

# 初识 Fuzzing 工具 WinAFL

 paper.seebug.org/323

作者: xd0ol1(知道创宇404实验室)

## 0 引子

本文前两节将简要讨论 fuzzing 的基本理念以及 WinAFL 中所用到的插桩框架 DynamoRIO, 而后我们从源码和工具使用角度带你了解这个适用于 Windows 平台的 fuzzing 利器。

## 1 Fuzzing 101

就 fuzzing 而言, 它是一种将无效、未知以及随机数据作为目标程序输入的自动化或半自动化软件测试技术, 现而今大多被用在漏洞的挖掘上, 其最基本的实现方案如下图所示, 虽然看着不复杂, 但在实际应用中却并非易事:

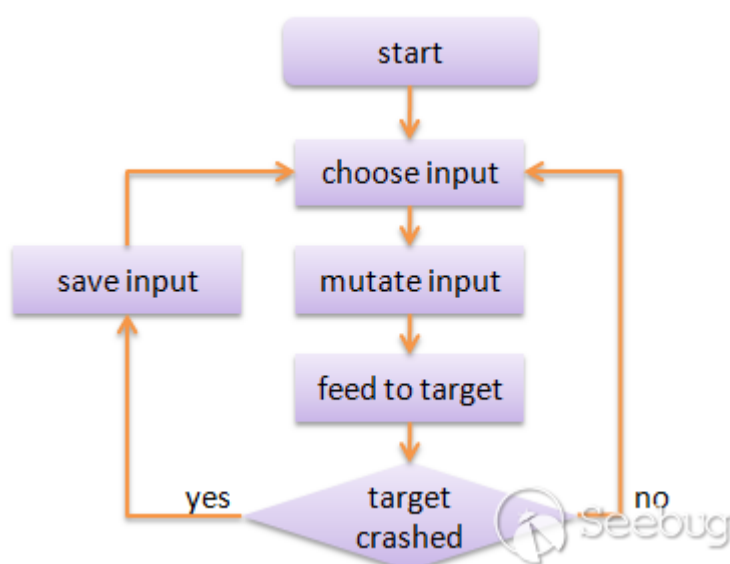


图0 基本的fuzzing实现方案

按输入用例获取方式的不同, 一般可分为基于突变的 dumb fuzzing、基于生成的 smart fuzzing 和基于进化算法的 fuzzing, 前两类相对比较成熟了, 而第三类仍将是今后发展的主要方向。其中, 基于进化算法的 fuzzing 会借助目标程序的反馈来不断完善测试用例, 这就要求在设计时给出相关的评估策略, 最常见的是以程序运行时的代码覆盖率作为衡量标准。

当然, fuzzer 的设计不应局限在相关理论的原型证明上, 关键得经过实践证明才能算是真正有效的。

## 2 DynamoRIO 动态二进制插桩

我们再来看下后文涉及的插桩, DBI (Dynamic Binary Instrumentation) 是一种通过注入探针代码实现二进制程序动态分析的技术, 这些插桩代码会被当作正常的指令来执行。常见的此类框架包括 PIN、Valgrind、DynamoRIO 等, 这里我们要关注的是 DynamoRIO。

通过 DynamoRIO, 我们可以监控程序的运行代码, 同时它还允许我们对运行的代码进行修改。准确来说, DynamoRIO 就相当于一个进程虚拟机, 被监控程序的所有代码都被转移到其上的缓冲区空间中模拟执行, 具体架构如下:

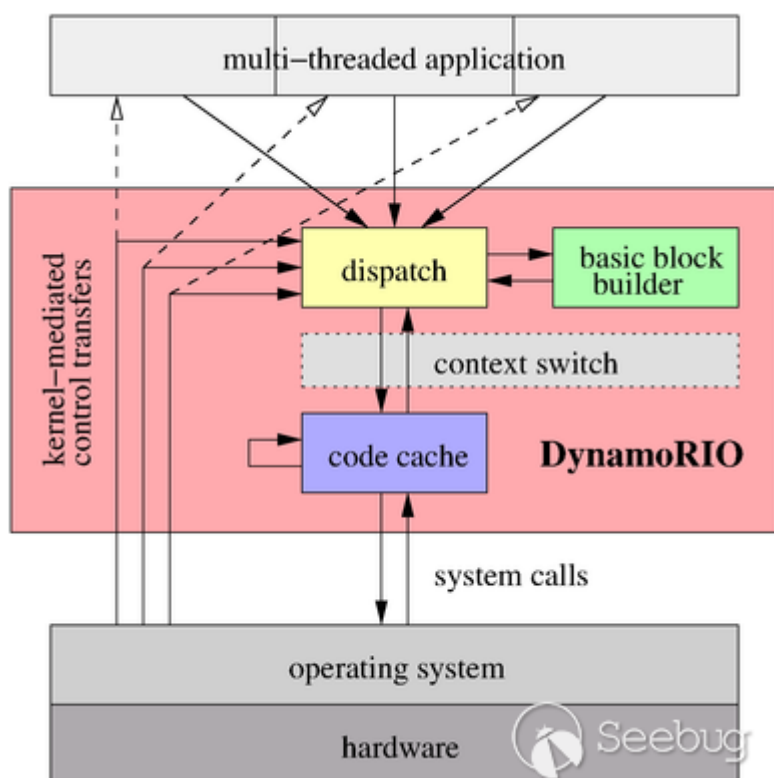


图1 DynamoRIO的架构设计

其中，基本块（basic block）是一个重要的概念。想象一下，将监控进程中的所有指令以控制转移类指令为边界进行分割，那么它们会被分割成许许多多的块，这些块以某一指令开始，但都是以控制转移类指令结束的，如下图：

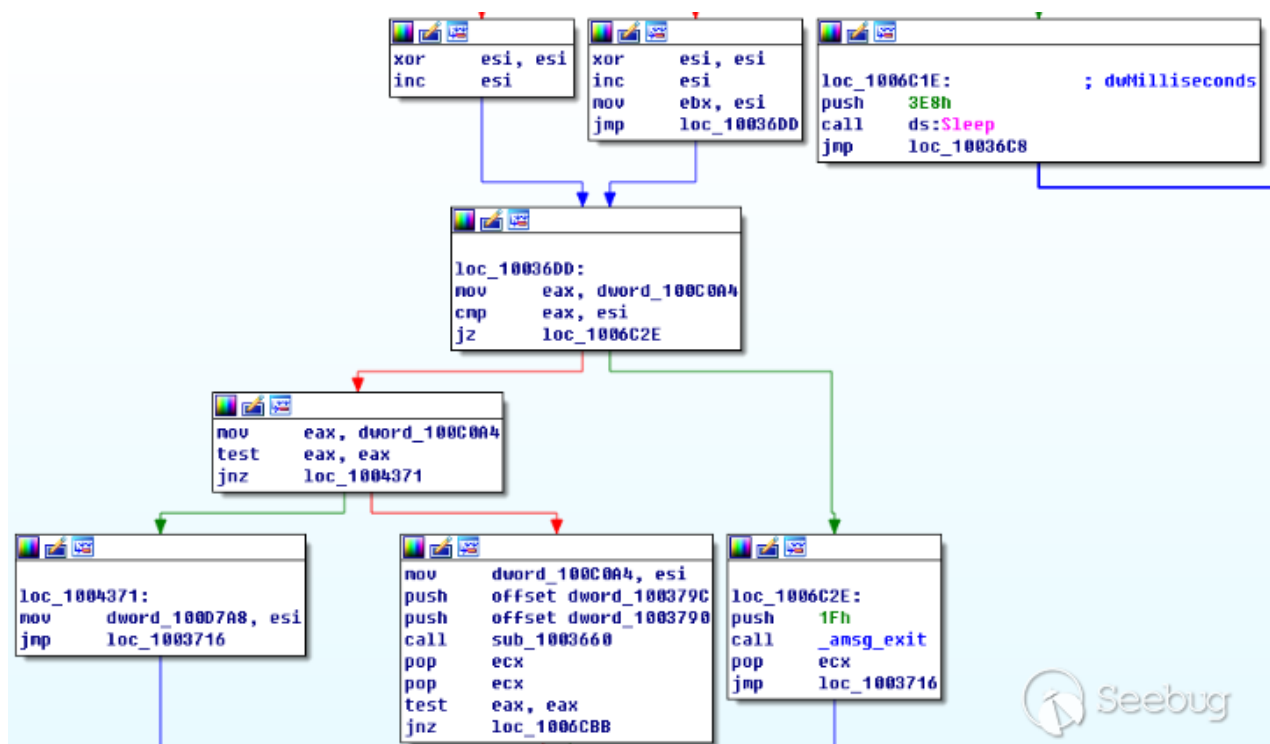


图2 基本块（basic block）的概念

这些指令块就是 DynamoRIO 中定义的基本块概念，即运行的基本单元。DynamoRIO 每次会模拟运行一个基本块中的指令，当这些指令运行完成后，将会通过上下文切换到另一基本块中运行，如此往复，直至被监控进程运行结束。

此外，该框架还为我们提供了丰富的函数编程接口，可以很方便的进行插件（client）开发，主要依赖于各种事件回调处理，同时做好指令过滤对提升性能也是很有帮助的。

### 3 WinAFL Fuzzer

接下去我们就来看下本文的重点，即 WinAFL 这个具体的 fuzzer，本节内容分为3块，首先是概述部分，而后会对此工具的关键源码进行分析，最后我们将借助构造好的存在漏洞的程序进行一次实际 fuzzing。

#### 3.1 概述

对于 fuzzer 来说，AFL（American Fuzzy Lop）想必大家是不会陌生的，但由于其代码设计的原因使得它并不支持 Windows 平台，而 WinAFL 项目正是此 fuzzer 在 Windows 平台下的移植。AFL 借助编译时插桩和遗传算法实现其功能，由于平台支持的关系，在 WinAFL 中该编译时插桩被替换成了 DynamoRIO 动态插桩，此外还基于 Windows API 对相关函数进行了重写。

在使用 WinAFL 进行 fuzzing 时需要指定目标程序及对应的输入测试用例文件，且必须存在这么一个用于插桩的目标函数，此函数的执行过程中包括了打开和关闭输入文件以及对该文件的解析，这样在插桩处理后能够保证目标程序循环的执行文件 fuzzing，避免每次 fuzzing 操作都重新创建新的目标进程。同时，fuzzing 的输入文件会按照相应算法进行变换，且根据得到的目标模块覆盖率判断其是否被用于后续的 fuzzing 操作。

#### 3.2 关键源码分析

我们这里分析的 WinAFL 版本为 1.08，可从 GitHub 上获取。其中 afl\_docs 目录包含了关于设计原理、技术细节等相关说明文档，bin 目录则存放有已经编译好的相关程序，而 testcases 目录是各种测试用例文件，剩下的大部分是源码文件。总体来看，与源码相关的文件实际上不多，代码量在10k+左右，最关键的是 `afl-fuzz.c` 和 `winafl.c` 两个文件，这也是我们主要分析的。此外源码中还包括了一些辅助工具，例如显示跟踪位图信息的 `afl-showmap.c` 以及用于测试用例文件集合最小化的 `winafl-cmin.py`，而用于测试用例文件最小化的 `afl-tmin` 工具目前尚未被移植到该平台。当然，更多设计相关的说明还是具体参考 `technical_details.txt` 文件。

##### 3.2.1 fuzzer模块

我们先看下 `afl-fuzz.c`，此部分代码实现了 fuzzer 的功能，对于 fuzzing 中用到的输入测试文件，程序将使用结构体 `queue_entry` 链表进行维护，我们可在输出结果目录找到相应的 `queue` 文件夹，如下是添加测试用例的代码片段：

```
649  /* Append new test case to the queue. */
650
651  static void add_to_queue(u8* fname, u32 len, u8 passed_det) {
652
653      struct queue_entry* q = ck_alloc(sizeof(struct queue_entry));
654
655      q->fname      = fname;
656      q->len        = len;
657      q->depth      = cur_depth + 1;
658      q->passed_det = passed_det;
```




图3 添加新的测试文件

而输入文件的 fuzzing 则由 `fuzz_one` 函数来完成，此过程涵盖了多个阶段，包括位翻转、算术运算、整数插入这些确定性的 fuzzing 策略以及其它一些非确定性的 fuzzing 策略。且 fuzzing 中采用的突变方式和程序状态并不存在什么特殊关联，表面看该步骤完全是盲目的：

```

264  /* Fuzzing stages */
265
266  enum {
267      /* 00 */ STAGE_FLIP1,
268      /* 01 */ STAGE_FLIP2,
269      /* 02 */ STAGE_FLIP4,
270      /* 03 */ STAGE_FLIP8,
271      /* 04 */ STAGE_FLIP16,
272      /* 05 */ STAGE_FLIP32,
273      /* 06 */ STAGE_ARITH8,
274      /* 07 */ STAGE_ARITH16,
275      /* 08 */ STAGE_ARITH32,
276      /* 09 */ STAGE_INTEREST8,
277      /* 10 */ STAGE_INTEREST16,
278      /* 11 */ STAGE_INTEREST32,
279      /* 12 */ STAGE_EXTRAS_UO,
280      /* 13 */ STAGE_EXTRAS_UI,
281      /* 14 */ STAGE_EXTRAS_AO,
282      /* 15 */ STAGE_HAVOC,
283      /* 16 */ STAGE_SPLICE
284  };

```

```

4615  /* Take the current entry from the queue, fuzz it for a while. This
4616      function is a tad too long... returns 0 if fuzzed successfully, 1 if
4617      skipped or bailed out. */
4618
4619  static u8 fuzz_one(char** argv) {
4620
4621      s32 len, fd, temp_len, i, j;
4622      u8 *in_buf, *out_buf, *orig_in, *ex_tmp, *eff_map = 0;
4623      u64 havoc_queued, orig_hit_cnt, new_hit_cnt;
4624      u32 splice_cycle = 0, perf_score = 100, orig_perf, prev_cksum, eff_cnt = 1;
4625
4626      u8 ret_val = 1;
4627
4628      u8 a_collect[MAX_AUTO_EXTRA];
4629      u32 a_len = 0;

```

图4 测试文件的fuzzing

对上述的每个 fuzzing 策略，程序首先需要对测试用例做相应的修改，然后运行目标程序并处理得到的fuzzing结果：

```

4275  /* Write a modified test case, run program, process results. Handle
4276      error conditions, returning 1 if it's time to bail out. This is
4277      a helper function for fuzz_one(). */
4278
4279  static u8 common_fuzz_stuff(char** argv, u8* out_buf, u32 len) {
4280
4281      u8 fault;
4282
4283      if (post_handler) {
4284
4285          out_buf = post_handler(out_buf, &len);
4286          if (!out_buf || !len) return 0;
4287      }
4288
4289      write_to_testcase(out_buf, len);
4290
4291      fault = run_target(argv);
4292

```

图5 处理每个fuzzing策略

```

743  /* Check if the current execution path brings anything new to the table.
744  Update virgin bits to reflect the finds. Returns 1 if the only change is
745  the hit-count for a particular tuple; 2 if there are new tuples seen.
746  Updates the map, so subsequent calls will always return 0.
747
748  This function is called after every exec() on a fairly large buffer, so
749  it needs to be fast. We do this in 32-bit and 64-bit flavors. */
750
751  #define FFL(_b) (0xffULL << ((_b) << 3))
752  #define FF(_b) (0xff << ((_b) << 3))
753
754  static inline u8 has_new_bits(u8* virgin_map) {
755
756  #ifdef __x86_64__
757
758      u64* current = (u64*)trace_bits;
759      u64* virgin = (u64*)virgin_map;
760
761      u32 i = (MAP_SIZE >> 3);

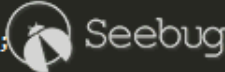
```

当然，在对测试文件进行 fuzzing 前可能还需进行必要的修正：

```

2360 /* Calibrate a new test case. This is done when processing the input directory
2361 | to warn about flaky or otherwise problematic test cases early on; and when
2362 | new paths are discovered to detect variable behavior and so on. */
2363
2364 static u8 calibrate_case(char** argv, struct queue_entry* q, u8* use_mem,
2365 | | | | | | | | | | u32 handicap, u8 from_queue) {
2366
2367     u8 fault = 0, new_bits = 0, var_detected = 0, first_run = (q->exec_cksum == 0);
2368     u64 start_us, stop_us;
2369
2370     s32 old_sc = stage_cur, old_sm = stage_max, old_tmout = exec_tmout;
2371     u8* old_sn = stage_name;


```



```

4149 /* Trim all new test cases to save cycles when doing deterministic checks. The
4150 | trimmer uses power-of-two increments somewhere between 1/16 and 1/1024 of
4151 | file size, to keep the stage short and sweet. */
4152
4153 static u8 trim_case(char** argv, struct queue_entry* q, u8* in_buf) {
4154
4155     static u8 tmp[64];
4156     static u8 clean_trace[MAP_SIZE];
4157
4158     u8 needs_write = 0, fault = 0;
4159     u32 trim_exec = 0;
4160     u32 remove_len;
4161     u32 len_p2;
4162
4163     /* Although the trimmer will be less useful when variable behavior is
4164     | detected, it will still work to some extent, so we don't check
4165     | this. */

```



此外，在 fuzzing 过程中，相关结果的状态信息会不断进行更新，该界面展示是由 `show_stats` 函数实现的：

```

3599  /* A spiffy retro stats screen! This is called every stats_update_freq
3600  | | execve() calls, plus in several other circumstances. */
3601
3602  static void show_stats(void) {
3603
3604      static u64 last_stats_ms, last_plot_ms, last_ms, last_execs;
3605      static double avg_exec;
3606      double t_byte_ratio;
3607
3608      u64 cur_ms;
3609      u32 t_bytes, t_bits;
3610
3611      u32 banner_len, banner_pad;
3612      u8 tmp[256];
3613
3614      cur_ms = get_cur_time();
3615

```



图8 实现fuzzing过程的界面展示

### 3.2.2 插桩模块

下面继续来看 `winafl.c`，此文件对应编写的 DynamoRIO 插件代码，它有两个作用：

1. 循环调用 fuzzing 的目标函数；
2. 更新覆盖率相关的位图文件信息。

程序首先会进行初始化操作并注册各类事件回调函数，其中最重要的是基本块处理事件和模块加载事件：

```

749  DR_EXPORT void
750  dr_client_main(client_id_t id, int argc, const char *argv[])
751  {
752      drreg_options_t ops = {sizeof(ops), 2 /*max slots needed: aflags*/, false};
753
754      dr_set_client_name("WinAFL", "");
755
756      drmgr_init();
757      drx_init();
758      drreg_init(&ops);
759      drwrap_init();
760
761      options_init(id, argc, argv);
762
763      dr_register_exit_event(event_exit);
764
765      drmgr_register_exception_event(onexception);
766
767      if(options.coverage_kind == COVERAGE_BB) {
768          drmgr_register_bb_instrumentation_event(NULL, instrument_bb_coverage, NULL);
769      } else if(options.coverage_kind == COVERAGE_EDGE) {
770          drmgr_register_bb_instrumentation_event(NULL, instrument_edge_coverage, NULL);
771      }
772
773      drmgr_register_module_load_event(event_module_load);
774      drmgr_register_module_unload_event(event_module_unload);
775      dr_register_nudge_event(event_nudge, id);

```



图9 注册各类事件回调函数

在相应的模块加载事件回调函数中，如果当前模块为 fuzzing 的目标模块，那么会对其中相应的目标函数进行插桩处理：



```

515     if(options.fuzz_module[0]) {
516         if(strcmp(module_name, options.fuzz_module) == 0) {
517             if(options.fuzz_offset) {
518                 to_wrap = info->start + options.fuzz_offset;
519             } else {
520                 //first try exported symbols
521                 to_wrap = (app_pc)dr_get_proc_address(info->handle, options.fuzz_method);
522                 if(!to_wrap) {
523                     //if that fails, try with the symbol access library
524                     drsym_init(0);
525                     drsym_lookup_symbol(info->full_path, options.fuzz_method, (size_t *)&to_wrap, 0);
526                     drsym_exit();
527                     DR_ASSERT_MSG(to_wrap, "Can't find specified method in fuzz_module");
528                     to_wrap += (size_t)info->start;
529                 }
530             }
531             drwrap_wrap_ex(to_wrap, pre_fuzz_handler, post_fuzz_handler, NULL, options.fuzz_offset);
532         }

```

图10 对目标函数进行插桩

即在目标函数执行前，通过 `pre_fuzz_handler` 调用记录下当前的寄存器环境，而在目标函数执行后，又会通过 `post_fuzz_handler` 调用进行寄存器环境的恢复，从而实现了待 fuzzing 目标函数的不断循环：

```

460     } else {
461         debug_data.post_handler_called++;
462         dr_fprintf(winafl_data.log, "In post_fuzz_handler\n");
463     }
464
465     fuzz_target.iteration++;
466     if(fuzz_target.iteration == options.fuzz_iterations) {
467         dr_exit_process(0);
468     }
469
470     mc->xsp = fuzz_target.xsp;
471     mc->pc = fuzz_target.func_pc;
472
473     drwrap_redirect_execution(wrapcxt);
474 }

```

图11 恢复寄存器环境

此外另一关键问题是对位图文件的处理，关于位图文件的覆盖率计算有两种模式，即基本块

(basic block) 覆盖率模式和边界 (edge) 覆盖率模式。在 fuzzing 过程中会维护一个64KB大小的位图文件用于记录此覆盖率及其命中次数，在边界覆盖率模式下每个字节代表了特定的源地址和目标地址配对，这种模式更有助于形象化表述程序的执行流程，因为漏洞往往是由未知的或非正常的执行状态转换导致的，而非简单的基本块覆盖。对应的事件函数为 `instrument_bb_coverage` 和 `instrument_edge_coverage`，也就是注册的基本块处理回调函数，位图文件的更新是通过插入的新增指令来实现的，对于边界覆盖率的情况其代码如下，相应基本块覆盖率的情形与之类似：

```

352 //load the pointer to previous offset in reg3
353 drmgr_insert_read_tls_field(drcontext, winaf1_tls_field, bb, inst, reg3);
354
355 //load address of shm into reg2
356 if(options.thread_coverage) {
357     opnd1 = opnd_create_reg(reg2);
358     opnd2 = OPND_CREATE_MEMPTR(reg3, sizeof(void *));
359     new_instr = INSTR_CREATE_mov_ld(drcontext, opnd1, opnd2);
360     instrlist_meta_preinsert(bb, inst, new_instr);
361 } else {
362     opnd1 = opnd_create_reg(reg2);
363     opnd2 = OPND_CREATE_INTPTR((uint64)winaf1_data.afl_area);
364     new_instr = INSTR_CREATE_mov_imm(drcontext, opnd1, opnd2);
365     instrlist_meta_preinsert(bb, inst, new_instr);
366 }
367
368 //load previous offset into register
369 opnd1 = opnd_create_reg(reg);
370 opnd2 = OPND_CREATE_MEMPTR(reg3, 0);
371 new_instr = INSTR_CREATE_mov_ld(drcontext, opnd1, opnd2);
372 instrlist_meta_preinsert(bb, inst, new_instr);
373
374 //xor register with the new offset
375 opnd1 = opnd_create_reg(reg);
376 opnd2 = OPND_CREATE_INT32(offset);
377 new_instr = INSTR_CREATE_xor(drcontext, opnd1, opnd2);
378 instrlist_meta_preinsert(bb, inst, new_instr);
379
380 //increase the counter at reg + reg2
381 opnd1 = opnd_create_base_disp(reg2, reg, 1, 0, OPSZ_1);
382 new_instr = INSTR_CREATE_inc(drcontext, opnd1);
383 instrlist_meta_preinsert(bb, inst, new_instr);
384
385 //store the new value
386 offset = (offset >> 1)&(MAP_SIZE - 1);
387 opnd1 = OPND_CREATE_MEMPTR(reg3, 0);
388 opnd2 = OPND_CREATE_INT32(offset);
389 new_instr = INSTR_CREATE_mov_st(drcontext, opnd1, opnd2);
390 instrlist_meta_preinsert(bb, inst, new_instr);

```

图12 插入更新边界覆盖率的指令

### 3.3 WinAFL 的使用

最后我们来进行一次实际的 fuzzing，用到的目标程序是基于所给的 gdiplus.cpp 源码修改得到的，其中手动引入了一个 crash，代码如下：

```

int (*func)(int x); //定义func函数指针
.....
func = NULL;
printf("%d", func(0)); //程序crash

```

首先我们需要确定 fuzzing 的目标函数，即设置 `-target_offset` 或 `-target_method` 对应的参数。在此例中 main 函数是符合条件的目标函数，若要使用 `-target_offset`，则可简单通过 IDA 来查看此函数的偏移，此例中为 `0x1090`：

```

00401090 ; ===== S U B R O U T I N E =====
00401090
00401090 ; Attributes: bp-based frame
00401090
00401090 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401090 _main proc near ; CODE XREF: __tmainCRTStartup+1D1p
00401090

```



图13 查看main函数的偏移

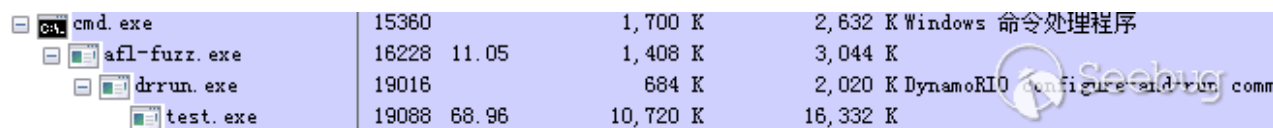
如果存在符号文件，那么可以直接设置 `-target_method` 的参数为main。对于 `-coverage_module` 的参数，我们可以执行如下命令来获取，注意 DynamoRIO 的目录需根据实际情况来设置。在得到的 log 文件中给出了目标程序执行过程中所加载的模块，同时，必须保证运行结果为“Everything appears to be running normally.”：

```
C:\temp\DynamoRIO\bin32\drun.exe -c winafll.dll -debug -target_module test.exe -target_offset 0x1090 -fuzz_iterations 10 -nargs 2 -- test.exe in\input.bmp
```

然后，我们就可以输入如下的命令进行 fuzzing 了，其中“@@”表示待 fuzzing 的测试用例文件在 in 目录下：

```
afl-fuzz.exe -i in -o out -D C:\temp\DynamoRIO\bin32 -t 20000 -- -coverage_module gdiplus.dll -coverage_module WindowsCodecs.dll -fuzz_iterations 5000 -target_module test.exe -target_method main -nargs 2 -- test.exe @@
```

但上述命令参数中并没有出现 DynamoRIO 插件 winafll.dll，事实上此命令执行后又创建了新的子进程，如下图：



cmd.exe	15360		1,700 K	2,632 K	Windows 命令处理程序
afl-fuzz.exe	16228	11.05	1,408 K	3,044 K	
drun.exe	19016		684 K	2,020 K	DynamoRIO
test.exe	19088	68.96	10,720 K	16,332 K	

图14 afl-fuzz进程树

我们可以得到 drun.exe 执行的命令参数如下：

```
C:\temp\DynamoRIO\bin32\drun.exe -pidfile childpid_95fa18fc9031bf0d.txt -no_follow_children -c winafll.dll -coverage_module gdiplus.dll -coverage_module WindowsCodecs.dll -fuzz_iterations 5000 -target_module test.exe -target_method main -nargs 2 -fuzzer_id 95fa18fc9031bf0d -- test.exe out\cur_input
```

如果没问题的话，那么我们会看到如下的 fuzzing 界面，至于 WinAFL 的编译以及其它参数设置可参考 [README](#) 文件：

```

+-----+
WinAFL 1.08 based on AFL 1.96b <test.exe>
+-----+
+- process timing -----+- overall results -----+
|   run time : 0 days, 0 hrs, 28 min, 53 sec   | cycles done : 0      |
| last new path : 0 days, 0 hrs, 3 min, 10 sec | total paths : 65    |
| last uniq crash : 0 days, 0 hrs, 12 min, 8 sec | uniq crashes : 9    |
| last uniq hang : none seen yet              | uniq hangs : 0      |
+- cycle progress -----+- map coverage -----+
| now processing : 5 <7.69%>                  | map density : 4647 <7.09%> |
| paths timed out : 0 <0.00%>                 | count coverage : 1.62 bits/tuple |
+- stage progress -----+ findings in depth -----+
| now trying : havoc                          | favored paths : 33 <50.77%> |
| stage execs : 65.6k/80.0k <81.94%>         | new edges on : 46 <70.77%> |
| total execs : 280k                          | total crashes : 653 <9 unique> |
| exec speed : 183.4/sec                      | total hangs : 0 <0 unique> |
+- fuzzing strategy yields -----+ path geometry -----+
| bit flips : 21/5424, 4/5422, 1/5418        | levels : 3          |
| byte flips : 0/678, 0/676, 1/672           | pending : 64        |
| arithmetics : 9/37.9k, 0/22.1k, 0/24.6k    | pend fav : 33       |
| known ints : 1/2028, 1/12.2k, 1/15.1k      | own finds : 64       |
| dictionary : 0/0, 0/0, 2/2000              | imported : n/a      |
|   havoc : 23/80.0k, 0/0                    | variable : 4         |
|   trim : 46.19%/331, 0.00%                 |                      |
+-----+

```

图15 WinAFL执行时的界面

fuzzing 中各阶段的结果都将保存在 -o 选项设置的 out 目录中，其中 crash 或 hangs 目录保存着导致 bug 的测试用例文件，至于目标程序是否存在可利用的漏洞则需要进一步的确认：

crashes	2017/6/7 12:13	文件夹	
hangs	2017/6/7 11:56	文件夹	
queue	2017/6/7 12:22	文件夹	
.cur_input	2017/6/7 12:25	CUR_INPUT 文件	1 KB
fuzz_bitmap	2017/6/7 12:22	文件	64 KB
fuzzer_stats	2017/6/7 12:24	文件	1 KB
plot_data	2017/6/7 12:22	文件	3 KB

图16 保存fuzzing结果的目录

## 4 结语

本文大体介绍了 WinAFL 这个 fuzzing 工具，但实际应用起来还是有很多方面需要考虑的。另外，笔者目前还是初学，错误之处还望各位斧正，欢迎一起交流:P

## 5 参考

- [1] A fork of AFL for fuzzing Windows binaries
- [2] Dynamic Instrumentation Tool Platform
- [3] American fuzzy lop
- [4] Real World Fuzzing
- [5] Code Coverage
- [6] Effective file format fuzzing

本文由 Seebug Paper 发布，如需转载请注明来源。本文地址：<https://paper.seebug.org/323/>