Ingeniería Inversa
Clase 7

Instrumentación Binaria

martin.uy
Open by default.

Instrumentación Binaria

- ¿Qué es?
  - Introducción de código al código original de la aplicación que, generalmente, no busca alterar su resultado funcional (transparente)
  - Inyección de trampolines (callbacks)
  - Modificación de instrucciones (traducción binaria)
  - Intrumentación sobre el código fuente o sobre el binario
  - Instrumentación previo o durante la ejecución

Instrumentación Binaria

- ¿Para qué?
  - Profiling – obtener información para optimización de performance
  - Cobertura de código (code-coverage)
  - Análisis del comportamiento (entender funcionalidad)
  - Análisis de memoria (leaks, dangling pointers)
  - Fuzzing en memoria
  - Ejecución sobre otra arquitectura (traducción binaria)
  - Testing (triggerear flujos de ejecución)

Instrumentación Binaria

- Aplicable a binarios (PE, ELF, classfiles, etc.)
- Frameworks de instrumentación binaria:
  - DynamoRIO (Windows, Linux, Android)
  - PIN (Windows, Linux)
  - Windows API Monitor (Windows)
  - QEMU (Linux)
  - ASM (Java)
  - Byteman (Java)

Instrumentación Binaria

- DynamoRIO
  - Windows, Linux, Android
  - Open source (licencia BSD)
  - AArch32, AArch64, IA-32, x86_64
  - http://dynamorio.org

Instrumentación Binaria

- Ejemplos
  - `./bin64/drrun -c ./samples/bin64/libbbsize.so -- ls /`

```
Number of basic blocks seen: 3560
                Maximum size: 43 instructions
                Average size:   4.8 instructions
```

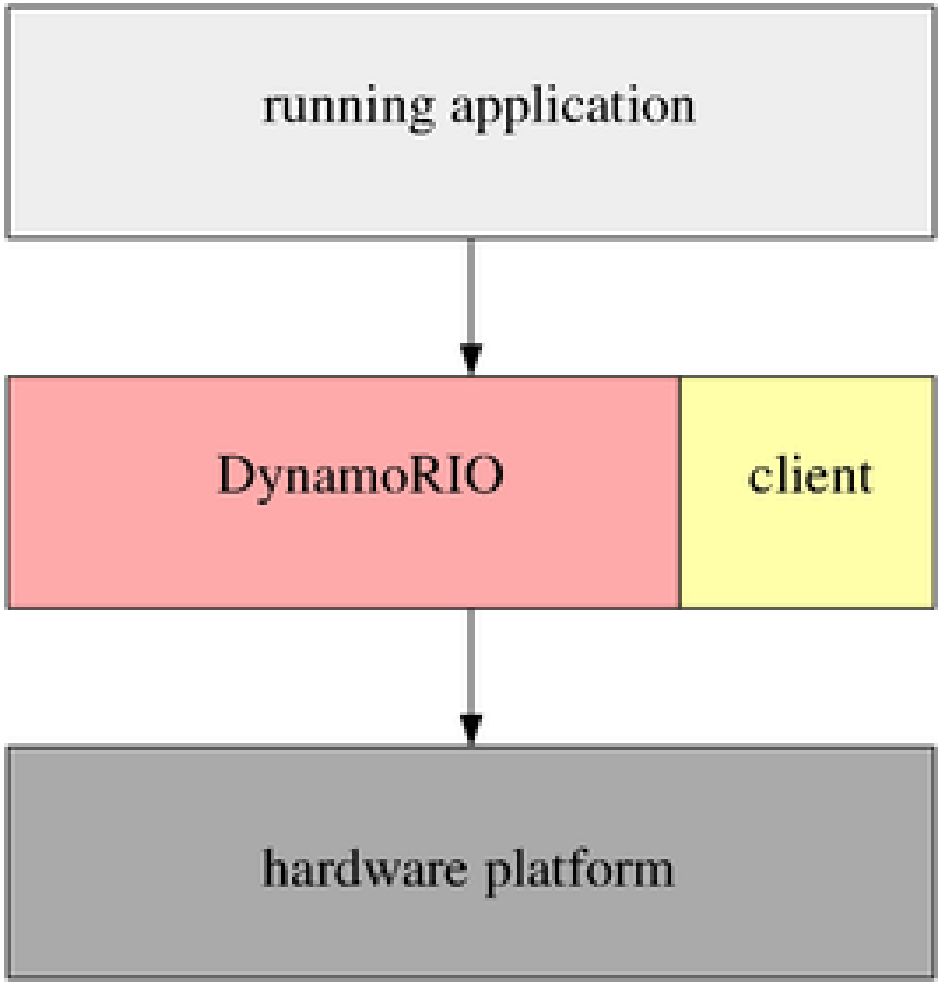
Instrumentación Binaria

- Ejemplos
  - `./bin64/drrun -c ./samples/bin64/libopcodes.so -- ls /`

```
Top 15 opcode execution counts in 64-bit AMD64 mode:
      11137 : xor
      14808 : shr
      16073 : pop
      16742 : sub
      20663 : push
      21278 : jnz
      21753 : jnz
      24025 : jz
      27753 : jz
      29073 : movzx
      29707 : and
      32082 : lea
      52862 : add
      57644 : test
      59205 : cmp
      90661 : mov
     101790 : mov
```

Instrumentación Binaria

- Arquitectura



Instrumentación Binaria

- Librería cliente
  - Librería dinámica (PIC)
  - Tiene implementados los hooks de la instrumentación
  - Desarrollada por el instrumentador
  - Linkea dinámicamente a las librerías de DynamoRIO
  - Se carga al proceso instrumentado desde el comienzo

Instrumentación Binaria

- La librería cliente recibe eventos de DynamoRIO a través de los callbacks que registra
- Se pueden registrar múltiples callbacks para un mismo evento y pueden haber múltiples librerías cliente
- `dr_client_main`
  - Punto de entrada a la librería cliente
  - Inicialización de extensiones y registro de callbacks
  - Se llama cuando el proceso es creado

Instrumentación Binaria

- DynamoRIO brinda una API de propósito general: no es aconsejable “confiar” en librerías cargas en el proceso instrumentado
  - Abrir, leer, escribir archivos
  - Primitivas de sincronización (ej. Mutex)
  - Alocación de memoria
  - Creación de threads
  - Etc.

Instrumentación Binaria

- Ejemplos de eventos a los que la librería cliente se puede suscribir:
  - Creación de basic blocks o instrucciones
  - Inicialización/finalización de threads
  - Loading/unloading de librerías
  - Intercepción de syscalls
  - Intercepción de señales o excepciones

Instrumentación Binaria

- Hay múltiples APIs de instrumentación
- Multi-Instrumentation Manager
  - Funciona bajo el esquema de 4 pasadas por el código ejecutable
  - 1) App2App
    - Transformaciones al código de la aplicación, previas a la instrumentación
  - 2) Análisis
    - Análisis del código de la aplicación, una vez aplicadas las transformaciones de la fase App2App. En esta fase no se modifica el código

Instrumentación Binaria

- Multi-Instrumentation Manager
  - 3) Instrumentación
    - Transformaciones al código de la aplicación propias de la instrumentación. Pueden ser transformaciones de alto nivel, que requieren múltiples instrucciones. Ej. inserción de clean-calls
  - 4) Instrumentation2Instrumentation
    - Pasada para ver y transformar el código generado durante la instrumentación. Pueden, por ejemplo, realizarse optimizaciones
  - Cada fase es opcional

Instrumentación Binaria

- Multi-Instrumentation Manager
  - Registro de callbacks para las diferentes fases de la instrumentación

```
if (!
drmgr_register_bb_instrumentation_ex
_event(app2app_cb, analysis_cb,
instruction_cb, instr2instr_cb, NULL))
        DR_ASSERT(false);
```

Instrumentación Binaria

- Multi-Instrumentation Manager
  - Callback para la fase de instrumentación
    - Llamado una vez por instrucción en el basic block

```
static dr_emit_flags_t
instruction_cb(void* drcontext, void*
tag, instrlist_t* bb, instr_t* instr, bool
for_trace, bool translating, void*
user_data);
```

Instrumentación Binaria

- Creación de basic blocks
  - Basic block: secuencia de instrucciones que termina con una instrucción de control de flujo
  - Representación de instrucciones: `instr_t` y `instrlist_t` (*dr_ir_instr.h* y *dr_ir_instrlist.h*)
  - Podemos modificar, agregar o eliminar instrucciones

Instrumentación Binaria

- Creación de basic blocks
  - Antes de ejecutarse un basic block de la aplicación, se copia a la “code cache” y allí se disparan los eventos de instrumentación
  - DynamoRIO mantiene el control sobre la ejecución al final del basic block para seguir instrumentando con la misma estrategia (a medida que se van ejecutando nuevos basic blocks)
    - No se instrumenta todo el programa de una. Las partes que nunca se ejecutan no son instrumentadas

Instrumentación Binaria

- Inserción de instrucciones
  - Meta-instrucciones
    - Transparentes para la aplicación, usadas para monitoreo
    - Ej. llamada a una función de la librería cliente
    - No son instrumentadas por DynamoRIO
  - Instrucciones de aplicación
    - Modifican el estado de la aplicación

Instrumentación Binaria

- APIs para encodear, decodear y desensamblar instrucciones
  - Estructura: `instr_t`
- Clean Calls
  - Insertar una llamada a código C (hook) en medio de un basic block
  - Se hace la llamada cada vez que se ejecuta el basic block
  - Se preserva el estado de la aplicación (registros, registros de punto flotante, stack, etc.)

Demo 7.1

Instrumentación

Instrumentación Binaria

- ¿Cómo funciona la instrumentación internamente?
  - *drrun* hace un execve y queda en ejecución *libdynamorio.so.6.2*
  - Se ejecuta la función *_start* dentro de esta librería (implementada en assembly para x86)
  - *_start* relocaliza la librería y llama a *privload_early_inject*
  - Esta función utiliza un loader de DynamoRIO para cargar el binario ELF -a ser instrumentado- y lo inicializa (*dynamorio_app_init*)
  - Finalmente se llama a *dynamo_start*

Instrumentación Binaria

- Para este momento, el proceso tiene ya mapeadas la aplicación a ser instrumentada (ej. *main*) y la librería cliente donde están los hooks de la instrumentación (ej. *ins_example.so*)
- Se llama a la función “dispatch” que le permite mantener a DynamoRIO el control durante la ejecución instrumentada
  - Este loop infinito ejecuta hasta que termina el proceso
  - “dispatch” instrumenta basic blocks, los pone a ejecutar y recupera el control (porque los basic blocks instrumentados vuelven a “dispatch”)

Instrumentación Binaria

- La función *build_basic_block_fragment*, llamada por “dispatch”, crea basic blocks instrumentados
  - Se llaman “fragmentos” a los basic blocks instrumentados
  - Los fragmentos se representan mediante la estructura `fragment_t`
  - Ejemplo de llamada para el 1er basic block de *main*: el parámetro *start* tiene valor 0x400144

Instrumentación Binaria

- Código original de main:

```
(gdb) x/10i 0x400144
0x400144:  push   %rbp
0x400145:  mov    %rsp,%rbp
0x400148:  mov    $0x0,%eax
0x40014d:  callq  0x400166
0x400152:  nop
0x400153:  mov    $0x3c,%rax
0x40015a:  mov    $0x0,%rdi
0x400161:  syscall
0x400163:  nop
0x400164:  pop    %rbp
```

```
void _start() {
    foo();
    asm(
        "nop\n"
        "mov $60, %rax\n"
        "mov $0, %rdi\n"
        "syscall\n"
    );
}
```

Instrumentación Binaria

- *build_basic_block_fragment* llama a los hooks de la librería cliente para obtener la lista final de instrucciones instrumentadas
- Una vez obtenida dicha lista, la función *emit_fragment_common* va a crear el nuevo fragmento
  - Esto implica crear en memoria un nuevo segmento ejecutable para las instrucciones (como haría un compilador JIT)

Instrumentación Binaria

- Ejemplo de un `fragment_t` creado a partir del primer basic block de main:

```
$2 = {tag = 0x400144 "UH\211",
<incomplete sequence \345\270>, flags =
16777264, size = 435, prefix_size = 0
'\000', fcache_extra = 9 '\t',
  start_pc = 0x54691008 "eH\243",
in_xlate = {incoming_stubs = 0x0,
translation_info = 0x0}, next_vmarea =
0x0, prev_vmarea = 0x546c3090, also = {
    also_vmarea = 0x0, flushtime = 0}}
```

Instrumentación Binaria

- Allí se puede visualizar información tal como:
  - tag: dirección virtual del basic block original
  - start_pc: dirección virtual del basic block instrumentado
- En */proc/<PID>/maps* podemos verificar como la dirección en start_pc (0x54691008) corresponde a un segmento ejecutable:

54691000-54692000 rwxp 00000000 00:00 0

Instrumentación Binaria

- Instrucciones en 0x54691008 (basic block instrumentado):

```
(gdb) x/50i 0x54691008
0x54691008:  movabs %rax,%gs:0x0
0x54691013:  movabs %gs:0x20,%rax
0x5469101e:  mov    %rsp,0x18(%rax)
0x54691022:  mov    0x2e8(%rax),%rsp
0x54691029:  movabs %gs:0x0,%rax
0x54691034:  lea    -0x2a8(%rsp),%rsp
0x5469103c:  callq  0x5468acc0
0x54691041:  callq  0x11087
0x54691046:  callq  0x5468ad80
```

Instrumentación Binaria

- Esas instrucciones son el resultado de la pasada instruction2instruction, y lo que es finalmente ejecutado
- En el listado anterior se puede ver la instrucción *callq 0x11087*
  - *ins_example.so* está mapeado en 0x10000
  - En el offset 0x1087 encontramos a la función *runtime_cb*
  - En instruction2instruction se insertó un clean call a esta función por cada basic block

Instrumentación Binaria

- ins_example.so

```
0000000000001087 <runtime_cb>:
1087:    push   %rbp
1088:    mov    %rsp,%rbp
108b:    lea    0x2d7(%rip),%rdi
1092:    mov    $0x0,%eax
1097:    callq  ba0 <dr_printf@plt>
109c:    nop
109d:    pop    %rbp
109e:    retq
```

```
static void
runtime_cb(void) {
    dr_printf("runtime
call to hook
method!\n");
}
```

Instrumentación Binaria

- Estas instrucciones (*callq 0x11087*) son clean calls
- Los clean calls son precedidos por la llamada a una función que salva el contexto (*callq 0x5468acc0*) y sucedidos por una que restaura el contexto (*callq 0x5468ad80*)

Instrumentación Binaria

- El código visto en la pasada instrumentation2instrumentation tiene un call a 0x400166
  - A nivel de código fuente en C (*main.c*), esto corresponde a la llamada a la función *foo*
- Sin embargo, si el bloque instrumentado llamara directamente a 0x400166, DynamoRIO perdería el control y no podría seguir instrumentando basic blocks
- Por lo tanto, a nivel del fragmento, la llamda a 0x400166 fue sustituida por el siguente código:

Instrumentación Binaria

```
0x54691144:  mov    $0x0,%eax
...
0x5469118c:  movabs %rax,%gs:0x0
0x54691197:  movabs %gs:0x20,%rax
0x546911a2:  mov    0x18(%rax),%rsp
0x546911a6:  movabs %gs:0x0,%rax
0x546911b1:  pushq  $0x400152
0x546911b6:  jmpq   0x546b1030
```

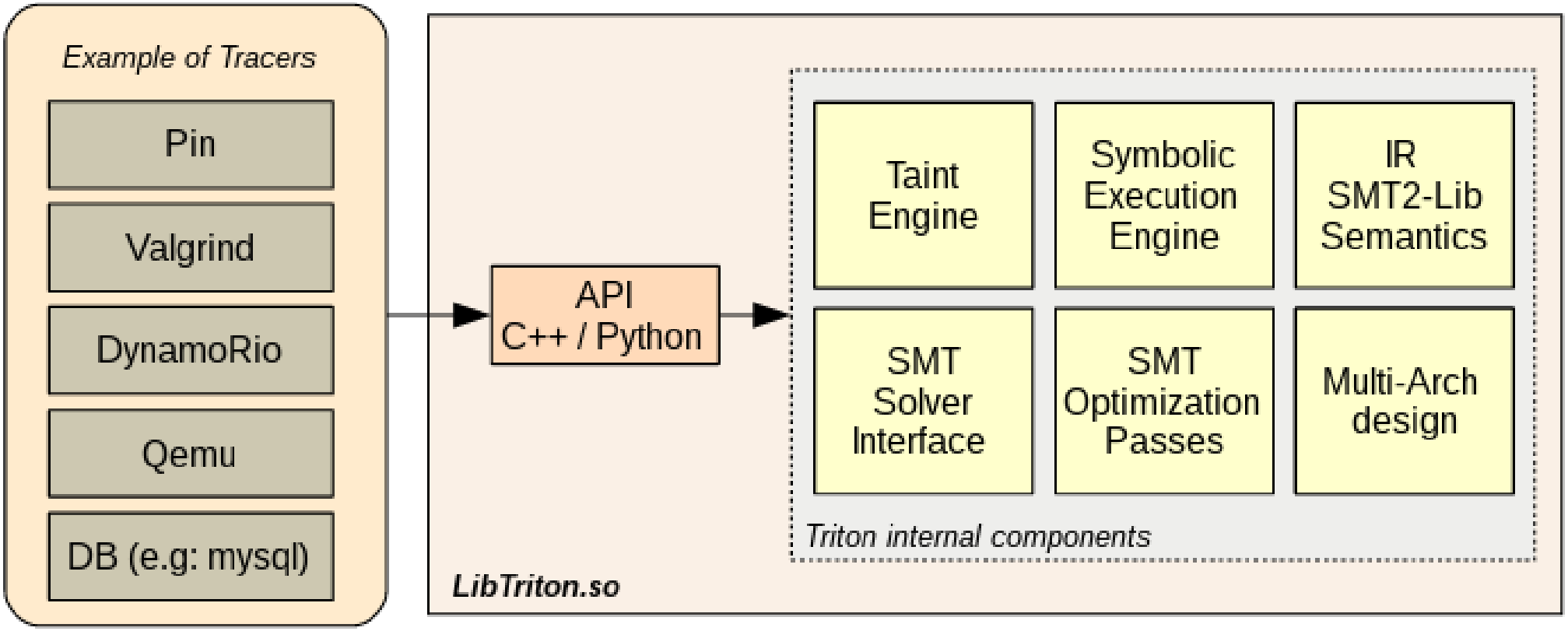
Instrumentación Binaria

- En lugar de llamar a 0x400166, se salta a 0x546b1030
- La dirección de retorno de llamar a *foo* es pusheada al stack
- ¿Qué hace el código en 0x546b1030?
  - Salva el contexto
  - Llama a “dispatch”
- Se repite el ciclo, en este caso instrumentando el basic block de *foo*

Dynamic Binary Analysis

- Basados en los frameworks de instrumentación binaria pueden construirse herramientas de más alto nivel para realizar chequeos sobre el binario en tiempo de ejecución
- Por ejemplo, Valgrind tiene la capacidad de hookear alocaciones y liberaciones de memoria para detectar leaks
- Triton es un framework de DBA desarrollado por Quarkslab con licencia abierta y multiplataforma
  - Combina la capacidad de realizar ejecución simbólica con SMT solvers

Dynamic Binary Analysis



El motor de SMT que utiliza Triton es z3

Dynamic Binary Analysis

- Taint analysis
  - Trazas de los registros y memoria que son controlados por el usuario (input)
  - Los inputs se consideran inseguros. Todas las instrucciones que los manipulen son especialmente interesantes desde la perspectiva de la seguridad. Es lo que “controla el atacante”
  - Una política de taint analysis tiene 3 componentes: 1) reglas de introducción, 2) reglas de propagación, y 3) reglas de chequeo

Dynamic Binary Analysis

- Taint analysis
  - Reglas de introducción: registros, memoria
  - Reglas de propagación:
    - Sobreaproximación (Triton)
      - Falsos positivos
    - Aproximación precisa
    - Subaproximación
      - Falsos negativos
  - La propagación es un trade-off de precisión y eficiencia (memoria + CPU)

Dynamic Binary Analysis

```
mov ax, 0x1122                  ; RAX is untainted
mov al, byte ptr [user_input] ; RAX is tainted
cmp ah, 0x99                    ; can we control this comparison?
```

En este caso, la sobre-aproximación va a suponer que la comparación puede ser controlada por el usuario. Eso es un falso positivo

Para estos casos, se puede utilizar ejecución simbólica y preguntarle al SMT solver si hay algún valor que satisfaga la restricción

Dynamic Binary Analysis

- Ejecución simbólica
  - Convertir registros y memoria a valores simbólicos
  - Hacer preguntas que puede responder un SMT solver
  - Ejemplo:
    - convertir registro eax en simbólico
    - procesar una instrucción que involucra el valor simbólico de eax
    - pedir un valor inicial de eax tal que una vez procesada la instrucción, se satisfaga una cierta condición

Dynamic Binary Analysis

```
Triton = TritonContext()
Triton.setArchitecture(ARCH.X86)

# rax is now symbolic
Triton.convertRegisterToSymbolicVariable(Triton.registers.eax)

# process instruction
Triton.processing(Instruction("\x83\xc0\x07")) # add eax, 0x7

# get rax ast
eaxAst =
Triton.getAstFromId(Triton.getSymbolicRegisterId(Triton.registers.eax))

# constraint
c = eaxAst ^ 0x11223344 == 0xdeadbeaf

print 'Test 5:', Triton.getModel(c)[0] # Out: SymVar_0 = 0xCF8F8DE4
```

Dynamic Binary Analysis

- Emulación de código
  - Procesar instrucciones localizadas en un cierto rango de direcciones virtuales:

```
0x40056d: "\x55",                                   #   push    rbp
0x40056e: "\x48\x89\xe5",                           #   mov     rbp,rsp
0x400571: "\x48\x89\x7d\xe8",                       #   mov     QWORD PTR [rbp-0x18],rdi
0x400575: "\xc7\x45\xfc\x00\x00\x00\x00",           #   mov     DWORD PTR [rbp-0x4],0x0
0x40057c: "\xeb\x3f",                               #   jmp     4005bd <check+0x50>
0x40057e: "\x8b\x45\xfc",                           #   mov     eax,DWORD PTR [rbp-0x4]
0x400581: "\x48\x63\xd0",                           #   movsxd  rdx,eax
0x400584: "\x48\x8b\x45\xe8",                       #   mov     rax,QWORD PTR [rbp-0x18]
```

Dynamic Binary Analysis

- Emulación de código
  - Crear instrucciones (opcode + dirección virtual)
    - Instruction(), setOpcode, setAddress
  - Pedirle a Triton que las procese
    - Triton.processing(inst)
  - Obtener el valor de RIP después de ejecutarlas (en términos de direccionamiento virtual)
    - ip = Triton.buildSymbolicRegister(Triton.registers.rip).evaluate()

Dynamic Binary Analysis

- Emulación de código
  - Setear valores concretos a la memoria y a los registros
    - Triton.setConcreteMemoryValue(0x601040, 0x00)
    - Triton.setConcreteRegisterValue(Triton.registers.rdi, 0x1000)
  - Simbolizar la memoria
    - Triton.convertMemoryToSymbolicVariable(MemoryAccess(address, CPUSIZE.BYTE))

Dynamic Binary Analysis

- Emulación de código
  - Obtener valores concretos de la memoria y los registros
    - Triton.getConcreteMemoryValue(MemoryAccess(write+4, CPUSIZE.DWORD))
    - Triton.getConcreteRegisterValue(Triton.registers.rax)
  - A una instrucción podemos desensamblarla y obtener los operandos
    - inst.getDisassembly()
    - inst.getOperands()

Dynamic Binary Analysis

- Emulación de código
  - Podemos analizar las “micro-instrucciones” o “instrucciones atómicas” que implica una instrucción
    - Muchos compiladores utilizan una representación intermedia (IR) para este tipo de instrucciones
  - Ej. movabs rax, 0x4142434445464748 implica:
    - Setear rax con un cierto valor
    - Avanzar rip para apuntar a la siguiente instrucción
  - inst.getSymbolicExpressions()

Dynamic Binary Analysis

- Emulación de código
  - Podemos analizar que “micro-operación” afectó por último un registro o una dirección de la memoria
    - Triton.getSymbolicRegisters().items()
    - Triton.getSymbolicMemory().items()
  - Cuando la memoria o el registro son simbólicos (Triton.buildSymbolicRegister(Triton.registers.ah)), podemos pedir las micro-operaciones que lo modificaron, u obtener un valor concreto

Dynamic Binary Analysis

- Emulación de código
  - Una vez realizada la emulación, podemos obtener todas las restricciones de flujos de la ejecución (resultado de cada branch)
    - getPathConstraints → getBranchConstraints
    - Ej: 0x11223344: jne 0x55667788
    - Flag: verdadero si el branch es tomado
    - Dirección de origen: 0x11223344
    - Dirección de destino: 0x55667788 si el branch es tomado o siguiente dirección en caso de que no
    - pc: nodo que representa el branch dentro del Árbol Abstracto de la Sintáxis (AST)

Demo 7.2

Ejecución simbólica (Triton)

Lab

7.1

Crear una librería cliente de DynamoRIO que detecte los parámetros a funciones que son punteros a memoria dinámicamente alocada (x86_64, SystemV ABI)

Lab

7.2: Utilizar ejecución simbólica en Triton para encontrar un input que haga a la función *check* retornar 1:

```
int check(int i) {
  const unsigned char* c = (unsigned char*)&i;
  if (((c[0] ^ c[1]) == 0x3C) && ((c[0] * c[3]) ==
0x40) && c[1] != 0) {
    return 1;
  }
  return 0;
}
```

Referencias

- http://dynamorio.org/docs/
- Triton - dynamic binary analysis framework
  - https://github.com/JonathanSalwan/Triton
  - https://triton.quarkslab.com