Compilador ROP

Christian Heitman

Fundación Dr. Manuel Sadosky

Septiembre de 2013

Ministerio de Ciencia, Tecnología e Innovación Productiva
Presidencia de la Nación

fundación SADOSKY
Investigación y Desarrollo en TIC

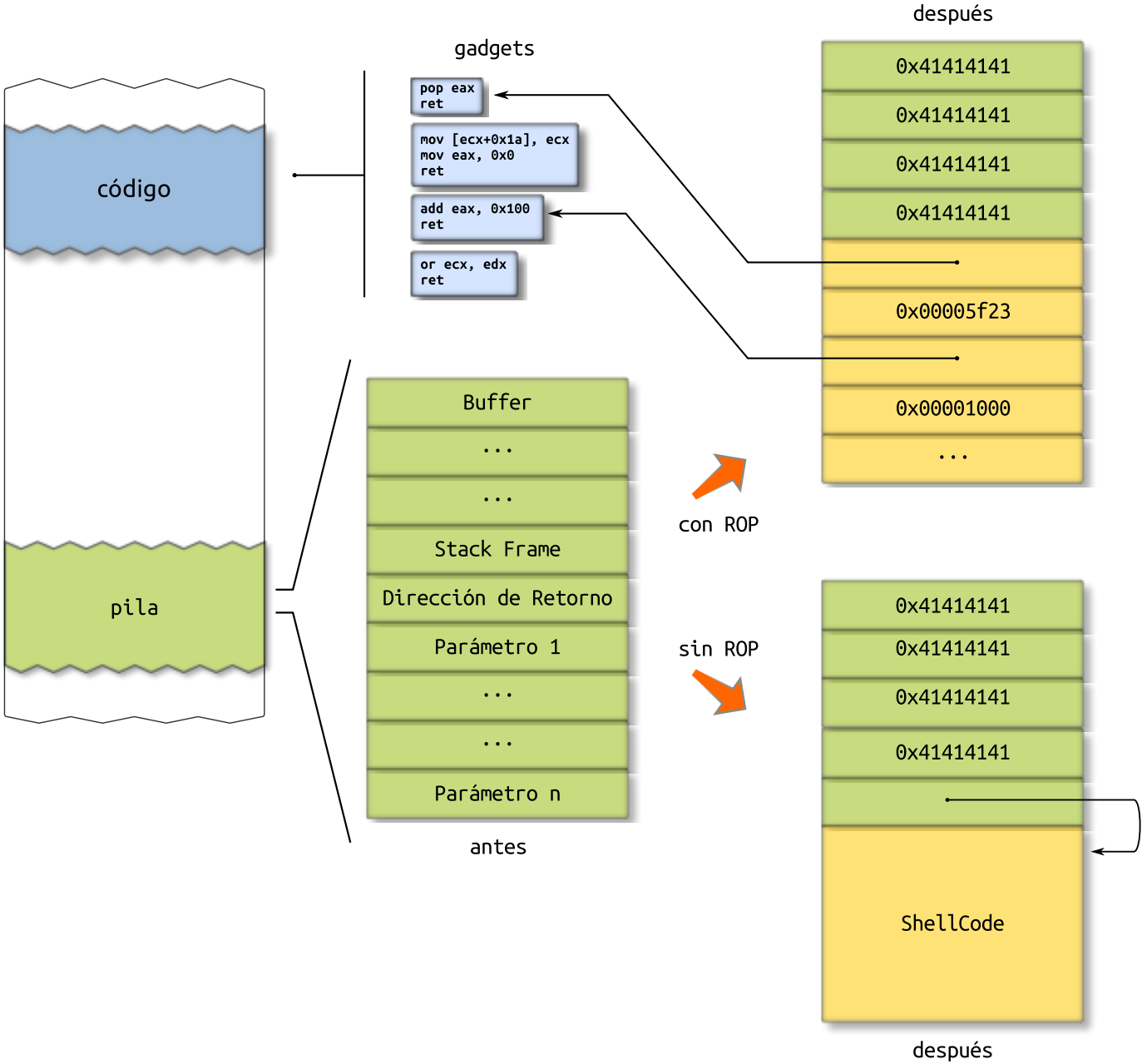
cessi ArgenTIna

CICOMRA

Return Oriented Programming

- Los mecanismos de protección actual previenen la ejecución de código en páginas de datos.
- La técnica ROP permite evadir estos mecanismos.
- Consiste en utilizar fragmentos de código dentro de un binario, llamados gadgets, para computar ciertas operaciones.
- Encadenando varios gadgets se puede armar funciones arbitrarias.

Return Oriented Programming



ROPC

- Es un PoC de un compilador ROP “Turing Complete”.
  - GitHub del autor: `http://github.com/pakt`
- Posee un lenguaje de programación “similar” a C + ASM, llamado ROPL.
  - Saltos condicionales
  - Funciones (soporta recursión)
  - Variables locales
  - Etiquetas
  - Punteros, dereferenciación de memoria, etc.

```
1  fun main(){
2      msg = "Mundo"
3      x = 1
4
5  print:
6      !printf("Hola, %s!\n", msg)
7      x = x + 1
8      cmp x, 10
9      jne print
10 }
11
```

Hola Mundo (ROPL)

ROPC

- Está basado en el paper “Q: Exploit Hardening Made Easy”.
- Está escrito en ‘OCaml’.

```
let msg = "Mundo";;

Printf.printf "Hola, %s!\n" msg;;

```

Hola Mundo

```
let rec sumar lista =
    match lista with
    | []            -> 0
    | elem :: elems -> elem + sumar elems;;

let rv = sumar [1; 2; 3; 4; 5];;

Printf.printf "Suma : %d\n" rv;;

```

Sumar una lista de enteros

Q / BAP

- Q es similar a ROPC pero no está disponible.
- BAP es una plataforma de análisis de binarios, con enfásis en la verificación formal.
- ROPC utiliza BAP para varias cosas:
  - Pasar código x86 a un lenguaje intermedio
  - Ejecución simbólica
  - Interacción con SMT solvers

SMT (Satisfiability Modulo Theories)

- Extensión de SAT solver para trabajar con otros tipos de teorías, por ejemplo, teoría aritmética.
- Con SAT podemos responder cosas del estilo:
  - Existen r, p, q tales que $(r \wedge \neg p) \vee (q \wedge p)$
- Con SMT podemos responder cosas del estilo:
  - Existen x, y tales que $2x + 5y = 12 \wedge x > 2 \wedge y > -1$
- SMT permite modelar fácilmente código binario y podemos preguntar sobre propiedades del mismo
  - Por ejemplo, el gadget g es semánticamente equivalente a una suma?
  - Lo logramos expresando el gadget mediante fórmulas y añadiendo las restricciones que queremos que cumpla.

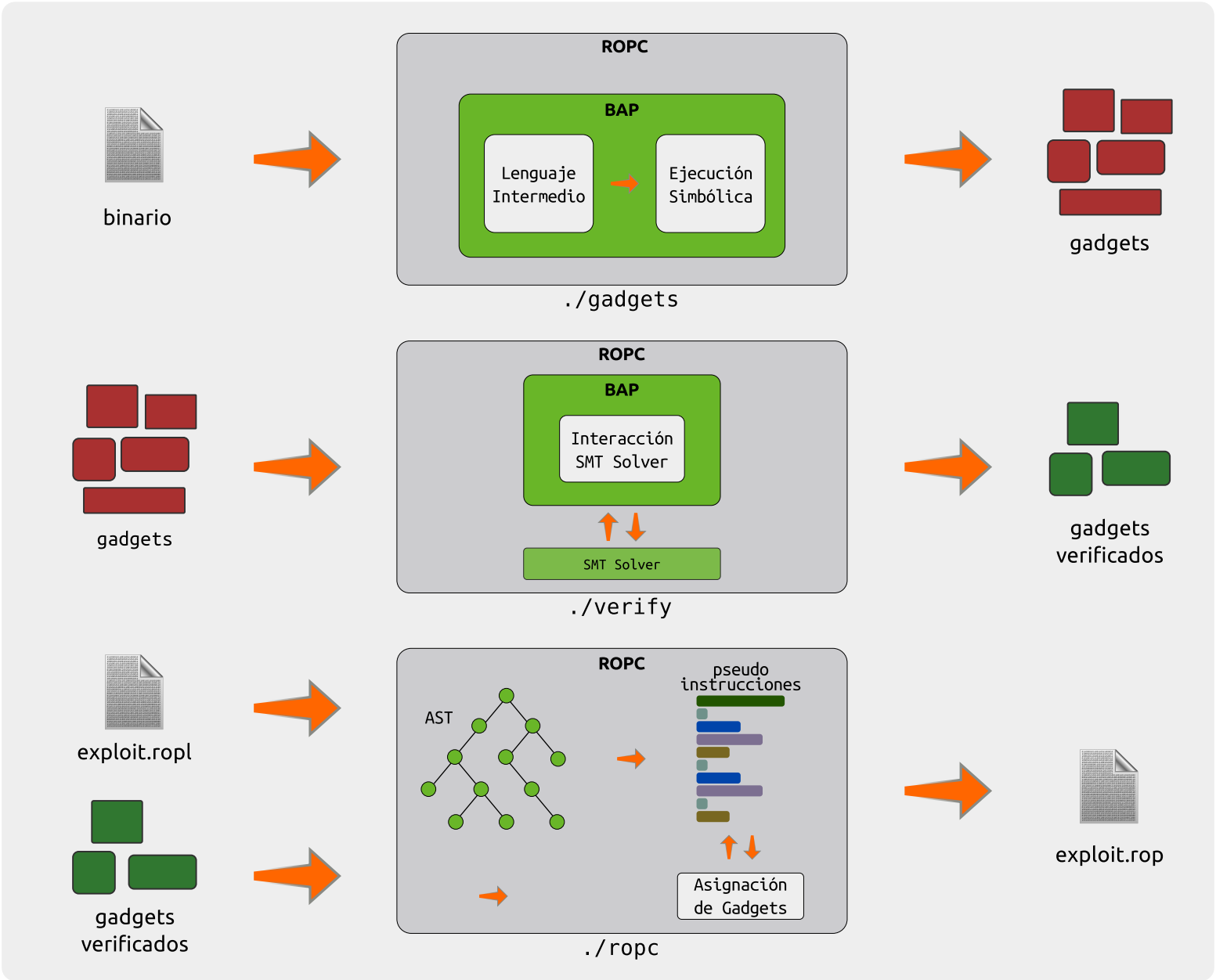
ROPC - Estructura/Etapas

- Gadgets:
  - Descubrimiento : procedimiento estandar
  - Clasificación : mediante ejecución simbólica
    - Cargar una constante : reg $\leftarrow$ valor
    - Copiar un registro : reg_dst $\leftarrow$ reg_src
    - Operación aritmética : reg $\leftarrow$ reg_1 OP reg_2
    - Escribir a memoria : mem[addr + offset] $\leftarrow$ reg
    - Leer de memoria : reg $\leftarrow$ mem[addr + offset]
    - Leer y operar : reg $\leftarrow$ reg OP mem[addr + offset]
    - Operar y escribir : mem[addr + offset] $\leftarrow$ reg OP mem[addr + offset]
  - Verificación : mediante SMT solvers

ROPC - Estructura/Etapas

- Parsing:
  - Parsea un programa en “ROPL” y genera un AST (Abstract Syntax Tree)
- Transformación:
  - Transforma el AST a una lista de pseudo instrucciones y en forma SSA (Static Single Assigment)
- Compilación:
  - Matching casi directo entre instrucciones simplificadas y gadgets
  - Busca una asignación de registros y gadgets que eviten conflictos entre los mismos.

ROPC - Workflow



LLVM

- Conjunto de herramientas para el desarrollo de compiladores.
- Tiene un lenguaje intermedio llamado “LLVM IR”
- Existen front-ends para C/C++/Objective-C, etc que generan “LLVM IR”
- Ejemplo del IR:

```c
#include <stdio.h>

int main(int argc, char * argv[]) {
    char * msg = "Mundo";
    int i = 0;

    while (i < 10) {
        printf("Hola, %s!\n", msg);
        i = i + 1;
    }

    return 0;
}

```

Código C

LLVM

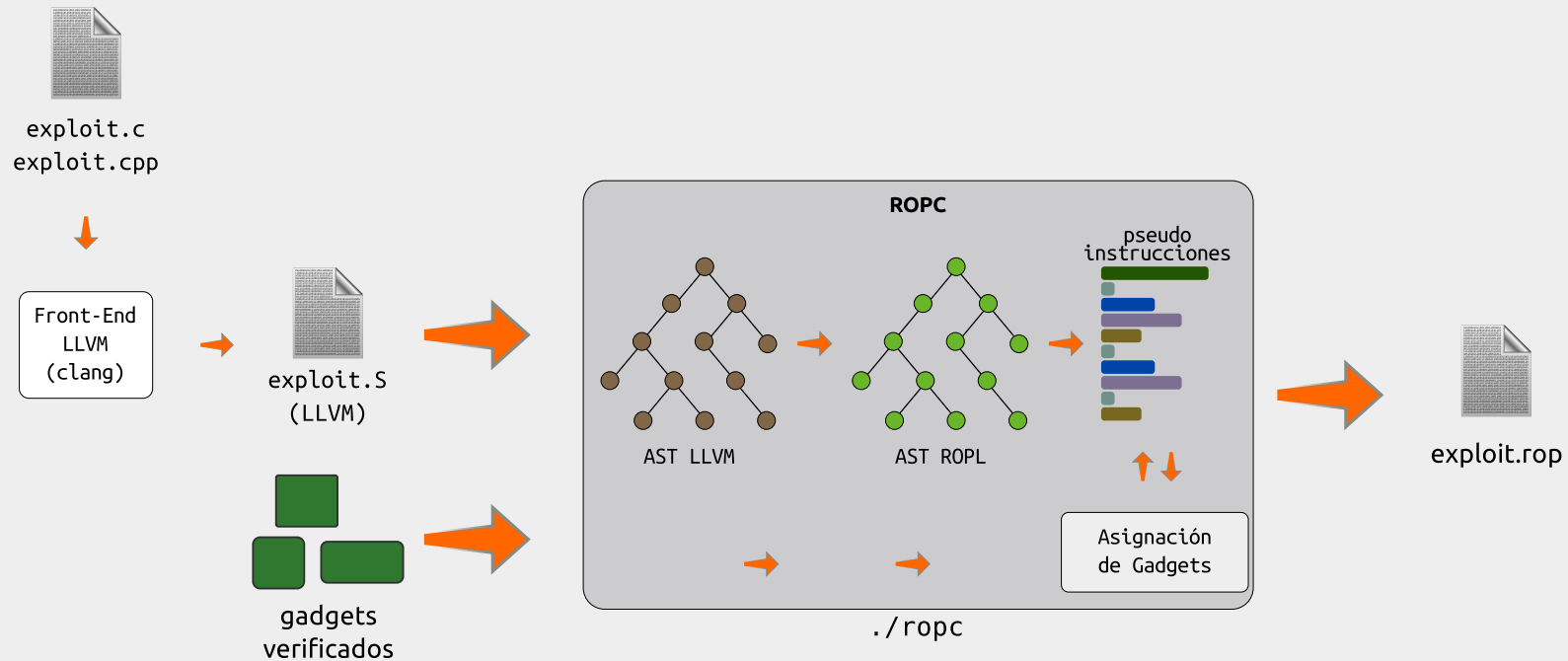
```
; ModuleID = 'hello_world_loop.c'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:..."
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [6 x i8] c"Mundo\00", align 1
@.str1 = private unnamed_addr constant [11 x i8] c"Hola, %s!\0A\00", align 1

define i32 @main(i32 %argc, i8** %argv) nounwind uwtable {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i8**, align 8
  %msg = alloca i8*, align 8
  %i = alloca i32, align 4
  store i32 0, i32* %1
  store i32 %argc, i32* %2, align 4
  store i8** %argv, i8*** %3, align 8
  store i8* getelementptr inbounds ([6 x i8]* @.str, i32 0, i32 0),
    i8** %msg, align 8
  store i32 0, i32* %i, align 4
  br label %4

; <label>:4                                       ; preds = %7, %0
  %5 = load i32* %i, align 4
  %6 = icmp slt i32 %5, 10
  br i1 %6, label %7, label %12

; <label>:7                                       ; preds = %4
  %8 = load i8** %msg, align 8
  %9 = call i32 (i8*, ...)* @printf(
    i8* getelementptr inbounds ([11 x i8]* @.str1, i32 0, i32 0),
    i8* %8)
  %10 = load i32* %i, align 4
  %11 = add nsw i32 %10, 1
  store i32 %11, i32* %i, align 4
  br label %4

; <label>:12                                      ; preds = %4
  ret i32 0
}

declare i32 @printf(i8*, ...)

```

IR LLVM

ROPC - Soporte Básico para LLVM

- La idea es poder usar cualquier lenguaje que tenga un front-end de LLVM para desarrollar un exploit
- Se trató de modificar lo menos posible el código de ROPC.
- La opción eligida fue traducción de ASTs: LLVM $\rightarrow$ ROPL
- Hay ventajas y desventajas:
  - Modificaciones mínimas al código actual
  - Hay que lidiar con algunas limítaciones de ROPL
- Por lo tanto,
  - Se implementó un parser/lexer de LLVM
  - Se implementó un traductor entre el AST de LLVM y el AST de "ROPL".
  - Antes, se hizo refactoring del código y se crearon módulos bien definidos.
- Una vez conseguido eso se pudo compilar ROP partiendo de un .c (bueno, maso. . .)

ROPC - Workflow con LLVM



Parseando LLVM

- LLVM incluye mucha información que, a fines prácticos de lo que estamos haciendo, no es necesaria:
  - Atributos de funciones
  - Información de tipado
  - Metadata
  - Etc.
- Durante el parseo y la generación del AST, se descarta esta información.
- El IR de LLVM es bastante extenso
  - Escribimos programas básicos en C
  - Los compilamos al IR (con clang)
  - Implementamos las instrucciones necesarias para poder compilarlos a ROP.
- El AST generado está pensado para que sea lo más parecido al de ROPL
  - Esto facilita la implementación del traductor de ASTs

Traducción entre ASTs (LLVM $\rightarrow$ ROPL)

- Hay instrucciones que no tiene un matching inmediato
- Hay funcionalidades que no están soportadas en ROPL
- Para estos casos, se hace la traducción en 2 etapas,
  - Simplificación / Reescritura del AST de LLVM
  - Resolución
- Por ejemplo,
  - Branches:
    - Consta de 2 instrucciones diferentes, comparación y salto, la condición de salto está en una y las etiquetas en otra.
  - GetElementPtr:
    - Es una instrucción “compleja” para calcular el offset dentro de un arreglo / estructura.
  - Return:
    - La instrucción “return” de ROPL no retorna valores.

ROPC - Ejemplo

```
1  fun fib(n, out){
2      x = 0
3      y = 0
4
5      cmp n, 0
6      je copy
7      cmp n, 1
8      je copy
9
10     fib(n-1, @x)
11     fib(n-2, @y)
12
13     [out] = x+y
14     jmp exit
15 copy:
16     [out] = n
17 exit:
18 }
19
20 fun main(){
21     fmt = "%d\n"
22     i = 0
23     x = 0
24 print:
25     fib(i, @x)
26     !printf(fmt, x)
27     i = i+1
28     cmp i, 11
29     jne print
30 }
31
```

fib.ropl

```
1  Assign(x, 0)
2  Assign(y, 0)
3  Cmp(n, 0)
4  Branch([e], copy)
5  Cmp(n, 1)
6  Branch([e], copy)
7  Call(fib, BinOp(n,-,1),&x)
8  Call(fib, BinOp(n,-,2),&y)
9  WriteMem(out, BinOp(x,+,y))
10 Branch([@], exit)
11 Label(copy)
12 WriteMem(out, n)
13 Label(exit)
14
15 # Fun: main, args:
16 AssignTab(fmt, ;37;100;10;0)
17 Assign(i, 0)
18 Assign(x, 0)
19 Label(print)
20 Call(fib, i,&x)
21 !Call(printf, fmt,x)
22 Assign(i, BinOp(i,+,1))
23 Cmp(i, 11)
24 Branch(~[e], print)
25
```

fib.ropl (parseado)

ROPC con Soporte para LLVM - Ejemplo

```c
void fib(int n, int * out) {
    int x = 0;
    int y = 0;

    if (n == 0 || n == 1) {
        *out = n;

        return;
    }

    fib(n-1, &x);
    fib(n-2, &y);

    *out = x + y;

    return;
}

int main() {
    char fmt[] = "%d\n";
    int i = 0;
    int x = 0;

    while (i != 11) {
        fib(i, &x);
        printf(fmt, x);
        i = i + 1;
    }

    return 0;
}
```

fib.c

```
define void @fib(i32 %n, i32* %out) nounwind uwtable {
  %1 = alloca i32, align 4
  %2 = alloca i32*, align 8
  %x = alloca i32, align 4
  %y = alloca i32, align 4
  store i32 %n, i32* %1, align 4
  store i32* %out, i32** %2, align 8
  store i32 0, i32* %x, align 4
  store i32 0, i32* %y, align 4
  %3 = load i32* %1, align 4
  %4 = icmp eq i32 %3, 0
  br i1 %4, label %8, label %5

; <label>:5                                       ; preds = %0
  %6 = load i32* %1, align 4
  %7 = icmp eq i32 %6, 1
  br i1 %7, label %8, label %11

; <label>:8                                       ; preds = %5, %0
  %9 = load i32* %1, align 4
  %10 = load i32** %2, align 8
  store i32 %9, i32* %10
  br label %20

; <label>:11                                      ; preds = %5
  %12 = load i32* %1, align 4
  %13 = sub nsw i32 %12, 1
  call void @fib(i32 %13, i32* %x)
  %14 = load i32* %1, align 4
  %15 = sub nsw i32 %14, 2
  call void @fib(i32 %15, i32* %y)
  %16 = load i32* %x, align 4
  %17 = load i32* %y, align 4
  %18 = add nsw i32 %16, %17
  %19 = load i32** %2, align 8
  store i32 %18, i32* %19
  br label %20

; <label>:20                                      ; preds = %11, %8
  ret void
}
```

función “fib” (LLVM)

```
define i32 @main() nounwind uwtable {
  %1 = alloca i32, align 4
  %fmt = alloca [4 x i8], align 1
  %i = alloca i32, align 4
  %x = alloca i32, align 4
  store i32 0, i32* %1
  %2 = bitcast [4 x i8]* %fmt to i8*
  call void @llvm.memcpy.p0i8.p0i8.i64(i8* %2,
    i8* getelementptr inbounds ([4 x i8]* @main.fmt, i32 0, i32 0),
    i64 4, i32 1, i1 false)
  store i32 0, i32* %i, align 4
  store i32 0, i32* %x, align 4
  br label %3

; <label>:3                                       ; preds = %6, %0
  %4 = load i32* %i, align 4
  %5 = icmp ne i32 %4, 11
  br i1 %5, label %6, label %13

; <label>:6                                       ; preds = %3
  %7 = load i32* %i, align 4
  call void @fib(i32 %7, i32* %x)
  %8 = getelementptr inbounds [4 x i8]* %fmt, i32 0, i32 0
  %9 = load i32* %x, align 4
  %10 = call i32 (i8*, ...)* @printf(i8* %8, i32 %9)
  %11 = load i32* %i, align 4
  %12 = add nsw i32 %11, 1
  store i32 %12, i32* %i, align 4
  br label %3

; <label>:13                                      ; preds = %3
  ret i32 0
}
```

función “main” (LLVM)

ROPC con Soporte para LLVM - Ejemplo

```
# Fun: fib, args: var_n,var_out
Assign(pointee_var_1, 0)
Assign(var_1, &pointee_var_1)
Assign(pointee_var_2, 0)
Assign(var_2, &pointee_var_2)
Assign(pointee_var_x, 0)
Assign(var_x, &pointee_var_x)
Assign(pointee_var_y, 0)
Assign(var_y, &pointee_var_y)
WriteMem(var_1, var_n)
WriteMem(var_2, var_out)
WriteMem(var_x, 0)
WriteMem(var_y, 0)
Assign(var_3, ReadMem(var_1))
Cmp(var_3, 0)
Branch([e], label_8)
Branch(~[e], label_5)
Label(label_5)
Assign(var_6, ReadMem(var_1))
Cmp(var_6, 1)
Branch([e], label_8)
Branch(~[e], label_11)
Label(label_8)
Assign(var_9, ReadMem(var_1))
Assign(var_10, ReadMem(var_2))
WriteMem(var_10, var_9)
Branch([@], label_20)
Label(label_11)
Assign(var_12, ReadMem(var_1))
Assign(var_13, BinOp(var_12,-,1))
Call(fib, var_13,var_x)
Assign(var_14, ReadMem(var_1))
Assign(var_15, BinOp(var_14,-,2))
Call(fib, var_15,var_y)
Assign(var_16, ReadMem(var_x))
Assign(var_17, ReadMem(var_y))
Assign(var_18, BinOp(var_16,+,var_17))
Assign(var_19, ReadMem(var_2))
WriteMem(var_19, var_18)
Branch([@], label_20)
Label(label_20)

```

función “fib” (parseado)

```

# Fun: main, args:
AssignTab(main.fmt, ;37;100;10;0)
Assign(pointee_var_1, 0)
Assign(var_1, &pointee_var_1)
AssignTab(var_fmt, ;0;0;0;0)
Assign(pointee_var_i, 0)
Assign(var_i, &pointee_var_i)
Assign(pointee_var_x, 0)
Assign(var_x, &pointee_var_x)
WriteMem(var_1, 0)
Assign(var_2, var_fmt)
Assign(offset, 0)
Assign(tmp_var_lbl, BinOp(main.fmt,+,offset))
Call(intrinsic_memcpy, var_2,tmp_var_lbl,4)
WriteMem(var_i, 0)
WriteMem(var_x, 0)
Branch([@], label_3)
Label(label_3)
Assign(var_4, ReadMem(var_i))
Cmp(var_4, 11)
Branch([e], label_13)
Branch(~[e], label_6)
Label(label_6)
Assign(var_7, ReadMem(var_i))
Call(fib, var_7,var_x)
Assign(offset, 0)
Assign(var_8, BinOp(var_fmt,+,offset))
Assign(var_9, ReadMem(var_x))
!Call(printf, var_8,var_9)
Assign(var_11, ReadMem(var_i))
Assign(var_12, BinOp(var_11,+,1))
WriteMem(var_i, var_12)
Branch([@], label_3)
Label(label_13)

```

función “main” (parseado)

ROPC con Soporte para LLVM - Ejemplo

```
christian@laptop:test$ ../bin/gadget a.out candidates.bin > gadget-log 2>&1
christian@laptop:test$ ../bin/verify candidates.bin verified.bin > verify-log 2>&1
christian@laptop:test$ ../bin/ropc ropl examples-ropl/fib.ropl verified.bin > ropc-ropl-log 2>&1
christian@laptop:test$ ls compiled-ropl.bin
compiled-ropl.bin
christian@laptop:test$ ../bin/ropc llvm examples-llvm/fib.S verified.bin > ropc-llvm-log 2>&1
christian@laptop:test$ ls compiled-llvm.bin
compiled-llvm.bin
christian@laptop:test$ ./a.out compiled-ropl.bin
buf=0x09b70170
roundup buf=0x09b80000
0
1
1
2
3
5
8
13
21
34
55
Segmentation fault (core dumped)
christian@laptop:test$ ./a.out compiled-llvm.bin
buf=0x095e2170
roundup buf=0x095f0000
0
1
1
2
3
5
8
13
21
34
55
Segmentation fault (core dumped)
christian@laptop:test$
```

Problemas Encontrados

- Limitaciones de ROPL...
- Variables globales
  - En ROPC, todas las variables son globales
  - Hay una tabla donde están definidas
  - Las variables globales (de LLVM), se inicializan al comienzo del "main" en la traducción.
- Intrinsics de LLVM
  - Funciones conocidas, muy difundidas y con semántica definida:
    - 'memcpy'→ 'llvm.memcpy'
    - 'memmove'→ 'llvm.memmove'
    - 'memset'→ 'llvm.memset'
    - Etc.
  - Pueden aparecer en código LLVM sin, necesariamente, haberlas usado en el código C.

Limitaciones

- Por el momento, es solo un PoC...
- El código ROP generado es muy grande para ser usado en un ambiente de 'producción'.
- Escrito en OCaml, lenguaje no muy difundido...

Trabajo Futuro

- Optimizar código generado
- Soportar un subconjunto más grande de LLVM

Referencias

- ROPC - Soporte para LLVM
  - `http://github.com/programa-stic/ropc-llvm`
- ROPC
  - `http://github.com/pakt/ropc`
- Non-exec stack
  - `http://seclists.org/bugtraq/2000/May/90`
- Future of buffer overflows ?
  - `http://seclists.org/bugtraq/2000/Nov/32`
- “Advanced return-into-lib(c) exploits (PaX case study)”
  - `http://www.phrack.com/issues.html?issue=58&id=4&mode=txt`

Referencias

- “The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)”
  - `http://cseweb.ucsd.edu/~hovav/dist/geometry.pdf`
- “Q: Exploit Hardening Made Easy”
  - `http://users.ece.cmu.edu/~ejschwar/papers/usenix11.pdf`
- BAP : The Next-Generation Binary Analysis Platform
  - `http://bap.ece.cmu.edu/`
- SMT Solvers for Software Security
  - `http://www.usenix.org/system/files/conference/woot12/woot12-final26.pdf`

¡Gracias!

Me contactan en:

cnheitman@fundacionsadosky.org.ar