。本章也是使用这些分析工具来编译构建的各种插桩练习文件。练习编写了简单的 PIN 分析工具，例如一个记录 API (如 APIMiner) 的分析工具与另一个支持修改进程实时状态以变更执行代码流路径的分析工具。