Using Binary Instrumentation for Vulnerability Discovery (or even mitigation!)

Joxean Koret

Binary Instrumentation and Vulnerability Discovery

1. Introduction
2. Writing a bug finding tool
3. Problems
4. DEMO
5. Examples of bugs found
6. Conclusions

Introduction

Introduction

What is instrumentation?

- According to Wikipedia, it’s the ability to monitor or measure the level of a product's performance, to diagnose errors and to write trace information.

There are 2 types of instrumentation approaches:

- Source instrumentation. Not possible to use for auditing closed sourced products.
- Binary instrumentation. An approach available for any software as long as we can execute it.

Introduction

Binary Instrumentation is often called Dynamic Binary Instrumentation (DBI) as it instruments a program while executing it at binary level.

- Basically, it inserts instrumentation code before and after a piece of code (a basic block) is executed.
- Instrumentation can be inserted at a user-defined granularity level like:
    - Instruction level. For every single assembly instruction.
    - Basic block level. For each basic block that is discovered by the DBI tool.
    - Function level. Only at the beginning, usually, of each function the DBI tool discovered.

Introduction

As DBI lets us go as deep as we want to instrument a program, we can monitor the full execution of a program at whatever granularity level we want. This is great, but it also has some drawbacks:

- A program instrumented with a DBI tool will always run slower than running the program alone.
- Depending on the granularity we choose and the amount of instrumentation inserted, the program can go so slow that DBI turns out to be impossible to use. Or at least turn out to be “not so useful”.
    - Some examples I have tried: Oracle database and modern PC games.

Introduction: Tools

There are various DBI toolkits out there but I’ll just talk about the most commonly used ones:

- DynamoRIO. Open Source and very fast. However, tends to fail for not an easy reason to understand and there are softwares that you simply cannot instrument with it. Besides, the API is neither easy nor pretty, in my opinion.
- Intel PIN. Commercial software with a ridiculous license and very slow. However, the only DBI tool that is reliable and it has a super easy API.

Other DBI tools that I haven’t used yet:

- Frida. Open Source and pretty fast. However, it seems not to support statically compiled binaries.

DynamoRIO

Introduction: DynamoRIO

Example DynamoRIO instrumentation tool (part I):

```
uint num_dyn_instrs;

static void event_init(void);
static void event_exit(void);
static dr_emit_flags_t  event_basic_block(void *drcontext, void *tag, instrlist_t *ilist,
                                          bool for_trace, bool translating);

DR_EXPORT void dr_init(client_id_t id) {
    /* register events */
    dr_register_bb_event  (event_basic_block);
    dr_register_exit_event(event_exit);
    /* process initialization event */
    event_init();
}
```

Introduction: DynamoRIO

Example DynamoRIO instrumentation tool (part II):

```
static void event_init(void) {
    num_dyn_instrs = 0;
}

static void event_exit(void) {
    dr_printf(“Total number of instruction executed: %u\n”, num_dyn_instrs);
}

static dr_emit_flags_t  event_basic_block(void *drcontext, void *tag, instrlist_t *ilist,
                                          bool for_trace, bool translating) {
    int num_instrs;
    num_instrs = ilist_num_instrs(ilist);
    insert_count_code(drcontext, ilist, num_instrs);
    return DR_EMIT_DEFAULT;
}
```

Introduction: DynamoRIO

Example DynamoRIO instrumentation tool (part III):

```
static int ilist_num_instrs(instrlist_t *ilist)  {
    instr_t *instr;
    int         num_instrs = 0;
    /* iterate over instruction list to count number of instructions */
    for (instr = instrlist_first(ilist); instr != NULL; instr = instr_get_next(instr))
        num_instrs++;
    return num_instrs;
}

static void do_ins_count(int num_instrs) { num_dyn_instrs += num_instrs; }

static void insert_count_code(void * drcontext, instrlist_t * ilist, int num_instrs) {
    dr_insert_clean_call(drcontext, ilist, instrlist_first(ilist),
                                    do_ins_count,  false,  1,
                                    OPND_CREATE_INT32(num_instrs));
}
```

Intel PIN

Introduction: Intel PIN

Example Intel PIN instrumentation tool (part I):

```
/* ===================================================================== */
/* Print Help Message                                                    */
/* ===================================================================== */

INT32 Usage()
{
    cerr << "This tool counts the number of dynamic instructions executed" << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

/* ===================================================================== */
/* Main                                                                  */
/* ===================================================================== */
/*   argc, argv are the entire command line: pin -t <toolname> -- ...    */
/* ===================================================================== */

int main(int argc, char * argv[])
{
    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

    OutFile.open(KnobOutputFile.Value().c_str());

    // Register Instruction to be called to instrument instructions
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

Introduction: Intel PIN

Example Intel PIN instrumentation tool (part II):

```
ofstream OutFile;

// The running count of instructions is kept here
// make it static to help the compiler optimize docount
static UINT64 icount = 0;

// This function is called before every instruction is executed
VOID docount() { icount++; }

// Pin calls this function every time a new instruction is encountered
VOID Instruction(INS ins, VOID *v)
{
    // Insert a call to docount before every instruction, no arguments are passed
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "inscount.out", "specify output file name");

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    // Write to a file since cout and cerr maybe closed by the application
    OutFile.setf(ios::showbase);
    OutFile << "Count " << icount << endl;
    OutFile.close();
}
```

Introduction: Intel PIN and DynamoRIO

As we can see, it’s way easier to write an Intel PIN tool than a DynamoRIO tool.

However, the Intel PIN license is a “no go” for many use cases:

- Example: do you want to distribute a binary build? You can’t.

The choice of a DBI toolkit will largely depend on what do you want it for.

- If you want to make a commercial tool or distribute binary versions, use DynamoRIO. If it works in your case.
- If you want to use a FOSS tool, use DynamoRIO. Again, if it works for you.
- For anything else, use Intel Pin.

Introduction: Intel PIN

Because of its stability and reliability, most reverse engineers use Intel PIN.

After all, most of the time you just want something for your specific use case and most likely aren't going to distribute anything because is of no value for others.

And in the case of distributing the tool, most reverse engineers are fine with distributing source code. Even companies doing commercial products:

- For example: IDA's Intel Pin debugger module is distributed in source form.

For the example tool I've written for this talk, I will use Intel PIN.

Writing a bug finding tool

Writing a bug finding tool

DBI tools are very handy for writing bug finding tools. We can instrument at whatever granularity level we want and we can pretty much instrument/monitor anything the application does.

In my case, I have written an Intel PIN tool with the aim of detecting (and even mitigating!) the following common memory management bugs:

- Double free-s.
- Invalid free-s.
- Use-after-free-s.
- Writes to freed memory.

Writing a bug finding tool

What we have to do in order to find such bugs?

- Find the dynamic memory allocation functions and hook them.
- Monitor how the buffers are manipulated.
- Report when a vulnerability is found.
- …
- Profit!

However, is not that simple and little problems appear while coding such tools.

During this talk I will talk about some of them.

Writing a bug finding tool

As I said, we first need to hook (instrument) the allocation functions. Which functions are they?

- Malloc
- Calloc
- Realloc
- Free

How can we do hook these functions using Intel Pin?

- First, we will need to check which module exports these functions.
- And then, just instrument these functions.

Writing a bug finding tool

The memory allocator functions will be “stored” in some specific module.

- Like the libc, in most cases.

We will need to find the appropriate module and hook these functions. How?

Writing a bug finding tool

We will need to install a callback to instrument “image loading”:

```
//---------------------------------------------------------------------------
int main(int argc, char *argv[])
{
  // Initialize symbols
  PIN_InitSymbols();

  // Initialize PIN library. Print help message if -h(elp) is specified
  // in the command line or the command line is invalid
  if ( PIN_Init(argc,argv) )
    return usage();

  g_mitigate = (knob_mitigate.Value() != 0);

  if ( knob_memory_tracker.Value() != 0 )
  {
    g_hooks = new CHooksInstaller();
    g_checker = new CMemoryChecker();
    g_checker->break_always = (knob_break_always.Value() != 0);
  }

  // Register Image to be called to instrument functions.
  IMG_AddInstrumentFunction(image_cbk, 0);
  IMG_AddUnloadFunction(image_unload_cbk, 0);

  // Register function to be called to instrument instructions
  INS_AddInstrumentFunction(ins_cbk, 0);

  // Register function to be called when the application exits
  PIN_AddFiniFunction(fini_cbk, 0);

  // Start the program, never returns
  PIN_StartProgram();

  return 0;
}
```

Writing a bug finding tool

Then, we will need to install our hooks in the appropriate module (part I):

```
//----------------------------------------------------------------------
static VOID image_cbk(IMG img, VOID *v)
{
  if ( g_hooks != NULL )
    g_hooks->install_hooks(img);

  if ( g_checker != NULL )
    g_checker->add_image(img);
}
```

Writing a bug finding tool

Then, we will need to install our hooks in the appropriate module (part II):

```
//-----------------------------------------------------------------------------
void CHooksInstaller::install_hooks(IMG img)
{
  str_vec_t malloc_funcs;
  malloc_funcs.push_back("malloc");
  malloc_funcs.push_back("_malloc");

  str_vec_t calloc_funcs;
  calloc_funcs.push_back("calloc");
  calloc_funcs.push_back("_calloc");

  str_vec_t free_funcs;
  free_funcs.push_back("free");
  free_funcs.push_back("_free");

  str_vec_t realloc_funcs;
  realloc_funcs.push_back("realloc");
  realloc_funcs.push_back("_realloc");

  hook_functions(img, malloc_funcs, FTT_MALLOC);
  hook_functions(img, calloc_funcs, FTT_CALLOC);
  hook_functions(img, free_funcs, FTT_FREE);
  hook_functions(img, realloc_funcs, FTT_REALLOC);

  hook_memory_access();
}
```

Writing a bug finding tool

Then, we will need to install our hooks in the appropriate module (part III):

```
//-------------------------------------------------------
void CHooksInstaller::hook_functions(
  IMG img,
  const str_vec_t &funcs,
  FUNC_TYPE_T type)
{
  str_vec_t::const_iterator it;
  str_vec_t::const_iterator end = funcs.end();
  for ( it = funcs.begin(); it != end; ++it )
  {
    if ( hook_one_function(img, (*it).c_str(), type) )
      break;
  }
}
```

Writing a bug finding tool

Then, we will need to install our hooks in the appropriate module (part IV):

```
//-----------------------------------------------------------------------
bool CHooksInstaller::hook_one_function(
  IMG img,
  const char *func,
  FUNC_TYPE_T type)
{
  bool ret = false;
  RTN target_rtn = RTN_FindByName(img, func);
  if (RTN_Valid(target_rtn))
  {
    RTN_Open(target_rtn);
    RTN_InsertCall(target_rtn, IPOINT_BEFORE, (AFUNPTR)mem_before_cbk,
                   IARG_FAST_ANALYSIS_CALL,
                   IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                   IARG_FUNCARG_ENTRYPOINT_VALUE, 1,
                   IARG_ADDRINT, type,
                   IARG_FUNCARG_ENTRYPOINT_REFERENCE, 0,
                   IARG_END);
    RTN_InsertCall(target_rtn, IPOINT_AFTER, (AFUNPTR)mem_after_cbk,
                   IARG_FAST_ANALYSIS_CALL,
                   IARG_FUNCRET_EXITPOINT_VALUE,
                   IARG_ADDRINT, type,
                   IARG_END);

    RTN_Close(target_rtn);
    ret = true;
  }
  return ret;
}
```

Writing a bug finding tool

In the previous slide we simply search for the specific functions to be hooked (malloc, _malloc, free, _free, etc…) using RTN_FindByName.

If any of these functions is found, we ‘hook’ it by RTN_InsertCall.

In my example tool, I’m doing it *before* and *after* the functions are called.

- The arguments to the functions will only be available before the functions are called.
- The result of the functions will only be available after the functions are called.

Writing a bug finding tool

Then, in the callbacks we just installed we will handle the casuistic for the 2 functions we’re going to support:

- malloc
- free

Well, we’re actually hooking 4 functions (malloc, calloc, realloc and free) but we can consider the other 2 functions just specialized versions of malloc.

Writing a bug finding tool

Handling the (m|c)alloc/free functions from the instrumentation callback (part I):

```
//-----------------------------------------------------------------
static void PIN_FAST_ANALYSIS_CALL mem_before_cbk(
  ADDRINT a1,
  ADDRINT a2,
  FUNC_TYPE_T type,
  ADDRINT *ref_a1)
{
  if ( type == FTT_MALLOC )
  {
    if ( g_checker != NULL )
      g_checker->check_before_malloc(a1, type);
  }
  else if ( type == FTT_CALLOC )
  {
    if ( g_checker != NULL )
      g_checker->check_before_malloc(a1 * a2, type);
  }
  else if ( type == FTT_FREE )
  {
    if ( g_checker != NULL )
      g_checker->check_before_free(a1, a2, type, ref_a1);
  }
```

Writing a bug finding tool

Handling the (m|c)alloc/free functions from the instrumentation callback (part II):

```
//---------------------------------------------------------------------
void CMemoryChecker::check_before_malloc(size_t size, FUNC_TYPE_T type)
{
  PIN_LockClient();

  if ( (signed)size < 0 )
  {
    printf("WARNING! Negative size given to a malloc call!\n");
    add_breakpoint("Negative size given to a malloc call");
  }
  else if ( size == 0 )
  {
    printf("WARNING! Zero allocation detected!\n");
    add_breakpoint("WARNING! Zero allocation detected!\n");
  }

  mem_area_t area;
  area.size = size;
  area.available = true;
  area.ignore = false;
  area.ignore_write = false;
  area.tid = PIN_GetTid();
  areas.push_back(area);

  PIN_UnlockClient();
}
```

Writing a bug finding tool

In the class method CMemoryChecker::check_before_malloc() we're finally adding our first rules to find vulnerabilities.

By just checking the argument "size" (of type size_t) given to malloc/calloc we can find 2 different but related vulnerability types:

- **Negative mallocs**. For example, in 32 bits platforms malloc(-1) will try to allocate 4 GB of memory, which is almost always wrong, will most likely fail and is a clear sign of a vulnerability.
- **Zero allocations**. A zero allocation usually returns a valid pointer with as much memory as the size of a pointer in the specific architecture. If more than the reserved bytes are written to the buffer, a heap overflow happens.

Writing a bug finding tool

One interesting thing in the previously shown code is the calls to add_breakpoint.

This function sets a flag to call the API PIN_ApplicationBreakpoint from an installed instruction level instrumentation callback.

We can connect from GDB, IDA or Visual Studio to the Intel PIN remote debugger and when PIN_ApplicationBreakpoint is called, a breakpoint is triggered in it.

Executions stops exactly at the point where the violation is made and one can check the trace back, arguments, etc...

Writing a bug finding tool

By just hooking malloc and calloc functions we have added support to dynamically discover 2 bug classes.

For discovering use-after-frees, double-frees, etc… we need to do some more things:

- Basically, keep a vector of all the allocations, the memory address returned and their sizes.
- When free is called, check if the pointer is valid, was previously freed, etc...

Let’s see the implementation in my example tool...

Writing a bug finding tool

Finding double frees (part I):

```
//-----------------------------------------------------------------------
void CMemoryChecker::check_before_malloc(size_t size, FUNC_TYPE_T type)
{
  PIN_LockClient();

  if ( (signed)size < 0 )
  {
    printf("WARNING! Negative size given to a malloc call!\n");
    add_breakpoint("Negative size given to a malloc call");
  }
  else if ( size == 0 )
  {
    printf("WARNING! Zero allocation detected!\n");
    add_breakpoint("WARNING! Zero allocation detected!\n");
  }

  mem_area_t area;
  area.size = size;
  area.available = true;
  area.ignore = false;
  area.ignore_write = false;
  area.tid = PIN_GetTid();
  areas.push_back(area);

  PIN_UnlockClient();
}
```

Writing a bug finding tool

Finding double frees (part II):

```
//------------------------------------------------------------------------
void CMemoryChecker::check_after_malloc(ADDRINT ret, FUNC_TYPE_T type)
{
  PIN_LockClient();

  mem_area_vec_t::reverse_iterator it;
  for ( it = areas.rbegin(); it != areas.rend(); ++it )
  {
    if ( it->tid == PIN_GetTid() )
    {
      ADDRINT ea = it->ea;
      if ( ea == ret || (ret >= ea && ret <= (ea + it->size)) )
        areas.erase((it+1).base());
    }
  }

  for ( it = areas.rbegin(); it != areas.rend(); ++it )
  {
    if ( it->tid == PIN_GetTid() )
    {
      if ( ret == 0 )
        areas.erase((it+1).base());
      else
        it->ea = ret;

      break;
    }
  }

  PIN_UnlockClient();
}
```

Writing a bug finding tool

Finding double frees (part III):

```
//------------------------------------------------------------------------------
void CMemoryChecker::check_before_free(ADDRINT a1,
  ADDRINT a2,
  FUNC_TYPE_T type,
  ADDRINT *ref_a1)
{
  PIN_LockClient();

  bool found = false;
  mem_area_vec_t::reverse_iterator it;
  for ( it = areas.rbegin(); it != areas.rend() && a1 != 0; ++it )
  {
    if ( it->tid == PIN_GetTid() && it->ea == a1 )
    {
      found = true;
      if ( !it->available )
      {
        printf("WARNING! Freeing already available memory (0x" EA_FMT " - 0x" EA_FMT ")!\n", it->ea, it->ea + it->size);
#if 0
        dump_areas();
#endif
        if ( break_always )
          add_breakpoint("Freeing already available memory!");

        if ( g_mitigate )
        {
          printf("NOTICE: Mitigation is enabled, returning a null pointer...\n");
          *ref_a1 = 0;
        }

        it->ignore = true;
      }

      it->available = false;
      break;
    }
  }
```

Writing a bug finding tool

In the last code slide we can see how we can detect double frees and stop at the debugger when we detect one by triggering a breakpoint.

But we can even mitigate such a bug!

- Well, kind of. We just send a null pointer to free.
- It’s documented that free will *NOT* fail when given a null pointer and the execution should continue “happily”.
- However, where there is a bug, there can be others, and the application can crash at some other, probably half-random, point.

Writing a bug finding tool

We can also detect other memory related bugs involving calls to free:

- Freeing invalid pointers.

From an Intel PIN tool we can check if an address is valid so, in addition to detect double frees, we can detect invalid frees.

Let us see how...

Writing a bug finding tool

Finding invalid frees:

```
if ( !found && a1 != 0 )
{
  // Trick to avoid false positives: try to read 4 bytes from that
  // memory page we don't have information about.
  char buf[4];
  size_t bytes = PIN_SafeCopy(buf, (ADDRINT*)a1, sizeof(buf));
  if ( bytes < 4 )
  {
    printf("WARNING! Freeing an invalid memory page at 0x" EA_FMT "!\n", a1);
    add_breakpoint("Freeing an invalid memory page");

    if ( g_mitigate )
    {
      printf("NOTICE: Mitigation is enabled, returning a null pointer #2...\n");
      *ref_a1 = 0;
    }
  }
  else
  {
    //printf("WARNING! An unknown memory page has been freed!\n");
    mem_area_t area;
    area.ea = a1;
    area.size = sizeof(void*);
    area.available = false;
    area.ignore = false;
    area.ignore_write = false;
    area.tid = PIN_GetTid();
    areas.push_back(area);
  }
}

// The libc actually needs to write to freed pages so, while inside
// the "free" implementation, ignore any error.
inside_alloc[PIN_ThreadId()] = true;

PIN_UnlockClient();
```

Writing a bug finding tool

As we can see, we can detect invalid frees and, as before, we can even kind of mitigate them.

- Same problems as with double frees apply, though.

Now, time to detect even more bug types: write-after-frees.

- Basically, we will detect when a freed page is being written to.

Let's see how...

Writing a bug finding tool

Detecting writes to freed pages (part I):

```
//-----------------------------------------------------------
static void ins_cbk(INS ins, void *v)
{
  INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)ins_instruction_cbk,
                 IARG_FAST_ANALYSIS_CALL, IARG_CONST_CONTEXT,
                 IARG_END);

  // Iterate over each memory operand of the instruction.
  UINT32 mem_ops = INS_MemoryOperandCount(ins);
  for ( UINT32 mem_op = 0; mem_op < mem_ops; mem_op++)
  {
    if ( INS_MemoryOperandIsWritten(ins, mem_op) )
    {
      INS_InsertPredicatedCall(
          ins, IPOINT_BEFORE, (AFUNPTR)memory_access_callback,
          IARG_INST_PTR,
          IARG_MEMORYOP_EA, mem_op,
          IARG_END);
    }
  }

}
```

Writing a bug finding tool

Detecting writes to freed pages (part II):

```
//----------------------------------------------
static VOID memory_access_callback(VOID *ip, VOID *addr)
{
  if ( g_checker != NULL )
    g_checker->check_write((ADDRINT)addr, 0);
}
```

Writing a bug finding tool

Detecting writes to freed pages (part III):

```
//-----------------------------------------------------------------------
size_t CMemoryChecker::check_write(ADDRINT ea, size_t size)
{
  PIN_LockClient();

  bool in_alloc = is_in_allocator();

  size_t ret = size;
  bool found = false;
  mem_area_vec_t::iterator it;
  for ( it = areas.end()-1; it != areas.begin() && ea != 0; --it )
  {
    if ( ea >= it->ea && ea <= (it->ea + it->size) )
    {
      found = true;
      if ( !it->available && !it->ignore_write )
      {
        if ( !in_alloc )
        {
          // Only report once per area
          it->ignore_write = true;
          add_breakpoint("Writing into a freed page");
          printf("WARNING: Writing into a freed page, write to 0x" EA_FMT
#if 0
          dump_areas();
#endif
        }
      }
      break;
    }
  }
```

Writing a bug finding tool

Once the malloc/realloc/free tracker is working, it’s possible to add other rules to detect other types of bugs like, for example, calls to realloc passing an already freed pointer.

However, getting the tracker to work is… non trivial. And most likely, specific to each heap allocator :(

Let’s talk about the problems of this approach to find bugs...

Problems

Problems

The most obvious problem of using this approach for finding bugs is the following:

- A dynamic analyzer will only analyse whatever is executed.

If there is a bug in function do_buggy_things() but with the inputs given to the program we don’t reach that portion of code, we will never see that bug.

Problems

Writing a malloc/realloc/free tracker is hard. It might sound easy, but is not. Some examples:

- Inside free(), the heap manager might decide to do “things” with more than a single page.
- Inside malloc/realloc(), the heap manager might decide to merge previously freed pages into one single page.
- Detecting writes to freed memory means having to determine when is it done by the heap manager and when is it done by the application.

Such a tool requires to have a perfectly working tracker or most of the bugs will be false positives otherwise.

Problems

Another problem of this approach is that the tracker is most likely specific to each heap manager that we want to support.

- Each heap manager will behave differently.

There are many heap managers out there:

- Windows' own heap, specific for each Windows version.
- The GNU allocator.
- Jemalloc.
- …

Some of them aren't generic but program specific, actually.

Problems

Another problem inherent to such tools is analysing the outputs of it.

Analyzing bugs found with such a tool can be summarized, often, to this message from a friend of mine:

Problems

Some bugs are pretty easy to analyse:

- Zero allocations. You can break in the debugger at malloc/realloc and check the size. As easy as it sounds.

Others aren't that easy to analyse:

- An example: a call to realloc passing a freed pointer that was freed in a totally different part of the code.
- Another example, a write to a previously freed memory page: Is it the heap allocator itself? Did the allocator reused that page and is available now? Where was the freed memory page first reserved, where it was freed and what is the relationship with the final write we observed?

Time for a DEMO!

And some examples of discovered bugs

Example bugs found

A zero allocation found in radare2 (fixed in <2 minutes because… pancake):

```
WARNING! Zero allocation detected!

Program received signal SIGTRAP, Trace/breakpoint trap.
__GI___libc_malloc (bytes=0) at malloc.c:2902
2902    malloc.c: No such file or directory.
(gdb) back
#0  __GI___libc_malloc (bytes=0) at malloc.c:2902
#1  0x00007f0c3adacd3f in cons_stack_dump (recreate=true) at cons.c:71
#2  0x00007f0c3adadf16 in r_cons_push () at cons.c:572
#3  0x00007f0c3b52b87f in r_core_cmd_str (core=0x55c31a823580, cmd=0x55c31a6220e8 "ieq") at cmd.c:3648
```

Example bugs found

A zero allocation found in NVIDIA 390 user-land libraries:

```
WARNING! Zero allocation detected!

Program received signal SIGTRAP, Trace/breakpoint trap.
__GI___libc_malloc (bytes=0) at malloc.c:2902
2902	malloc.c: No such file or directory.
(gdb) back
#0  __GI___libc_malloc (bytes=0) at malloc.c:2902
#1  0x00007f32331db501 in ?? () from /usr/lib/nvidia-390/libGL.so.1
#2  0x00007f323317e4c0 in ?? () from /usr/lib/nvidia-390/libGL.so.1
#3  0x00007f3246b2b67a in call_init (l=0x7f3233474000, argc=argc@entry=1, argv=argv@entry=0x7fff608290f8, env=env@entry=0x7fff60829108) at dl-init.c:58
#4  0x00007f3246b2b7cb in call_init (env=0x7fff60829108, argv=0x7fff608290f8, argc=1, l=<optimized out>) at dl-init.c:30
#5  _dl_init (main_map=0x7f3246d42168, argc=1, argv=0x7fff608290f8, env=0x7fff60829108) at dl-init.c:120
#6  0x00007f3246b1bc6a in _dl_start_user () from /lib64/ld-linux-x86-64.so.2
```

Conclusions

Conclusions

DBI toolkits are extremely useful tools for analyzing programs and discovering vulnerabilities.

However, while it might look easy at first to write such a tool, it’s actually pretty hard.

Analysing the resulting bug candidates is very time consuming and automating it is not trivial.

Writing a DBI tool by itself is not enough: a fuzzer, for example, is required to discover bugs hidden deep in the code.

Using a DBI tool with a coverage guided fuzzer looks like a very good idea.

Conclusions

From an exploitability point of view, most of the bugs found are not interesting:

- Bugs at start-up.
- Bugs, like zero allocations, that cannot be exploited.

Also, most of the software we use is so plagued with bugs that the amount of bugs that such a tool can discover in a normal application is too big to analyze.

All of that said, however, perhaps such a tool can be mixed with other program analysis techniques at least to lift the results and determine what looks like interesting and what doesn’t. I haven’t explored this option yet.

Questions?

And that's it! The source code of MemBugTool (call me original…) is available here:

- http://github.com/joxeankoret/membugtool