INTRO TO
JOERN: PATTERN-BASED VULNERABILITY MINING

Valerio Costamagna - @vaio_co
Marco Bartoli - @wsxarcher

HackInBo®
Spring 2017 Edition

AGENDA

- Introduzione:
  - Pattern-based vulnerability
  - Property graph
- Vulnerability mining con Joern:
  - Code property graph
  - Gremlin query language
- Spot the bug:
  - Esempi di query

PATTERN-BASED ?!?

- Bug in libssh2 trovato da Stefan Esser @ Syscan'13 utilizzando grep e la seguente regular expression

```
'ALLOC[A-Z0-9_]*\s*\([^,]*,[^;]*[*+-][^>][^;]*\)\s*;'
```

- Si puo' fare meglio di GREP?

- SI! in sintesi: CODE PROPERTY GRAPH !!!

PROPERTY GRAPH

- Multigrafi con nodi/archi etichettati con un set di proprieta' (chiave-valore)
  - nodo 4 ha le proprieta' {name: josh; age:32}
  - 4 -> created -> 3

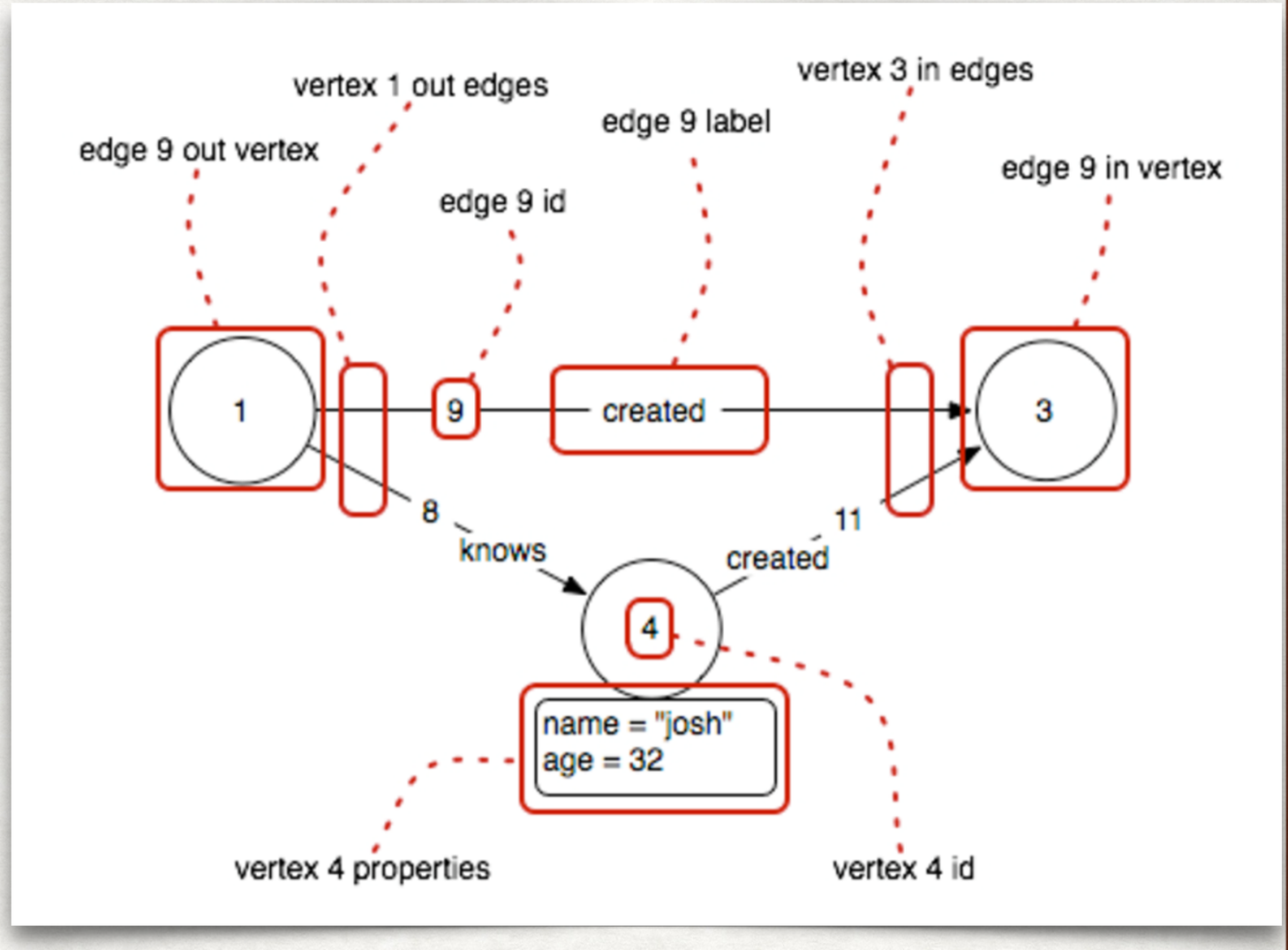


image from: https://github.com/tinkerpop/gremlin/wiki/Basic-Graph-Traversals

CODE PROPERTY GRAPH

FOR MINING VULNS

- Utilizzati per la prima volta da Fabian Yamaguchi per la ricerca di vulnerabilita' nel codice: CODE PROPERTY GRAPH
- Rappresentano diverse astrazioni del codice:
  - Abstract Syntax Tree (AST)
  - Control Flow Graph (CFG)
  - Program Dependence Graph (PDG) [*]

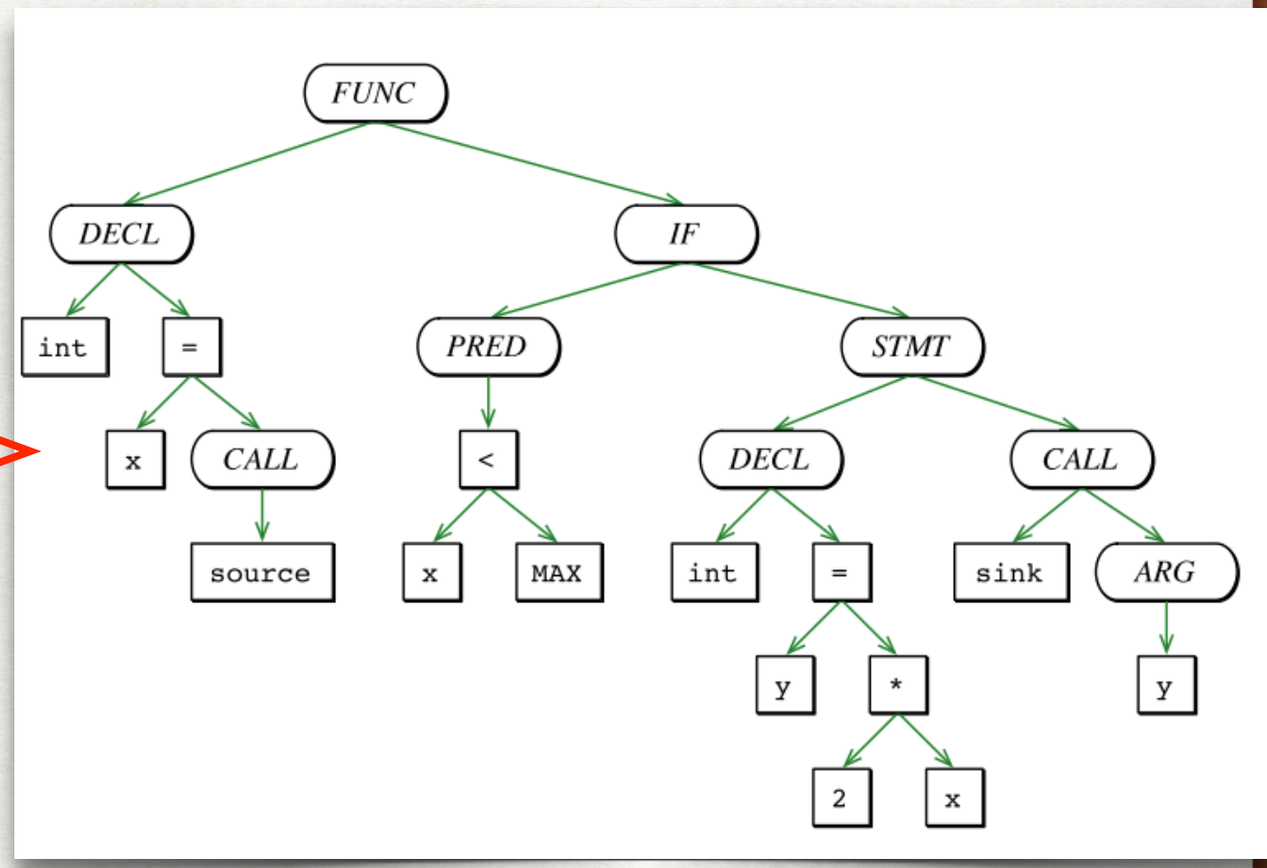
[*] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems, 9:319–349, 1987.

ABSTRACT SYNTAX TREE

AST

- Intermediate representation solitamente prodotta dal parser del compilatore
- Utilizza alberi per rappresentare *statements* ed *expressions* presenti all'interno del codice

```
void foo()                                  1
{                                           2
     int x = source();                      3
     if (x < MAX)                           4
     {                                      5
          int y = 2 * x;                    6
          sink(y);                          7
     }                                      8
}                                           9
```



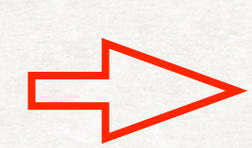
- I nodi di un AST rappresentano gli operatori (es: addizioni o assegnazioni)
- Le foglie rappresentano gli operandi (es: constanti o identificatori)

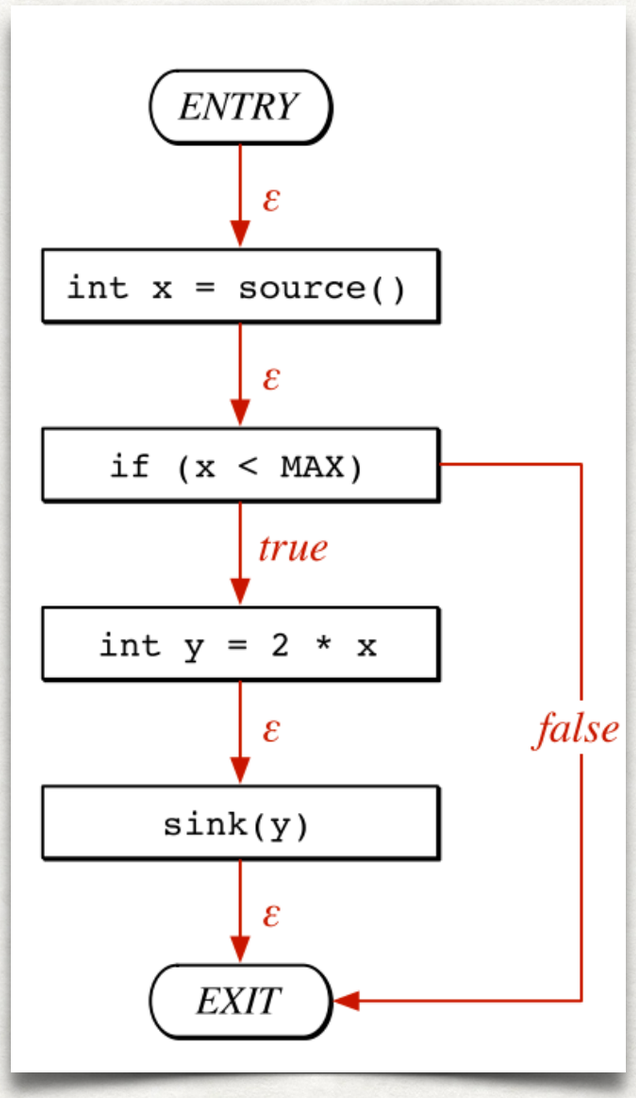
CONTROL FLOW GRAPH

CFG

- Esplicita l'ordine di esecuzione degli statements e le condizioni da rispettare per compiere un particolare *path*

```
1  void foo()
2  {
3      int x = source();
4      if (x < MAX)
5      {
6          int y = 2 * x;
7          sink(y);
8      }
9  }
```





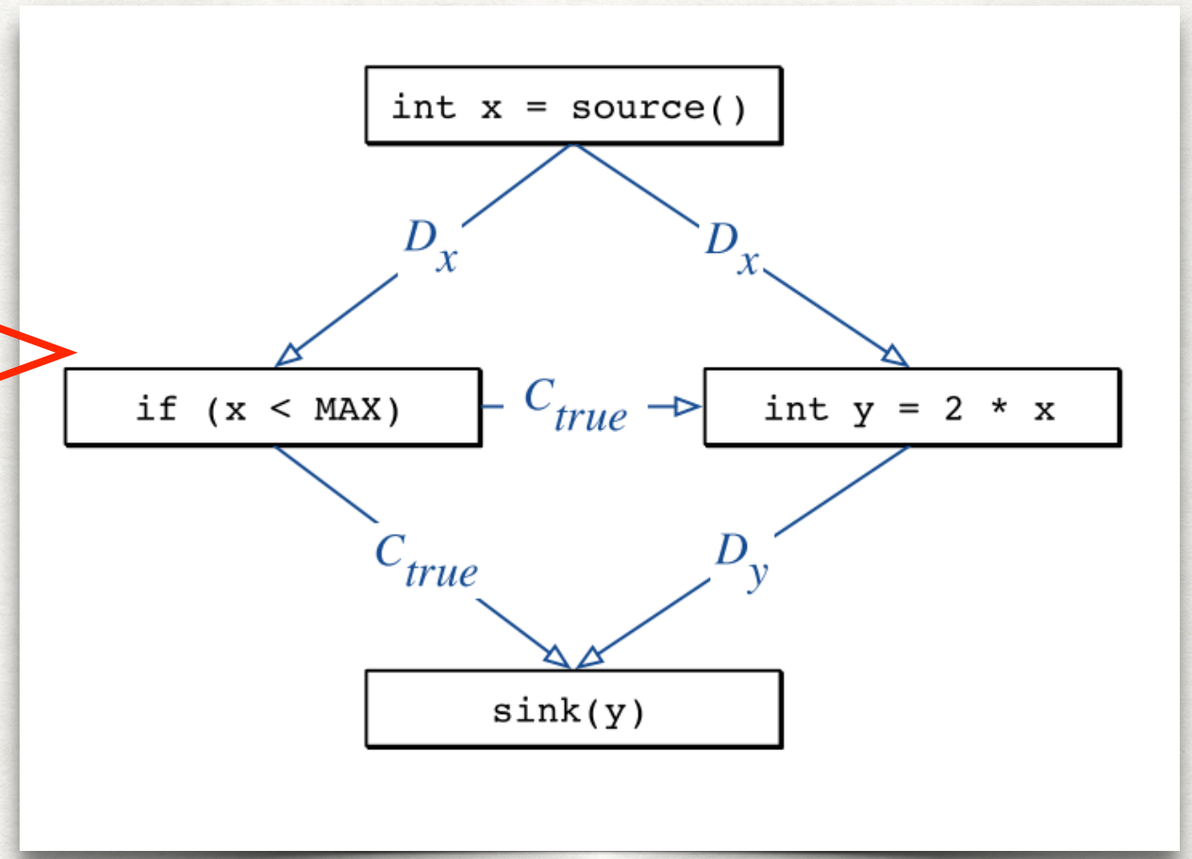
- Statements e predicati sono rappresentati dai nodi
- Direct edges rappresentano il trasferimento di controllo. diverse etichette: statements (epsilon), predicati (true/false)

PROGRAM DEPENDENCE GRAPH

PDG

- PDG esplicita dipendenze tra statements e predicati, permette la creazione dell'insieme di variabili *defined* and *used* (def/use) per ogni statement

```
void foo()                                  1
{                                           2
    int x = source();                       3
    if (x < MAX)                            4
    {                                       5
        int y = 2 * x;                      6
        sink(y);                            7
    }                                       8
}                                           9
```



- Due tipi di archi
  - Data dependency edges: rappresentano l'influenza di una var su un'altra
  - Control dependency edges: l'influenza dei predicati sul valore di una var

Joern

A Robust Code Analysis Platform for C/C++

- Creato da F. Yamaguchi, utilizza un graph database per memorizzare le diverse rappresentazioni del codice
- Permette di eseguire query sul DB per ricercare vulnerabilità
- Basato su *Apache TinkerPop* (computing framework per graph databases) e il linguaggio di graph traversal *Gremlin*

Note: attualmente il design di joern è stato modificato passando ad un nuovo progetto chiamato octopus piu' scalabile e basato su Tinkerpop3. Questa presentazione fa riferimento al vecchio sistema Joern e TP2.

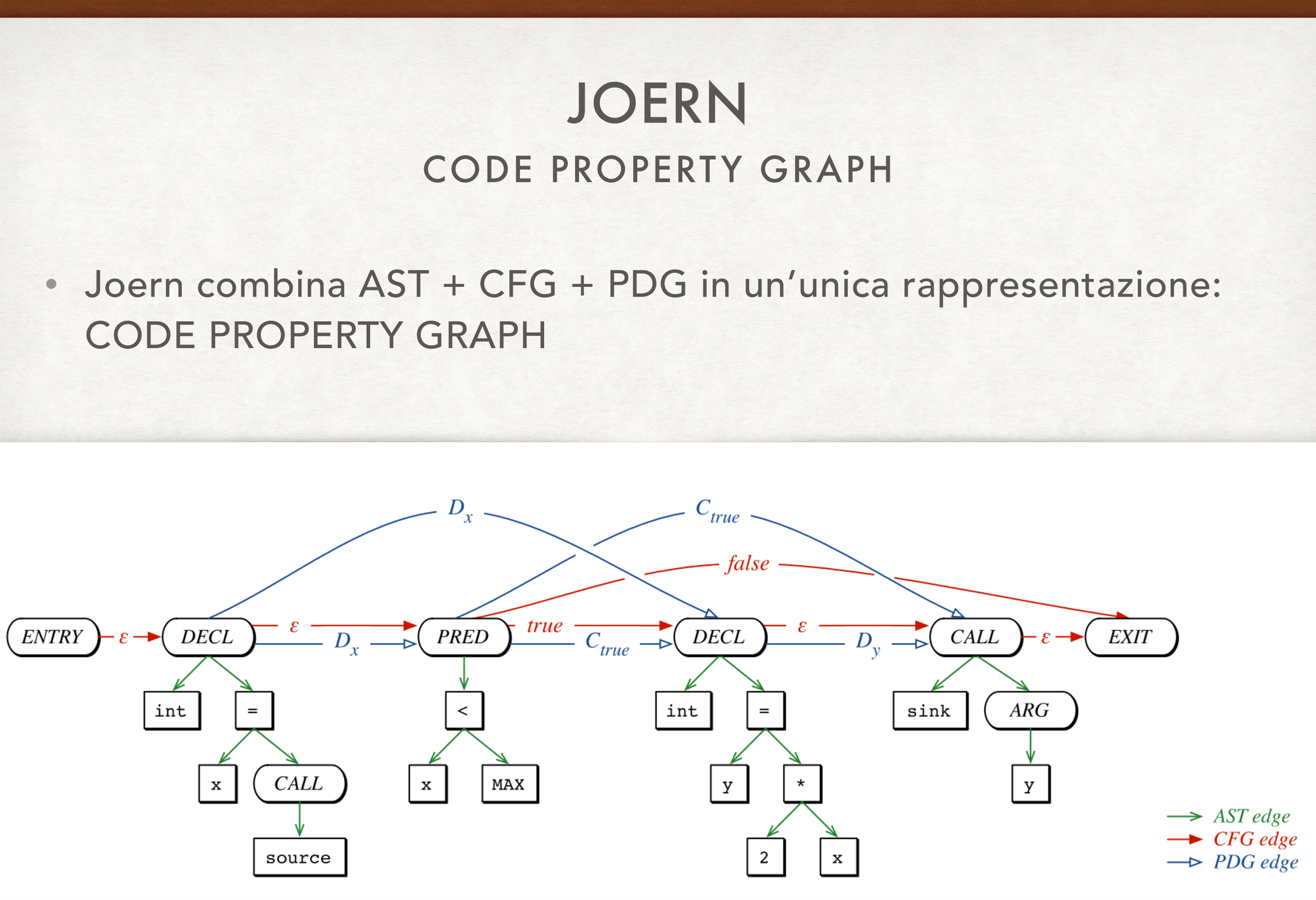
Vulnerability types	**Code representations**			
	AST	AST+PDG	AST+CFG	AST+CFG+PDG
Memory Disclosure				✓
Buffer Overflow		(✓)		✓
Resource Leaks			✓	✓
Design Errors				
Null Pointer Dereference				✓
Missing Permission Checks		✓		✓
Race Conditions				
Integer Overflows				✓
Division by Zero		✓		✓
Use After Free			(✓)	(✓)
Integer Type Issues				✓
Insecure Arguments	✓	✓	✓	✓

TABLE II: Coverage of different code representation for modeling vulnerability types.

JOERN

CODE PROPERTY GRAPH

- Joern combina AST + CFG + PDG in un'unica rappresentazione: CODE PROPERTY GRAPH



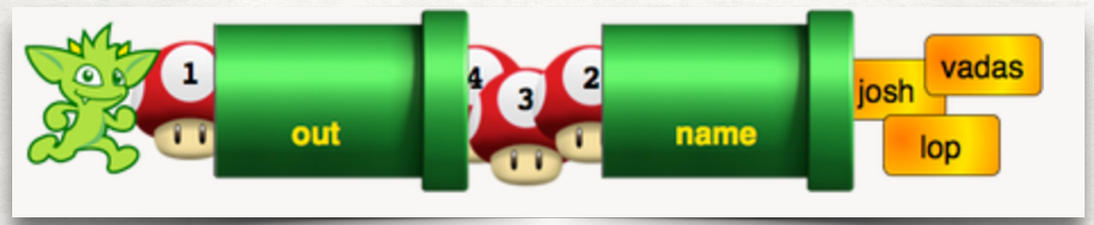
images courtesy of Fabian Yamaguchi

- Graph traversal language of TinkerPop

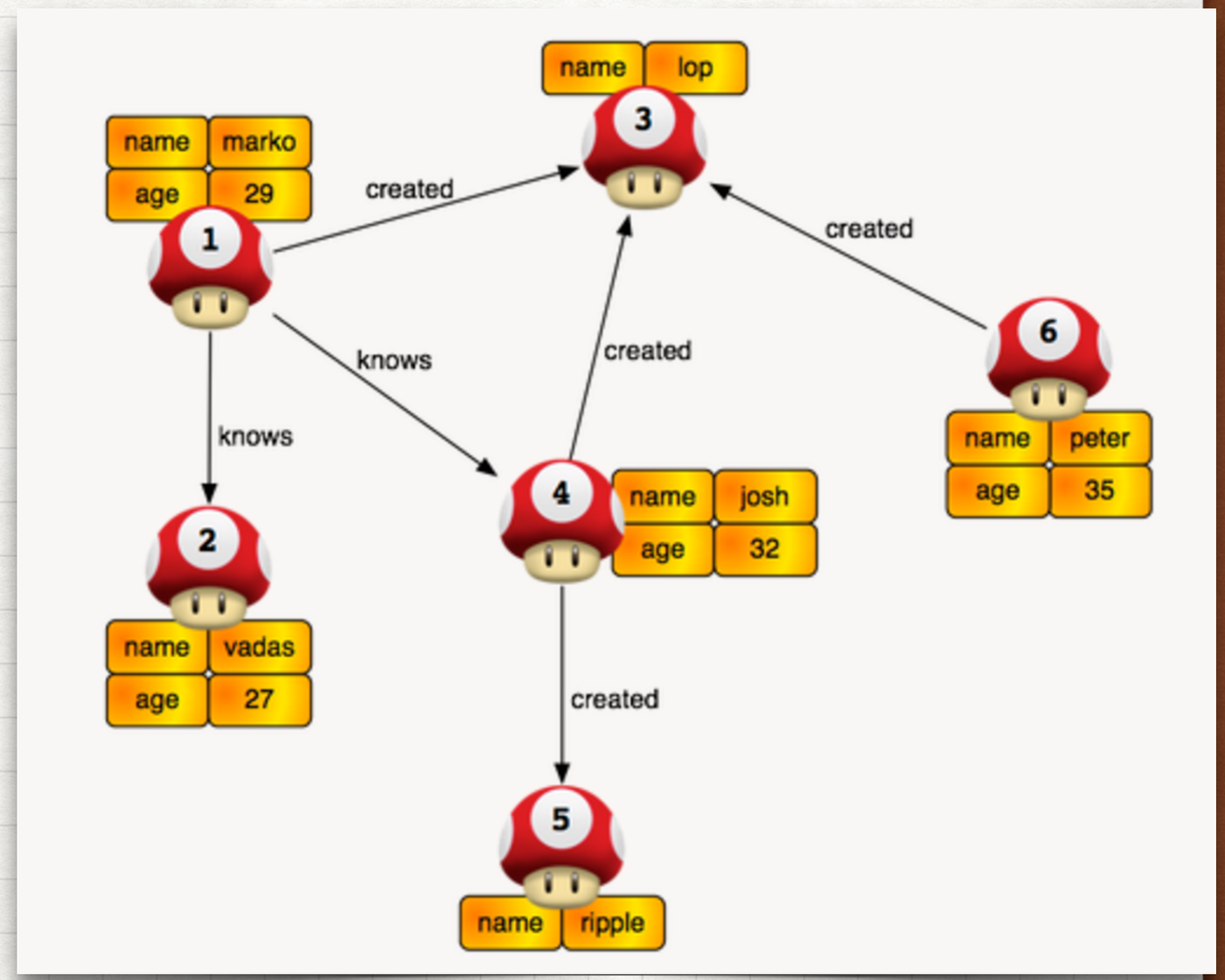
Gremlin is the graph traversal language of Apache TinkerPop. Gremlin is a functional, data-flow language that enables users to succinctly express complex traversals on (or queries of) their application's property graph. Every Gremlin traversal is composed of a sequence of (potentially nested) steps. A step performs an atomic operation on the data stream. Every step is either a *map*-step (transforming the objects in the stream), a *filter*-step (removing objects from the stream), or a *sideEffect*-step (computing statistics about the stream). The Gremlin step library extends on these 3-fundamental operations to provide users a rich collection of steps that they can compose in order to ask any conceivable question they may have of their data for Gremlin is Turing Complete.

- Gremlin è utilizzato per definire i “traversal” che percorrono il grafo
- Ogni traversal è composto da steps che definiscono le query da effettuare

GRAPH TRAVERSAL




```
gremlin> g.v(1).out.name
==>vadas
==>lop
==>josh
```




```
gremlin> g.v(1).out('knows').filter{it.age < 30}.name
==>vadas
```

HACKINBO® Spring 2017 Edition

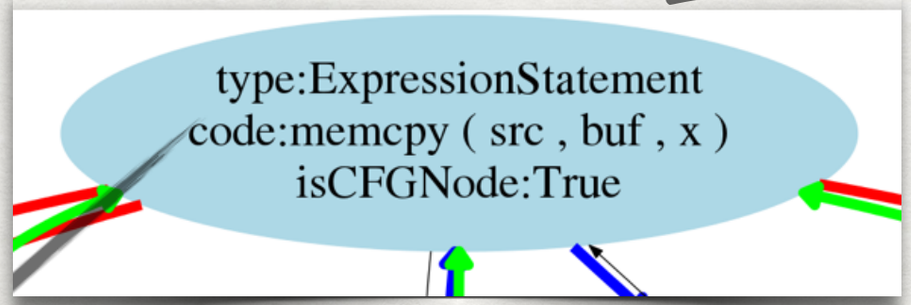
VULN CODE EXAMPLE

INTRA-PROCEDURE

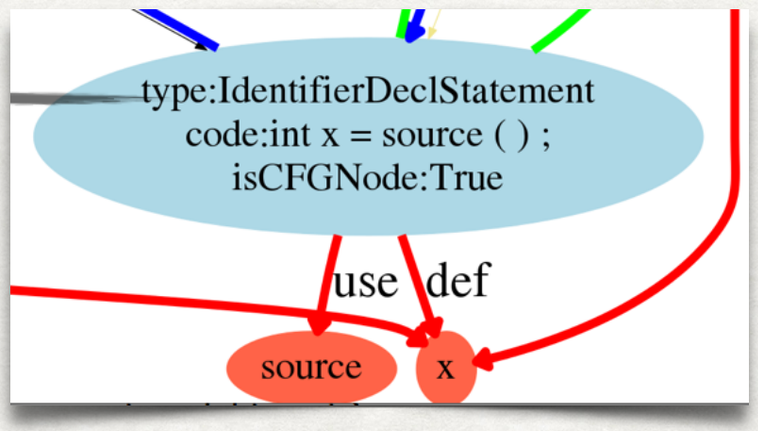
```
44 void vuln()
45 {
46     unsigned int x = source();
47     char* src; //attacker-data
48     char* buf = malloc(x+1);
49     memcpy(src,buf,x);
50 }
51 void not_vuln(){
52     unsigned int x = source();
53     char* src; //attacker-data
54     if (x > MAX )
55         x = MAX;
56     char* buf = malloc(x+1);
57     memcpy(src,buf,x);
58 }
59 void source()
60 {
61     //attacker-controllable
62     return readInput();
63 }
```

- source() ritorna un valore controllabile dall'attaccante (es: dati letti da file, network, etc...)
- problem: intra-procedure data flow tra source -> memcpy

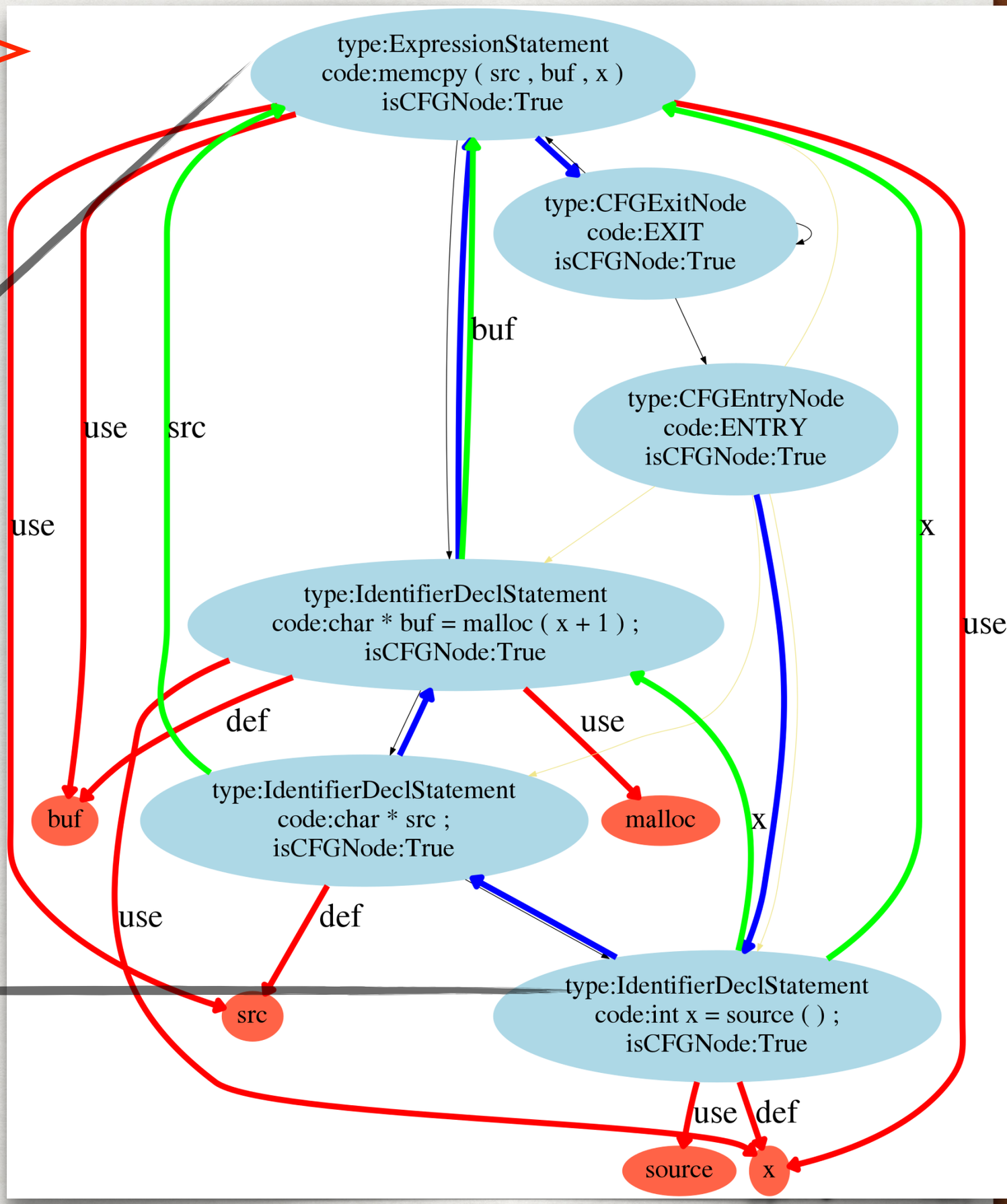
```
44 void vuln()
45 {
46     unsigned int x = source();
47     char* src; //attacker-data
48     char* buf = malloc(x+1);
49     memcpy(src,buf,x);
50 }
```



Nodo memcpy



Nodo source



QUERY EXAMPLE - I

Lista di tutti i nodi con *type:CallExpression* e *code:memcpy*

```
joern> queryNodeIndex('type:CallExpression AND code:*memcpy*')
(n56 {childNum:"0",code:"memcpy ( src , buf , x )",functionId:50,type:"CallExpression"})
(n107 {childNum:"0",code:"memcpy ( src , buf , x )",functionId:101,type:"CallExpression"})
```

Visualizzare uno specifico nodo dato il suo *Id*

```
joern> g.v(50)
(n50 {location:"6:0:85:204",name:"vuln",type:"Function"})
```

QUERY EXAMPLE - II

Lista dei call-site dove è presente un data-flow tra source e memcpy:

```
joern> getCallsTo("source").statements().out("REACHES").filter{ it.code.matches(".*memcpy.*")  }.locations()
[u'memcpy ( src , buf , x )', u'vuln', u'                                                    /intra.c']
[u'memcpy ( src , buf , x )', u'not_vuln',                                                      /intra.c']
```

Tra i risultati precedenti è presente anche la funzione “not_vuln”, come richiesto dalla query, ma…

… possiamo verificare se sul percorso la variabile viene utilizzata in una condition…

```
44 void vuln()
45 {
46     unsigned int x = source();
47     char* src; //attacker-data
48     char* buf = malloc(x+1);
49     memcpy(src,buf,x);
50 }
```

```
51 void not_vuln(){
52     unsigned int x = source();
53     char* src; //attacker-data
54     if (x > MAX )
55         x = MAX;
56     char* buf = malloc(x+1);
57     memcpy(src,buf,x);
58 }
```

VULN CODE EXAMPLE

INTER-PROCEDURE

```
26 void vuln()
27 {
28     unsigned int x = source();
29     woo(x);
30 }
31 void not_vuln(){
32     unsigned x = source();
33     if (x > MAX )
34         x = MAX;
35     woo(x);
36 }
37
38 void woo(unsigned int x){
39     int *buf = malloc(sizeof(int) * x);
40     //your favourite bug in sink..
41     sink(buf,x);
42 }
```

- source() ritorna un valore controllabile dall'attaccante (es: dati letti da file, network, etc…)
- woo() contiene la chiamata alla funzione sink() di cui vogliamo controllare gli argomenti
- tainting problem: inter-procedure data flow tra source -> sink

LET'S SPOT THE BUG

taint-style vulnerability pattern: flussi da una sorgente controllabile dall'attaccante ad un sensitive sink senza subire alcuna validation

```
getCallsTo(sink)
.taintedArgs(
        [arg1Source,..., argnSource]
)
.unchecked(
        [arg1Sanitizer, ... argnSanitizer]
)
```

What the source looks like ↓

What sanitization looks like ↓

image courtesy of Fabian Yamaguchi

QUERY VIVISECTION

```
10 ANY = { [1]._() }
11 arg2 = { sourceMatches('.*source.*') }
12 check = { it, sym -> it.code.contains(sym)  }
13
14 getCallsTo("sink")
15 .taintedArgs([ANY, arg2])
16 .unchecked([null, check])
17
```

- *getCallsTo(“sink”)* - Ottiene tutte le chiamate (call-site) a sink(…)
- *taintedArgs([ANY, arg2])* - Per ogni call-site, restituisce gli argomenti che rispettano la closure (cioè quelli che sono prodotti da una funzione “.*source.*”)
- *unchecked([null, check])* - Per ogni argomento restituito dallo step precedente, filtra solo quelli che rispettano la closure (cioè quelli che non subiscono nessuna validazione)
- *Note: per dettagli su come è implementata la ricerca inter-procedure consultare le references*

LET'S SPOT THE BUG

```
26 void vuln()
27 {
28     unsigned int x = source();
29     woo(x);
30 }
```

```
31 void not_vuln(){
32     unsigned x = source();
33     if (x > MAX )
34         x = MAX;
35     woo(x);
36 }
```

```
38 void woo(unsigned int x){
39     int *buf = malloc(sizeof(int) * x);
40     //your favourite bug in sink..
41     sink(buf,x);
42 }
```

```
10 ANY = { [1]._() }
11 arg2 = { sourceMatches('.*source.*') }
12 check = { it, sym -> it.code.contains(sym)  }
13
14 getCallsTo("sink")
15 .taintedArgs([ANY, arg2])
16 .unchecked([null, check])
17
```

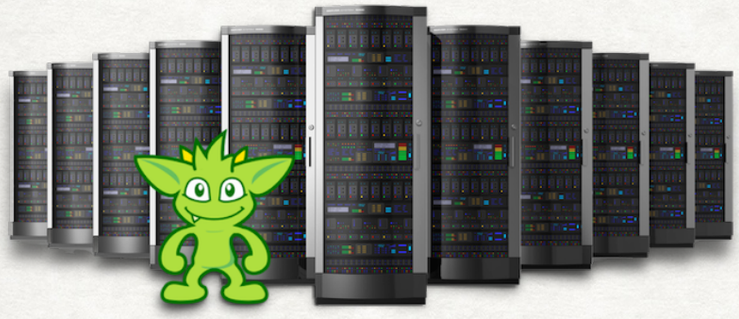
```
sid@mart                                  0$ python chunk.py
got ids: 1
[[u'int x = source ( ) ;', u'vuln', u'/ho                                  TS/hackinbo/main.c']]
```

HACKINBO Spring 2017 Edition

USE CASE: ANDROID

- framework core scritto in C/C++
- lib esterne C/C++
- Binder (IPC)
- targets:
  - system services comunicano via Binder
  - Java Native Interface (JNI)

HARDWARE



- i7 4.2GHz
- 32 gb RAM
- 500 gb SSD

Android 7.1.2 - framework e external

Android	# malloc	# memcpy	#readInt32	# f()
externals	5010	9351	1293	471.164
framework	341	865	1283	41.570
	5351	10216	2576	512.734

MINING IN THE PAST

risultati di un semplice inter-procedural traversal: data-flow tra *readInt32* e altre funzioni interessanti per mem corruption.

```
marte% time python mallocUnchecked.py
got ids: 399
size_t          = data . readInt32 ( ) ;                              .cpp
size_t totalSize = data . readInt32 ( ) ;        BnCrypto :: onTransact        ICrypto.cpp
uint32_t        = data . readInt32 ( ) ;                                       .cpp
uint32_t        = data . readInt32 ( ) ;                                       .cpp
size_t size = data . readInt32 ( ) ;             BnHDCP :: onTransact          IHDCP.cpp
size_t size = data . readInt32 ( ) ;             BnHDCP :: onTransact          IHDCP.cpp
python mallocUnchecked.py  0,04s user 0,02s system 2% cpu 2,429 total
```






0DAY

0DAY

CVE-2015-3834

CVE-2015-6612

JOERN MEETS ANDROID

CVE-2017-???

P2 Bug stackoverflow in [redacted] mr...@google.com Assigned [redacted] Apr 20, 2017 12:23AM

- query intra-procedure:

trovare i flussi da source (*getArrayLength*) a sink (*memcpy*) che non passano attraverso validations o che incontrano solo ‘weak’ validations (come “==“)

```
FUNCTION_NAME(
        JNIEnv *env, jclass, jint, jint , jobject )  {

    JNIHelper helper(env);
    [...]
    TYPE AParams;
    memset(&AParams, 0, sizeof(AParams));
    [...]
    AParams.num_len = helper.getArrayLength(user_array);
    if (AParams.num_len == 0) {
        return false;
    }
    for (int i = 0; i < AParams.num_len; i++) {
        [...]
        memcpy(AParams.buf[i].VAR, VAR, sizeof(VAR));
```

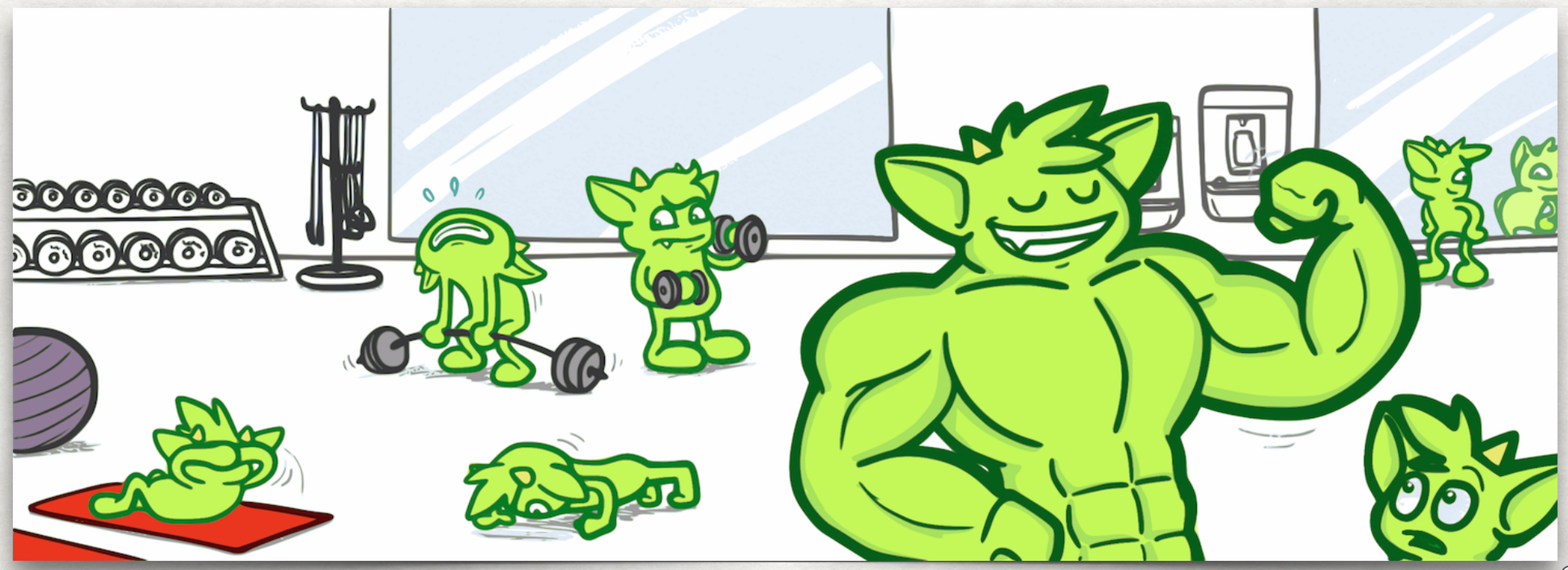
buf ha dimensione fissa

LIMITATIONS

- Joern costruisce i grafi staticamente
- Limiti principali:
  - Chiamate dinamiche:
    - puntatori a funzione, virtual call, GetProcAddress/dlsym
  - Shared memory (multi-threading)
- Attualmente il supporto C++ è limitato, per esempio:
  - Overriding e templates non sono gestiti completamente

PIMP YOUR GREMLIN

- Gremlin è estensibile utilizzando Groovy (linguaggio di scripting della JVM)
- Si possono scrivere *closure* (concatenabili) con una propria business logic per costruire query più complesse



CONCLUSION

- Joern/Octopus è molto di più (code clustering binary analysis, etc)..

```
sid@marte:~$ joern-
joern-apiembedder       joern-console       joern-knn           joern-plot-ast          joern-stmt-embedder
joern-ast2features      joern-demux         joern-list-files    joern-plot-proggraph    joern-stream-apiembedder
joern-astlabel          joern-edge          joern-list-funcs    joern-plot-slice        joern-tag
joern-cluster           joern-editor        joern-location      joern-regex             joern-transform
joern-code              joern-hide          joern-lookup        joern-slice             joern-unhide
```

- … ma non è (ancora?) una *0day machine*

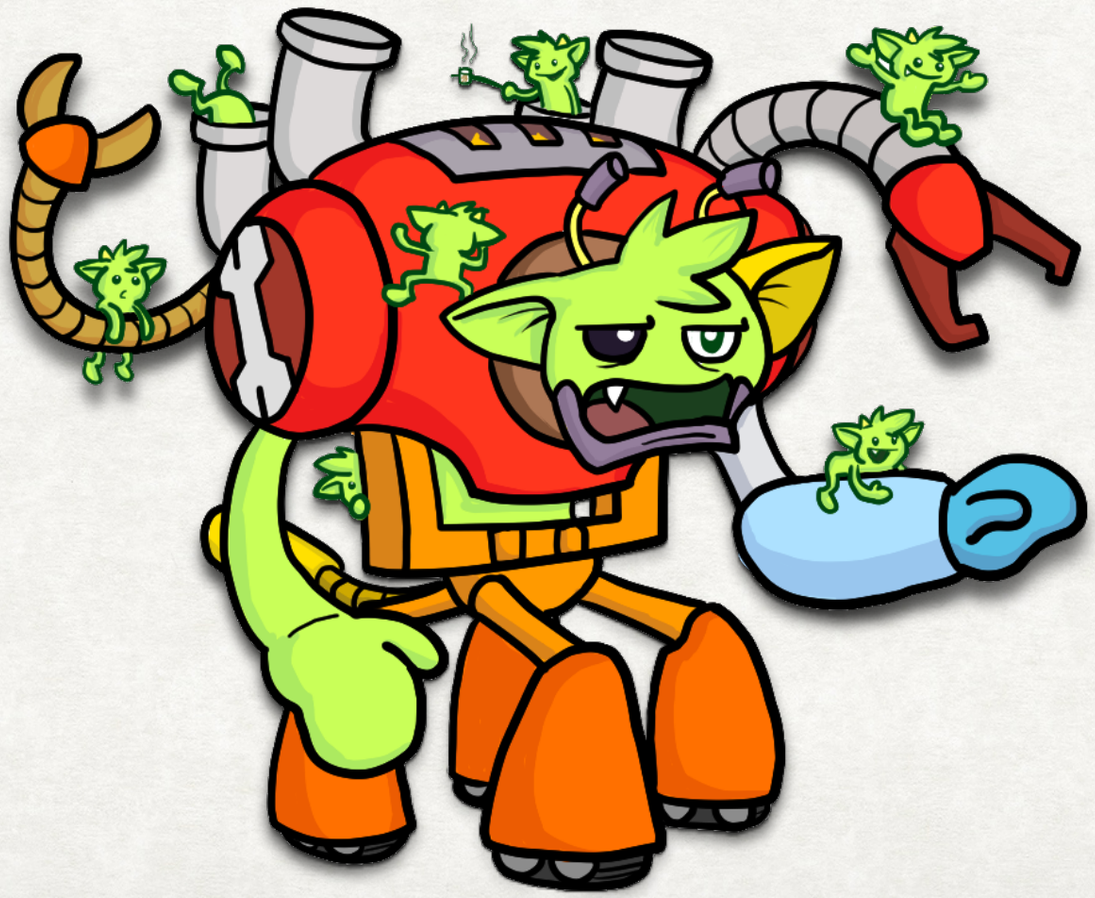
- Permette di ridurre drasticamente il tempo-uomo per la ricerca di vulnerabilità e può essere utilizzato in combinazione con altri approcci (dynamic analysis, fuzzing, symbolic execution, etc) per ottenere un'analisi aumentata.

- more code @ roptors.re

happy bug hunting!!!

GRAZIE!!!!

DOMANDE ??



REFERENCES

- http://www.mlsec.org/joern/docs.shtml
- https://github.com/octopus-platform/joern/
- Generalized Vulnerability Extrapolation using Abstract Syntax Trees. F. Yamaguchi, M. Lottmann, and K. Rieck. 28th Annual Computer Security Applications Conference (ACSAC'12). Outstanding Paper Award.
- Modeling and Discovering Vulnerabilities with Code Property Graphs. F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. 35th IEEE Symposium on Security and Privacy (S&P'14)
- Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. 36th IEEE Symposium on Security and Privacy (S&P'15)
- VCCFinder: Finding Pot. Vulnerabilities in Open-Source Projects to Assist Code Audits. H. Perl, D. Arp, S. Dechand, F. Yamaguchi, S. Fahl, Y. Acar, K. Rieck, and M. Smith. 22nd ACM Conference on Computer and Communications Security (CCS‟15)