第三届XMan夏令营

Vulnerability Discovering

By r3kapig

About me

- swing
  - CTF Player @FlappyPig @JD-r3kapig @lotus-r3kapig
  - http://bestwing.me beswing@outlook.com
- Software Security
  - Focus on Vulnerability Discovering
  - Active in CTF, PWN/Reverse/Misc/



How to ?

- Code audit
  - Google v8 https://github.com/v8/v8
  - Redis
- Reverse
  - Adobe PDF
- Fuzzing
  - all

Code audit

- open souce
- Starting from the whole, key components are audited separately
- target：
  - buffer overflow
  - printf format strings
  - UAF

Fuzzing

- **Fuzzing** or **fuzz testing** is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program.

- Feture：
  - Automated or semi-automated
  - High return

目录
Contents

数据流分析

DATA-FLOW ANALYSIS

Part 01

什么是数据流分析

what

Part 1

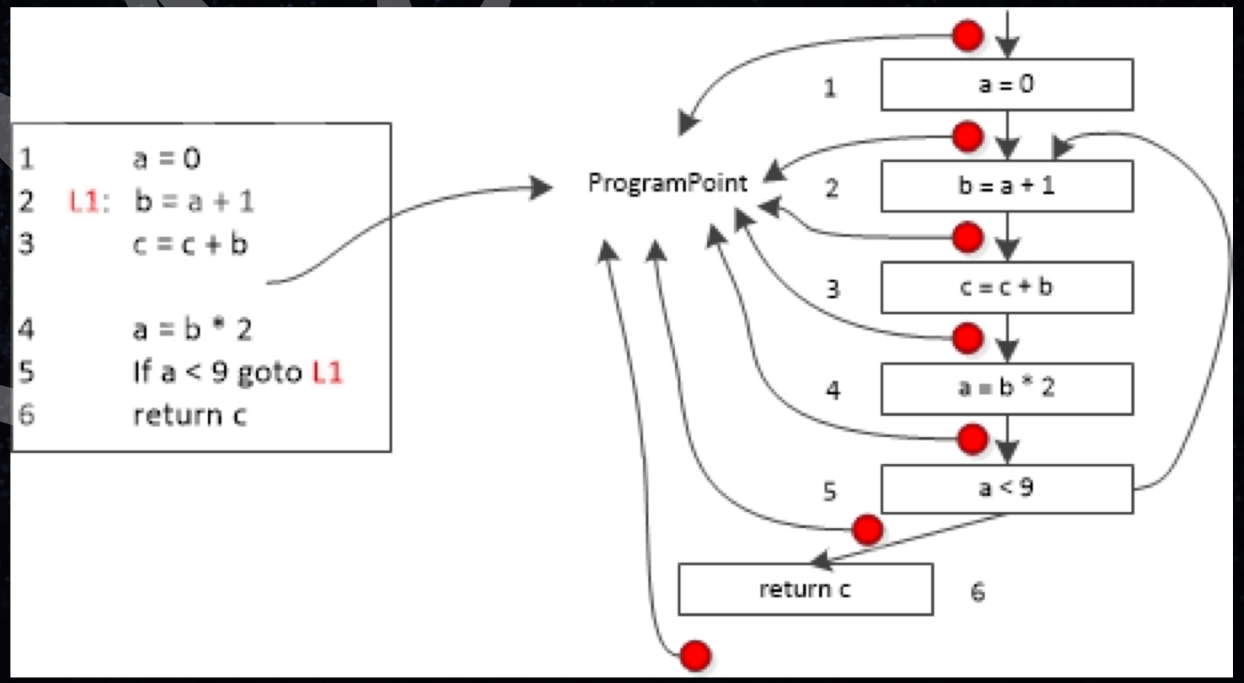
在编译原理中有一个概念叫数据流，那么什么是数据流？

数据流分析指的是一组用来获取有关数据如何沿着程序执行路径流动的相关信息的技术.《

数据流分析的目的是提供一个过程（或一大段程序）如何操纵其数据的全局信息.《高级编

Introduction

数据流分析是一种用来获取相关数据沿着程序执行路径流动的信息分析技术



分类

- 根据对程序路径的分析精度分类：
  - 流不敏感分析（flow insensitive）：不考虑语句的先后顺序，按照程序语句的物理位置从上往下顺序分析每一语句，忽略程序中存在的分
  - 流敏感分析（flow sensitive）：考虑程序语句可能的执行顺序，通常需要利用程序的控制流图（CFG）
  - 路径敏感分析（path sensitive）：不仅考虑语句的先后顺序，还对程序执行路径条件加以判断，以确定分析使用的语句序列是否对应着一条可实际运行的程序执行路径

分类

- 根据分析程序路径的深度分类：
  - 过程内分析（intraprocedure analysis）：只针对程序中函数内的代码
  - 过程间分析（inter-procedure analysis）：考虑函数之间的数据流，即需要跟踪分析目标数据在函数之间的传递过程

漏洞审计

- 由于一些程序漏洞的特征恰好可以表现为特定程序变量在特定的程序点上的性质、状态或取值不满足程序安全的规定，因此数据流分析可以直接用于检测这些漏洞。
  - 指针二次释放问题

```
free(p);
[...]
free(p);
```

```
p = q;
free(q);
[...]
*p = 1;
```

漏洞审计

- 由于一些程序漏洞的特征恰好可以表现为特定程序变量在特定的程序点上的性质、状态或取值不满足程序安全的规定，因此数据流分析可以直接用于检测这些漏洞。

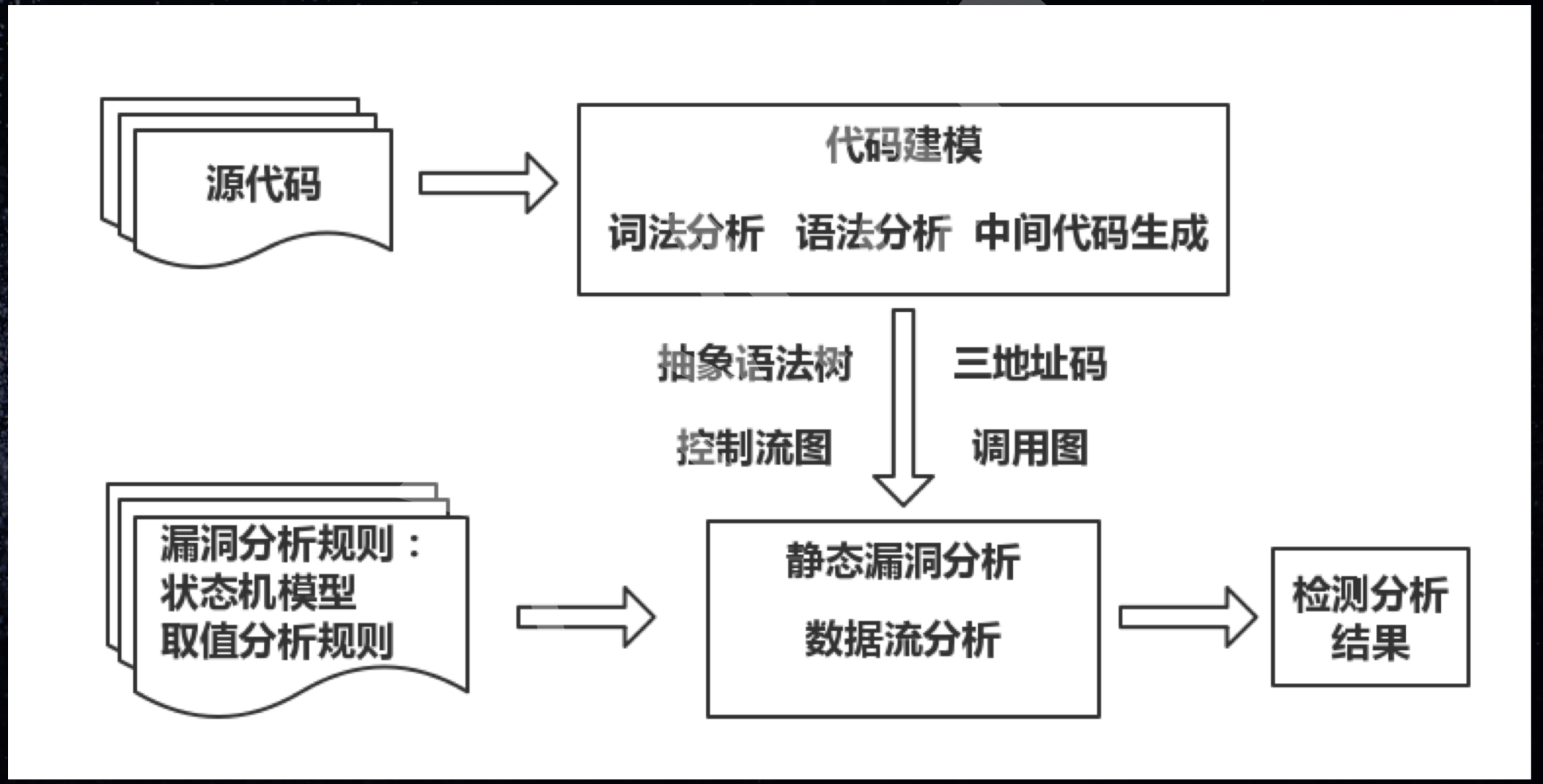
  - 再比如一个数据越界问题
  - a[i] = 1;
    - 使用数据流分析方法一方面记录数组 a 的长度，另一方面分析变量 i 的取值，并进行比较，以判断数据的访问是否越界。

漏洞审计

- 由于一些程序漏洞的特征恰好可以表现为特定程序变量在特定的程序点上的性质、状态或取值不满足程序安全的规定，因此数据流分析可以直接用于检测这些漏洞。

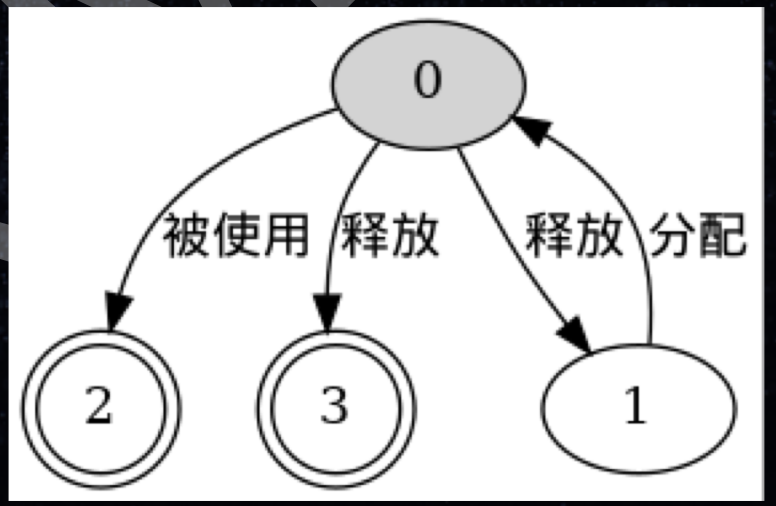
  - 再比如一个缓冲区溢出问题
  - strcpy(x, y);
    - 记录下变量 x 被分配空间的大小和变量 y 的长度，如果前者小于后者，则判断存在缓冲区溢出。

数据流分析


源代码
代码建模
词法分析 语法分析 中间代码生成
抽象语法树
三地址码
控制流图
调用图
漏洞分析规则：
状态机模型
取值分析规则
静态漏洞分析
数据流分析
检测分析
结果

漏洞分析规则

- 程序漏洞通常和程序中变量的状态或者变量的取值相关。状态自动机可以描述和程序变量状态相关的漏洞分析规则，自动机的状态和变量相应的状态对应。和变量取值相关的检测规则通常包含和程序语句或者指令相关的对变量取值的记录规则以及在特定情况下变量取值需要满足的约束。



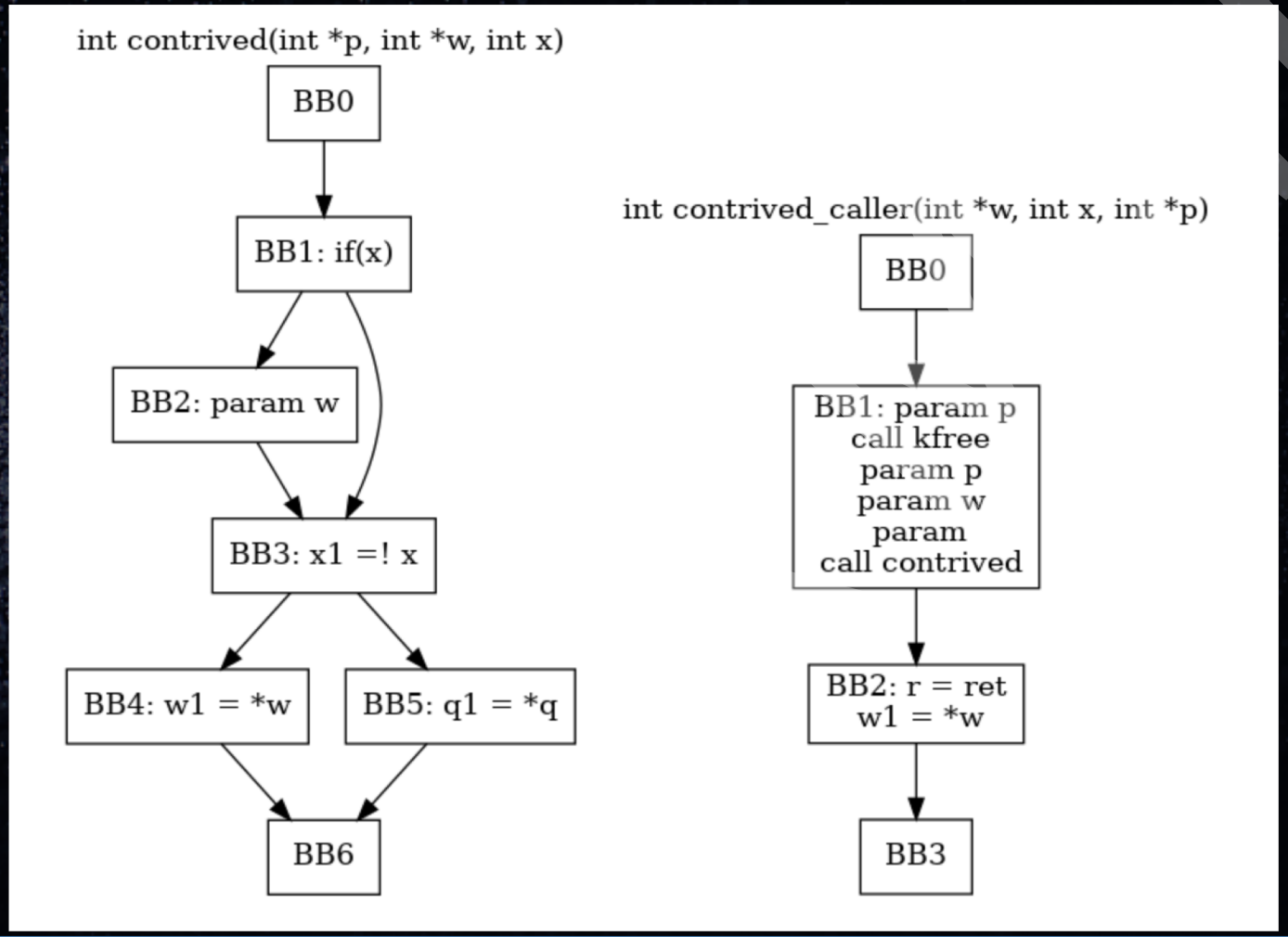
实例分析

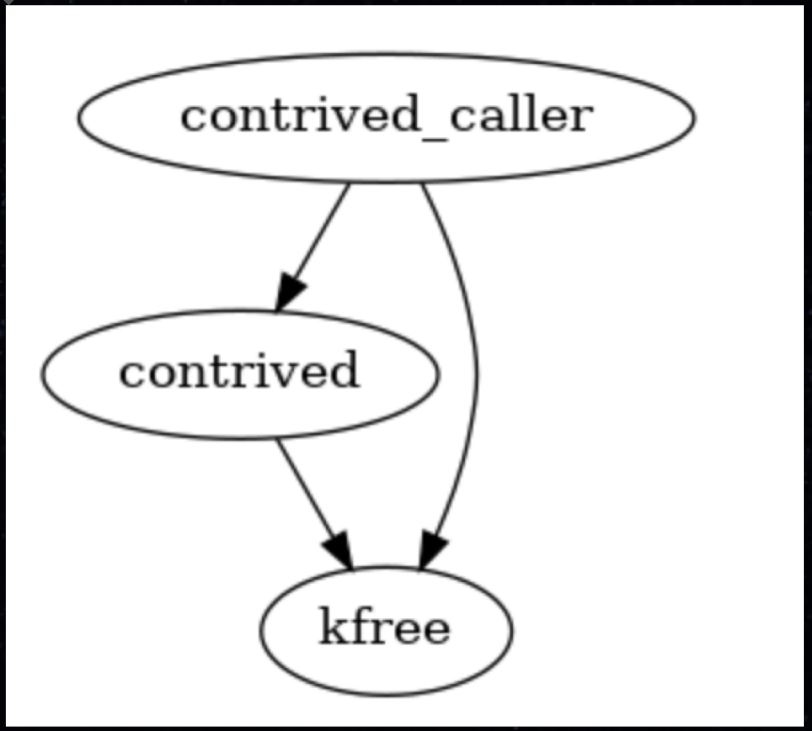
- 检测指针变量的错误使用

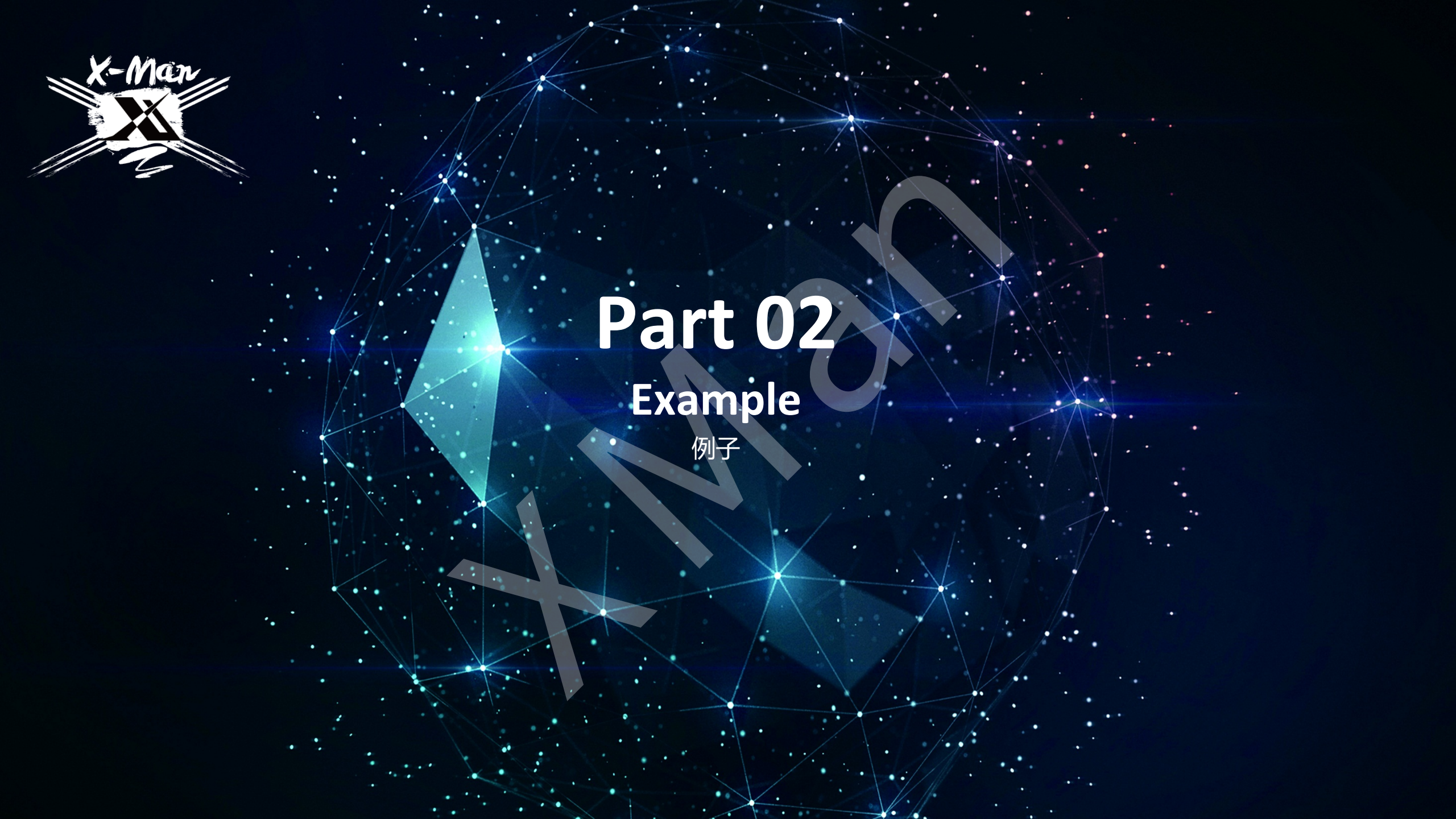
```
int contrived(int *p, int *w, int x) {
    int *q;
    if (x) {
        kfree(w); // w free
        q = p;
    }
    [...]
    if (!x)
        return *w;
    return *q;  // p use after free
}
int contrived_caller(int *w, int x, int *p) {
    kfree(p);   // p free
    [...]
    int r = contrived(p, w, x);
    [...]
    return *w;  // w use after free
}
```

实例分析

- 如果采用路径敏感的数据流分析






X-Man
Part 02
Example
例子
XMan

符号执行

SYMBOLIC EXECUTION

目录
Contents

目录
Contents

Part 01

什么是符号执行

- Here is a example

Example

```
input = raw_input('Input your password: ')

if input == 'hunter2':

    print 'Success.'

else:

    print 'Try again.'
```

There are many solutions:

- Use objdump or readelf to find the string 'hunter2'.
- Use ltrace to find the comparison.

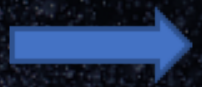
Example

So What's Symbolic Execution? It's a system that walks through all* possible paths of a program.

Let's work through the example.

Symbol

Here is what we want



```
input = raw_input 'Input your password: '
if input == ‘hunter2'
  print 'Success.'
else
  print 'Try again.'
```

Symbol

Now we are here

```
input = raw_input 'Input your password: '

if input == ‘hunter2'

  print 'Success.'

else

  print 'Try again.'
```

What is a symbol :

$$x^2 + 2x + 3 = 4$$

Just like the 'x'

Symbol

Think of a symbol as x, except that it's a variable in the program.

We don't know what x is. We want to find out. Same with a symbol.

x depends on the equation(s) that constrain it.

Branch

Now we are here

```
input = raw_input 'Input your password: '

if input == 'hunter2'

  print 'Success.'

else

  print 'Try again.'
```

Path 1: if input equals 'hunter2'

```
input = raw_input('Enter the password: ')

if input == 'hunter2'   # it is equal!
```

Path 2: if input does not equal 'hunter2'

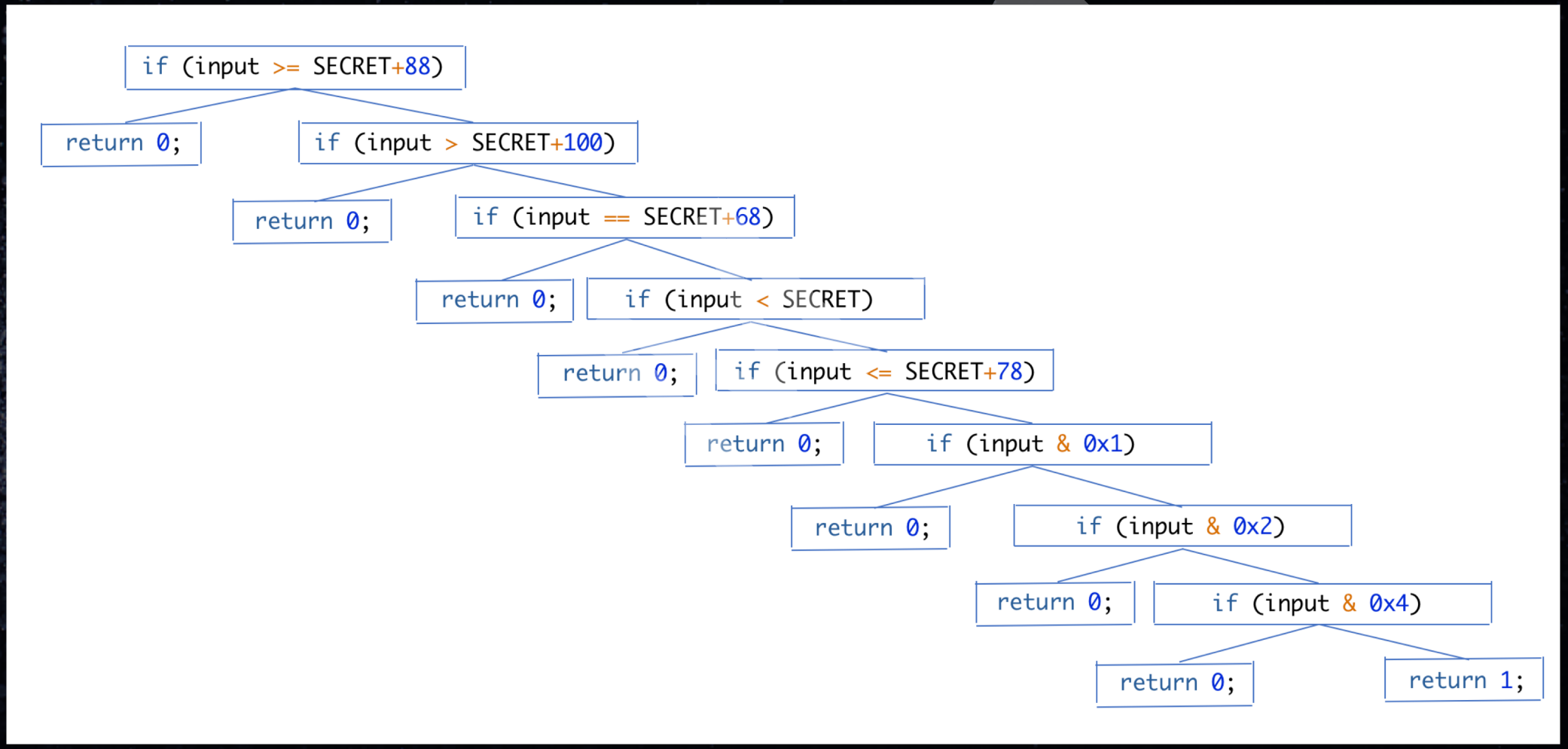
```
input = raw_input('Enter the password: ')

if input == 'hunter2'   # it's not equal.
```

A Complex Example

```
#define SECRET 100
int check_code(int input) {
    if (input >= SECRET+88) return 0
    if (input > SECRET+100) return 0
    if (input == SECRET+68) return 0
    if (input < SECRET) return 0
    if (input <= SECRET+78) return 0
    if (input & 0x1) return 0
    if (input & 0x2) return 0
    if (input & 0x4) return 0
    return 1;
}
```


What are the possible paths?

A Complex Example



A Complex Example

We can perform any tree search algorithm to find the node that returns 1.

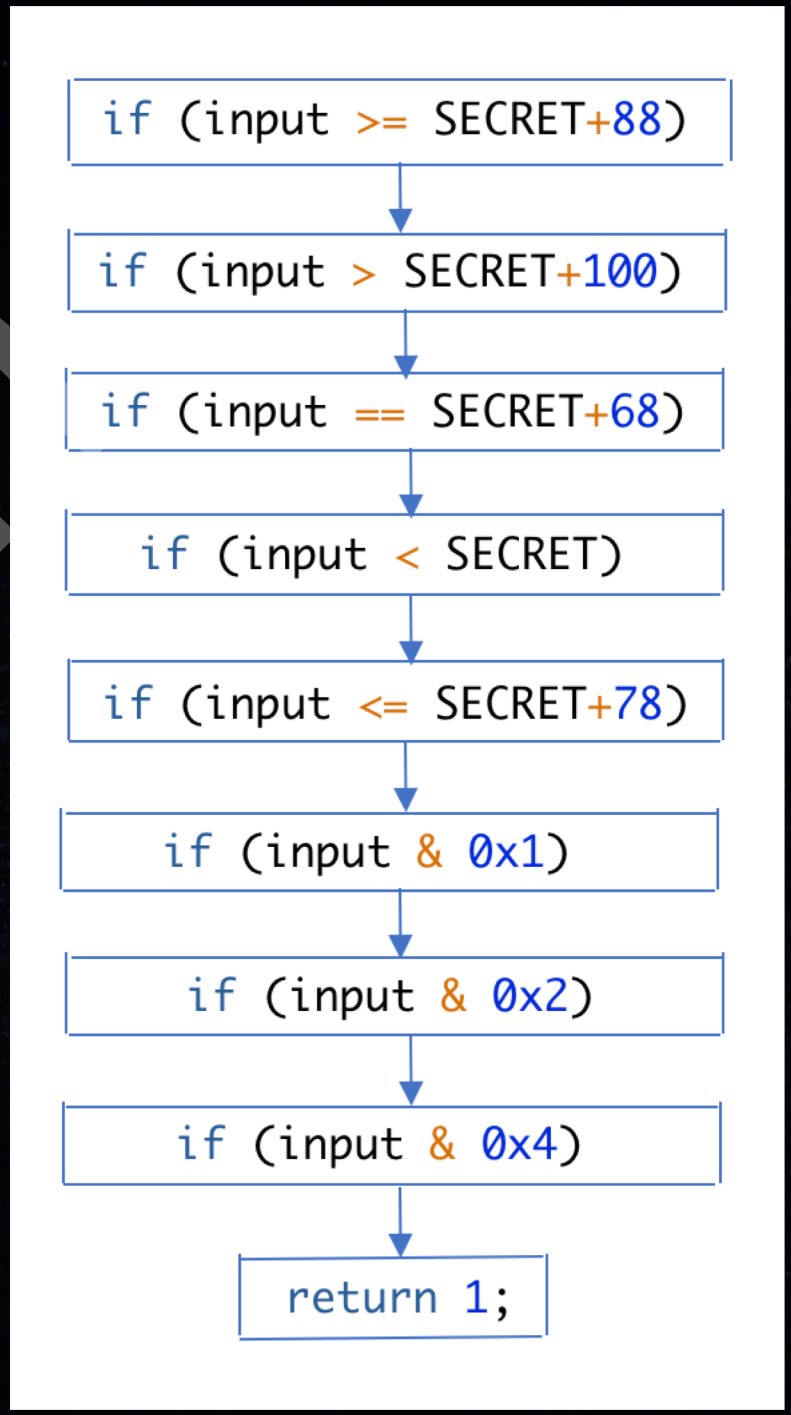
Breadth-first search is a great choice (and, by default, what Angr uses.)

A Complex Example

Once we have a path, we can build an equation that can be solved with a satisfiability (SMT) solver:

```
   input >= SECRET+88
∧ input > SECRET+100
∧ input == SECRET68
∧ input < SECRET
∧ input <= SECRET+78
∧ input & 0x1
∧ input & 0x2
∧ input & 0x4
```

Remember, SECRET = 100.



The really life

e many symbols and many branches. The exponential growth

The really life

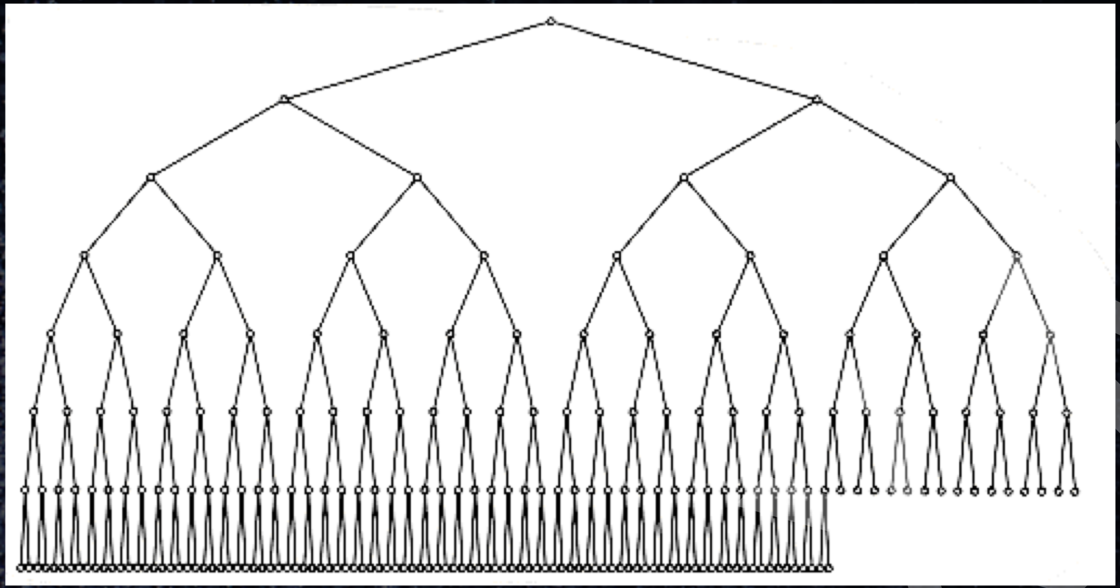
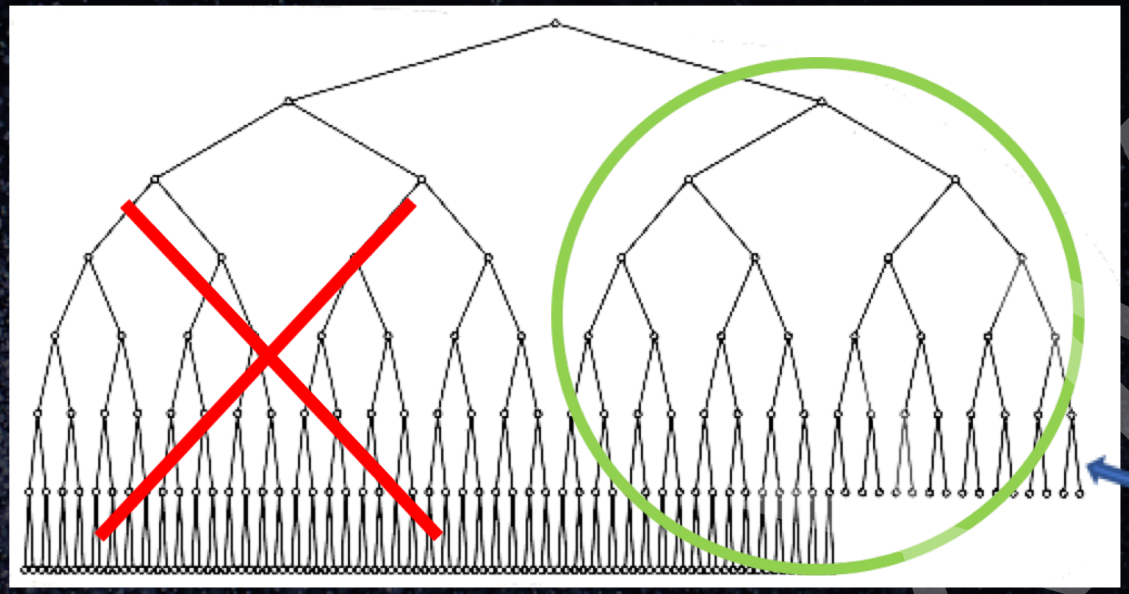


Image source: http://www.icodeguru.com/vc/10book/books/book3/chap6.htm

One of the biggest problems with symbolic execution:

With each if statement, the number of possible branches might double. The growth of the problem is exponential with respect to the size of the program.

The really life



If you can identify conditions that would indicate that it is unlikely that continuing would lead to a successful state, you can terminate the path immediately and cross off large sections of the state graph. We only have to search half of the graph! Saves a lot

Computer

How do we step through the program, find the branch we want, and so

ilt a powerful tool called Angr to do that for us. It's written in Python and i

Angr

https://github.com/angr

Template

```
import angr
def should_accept_path(path):
    # Check if this path should be accepted as successful.
    return ...
def should_avoid_path(path):
    # Check if this path is unlikely to result in a successful state.
    return ...
# Load the binary.
project = angr.Project('binary')
# Specify the starting state.
initial_state = project.factory.entry_state()
# Create the path group.
path_group = project.factory.path_group(initial_state)
found_path = None
# Continue to step until we find an accepted path or all paths terminate.
while len(path_group.active) > 0 && found_path is None:
    # Step all active paths.
    path_group.step()
    # Iterate through all active paths to figure out if we should drop any
    # or if any are successful.
    paths_to_avoid = []
    for active_path in path_group.active:
        if should_accept_path(active_path):
            # If we find an accepted path, exit the loop.
            found_path = active_path
            break
        elif should_avoid_path(active_path):
            # If we find a path that needs to be avoided, mark it as such.
            paths_to_avoid.append(active_path)
    # Remove any paths we marked as needing to drop from the stash.
    path_group.drop(stash='active', filter_func=lambda path: path in paths_to_avoid)
```

Template

The previous algorithm is so common that Angr wrote a single function to do it for you, called the 'explore' function:

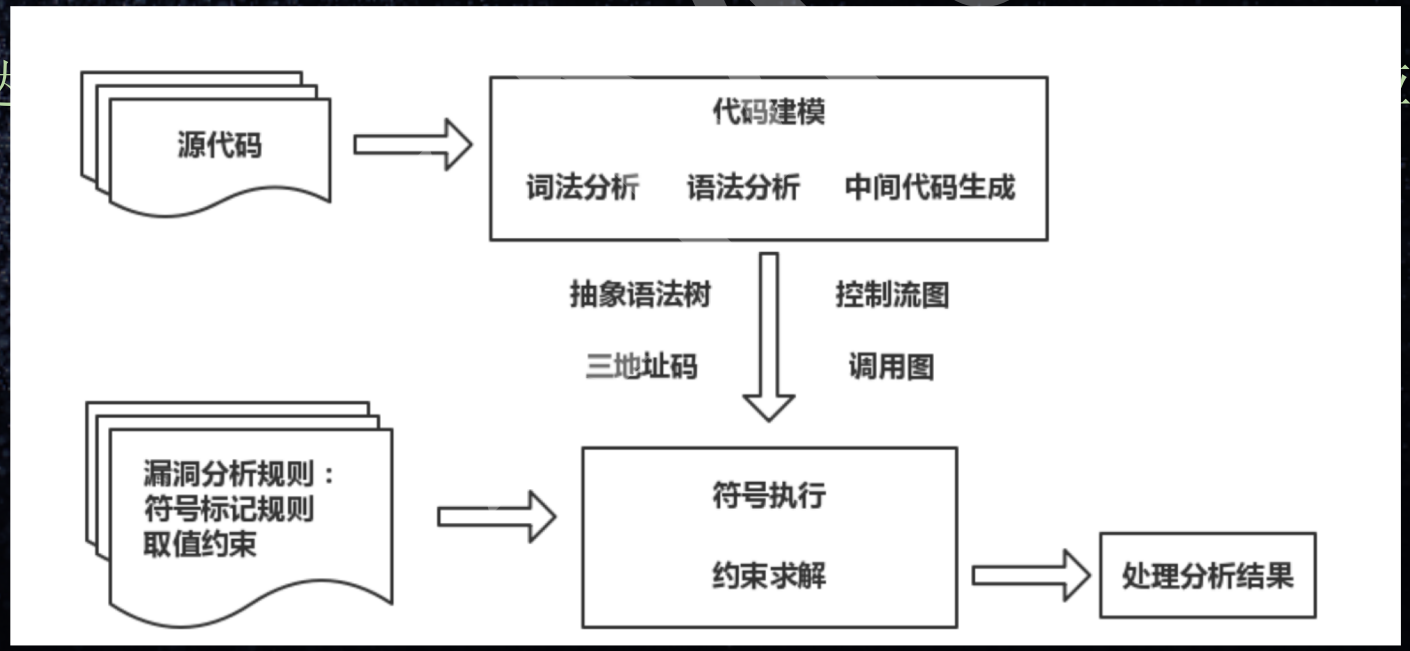
```
path_group.explore(find=should_accept_path, avoid=should_avoid_path)
```

… will add any path that is accepted to the list 'path_group.found

Additionally, searching or avoiding a specific instruction address is common enough that the find and avoid parameters also accept addresses:

检测漏洞

程序中变量的取值可以被表示为符号值和常量组成的计算表达式，而一些程序漏洞可以表现为某些相关变量的取值不满足相应的约束，这时通过判断表示变量取值的表达的漏洞。



检测漏洞

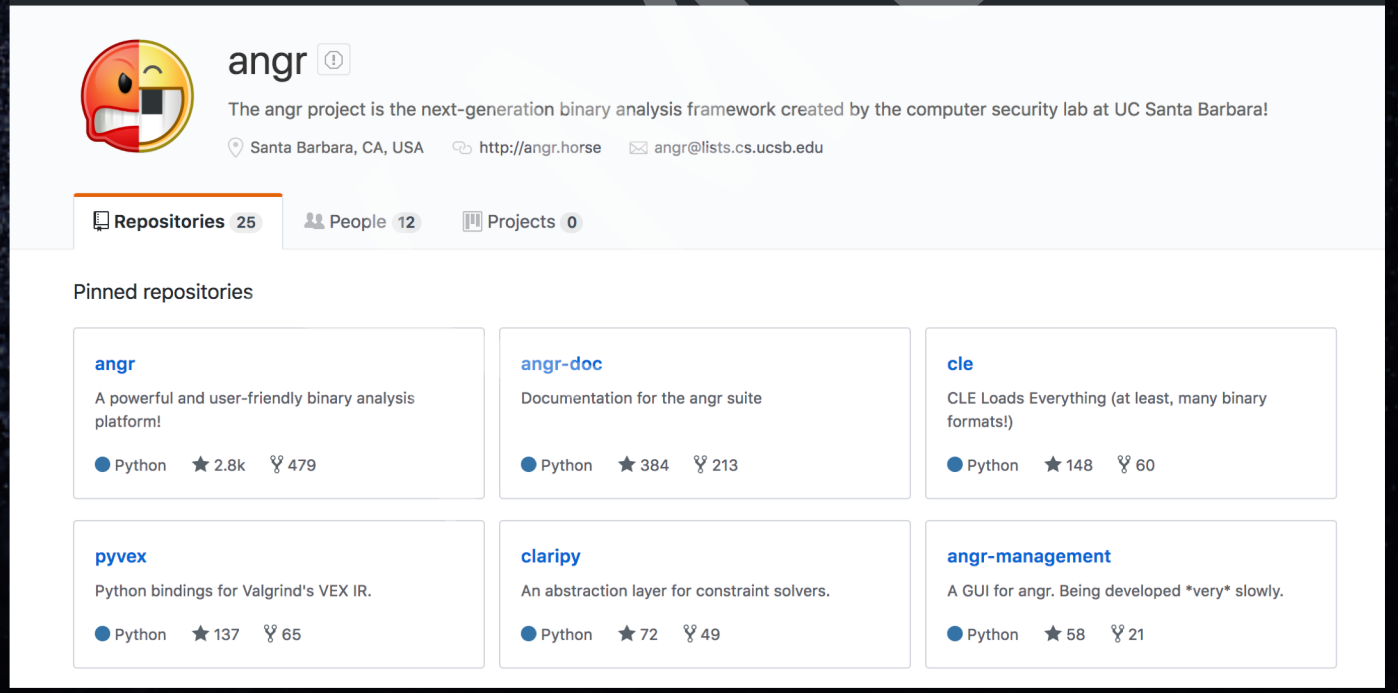
```
int a[10];
scanf("%d", &i);
if (i > 0) {
    if (i > 10
        i = i % 10
    a[i] = 1
```

首先，将表示程序输入的变量 i 用符号 x 表示其取值，通过分别对 if 条件语句的两条分支进行分析，可以发现在赋值语句 a[i] = 1 处，

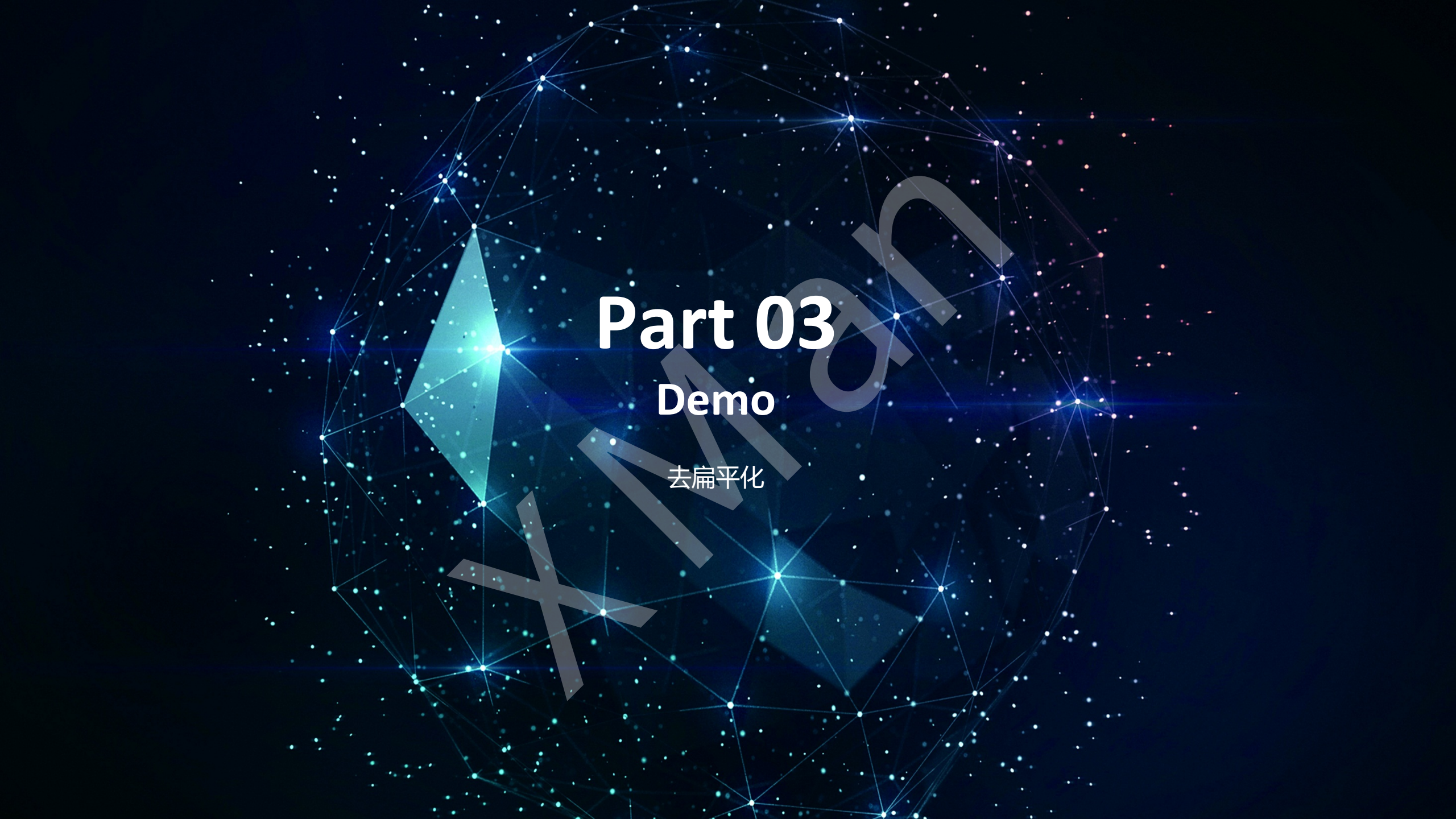

Part 02
angr
A easy CTF program
xman

angr

angr 是一个多架构的二进制分析平台，具备对二进制文件的动态符号执行能力和多种静态分析能力。在近几年的 CTF 中也大有用途。

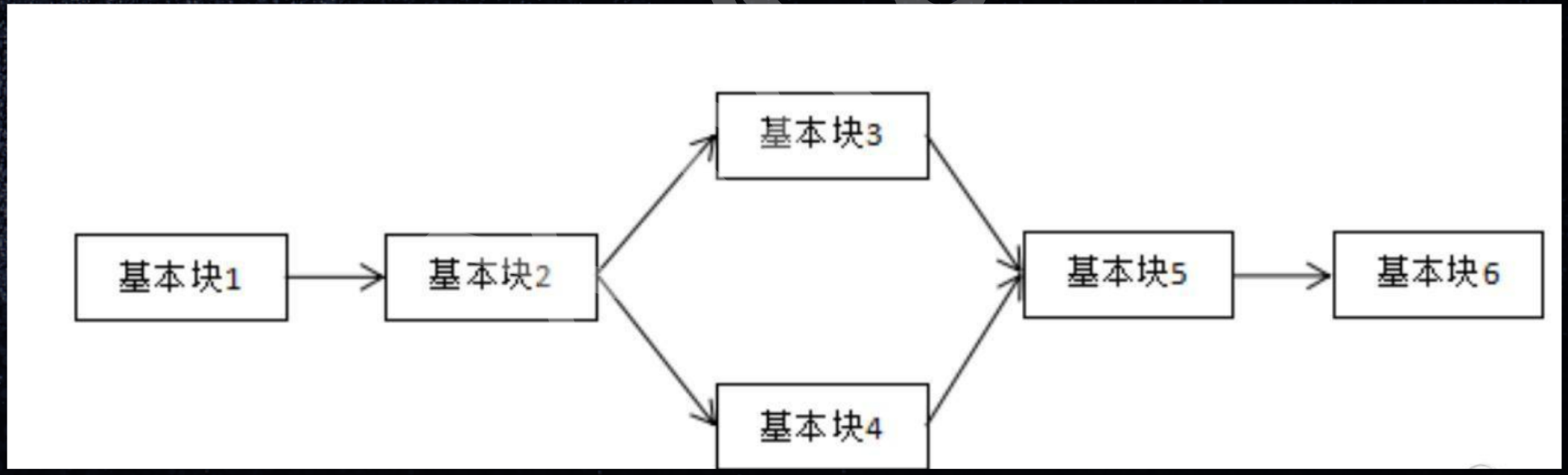



Part 03
Demo
A easy CTF program
xMan


Part 03
Demo
去扁平化
xman

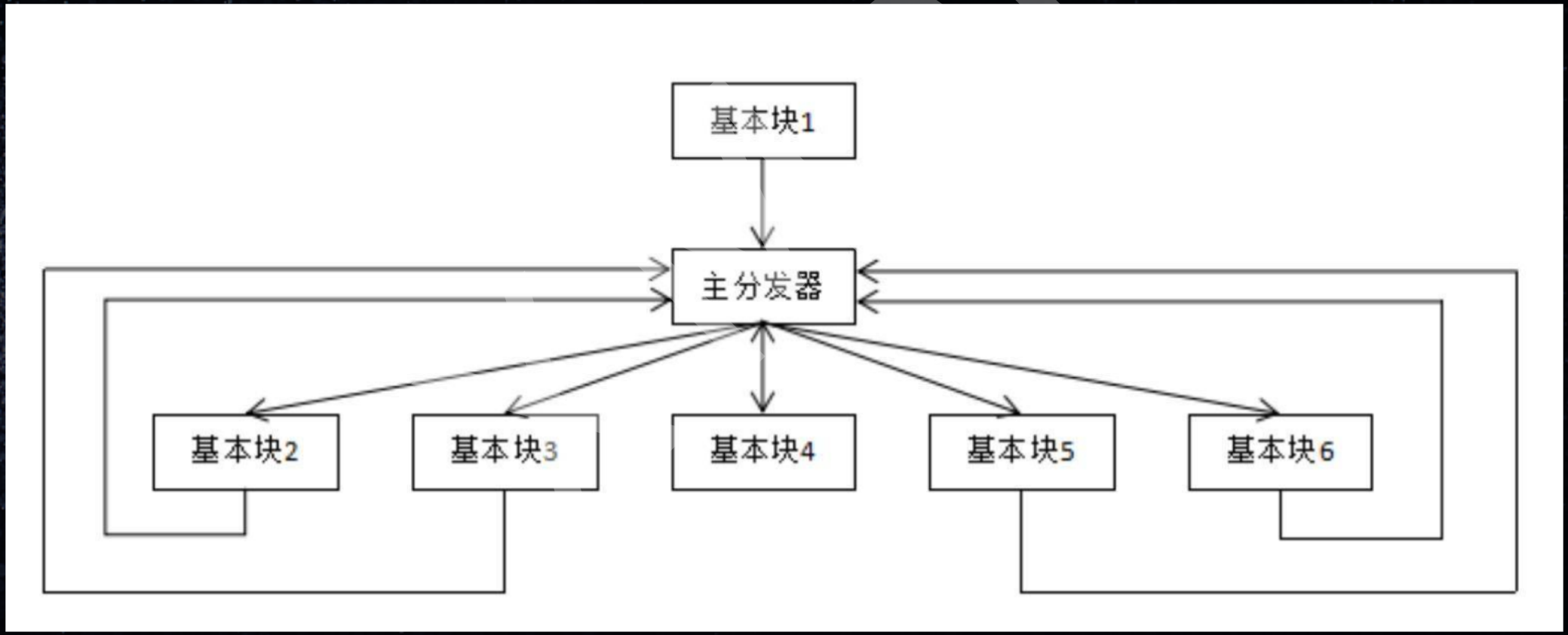
什么是扁平化

- 控制流平坦化(control flow flattening)的基本思想主要是通过一个主分发器来控制程序基本 块的执行流程，例如下图是正常的执行流程



什么是扁平化

- 经过控制流平坦化后的执行流程就如下图

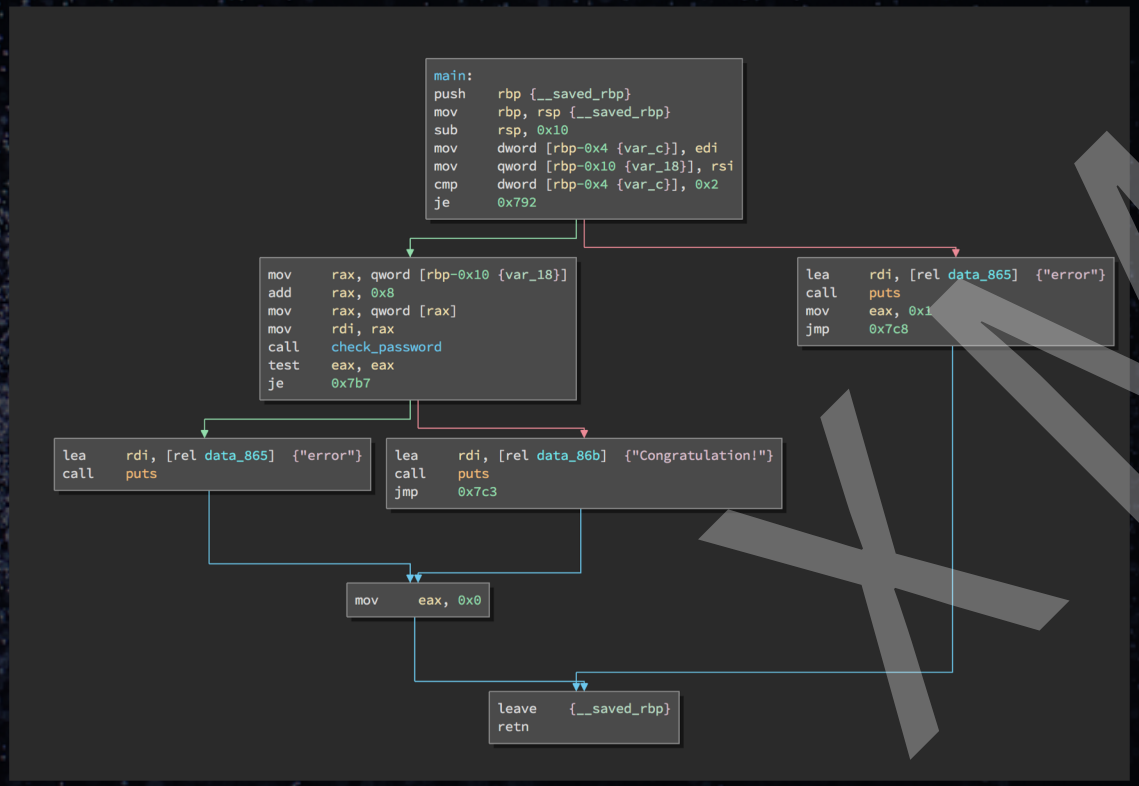


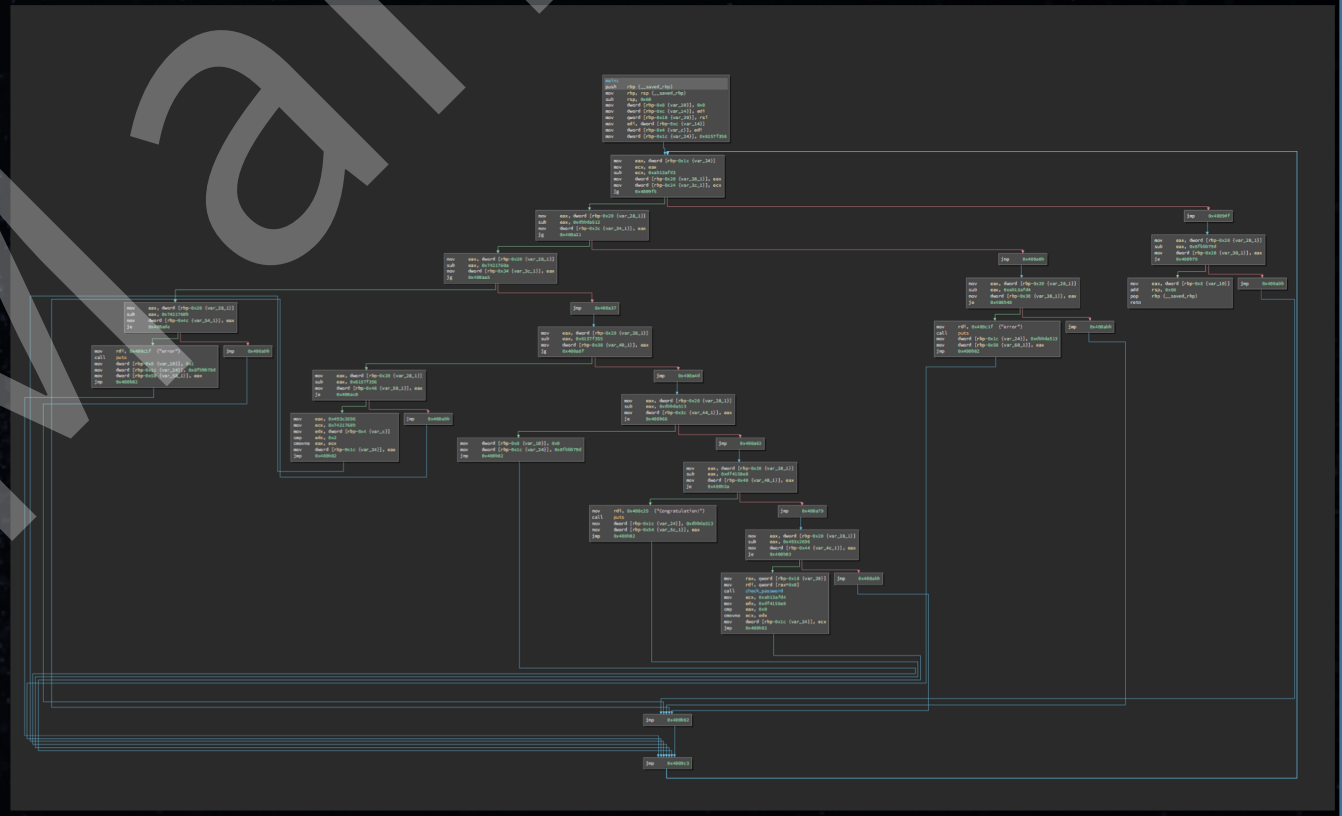
符号执行

- A

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int check_password(char *passwd) {
    int i, sum = 0;
    for (i = 0; ; i++) {
        if (!passwd[i]) {
            break;
        }
        sum += passwd[i];
    }
    if (i == 4) {
        if (sum == 0x1a1 && passwd[3] > 'c' && passwd[3] < 'e' &&passwd[0] == 'b') {
            if ((passwd[3] ^ 0xd) == passwd[1]) {
                return 1;
            }
            puts("Orz...");
        }
    }
    else{
        puts("len error");
    }
    return 0;
}

int main(int argc, char **argv) {
    if (argc != 2){
        puts("error");
        return 1;
    }
    if (check_password(argv[1])){
        puts("Congratulation!");
    }
    else{
        puts("error");
    }
    return 0;
}
```

去扁平化


main:
push rbp {__saved_rbp}
mov rbp, rsp {__saved_rbp}
sub rsp, 0x10
mov dword [rbp-0x4 {var_c}], edi
mov qword [rbp-0x10 {var_18}], rsi
cmp dword [rbp-0x4 {var_c}], 0x2
je 0x792
mov rax, qword [rbp-0x10 {var_18}]
add rax, 0x8
mov rax, qword [rax]
mov rdi, rax
call check_password
test eax, eax
je 0x7b7
lea rdi, [rel data_865] {"error"}
call puts
mov eax, 0x1
jmp 0x7c8
lea rdi, [rel data_865] {"error"}
call puts
lea rdi, [rel data_86b] {"Congratulation!"}
call puts
jmp 0x7c3
mov eax, 0x0
leave {__saved_rbp}
retn



插桩

INSTRUMENTATION

其他

OTHER

Man


BUSINESS TEMPLATE
THANKS
If any questions, Plz contact me.
xMan