Pin para Todos y Todas

Pablo G. Solé

Introducción a Pin

Introducción a Pin

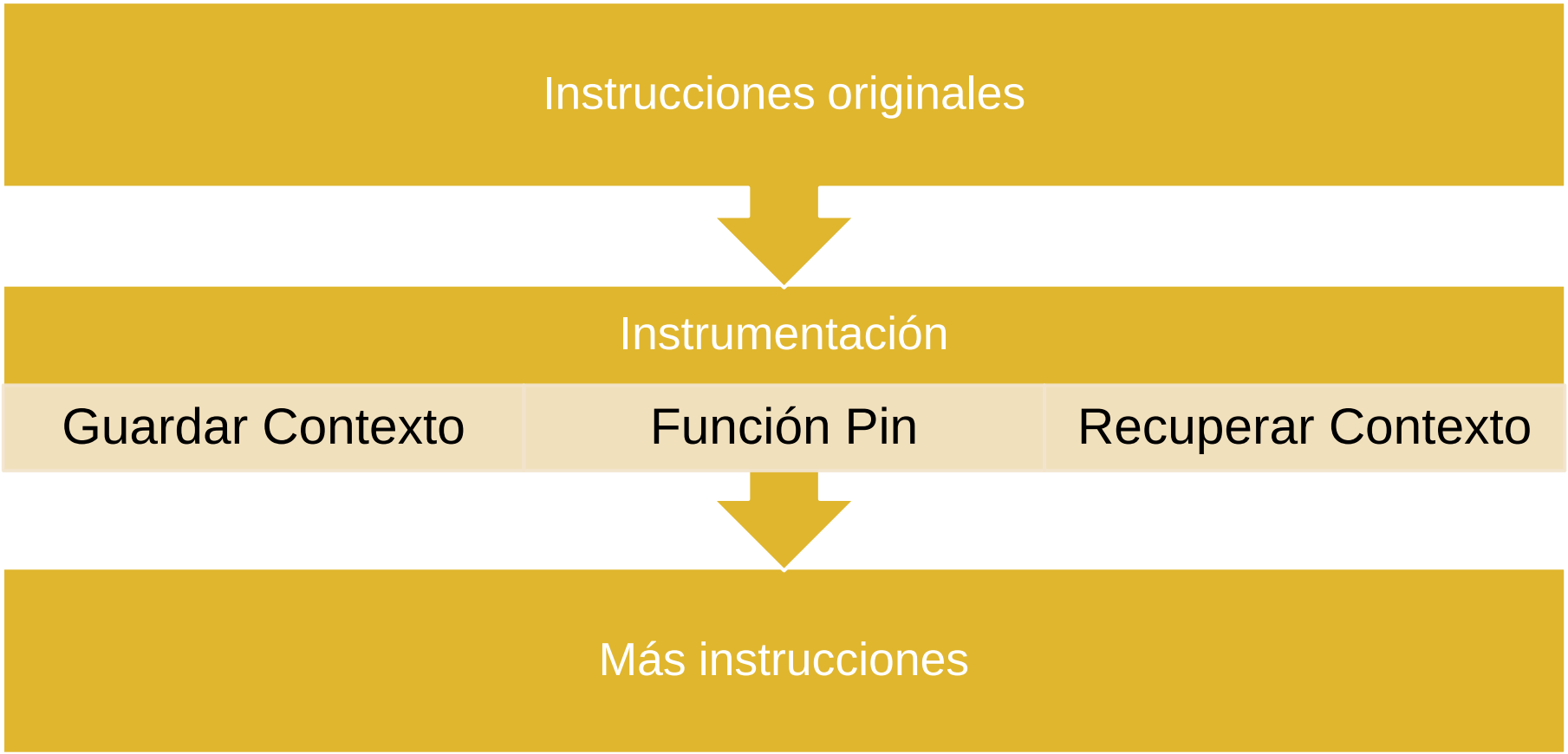
- Pin es una plataforma de DBI (Dynamic Binary Instrumentation)

- Desarrollada por Intel.
- Multiplataforma (Linux, Windows, Mac)
- Multiarquitectura (x86, x64, arm, mips)

Introducción a Pin

- Permite instrumentar binarios (inyectar código)
- Las Pin Tools se realizan en C++ únicamente
- Es un Kit especifico por compilador (MSVC, GCC)
- Garantiza que nuestra función compilada pueda inyectarse sin afectar la ejecución del binario.

Introducción a Pin



Casos de Uso

- Profiling
  - Funciones accedidas
  - Orden de acceso a funciones
  - Análisis de efectividad de una cache
- Code Coverage
  - Cantidad de instrucciones / basic blocks ejecutados
- Seguridad
  - Consistencia de memoria
    - Heap Overflows
    - Use-after-free
    - Stack overflows

Como instrumentar

- Diferentes momentos donde instrumentar
  - Eventos generales de la ejecución
    - Load/Unload Image
    - Start/Fini Thread
    - Start/Fini Application
    - Detach
  - Eventos de Instrumentación (primera vez)
    - Routine
    - Trace
    - BasicBlock
    - Instruction

Como instrumentar

- Diferentes momentos donde instrumentar
  - Eventos de Análisis (en paralelo)
    - Routine
    - Trace
    - BasicBlock
    - Instruction
  - Context Switch
  - Entrada y salida de una Syscall

Introducción a Pinocchio

(porque no solo C++ vino a jugar)

Introducción a Pinocchio

- Framework de Instrumentación basado en V8
- Proyecto Open Source (está en GitHub)
- Actualmente especifico para Windows y IA32
- Toda la funcionalidad de Pin en Javascript
- Un entorno más amigable y OO para realizar nuestras PinTools

¿Por qué Javascript?

- Reutilizar los esfuerzos de las Browser Wars
- Pocos lenguajes reciben tanta atención y esfuerzos de optimización como JavaScript

- Es Scripting, mutable, OO, loosely typed
- Es portable: una PinTool para hacer PinTools

Introducción a Pinocchio

- Énfasis en la performance
- V8 tiene 2 motores de JIT incorporados
- Se intentó disminuir el overhead en cada paso
- Es practico y usable incluso con binarios grandes
- Es extensible y puede complementarse con plugins en otros lenguajes

Ejemplos de Pinocchio

- Iteradores para threads, images, sections, etc.

```
for (x in images)
  log(images[x].name + " - " +
      images[x].loadOffset.hex());
```

…
C:\Windows\syswow64\CRYPTBASE.dll - 75020000
C:\Windows\syswow64\SspiCli.dll - 75030000
C:\Windows\syswow64\GDI32.dll - 75090000
C:\Windows\syswow64\USP10.dll - 752a0000
C:\Windows\syswow64\msvcrt.dll - 76020000

Ejemplos de Pinocchio

- Eventos para cada tipo de instrumentación

```
function newimage(img) {
  log(img.name + " - " + img.loadOffset.hex());
}
function newthread(threadId, ctx, flags) {
  log("New Thread " + current.thread.tid);
  log("Thread Stack:" + ctx.get(REG_ESP).hex());
}

events.attach("loadimage", newimage);
events.attach("startthread", newthread);
```

Ejemplos de Pinocchio

```
function trace(tr) {
  log("TRACE: " + tr.address.hex());
  for (x in tr.basicblocks) {
    bbl = tr.basicblocks[x];
    log("  BBL: " + bbl.address.hex());
    for (y in bbl.instructions) {
       ins = bbl.instructions[y];
       log("    INS: " + ins.address.hex() + " - " +
  ins.disassemble);
    }
  }
}
```

```
TRACE: 77989950
  BBL: 77989950
    INS: 77989950 - mov esp, esi
    INS: 77989952 - pop ebx
    INS: 77989953 - pop edi
    INS: 77989954 - pop esi
    INS: 77989955 - pop ebp
    INS: 77989956 - ret 0x10
```

.

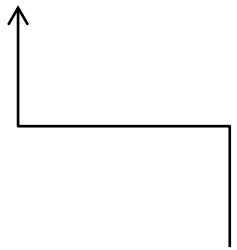
Ejemplos de Pinocchio

- Objectos AnalysisFunction (En Paralelo)

```
function af_ctor() {
  log=require("console").log;
}
function af_cback(addr) {
  log(addr.hex());
}

args = new ArgsArray(IARG_INST_PTR);
af = AnalysisFunction(af_cback, af_ctor, null, args);

bbl.attach(“before”, af);
```



Ejemplos de Pinocchio

- Accesos a memoria y otras yerbas

```
KUSER_SHARED_DATA = 0x7ffe0000;

//Path de windows
log(readTwoByteString(KUSER_SHARED_DATA + 0x30));

//Base address de ntdll.dll (WOW64)
log(readDword(KUSER_SHARED_DATA + 0x36C).hex());
```

Ejemplos de Pinocchio

- Accesos a memoria y otras yerbas

```
mem = new MemoryBuffer(0x7ffe0000, 0x1000);
KUSER_SHARED_DATA = new Uint32Array(mem);

//Base address de ntdll.dll (WOW64)
log(KUSER_SHARED_DATA[0xDB].hex());
```

Introducción a V8

Introducción a V8

- Motor de JavaScript de Google
  - Es lo que corre por debajo de Chrome
- Escrito en C++
- Multiplataforma (windows, linux, mac, android, …)
- Multiarquitectura (ia32, ia64, arm, mips)
- Autocontenido (listo para ser embebido)
- Notablemente poco documentado

Introducción a V8

- Existen varios frameworks basados en V8
  - NodeJS
  - SilkJS
  - SorrowJS
  - Etc…
- Esquemas estandarizados para extender V8
  - FunctionTemplate
  - ObjectTemplate
  - InstanceTemplate
  - Accessors

Introducción a V8

- El compilador Just-In-Time ensambla JavaScript a código ejecutable
- Dos modos:
  - No optimizado: rápido de producir, no optimo.
  - Optimizado: más costoso, solo para funciones muy utilizadas.
- Todo el código comienza no optimizado y el compilador re-escribe las funciones

Introducción a V8

- El código optimizado asume información sobre el tipo y la estructura de los objetos
- Si alguna asunción falla se descarta la optimización y vuelve a no-optimizado (Bailout)
- Por más info ver en Google IO 2012 sobre como evitar bailouts en V8

Contextos

Implementación de Pinocchio

V8 en Pin

- Pin impone limitaciones sobre qué se puede hacer en C++
  - try/catch/throw
  - Inicializadores globales
  - atexit()
- El acceso a la API de Windows también está limitado
  - CreateThread no puede usarse
    (pero existe PIN_SpawnInternalThread)

V8 en Pin

- En V8 todo lo especifico a la plataforma existe en un archivo *platform-win32-pin.cc*
  - Crear Threads
  - Thread Local Storage
  - Semáforos
  - Mutex

V8 en Pin

- Con estas modificaciones ya podemos crear un contexto:
  - Isolate::New()
  - Isolate::Scope iscope(isolate)
  - Locker lock(isolate)
  - HandleScope hscope
  - Context::New()
  - Context::Scope cscope(context)
  - Profit!

Extendiendo V8

- Existen Templates para hace accesibles objetos de C++ en JavaScript

```
Tmpl = FunctionTemplate::New(ExternalPointerConstructor);

ObjTmpl = Tmpl->InstanceTemplate();

ObjTmpl->SetAccessor(String::New("pointer"),
  GetPointerAddress, SetPointerAddress);

Global->Set(String::New("ExternalPointer"), Tmpl-
  >GetFunction());
```

```
var ptr = new ExternalPointer(0x12345678);
var addr = ptr.pointer;
ptr.pointer = 0xcafecafe;
```

Extendiendo V8

- En C++ las funciones encapsuladas reciben un objeto como único parámetro:

  const Arguments&

- Para funciones que tienen que ejecutarse varios millones de veces no es práctico
  - Overhead para construirlo
  - Overhead para leerlo

Funciones Inline en V8

- La implementación en JavaScript de tipos básicos utiliza “inline”s:

```
 (!%_IsSmi(length) ||
%EstimateNumberOfElements(object) < (length
>> 2));
```

Funciones Inline en V8

```
  void FullCodeGenerator::EmitIsSmi(CallRuntime* expr) {
 ZoneList<Expression*>* args = expr->arguments();
 ASSERT(args->length() == 1);

 VisitForAccumulatorValue(args->at(0));

 Label materialize_true, materialize_false;
 Label* if_true = NULL;
 Label* if_false = NULL;
 Label* fall_through = NULL;
 context()->PrepareTest(&materialize_true, &materialize_false,
                        &if_true, &if_false, &fall_through);

 PrepareForBailoutBeforeSplit(expr, true, if_true, if_false);
 __ test(eax, Immediate(kSmiTagMask));
 Split(zero, if_true, if_false, fall_through);

 context()->Plug(if_true, if_false);
}
```

```
MOV EAX, [ESP+XX]

TEST EAX, 1

JZ true

MOV EAX, 0
JMP done

:true
MOV EAX, 1
:done
```

Punteros entre V8, PIN y JS

- 3 objetos en javascript diferentes
  - ExternalPointer(<address>)
    - Wrappea un puntero en un objeto JS
    - Generalmente un puntero a memoria no controlada por V8
  - OwnPointer(size)
    - Alloca un buffer y lo wrappea en un objeto JS
    - La vida del buffer está controlada por el GC del objeto JS
  - OwnString(string)
    - Crea un string de C++ y lo wrappea en un objeto JS
    - Pin utiliza strings de C++ pero V8 no

Enteros entre V8, PIN y JS

- V8 tiene 2 tipos de enteros
  - SMI (Small Integer): Entero **con** signo de 31bits
  - HeapNumber: Objeto conteniendo un double de 53bits

- 2 funciones Inline implementadas en assembler
  - ReadInteger
    - Lee un SMI u obtiene los 32bits menos significativos de un objeto HeapNumber.
  - CreateInteger
    - Taggea a SMI si el entero cabe o crea un HeapNumber y asigna los 32bits menos significativos.

Ejemplos

```
FUN(PIN_GetPid)
{
    const int argument_count = 0;
    __ PrepareCallCFunction(argument_count, ebx);
    __ CallCFunction( ExternalReference::PIN_GetPid_function(isolate()), argument_count);
    __ mov(ecx, eax);
    __ CreateInteger(ecx, edi, eax, ebx);
    context()->Plug(edi);
}
```

Ejemplos

```
FUN(BBL_Size)
{
    ZoneList<Expression*>* args = expr->arguments();
    ASSERT(args->length == 1);

    VisitForAccumulatorValue(args->at(0));
    __ ReadInteger(eax);

    const int argument_count = 1;

    __ PrepareCallCFunction(argument_count, ebx);
    __ mov (Operand(esp, kPointerSize * 0), eax);
    __ CallCFunction(ExternalReference::BBL_Size_function(isolate()), argument_count);
    __ mov(ecx, eax);
    __ CreateInteger(ecx, edi, eax, ebx);
    context()->Plug(edi);
}
```

Proxies de Eventos

- Tradicionales
  - Start/Fini Thread
  - Load/Unload Image
  - Start/Fini App
  - Detach
- Rápidos
  - Routine
  - Trace
  - Instruction
- Analysis Function

Eventos Tradicionales

- Proxy en C++
  - Entra al contexto JS principal
  - Prepara los argumentos del evento
  - Llama a un dispatcher en JS
- Dispatcher JS
  - Prepara los argumentos para los callbacks
  - Itera sobre un array de objetos Event
  - Verifica que el evento este enabled
  - Llama al callback con los argumentos

Eventos Tradicionales

```
global.startThreadDispatcher = function(tid, external, flags) {
    var name = "startthread";
    var ctx = new $Context(external);
    if (!IS_NULL_OR_UNDEFINED($EventList[name])) {
        $current.thread = global.threads[tid];
        for (var idx in $EventList[name]) {
            var evt = $EventList[name][idx];
            if (evt._enabled) {
                $current.event = evt;
                 evt.callback(tid, ctx, flags);
}
        }
        $current.event = {};
        $current.thread = {};
    }
}
```

Eventos Rápidos

- Proxy C++
  - Solo se agrega cuando se usa
  - Fast bailout en C++ si no hay eventos para ejecutar
  - Entra Contexto principal
  - Llama Dispatcher en JS
- Dispatcher JS
  - Los objetos de argumento están pre-creados
  - Solo se asigna el recurso al que apuntan
  - Itera sobre un Array de callbacks habilitados
  - Llama al callback

Eventos Rápidos

```
global.fastDispatcher_0 = function(external) {
    $fastEvents_0_arg.external = external;
    for (var idx in $fastEvents_0)
        $fastEvents_0[idx]($fastEvents_0_arg);
}
```

Analysis Function

- Se ejecutan en paralelo en aplicaciones multithread
- Un Isolate/Context por thread
- No puede comunicarse entre threads directamente
  - Shared Memory
  - JSON
- Utiliza 3 funciones
  - Constructor: se ejecuta una vez por thread
  - Destructor: se ejecutan a terminar un thread
  - Callback: se ejecuta por cada evento
- Callback recibe argumentos definidos por el usuario

Analysis Function

- Objeto ArgsArray:
  - Construye los parámetros que recibe una AF
  - Reutilizable en más de una AF
  - Recibe constantes de tipo IARG_TYPE de Pin

args = new ArgsArray(*IARG_INST_PTR*, [*IARG_ADDRINT*, 0xcafecafe]);

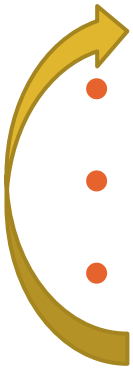
af = new AnalysisFunction(ctor, callback, null, args);
af2 = new AnalysisFunction(ctor, othercback, null, args);

Analysis Function

- Proxy C++
  - Construye un pseudo Context-Switch
  - Entra al Contexto JS
  - Entra al Dispatcher de Analysis Functions
  - Context-Switch a C++ desde JS

Analysis Function

Javascript



- Context-Switch a C++
- Recibe argumentos
- Llama a un Callback

C++

- Bailout si esta desactivada
- Envia argumentos
- Context-Switch a JS

Conclusiones

Overhead

- Proxy C++
  - Entrar al Contexto JS
  - Preparar los argumentos
  - Llamar al Callback en JS
- Acceso API Pin
  - Leer Integers/Punteros encapsulados
  - Hacer llamada Inline
  - Encapsular la respuesta

Futuro

- Terminar de soportar toda la API de Pin
- Desarrollar medios automáticos para compartir información entre Isolates
- Posibilidad de attachar un debugger
- Hacer interface para XED (disassembler)
- Hacer librería de RE para JS
- Documentar!
- Portar a Linux y x64

Conclusiones

- Es una solución práctica y autocontenida para Pin
- El overhead por usar JS está contenido por las optimizaciones
- Work in progress!

Links

- Repositorio: http://www.github.com/pablosole/pet
- Email: pablo.sole@gmail.com

Muchas Gracias!

pablo.sole@gmail.com