ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

# Análise estática de código para segurança

**ISEL – Instituto Superior de Engenharia de Lisboa**
Rua Conselheiro Emídio Navarro, 1 | 1959-007 Lisboa

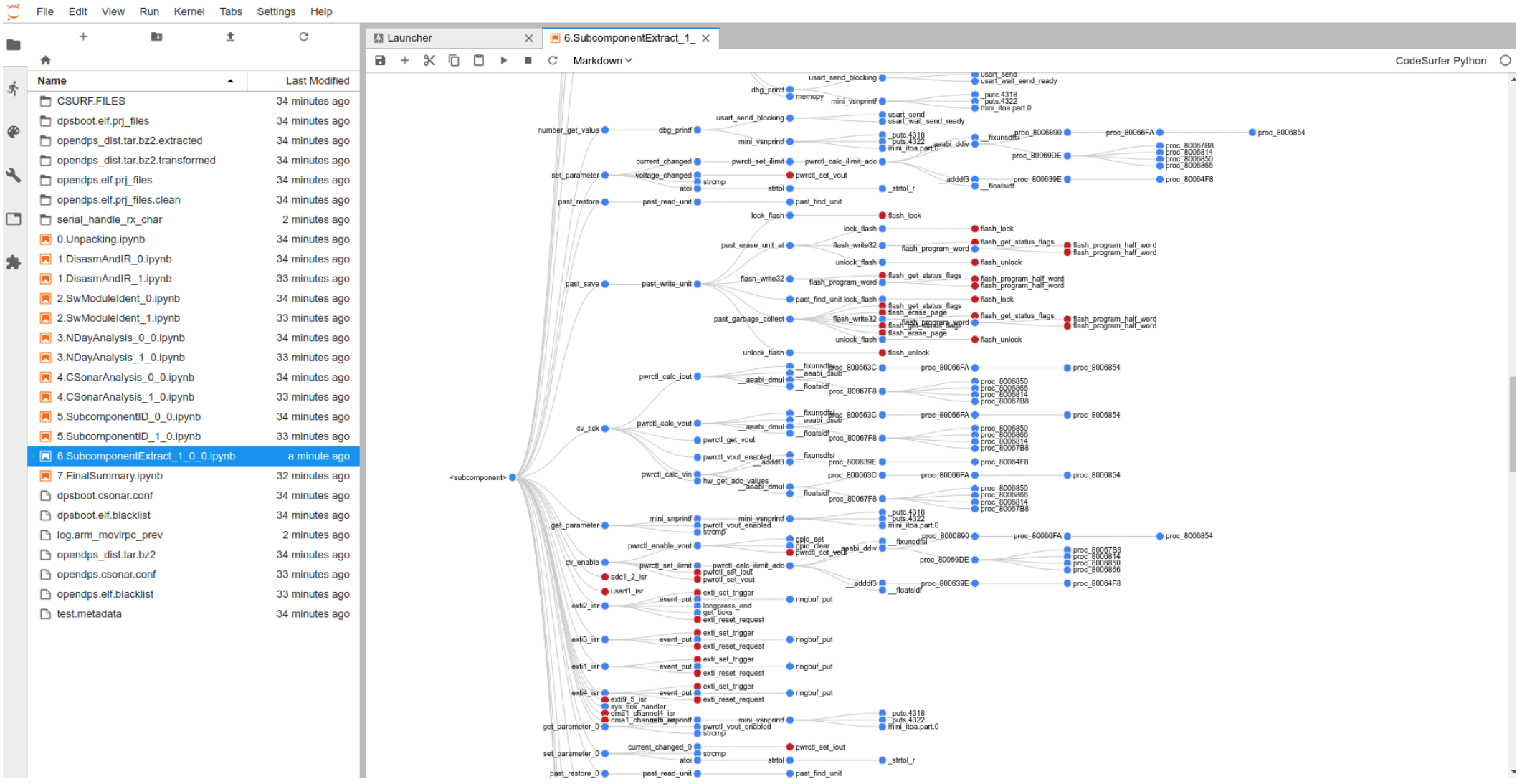
# Introdução

- Como encontrar bugs/vulnerabilidades no código?
- Análise manual de código
- Análise automática de código
  - Análise simbólica
  - Análise semântica
    - Fluxo de controlo
    - Fluxo de dados
- Exemplos de ferramentas para Static Application Security Testing (SAST)

# Análise manual de código

- Estratégias de compreensão do código
  - Seguir entradas maliciosas
  - Analisar por módulo ou algoritmo
- Estratégias de pontos candidatos
  - Estratégia genérica de procura de candidatos
  - Identificar pontos candidatos simples com ferramentas como o grep ou findstr
  - Seguir o resultado da injeção de ataques

# Ferramentas de navegação de código



# Como fazer análise automática de código?

- Objectivo: identificar vulnerabilidades no código procurando por função vulneráveis
- Testes unitários?
  - 90% dos erros envolvem interações de várias funções
  - Talvez seja por isso que o programador não detetou o problema
  - Erros ocorrem em várias condições difíceis de reproduzir em testes
- Abordagem *naive*
  - `grep gets *.c`
  - Obriga a que quem testa tenha de saber todas as funções vulneráveis
  - É preciso testar manualmente para cada função
  - Não distingue chamadas a funções de texto em *strings* ou comentários

```c
#include <stdio.h>
int main(int argc, char **argv)
{
    char buf[8]; // buffer for eight characters
    gets(buf); // read with gets (sensitive function!)
    printf("%s\n", buf); // print out data stored in buf
    return 0; // 0 as return value
}
```

# Ferramentas de Static Aplication Security Testing (SAST)

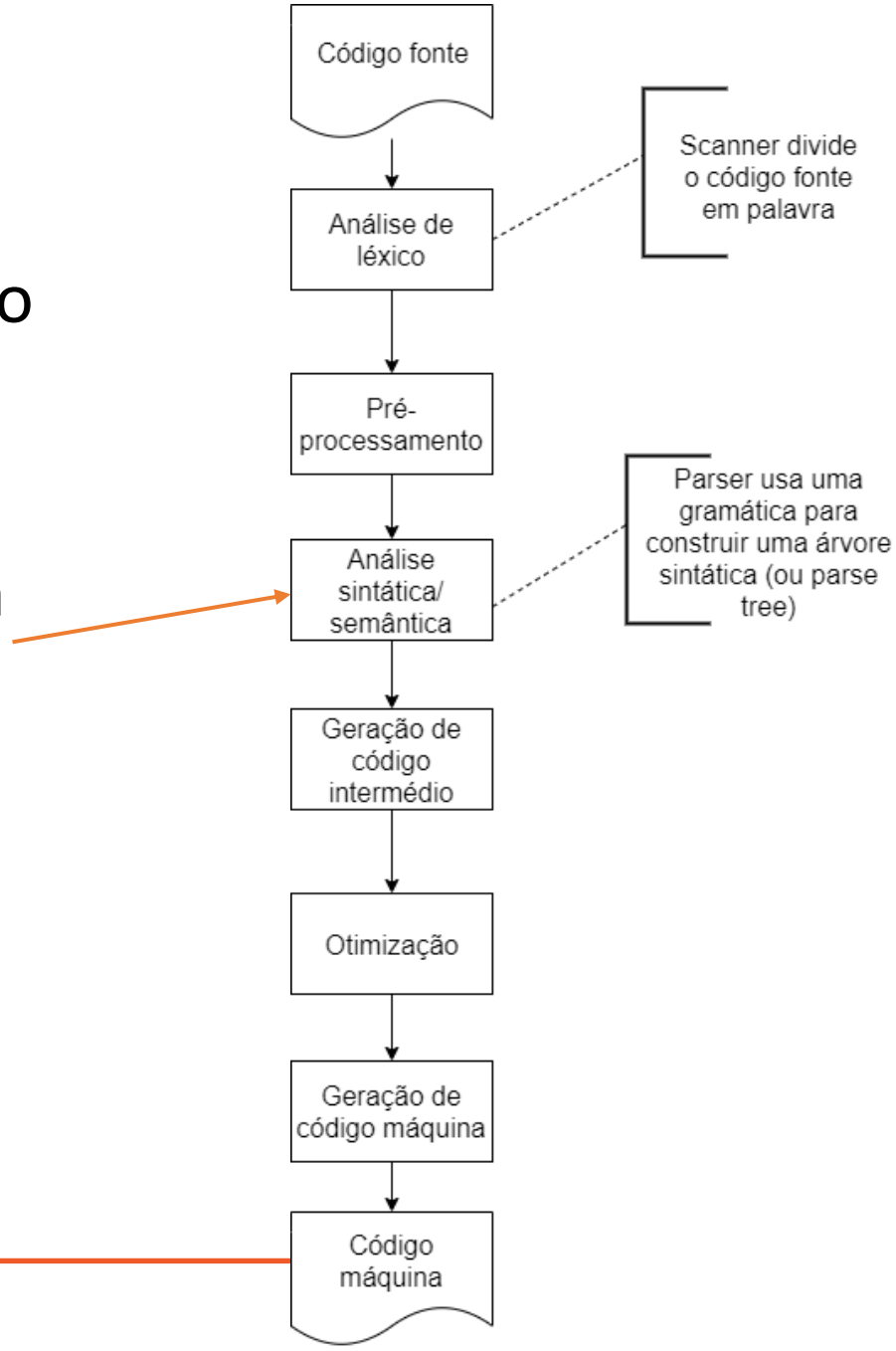
- O objectivo das ferramentas SAST é analisar código fonte e fornecer relatórios sobre vulnerabilidades encontradas, ajudando a decidir sobre:
  - Quais os elementos detetados que não são vulnerabilidades
  - Quais os elementos cujo risco é aceitável e por isso não são imediatamente abordados
  - Quais os elementos a serem mitigados e como o fazer
- Falsos negativos
  - Não detetam todas as situações
  - Limitados pela base de dados
  - Não é possível testar todas as condições em tempo útil
- Falsos positivos
  - Assinalam situações que não são vulnerabilidades

# Ferramentas de Static Aplication Security Testing (SAST)

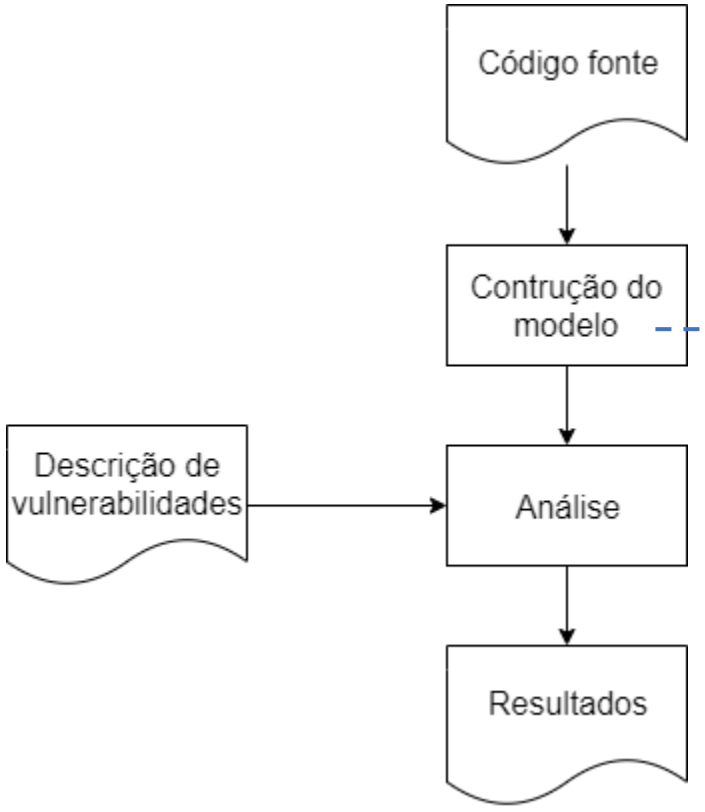
- Analisadores de léxico (simbólica)
  - Operam sobre as palavras geradas pelo scanner
  - Não confundem *`gets`*`how` com função `gets`
- Analisadores semânticos
  - Operam sobre a árvore sintática abstrata gerada pelo parser
  - Não confundem variável `gets` com chamada à função `gets`
- A análise semântica está organizada em
  - Verificação de tipos
  - Análise de controlo de fluxo
  - Análise de fluxo de dados

# Análise semântica

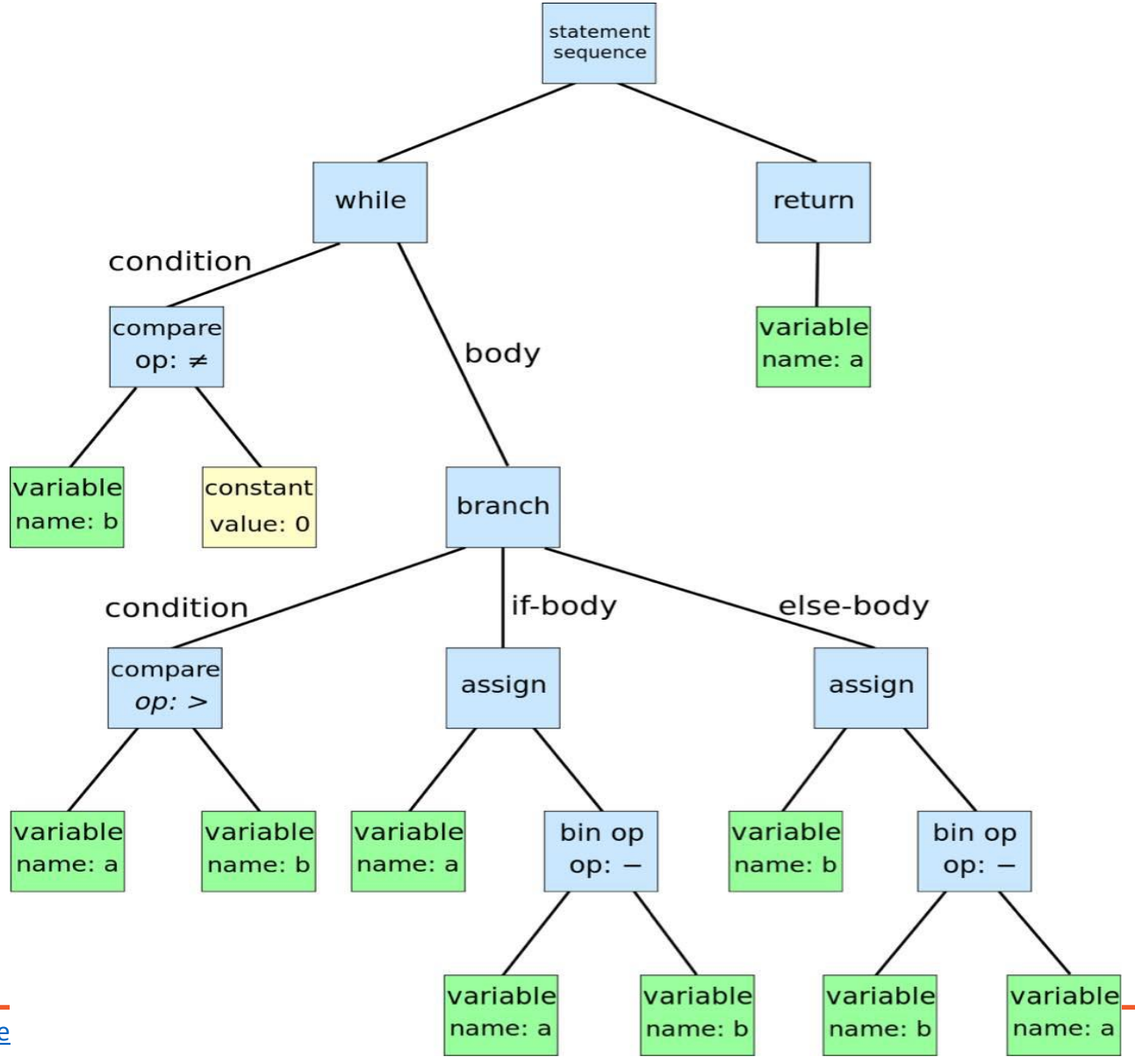
- Ferramentas de SAST têm semelhanças com um compilação de linguagens de programação
- A análise de léxico separa o texto nos *tokens* relevantes
- A terceira fase constrói a árvore sintática com base na gramática da linguagem



# Arquiteturas das Ferramentas – análise semântica




*exemplo*




https://en.wikipedia.org/wiki/Abstract_syntax_tree

# Exemplo de ferramenta para análise simbólica

- Exemplo
  - RATS: https://security.web.cern.ch/security/recommendations/en/codetools/rats.shtml
  - C, C++, Perl, PHP, Phyton
- Procurar por funções vulneráveis
- Usam base de dados com vulnerabilidades

| Função | Potencial vulnerabilidade | Solução | Risco |
|---|---|---|---|
| `access` | Pode levar a acesso a ficheiros indevidos em condições de race | Manipular descritores e não nomes simbólicos, se possível | Médio |
| `fread` | Pode levar a input com efeitos maliciosos | Verificar input | Baixo |
| `fscanf` | Pode resultar em *buffer overflow* | Usar especificadores de precisão | Elevado |

# Análise semântica

## Exemplos de análise semântica com verificação de tipo

- Erro de sinal
  - Inteiro com sinal é atribuído a valor sem sinal ou vice-versa
- Erro de truncamento
  - Inteiro representado com determinado número de bits é atribuído a uma variável com menos bits
- Algumas ferramentas possibilitam anotar o código com contratos que podem ser verificados
  - SAL – Standard Annotation Languague (Microsoft), C e C++

```
_checkReturn void *__cdecl malloc(__in size_t _Size);
```

  - JIF – Java + Information Flow (Cornell)

```
int {Alice → Bob} x;
```

# Análise semântica

## Controlo de fluxo

- Teste aos vários caminhos de execução à procura de erros

```
1   char * reserve_memory(int size) {
2       char *memory;
3       if (size > 0)
4           memory = (char*) malloc(size);
5       if (size == 1)
6           return NULL;
7       memory[0] = 0;
8       return memory;
9   }
```

- Vários caminhos possíveis <2,3,5,7,8>, <2,3,4,5,7,8> e <2,3,4,5,6>
- Constrói grafo de controlo de fluxo com base nas diferentes condições, ciclos e chamadas a funções

# Análise semântica

## Controlo de fluxo

- A análise de controlo de fluxo envolve um conjunto representativo de caminhos
  - Para cada caminho simula-se o percurso gerando alarmes quando relevante
  - Em geral não é possível testar com valores concretos mas com casos relevantes
    - `size=1, size>0, size<>1, size <=0`
- No exemplo há erros
  - 2,3,5,7,8 (array não iniciado)
  - 2,3,4,5,6 (memória não libertada)

```
1  char * reserve_memory(int size) {
2      char *memory;
3      if (size > 0)
4          memory = (char*) malloc(size);
5      if (size == 1)
6          return NULL;
7      memory[0] = 0;
8      return memory;
9  }
```

# Análise semântica

## Fluxo de dados

- Análise de comprometimento – verifica se funções sensíveis correm o risco de usar dados comprometidos ou privados
- As ferramentas que usam esta técnica marcam os dados como:
  - *tainted* – indicação de valor potencial comprometido; por exemplo **valores obtidos de entradas**, funções `scanf` ou `gets` em C
  - *untainted* – indicação de que o valor (por exemplo parâmetro de uma função) não pode estar comprometido; por exemplo *strings* de formatação da função `printf` em C.
- Exemplos de regras para propagação de comprometimento
  - Variável **a** está comprometida, **b=a** resulta no comprometimento de **b**
  - **`d = f(a)`** pode ou não levar ao comprometimento de **d** (a função **f** pode internamente chamar outra que realize o saneamento da entrada)

# NIST 500-268

- https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analysis
  - Publicado pela agência Norte-americana “National Institute of Standards and Technology”
- Obrigatoriamente as ferramentas têm de:
  - Identificar todas as classes de vulnerabilidades listadas no Anexo A
  - Reportar textualmente qualquer vulnerabilidades que tenha identificado
  - Para quaisquer vulnerabilidades não listadas no Anexo A, reportar a a classe usando um nome semanticamente equivalente
  - Para qualquer vulnerabilidades identificada, reportar pelo menos um local fornecendo o caminho do diretório, o nome do arquivo e o número da linha
  - Identificar vulnerabilidades apesar da complexidade de codificação listadas no Anexo B
  - Possuir uma taxa de falsos positivos aceitavelmente baixa.

# NIST 500-268 – exemplos do Anexo A

| Name | CWE ID | Description | Language(s) | Relevant Complexities |
|---|---|---|---|---|
| **Input Validation** | | | | |
| Basic XSS | 80 | Inadequately filtered input, allows a malicious script to be passed to a web application that in turn passes it to another client. | C,C++, Java, SPARK | taint, scope, address alias level, container, local control flow, loop structure, buffer address type |
| Resource Injection | 99 | Inadequately filtered input is used in an argument to a resource operation function. | C, C++, Java, SPARK | taint, scope, address alias level, container, local control flow, loop structure, buffer address type |
| OS Command Injection | 78 | Inadequately filtered input is used in an argument to a system operation execution function. | C, C++, Java, SPARK | taint, scope, address alias level, container, local control flow, loop structure, buffer address type |
| SQL Injection | 89 | Inadequately filtered input is used in an argument to a SQL command calling function. | C, C++, Java, SPARK | taint, scope, address alias level, container, local control flow, loop structure, buffer address type |

# NIST 500-268 – exemplos do Anexo B

| Complexity | Description | Enumeration |
|---|---|---|
| ... | | |
| Scope | scope of control flow related to weakness | local, within-file/inter-procedural, within-file/global, inter-file/inter-procedural, inter-file/global, inter-class |
| Taint | type of tainting to input data | argc/argv, environment variables, file or stdin, socket, process environment |

# Exemplo de SAST no contexto Web

- WAP – ferramenta de análise estática de código direcionada para aplicações web escritas em PHP
  - http://awap.sourceforge.net/
- Três entidades: entradas, funções sensíveis e funções de saneamento
  - Entradas: $_GET, $_POST
  - Funções sensíveis: mysql_query
  - Funções de saneamento: `mysql_real_escape_string` (faz o saneamento de *string* para que possam ser passadas à função `mysql_query`)
- Regras de propagação de comprometimento
  - Se o fluxo de dados parte de uma entrada e chega a uma função sensível **sem passar** por uma função de saneamento *=> vulnerabilidade*
  - Se o fluxo de dados parte de uma entrada e chega a uma função sensível **passando por uma função de saneamento adequada** *=> não existe vulnerabilidade*

# Caso Prático: Android

```
1  public class LeakageApp extends Activity{
2  private User user = null;
3  protected void onRestart(){
4    EditText usernameText =
       (EditText)findViewById(R.id.username);
5    EditText passwordText =
       (EditText)findViewById(R.id.pwdString);
6    String uname = usernameText.toString();
7    String pwd = passwordText.toString();
8    if(!uname.isEmpty() && !pwd.isEmpty())
9      this.user = new User(uname, pwd);
10 }
11 //Callback method in xml file
12 public void sendMessage(View view){
13   if(user == null) return;
14   Password pwd = user.getpwd();
15   String pwdString = pwd.getPassword();
16   String obfPwd = "";
17   //must track primitives:
18   for(char c : pwdString.toCharArray())
19     obfPwd += c + "_"; //String concat.
20
21   String message = "User: " +
22      user.getName() + " | Pwd: " + obfPwd;
23   SmsManager sms = SmsManager.getDefault()
24   sms.sendTextMessage("+44 020 7321 0905",
25     null, message, null, null);
26 }
```

- Exemplo de código Android com fuga de informação
- Linha 5: lê *password* de caixa de texto marcada como sendo para introduzir passwords
- Linha 24: envia a password por SMS

# SAST para Android: FlowDroid

- Ferramenta de análise estática para aplicações Android

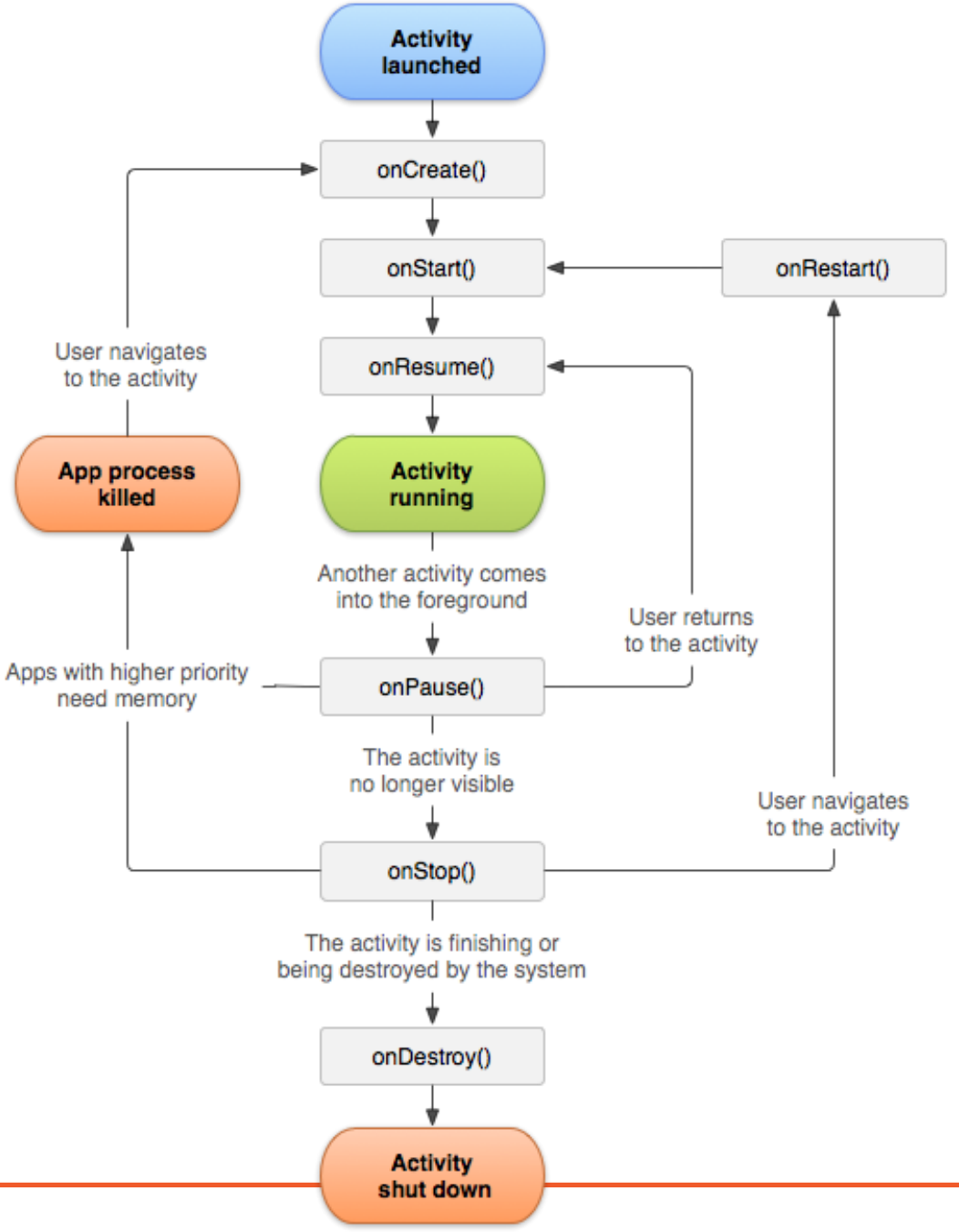
- Tem em conta todos os contexto da aplicação, objetos, campos e análise de comprometimento que considera o ciclo de vida das aplicações Android

- Analisa o código intermédio da aplicação (bytecode) e os ficheiros de configuração para encontrar potenciais fugas de informação

- Paper: https://www.bodden.de/pubs/far+14flowdroid.pdf
  - "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps", PLDI 2014

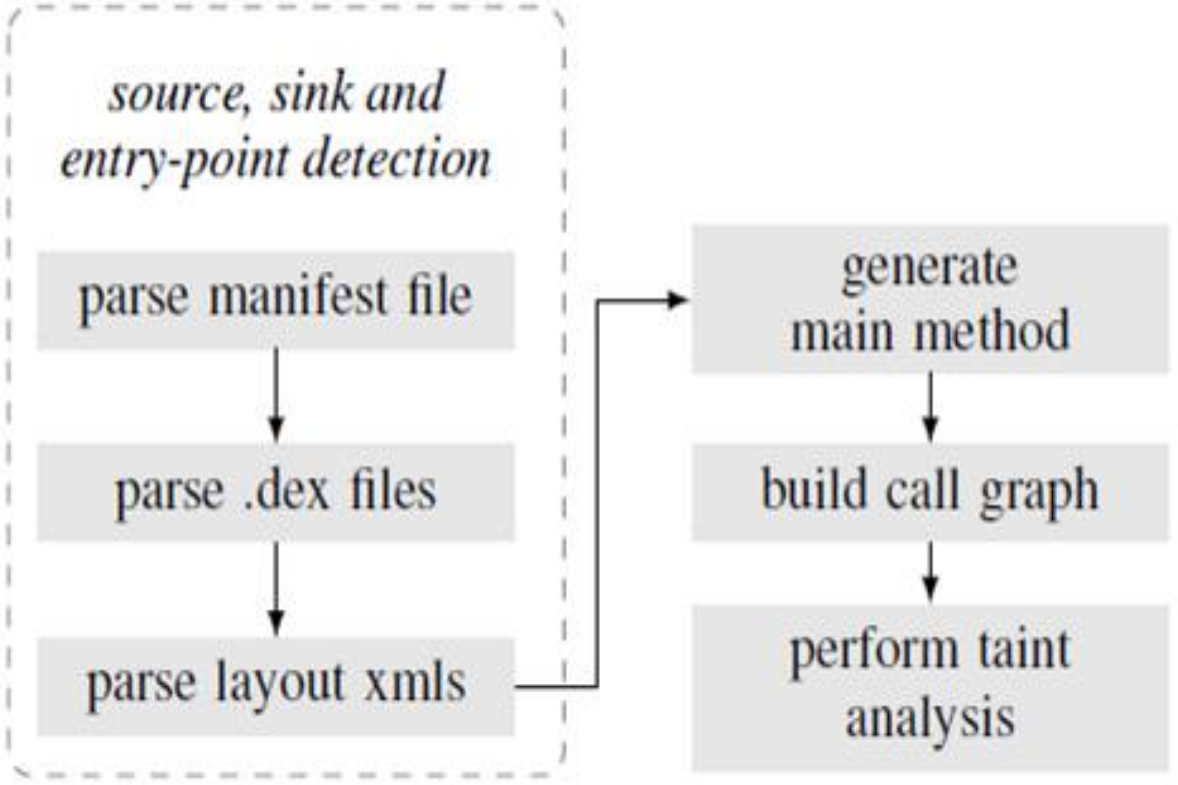
# Desafios do FlowDroid

- Precisa de modelo para o ciclo de vida das aplicações Android
  - Ciclo de vida guiado por *callbacks*, não há ponto de entrada único
- Não é possível determinar informação sensível apenas pelo código
  - Exemplo da caixa de texto que recebe password (`android:password`)
- As aplicações Java têm um modelo de execução não trivial, com despacho dinâmico de métodos e *aliasing*
- Taxa alta de falsos positivos e falsos negativos



# Funcionamento geral

- Procura os intervenientes no ciclo de vida da aplicação, bem como chamadas a **fontes** e **destinos** críticos
- Gera um método *main* a partir dos métodos identificados no ciclo de vida, o qual é usado para uma análise de controlo de fluxo
- No final são identificados os caminhos que levam informação de fontes a destinos críticos



# Exemplo

- Dois métodos fonte
  - Linha 9: chamada a `getCid()`, que retorna a identificação da célula de GSM.
  - Linha 11: chamada a `getLac()`, que retorna a o código da localização GSM
- Em conjunto, estas duas informações podem ser usadas para identificar a torre da célula GSM
- Linha 12: o código testa se o dispositivo está num determinado local em Berlin

```
1  void onCreate() {
2  TelephonyManager tm; GsmCellLocation loc;
3  // Get the location
4  tm = (TelephonyManager) getContext().
5      getSystemService
           (Context.TELEPHONY_SERVICE);
6  loc = (GsmCellLocation)
       tm.getCellLocation();
7
8  //source: cell-ID
9  int cellID = loc.getCid();
10 //source: location area code
11 int lac = location.getLac();
12 boolean berlin = (lac == 20228 && cellID
       == 62253);
13
14 String taint = "Berlin: " + berlin + " ("
       + cellID + " | " + lac + ")";
15 String f = this.getFilesDir() +
       "/mytaintedFile.txt";
16 //sink
17 FileUtils.stringToFile(f, taint);
18 //make file readable to everyone
19 Runtime.getRuntime().exec("chmod 666 "+f);
20 }
```

Listing 1: Android Location Leak Example

# Exemplo de *sources* e *sinks*

- <android.bluetooth.BluetoothAdapter: java.lang.String getAddress()> -> _SOURCE_
- <android.net.wifi.WifiInfo: java.lang.String getMacAddress()> -> _SOURCE_
- <android.telephony.gsm.GsmCellLocation: int getCid()> -> _SOURCE_
- <android.telephony.gsm.GsmCellLocation: int getLac()> -> _SOURCE_
- <org.apache.http.impl.client.DefaultHttpClient: org.apache.http.HttpResponse execute(org.apache.http.client.methods.HttpUriRequest)> -> _SINK_
- <android.telephony.SmsManager: void sendTextMessage(java.lang.String,java.lang.String,java.lang.String,android.app.PendingIntent,android.app.PendingIntent)> android.permission.SEND_SMS -> _SINK_
- <android.telephony.SmsManager: void sendMultipartTextMessage(java.lang.String,java.lang.String,java.util.ArrayList,java.util.ArrayList,java.util.ArrayList)> android.permission.SEND_SMS -> _SINK_

# FlowDroid e demo com DroidBench

- https://github.com/secure-software-engineering/FlowDroid

- Um conjunto de aplicações de teste, contendo 39 aplicações escritas propositadamente com fugas de dados sensíveis
  - https://github.com/secure-software-engineering/DroidBench

- Exemplo

```
java -jar soot-infoflow-cmd-jar-with-dependencies.jar -p
C:\Users\josem\AppData\Local\Android\Sdk\platforms\android-29\android.jar -s
FlowDroid\soot-infoflow-android\SourcesAndSinks.txt -a
DroidBench\apk\InterAppCommunication\SendSMS.apk
```

# Demo - resultados

- The ***sink*** staticinvoke Log.i(java.lang.String, java.lang.String)>("SendSMS: ", …) in method <org.cert.sendsms.Button1Listener: void onClick(android.view.View)> was called with values from the following ***sources***:
  - virtualinvoke $r5.<**android.telephony.TelephonyManager: java.lang.String getDeviceId()**>() in method <org.cert.sendsms.Button1Listener: void onClick(android.view.View)>
- The ***sink*** MainActivity.startActivityForResult(android.content.Intent,int) was called with values from the following ***sources***:
  - virtualinvoke $r5.<**android.telephony.TelephonyManager: java.lang.String getDeviceId()**>() in method <org.cert.sendsms.Button1Listener: void onClick(android.view.View)>