WinAFL 源码分析

hackedbylh / 2019-05-15 09:31:00 /

前言

winafl 是 afl 在 windows 的移植版， winafl 使用 dynamorio 来统计代码覆盖率，并且使用共享内存的方式让 fuzzer 知道每个测试样本的覆盖率信息。本文主要介绍 winafl 不同于 afl 的部分，对于 afl 的变异策略等部分没有介绍，对于 afl 的分析可以看

```
https://paper.seebug.org/496/#arithmetic
```

源码分析

winafl 主要分为两个部分 afl-fuzz.c 和 winafl.c ， 前者是 fuzzer 的主程序 ，后面的是收集程序运行时信息的 dynamorio 插件的源码。

afl-fuzz

main

winafl 的入口时 afl-fuzz.c ， 其中的 main 函数的主要代码如下

```
int main(int argc, char** argv) {

 // ■■■■■■■■■■■
 setup_post();
 if (!in_bitmap) memset(virgin_bits, 255, MAP_SIZE); // MAP_SIZE --> 0x00010000
 setup_shm();  // ■■■■■■■
 init_count_class16();

 setup_dirs_fds(); // ■■■■■■■■■■■■■■■■■■
 read_testcases();  // ■■■■■■■■■■

 // ■■■■■■■■■■■■■■■ ■■■■■■■■■■
 perform_dry_run(use_argv);

 // ■■■■■■■■
 while (1) {
   u8 skipped_fuzz;
   // ■■■■■■■■■■■■■■■■■■
   cull_queue();

   // ■■■■■■■■■■
   skipped_fuzz = fuzz_one(use_argv);

   queue_cur = queue_cur->next;
   current_entry++;
 }
}
```

- 首先设置一些 fuzz 过程中需要的状态值，比如共享内存、输入输出位置。
- 然后通过 perform_dry_run 把提供的所有测试用例让目标程序跑一遍，同时统计执行过程中的覆盖率信息。
- 之后就开始进行模糊测试的循环，每次取样本出来，然后交给 fuzz_one 对该样本进行 fuzz .

post_handler

该函数里面最重要的就是 fuzz_one 函数， 该函数的作用是完成一个样本的模糊测试，这里面实现了 afl 中的模糊测试策略，使用这些测试策略生成一个样本后，使用采用 common_fuzz_stuff 函数来让目标程序执行测试用例。common_fuzz_stuff 的主要代码如下

```
static u8 common_fuzz_stuff(char** argv, u8* out_buf, u32 len) {

 u8 fault;

 // ■■■■■■■■■■■■■■■■
 if (post_handler) {
```

```
  out_buf = post_handler(out_buf, &len);
  if (!out_buf || !len) return 0;

 }

 write_to_testcase(out_buf, len);

 // ■■■■■■■■■■■■■■■■■■■■■
 fault = run_target(argv, exec_tmout);
```

函数首先会判断是否提供了 post_handler ， 如果提供了 post_handler 就会使用提供的 post_handler 对变异得到的测试数据进行处理，
post_handler 函数指针在 setup_post 函数中设置。

```
static void setup_post(void) {
   HMODULE dh;
   u8* fn = getenv("AFL_POST_LIBRARY"); // ■■■■■■■■ post_handler  ■■ dll ■■■
   u32 tlen = 6;

   if (!fn) return;
   ACTF("Loading postprocessor from '%s'...", fn);
   dh = LoadLibraryA(fn);
   if (!dh) FATAL("%s", dlerror());
   post_handler = (u8* (*)(u8*,u32*))GetProcAddress(dh, "afl_postprocess"); // ■■dll ■■■■■■
   if (!post_handler) FATAL("Symbol 'afl_postprocess' not found.");

   /* Do a quick test. It's better to segfault now than later =) */
   post_handler("hello", &tlen);
   OKF("Postprocessor installed successfully.");
}
```

该函数首先从 AFL_POST_LIBRARY 环境变量里面拿到 post_handler 所在 dll 的路径， 然后设置 post_handler 为 dll 里面的 afl_postprocess
函数的地址。该函数在 fuzzer 运行的开头会调用。 post_handler 的定义如下

```
static u8* (*post_handler)(u8* buf, u32* len);
■■■ buf ■■■■■■■  len ■■■■■■■
■■■■ ■■■■■■■■■■■
```

所以 afl_postprocess 需要接收两个参数， 然后返回一个指向修正后的内存的地址。post_handler
这个机制用于对测试数据的格式做简单的修正，比如计算校验和，计算文件长度等。

run_target

post_handler 这一步过后，会调用 write_to_testcase 先把测试用例写入文件，默认情况下测试用例会写入 .cur_input (用户可以使用 -f 指定)

```
out_file = alloc_printf("%s\\.cur_input", out_dir);
```

然后调用 run_target 让目标程序处理测试用例，其主要代码如下

```
static u8 run_target(char** argv, u32 timeout) {

 // ■■■■■■■■■■■■■■■■■
 if(!is_child_running()) {
   destroy_target_process(0);
   create_target_process(argv);  // ■■■■■■■■ dynamorio ■■
   fuzz_iterations_current = 0;
 }

 if (custom_dll_defined)
     process_test_case_into_dll(fuzz_iterations_current);

 child_timed_out = 0;
 memset(trace_bits, 0, MAP_SIZE);

 result = ReadCommandFromPipe(timeout);
 if (result == 'K')
 {
     //a workaround for first cycle in app persistent mode
     result = ReadCommandFromPipe(timeout);
 }
```

```
 // ■ winafl.dll ■■■■■■■■■ ■■■■■■■■■■ P
 if (result != 'P')
 {
     FATAL("Unexpected result from pipe! expected 'P', instead received '%c'\n", result);
 }

 // ■ winafl.dll ■■■■■■■■■
 WriteCommandToPipe('F');

 result = ReadCommandFromPipe(timeout);
 // ■■■■ K ■■■■■■■■■■■
 if (result == 'K') return FAULT_NONE;

 if (result == 'C') {
     destroy_target_process(2000);
     return FAULT_CRASH;
 }

 destroy_target_process(0);
 return FAULT_TMOUT;
}
```

首先会去判断目标进程是否还处于运行状态，如果不处于运行状态就新建目标进程，因为在 `fuzz` 过程中为了提升效率 ，会使用 `dynamorio` 来让目标程序不断的运行指定的函数，所以不需要每次 `fuzz` 都起一个新的进程。

然后如果需要使用用户自定义的方式发送数据。 就会使用 `process_test_case_into_dll` 发送测试用例，比如 `fuzz` 的目标是网络应用程序。

```
static int process_test_case_into_dll(int fuzz_iterations)
{

 char *buf = get_test_case(&fsize);

 result = dll_run_ptr(buf, fsize, fuzz_iterations); /* caller should copy the buffer */

 free(buf);

 return 1;
}
```

这个 `dll_run_ptr` 在用户通过 `-l` 提供了`dll` 的路径后，`winafl` 会通过 `load_custom_library` 设置相关的函数指针

```
void load_custom_library(const char *libname)
{
 int result = 0;
 HMODULE hLib = LoadLibraryA(libname);
 dll_init_ptr = (dll_init)GetProcAddress(hLib, "_dll_init@0");

 dll_run_ptr = (dll_run)GetProcAddress(hLib, "_dll_run@12");
}
```

`winafl` 自身也提供了[两个示例](#)分别是 `tcp` 服务和 `tcp` 客户端。在 `dll_run_ptr` 中也可以实现一些协议的加解密算法，这样就可以 `fuzz` 数据加密的协议了。

在一切准备好以后 `winafl` 往命名管道里面写入 `F` ，通知 `winafl.dll`（`winafl` 中实现代码覆盖率获取的dynamorio 插件）运行测试用例并记录覆盖率信息。 `winafl.dll` 执行完目标函数后会通过命名管道返回一些信息， 如果返回 `K` 表示用例没有触发异常，如果返回 `C` 表明用例触发了异常。

在 `run_target` 函数执行完毕之后， `winafl` 会对用例的覆盖率信息进行评估，然后更新样本队列。

winafl.c

这个文件里面包含了 `winafl` 实现的 `dynamorio` 插件，里面实现覆盖率搜集以及一些模糊测试的效率提升机制。

dr_client_main

该文件的入口函数是 `dr_client_main`

```
DR_EXPORT void
dr_client_main(client_id_t id, int argc, const char *argv[])
{
```

```
    drmgr_init();
    drx_init();
    drreg_init(&ops);
    drwrap_init();

    options_init(id, argc, argv);

    dr_register_exit_event(event_exit);

    drmgr_register_exception_event(onexception);

    if(options.coverage_kind == COVERAGE_BB) {
        drmgr_register_bb_instrumentation_event(NULL, instrument_bb_coverage, NULL);
    } else if(options.coverage_kind == COVERAGE_EDGE) {
        drmgr_register_bb_instrumentation_event(NULL, instrument_edge_coverage, NULL);
    }

    drmgr_register_module_load_event(event_module_load);
    drmgr_register_module_unload_event(event_module_unload);
    dr_register_nudge_event(event_nudge, id);

    client_id = id;
    if (options.nudge_kills)
        drx_register_soft_kills(event_soft_kill);

    if(options.thread_coverage) {
        winafl_data.fake_afl_area = (unsigned char *)dr_global_alloc(MAP_SIZE);
    }

    if(!options.debug_mode) {
        setup_pipe();
        setup_shmem();
    } else {
        winafl_data.afl_area = (unsigned char *)dr_global_alloc(MAP_SIZE);
    }

    if(options.coverage_kind == COVERAGE_EDGE || options.thread_coverage || options.dr_persist_cache) {
        winafl_tls_field = drmgr_register_tls_field();
        if(winafl_tls_field == -1) {
            DR_ASSERT_MSG(false, "error reserving TLS field");
        }
        drmgr_register_thread_init_event(event_thread_init);
        drmgr_register_thread_exit_event(event_thread_exit);
    }

    event_init();
}
```

函数的主要逻辑如下

- 首先会初始化一些 dynamorio 的信息，
  然后根据用户的参数来选择是使用基本块覆盖率（instrument_bb_coverage）还是使用边覆盖率(instrument_edge_coverage)。
- 然后再注册一些事件的回调。
- 之后就是设置命名管道和共享内存以便和 afl-fuzz 进行通信。

覆盖率记录

通过 drmgr_register_bb_instrumentation_event
我们就可以在每个基本块执行之前调用我们设置回调函数。这时我们就可以统计覆盖率信息了。具体的统计方式如下：

instrument_bb_coverage 的方式

```
// ■■■■■■■■■■■■  MAP_SIZE ■■■■ ■■■■■■■■■
offset = (uint)(start_pc - mod_entry->data->start);
offset &= MAP_SIZE - 1; // ■■■■■■■ map■
afl_map[offset]++
```

instrument_edge_coverage 的方式

```
offset = (uint)(start_pc - mod_entry->data->start);
offset &= MAP_SIZE - 1; // ■■■■■■■ map■
afl_map[pre_offset ^ offset]++
pre_offset = offset >> 1
```

afl_map 适合 afl-fuzz 共享的内存区域， afl-fuzz 和 winafl.dll 通过 afl_map 来传递覆盖率信息。

效率提升方案

在 event_module_load会在每个模块被加载时调用，这个函会根据用户的参数为指定的目标函数设置一些回调函数，用来提升模糊测试的效率。主要代码如下：

```
static void
event_module_load(void *drcontext, const module_data_t *info, bool loaded)
{

   if(options.fuzz_module[0]) {
       if(strcmp(module_name, options.fuzz_module) == 0) {
           if(options.fuzz_offset) {
               to_wrap = info->start + options.fuzz_offset;
           } else {
               //first try exported symbols
               to_wrap = (app_pc)dr_get_proc_address(info->handle, options.fuzz_method);
               if(!to_wrap) {

                   DR_ASSERT_MSG(to_wrap, "Can't find specified method in fuzz_module");
                   to_wrap += (size_t)info->start;
               }
           }
           if (options.persistence_mode == native_mode)
           {
               drwrap_wrap_ex(to_wrap, pre_fuzz_handler, post_fuzz_handler, NULL, options.callconv);
           }
           if (options.persistence_mode == in_app)
           {
               drwrap_wrap_ex(to_wrap, pre_loop_start_handler, NULL, NULL, options.callconv);
           }
       }

   module_table_load(module_table, info);
}
```

在找到 target_module 中的 target_method 函数后，根据是否启用 persistence 模式，采用不同的方式给 target_method 函数设置一些回调函数，默认情况下是不启用 persistence 模式 ， persistence 模式要求目标程序里面有不断接收数据的循环，比如一个 TCP 服务器，会循环的接收客户端的请求和数据。下面分别分析两种方式的源代码。

不启用 persistence

会调用

```
drwrap_wrap_ex(to_wrap, pre_fuzz_handler, post_fuzz_handler, NULL, options.callconv);
```

这个语句的作用是在目标函数 to_wrap 执行前调用 pre_fuzz_handler 函数， 在目标函数执行后调用 post_fuzz_handler 函数。

下面具体分析

```
static void
pre_fuzz_handler(void *wrapcxt, INOUT void **user_data)
{
   char command = 0;
   int i;
   void *drcontext;

   app_pc target_to_fuzz = drwrap_get_func(wrapcxt);
   dr_mcontext_t *mc = drwrap_get_mcontext_ex(wrapcxt, DR_MC_ALL);
   drcontext = drwrap_get_drcontext(wrapcxt);

   // ■■■■■■■■ ■■■■ ■ pc ■■■■ ■■■■■■■■■■■■■■■■■■■■
   fuzz_target.xsp = mc->xsp;
   fuzz_target.func_pc = target_to_fuzz;

   if(!options.debug_mode) {
```

```
        WriteCommandToPipe('P');
        command = ReadCommandFromPipe();

        // ■■ afl-fuzz ■■ F ■ ■■ F ■■■■ fuzzing
        if(command != 'F') {
            if(command == 'Q') {
                dr_exit_process(0);
            } else {
                DR_ASSERT_MSG(false, "unrecognized command received over pipe");
            }
        }
    } else {
        debug_data.pre_hanlder_called++;
        dr_fprintf(winafl_data.log, "In pre_fuzz_handler\n");
    }

    //save or restore arguments■ ■■■■■■■■■■■ ■■■■■■■■■■■■
    if (!options.no_loop) {
        if (fuzz_target.iteration == 0) {
            for (i = 0; i < options.num_fuz_args; i++)
                options.func_args[i] = drwrap_get_arg(wrapcxt, i);
        } else {
            for (i = 0; i < options.num_fuz_args; i++)
                drwrap_set_arg(wrapcxt, i, options.func_args[i]);
        }
    }

    memset(winafl_data.afl_area, 0, MAP_SIZE);

    // ■ ■■■■■■■■■ tls ■■■■ ■■■■■■■■■■■■
    if(options.coverage_kind == COVERAGE_EDGE || options.thread_coverage) {
        void **thread_data = (void **)drmgr_get_tls_field(drcontext, winafl_tls_field);
        thread_data[0] = 0;
        thread_data[1] = winafl_data.afl_area;
    }
}
```

- 首先保存一些上下文信息，比如寄存器信息，然后通过命名管道像 afl-fuzz 发送 P 表示这边已经准备好了可以执行用例，然后等待 afl-fuzz 发送 F 后，就继续向下执行。
- 然后如果是第一次执行，就保存函数的参数，否则就把之前保存的参数设置好。
- 然后重置表示代码覆盖率的共享内存区域。

然后在 post_fuzz_handle 会根据执行的情况向 afl-fuzz
返回执行信息，然后根据情况判断是否恢复之前保存的上下文信息，重新准备开始执行目标函数。通过这种方式可以不用每次执行都新建一个进程，提升了 fuzz 的效率。

```
static void
post_fuzz_handler(void *wrapcxt, void *user_data)
{
    dr_mcontext_t *mc;
    mc = drwrap_get_mcontext(wrapcxt);

    if(!options.debug_mode) {
        WriteCommandToPipe('K');  // ■■■■■■■■■■ K ■ fuzz
    } else {
        debug_data.post_handler_called++;
        dr_fprintf(winafl_data.log, "In post_fuzz_handler\n");
    }

    /*
        We don't need to reload context in case of network-based fuzzing.
        ■■■■■■■ fuzz , ■■■reload.■■■■■■■■■■■■■■■
    */
    if (options.no_loop)
        return;

    fuzz_target.iteration++;
    if(fuzz_target.iteration == options.fuzz_iterations) {
        dr_exit_process(0);
```

```
    }

    // ■■ ■■■■ ■ pc ■■■■■■■■■■■■■■■
    mc->xsp = fuzz_target.xsp;
    mc->pc = fuzz_target.func_pc;
    drwrap_redirect_execution(wrapcxt);
}
```

启用 persistence

在 fuzz 网络应用程序时，应该使用该模式

```
-persistence_mode in_app
```

在这个模式下，对目标函数的包装就没有 pre_fuzz.... 和 post_fuzz..... 了， 此时就是在每次运行到目标函数就清空覆盖率，
因为程序自身会不断的调用目标函数。

```
/* ■■■■■■■■■■■■ aflmap■ ■■■■■■■■■■■■■■■■■■■■ */
static void
pre_loop_start_handler(void *wrapcxt, INOUT void **user_data)
{
    void *drcontext = drwrap_get_drcontext(wrapcxt);

    if (!options.debug_mode) {
        //let server know we finished a cycle, redundunt on first cycle.
        WriteCommandToPipe('K');

        if (fuzz_target.iteration == options.fuzz_iterations) {
            dr_exit_process(0);
        }
        fuzz_target.iteration++;

        //let server know we are starting a new cycle
        WriteCommandToPipe('P');

        //wait for server acknowledgement for cycle start
        char command = ReadCommandFromPipe();

        if (command != 'F') {
            if (command == 'Q') {
                dr_exit_process(0);
            }
            else {
                char errorMessage[] = "unrecognized command received over pipe: ";
                errorMessage[sizeof(errorMessage)-2] = command;
                DR_ASSERT_MSG(false, errorMessage);
            }
        }
    }
    else {
        debug_data.pre_hanlder_called++;
        dr_fprintf(winafl_data.log, "In pre_loop_start_handler\n");
    }

    memset(winafl_data.afl_area, 0, MAP_SIZE);

    if (options.coverage_kind == COVERAGE_EDGE || options.thread_coverage) {
        void **thread_data = (void **)drmgr_get_tls_field(drcontext, winafl_tls_field);
        thread_data[0] = 0;
        thread_data[1] = winafl_data.afl_area;
    }
}
```

总结

通过对 afl-fuzz.c 的分析，我们知道 winafl 提供了两种有意思的功能，即数据修正功能 和
自定义数据发送功能。这两种功能可以辅助我们对一些非常规目标进行 fuzz, 比如网络协议、数据加密应用。通过对 winafl.c 可以清楚的知道如何使用 dynamorio 统计程序的覆盖率， 并且明白了 winafl 通过多次在内存中执行目标函数来提升效率的方式， 同时也清楚了在程序内部自带循环调用函数时，可以使用 persistence 模式来对目标进行 fuzz，比如一些网络服务应用。

参考

https://paper.seebug.org/496/#arithmetic

http://riusksk.me/2019/02/02/winafl%E4%B8%AD%E5%9F%BA%E4%BA%8E%E6%8F%92%E6%A1%A9%E7%9A%84%E8%A6%86%E7%9B%96%E7%8E%87%E5

https://paper.seebug.org/323/#3-winafl-fuzzer

上一篇：内核漏洞挖掘技术系列(4)——sy... 下一篇：CVE-2019-9740 Pyt...

1. 0 条回复
   - 动动手指，沙发就是你的了！