

程序分析及约束求解器

Edgar Barbosa
SyScan360
北京 - 2014

作者简介

- 高级安全研究员- COSEINC(国际知名安全机构)
- 从事Windows内核、驱动设备和管理程序的逆向工程
- BluePill 硬件虚拟化Rootkit作者之一
- 现在专注于漏洞挖掘的自动化研究

提纲

- 程序分析
- 漏洞挖掘
- SAT 求解器
- SMT 求解器
- 中间语言

目的

- 本次演讲的目的是展示如何使用约束求解器，包括 SMT 约束求解器，用于逆向工程和漏洞挖掘之类的程序分析。

程序分析

漏洞挖掘

漏洞挖掘

- 近来，程序分析和逆向工程大多是专用于一个特定的目标：挖掘软件漏洞。
- 无论你是否喜欢，这就是事实。逆向工程的主要功能不仅仅是理解和修改应用程序，还作为漏洞挖掘的辅助工具。

如何挖掘漏洞？

- 如何挖掘闭源应用程序的漏洞？
- 黑盒测试是通过操纵其公开接口对软件系统进行评估的方法。
- 最知名的黑盒测试工具是模糊测试。

Fuzzing

- 模糊测试或者模糊化是一种软件测试技术，通常是自动化或者半自动化，它包括提供无效的、未预期的、随机的数据作为程序的输入。

Fuzzing Phases

1. 识别目标
2. 识别输入
3. 生成模糊化数据
4. 将模糊化数据作为输入执行程序
5. 监视异常
6. 验证可用性

Fuzzing – 输入类型

- 如果我们要对一台FTP服务器进行模糊测试，我们不能仅仅生成随机数据，并将其发送给服务器。这是非常低效的（除了极少数情况）。
- 建立一个能够理解FTP协议的模糊测试机是有必要的。这同样适合于其他任何协议或文件格式，如pdf或doc。

Fuzzing – 格式

- 问题是开发者必须事先知道关于输入格式的知识。
- 如果协议或格式是未知的怎么办？
- 如果使用的校验算法是未知的怎么办？
- 另外的情况下，尽管协议是公开的，但一些实现方式并未遵守协议规范。

逆向工程

- 结合逆向工程，我们可以提取协议和格式的信息。
- 一些高层次的信息在编译过程中会丢失，但用于理解应用程序工作的所有必要信息是编码在可执行文件中的。
- 这包括协议和文件解析器。

Fuzzing

- 模糊测试仍然是漏洞挖掘的有效手段。
- 它能产生很多崩溃，这些崩溃作为分析问题的起点，以便确定漏洞的可利用性。
- 是否存在自动化的模糊测试方法，而不需要程序员学习新的协议或文件格式规范？
- 我们很懒惰，学习新的格式和协议很费时。

漏洞挖掘自动化

- 我们想要一个具备以下功能的系统：
 - 理解输入数据是如何影响软件的执行
 - 审计程序函数而不需要包括协议和文件格式的任何先验知识
 - 报告错误并立即分析出错误根源
 - 不产生误报
 - 自动增加和程序路径的覆盖

约束求解器

SyScan360

约束求解器

- 约束求解器的帮助
- 帮助我们学习文件格式和协议，并自动增加代码路径覆盖率。
- 我们的想法是将程序分析的问题转换为约束求解器的问题来解决。
- 什么是约束求解器？

约束编程

*“Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the **user states the problem**, **the computer solves it**.” [E. Freuder]*

约束求解器

- 用户使用某种特定的语言描述对象（变量）的约束条件，求解器将试图求解出能够满足所有约束条件的每个变量的值。

布尔可满足性

- 最有名的可满足性问题是布尔可满足性问题 (SAT)
- NP-完全问题!
- 尽管有这么高的复杂度，它还是被用来解决模型检查、形式化验证和其它包括成千上万变量和约束条件的复杂问题!

命题公式

- SAT 问题被编码为公式。
- 在命题逻辑中，命题公式是一种结构良好，具有真值的语法公式。

SAT 求解

- 找出符合条件的命题逻辑公式
- 是否有可能满足这个问题?

$$(x_1 \vee x_3 \vee \bar{x}_4) \wedge (x_4) \wedge (x_2 \vee \bar{x}_3)$$

- 如果你想用SAT求解器解决你的问题，你需要将你的问题转化为CNF形式的布尔公式。

CNF

- 合取范式
- 这是共同的要求：布尔表达式按照范式或CNF的形式进行编写。一个合取范式公式包括：
 - clauses使用AND进行连接;
 - each clause, in turn, consists of literals使用OR进行连接;
 - 每个 literal 是一个变量名(a positive literal) , 或者是被NOT修饰的变量名 (a negative literal).

DIMACS 输入格式

- 文件可以用注释开始, 即行首使用字符‘c’。
- 紧随注释之后, 是一行 “p cnf nbvar nbclauses” 表示该实例是CNF格式; nbvar 出现在该文中的变量的数目; nbclauses 是包含在该文中的子句的确切数目。

DIMACS 输入格式

- 然后接下来是子句。每个子句是介于-nbvar和nbvar之间，不同非空的数序列，以0作为结尾，并且在同一行。正数表示变量，负数表示变量的否定。

排中律

c law-of-excluded-middle

c

p cnf 1 1

1 -1 0

SAT - DEMO

$$(x_1 \vee x_3 \vee \bar{x}_4) \wedge (x_4) \wedge (x_2 \vee \bar{x}_3)$$

SAT Encoding

(automatically generated from problem specification)

```
p cnf 51639 368352
-1 7 0
-1 6 0
-1 5 0
-1 -4 0
-1 3 0
-1 2 0
-1 -8 0
-9 15 0
-9 14 0
-9 13 0
-9 -12 0
-9 11 0
-9 10 0
-9 -16 0
-17 23 0
-17 22 0
```


i.e., $((\text{not } x_1) \text{ or } x_7)$
 $((\text{not } x_1) \text{ or } x_6)$
etc.

x_1, x_2, x_3 , etc. are our Boolean variables
(to be set to True or False)

Should x_1 be set to False??

10 Pages Later:

185 -9 0
185 -1 0
177 169 161 153 145 137 129 121 113 105 97
89 81 73 65 57 49 41
33 25 17 9 1 -185 0
186 -187 0
186 -188 0
...



i.e., (x_{177} or x_{169} or x_{161} or x_{153} ...
 x_{33} or x_{25} or x_{17} or x_9 or x_1 or (not x_{185}))

clauses / constraints are getting more interesting...

Note x_1 ...

4,000 Pages Later:

10236 —10050 0
10236 —10051 0
10236 —10235 0
10008 10009 10010 10011 10012 10013 10014
10015 10016 10017 10018 10019 10020 10021
10022 10023 10024 10025 10026 10027 10028
10029 10030 10031 10032 10033 10034 10035
10036 10037 10086 10087 10088 10089 10090
10091 10092 10093 10094 10095 10096 10097
10098 10099 10100 10101 10102 10103 10104
10105 10106 10107 10108 —55 —54 53 —52 —51 50
10047 10048 10049 10050 10051 10235 —10236 0
10237 —10008 0
10237 —10009 0
10237 —10010 0

...

Finally, 15,000 Pages Later:

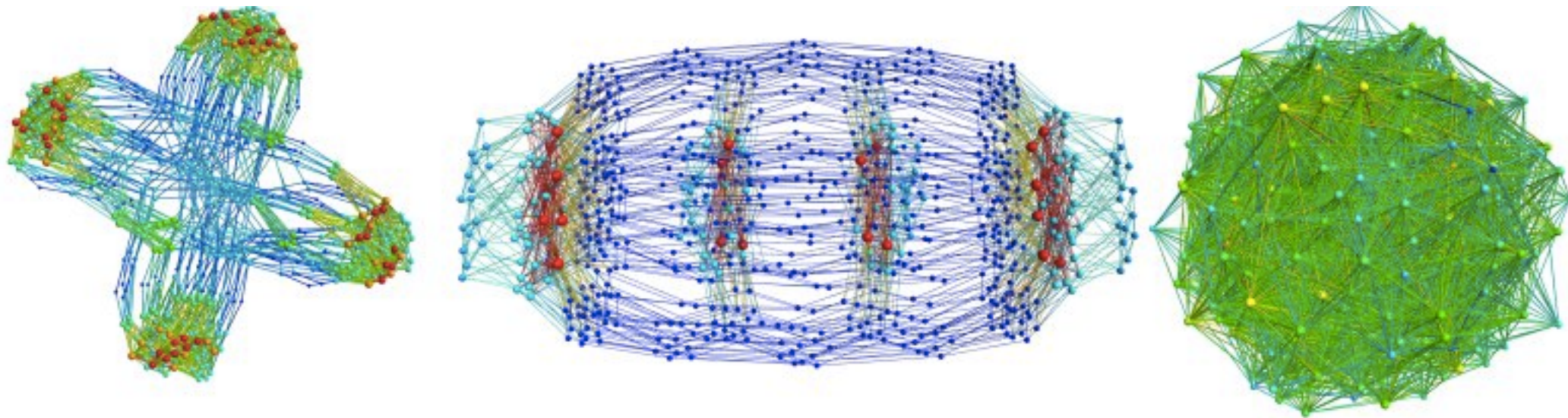
—7 260 0
7 —260 0
1072 1070 0
—15 —14 —13 —12 —11 —10 0
—15 —14 —13 —12 —11 10 0
—15 —14 —13 —12 11 —10 0
—15 —14 —13 —12 11 10 0
—7 —6 —5 —4 —3 —2 0
—7 —6 —5 —4 —3 2 0
—7 —6 —5 —4 3 —2 0
—7 —6 —5 —4 3 2 0
185 0

Search space of truth assignments: $2^{50000} \approx 3.160699437 \cdot 10^{15051}$

Current SAT solvers solve this instance in just a few seconds!

SAT 求解器

- SAT 求解器非常的强大
- 甚至有SAT 求解器的国际比赛



SAT Competition 2013

SAT 求解器

- 为了使用SAT 求解器进行漏洞挖掘，我们需要将X86指令的语义翻译成DIMACS 格式表示的布尔公式。这将非常非常的困难。
- 这是一个非常酷的项目，将比特币采矿问题转化为CNF 格式，然后使用SAT 求解器进行求解!!!
- <http://jheusser.github.io/2013/02/03/satcoin.html>
- 幸运的是我们有另一种非常强大的求解器, SAT求解器的演变: SMT 求解器!

SyScan360

SMT 求解器

SMT 求解器

- 类似于SAT求解器，但支持多种理论，不仅仅是布尔运算符。
- 极其地强大
- 表现力
 - 更容易表达X86-64指令的语义

SMT 求解器

- 允许我们确定必要的值来满足代码的约束条件。
- 微软 Z3 是用来证明 hyper-V 的虚拟机管理程序核心代码的正确性。
- 据报导，微软SAGE项目已经发现了微软产品的几个bug。

微软 Z3

- Z3 是可满足模理论的 (SMT) 求解器。也就是说,它是一个自动化的,通过内置理论对一阶逻辑多种排列进行可满足性校验, 包括对量词的支持。目前支持的理论如下:
 - equality over free (又名uninterpreted) 函数和谓词符号,
 - 实数和整形运算(有限支持非线性运算),
 - 位向量,
 - 阵列,
 - 元组/记录/枚举类型和代数 (递归) 数据类型

Z3 SMT 求解器

- 微软 Z3 SMT 求解器
- 在线版本详见 <http://rise4fun.com/Z3>
- Linux/Mac/Windows
- 免费用于非商业项目
- 商业许可 14,950.00 美元.

Z3 理论

- 基础知识
- 算术
- 位向量
- 阵列

Z3 理论-基础

Op	Mnmonics	Description
0	true	恒真
1	false	恒假
2	=	相等
3	distinct	不同
4	ite	if-then-else
5	and	n元 合取
6	or	n元 析取
7	iff	双向意义
8	xor	异或
9	not	否定
10	implies	意义

Z3 理论 – 位向量

Op	Mnmonics	Parameters	Description
0	bit1		constant comprising of a single bit set to 1
1	bit0		constant comprising of a single bit set to 0.
2	bvneg		Unary subtraction.
3	bvadd		addition.
4	bvsub		subtraction.
5	bvmul		multiplication.
6	bvsdiv		signed division.
7	bvudiv		unsigned division. The operands are treated as unsigned numbers.
8	bvsrem		signed remainder.
9	bvurem		unsigned remainder.
10	bvsmod		signed modulus.
11	bvule		unsigned <=.
12	bvsle		signed <=.
13	bvuge		unsigned >=.
14	bvsge		signed >=.
15	bvult		unsigned <.
16	bvslt		signed <.
17	bvugt		unsigned >.
18	bvsgt		signed >.

Z3 理论 – 位向量

- 19 bvand n-ary (associative/commutative) bit-wise and.
- 20 bvor n-ary (associative/commutative) bit-wise or.
- 21 bvnot bit-wise not.
- 22 bvxor n-ary bit-wise xor.
- 23 bvnand bit-wise nand.
- 24 bvnor bit-wise nor.
- 25 bvxnor bit-wise exclusive nor.
- 26 concat bit-vector concatenation.
- 27 sign n n-bit sign extension.
- 28 zero n n-bit zero extension.
- 29 extract hi:low hi-low bit-extraction.
- 30 repeat n repeat \$n\$ times.
- 31 bvredor or-reduction.
- 32 bvredand and-reduction.
- 33 bvcomp bit-vector comparison.
- 34 bvshl shift-left.
- 35 bvlsr logical shift-right.
- 36 bvrshr arithmetical shift-right.
- 37 bvrotate n n-bit left rotation.
- 38 bvrotate n n-bit right rotation.

测试

```
(simplify (bvule #x0a #xf0)) ; unsigned less or equal
(simplify (bvult #x0a #xf0)) ; unsigned less than
(simplify (bvuge #x0a #xf0)) ; unsigned greater or equal
(simplify (bvugt #x0a #xf0)) ; unsigned greater than
(simplify (bvsle #x0a #xf0)) ; signed less or equal
(simplify (bvslt #x0a #xf0)) ; signed less than
(simplify (bvsge #x0a #xf0)) ; signed greater or equal
(simplify (bvsgt #x0a #xf0)) ; signed greater than
```

提问

```
(declare-const a (_ BitVec 4))  
(declare-const b (_ BitVec 4))  
(assert (not (= (bvule a b) (bvsle a b))))  
(check-sat)  
(get-model)
```

Z3 求解器

演示

翻译和中间语言

SyScan360

代码约束

.text:00863614	movzx	ecx, word ptr [eax]	
.text:00863617	push	esi	
.text:00863618	xor	esi, esi	
.text:0086361A	test	cx, cx	
.text:0086361D	jz	short loc_863647	
.text:0086361F	movzx	ecx, cx	
.text:00863622			
.text:00863622 loc_863622:			
.text:00863622	cmp	cx, 20h	
.text:00863626	jz	short loc_863655	
.text:00863628	cmp	cx, 9	
.text:0086362C	jz	short loc_863655	
.text:0086362E			
.text:0086362E loc_86362E:			
.text:0086362E	cmp	cx, 22h	
.text:00863632	jz	loc_86435A	
.text:00863638			
.text:00863638 loc_863638:			
.text:00863638	push	eax	; lpsz

翻译

- 如何使用Z3对下列指令进行建模?
- 假设我们控制EBX 的值. 如何使用Z3找出EBX 的值, 使得JZ为真?

```
mov eax, ebx  
sub eax, 0x50  
cmp eax, 0x40  
jz  _branch2
```

翻译

- 既然我们要使用SMT 求解器求解x86 代码的约束限制, 我们需要将x86 指令翻译成SMT 公式!
- 我们有2个备选方案:
 1. 尝试直接将X86转换成SMT 公式
 2. 将X86翻译成中间语言 (IL) , 然后将IL翻译到SMT公式

x86 -> IL -> SMT

- 大多数程序分析工具首先将x86 翻译成一些中间语言，然后再将中间语言翻译成SMT公式。
- 明显的优势: 如果你需要支持其它的指令集，以ARM为例, 你只需要建立ARM指令到中间语言的翻译。

REIL

- 当前有几个可用的中间语言。 REIL 语言是我第一个经历的中间语言。
- REIL 是一种逆向工程中间语言。
- 由Zynamics开发 (现在由 Google接管)
- 用于BinNavi 产品
- 翻译x86-64 和ARM 到REIL
- 有比REIL更好的中间语言. 但REIL 更容易理解.

x86 指令集

- 关于为x86指令实现中间语言的若干思考
- x86 指令有间接影响效应
- x86 指令的语义非常的复杂

x86 – 间接影响

- push eax (intrinsic operands)

t1 ← eax

esp ← esp - 4

[esp] ← t1

- add eax, ebx

eax ← eax + ebx

update(eflags) //OF,SF,ZF,AF,CF,PF

REIL 指令集

- 少数指令
- 很好，因为我们只需要建立少数的从REIL指令到SMT公式的翻译。
- 不幸的是REIL具有若干的局限性

REIL- 算术

- add – 两数相加
- sub – 两数相减
- mul – 无符号乘运算
- div – 无符号除运算
- mod – 无符号求模运算
- bsh – 逻辑移位操作

REIL – 位运算指令

- and – 布尔与
- or – 布尔或
- xor – 布尔异或

REIL – 数据传输指令

- LDM – 从内存中加载一个值
- STM – 将值存储到内存中
- STR – 将值存储到寄存器

000000010025D300 *ldm* eax, , word t0

REIL – 条件

- BISZ – 将值与0比较
- JCC – 条件跳转

```
0000000010025D60D    bisz    word t10,    , byte ZF
0000000010025DA00    jcc     byte ZF,    , 0x10025E6
```

REIL – 其他

- UNDEF
- UNKN
- NOP

支持

- 通用x86 指令
- 不支持:
 - FPU
 - SSE, sse2, sse3
 - MMX
 - 不支持段选择子:-(
 - FS, GS

基本块(x86)

010025CB notepad.exe::_SkipBlanks@4

010025D3 movzx ecx, word ds:[eax]

010025D6 cmp word cx, word 0x20

010025DA jz loc_10025E6

基本块(REIL)

```
000000010025D300: ldm [DWORD eax, EMPTY , WORD t0]
000000010025D301: or [DWORD 0, WORD t0, DWORD ecx]
000000010025D600: and [DWORD ecx, WORD 65535, WORD t1]
000000010025D601: and [WORD t1, WORD 32768, WORD t2]
000000010025D602: and [WORD 32, WORD 32768, WORD t3]
000000010025D603: sub [WORD t1, WORD 32, DWORD t4]
000000010025D604: and [DWORD t4, DWORD 32768, WORD t5]
000000010025D605: bsh [WORD t5, WORD -15, BYTE SF]
000000010025D606: xor [WORD t2, WORD t3, WORD t6]
000000010025D607: xor [WORD t2, WORD t5, WORD t7]
000000010025D608: and [WORD t6, WORD t7, WORD t8]
000000010025D609: bsh [WORD t8, WORD -15, BYTE OF]
000000010025D60A: and [DWORD t4, DWORD 65536, DWORD t9]
000000010025D60B: bsh [DWORD t9, DWORD -16, BYTE CF]
000000010025D60C: and [DWORD t4, DWORD 65535, WORD t10]
000000010025D60D: bisz [WORD t10, EMPTY , BYTE ZF]
000000010025DA00: jcc [BYTE ZF, EMPTY , DWORD 16786918]
```

REIL bb → z3 (1/2)

```
(set-logic QF_BV)
(declare-fun t0 () (_ BitVec 32))
(declare-fun ecx () (_ BitVec 32))
(declare-fun t1 () (_ BitVec 32))
(declare-fun t2 () (_ BitVec 32))
(declare-fun t3 () (_ BitVec 32))
(declare-fun t4 () (_ BitVec 32))
(declare-fun t5 () (_ BitVec 32))
(declare-fun SF () Bool)
(declare-fun t6 () (_ BitVec 32))
(declare-fun t7 () (_ BitVec 32))
(declare-fun t8 () (_ BitVec 32))
(declare-fun OF () Bool)
(declare-fun t9 () (_ BitVec 32))
(declare-fun CF () Bool)
(declare-fun t10 () (_ BitVec 32))
(declare-fun ZF () Bool)
```

REIL bb → z3 (2/2)

```
(assert (= t10 (bvand (bvsb (bvand (bvor #x00000000 t0) #x0000FFFF) #x00000020) #x0000FFFF)))
(assert (= t6 (bvxor (bvand (bvand (bvor #x00000000 t0) #x0000FFFF) #x00008000) (bvand #x00000020 #x00008000))))
(assert (= SF (bvugt (bvlshr (bvand (bvsb (bvand (bvor #x00000000 t0) #x0000FFFF) #x00000020) #x00008000) #x0000000F)
#x00000000)))
(assert (= t7 (bvxor (bvand (bvand (bvor #x00000000 t0) #x0000FFFF) #x00008000) (bvand (bvsb (bvand (bvor #x00000000
t0) #x0000FFFF) #x00000020) #x00008000))))
(assert (= OF (bvugt (bvlshr (bvand (bvxor (bvand (bvand (bvor #x00000000 t0) #x0000FFFF) #x00008000) (bvand
#x00000020 #x00008000)) (bvxor (bvand (bvand (bvor #x00000000 t0) #x0000FFFF) #x00008000) (bvand (bvsb (bvand (bvor
#x00000000 t0) #x0000FFFF) #x00000020) #x00008000))) #x0000000F) #x00000000)))
(assert (= t5 (bvand (bvsb (bvand (bvor #x00000000 t0) #x0000FFFF) #x00000020) #x00008000)))
(assert (= t8 (bvand (bvxor (bvand (bvand (bvor #x00000000 t0) #x0000FFFF) #x00008000) (bvand #x00000020 #x00008000))
(bvxor (bvand (bvand (bvor #x00000000 t0) #x0000FFFF) #x00008000) (bvand (bvsb (bvand (bvor #x00000000 t0)
#x0000FFFF) #x00000020) #x00008000)))))
(assert (= t9 (bvand (bvsb (bvand (bvor #x00000000 t0) #x0000FFFF) #x00000020) #x00010000)))
(assert (= ZF (bvugt (ite (= (bvand (bvsb (bvand (bvor #x00000000 t0) #x0000FFFF) #x00000020) #x0000FFFF) #x00000000)
#x00000001 #x00000000) #x00000000)))
(assert (= t3 (bvand #x00000020 #x00008000)))
(assert (= t2 (bvand (bvand (bvor #x00000000 t0) #x0000FFFF) #x00008000)))
(assert (= CF (bvugt (bvlshr (bvand (bvsb (bvand (bvor #x00000000 t0) #x0000FFFF) #x00000020) #x00010000) #x00000010)
#x00000000)))
(assert (= t4 (bvsb (bvand (bvor #x00000000 t0) #x0000FFFF) #x00000020)))
(assert (= ecx (bvor #x00000000 t0)))
(assert (= t1 (bvand (bvor #x00000000 t0) #x0000FFFF)))
(check-sat)
(get-model)
```

求解

sat

```
(model
  (define-fun t0 () (_ BitVec 32)
    #x00000000)
  (define-fun t1 () (_ BitVec 32)
    #x00000000)
  (define-fun ecx () (_ BitVec 32)
    #x00000000)
  (define-fun t4 () (_ BitVec 32)
    #xffffffff0)
  (define-fun CF () Bool
    true)
  (define-fun t2 () (_ BitVec 32)
    #x00000000)
  (define-fun t3 () (_ BitVec 32)
    #x00000000)
  (define-fun ZF () Bool
    false)
  (define-fun t9 () (_ BitVec 32)
    #x00010000)
  (define-fun t8 () (_ BitVec 32)
    #x00000000)
  (define-fun t5 () (_ BitVec 32)
    #x00008000)
  (define-fun OF () Bool
    false)
  (define-fun t7 () (_ BitVec 32)
    #x00008000)
  (define-fun SF () Bool
    true)
  (define-fun t6 () (_ BitVec 32)
    #x00000000)
  (define-fun t10 () (_ BitVec 32)
    #x0000ffe0)
)
```


自动化

程序分析和约束求解器

漏洞挖掘自动化

- 有多种方法可以对漏洞挖掘自动化的尝试进行评估.
- 有些人喜欢静态分析, 有些人喜欢动态分析.
- 中间语言的选择非常的个人化.
- 你必须尝试一些方法, 并检查哪一种更适用于你的目标.

自动化

- 我所提出的方法是基于微软不可思议的SAGE 项目.
- 动态分析
- 已经在微软的产品上发现了几个bugs

进程

- 使用初始种子文件执行目标应用程序
- 执行跟踪
- 污点分析
- 将X86代码翻译成SMT公式
- SMT 用来生成新的输入
- 增加代码/路径覆盖率

执行跟踪

- 用于执行跟踪的强大工具
- 二进制指令分析仪: PIN, DynamoRio
- 调试器(slower)

污点分析

- 你不希望将整个跟踪转化为SMT公式
- 你只筛选出受到用户输入（文件）影响的指令
- 污点分析可以在中间语言顶层或者直接从x86指令反汇编实现。
- 最大的问题：系统调用！

污点分析- 系统调用

- 当某些系统调用是无文档的情况下，如何在系统中进行污点分析？
- 当一个无文档的系统调用使用了一个被污染的域，我们如何知道什么被污染了？我们如何知道系统调用返回的信息有没有被污染？我们是否需要追踪内核代码？
- 大多数系统只会考虑系统调用的一个很小子集：read, write, open, ..., 然后为污点传播建立硬编码规则。

翻译

- 这是你决定是否要使用中间语言的最根本原因之一。
- 创建一个直接从x86 到 SMT 的翻译是可行的。
- 最基本的解决方案包括使用一些模版引擎，将翻译硬编码到模版引擎。
- 由于 SMT-LIB 不接受同一变量的多重赋值, 你不得不为变量创建多种版本系统 (类似于SSA)
- 你也可以基于很棒的Z3Py接口使用Python直接进行翻译.

策略

- 从Z3求解获得结果之后, 你获得了新的输入。
- 你想用怎样的搜索策略? 广度优先? 深度优先? 随机?
- 你也可以优先考虑一些痕迹特征, 包括一些有趣的模式, 例如循环, 内存分配大小的计算, 等等.

演示

总结

SyScan360

总结

- 这只是关于约束求解器用于逆向工程任务的介绍性演讲
- 程序分析很难，有很多犄角旮旯，具有挑战性
- 指令集翻译很难而且很耗时
- 在逆向工程和程序分析中，有很多事情可以而且需要自动化
- SMT 求解器非常强大，但并不是在所有情况下都用它
- 我们需要更多的开源工具

谢 谢