OptiROP: Hunting for ROP gadgets in style

NGUYEN Anh Quynh, COSEINC
<aquynh -at- gmail.com>

Blackhat USA 2013, August 1st

Agenda

1. Return-Oriented-Programming (ROP) gadgets & shellcode
   - Problems of current ROP tools
2. OptiROP: desires, ideas, design and implementation
   - Semantic query for ROP gadgets
   - Semantic gadgets
3. Live demo
4. Conclusions

Attack & defense

Software attack

- Abuse programming/design flaws to exploit the system/app
- Trigger the vulnerability with malicious input to execute attacker's code

Exploitation mitigation

- Accepting that software can be buggy, but make it very hard to exploit its bugs
- Multiple mitigation mechanisms have been proposed and implemented in modern system
- Data Executable Prevention (DEP) is widely deployed
  - Make sure input data from attacker is unexecutable, thus input cannot embed malicious payload inside
  - Introduced into hardware level, and present everywhere nowadays

DEP bypass

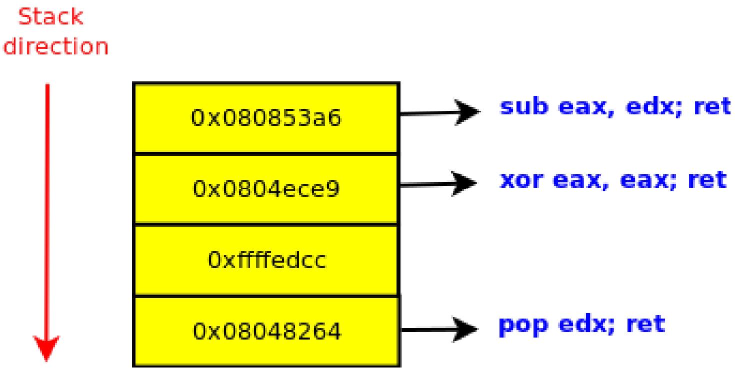
- Return-Oriented-Programming (ROP) was propsed to defeat DEP
- Make sure attack code is not needed to be inside input data anymore, thus efficiently overcome DEP-based defense
- Become the main technique to write shellcode nowadays

ROP concept

- Non-code-injection based attack: reuse available code in exploited memory space of vulnerable app
- (Ab)Use code snippets (called ROP gadgets) and chain them together to execute desired action
- ROP gadget mostly come from unintended instructions
- Proved to be Turing-completed

ROP example

- Sequence of ROP gadgets set EAX = 0x1234



ROP shellcode

- Chain gadgets together to achieve traditional code injection shellcode
- Usually implemented in multiple stages
  - Stage-0 shellcode (ROP form) remaps stage-1 payload memory to be executable
  - Transfer control to stage-1 payload (old-style shellcode)

ROP shellcode to dominate in future?

- More restriction on what ROP gadgets can do to launch ROP-free shellcode stage
  - Windows 8 ROP mitigation enforces policies on who/where can call VirtualAlloc() /VirtualProtect() to enable memory executable at run-time
  - Pushed exploitation to do more work in stage-0
- Future system might totally forbid code injection
  - No more stage-1 old-style shellcode, but full-ROP shellcode?
  - IOS already implemented this mitigation
    - ⋆ Writable pages have NX permission & only signed pages are executable
    - ⋆ ROP is the only choice for shellcode

ROP tools

ROP programming is hard

- How to find right gadgets to chain them to do what we want?
- Full ROP shellcode can be a nightmare
- ROP tools available to help exploit writer on the process of finding and chaining gadgets

ROP tools is not much helpful ☹

- Given a binary containing gadgets at run-time, collect all the gadgets available
- Let users find the right gadgets from the collection
- Mostly stop here, and cannot automatically find the right gadgets nor chain them for desired action
- Manually tedious boring works at this stage for exploitation writers

Gadget catalogs

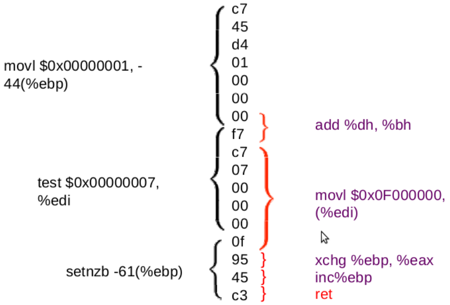
Gadget type	Semantic	Example

LOAD	Load value to register	mov eax, ebp
		mov eax, 0xd7
		mov eax, [edx]

STORE	Store to memory	mov [ebx], edi mov [ebx], 0x3f
ADJUST	Adjust reg/mem	add ecx, 9 add ecx, esi
CALL	Call a function	call [esi] call [0x8400726]
SYSCALL	Systemcall for *nix	int 0x80 sysenter

Internals of traditional ROP tools

- Gathering gadgets
  - Locate all the return instructions (RET) [1]
  - Walk back few bytes and look for a legitimate sequence of instructions. Save the confirmed gadgets
- Given user request, searching for suitable gadgets
  - Go through the list of collected gadgets and match each with user's creterias (mostly using regular expression searching on gadget text)



[1] JUMP-oriented ROP is similarly trivial, but not discussed here

Gadget hunting example

ROP/ASM instructions: mov r32 [r32 % -leave

Search

Found 240 gadgets/chains

```
>0x7c80ac6a : mov eax [eax+0x18] ; ret ;;
--------------------------------------------------------------------------
>0x7c8099c6 : mov eax [eax+0x20] ; ret ;;
--------------------------------------------------------------------------
>0x7c8097d6 : mov eax [eax+0x24] ; ret ;;
--------------------------------------------------------------------------
>0x7c830777 : mov eax [eax+0x34] ; ret ;;
--------------------------------------------------------------------------
>0x7c812fb4 : mov eax [eax+0x48] ; ret ;;
--------------------------------------------------------------------------
>0x7c80a4bb : mov eax [eax+0xc4] ; ret ;;
--------------------------------------------------------------------------
>0x7c83584e : mov eax [eax+0x186c] ; ret ;;
--------------------------------------------------------------------------
>0x7c812fb1 : mov eax [eax+0x10] ; mov eax [eax+0x48] ; ret ;;
--------------------------------------------------------------------------
>0x7c80ac67 : mov eax [eax+0x30] ; mov eax [eax+0x18] ; ret ;;
--------------------------------------------------------------------------
```

ROPME in action to find some LOAD gadgets

Problems of hunting for ROP gadgets (1)

Syntactic searching: advantages

- Easy to implement and became universal solution
- Proven, and implemented by all ROP gadget searching tools nowadays

Syntactic searching: Problems

- Non-completed: do not return all suitable gadgets
- Too many irrelevant gadgets returned
- Time consuming: Require trial-N-error searching repeatedly
- Waste gadgets: Sometimes gadgets are scarce, so must be used properly

Problems of hunting for ROP gadgets (2)

- Problem: gadgets copy ebx to eax (eax = ebx)?
- (De-facto) Answer: syntactic (regular expression based) searching on collected gadgets
  - mov eax, ? → mov eax, ebx; ret
  - xchg eax, ? → xchg eax, ebx; ret
  - lea eax, ? → lea eax, [ebx]; ret
  - ...
- Query 1: any other promising queries?
  - xchg ebx, eax; ret
  - imul eax, ebx, 1; ret
  - anything else missing??
- Query 2: how many queries and efforts needed to find this simple gadget???

Problems of hunting for ROP gadgets (2)

- Question 3: Still looking for gadgets copy ebx to eax (eax = ebx). Which syntactic query can find below gadget?
  - xor eax, eax; pop esi; add eax, ebx; ret
  - xor eax, eax, not eax; and eax, ebx; ret
  - xchg ebx, ecx; xchg ecx, eax; ret
  - push ebx; xor eax, eax; pop eax; ret
- Query 4: Gadget to pivot (migrate) stack?
  - ROPME suggests to try *all* following queries
    - xchg esp %
    - xchg r32 esp %
    - ? esp %
  - Any missing queries?
    - leave

Other problems

- No semantics reported for suitable gadgets
  - Which registers are modified?
  - Which EFlags are modified?
  - How many bytes the stack pointer is advanced?
- No tool can chain available gadgets for requested semantic gadget
  - Ex: xor eax, eax; ret + xchg edx, eax; ret $\Longleftrightarrow$ edx = 0
  - Ex: mov esi, 0xfffffff8; ret + lea eax, [esi + edx + 8]; ret $\Longleftrightarrow$ eax = edx

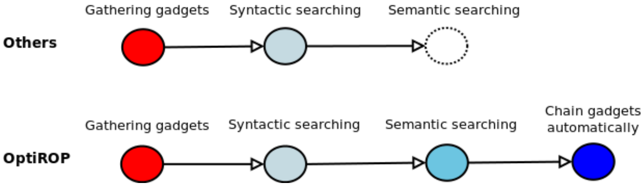
OptiROP to save exploitation writers!

OptiROP desires

- Semantic searching for ROP gadgets
  - Semantic query rather than syntactic
    - Rely on meaning of gadgets rather than how it looks like (syntactic)
  - Ex: eax = [ebx] → mov eax, [ebx] & xchg [ebx], eax
  - Ex: esi = edi → lea esi, [edi] & imul esi, edi, 1
- User-provided criteria allowed: modified registers, stack pointer advanced, ...
- Providing detail semantics of found gadgets
  - mov eax, edx; add esp, 0x7c; ret
  - → Modified registers: eax, AF, CF, OF, SF, ZF
  - → esp += 0x80
- Chain available gadgets if natural gadget is unavailable
  - Pick suitable gadgets to chain them for desired gadget
  - Ex: xor eax, eax; ret + xchg edx, eax; ret ⟺ edx = 0
- (x86 + x86-64) * (Windows PE + MacOSX Mach-O + Linux ELF + Raw binary)

OptiROP versus others

Features	RopMe	RopGadget	ImmDbg	OptiROP
Syntactic query	✓	X	X	✓
Semantic query	X	X	✓[2]	✓
Chain gadgets	X	X[3]	X	✓
PE|ELF|M-O (x86)	✓|✓|✓	✓|✓|N	✓|N|N	✓|✓|✓
PE|ELF|M-O (x86-64)	✓|✓|✓	✓|✓|X	X|X|X	✓|✓|✓



[2] Basic function

[3] Have simple syntactic-based gadget chaining for predefined shellcode

Gadget catalogs

Gadget type	Semantic query	Sample output
LOAD	eax = ebp	mov eax, ebp; ret xchg eax, ebp; ret lea eax, [ebp]; ret
STORE	[ebx] = edi	mov [ebx], edi; ret xchg [ebx], edi; ret
ADJUST	ecx += 9	add ecx, 9; ret sub ecx, 0xfffffff7; ret
CALL	call esi	xchg eax, esi + call eax
SYSCALL	int 0x80	int 0x80

OptiROP ideas

- Generate semantic logical formula on collected gadget code
- Allow semantic query: describing high level desired action of needed gadget
- Perform matching/searching semantic query based on logical formula of collected gadgets using SMT solver
- Combine logical formulas (of different gadgets) to produce desired semantic actions

Challenges

- Machine instructions overlap (in semantics)

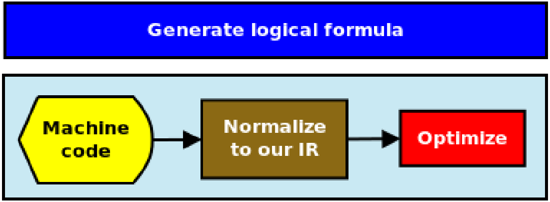
mov eax, ebx $\Longleftrightarrow$ lea eax, [ebx]

- Instructions might have multiple implicit side effects

push eax $\Longleftrightarrow$ ([esp] = eax; esp -= 4)

Solutions

- Normalize machine code to Intermediate Representation (IR)
  - IR must be simple, no overlap
  - IR express its semantic explicitly, without side effect
  - IR supports Static Single-Assignment (SSA) for the step of generating logical formula
- Translate machine code to our selected IR
- Optimize resulted IR
- Generate logical formula from output IR



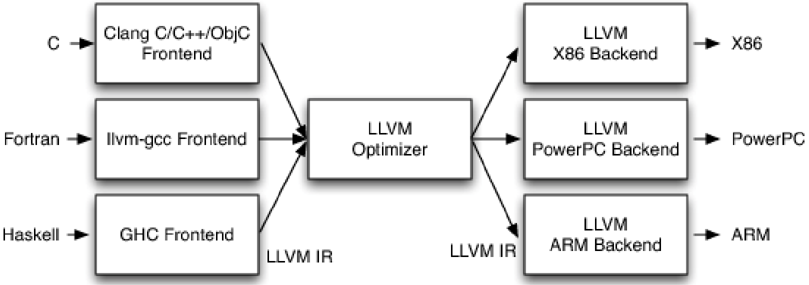
Introduction on LLVM

LLVM project

- Open source project on compiler: http://www.llvm.org
- A set of frameworks to build compiler
- LLVM Intermediate Representation (IR) with lots of optimization module ready to use

LLVM model

Compiler model



LLVM model: separate Frontend - Optimization - Backend

LLVM IR

- Independent of target architecture
- RISC-like, three addresses code
- Register-based machine, with infinite number of virtual registers
- Registers having type like high-level programming language
  - void, float, integer with arbitrary number of bits (i1, i32, i64)
- Pointers having type (to use with Load/Store)
- Support Single-Static-Assignmenti (SSA) by nature
- Basic blocks having single entry and single exit
- Compile from source to LLVM IR: LLVM bitcode

LLVM instructions

- 31 opcode designed to be simple, non-overlap
  - Arithmetic operations on integer and float
    - *add*, *sub*, *mul*, *div*, *rem*, ...
  - Bit-wise operations
    - *and*, *or*, *xor*, *shl*, *lshr*, *ashr*
  - Branch instructions
    - Low-level control flow is unstructured, similar to assembly
    - Branch target must be explicit :-(
    - *ret*, *br*, *switch*, ...
  - Memory access instructions: *load*, *store*
  - Others
    - *icmp*, *phi*, *select*, *call*, ...

Example of LLVM IR

```c
unsigned add2(unsigned a, unsigned b) {
    if (a == 0) return b;
    return add2(a-1, b+1);
}
```

```llvm
define i32 @add2<i32 %a, i32 %b) {
entry:
    %tmp1 = icmp eq i32 %a, 0
    br i1 %tmp1, label %done, label %recurse

recurse:
    %tmp2 = sub i32 %a, 1
    %tmp3 = add i32 %b, 1
    %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
    ret i32 %tmp4

done:
    ret i32 %b
}
```

C code - LLVM IR code

Optimize LLVM bitcode

- The core components of the LLVM architecture
- Optimize performed on the bitcode (LLVM Pass) with combined/selected LLVM passes
  - Optimization to collect/visualize information
  - Optimization to transform the bitcode
  - Others
- 182 passes ready to use in LLVM 3.2

Why LLVM IR?

- Good IR for normalization phase
- Only use subset of LLVM instructions
  - Ignore instructions about high-level information from source code
- Handy frameworks to process the output IR
- Possible to optimize the LLVM bitcode resulted from the step of translating machine code $\rightarrow$ LLVM IR
  - Use suitable LLVM passes to optimize the normalized IR (thus having more compact IR at the output)

Translate machine code to LLVM IR

- Similar to building compiler frontend for "machine code language"
- Tough due to the unstructured characteristics of machine code
  - Target of indirect branches
  - Self-modified code
- From machine code, build the Control Flow Graph (CFG) consisting of basic blocks (BB)
- Translate all instructions in each BB to LLVM IR
  - Reference the ISA manual of corresponding platforms (e.x: Intel/AMD manual)

Translate x86 code to LLVM IR

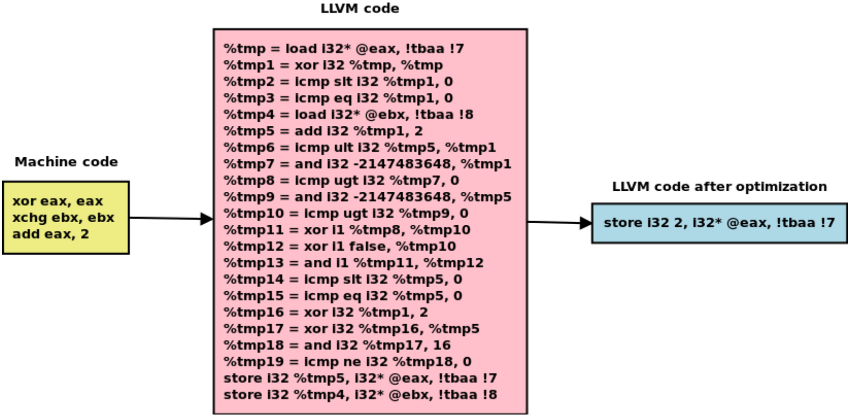
```
and eax, ebx

~
~
~
~
~
~
~
~
~
~
```

```
%tmp = load i32* @_eax, align 4, !tbaa !1
%tmp1 = load i32* @_ebx, align 4, !tbaa !2
%tmp2 = and i32 %tmp1, %tmp
%tmp3 = icmp slt i32 %tmp2, 0
%tmp4 = icmp eq i32 %tmp2, 0
store i1 false, i1* @_AF, align 1, !tbaa !15
store i1 false, i1* @_OF, align 1, !tbaa !13
store i1 %tmp4, i1* @_ZF, align 1, !tbaa !9
store i1 false, i1* @_CF, align 1, !tbaa !10
store i32 %tmp2, i32* @_eax, align 4, !tbaa !1
store i1 %tmp3, i1* @_SF, align 1, !tbaa !11
```

Example of translating x86 code to LLVM IR

Optimize LLVM bitcode

- Constant propagation (-constprop)
  - (x = 14; y = x + 8) ⇒ (x = 14; y = 22)
- Eliminate dead store instructions (-dse)
  - (y = 3; ...; y = x + 1) ⇒ (...; y = x + 1)
- Combine instructions (-instcombine)
  - (y = x + 1; z = y + 2) ⇒ (z = x + 3)
- Simplify CFG (-simplifycfg)
  - Remove isolated BB
  - Merges a BB into its predecessor if there is only one and the predecessor only has one successor
  - Merge a BB that only contains an unconditional branch



Machine code
xor eax, eax
xchg ebx, ebx
add eax, 2
LLVM code
%tmp = load i32* @eax, !tbaa !7
%tmp1 = xor i32 %tmp, %tmp
%tmp2 = icmp slt i32 %tmp1, 0
%tmp3 = icmp eq i32 %tmp1, 0
%tmp4 = load i32* @ebx, !tbaa !8
%tmp5 = add i32 %tmp1, 2
%tmp6 = icmp ult i32 %tmp5, %tmp1
%tmp7 = and i32 -2147483648, %tmp1
%tmp8 = icmp ugt i32 %tmp7, 0
%tmp9 = and i32 -2147483648, %tmp5
%tmp10 = icmp ugt i32 %tmp9, 0
%tmp11 = xor i1 %tmp8, %tmp10
%tmp12 = xor i1 false, %tmp10
%tmp13 = and i1 %tmp11, %tmp12
%tmp14 = icmp slt i32 %tmp5, 0
%tmp15 = icmp eq i32 %tmp5, 0
%tmp16 = xor i32 %tmp1, 2
%tmp17 = xor i32 %tmp16, %tmp5
%tmp18 = and i32 %tmp17, 16
%tmp19 = icmp ne i32 %tmp18, 0
store i32 %tmp5, i32* @eax, !tbaa !7
store i32 %tmp4, i32* @ebx, !tbaa !8
LLVM code after optimization
store i32 2, i32* @eax, !tbaa !7

Satisfiability Modulo Theories (SMT) solver

- Theorem prover based on decision procedure
- Work with logical formulas of different theories
- Prove the satisfiability/validity of a logical formula
- Suitable to express the behaviour of computer programs
- Can generate the model if satisfiable

Z3 SMT solver

- Tools & frameworks to build applications using Z3
  - Open source project: http://z3.codeplex.com
  - Support Linux & Windows
  - C++, Python binding
- Support BitVector theory
  - Model arithmetic & logic operations
- Support Array theory
  - Model memory access
- Support quantifier *Exist* ($\exists$) & *ForAll* ($\forall$)

Create logical formula

Encode arithmetic and moving data instructions

Malware code

```
mov esi, 0x48
mov edx, 0x2007
```

Logical formula

(esi == 0x48) and (edx == 0x2007)

Encode control flow

Malware code

```
cmp eax, 0x32
je $_label
xor esi, esi
...
_label:
mov ecx, edx
```

Logical formula

(eax == 0x32 and ecx == edx) or
(eax != 0x32 and esi == 0)

Create logical formula (2)

NOTE: watch out for potential conflict in logical formula

Malware code

```
mov esi, 0x48
...
mov esi, 0x2007
```

Logical formula

(esi == 0x48) and (esi == 0x2007)

Malware code

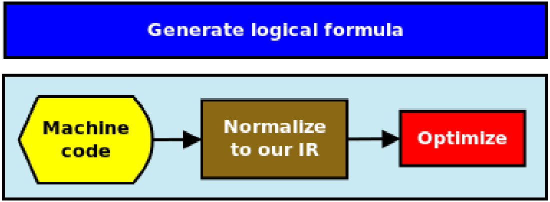
```
mov esi, 0x48
...
mov esi, 0x2007
```

Logical formula with Single-Static-Assignment (SSA)

(esi == 0x48) and (esi1 == 0x2007)

Steps to create logical formula

- Normalize machine code to LLVM IR
  - LLVM IR is simple, no overlap, no side effect (semantic explicitly)
  - LLVM IR supports Static Single-Assignment (SSA) for the step of generating logical formula
- Translate machine code to LLVM IR
- Optimize resulted LLVM bitcode
- Generate logical formula from LLVM bitcode

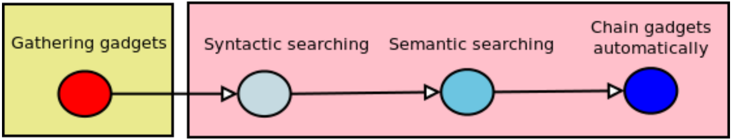


Generate logical formula from LLVM IR

- Wrote a LLVM pass to translate bitcode to SMT logical formula
- Go through the CFG, performing block-by-block on the LLVM bitcode
- Generate formula on instruction-by-instruction, translating each instruction to SMT formula
  - Use theory of BitVector or Array, depending on instruction
    - BitVector to model all arithmetic and logic operations
    - Array to model memory accesses

OptiROP model

- Preparation stage
  - Automatically done by OptiROP on given executable binary
  - Looking for gadgets, then generate and save gadget formulas
  - Also save modified registers & stack pointer (ESP/RSP) advanced
- Searching (hunting) stage
  - Involved users: semantic query and selection criteria
  - Looking for gadgets from the set of collected gadget code and formulas from preparation stage

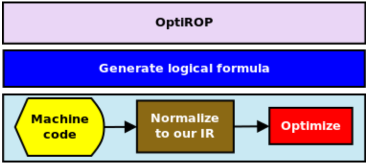


Preparation stage (1)

- Looking for gadgets is done in traditional way
  - Locate all the return instructions (RET)
  - Walk back few bytes (number of bytes is configurable) and verify if the raw code (until RET) is a legitimate sequence of instructions
  - Save all the found gadgets

Preparation stage (2)

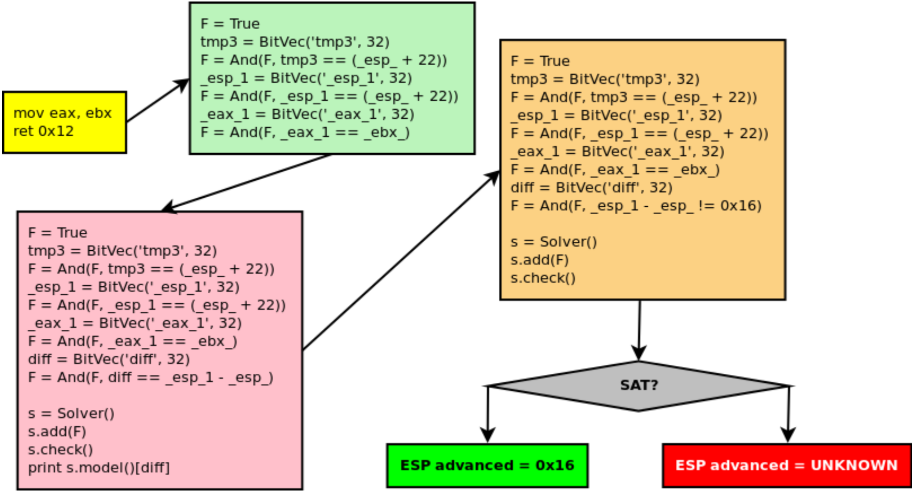
- On each gadget code (asm form) found in stage 1, generate one SMT formula
  - Normalize gadget code (machine code → LLVM bitcode)
  - Optimize gadget code (LLVM bitcode → Optimized LLVM bitcode)
  - Generate SMT formula on normalized+optimized code (Optimized LLVM bitcode → SMT formula)
- With each gadget, also save modified registers & stack pointer (ESP/RSP) advanced
  - Modified registers are recognized by modified registers content at the end of SMT formula
  - Stack pointer advanced is calculated on gaget formula thanks to SMT solver



Stack pointer advanced

- Ask SMT solver for the difference between final value and initial value of stack pointer
  - ESP_1 - ESP
  - Step 1: Get a model (always satisfied)
  - Step 2: Ask for *another* model with different difference value
    - Satisfiable: ESP_1 - ESP == FIXED → esp advanced fixed number of bytes
    - Unsatisfiable: esp advanced unknown number of bytes (context dependent)

Stack pointer advanced - example



Primitive gadgets (1)

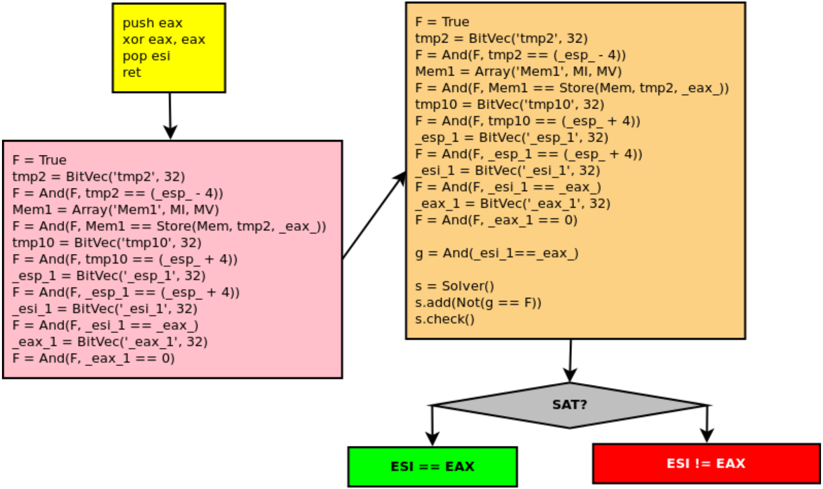
- Gadgets mostly evolve around registers, and require registers
  - P1: Gadget set register to another register
    - Ex: xor eax, eax; or eax, ebx; ret → eax = edx
  - P2: Gadget set register to immediate constant (fixed concrete value)
    - Ex: mov edi, 0x0; lea eax, [edi]; pop edi; ret → eax = 0
- Hunting for primitive gadgets (P1 & P2) from the set of collected gadget code & formulas

Primitive gadgets (2)

- "Natural" primitive gadgets
  - PN1: Gadget set register to another register
    - Ex: xor eax, eax; add eax, ebx; ret → eax = edx
  - PN2: Gadget set register to immediate constant (fixed concrete value)
    - Ex: or ebx, 0xffffffff; xchg eax, ebx; ret → eax = 0xffffffff
  - "Free" register: POP gadget that set register to value poping out of stack bottom (thus can freely get any constant)
    - Ex: # push 0x1234 + pop eax; inc ebx; ret → eax = 0x1234
- "Chained" primitive gadgets
  - PC1: Gadget set register to another register
    - Ex: (lea ecx, [edx]; ret) + (mov eax, edx; ret) → eax = edx
  - PC2: Gadget set register to immediate constant (fixed concrete value)
    - Ex: (or ebx, 0xffffffff; ret) + (xchg eax, ebx; ret) → eax = 0xffffffff
  - PC3: Equation-derived gadget: Gadget derived from computed equation, and require constraint to achieve target gadget
    - Ex: (imul ecx, [esi], 0x0; ret) + (add ecx, eax; ret) → ecx = eax

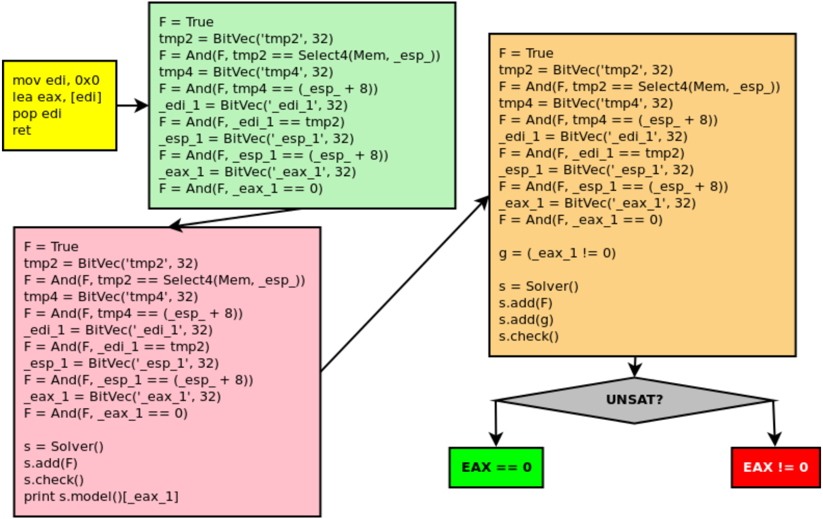
Hunting for natural gadget: PN1

- Use SMT solver to prove the equivalence of 2 formulas



Hunting for natural gadget: PN2

- Similar to finding stack pointer advanced value, use SMT solver to find if a register has constant value



Chained gadget

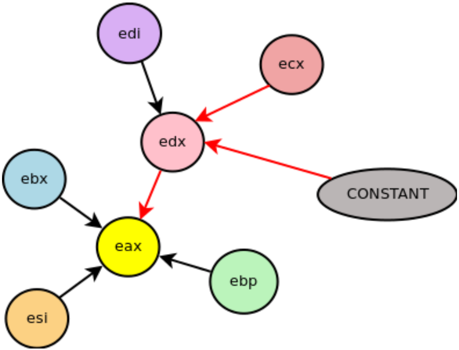
- Try to chain natural gadgets to achieve higher level gadget
  - PC1+PC2: Combine simple PN1 & PN2 gadgets together
    - Ex: ((mov ebx, edx; ret)) + (xchg ebx, ecx; ret) + (lea eax, [ecx]; ret) → eax = edx
    - Ex: (imul ecx, [esi], 0x0; ret) + (xchg ecx, eax; ret) → eax = 0
    - Ex: (# push 0x1234) + (pop ebp; ret) + (xchg ebp, eax; ret) → eax = 0x1234
  - PC3: Equation-derived gadget: Gadget derived from computed equation, and require constraint to achieve target gadget
    - Ex: (imul ecx, [esi], 0x0; ret) + (add ecx, eax; ret) → ecx = eax
    - Ex: (# push 0xffffedcc)+(pop edx; ret)+(xor eax, eax; ret)+(sub eax, edx; ret) → eax = 0x1234

Hunting for chained gadget: PC1+PC2 (1)

- Idea: Chain (r2 = r1) + (r3 = r2) → r3 = r1
- Gadgets can be either of PN1 or PN2 type
  - Ex: ((mov ebx, edx; ret)) + (xchg ebx, ecx; ret) + (lea eax, [ecx]; ret) → eax = edx
  - Ex: (imul ecx, [esi], 0x0; ret) + (xchg ecx, eax; ret) → eax = 0
- Combined with "free register" gadget to assign arbitrary arbitrary constant to register
  - Ex: (# push 0x1234) + (pop ebp; ret) + (xchg ebp, eax; ret) → eax = 0x1234

Hunting for chained gadget: PC1+PC2 (2)

- Algorithm: Build a tree of PN1 & PN2 gadget, then bridge the nodes together (graph theory)
- Ex: with eax = {ebx, edx, esi, ebp}; edx = {edi, ecx, CONSTANT}:
  - Bridge ecx → eax: (edx = ecx) + (eax = edx) → (eax = ecx)
  - Bridge CONSTANT → eax: (edx = CONSTANT) + (eax = edx) → (eax = CONSTANT)

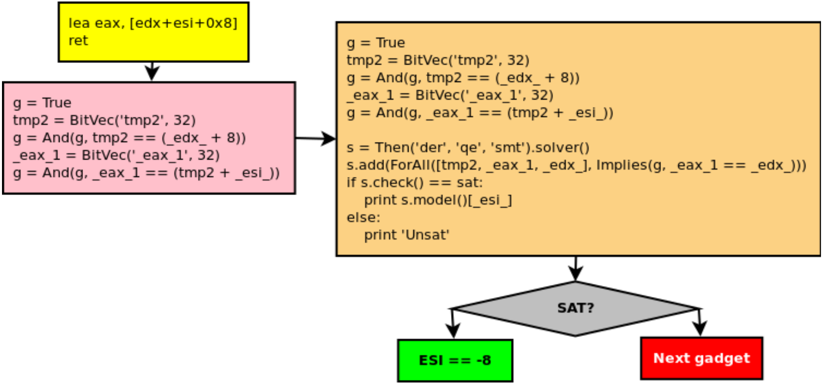


Hunting for chained gadget: PC3 (1)

- PC3: Equation-derived gadget: Gadget derived from computed equation
- Require constraint to achieve target gadget
- Ex: (# xor eax, eax; inc eax; ret) + (imul eax, edx; ret) + (add eax, ebx) → eax = 0x1234
- Ex: (mov eax, 0x13; ret) + (# push 0x1221) + (pop edx; ret) + (add eax, ebx) → eax = 0x1234

Hunting for chained gadget: PC3 (2)

- Generate SMT formula based on known constraints (fixed & free registers), then ask SMT solver for a model of free registers
  - Ex: (# push 0xfffffff8) + (pop esi; inc ebp; ret) + (lea eax, [edx+esi+0x8]; ret) → eax = edx
  - Ex: (xor eax, eax; ret) + (not eax; ret) + (and eax, edx; ret) + (add eax, ebx) → eax = edx



LOAD gadget

- r1 = r2
  - PN1 or PC2 or PC3
  - Combined gadget "PUSH r1" with "Free" register gadget of r2
  - Combine all methods
    - ★ (r3 = r2) + (r4 = 0) + (r1 = r4 + r3) → r1 = r2
- r1 = CONSTANT
  - PN2 or PC2 or PC3
  - Combine all methods
    - ★ (r3 = 0x10) + (r2 = 0x38) + (r1 = r2 - r3) → r1 = 0x28
- r1 = [r2]
  - Chain gadget set memory address to a register + gadget reading memory
    - ★ (r3 = r2) + (r4 = [r3]) + (r1 = r4) → r1 = r2

STORE gadget

- Query [r1] = r2
- Query [CONSTANT] = r2
  - Similar to LOAD gadget: use/chain primitive gadgets

ADJUST gadget

- r1 += CONSTANT
  - (r += 8) + (r +=8) + (r += 1) → r+= 17
    - (add eax, 8; ret) + (add eax, 8; ret) + (inc eax; ret) → eax += 17
- Find all the "fixed" register gadget of this register
- Ask SMT solver for a model of linear equation so the total is CONSTANT
- Try to get a model with minimal values of linear variables
  - Formula: a1 * 8 + a2 *1 == 17 & (a1 + a2) = MIN → a1 = 2, a2 = 1

CALL gadget

- call r
  - Chain (r = r1) + (CALL r1)
- call [r]
  - Chain (r1 = r) + (CALL [r1])
- call [CONSTANT]
  - Chain (r1 = CONSTANT) + (CALL [r1])

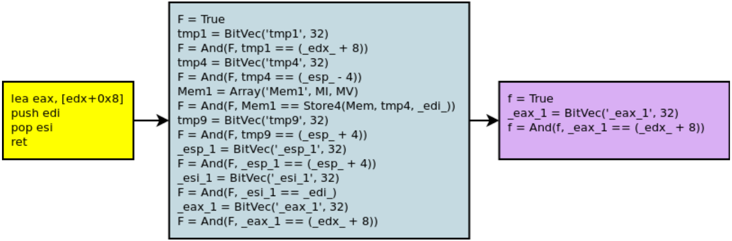
Live demo

Performance optimization used in OptiROP

- Fast-path optimization
  - Fast paths are executed first, slow paths come later (x3)
  - SMT solver is executed as the last choice, when nothing else can reason about the formula
- Caching processed formulas to avoid recalculation (x2)
- Parallel searching (x8)
  - Multiple threads, each thread verifies one candidated formula independently
- Pre-calculated as-much-as-possible (x10)
  - Modified registers, stack pointer advanced, fixed registers, free registers, ...
- Code slicing applied on selected queries (x2)

Code slicing

- Only consider the set of instructions that may affect the values at some point of interest
- Slicing performed on related registers significantly reduce the size of formula to be verified
- Slicing is done on the gadget's formula, rather than earlier phases



Gadget code - SMT formula - Slicing on EAX

OptiROP implementation

- Web + commandline interface
- Framework to translate x86 code to LLVM IR
- Framework to generate SMT formula from LLVM bitcode
- Framework for code slicing on SMT formula
- Support neutral disassembly engine to disassemble machine code (normalization phase)
- (x86 + x86-64) * (Windows PE + MacOSX Mach-O + Linux ELF + Raw binary)
- Use Z3 solver to process logical formulas (opaque predicate)
- Implemented in Python & C++

Future works

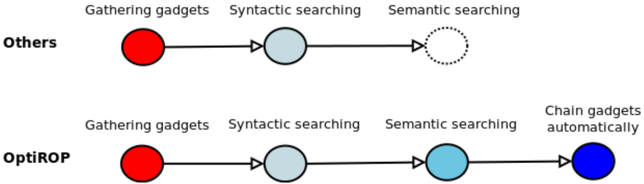
- More methods to chain gadgets
- More higher-level gadgets
- Full-compiler-based implementation
- Support other hardware platforms: ARM (anything else?)
- Deployed OptiROP as an independent toolset for exploitation developers
- To be launched to public later (after more bugfixes/code polishing)
  - Watch out for http://optirop.coseinc.com

Conclusions

- OptiROP is an innovative approach to find ROP gadgets
  - Natural and easy semantic questions supported
  - User-provided criterias can filter out unwanted gadgets
  - Chain selected gadgets if natural gadget is unavailable
  - (x86 + x86-64) * (Windows PE + MacOSX Mach-O + Linux ELF)
  - Commandline & web-based tool available
  - Internally used compiler techniques & SMT solver
- Will be freely available to public soon ☺

OptiROP versus others

Features	RopMe	RopGadget	ImmDbg	OptiROP
Syntactic query	✓	X	X	✓
Semantic query	X	X	✓	✓
Chain gadgets	X	X	X	✓
PE|ELF|M-O (x86)	✓|✓|✓	✓|✓|X	✓|X|X	✓|✓|✓
PE|ELF|M-O (x86-64)	✓|✓|✓	✓|✓|X	X|X|X	✓|✓|✓



References

- LLVM project: http://llvm.org
- LLVM passes: http://www.llvm.org/docs/Passes.html
- Z3 project: http://z3.codeplex.com
- ROPME: http://ropshell.com
- ROPgadget: http://shell-storm.org/project/ROPgadget/
- ImmunityDbg: http://www.immunityinc.com
- Nguyen Anh Quynh, OptiSig: semantic signature for metamorphic malware, Blackhat Europe 2013
- Nguyen Anh Quynh, OptiCode: machine code deobfuscation for malware analyst, Syscan Singapore 2013

Questions and answers

OptiROP: Hunting for ROP gadgets in style

Nguyen Anh Quynh <aquynh -at- gmail.com>