Everybody be cool, this is a roppery!

Vincenzo Iozzo (vincenzo.iozzo@zynamics.com) zynamics GmbH
Tim Kornau (tim.kornau@zynamics.com) zynamics GmbH
Ralf-Philipp Weinmann (ralf-philipp.weinmann@uni.lu) Université du Luxembourg
BlackHat Vegas 2010

Overview

1. Introduction
2. Gentle overview
3. Finding gadgets
4. Compile gadgets
5. Some fancy demos
6. Further work

Introduction

Exploitation with non-executable pages is not much fun

But we have funny ideas

Exploitation with non-executable pages is not much fun.. Unless you use “return-oriented programming”

Gentle introduction

CAN SEC WEST

NO MORE FREE BUGS

iPhone

But life is hard

Code signing

ROP

Sandboxing

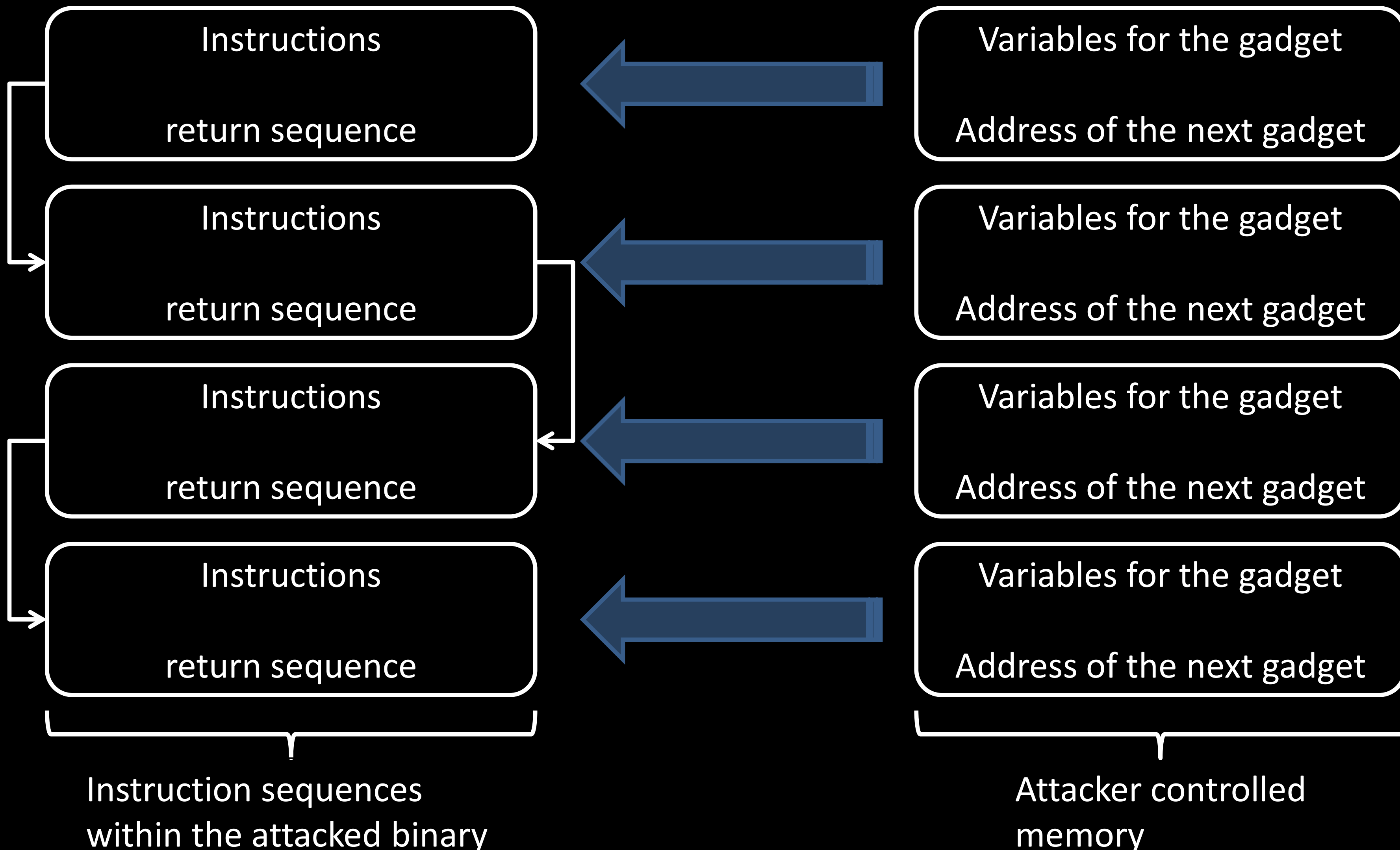
We were lucky!

Code Signing

Used to make sure that only signed (Apple verified) binaries can be executed

- If a page has write permissions it can’t have executable permissions
- No executable pages on the heap
- Only signed pages can be executed

ROP


Instructions
return sequence
Instructions
return sequence
Instructions
return sequence
Instructions
return sequence
Variables for the gadget
Address of the next gadget
Variables for the gadget
Address of the next gadget
Variables for the gadget
Address of the next gadget
Variables for the gadget
Address of the next gadget
Instruction sequences within the attacked binary
Attacker controlled memory

ROP - Workflow

1. Find the gadgets
2. Chain them to form a payload
3. Test the payload on your target

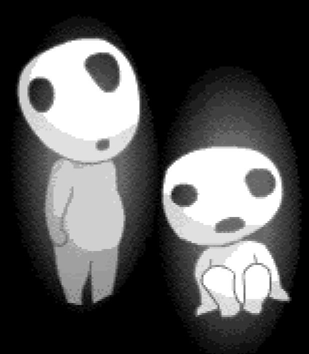
Finding Gadgets Overview

1. Goal definition
2. Motivation
3. Strategy
4. Algorithms
5. Results
6. Further improvement

Goal definition

Build an algorithm which is capable of locating gadgets within a given binary automatically without major side effects.

Motivation I



Little spirits need access to a wide range of devices.
Because what is a device without a spirit?

Motivation II

We want to be able to execute our code:

- in the presents of non-executable protection (AKA NX bit)

- when code signing of binaries is enabled.

- but we do not aim at ASLR.

Strategy I

- Build a program from parts of another program
- These parts are named gadgets
- A gadget is a sequence of (useable) instructions
- Gadget combination must be possible
  - end in a “free-branch”
- Gadgets must provide a useful operation
  - for example A + B

Strategy II

- The subset of useful gadgets must be locatable in the set of all gadgets
- Only the “simplest” gadget for an operation should be used
- Side effects of gadgets must be near to zero to avoid destroying results of previous executed code sequences.
- Use the REIL meta language to be platform independent.

Strategy III

A small introduction to the REIL meta language

- small RISC instruction set (17 instructions)
  - Arithmetic instructions (ADD, SUB, MUL, DIV, MOD, BSH)
  - Bitwise instructions (AND, OR, XOR)
  - Logical instructions (BISZ, JCC)
  - Data transfer instructions (LDM, STM, STR)
  - Other instructions (NOP, UNDEF, UNKN)

- register machine
- unlimited number of temp registers
- side effect free
- no exceptions, floating point, 64Bit, ..

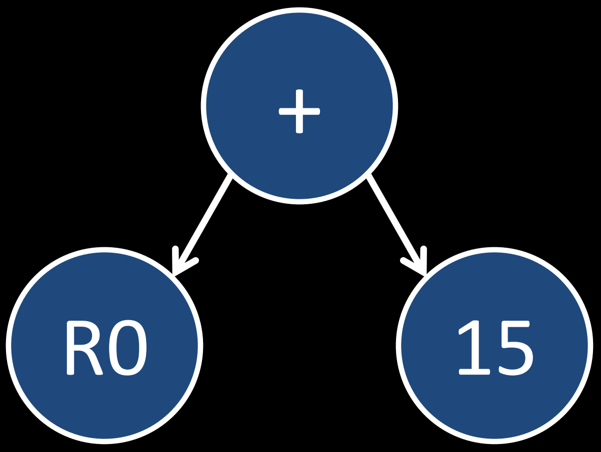
Algorithms

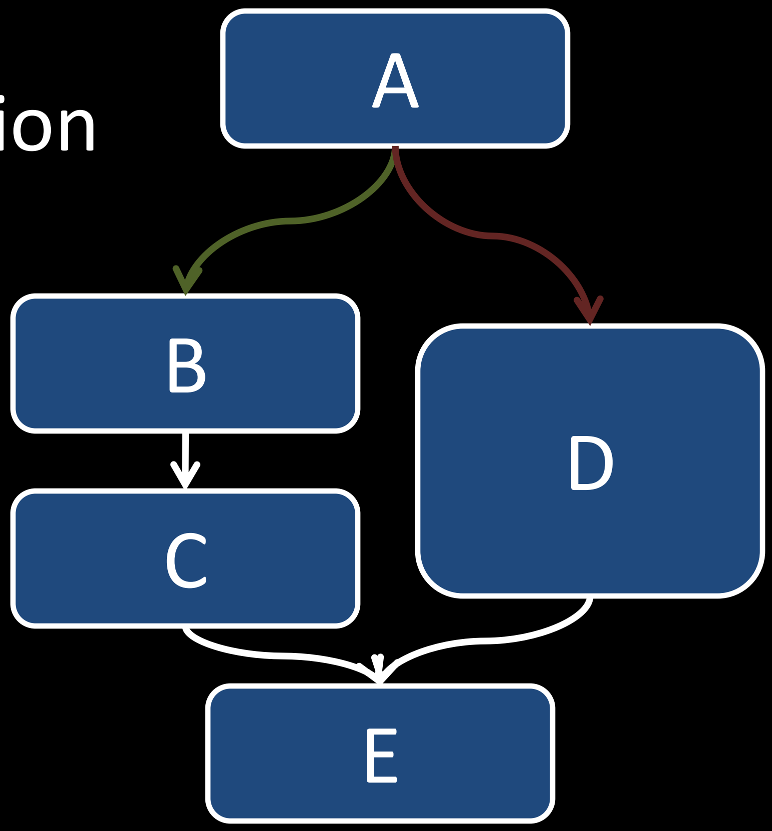
- Stage I → Collect data from the binary
- Stage II → Merge the collected data
- Stage III → Locate useful gadgets in merged data

Algorithms stage I (I)

Goal of the stage I algorithms:

- Collect data from the binary
    1. Extract expression trees from native instructions
    2. Extract path information

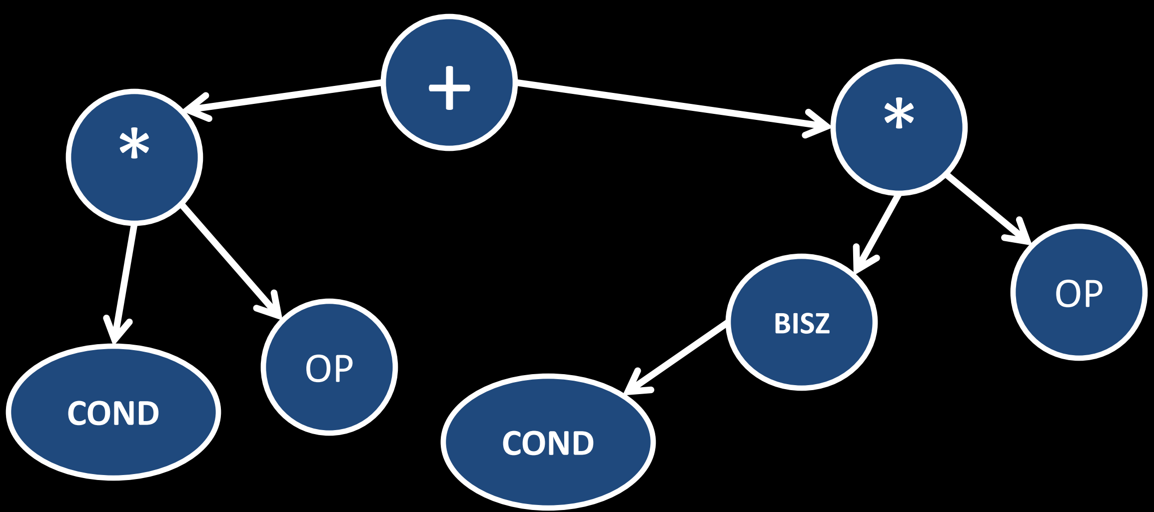




Algorithms stage I (II)

Details of the stage I algorithms:

1. Expression tree extraction
   - Handlers for each possible REIL instruction
     1. Most of the handlers are simple transformations
     2. STM and JCC need to be treated specially
2. Path extraction
   - Path is extracted in reverse control flow order



Algorithms stage II (I)

Goal of the stage II algorithms:

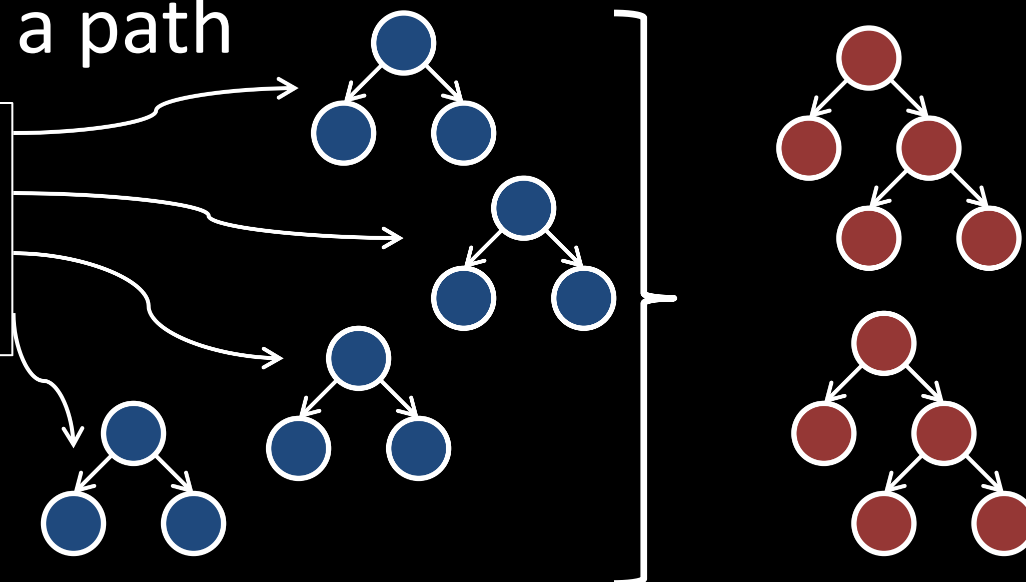
- Merge the collected data from stage I
  1. Combine the expression trees for single native instructions along a path
  2. Determine jump conditions on the path
  3. Simplify the result

Algorithms stage II (II)

Details of the stage II algorithms:

- Combine the expression trees for single native instructions along a path

1. 0x00000001 ADD R0, R1, R2
2. 0x00000002 STR R0, R4
3. 0x00000003 LDMFD SP! {R4,LR}
4. 0x00000004 BX LR



Algorithms stage II (III)

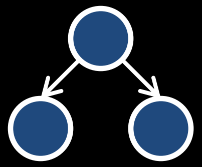
Details of the stage II algorithms:

- Determine jump conditions on the path:

1. 0x00000001 SOME INSTRUCTION
2. 0x00000002 BEQ 0xADDRESS
3. 0x00000003 SOME INSTRUCTION
4. 0x00000004 SOME INSTRUCTION

Z FLAG MUST BE FALSE

Generate condition tree



- Simplify the result:

R0 = ((((((R2+4)+4)+4)+4) OR 0) AND 0xFFFFFFFF)
R0 = R2+16

Algorithms stage III (I)

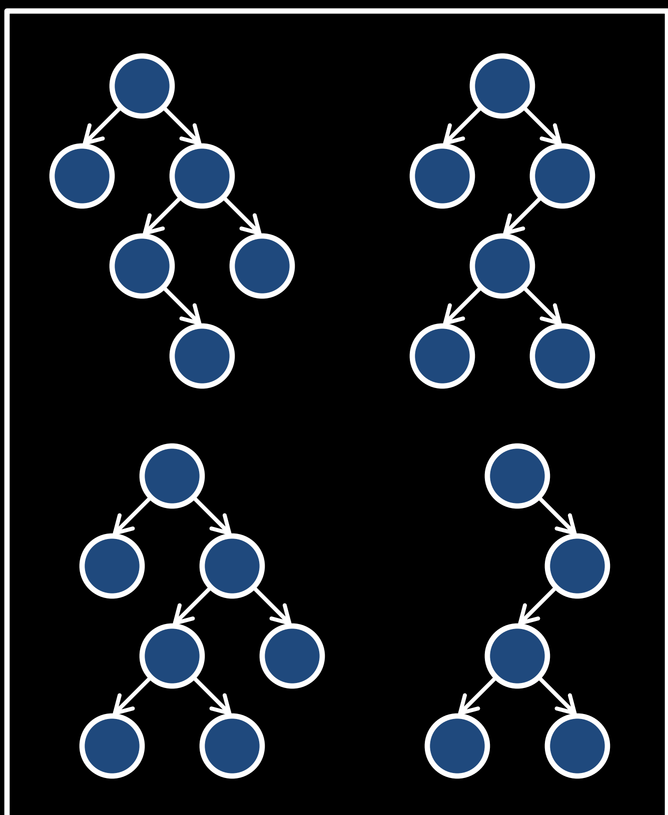
Goal of the stage III algorithms:

- Search for useful gadgets in the merged data
  - Use a tree match handler for each operation.
- Select the simplest gadget for each operation
  - Use a complexity value to determine the gadget which is least complex. (side-effects)

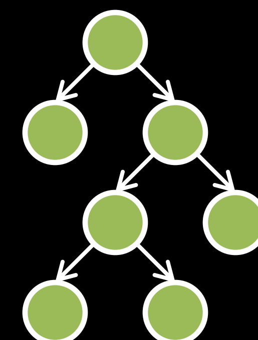
Algorithms stage III (II)

Details of the stage III algorithms:

- Search for useful gadgets in the merged data



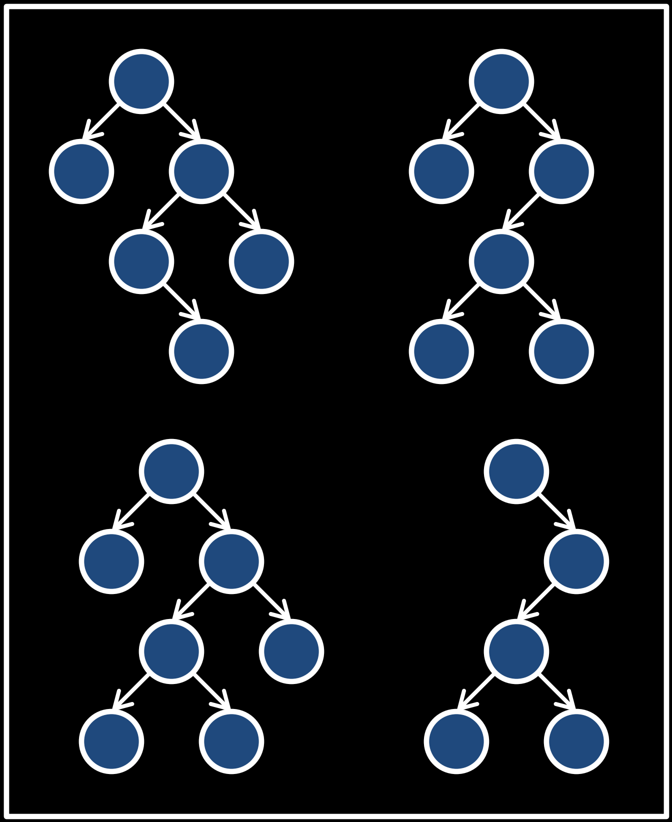
Trees of a gadget candidate are compared to the tree of a specific operation.
Can you spot the match ?

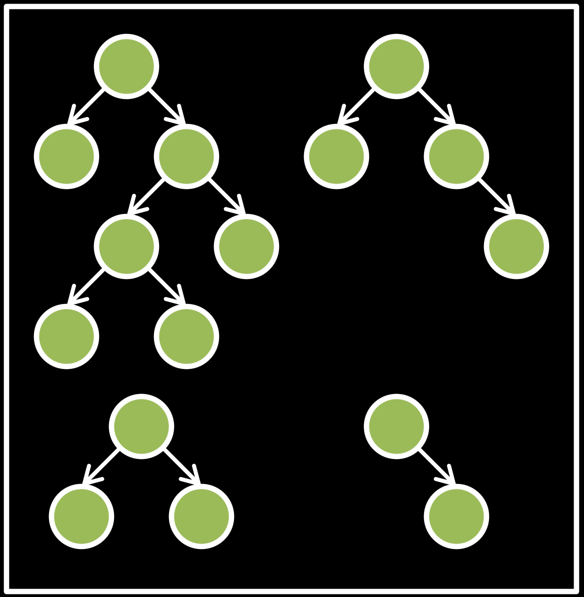


Algorithms stage III (III)

Details of the stage III algorithms:

- Select the simplest gadget for each operation





There are in most cases more instruction sequences which provide a specific operation. The overall complexity of all trees is used to determine which gadget is the simplest.

Results of gadget finding

- Algorithms for automatic return-oriented programming gadget search are possible.
- The described algorithms automatically find the necessary parts to build the return-oriented program.
- Searching for gadgets is not only platform but also very compiler dependent.

So what is next

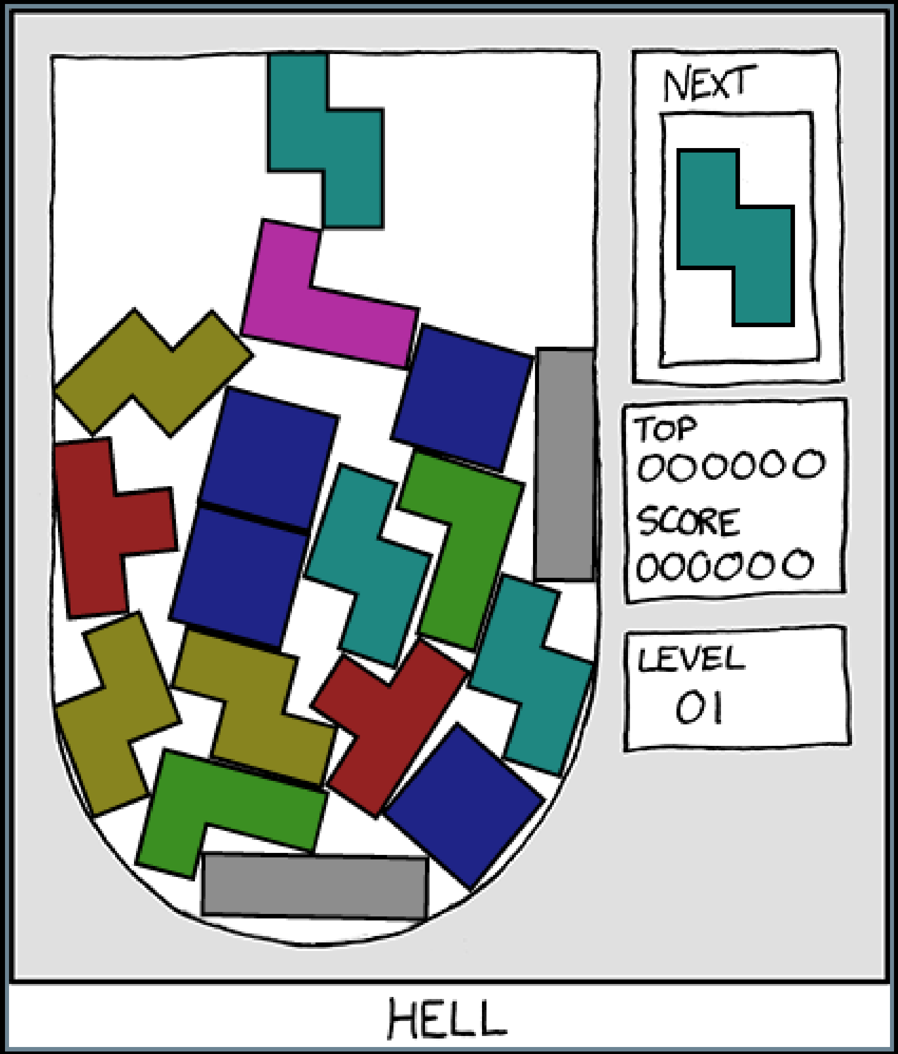
After automatic gadget extraction
we need a simple and effective way
to combine them.

Chaining gadgets

Chaining gadgets

- … by hand is like playing Tetris
- With very ugly blocks
- Each gadget set defines custom ISA
- We have better scores that at...

Chaining gadgets



Chaining gadgets

Hence we have decided to bring in some help...

The Wolf

- A ROP compiler for gadget sets with side-effects
- Very basic language
- Allows for easy ROPperies on ARM devices

Living with side-effects

- “allowread”: specifies readable memory ranges
- “allowcorrupt”: expendable memory ranges
- [corruption may occur here]
- protect: registers must stay invariant
- [SP and PC implicitly guarded]

Statements

- (multi-)assignment
- Conditional goto statement
- Call statement (calling lib functions)
- Data definitions
- Labels for data/code

Multi-assignment

Example from PWN2OWN payload:

```
(r0, r1, r2) <<_| (mem[sockloc], sin, SIZE_SIN)
```

targets

memory read

constant

assignment operator

data reference

Loops

define label for conditional jump

```
label(clear_loop)
r1 = 256
(mem[r0], r2, r1) <<_| (0, (3*r1) & 255, r1-1)
r0 = r0+4
gotoifnz(r1, clear_loop)
```

RHS may contain arithmetic-logical calculations:

{+,-,*,/, %, ^, |, &, <<, >>}

Hired help: STP

- Mr. Wolf is a high-level problem solver: he likes to delegate
- Menial work: let someone else do it
- In this case STP
- [Simple Theorem Prover]

What is STP?

- Constraint solver for problems involving bit-vectors and arrays
- Open-source, written by Vijay Ganesh
- Used for model-checking, theorem proving, EXE, etc.
- Gives Boolean answer whether formula is satisfiable & assignment if it is

STP formulae

Just a bunch of assertions in QF_ABV

Simple example:

- x0 : BITVECTOR(4);
- ...
- x9 : BITVECTOR(4);
- ASSERT (BVPLUS(4,BVMULT(4,x0, 0hex6), 0hex0, 0hex0,
- BVMULT(4,x3, 0hex7), BVMULT(4,x4, 0hex4),
- BVMULT(4,x5, 0hex6), BVMULT(4,x6, 0hex4),
- 0hex0, 0hex0, BVMULT(4,x9, 0hex8),0hex0) = 0hex7);

High-level algorithm

For multi-assignments:

1. Find all gadgets assigning to targets
2. Verify constraints for each (protect/memread/memcorrupt)
3. Find all gadgets for expressions on RHS
4. Chain expression gadgets
5. Connect LHS and RHS

Notes on chaining algorithm

- Chaining for arithmetic/logical expressions may use registers/memory locations for temporary results
- Multi-assignments give us freedom
- Algorithm sometimes may fail because constraints cannot be satisfied [insufficient gadgets]

K got the payload, now?

You could test it on a jailbroken phone

- Does not match reality!
- No code signing for instance
- Still an option if exploit reliability is not your primary concern

K got the payload, now?

You could test it on a developer phone

- Have a small application to reproduce a “ROP scenario”
- Depending on the application you’re targeting the sandbox policy is different
- Still closer to reality

Simple plan

- Allocate a buffer on the heap
- Fill the buffer with the shellcode
- Point the stack pointer to the beginning of the attacker controlled memory
- Execute the payload
- Restore

Future work

- Port to other platforms (eg: x86)
- Abstract language to describe gadgets
- Try to avoid “un-decidable” constraints
- Make it more flexible to help when ASLR is in place

Thanks for your time

Questions?