BARFing Gadgets

Christian Heitman
cnheitman at fundacionsadosky . org . ar

Programa STIC
Fundación Sadosky

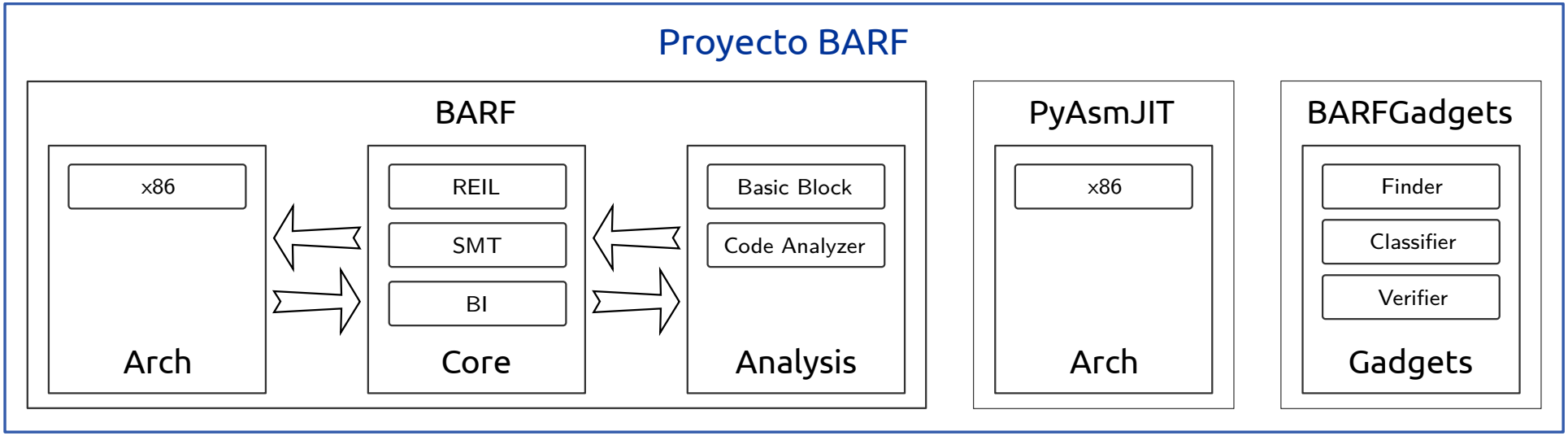
Análisis de Binarios

- Inspeccionar binarios para conocer su seguridad y encontrar vulnerabilidades
- Análisis de malware
- Ingeniería inversa para entender cómo funciona algo
- Tareas poco automatizables
- Existen muchas herramientas pero tiene limitaciones:
  - Licencias costosas
  - Soportan arquitecturas/OSs específicos, etc.
- IDA Pro, Hopper, OllyDbg, ImmDbg, WinDbg, Radare, etc
- BAP, BitBlaze, Jakstab, etc

BARF – Binary Analysis and Reverse Engineering Framework

- Orientado a la asistencia del usuario
- Diseño extensible
- Multiplataforma
- Código abierto
- Escrito en un lenguage de fácil uso (**Python**)
- Encapsular técnicas/métodos/algoritmos para analizar binarios de manera genérica
- **url:** http://www.github.com/programa-stic/barf-project
- Licencia BSD 2-Clause

BARF – Binary Analysis and Reverse Engineering Framework



Dependencias	
Capstone	PyBFD
Z3	SMTLIBv2 (PySymEmu)
CVC4	PEFile

BARF – Binary Analysis and Reverse Engineering Framework

- Se divide en **3** componentes principales:
  - **Core:** Módulos esenciales. Definiciones de **REIL**, **SMT** e interfaz con binarios.
  - **Arch:** Cada submódulo describe una arquitectura particular. Por el momento, **x86**. Próximo paso, **ARM**.
  - **Analysis:** Módulos de análisis. Independientes de la arquitectura. Al momento: reconstrucción CFG y analizador de código (mediante SMT).
- También, un paquete para poder ensamblar y ejecutar **on-the-fly** código assembler. Usado para testear traducción **x86 → REIL**
- El último, una herramienta para encontrar gadgets. Más de esto después...

Lenguaje de Representación Intermedia

- Permite independizarse de la arquitectura
- Generar algoritmos de análisis que pueden usarse para cualquier binario
- Esto requiere poder traducir de una arquitectura **X** a un lenguaje intermedio **Y** (*Binary Translation*)
- Hay varias opciones, en este caso elegimos **REIL**

REIL – Reverse Engineering Intermediate Language

- Es un lenguage reducido tipo RISC de **3 operandos**
- Sólo tiene **17** instrucciones
- Fácil de entender
- Usado en BinNavi, herramienta de análisis estático de programas binarios (desarrollada por Zynamics)
- Algunos ejemplos:
  - **`add [DWORD t0, DWORD t1, DWORD t2]`**
  - **`bisz [DWORD t0, EMPTY, DWORD t1]`**
- **url:** http://www.zynamics.com/downloads/csw09.pdf

REIL – Set de Instrucciones

Arithmetic (6)	Bitwise (3)
add	and
sub	or
mul	xor
div	**Conditional (2)**
mod	bisz
bsh	jcc
Data Transfer (3)	**Other (3)**
ldm	undef
stm	unkn
str	nop

- **REIL** tiene algunas **limitaciones**, por ejemplo:
  - No tiene soporte para instrucciones de punto flotante
  - Tampoco para instrucciones como system calls, interrupts, etc.

Asm → REIL

- Traducción de una architectura **X** a **REIL**
- Traducción de *side-effects*
- Muchas instrucciones hacen más de lo que su nombre indica
- Por ejemplo, en **x86** la instrucción:
  - **`add`:** Además de sumar 2 registros actualiza **6 flags**
  - **`push/pop`:** Además de poner y sacar cosas en la pila, actualiza el registro **`esp`**
- Es importante que en el **LI** cada instrucción tenga un solo efecto sobre el estado del programa
- Todo debe quedar expresado en el lenguaje intermedio

SMT Solvers

- Un **SAT solver** permite encontrar asignaciones para fórmulas booleanas que la hagan verdadera
  - `(X OR Y OR Z) AND (W OR NOT V) AND (NOT W OR NOT Z)`
- Por ejemplo:
  - *X* = True, *Y* = True, *Z* = False, *W* = True, *V* = False
  - *X* = False, *Y* = False, *Z* = True, *W* = False, *V* = False
  - Etc...
- Los **SMT solvers** son una extensión de los SAT solvers
- Permiten trabajar con otro tipo de fórmulas: aritmética de enteros, punto flotante, aritmética módulo N, etc
- Por ejemplo, qué valores de X, Y, Z hacen que la siguiente fórmula sea verdadera?
  - `(X + 2 * Y >= Z) AND (Z >= 2) AND (Y + Z >= 6)`

SMT Solvers

- Permiten representar código assembler de manera muy natural mediante la teoría de la aritmética de módulo N (*BitVec*)
- Permiten razonar sobre fragmentos de código
- Por ejemplo,
  - **`add eax, ebx; or eax, ecx`**
- Se traduce a fórmulas como:
  - **`(eax_1 = eax_0 + ebx_0) ∧ (eax_2 = eax_1 | ecx_0)`**
- Podemos preguntar, ¿qué valores iniciales deben tener **`eax`**, **`ebx`** y **`ecx`** para que el valor final de **`eax`** sea **`0x12345678`**?

REIL → SMT

- REIL es fácilmente expresable mediante fórmulas pues:
  - Son solo 17 instrucciones
  - Cada instrucción tiene un solo efecto en el estado del programa
- Por ejemplo, **`add [DWORD t0, DWORD t1, DWORD t2]`**
- Se representa como (lenguaje *smt-lib*):
  - **`(= t2 (bvadd t0 t1))  ; t2 = t1 + t0`**
- Se pueden agregar restricciones sobre los operandos:
  - **`(bvuge t1 #x000000ff)  ; t1 >= 255`**
  - **`(= t2 #x12345678)  ; t2 == 0x12345678`**
- Y preguntar si existen valores de **`t0, t1, t2`** que las cumplan

Asm → LI → SMT

- Podemos representar porciones de código assembler, por ejemplo, en **x86**: **`add eax, ebx`**

Traducción a REIL

```
add   [DWORD eax, DWORD ebx, QWORD t2]
and   [DWORD eax, DWORD 0x80000000, DWORD t3]
and   [DWORD ebx, DWORD 0x80000000, DWORD t4]
and   [QWORD t2, QWORD 0x80000000, DWORD t5]
xor   [DWORD t3, DWORD t4, DWORD t6]
xor   [DWORD t6, DWORD 0x1, DWORD t7]
xor   [DWORD t3, DWORD t5, DWORD t8]
and   [DWORD t7, DWORD t8, DWORD t9]
bsh   [DWORD t9, DWORD 0xfffffffe1, BYTE of]
and   [QWORD t2, QWORD 0x80000000, QWORD t10]
bsh   [QWORD t10, QWORD 0xffffffffffffffe1, BYTE sf]
and   [QWORD t2, QWORD 0xffffffff, QWORD t11]
bisz  [QWORD t11, EMPTY, BYTE zf]
and   [QWORD t2, QWORD 0x100000000, QWORD t12]
bsh   [QWORD t12, QWORD 0xffffffffffffffe0, BYTE cf]
str   [QWORD t2, EMPTY, DWORD t1]
str   [DWORD t1, EMPTY, DWORD eax]
```

Traducción a smtlibv2

```
(= t2_1 (bvadd ((_ zero_extend 32) eax_0)
               ((_ zero_extend 32) ebx_0)))
(= t3_1 (bvand eax_0 #x80000000))
(= t4_1 (bvand ebx_0 #x80000000))
(= t5_1 ((_ extract 31 0) (bvand t2_1 #x0000000080000000)))
(= t6_1 (bvxor t3_1 t4_1))
(= t7_1 (bvxor t6_1 #x00000001))
(= t8_1 (bvxor t3_1 t5_))
(= t9_1 (bvand t7_1 t8_1))
(= of_1
   (ite (bvsge #xfffffe1 #x00000000)
        ((_ extract 7 0) (bvshl t9_1 #xfffffe1))
        ((_ extract 7 0) (bvlshr t9_1 (bvneg #xfffffe1)))))
(= t10_1 (bvand t2_1 #x0000000080000000))
(= sf_1
   (ite (bvsge #xffffffffffffffe1 #x0000000000000000)
        ((_ extract 7 0) (bvshl t10_1 #xffffffffffffffe1))
        ((_ extract 7 0)
             (bvlshr t10_1 (bvneg  #xffffffffffffffe1)))))
(= t11_1 (bvand t2_1 #x00000000ffffffff))
(= zf_1 (ite (= t11_1 #x0000000000000000) #x01 #x00))
(= t12_1 (bvand t2_1 #x0000000100000000))
(= cf_1
   (ite (bvsge #xffffffffffffffe0 #x0000000000000000)
        ((_ extract 7 0) (bvshl t12_1 #xffffffffffffffe0))
        ((_ extract 7 0)
              (bvlshr t12_1 (bvneg #xffffffffffffffe0)))))
(= ((_ extract 31 0) t2_1) t1_1)
(= t1_1 eax_1)
```

Return-Oriented Programming

- Técnica de explotación de software
- Permite ejecutar de código de manera arbitraria sin necesidad de inyectar código
- Se logra a través de la concatenación de secuencias de pocas instrucciones llamadas *gadgets*
- Ejemplos:
  - `mov eax, ebx; ret`
  - `mov ebx, 0; pop ebp; ret`
  - `xor ecx, ecx; add ebx, 0xAF2D; ret`
- Es una técnica indispensable para lograr la explotación debido a los mecanísmos de protección de los sistemas operativos actuales (DEP)

BARFgadgets

- Es una herramienta para encontrar, clasificar y verificar *ROP gadgets* de manera **automática**
- Permite categorizar *gadgets* de acuerdo a su semántica
- Paso previo a la generación automática de payloads (compilar a ROP)
- Por el momento, solo soporta **x86** (32/64 bits) aunque las etapas de **clasificación** y **verificación** son independientes de la arquitectura
- En este caso, **BARF** provee:
  - Traducción de **x86** → **REIL**
  - Emulación de **REIL**
  - Traducción de **REIL** → **SMT**
  - Interacción con **SMT solvers** (módulo *Code Analyzer*)

BARFgadgets - Clasificación

Tipo	Descripción
Cargar una Constante	$reg_{dst} \leftarrow valor$
Copiar un Registro	$reg_{dst} \leftarrow reg_{src}$
Operación Binaria	$reg_{dst} \leftarrow reg_{src1}\ Op\ reg_{src2}$
Escribir a Memoria	$mem[reg_{addr}+offset] \leftarrow reg_{src}$
Leer de Memoria	$reg_{dst} \leftarrow mem[dst_{addr}+offset]$
Leer y Operar	$reg_{dst} \leftarrow reg_{src}\ Op\ mem[reg_{addr}+offset]$
Operar y Escribir	$mem[reg_{addr}+offset] \leftarrow reg_{src}\ Op\ mem[dst_{addr}+offset]$

- Se pueden agregar otras clasificaciones sin mucho esfuerzo

BARFgadgets - Clasificación

- Se realiza a través de la emulación de código
  - Se genera un contexto random
  - Se ejecuta el *gadget* (en REIL)
  - Se analiza el contexto inicial y el final
  - Se repite varias veces.
- Por ejemplo, **`add eax, ebx ; pop ebp ; ret`**
- Si siempre que se ejecuta resulta que:

$$eax_{final} = eax_{inicial} + ebx_{inicial}$$

- Entonces, se clasifica como un *gadget aritmético*
- La emulación no es suficiente para afirmar que la semántica del gadget efectivamente sea la asignada
- Hace falta realizar una verificación mediante un **SMT solver**

BARFgadgets - Verificación

- Se traduce el *gadget* a fórmulas
- Se agregan restricciones específicas para el tipo de *gadget* que se está verificando
- Se busca encontrar un contraejemplo a la clasificación previamente establecida
- Siguiendo con en el ejemplo anterior, ¿Existe algún valor inicial de **eax** y **ebx** que haga que el valor final de **eax** sea distinto de la suma de ambos?
- Si no existe ningún valor inicial, podemos estar seguros que la semántica del *gadget* es la correcta

BARFgadgets

Demo

BARFgadgets – Lo que viene...

- Sintetizar *gadgets*, por ejemplo, a partir de:
  - `mov eax, ebx ; ret`
  - `mov ebx, 0x0 ; pop ebp ; ret`
- Generar:
  - `eax ← 0x0`
- Búsqueda semántica, es decir, poder buscar cosas del estilo:
  - $\texttt{r32}_{\texttt{dst}} \leftarrow \texttt{r32}_{\texttt{src}} + \texttt{mem[r32}_{\texttt{addr}} + \texttt{offset]}$
- Y que devuelva:
  - `add eax, [ebx + 0x8] ; ret`
  - `mov ebx, [ecx] ; add eax, ebx ; ret`
  - Etc.
- Soporte para ARM

Gracias!