

Workshop

Finding **STRANGER THINGS** *in Code*



Suchakra Sharma

Staff Scientist, ShiftLeft Inc.

March 5, 2020
Pasadena, CA



Let's Prep First

- *Clone Workshop Repo*
 - <https://github.com/tuxology/joern-workshop>
 - `apt install source-highlight graphviz unzip`
- *Download **joern-cli.zip** and extract it in the workshop directory*
 - <https://github.com/ShiftLeftSecurity/joern/releases>
 - `unzip joern-cli.zip`
- *Download **VLC v3.0.8** source and extract in workshop directory*
 - <http://get.videolan.org/vlc/3.0.8/vlc-3.0.8.tar.xz>
 - `tar -xvf vlc-3.0.8.tar.xz`



Suchakra Sharma

Staff Scientist, ShiftLeft Inc.

Github: [tuxology](#)

Twitter: [@tuxology](#)

Email: suchakra@shiftleft.io

PhD, École Polytechnique de Montréal

*Loves systems, code analysis,
performance analysis, hardware
tracing, samosas and poutine!*

PepTalk

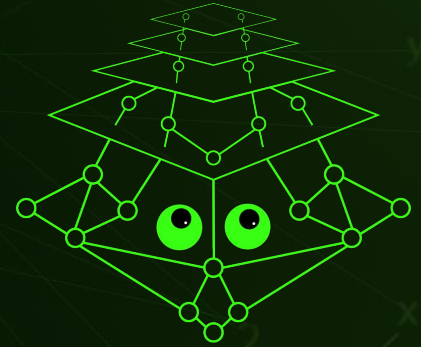


Why are you here?

- **You want to,**
 - Hack and secure your applications
 - Learn how programming languages work
 - HUnt bugs and gain insights about your code
- **Have questions like,**
 - What is static code analysis?
 - How do I do it interactively?
 - What is inside this mammoth code base?
 - What is even code? 🗨️

Pre-Workshop Poll

- ***I have used or know about,***
 - Interactive debuggers (GDB, rr)
 - SAST tools like Veracode/Coverity/FindSecBugs
 - Radare/PEDA/IDA Pro for security analysis
- ***I usually,***
 - Use Github to search through my code
 - Exclusively use an IDE/Editor to sift through my code
 - Grep through my source code looking for weird strings
 - Don't care about code



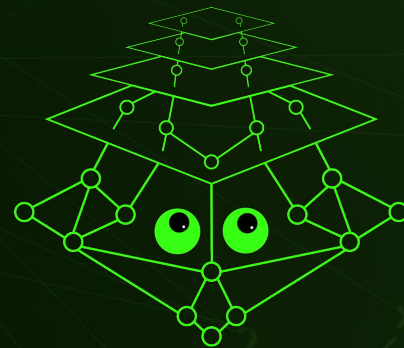
JOERN

Interactive Code Analysis

"Each program is its own universe, and hacking is about exploring, documenting and exploiting its rules"

~ Fabian

- Debugging goes hand in hand with running code
 - AddressSanitizer, ThreadSanitizer, profilers, linters
- Many tools run, and then give results but this approach flips the table - we give the tool to ask questions about the code
- It's like play-pause debugging, but for static analysis
 - IDA Pro, Radare, PEDA etc.



JOERN



Programming Languages

What is even *code*?

```
int y = x + 50;
```

sink ARG

y

2

x

*

y

=

int

- DECL

STMT

What is even *code*?

```
int y = x + 50;
```

INTEGER ID (y) EQUAL ID (x)

ADD CONST (50) SEMICOLON

Lexical Analysis

Tokens

What is even *code*?

```
int y = x + 50;
```

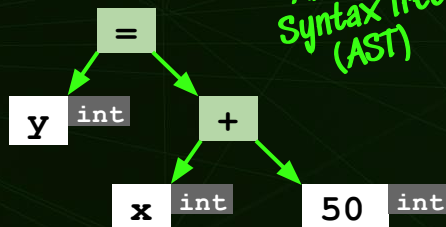
Tokens

INTEGER ID (y) EQUAL ID (x)
ADD CONST (50) SEMICOLON

Lexical Analysis



*Abstract
Syntax Tree
(AST)*



Syntactic & Semantic Analysis

What is even *code*?

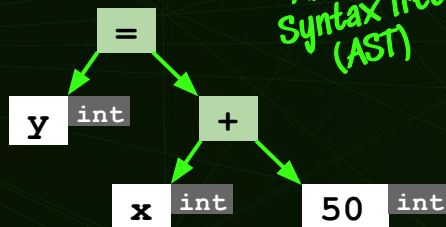
```
int y = x + 50;
```

Tokens

INTEGER ID (y) EQUAL ID (x)
ADD CONST (50) SEMICOLON

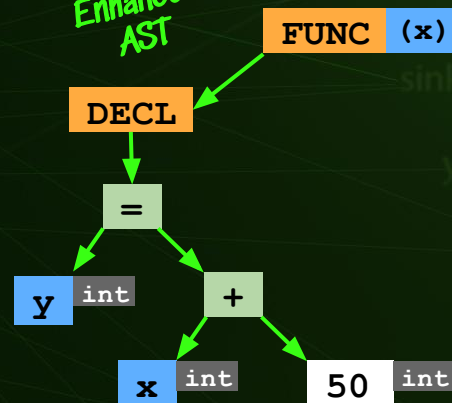
Lexical Analysis

Abstract
Syntax Tree
(AST)



Syntactic & Semantic Analysis

Enhanced
AST



What is even *code*?

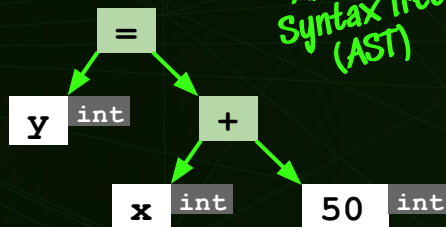
```
int y = x + 50;
```

Tokens

INTEGER ID (y) EQUAL ID (x)
ADD CONST (50) SEMICOLON

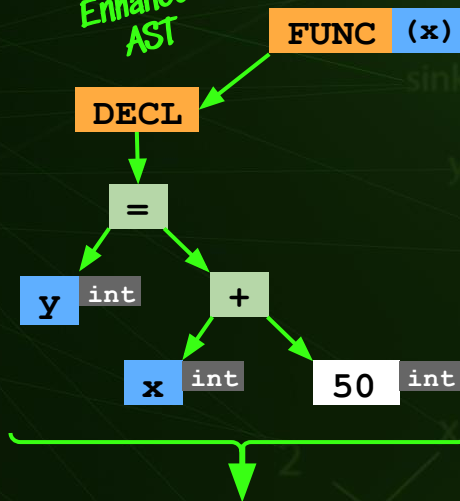
Lexical Analysis

Abstract
Syntax Tree
(AST)

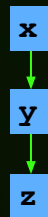


Syntactic & Semantic Analysis

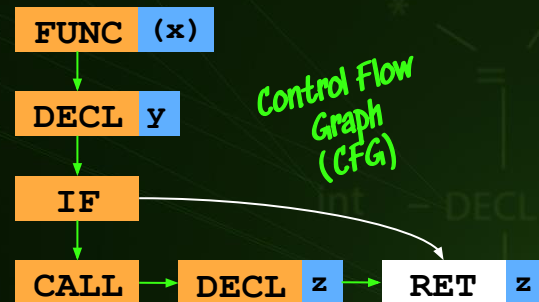
Enhanced
AST



Program
Dependence
Graph
(PDG)



Control Flow
Graph
(CFG)



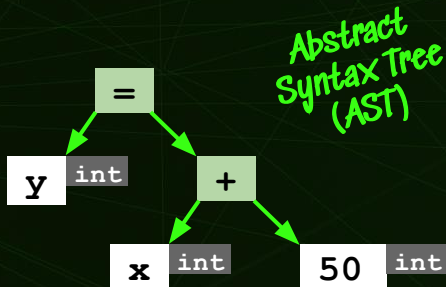
What is even *code*?

```
int y = x + 50;
```

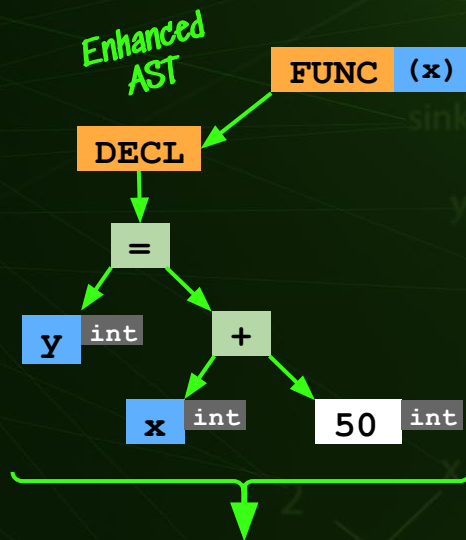
Tokens

INTEGER ID (y) EQUAL ID (x)
ADD CONST (50) SEMICOLON

Lexical Analysis



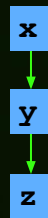
Syntactic & Semantic Analysis



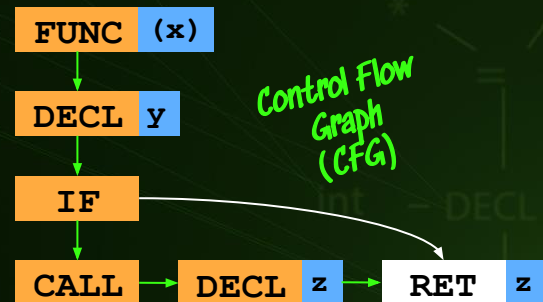
Optimizations
+
Register Alloc
+
Machine Code

```
10 = 50
11 = x + 10
y = 11
```

Translation



Program
Dependence
Graph
(PDG)



Building Blocks of Code

```
import org.springframework.web.bind.annotation.RestController;

@RestController
public class PatientController {

    private static Logger log =
        LoggerFactory.getLogger(PatientController.class);

    ...

    @RequestMapping(value = "/patients", method = RequestMethod.GET)
    public Iterable<Patient> getPatient(Int id) {
        Patient pat = patientRepository.findById(id);

        if (pat != null) {
            log.info("First Patient is {}", pat.toString());
        }

        return patientRepository.findAll();
    }
}
```

Building Blocks of Code

```
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController
```

```
public class PatientController {
```

```
    private static Logger log =
```

```
        LoggerFactory.getLogger(PatientController.class);
```

```
    ...
```

```
    @RequestMapping(value = "/patients", method = RequestMethod.GET)
```

```
    public Iterable<Patient> getPatient(Int id) {
```

```
        Patient pat = patientRepository.findById(id);
```

```
        if (pat != null) {
```

```
            log.info("First Patient is {}", pat.toString());
```

```
        }
```

```
        return patientRepository.findAll();
```

```
    }
```

Building Blocks of Code

```
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController  
public class PatientController {  
  
    private static Logger log =  
        LoggerFactory.getLogger(PatientController.class);  
  
    ...  
}
```

Class / Type

Member Variable

Package / Namespace

Annotation

Local Variable

Method Parameter

```
@RequestMapping(value = "/patients", method = RequestMethod.GET)  
public Iterable<Patient> getPatient(Int id) {  
    Patient pat = patientRepository.findById(id);  
  
    if (pat != null) {  
        log.info("First Patient is {}", pat.toString());  
    }  
  
    return patientRepository.findAll();  
}
```

Method Definition

Method Block

Method Instance

Literal

Method Return

Building Blocks of Code

PieClass

Defines

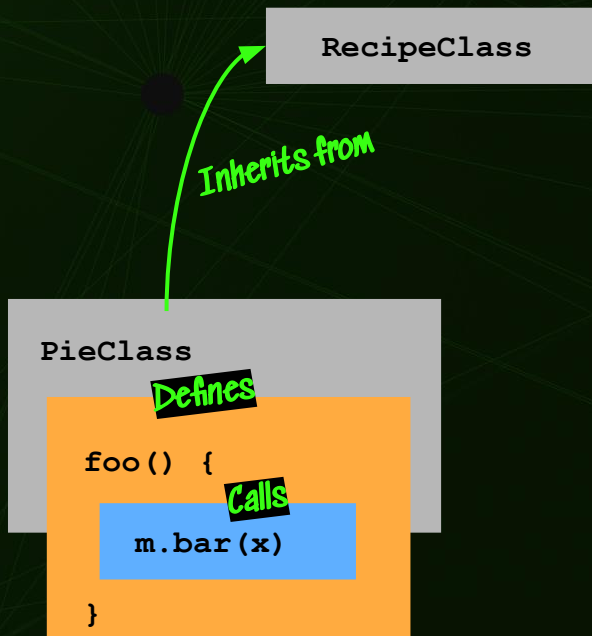
```
foo() {
```

Calls

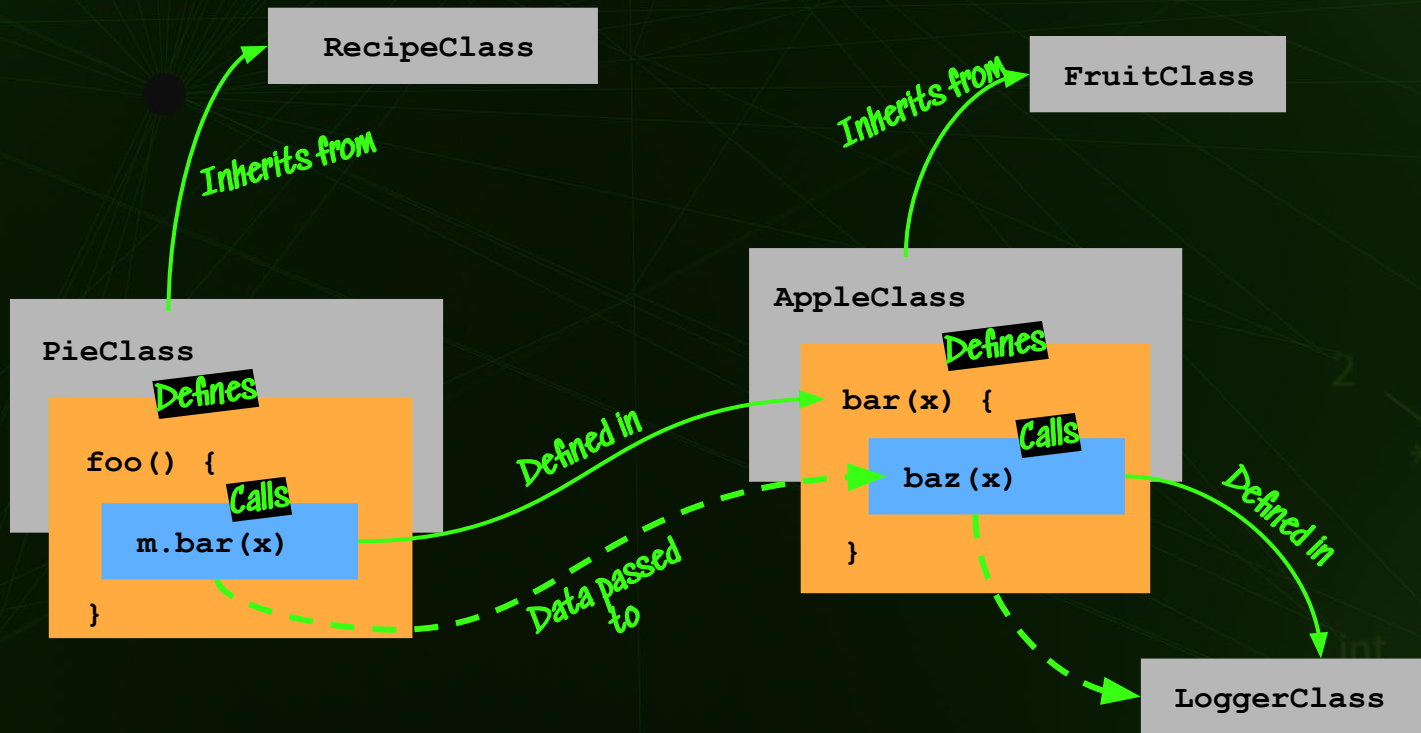
```
  m.bar(x)
```

```
}
```


Building Blocks of Code



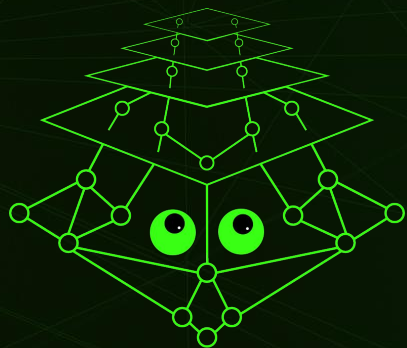
Building Blocks of Code



The background is a dark green grid with faint, glowing green lines and nodes. Scattered throughout are small, semi-transparent snippets of code, including labels like 'MAX', 'foo', 'sink', 'int', 'if', 'source', 'DECL', and 'STMT'.

ALL THE CODE IS A GRAPH

*If we think in graphs while coding, we should
think in graphs while debugging*

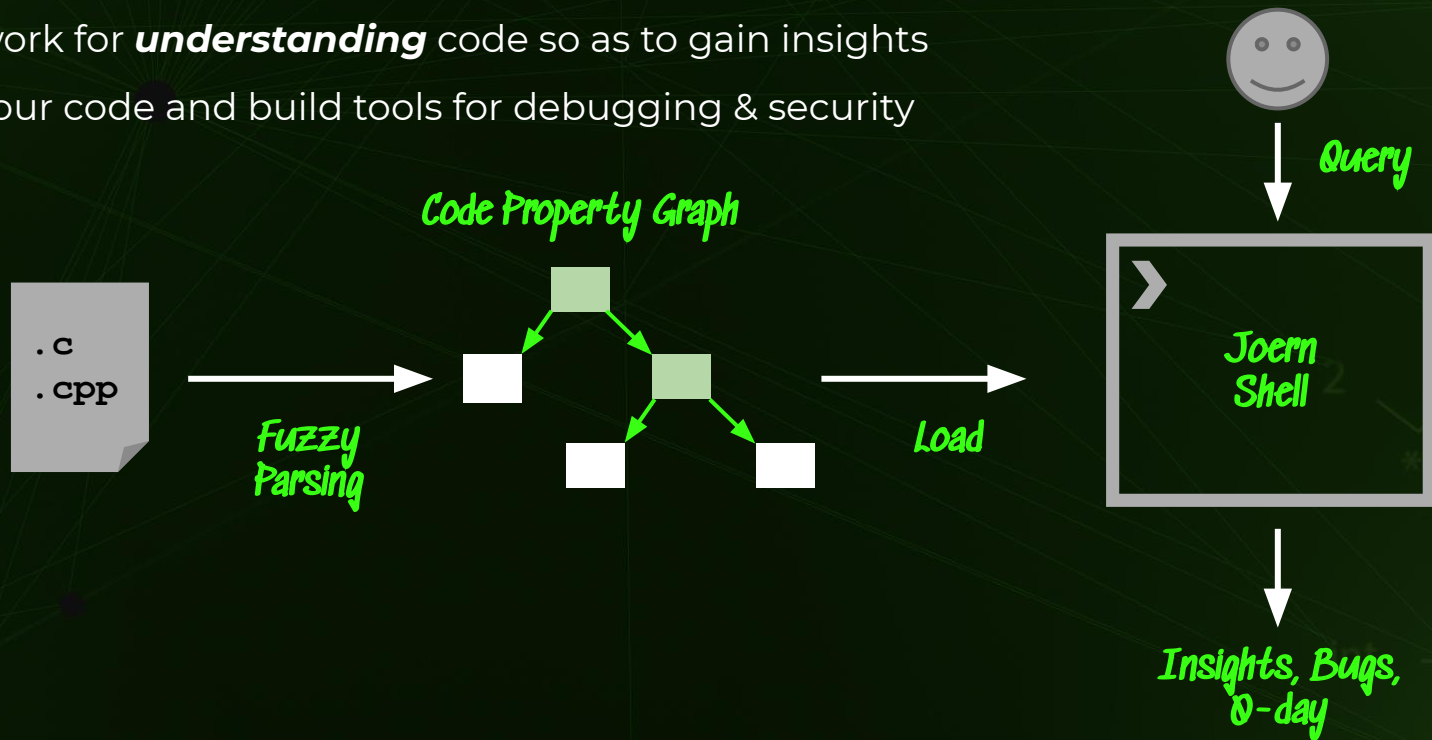


JOERN

[Yo! Urn]

What is Joern?

Framework for ***understanding*** code so as to gain insights about your code and build tools for debugging & security





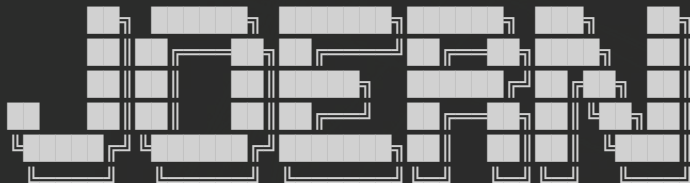
Module 1

Code Surfin'

Module 1

1. Parsing and Generating a CPG (VLC v3.0.8)

```
$ wget https://github.com/ShiftLeftSecurity/joern/releases/latest/download/joern-cli.zip
$ unzip joern-cli; cd joern-cli
$ ./joern-parse ~/joern-workshop/vlc-3.0.8/
$ ./joern
```



Welcome to Ocular/Joern

```
joern> loadCpg("cpg.bin")
```

```
res0: Option[Cpg] = Some(io.shiftleft.codepropertygraph.Cpg@4f7a2262)
```

```
joern> cpg.<TAB>
```

Module 1

2. Basic Navigation - Methods

```
suchakra@isengard: ~  
// List all methods that match `.*parse.*` to the shell  
joern> cpg.method.name(".*parse.*").name.l  
  
// Dump all methods that match `.*parse.*` to the shell (syntax-highlighted)  
joern> cpg.method.name(".*parse.*").dump  
  
// Create K-V pair of all methods that match `.*parse.*` and their code  
joern> cpg.method.name(".*parse.*").map( m=> (m.name, m.start.dump)).l  
  
// Dump all methods that match `.*parse.*` to file (no highlighting)  
joern> cpg.method.name(".*parse.*").dumpRaw |> "/tmp/foo.c"  
  
// View all methods that match `.*parse.*` in a pager (e.g., less)  
joern> browse(cpg.method.name(".*parse.*").dump)
```

Module 1

2. Basic Navigation - Methods

```
suchakra@isengard: ~  
+ x suchakra@isengard: ~  
  
// Dump dot representations of ASTs for all methods that  
// match `parse` into file  
joern> cpg.method.name("parse_public_key_packet").dot |> "/tmp/foo.dot"
```

Module 1

2. Basic Navigation - Methods

```
x
suchakra@isengard: ~
+ x suchakra@isengard: ~

// Find all local variables defined in a method
joern> cpg.method.name("parse_public_key_packet").local.name.1

// Find which file and line number they are in
joern> cpg.method.name("parse_public_key_packet").location.map( x=> (x.lineNumber.get,
x.filename)).1

// Find the type of the first local variable defined in a method
joern> cpg.method.name("parse_public_key_packet").local.typ.name.1.head

// Find all outgoing calls (call-sites) in a method
joern> cpg.method.name("parse_public_key_packet").callOut.name.1

// Find which method calls a method
joern> cpg.method.name("parse_public_key_packet").caller.name.1
```


3. Basic Navigation - Types and Filters

```
suchakra@isengard: ~  
+ x suchakra@isengard: ~  
  
// List all local variables of type `vlc_.*`  
joern> cpg.types.name("vlc_.*").localsOfType.name.1  
  
// Find member variables of a struct  
joern> cpg.types.name("vlc_log_t").map( x=> (x.name, x.start.member.name.1)).1  
  
// Find local variables and filter them by their type  
joern> cpg.local.filter(_.typ.name("vlc_log_t")).name.1  
  
// Which method are they used in?  
joern> cpg.local.filter(_.typ.name("vlc_log_t")).method.dump  
  
// Get the filenames where these methods are  
joern> cpg.local.filter(_.typ.name("vlc_log_t")).method.file.name.1
```

4. Basic Insights - Overview

```
suchakra@isengard: ~  
// Identify functions with more than 4 parameters  
joern> cpg.method.where(_.parameter.size > 4).l  
  
// Identify functions with > 4 control structures (cyclomatic complexity)  
joern> cpg.method.where(_.controlStructure.size > 4).l  
  
// Identify functions with more than 500 lines of code  
joern> cpg.method.where(_.numberOfLines >= 500).l  
  
// Identify functions with multiple return statements  
joern> cpg.method.where(_.ast.isReturn.l.size > 1)
```

4. Basic Insights - Overview

```
// Identify functions with more than 4 loops
```

```
joern> cpg.method.where(_.ast.isControlStructure.parserTypeName("(For|Do|While).*").size >  
4).1
```

```
// Identify functions with nesting depth larger than 3
```

```
joern> cpg.method.where(_.depth(_.isControlStructure) > 3).name.1
```

5. Basic Insights - Calls into libraries

```
suchakra@isengard: ~  
// All names of external methods used by the program  
joern> cpg.method.external.name.l.distinct.sorted  
  
// All calls to strcpy  
joern> cpg.call("str.*").code.l  
  
// All methods that call strcpy  
joern> cpg.call("str.*").method.name.l  
  
// Looking into parameters: second argument to sprintf is NOT a literal  
joern> cpg.call("sprintf").argument(2).filterNot(_.isLiteral).code.l  
  
// Quickly see this method above  
joern> cpg.call("sprintf").argument(2).filterNot(_.isLiteral).dump
```



Module 2

Memory Mémoire

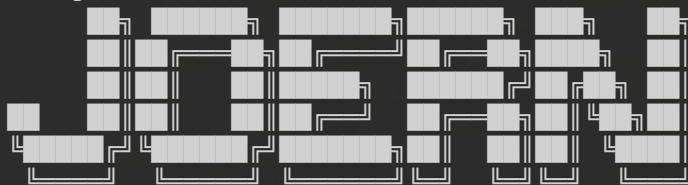
Module 2

1. Generating CPG for alloc_party.c

```
joern> exit
```

```
$ ./joern-parse ~/joern-workshop/alloc_party --out alloc.bin
```

```
$ ./joern
```



```
Welcome to Ocular/Joern
```

```
joern> loadCpg("alloc.bin")
```

```
res0: Option[Cpg] = Some(io.shiftleft.codepropertygraph.Cpg@4f7a2262)
```

```
joern> cpg.<TAB>
```


Module 2

2. Memory Allocation Bugs - Zero Alloc/Overflow

```
/*
 * So we have a situation where the malloc's argument contains an arithmetic operation
 *
 * This can lead to two cases:
 *   1. Zero Allocation, if the operation makes the argument 0 (we get a NULL ptr)
 *   2. Overflow, if the computed allocation is smaller and we use memcpy() eventually
 */

void *alloc_havoc(int y) {
    int z = 10;
    void *x = malloc(y * z);
    return x;
}
```

Module 2

2. Memory Allocation Bugs - Zero Alloc/Overflow

```
// The location where malloc has an arithmetic operation
```

```
joern> cpg.call("malloc").filter(_.argument(1).arithmetics).code.1
```

```
// Identify if there is a flow from a given method's param to a malloc()'s param with an  
// arithmetic operation
```

```
joern> var source = cpg.method.name(".*alloc.*").parameter
```

```
joern> var sink = cpg.call("malloc").filter(_.argument(1).arithmetics).argument
```

```
joern> sink.reachableByFlows(source).p
```

```
// Ref: https://suchakra.wordpress.com/2019/10/07/zero-day-snafus-hunting-memory-allocation-bugs/
```

Module 2

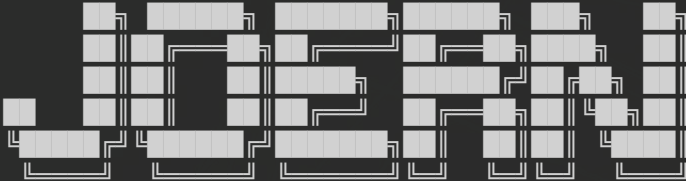
2. Memory Allocation Bugs - Double Free

```
// Find how many data flows are there starting from the same malloc() call-site  
// to parameters of all free() call-sites. If more than one, we can inspect them for double free
```

```
joern> var source = cpg.call(".*alloc.*")  
joern> var sink = cpg.call("free").argument  
joern> sink.reachableByFlows(source).p
```

Module 2

3. Back to the VLC CPG

```
suchakra@isengard: ~  
joern> exit  
$ ./joern  
  
  
Welcome to Ocular/Joern  
joern> loadCpg("vlc.bin")  
res0: Option[Cpg] = Some(io.shiftleft.codepropertygraph.Cpg@4f7a2262)
```

Module 2

3. Memory Allocation Bugs - Buffer Overflow

```
/**
 * Find calls to malloc where the first argument contains an arithmetic expression,
 * the allocated buffer flows into memcpy as the first argument, and the third
 * argument of that memcpy is unequal to the first argument of malloc. This is
 * an adaption of the old-joern query first shown at 31C3 that found a
 * buffer overflow in VLC's MP4 demuxer (CVE-2014-9626).
 */

val src = cpg.call("malloc").filter(_.argument(1).arithmetics).l

cpg.call("memcpy").whereNonEmpty { call => call.argument(1)
  .reachableBy(src.start)
  .filterNot(_.argument(1)
    .codeExact(call.argument(3).code))
}.code.l
```



Module 3

Scripting Away to Glory

Module 3

1. Scripting - Buffer Overflow

```
/*
 * We can wrap the previous query as a script that we can use internally
 * anytime we like. Just wrap it in a method!
 */

joern> def buffer_overflows(cpg : io.shiftleft.codepropertygraph.Cpg) = {
  val src = cpg.call("malloc").filter(_argument(1).arithmetics).l

  cpg.call("memcpy").whereNonEmpty { call => call.argument(1)
    .reachableBy(src.start)
    .filterNot(_argument(1)
      .codeExact(call.argument(3).code))
  }.code.l
}

defined function buffer_overflows
joern> buffer_overflows(cpg) // run the script from within Joern Shell!
```

Module 3

`p_block->i_buffer == MAX_UINT64` **causes an overflow!**

```
suchakra@isengard: ~  
joern> buffer_overflows(cpg).filter(_.method.name(".*ParseT.*")).l.start.dump  
res57: List[String] = List(  
  """static subpicture_t *ParseText( decoder_t *p_dec, block_t *p_block )  
  {  
    decoder_sys_t *p_sys = p_dec->p_sys;  
    subpicture_t *p_spu = NULL;  
    if( p_block->i_flags & BLOCK_FLAG_CORRUPTED )  
      return NULL;  
    ...  
    /* Should be resilient against bad subtitles */  
    if( p_sys->iconv_handle == (vlc_iconv_t)-1 || p_sys->b_autodetect_utf8 )  
    {  
      psz_subtitle = malloc( p_block->i_buffer + 1 );  
      if( psz_subtitle == NULL )  
        return NULL;  
      memcpy( psz_subtitle, p_block->p_buffer, p_block->i_buffer ); /* <=== */  
      psz_subtitle[p_block->i_buffer] = '\\0';  
    }  
  }  
  """  
)
```

Module 3

2. Scripting - Build Your Own Joern Scripts

```
// Save the following file as heap.sc
```

```
def buffer_overflows(cpg : io.shiftleft.codepropertygraph.Cpg) = {  
  val src = cpg.call("malloc").filter(_.argument(1).arithmetics).l  
  
  cpg.call("memcpy").whereNonEmpty { call => call.argument(1)  
    .reachableBy(src.start)  
    .filterNot(_.argument(1)  
      .codeExact(call.argument(3).code))  
  }.code.l  
}
```

```
// Import and execute from Joern
```

```
joern> import $file.^.heap
```

```
joern> heap.buffer_overflows(cpg)
```

Module 3

3. Scripting - DIY Tooling!

```
// Save the following file as buffer_overflow.sc
```

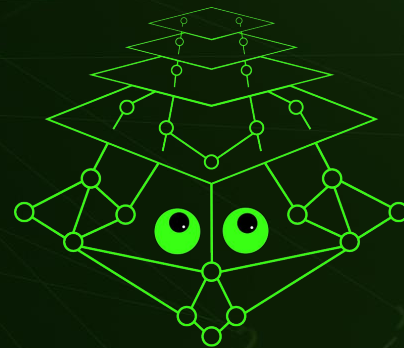
```
@main def execute(payload: String) = {  
  loadCpg(payload) ;  
  val src = cpg.call("malloc").filter(_.argument(1).arithmetics).l  
  
  cpg.call("memcpy").whereNonEmpty { call => call.argument(1)  
    .reachableBy(src.start)  
    .filterNot(_.argument(1)  
      .codeExact(call.argument(3).code))  
  }.code.l  
}
```

```
// Run externally as your own tool!
```

```
$ ./joern --script buffer_overflow.sc --params payload=vlc.bin
```

Acknowledgements

- Fabian Yamaguchi, friend and inventor of CPG and Joern
 - Talk: <https://fabs.codeminers.org/talk/2019-huawei/>
- Joern Community
 - Markus, Niko, Michael, Chetan
- Jöern (@joernchen) - yes, there is a real person for which the project was named



JOERN

Thanks Folks

Community: <https://gitter.im/joern-code-analyzer/community>

Website: <http://joern.io>

Workshop Resources: <https://github.com/tuxology/joern-workshop>



Twitter: @tuxology

Mail: mail@suchakra.in