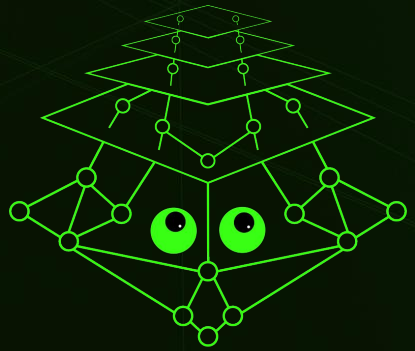
A Practical Introduction to the Code Analysis Platform Joern

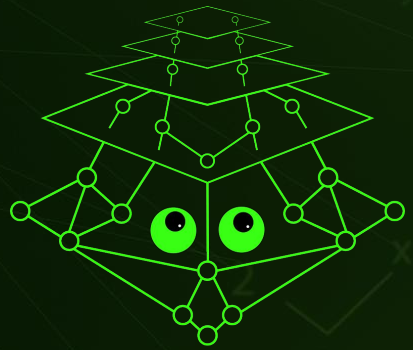
Fabian ‘fabs’ Yamaguchi



JOERN

Interactive Code Analysis

- Philosophy: each program is its own universe, and hacking is about exploring, documenting and exploiting its rules
- Fully-automated static scanners are of limited use in this setting - but augmenting the auditor’s capabilities with powerful code analysis primitives is fruitful!
- Provide the primitives to script as much of the analysis as possible - to incrementally increase and store knowledge about the target => think IDA Pro, Radare2, or Burp, not Veracode



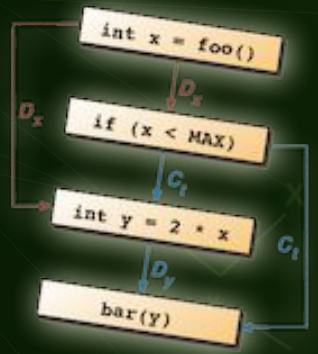
JOERN

What is this Project about?

- We want to explore how partial automation can help hackers working on in-depth analysis of high profile targets
- This is not about finding simple bugs (fuzzers are better here), it is about having a workbench for long audits
- The focus is on code understanding

A Brief Look into the Back Mirror

- 2013: Initial release of Joern as a rough research prototype - developed as part of a PhD thesis on pattern-based vulnerability discovery via code property graphs
- Late 2016: Research prototype abandoned
- 2017: Work on commercial version ”Ocular” initiated - core technology behind a commercial product offering
- 2018: Open-sourcing of a code property graph specification
- Late 2019: Partial open-sourcing of Ocular as Joern-Reborn



(2013 logo :))

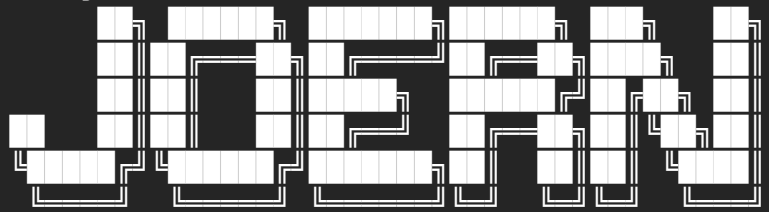
Fuzzy Parsing

- Fuzzy parser for C/C++. Analyze code bases even if header files or the right compiler are not available
- This means that you do not have to waste time configuring the target to work with the tool
- You can also use it on the code that falls out of decompilers, or on code that “fell off a truck” (like in Marco’s baseband work)
- Includes a fuzzy preprocessor to make use of headers if they are available

Getting started: parsing, launching, loading

fabs@workstation:~/joern/

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

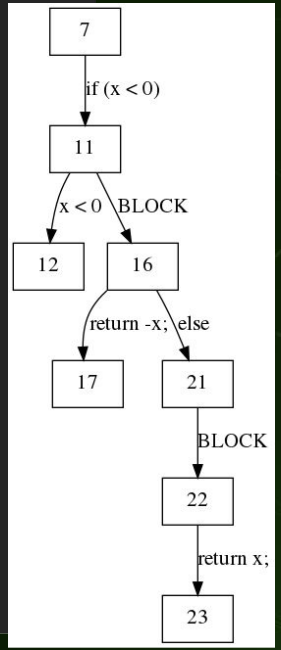


```
Welcome to Ocular/Joern
joern> loadCpg("cpg.bin.zip")
res0: Option[Cpg] = Some(io.shiftleft.codepropertygraph.Cpg@3e1ca7c3)
joern> cpg.<TAB>
```

“Integration”: Dumping Code and Piping it Out

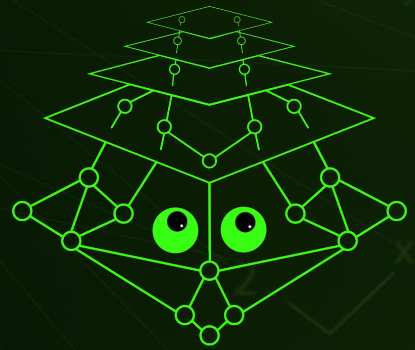
fabs@workstation:~/joern/

```
// Dump all methods that match `.*parse.*` to the shell (syntax-highlighted)
joern> cpg.method.name(".*parse.*").dump
// 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)
// Dump dot representations of ASTs for all methods that
// match `parse` into file
joern> cpg.method.name(".*parse.*").dot |> "/tmp/foo.dot"
```



Interactive Shell

- Interactive shell for code exploration and query crafting
- The shell has syntax completion to learn the language
- It allows you to browse syntax highlighted code so that you don’t have to leave the tool
- Query results and code can be piped into files (“integration”)
- Batch processing turns it into a runtime



JOERN

Complexity Metrics - Something simple to start off with

fabs@workstation:~/joern/

```
// Identify functions with more than 4 parameters
cpg.method.where(_.parameter.size > 4).l
// Identify functions with > 4 control structures (cyclomatic complexity)
cpg.method.where(_.controlStructure.size > 4).l
// Identify functions with more than 500 lines of code
cpg.method.where(_.numberOfLines >= 500).l
// Identify functions with multiple return statements
cpg.method.where(_.ast.isReturn.l.size > 1)
// Identify functions with more than 4 loops
cpg.method.where(_.ast.isControlStructure.parserTypeName("(For|Do|While).*").size > 4).l
// Identify functions with nesting depth larger than 3
cpg.method.where(_.depth(_.isControlStructure) > 3).name.l
```

Exploring calls into libraries

fabs@workstation:~/joern/

```
// All names of external methods used by the program
Cpg.method.external.name.l.distinct.sorted
// All calls to strcpy
cpg.call("str.*").code.l
// All methods that call strcpy
cpg.call("str.*").method.name.l
// Looking into parameters: second argument to sprintf is NOT a literal
cpg.call("sprintf").argument(2).filterNot(_.isLiteral).code.l
```

Storing and Making use of What You've Already Found Out

fabs@workstation:~/joern/

```
// Create a new graph to hold an additive diff (DiffGraph)
implicit val diffGraph = new io.shiftleft.passes.DiffGraph()

// Methods that accept a "char *" and a "size_t"
cpg.method.filter(_.parameter.evalType("size_t"))
          .filter(_.parameter.evalType(".*void.*"))
          .newTagNodePair("copy_operation").store

diffGraph.apply(cpg)
// You can now retrieve copy operations that you marked earlier!
// The kind of workflow you know from IDA
cpg.tag.name("copy_operation").parameter...
```

Let’s see which functions are called most often (“language”)

fabs@workstation:~/joern/

```
// Sort methods by number of callers and dump the first 100
joern> cpg.method.map(x => (x.start.callIn.size,
x.name)).l.sorted.reverse.take( 100)
res16: List[(Int, String)] = List(
  (108003, "<operator>.indirectMemberAccess" ),
  (87500, "<operator>.assignment" ),
  (42012, "<operator>.memberAccess" ),
  (22498, "<operator>.addressOf" ),
  (20280, "<operator>.computedMemberAccess" ),
     ...
  (5436, "free"),
  (3262, "msg_Dbg"),
)
```

Extend `cpg.method` via an implicit conversion

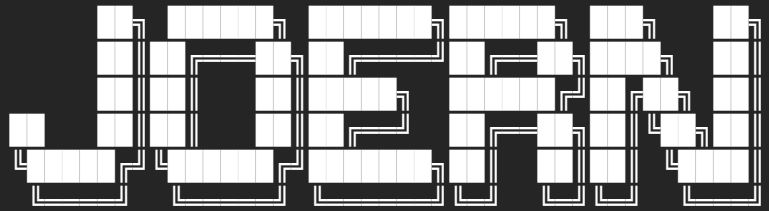
fabs@workstation:~/joern/

```
joern> implicit class MyMethod(method : Steps[Method]) {
         def top(n : Int) =
           method.map(x => (x.start.callIn.size,
      x.name)).l.sorted.reverse.take( 100)
       }
defined class MyMethod
joern> cpg.method.top(10)
res16: List[(Int, String)] = List(
  (108003, "<operator>.indirectMemberAccess" ),
  (87500, "<operator>.assignment" ),
  (42012, "<operator>.memberAccess" ),
      ...
)
```

Import your script at startup

fabs@workstation:~/joern/

```
~/joern $ mkdir -p scripts/myjoernhax/
~/joern $ echo 'println("Loading my hacks")' > scripts/myjoernhax/hacks.sc
~/joern $ mkdir -p ~/.shiftleft/ocular/
~/joern $ echo 'runScript("myjoernhax", cpg)' >> ~/.shiftleft/ocular/predef.scala
./joern
```



```
Loading my hacks
Welcome to Ocular/Joern
joern> runScript(“myjoernhax”, cpg) // <--- or kick-off manually after loading CPG
```

Easy Extensibility of the Language is Key

- The query language does not limit you to the code analysis ideas that its developers have
- It is an “internal Domain Specific Language” based on Scala, meaning that you can use all of Scala as part of your query
- Extending the language and query writing are the same
- You can use existing IDEs (IntelliJ) to work on complex custom program analysis features on top of Joern

IntelliJ as a Joern IDE

- Query language is correctly completed by IntelliJ
- Test fixtures (e.g., `DataFlowCodeToCpgFixture`) allow creation of Test graphs from C/C+ code
- Create queries as unit tests in `queries/src/test/scala/`
- Run queries as unit tests and use built-in debugger to debug your queries

```
class MallocMemcpyTests extends WordSpec with Matchers {
  val code: String =
    """
      int vulnerable(size_t len, char *src) {
        char *dst = malloc(len + 8);
        memcpy(dst, src, len + 7);
      }

      int non_vulnerable(size_t len, char *src) {
       char *dst = malloc(len + 8);
       memcpy(dst, src,len + 8);
      }
    """

  DataFlowCodeToCpgFixture(code) { cpg =>
    /**
      * Find calls to malloc where the first argument contains an arithmetic expres
      * 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).
      * */
    "find calls to malloc/memcpy system with different expressions in arguments" in

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

      cpg.call( regex = "memcpy").whereNonEmpty { call =>
```

```
MallocMemcpyTests › λ(cpg: Any)
Run:  MallocMemcpyTests.find calls to malloc/me...
Tests passed: 1 of 1 test – 69 ms
Test Results  69 ms
Testing started at 1:39 AM ...
/usr/lib/jvm/java-8-openjdk-amd64/bin/java -javaagent:/snap/inte
CPG construction finished in 485ms.
Start of enhancement: io.shiftleft.semanticcpg.passes.compat.arg
Using old CPG format not containing ARGUMENT edges.
```

Detecting Write Loops - Extension Mechanism Hard at Work

fabs@workstation:~/joern/

```
// Return (arrayName, List(subscripts))
// Noisy version without decoration language
cpg
    .call(".*assign.*")
    .argument(1).ast.isCall
    .name(".*op.*computedMemberAccess.*")
    .map { call =>
           val subscripts = call.argument(2).ast
                       .isIdentifier.code.toSet
           (call.argument(1), subscripts)
     }
```

fabs@workstation:~/joern/

```
// Return (arrayName, List(subscripts))
// Expressive version with decoration language
cpg
    .assignment.target.isArrayAccess
    .map { a =>
      (a.array, a.subscripts.code.toSet)
    }
```

Query for heap-based buffer overflows (malloc/memcpy - arithmetic)

fabs@workstation:~/joern/

```
/**
  * 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))
          }
```

Comparing it to the dark ages - and old-Joern query from 2013

fabs@workstation:~/joern/

```
/**
  * 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).
```

```
echo 'getCallsTo("malloc").ithArguments("0")
.sideEffect{cnt = it.code }
.match{ it.type =="AdditiveExpression"}.statements()
.out("REACHES")
.match{ it.type == "CallExpression" &&
     it.code.startsWith("memcpy")}.ithArguments("2")
.filter{it.code != cnt }
.match{it.type == "AdditiveExpression"}.id‘
```

What we found back then already

p_box->i_size = 7 triggers the overflow

```
static int MP4_ReadBox_name( stream_t *p_stream, MP4_Box_t *p_box )
{
  MP4_READBOX_ENTER( MP4_Box_data_name_t );
  p_box->data.p_name->psz_text =
  malloc( p_box->i_size + 1 - 8 ); /* +\0, -name, -size */
    if( p_box->data.p_name->psz_text == NULL )
        MP4_READBOX_EXIT( 0 );

 memcpy( p_box->data.p_name->psz_text, p_peek, p_box->i_size - 8 );
 p_box->data.p_name->psz_text[p_box->i_size - 8] = '\0';
 …
 MP4_READBOX_EXIT( 1 );
}
```

Wrapping queries in methods to scan other code in the future

fabs@workstation:~/joern/

```
joern> def buffer_overlows(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))
      }
                  }
defined function buffer_overflows
joern> buffer_overlows(cpg) // run the script
```

A “p_block->ibuffer == MAX_UINT64 causes an overflow in this method”

fabs@workstation:~/joern/

```
joern> buffer_overlows(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 resiliant 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';
    }
```

Use Contributed Scripts and send a PR to Get Yours Included

fabs@workstation:~/joern/

```
joern> scripts
res22: List[ScriptManager.ScriptDescription] = List(
  ScriptDescription(
    "ast-for-funcs",
    "Returns the corresponding AST for each function as Json object."
  ),
  ScriptDescription(
    "ast-for-funcs-dump",
    "Prints the corresponding AST for each function as Json string to a file."
  ),
  ScriptDescription(
    "cfg-for-funcs",
    "Returns the corresponding CFG for each function as Json object."
  ),
 ...
joern> runScript("cfg-for-funcs", cpg)
```

Key Deficiencies of the Query Language addressed by Dork

- From the compiler/runtime perspective, all nodes had the same type. Their logical types were encoded in a string only =>
- For all node types, the user needed to know and memorize
  - which fields contain meaningful information
  - which steps can be taken from the node (e.g., it makes sense to traverse from a method to its parameters, but not from a local to its parameters)
- Steep learning curve and difficult to implement completion.
- Bottom line: developing queries was cumbersome

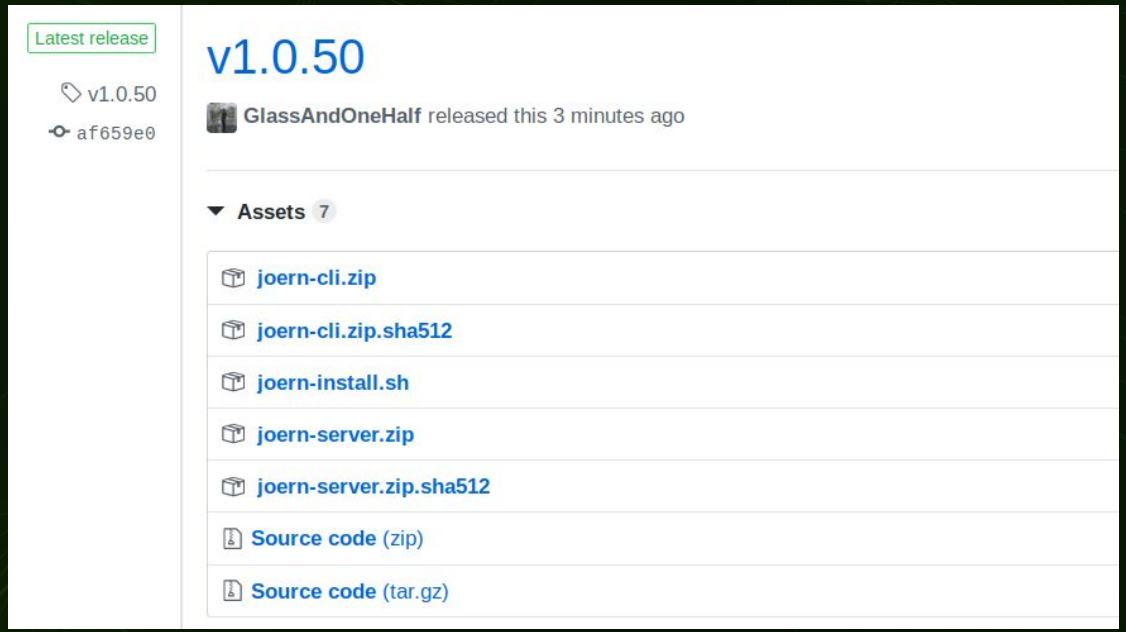
More “Integration”: Python Scripting via joernd

- joernd is a REST server (HTTP) that allows you to create projects, run queries, and read back results
- cpgclientlib is a thin Python library that communicated with this server => Joern can be scripted with Python
- You can add support for other languages by creating corresponding thin libraries

They say “on-prem” now

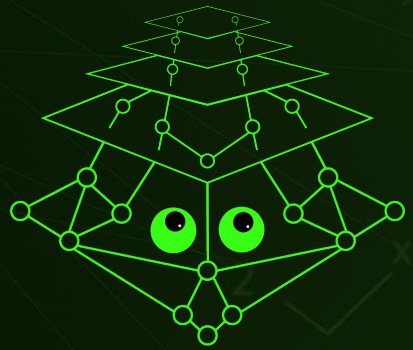
- Cloud is good for ordering Pizza, not for keeping your data
- Security begins by not giving your code, queries, scripts, and knowledge to other people (or companies) unless you have to
- Joern is a program you install on your computer, not on Amazon’s or Microsoft’s computer - it’s a component that you can install as you see fit
- If you want to share your scripts, you can, but you are not sharing them as you type
- Joern does not phone home to report “metrics” because this field is called “Security”.

Build from source or download binary distribution

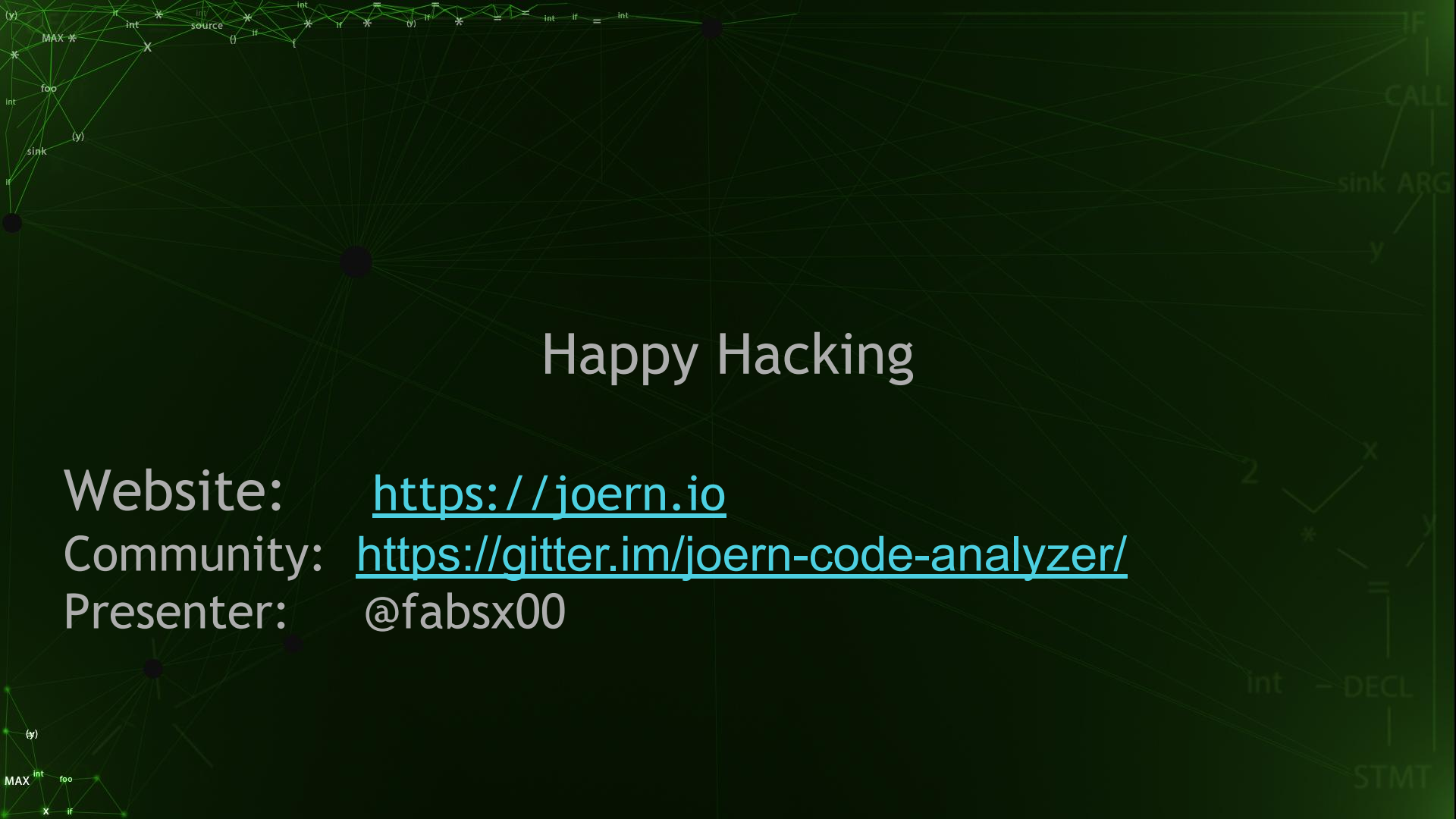


Concluding Remarks

- Tools for vulnerability discovery will only move the needle if they benefits the larger hacker community
- If you can’t download and immediately use it, it doesn’t exist
- The “market” (people in security who actually read code) is too small, which is why it has received only little good tooling.
- Wherever you work, help us push code auditing to the next level, run Joern - on your own computer - unwatched by and independent of the large corporations that form our “industry”. Tell us how the tool can be improved and share queries as you wish - and keep your 0day.



JOERN


Happy Hacking
Website: https://joern.io
Community: https://gitter.im/joern-code-analyzer/
Presenter: @fabsx00

Truncation of 32 bit platforms - in seek operation. Probably endless loop

fabs@workstation:~/joern/

```
joern> cpg.method.name("Read").filter(_.file.name(".*stream_extractor.*")).l
res56: List[Method] = List(
  Method(
    id -> 1346660L,
    name -> "Read",
    fullName -> "Read",
    //     -> that size_t is 32 bit on 32 bit platforms
    signature -> "static ssize_t(stream_extractor_t *,void *,  size_t)",
  )
)

// Caller
joern> cpg.method.name("archive_skip_decompressed").dump
res65: List[String] = List(
  """static int archive_skip_decompressed( stream_extractor_t* p_extractor,
uint64_t i_skip ) /* <=== */
{
    while( i_skip )
    {
        ssize_t i_read = Read( p_extractor, NULL,  i_skip );
```