CENSUS
IT Security Works

Program Instrumentation Without Source code

Dimitrios Tatsis (@dtouch3d)
tatsisd@census-labs.com

FOSSCOMM 2018 Heraklion, Crete

www.census-labs.com

> whoami

- IT Security Researcher @ CENSUS
- DynamoRIO Google Summer of Code 2013

> Binary Instrumentation

- Allows the analysis of a binary application by injecting code at the assembly level
- No source code required

> Use cases

- Debugging
- Performance Profiling
- Security Analysis
- Exploit Prevention
- Malware Analysis
- Binary Translation

> Dynamic vs Static Binary Instrumentation

- Static Binary Instrumentation:
  - Create a new binary that includes our instrumentation code (aka Static Binary Rewrite)

- Dynamic Binary Instrumentation:
  - While a binary program is being executed, insert instrumentation code
  - Much more common

> No Source Code

- Interoperability with 3rd-party components
- Security Analysis
- Heisenbugs
- Compiler bugs

Even with source, we often need to analyze the behavior of the program at the very low level

> Problems

- Architectures: x86, x86-64, ARM, ARM64
- OS: Linux, *BSD, Windows, MacOS, Android, iOS

- Different instructions, semantics, calling conventions and limitations, e.g.
  - Return address on x86/ARM
  - Application Binary Interface of x86-64 on Windows and Linux

> Layman's Instrumentation

- GDB
  - Debug binaries
  - Insert breakpoints
  - Scripting
- Ltrace
  - Inspect library calls
- Strace
  - Inspect system calls

> LD_PRELOAD

- Provide our own library that overrides known library functions like read() from libc
- During application startup, the linker/loader executes the code in .ini/.fini from every loaded library
- Hijack execution

> Ptrace

- Unix's standard way to debug processes
  - PTRACE_PEEKTEXT
  - PTRACE_PEEKDATA
  - PTRACE_GETREGS

- Total control over the target application

> Dynamic Binary Instrumentation Frameworks

- Intel Pin
  - Free as in free beer only ☹
  - x86, x86-64
  - Linux, Windows, MacOS
- Valgrind
  - Free
  - x86, x86-64, ARM, ARM64
  - Unix OSes

> Dynamic Binary Instrumentation Frameworks

- Frida
  - Free
  - x86, x86-64, ARM, ARM64
  - Focus on Android/iOS
- DynamoRIO
  - Free
  - x86, x86-64, ARM, ARM64
  - Linux/Windows

> Dynamic Binary Instrumentation Frameworks

- Exposed APIs
  - Event callbacks
  - Function hooking
  - Instructions modification
  - Shadow Memory

> Dynamic Binary Instrumentation Tools

- Valgrind
  - Memcheck: Memory corruption bugs
  - Hellgrind: Race error detection

- DynamoRIO
  - DrMemory: Memory corruption bugs
  - DrHeapstat: Heap memory Usage
  - DrFuzz: Fuzzing

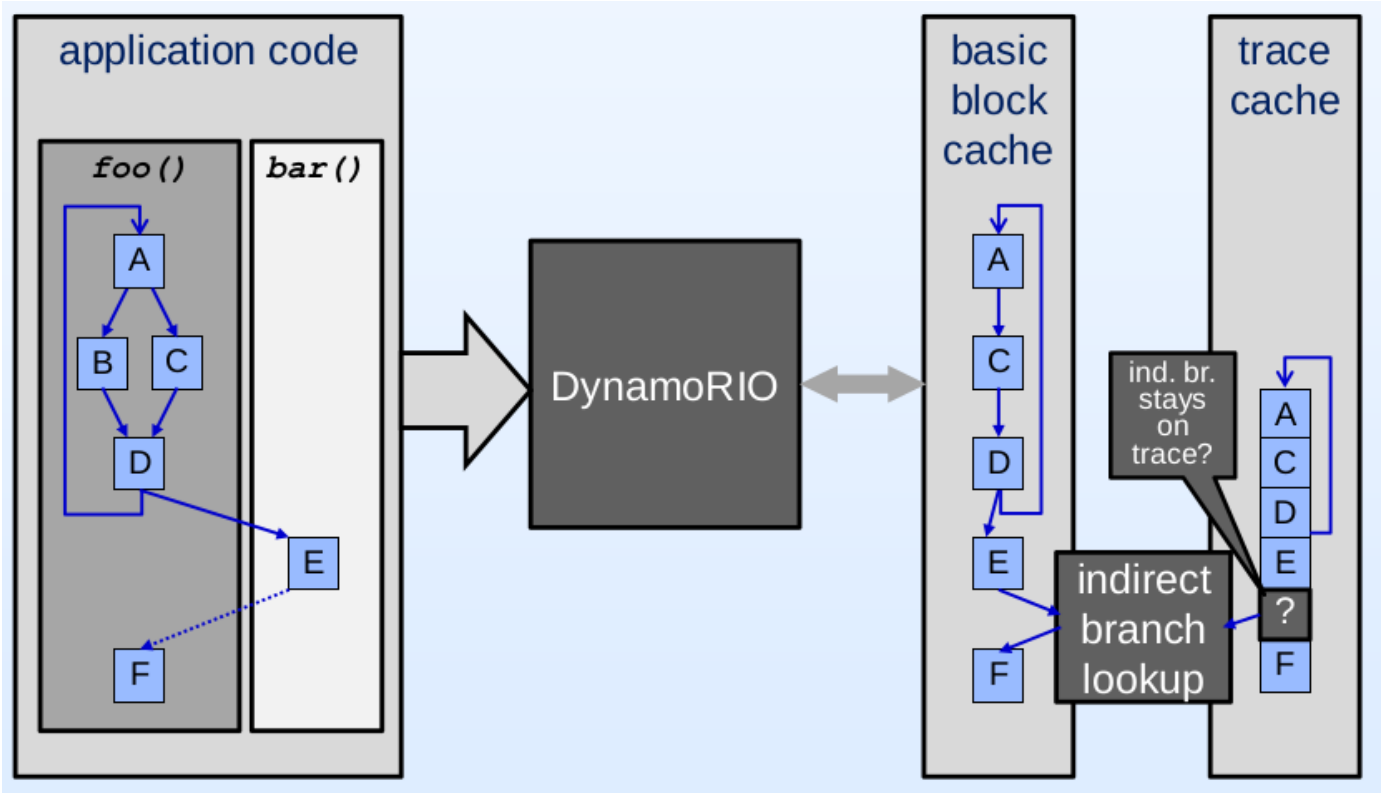
> DynamoRIO

- Created in the 2000's by MIT and HP
- Aims for transparency and efficiency
- Powerful API for code manipulation

> DynamoRIO

- Instrumentation Phase
  - Code about to be executed is analyzed
  - Instrumentation code inserted appropriately
  - Executed only once
- Execution Phase
  - Instrumented code is executed
  - Multiple times

> DynamoRIO



https://github.com/DynamoRIO/dynamorio/releases/download/release_7_0_0_rc1/DynamoRIO-tutorial-feb2017.pdf

> DynamoRIO

- Function hooking
  - drwrap_wrap()
- System calls
  - dr_register_pre_syscall_event()
- Basic Block Instrumentation
  - dr_register_bb_event()

> Function Hooking Example

- At every module (library) load event
- Check if library is libc
- Get read() address
- Hook read() with our own handler

> Function Hooking Example

```
static void
module_load_event(void *drcontext, const module_data_t *module, bool loaded)
{
    const char* module_name = dr_module_preferred_name(module);

    if (strncmp(module_name, "libc.so", strlen("libc.so")) == 0) {
        app_pc read_addr = dr_get_proc_address(mod->handle, "read");

        if (read_addr != NULL)
            drwrap_wrap(read_addr, read_handler, NULL);
    }
}

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


> Function Hooking Example

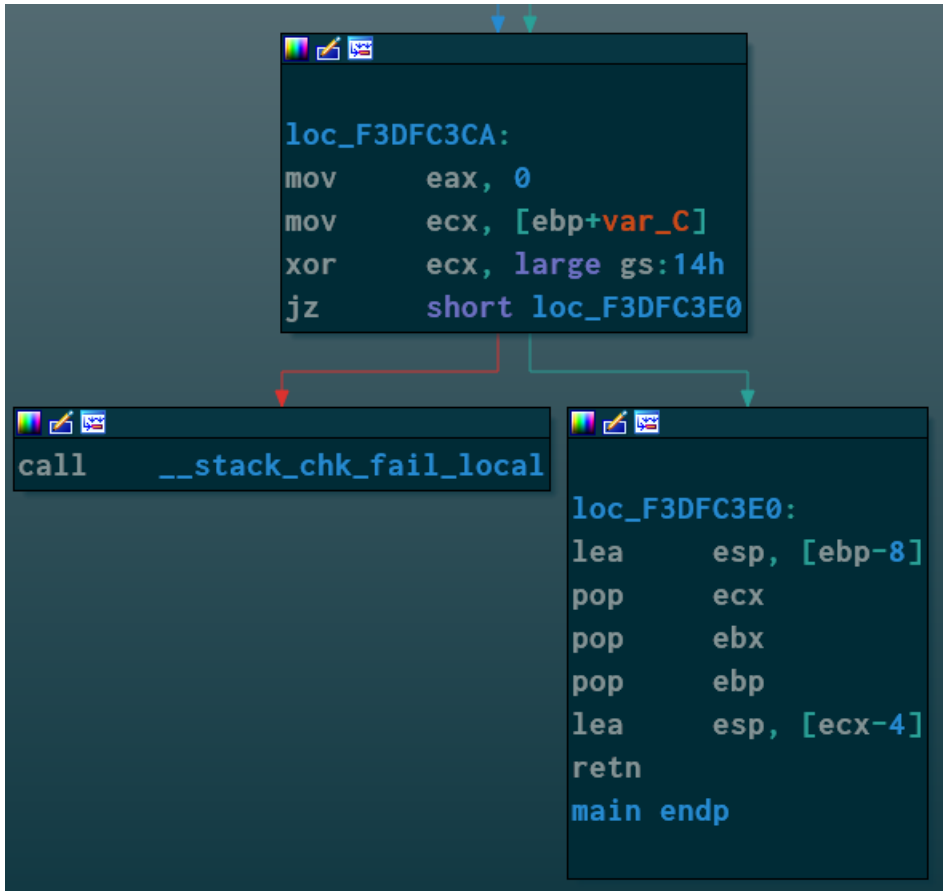
```
void
pre_read(void *wrapcxt, void** user_data)
{
    int fd = (int)drwrap_get_arg(wrapcxt, 0);
    void* buf = drwrap_get_arg(wrapcxt, 1);
    size_t size = (size_t)drwrap_get_arg(wrapcxt, 2);

    drfz_log("Called read(%d, %p, %zu)\n", fd, buf, n);
}
```

$ ./dynamorio/bin64/drrun -c libexample.so -- cat test.txt
Called read(3, 0xf3e28cc, 4096)

> Basic Block

- A set of assembly instructions executed linearly.
- Ends at a jump instruction

> Basic Block Example



> Basic Block Instrumentation

- DynamoRIO calls our handler once for every instruction in a basic block before execution

- Registered with
  - dr_register_bb_event()
  - drmgr_register_bb_instrumentation_event()

> Simple Memory Tracing Example

- Trace memory used by the target application
- Ignore instructions that do not belong to the application
  - e.g. DynamoRIO internal instructions
- For each *mov* instruction
  - Get the memory address used as source
  - Log the address

> Simple Memory Tracing Example

```
dr_emit_flags_t event_bb_insert(void *drcontext, void *tag, instrlist_t
        *bb, instr_t *instr, bool for_trace, bool translating, void *user_data)
{
    if (!instr_is_app(pc))
        return DR_EMIT_DEFAULT;

    if (instr_is_mov(instr)) {

        for (int i=0; i<instr_num_srcs(instr); i++) {
            opnd_t src = opnd_get_src(instr, i);
```


> Simple Memory Tracing Example

```
if (opnd_is_memory_reference(src)) {

    reg_id_t reg;

    /* Insert code to get memory address at runtime */
    insert_get_mem_addr(drcontext, ilist, instr, src, &reg);

    /* Insert code to log memory address at runtime */
    dr_insert_clean_call(drcontext, bb, instr
        clean_call_log_memaddr,  false, 1 /* number of arguments */,
        opnd_create_reg(reg)
    );
}
```


> Simple Memory Tracing Example

- To get address at runtime
  - Allocate two extra registers as workspace
  - Use drutil_insert_get_mem_addr()
  - Memory address is saved to register at execution phase
  - Release registers after usage

> Simple Memory Tracing Example

```
void
insert_get_mem_addr(void *drcontext, instrlist_t *ilist, instr_t *where,
        opnd_t *memref, reg_id_t *reg_addr)
{
    reg_id_t scrach;

    if (drreg_reserve_register(drcontext, ilist, where, NULL, reg_addr) !=
            DRREG_SUCCESS ||
        drreg_reserve_register(drcontext, ilist, where, NULL, &scratch) !=
        DRREG_SUCCESS) {

        DR_ASSERT_MSG(false, "Could not get scratch registers");
        return;
    }

    drutil_insert_get_mem_addr(drcontext, ilist, where, *memref, *reg_addr, scratch);
}
```


> Simple Memory Tracing Example

- To log memory address used
  - Create operand from the register that *will* hold the memory address at runtime
  - Use operand as function argument
  - Insert "clean call" before instruction

> Simple Memory Tracing Example

```
void
clean_call_log_memaddr(void* addr)
{
    dr_printf("app using memory @ %p\n", addr);
}
```


> Dr. Angry

- Generate testcases that maximize code coverage
- Useful in absence of testcases for fuzzing
- Concolic (Native + Symbolic) execution
- DynamoRIO + angr
- Coming soon to a repository near you

> Symbolic Execution

- Treat variables as unknowns (symbolic)
- Gather predicates that determine the code paths taken that depend on symbolic variables
- Solve for conditions that lead to a specific code path

> Symbolic Execution

```c
int main(int argc, char* argv[])
{
    int input, magic;
    read(stdin, &input, sizeof(input));

    magic = input + 0x123;

    if (magic == 0x1337) {
        printf("yeeeehaw\n");
    }
}
```

X = input

magic = X + 0x123

magic == 0x1337

=> X + 0x123 == 0x1337

=> X = 0x1214

> Dr. Angry

- Native execution
  - DynamoRIO
  - Get application snapshot at input
  - Taint analysis
  - Log all jump instructions that were *not* taken

> Dr. Angry

- Symbolic execution
  - angr
  - Start symbolic execution from snapshot
  - Treat specific input as symbolic
  - Solve for input that explores a new code path
  - Create new testcase
  - Repeat

> Dr. Angry

- Explore the state space of libjpeg
- JPEG files start with 0xFF 0xD8
- Frame markers, Huffman tables, etc.

> Dr. Angry Demo

Demogod take the wheel

> Dr. Angry Demo

```
2018-10-12 02:44:30+0300 [-] callNativeHook calling: 0xf7e957b1, retaddr: <BV32 0xf7e954ab>
2018-10-12 02:44:30+0300 [-] callNativeHook calling: 0xf7e957b1, retaddr: <BV32 0xf7e954ab>
2018-10-12 02:44:30+0300 [-] sm ended for addr 0xf7e6e606 : <SimulationManager with 2 avoid>
2018-10-12 02:44:30+0300 [-] finding address 0xf7e79718
2018-10-12 02:44:30+0300 [-] callNativeHook calling: 0xf7e956e6, retaddr: <BV32 0xf7e95423>
2018-10-12 02:44:30+0300 [-] callNativeHook calling: 0xf7e956e6, retaddr: <BV32 0xf7e95423>
2018-10-12 02:44:30+0300 [-] Got solution : ffd83729b6cce6ffc178c14da4dc9a86ddb9799898f7a54c08
```

```
~/drangry/testcases(d0c33b2*) » file ./*
./djpeg_05e2ab672505474faabae88dae3d1168: data
./djpeg_0d455f6ec5f64d228ebb4d735fafc790: JPEG image data
./djpeg_1340bda1decb4bda9a54cb31586b9867: JPEG image data
./djpeg_146c81ba23244087a85da16ee8105979: data
./djpeg_19334ebdfca647808d711e97c0f0e4ac: data
./djpeg_2563ada84c5d47aa96210a772b546faa: data
./djpeg_2c8f0e1dd5b745e68d7798a79d58d1bf: data
./djpeg_2d50966aaf384a0987f09050b09d46ab: data
./djpeg_353fc021717e4138b4677d7480297030: data
./djpeg_4038fab052ee4c14b38ef3682161d415: JPEG image data
```


> Conclusion

- Dynamic Binary Instrumentation is a powerful technique
- Instrumentation Frameworks provide many utilities that allow the creation of amazing tools
- Who needs source code anyway? ☺

> References

- https://dynamorio.org
- https://github.com/DynamoRIO/dynamorio/releases/download/release_7_0_0_rc1/DynamoRIO-tutorial-feb2017.pdf
- https://valgrind.org
- https://angr.io
- https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html

Thank you!

CENSUS

IT Security Works