

# RSAC<sup>®</sup>Conference2021

May 17 – 20 | Virtual Experience

SESSION ID: LAB2-R08

## Finding Stranger Things in Code

STATIC ANALYSIS USING JOERN

Suchakra Sharma

ShiftLeft Inc.



RESILIENCE

#RSAC

# Workshop Prep

- Clone Workshop Repo

- `git clone https://github.com/joernio/workshops`
- `cd workshops/2021-RSA`
- `apt install source-highlight graphviz unzip`

- Download **Joern** and install

- `wget https://github.com/joernio/joern/releases/latest/download/joern-install.sh`
- `chmod +x ./joern-install.sh`
- `sudo ./joern-install.sh`

- Download **VLC v3.0.12** source and extract in a convenient directory

- `wget http://get.videolan.org/vlc/3.0.12/vlc-3.0.12.tar.xz`
- `tar -xvf vlc-3.0.12.tar.xz`

# Workshop Prep

- Machine Requirements

- At least 5-7GB free RAM (close as many browser tabs as possible, pkill slack etc)
- At least 4 CPUs (preferably modern)
- OpenJDK 1.8+

- Important Links

- Joern Docs: <https://docs.joern.io>
- Queries: <https://queries.joern.io>
- Joern Community: <https://discord.gg/SrUX84xMFR> Join **#rsa-lab2-r08**

# Suchakra Sharma

Staff Scientist, ShiftLeft Inc.  
Joern Contributor

Github: tuxology

Twitter: @tuxology

Email: mail@suchakra.in

*PhD, Computer Engineering - loves systems, code analysis,  
performance analysis, hardware tracing, samosas and poutine!*

# Why are you here?

- **You may have the following question in your mind**
  - How do computer programs and programming languages work?
  - I know some bad coding practices. How can I “mass detect” them in large codebases?
  - How do static analysis tools work? Can I create my own tools?
  - So many of these tools. Can I get like a CLI shell to hunt bugs?
  - *How hard is it to find a Zero Day?*

# What you will learn

- **Gain ability to find vulnerabilities in large code-bases (such as VLC)**
- **Interactive code analysis and code exploration**
- **Convert your manual code auditing steps to automated analyses**
- **Get insights about how external libraries are being used by your own code**
- **Stop reliance on “vendor SAST” and roll your sleeves to find real bugs**
- **Some proficiency in Scala**

# Static Analysis Primitives

*What is even code?*

# 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() {
        Patient pat = patientRepository.findOne(1l);

        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 {}");
        }

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

**Annotations:** `@RestController`, `@RequestMapping`

**Package/Namespaces:** `org.springframework.web.bind.annotation`

**Class/Type:** `PatientController`

**Member Variable:** `log`

**Local Variable:** `pat`

**Method Parameter:** `Int Id`

**Method Definition:** `getPatient(Int Id)`

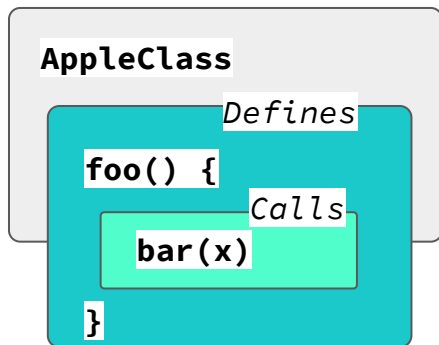
**Method Block:** `{ ... }`

**Method Instance:** `return patientRepository.findAll();`

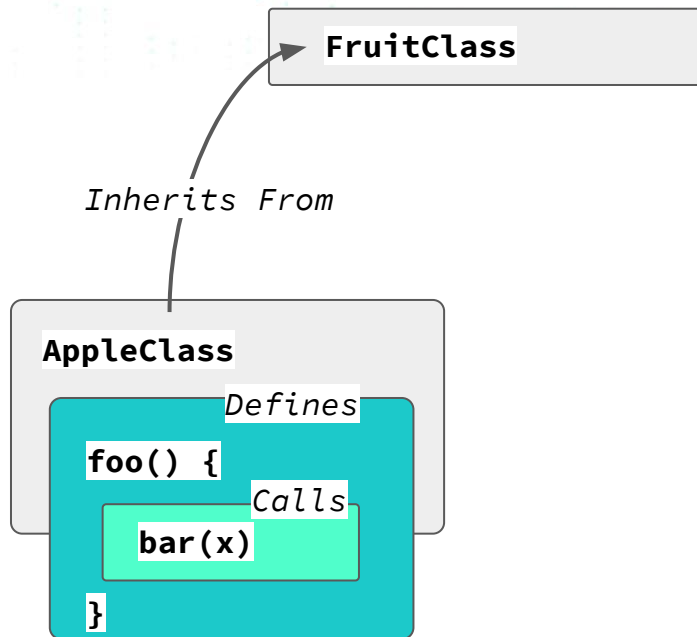
**Literal:** `"First Patient is {}"`

**Method Return:** `findAll()`

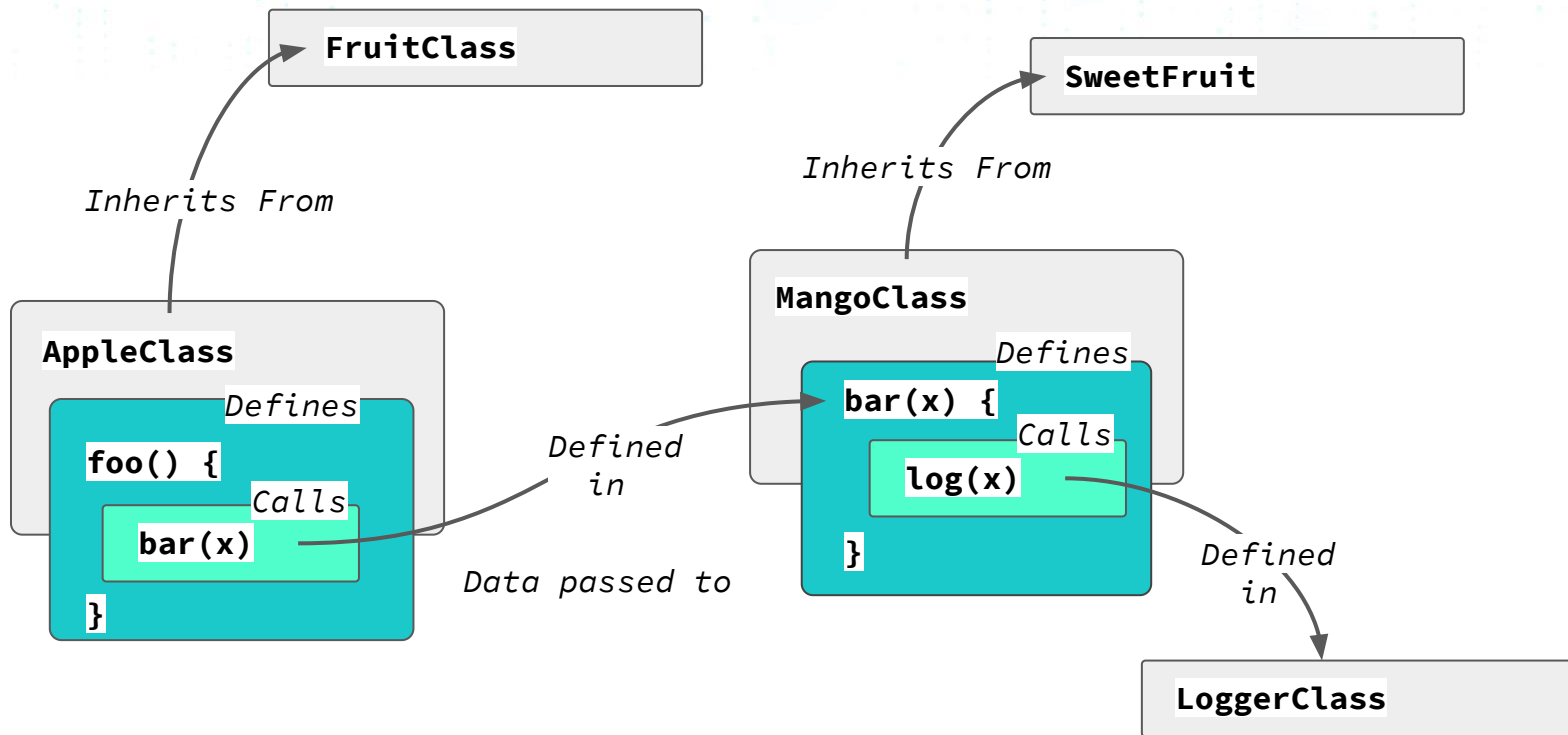
# Building Blocks of Code



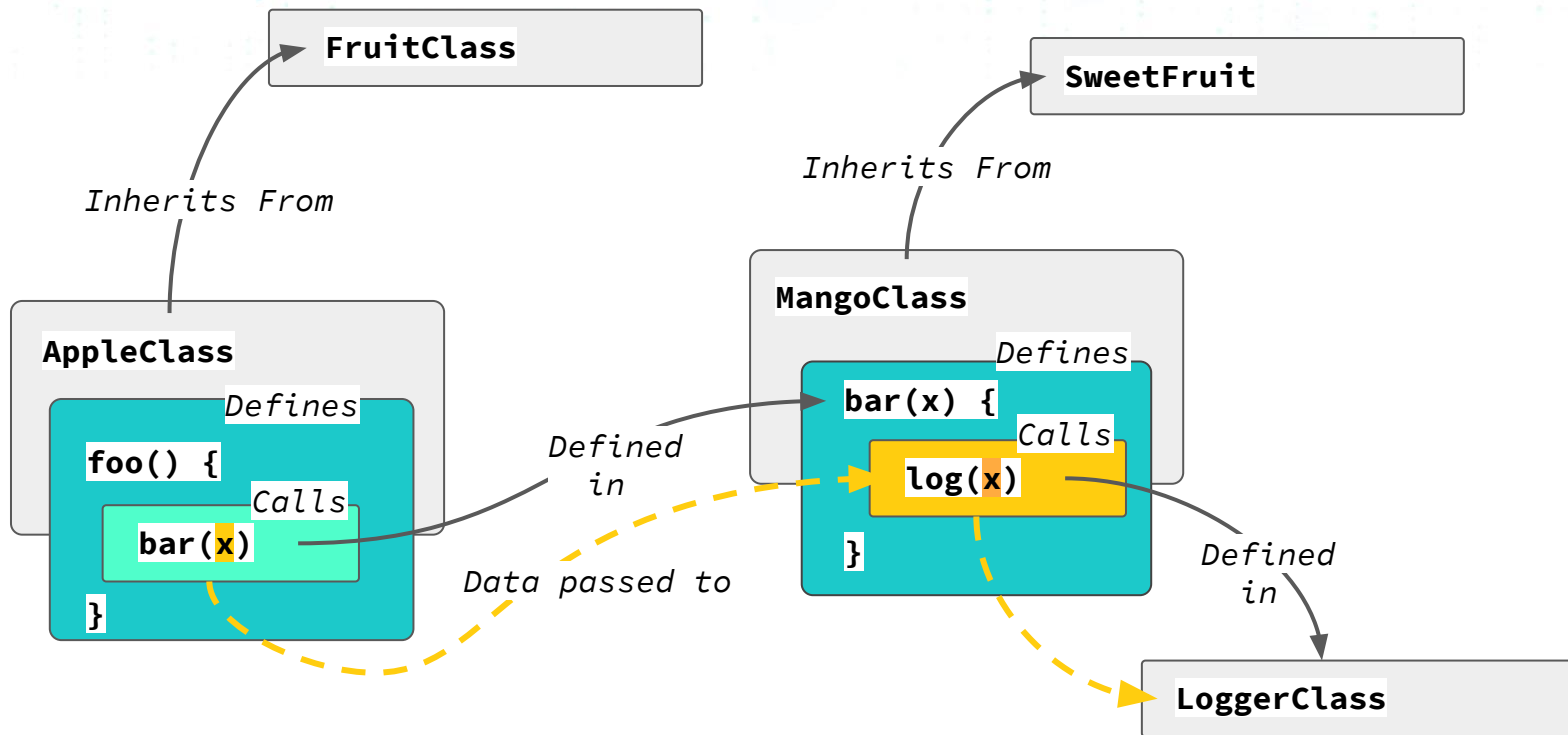
# Building Blocks of Code



# Building Blocks of Code



# Building Blocks of Code



# ALL THE CODE IS A GRAPH

*We think in graphs while coding - we should think in graphs while debugging*

# Workshop Plan

- **FAMILIARIZATION**
- **MODULE 1**
  - **Basic Code Navigation** - Searching functions, types, call-sites
  - **Basic Insights** - Finding rule violations
- **WORKING BREAK 1**
- **MODULE 2**
  - **Hunting Bugs** - Finding memory allocation related bugs
- **WORKING BREAK 2**
- **MODULE 3**
  - **VLC Vulnerability discovery exercise**
- **SHOW AND TELL**

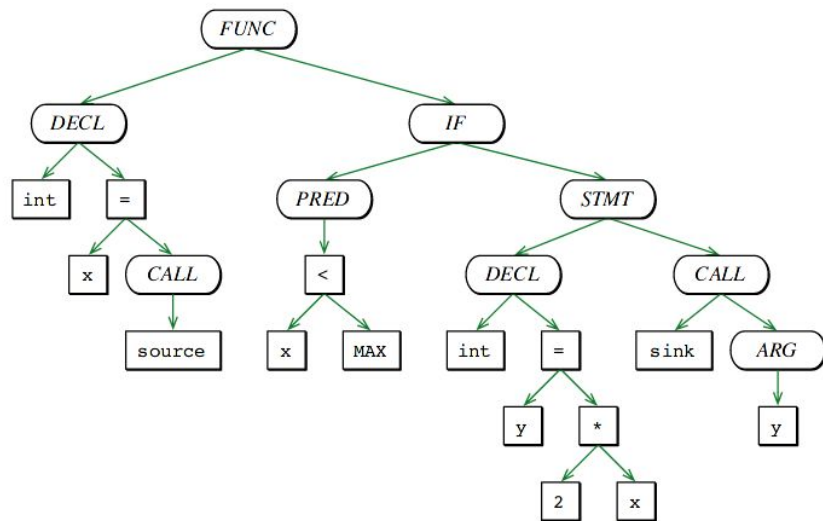
# Workshop Plan

- **FAMILIARIZATION**
- **MODULE 1**
  - **Basic Code Navigation** - Searching functions, types, call-sites
  - **Basic Insights** - Finding rule violations
- **WORKING BREAK 1**
- **MODULE 2**
  - **Hunting Bugs** - Finding memory allocation related bugs
- **WORKING BREAK 2**
- **MODULE 3**
  - **VLC Vulnerability discovery exercise**
- **SHOW AND TELL**

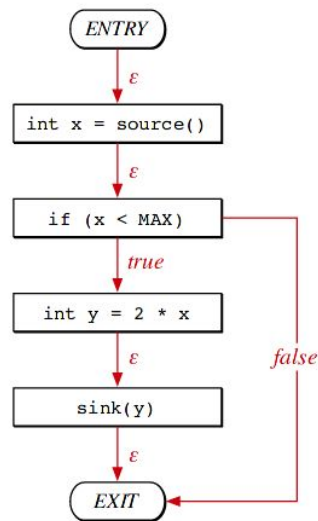


# Graph Data Structures For Code

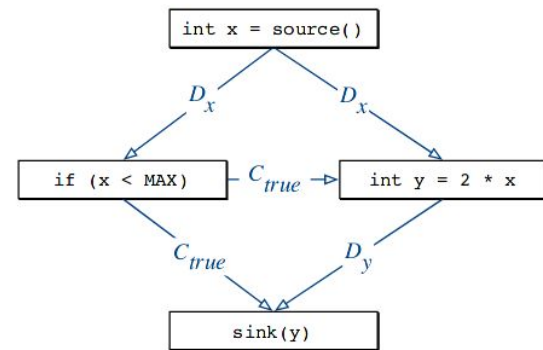
```
void foo()
{
    int x = source();
    if (x < MAX)
    {
        int y = 2 * x;
        sink(y);
    }
}
```



(a) Abstract syntax tree (AST)



(b) Control flow graph (CFG)

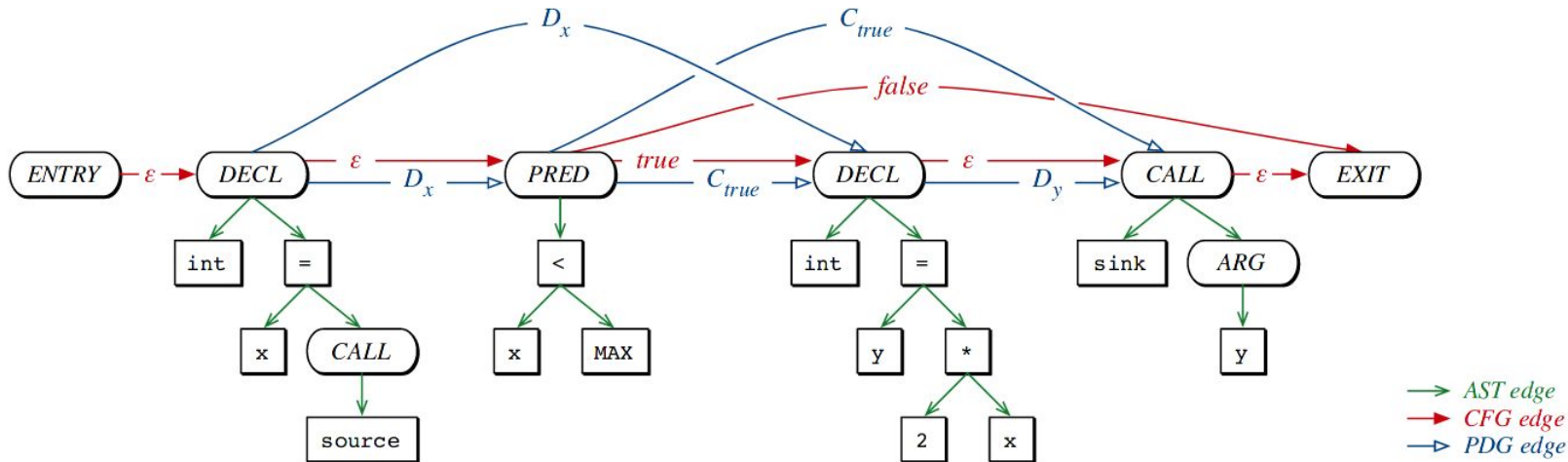


(c) Program dependence graph (PDG)

# Code Property Graph (CPG)

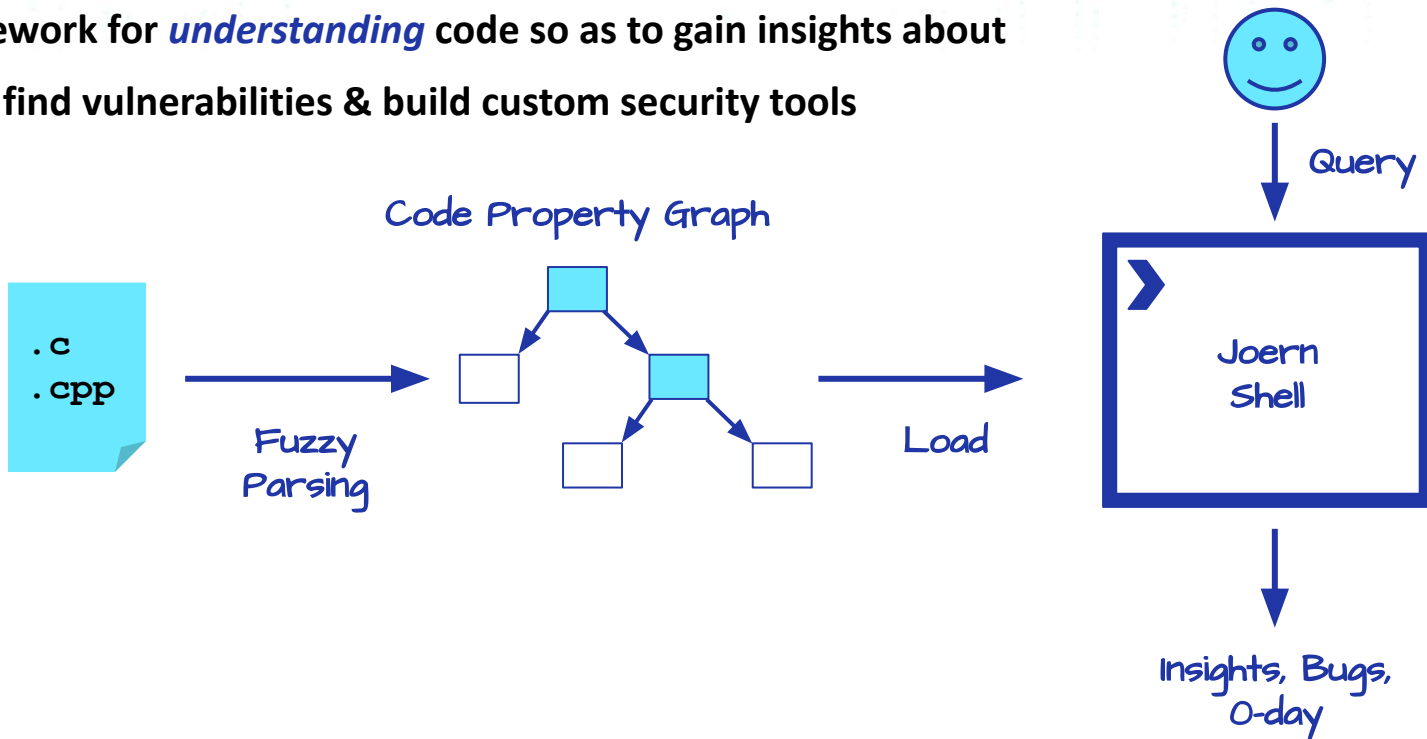
*Modeling and Discovering Vulnerabilities with Code Property Graphs*, Yamaguchi et al. (IEEE Symp. Sec and Priv., 2014)

```
void foo()  
{  
    int x = source();  
    if (x < MAX)  
    {  
        int y = 2 * x;  
        sink(y);  
    }  
}
```

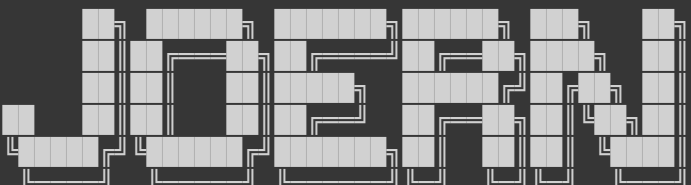


# Exploring CPG with Joern

- Framework for *understanding* code so as to gain insights about code, find vulnerabilities & build custom security tools



# Quickstart

```
suchakra@isengard: ~  
$ wget http://www.acme.com/software/thttpd/thttpd-2.29.tar.gz  
$ tar -xvf thttpd-2.29.tar.gz  
$ joern  
  
Type `help` or `browse(help)` to begin  
joern> importCode("/tmp/thttpd-2.29")  
res0: Cpg = io.shiftleft.codepropertygraph.Cpg@4f7a2262)  
joern> cpg.method.name("handle.*").name.1  
...  
joern> cpg.method.name("strcpy").caller.name.1  
res18: List[String] = List(  
  "get_filename"
```

# Workshop Plan

- **FAMILIARIZATION**
- **MODULE 1**
  - **Basic Code Navigation** - Searching functions, types, call-sites
  - **Basic Insights** - Finding rule violations
- **WORKING BREAK 1**
- **MODULE 2**
  - **Hunting Bugs** - Finding memory allocation related bugs
- **WORKING BREAK 2**
- **MODULE 3**
  - **VLC Vulnerability discovery exercise**
- **SHOW AND TELL**

# MODULE 1

## Code Navigation and Insights

# Import VLC Code



```
suchakra@isengard: ~  
$ joern  
  
Type `help` or `browse(help)` to begin  
joern> importCode"/tmp/vlc-3.0.12")  
res0: Cpg = io.shiftleft.codepropertygraph.Cpg@4f7a2262)  
joern> run.ossdataflow  
joern> save
```

# Basic Navigation - Methods

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



# Basic Navigation - Methods

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

# Basic Navigation - Repeating Graph Traversals

```
suchakra@isengard: ~  
+ x suchakra@isengard: ~  
  
// Find the sequence of callers going UP from a given method  
joern> cpg.method.name("parse_public_key_packet").repeat(_.caller)(_.emit).name.l  
  
// Find the callees of a method going DOWN until you hit a given method(CAN BE EXPENSIVE)  
joern>  
cpg.method.name("download_key").repeat(_.callee)(_.emit.until(_.isCallTo("parse_public_key_packet"))).name.l
```

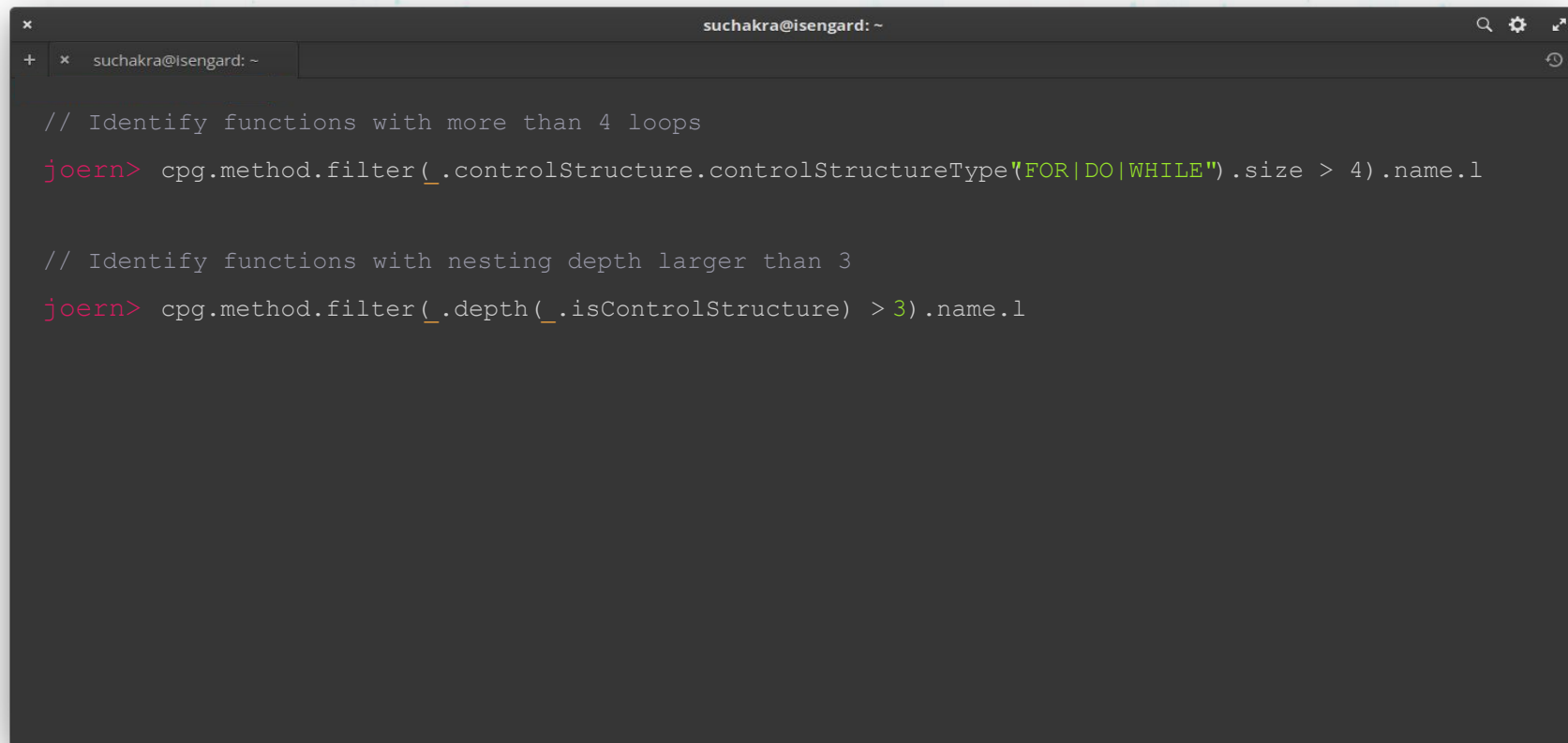
# Basic Navigation - Types, Variables and Filtering

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

# Basic Insights - Code Complexity

```
suchakra@isengard: ~  
// Identify functions with more than 4 parameters  
joern> cpq.method.filter(_parameter.size >4).name.l  
  
// Identify functions with > 4 control structures (cyclomatic complexity)  
joern> cpq.method.filter(_controlStructure.size >4).name.l  
  
// Identify functions with more than 500 lines of code  
joern> cpq.method.filter(_numberOfLines >=500).name.l  
  
// Identify functions with multiple return statements  
joern> cpq.method.filter(_ast.isReturn.l.size >1).name.l
```

# Basic Insights - Code Complexity



```
suchakra@isengard: ~  
+ x suchakra@isengard: ~  
  
// Identify functions with more than 4 loops  
joern> cpq.method.filter(_.controlStructure.controlStructureType("FOR|DO|WHILE").size > 4).name.l  
  
// Identify functions with nesting depth larger than 3  
joern> cpq.method.filter(_.depth(_.isControlStructure) > 3).name.l
```

# QUIZ



```
suchakra@isengard: ~  
  
QUIZ  
  
// Create a query that finds recursive functions  
joern>
```

# QUIZ



```
suchakra@isengard: ~  
+ x suchakra@isengard: ~  
  
// Create a query that finds recursive functions  
joern> cpg.method.filter(x => x.call.name.l.contains(x.name)).name.l  
res88: List[String] = List(  
  "dirfd",  
  "tdestroy_recurse",  
  "vlc_dictionary_insert_impl_",  
  ...  
)
```

# Workshop Plan

- **FAMILIARIZATION**
- **MODULE 1**
  - **Basic Code Navigation** - Searching functions, types, call-sites
  - **Basic Insights** - Finding rule violations
- **WORKING BREAK 1**
- **MODULE 2**
  - **Hunting Bugs** - Finding memory allocation related bugs
- **WORKING BREAK 2**
- **MODULE 3**
  - **VLC Vulnerability discovery exercise**
- **SHOW AND TELL**



# MODULE 2

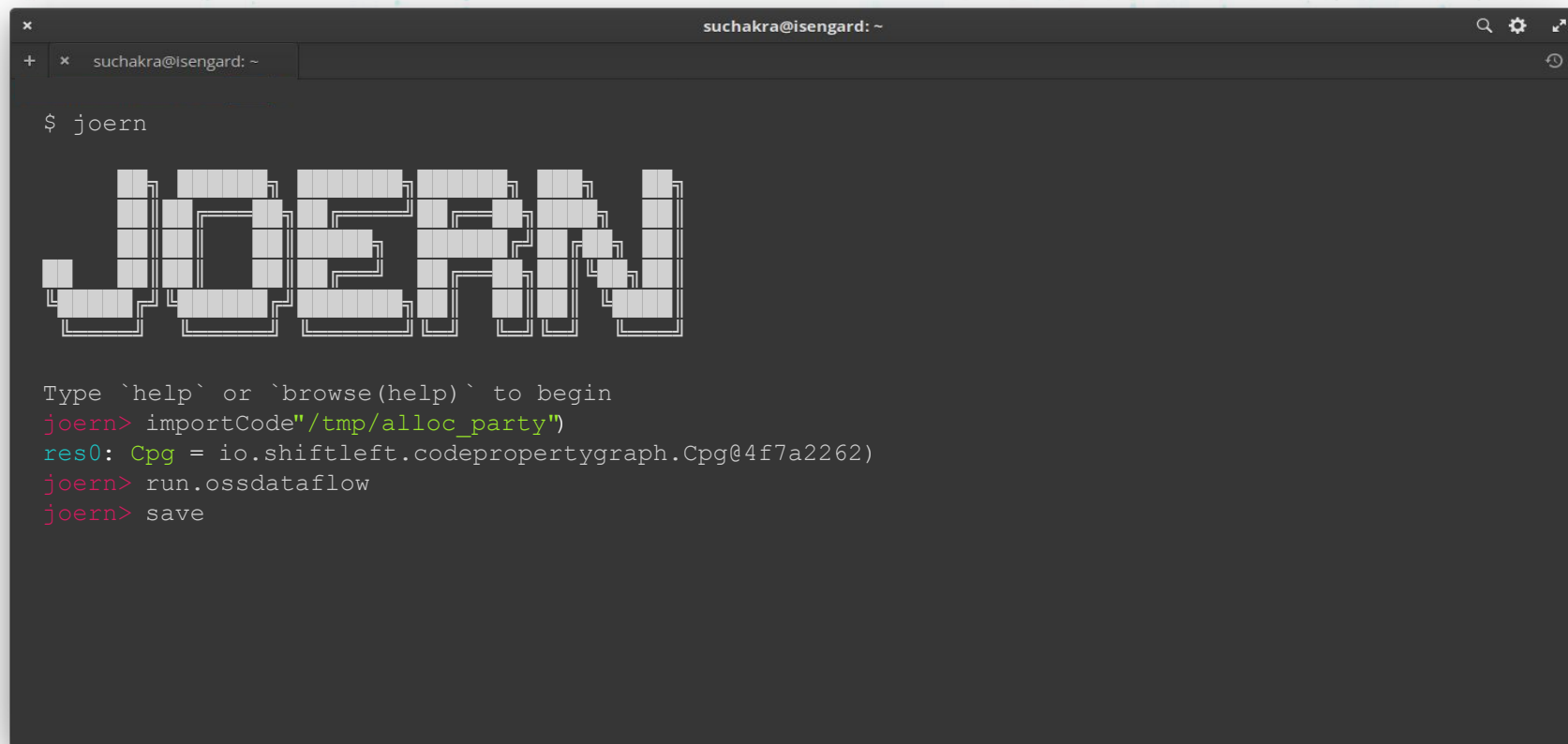
## Hunting Bugs

# 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;
}
```

# Import alloc\_party Code

```
suchakra@isengard: ~  
$ joern  
  
  
Type `help` or `browse(help)` to begin  
joern> importCode"/tmp/alloc_party")  
res0: Cpg = io.shiftleft.codepropertygraph.Cpg@4f7a2262)  
joern> run.ossdataflow  
joern> save
```

# Memory Allocation Bugs - Zero Alloc/Overflow

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

```
joern> cpg.call("malloc").where(_.argument(1).isCallTo(Operators.multiplication)).code.1
```

```
// Identify if there is a call from some method to any of these weird mallocs
```

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

```
joern> def sink = cpg.call("malloc").where(_.argument(1).isCallTo(Operators.multiplication)).argument
```

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

# QUIZ



```
suchakra@isengard: ~  
  
// Write a query to find a double free the sample code (Hint: Dataflow from allocation to freedom)  
joern>
```

# QUIZ



```
suchakra@isengard: ~  
+ x suchakra@isengard: ~  
  
// Write a query to find a double free the sample code (Hint: Dataflow from allocation to freedom)  
joern> def source = cpq.call(".*alloc.*")  
joern> def sink = cpq.call("free").argument  
joern> sink.reachableByFlows(source).p
```

# Workshop Plan

- **FAMILIARIZATION**
- **MODULE 1**
  - **Basic Code Navigation** - Searching functions, types, call-sites
  - **Basic Insights** - Finding rule violations
- **WORKING BREAK 1**
- **MODULE 2**
  - **Hunting Bugs** - Finding memory allocation related bugs
- **WORKING BREAK 2**
- **MODULE 3**
  - **VLC Vulnerability discovery exercise**
- **SHOW AND TELL**

## MODULE 3

# Finding Vulnerabilities in VLC



# Buffer Overflow Hunting in VLC - First Try!

```
suchakra@isengard: ~  
+ x suchakra@isengard: ~  
  
// Find the memcpy() calls where return value of calls from malloc having addition  
operations reaches the first argument of the memcpy  
  
def src = cpg.call("malloc").where(_ argument(1).isCallTo(Operators.addition)).l  
  
cpg.call("memcpy").where { call =>  
    call.argument(1)  
    .reachableBy(src)  
}.code.l
```

# Buffer Overflow Hunting in VLC - Dataflows [OPTIONAL]

```
suchakra@isengard: ~  
+ x suchakra@isengard: ~  
  
// Find dataflows from all these interesting sources and sinks  
  
joern> def source = cpg.call("malloc").where(_.argument(1).isCallTo(Operators.addition))  
defined function source  
  
joern> def sink = cpg.call("memcpy").argument  
defined function sink  
  
joern> sink.reachableByFlows(source).p
```

# DRY Functions

```
// Wrap possible buffer overflow query in a function
```

```
joern> def buffer_overflows(cpg : io.shiftleft.codepropertygraph.Cpg) = {  
    def src =  
    cpg.call("malloc").where(_ .argument(1).isCallTo(Operators.addition)).l  
    cpg.call("memcpy").where { call =>  
        call.argument(1)  
        .reachableBy(src)  
    }.code.l  
}
```

```
defined function buffer_overflows
```

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

# Scripting with Joern - DIY Tools

```
suchakra@isengard: ~  
// save the following text as mytools.sc in /home/$USER/bin/joern  
  
def buffer_overflows(cpg : io.shiftleft.codepropertygraph.Cpg ) = {  
  def src =  
cpg.call("malloc").where(_argument(1).isCallTo(Operators.addition)).l  
  cpg.call("memcpy").where { call =>  
    call.argument(1)  
    .reachableBy(src)  
  }.code.l  
}  
  
joern> import $file.mytools(cpg) // import your script  
joern> mytools.buffer_overflows(cpg) // run the script from within Joern Shell!
```

# Scripting with Joern - DIY Tools

```
// save the following text as buffer_overflows.sc in /home/$USER/bin/joern
// You can replace the open(graph) with other command like importCode() to work on
// fresh code. You could generate JSONs also, create reports etc..
```

```
@main def execute(graph: String) = {
  open(graph)
  println("Finding possible buffer overflows" )
  def src =
cpg.call("malloc").where(_ .argument(1) .isCallTo(Operators.addition)).l
  cpg.call("memcpy").where { call =>
    call.argument(1)
    .reachableBy(src)
  }.code.l
}
```

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

```
$ joern --script buffer_overflow.sc --params graph=vlc-3.0.12
```

## ACTIVITY

### 1. Find Possible Buffer Overflows in VLC

There is at least one “close but no cigar” and one real case. May be tough

### 2. Anything else that is interesting -\\_ (ツ) \\_/-

Open-ended tinkering with VLC. Lets see what you get!

# Workshop Plan

- **FAMILIARIZATION**
- **MODULE 1**
  - **Basic Code Navigation** - Searching functions, types, call-sites
  - **Basic Insights** - Finding rule violations
- **WORKING BREAK 1**
- **MODULE 2**
  - **Hunting Bugs** - Finding memory allocation related bugs
- **WORKING BREAK 2**
- **MODULE 3**
  - **VLC Vulnerability discovery exercise**
- **SHOW AND TELL**

# SHOW AND TELL



# p\_block->i\_buffer == MAX\_UINT64 causes overflow!!

```
joern> buffer_overflows(cpg).filter(_.method.name(".*ParseText.*")).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';
    }
}
```

**FIN**

<https://joern.io>