Graph Mining for Vulnerability Discovery

Konrad Rieck
Computer Security Group
University of Göttingen
University of Göttingen

Göttingen
117,000 citizens
31% students

Rennes
201,000 citizens
31% students
› **Konrad Rieck**

› Fun with security and machine learning for 10 years

› Research group at the University of Göttingen

› **Research focus: intelligent security systems**

› Intrusion detection, malware analysis, vulnerability discovery
Vulnerabilities in Software

 › Vulnerabilities — A root cause for security breaches

 › 02/2014: Security flaw in Apple’s TLS/SSL code
   *All Apple devices vulnerable to MITM attacks*

 › 04/2014: Security flaw in OpenSSL library
   *Memory readable on millions of Internet servers*

 › 09/2014: Security flaw Unix shell Bash
   *Millions of web services vulnerable via CGI*

 › 07/2015: Security flaw in Android platform
   *Remote code execution on majority of devices*
Finding Vulnerabilities

› Discovery of vulnerabilities far from trivial
  › Some low-hanging fruits (*strcat*, *strcpy*, *sprintf*, …)
  › More often subtle errors in programming

› Current strategies for discovery of vulnerabilities in code
  › Testing and fuzzing of implementations
  › Taint analysis and symbolic execution
  › ... still many bugs only discovered by manual auditing

‡ Fully automated discovery impossible (Rice's theorem)
Finding Vulnerabilities

Vulnerabilities in code

Researchers
Finding Vulnerabilities

Vulnerabilities in code

Researchers

Our focus
Our Research Focus

- **Methods to make vulnerability discovery more effective**
  - *No tools for monkeys:* supporting, not replacing the analyst
  - Address different scenarios encountered during analysis
  - If the method is not practical, we do not care about it

- **Main concept:** Assisted discovery of vulnerabilities

- **Analysis supported by data mining and machine learning**
  - Augment view of analyst and help her save time
  - Suggest interesting code and guide auditing
Our Approaches

› **Vulnerability Extrapolation** *(ACSAC 2012)*
  › Finding code similar to a known vulnerability

› **Missing-Check Detection** *(CCS 2013)*
  › Discovery of missing and faulty security checks

› **Code Property Graphs** *(IEEE S&P 2014)*
  › Mining for vulnerabilities using graph databases

› **Taint-Style Vulnerabilities** *(IEEE S&P 2015)*
  › Detection patterns for common vulnerabilities
Code Property Graphs

Vulnerability in LibSSH2

[...]
if (channelp) {
    /* set signal name (without SIG prefix) */
    uint32_t namelen =
        _libssh2_ntohu32(data + 9 + sizeof("exit-signal"));
    channelp->exit_signal =
        LIBSSH2_ALLOC(session, namelen + 1);
    [...]
    memcpy(channelp->exit_signal,
           data + 13 + sizeof("exit_signal"), namelen);
    channelp->exit_signal[namelen] = '\0';
    [...]
}
[...]

Discovered by Stefan Esser (SyScan'13)
Vulnerability in LibSSH2

```c
[...]
if (channelp) {
    /* set signal name (without SIG prefix) */
    uint32_t namelen =
        _libssh2_ntohu32(data + 9 + sizeof("exit-signal"));
    channelp->exit_signal =
        LIBSSH2_ALLOC(session, namelen + 1);
    [...]
    memcpy(channelp->exit_signal,
           data + 13 + sizeof("exit_signal"), namelen);
    channelp->exit_signal[namelen] = '\0';
    [...]
}
[...]
```

Discovered by Stefan Esser (SyScan’13)
Vulnerability in LibSSH2

Discovered by Stefan Esser (SyScan’13)

```c
if (channelp) {
    /* set signal name (without SIG prefix) */
    uint32_t namelen = _libssh2_ntohu32(data + 9 + sizeof("exit-signal"));
    channelp->exit_signal = LIBSSH2_ALLOC(session, namelen + 1);
    [...]
    memcpy(channelp->exit_signal,
           data + 13 + sizeof("exit_signal"), namelen);
    channelp->exit_signal[namelen] = '\0';
    [...]
}
[...]
```
Vulnerability in LibSSH2

```c
... if (channelp) {
  /* set signal name (without SIG prefix) */
  uint32_t namelen = _libssh2_ntohu32(data + 9 + sizeof("exit-signal"));
  channelp->exit_signal = LIBSSH2_ALLOC(session, namelen + 1);
  [...]
  memcpy(channelp->exit_signal,
         data + 13 + sizeof("exit_signal"), namelen);
  channelp->exit_signal[namelen] = '\0';
  [...]
}
...```

Discovered by Stefan Esser (SyScan'13)
Q: How did Stefan Esser find the bug?
   - Black-box and white-box fuzzing? Nope.
   - Dynamic taint tracking and symbolic execution? Nope.

A: A regular expression for grep!

```regex
ALLOC[A-Z0-9_]*\s*
\(([^[^,]*[^;]+[^\^[^>]*[^;]*[^)])\s*;
```
Q: How did Stefan Esser find the bug?
   - Black-box and white-box fuzzing? Nope.
   - Dynamic taint tracking and symbolic execution? Nope.

A: A regular expression for grep!

```
ALLOC[A-Z0-9_]*\s*
\(([^,]*,[^;]*[*+-][^>][^;]*\])\s*;
```
Vulnerability Discovery in Practice

› Expert knowledge key to discovery of vulnerabilities
  › Not necessary need for specialized auditing tools
  › Simple search tools often sufficient, e.g. grep
  › Encoding of expert knowledge as search query

› Our example
  › Knowledge: “Math risky in allocation functions”
  › Query: ALLOC[A-Z0-9_]*\s*\([\^,]*,\[^;]*\)[*+-][^>][^;]*\s*;
A Generic Framework

› Framework for searching vulnerabilities
  › Design of a code database for vulnerability discovery
  › Modeling vulnerabilities with queries for the system
  › Link between automatic analysis and expert knowledge

› Vision of my PhD student Fabian Yamaguchi…

Source code

Analysis

Database

Expert

Parsing

Control flow

Data flow

Query
What do we need?

- Comprehensive view on program code
  - Parsing *Syntactical analysis*
    - How does the program code look like?
  - Control flow *Control-flow analysis*
    - How is the program code executed?
  - Data flow *Data-flow analysis*
    - How is data processed by the code?
Abstract Syntax Trees (AST) •

› Representation of syntax and language constructs

› **Nodes**: Statements, declarations, calls, operators, ...

› **Edges**: nesting of language constructs

**Source code**

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

**AST**

![Diagram of an Abstract Syntax Tree (AST) for the source code provided. The diagram shows the nodes and edges representing the syntax and nesting of language constructs.](image-url)
Control-Flow Graph (CFG)

› Representation of code logic and execution
  › Nodes: statements and conditions
  › Edges: conditional flow of control
  › Derivation from nodes of AST

Source code

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

CFG

ENTRY

ε

int x = source()

ε

if (x < MAX)

true

int y = 2 * x

ε

false

sink(y)

ε

EXIT
Program Dependence Graph (PDG)

- **Representation of data flow and dependencies**
  - **Nodes**: statements and conditions
  - **Edges**: Control (C) and data (D) dependencies
  - Derivation from nodes and edges of CFG

**Source code**

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

```
int x = source()
if (x < MAX)
int y = 2 * x
sink(y)
```
Vulnerabilities often reflected in all representations

“… find call X reachable from Y processing Z …”

Idea: Merge AST/CFG/PDG in a combined representation

- Graph structure with shared nodes and edges
- Possibility to jump back and forth between views
- Basis for powerful search queries
Meet the **Code Property Graph**

- Nodes from AST & edges from AST, CFG and PDG
- Edge-labeled multi-graph with properties attached to nodes
- … or short: property graph
Meet the **Code Property Graph**

- Nodes from AST & edges from AST, CFG and PDG
- Edge-labeled multi-graph with properties attached to nodes
- … or short: property graph

Example: Call to sink reachable if \( x < \text{MAX} \) processing \( x \)
Queries for Graphs

- **Queries modelled as traversals in code property graph**
  - Inspiration from modern graph databases
  - *Traversals* = Walk from one set of nodes to another
  - Walk based on edge labels and node properties

- **Implementation with Gremlin (and Cypher)**
  - Query languages for graph databases
  - Supported by common databases, such as Neo4J
Chaining Traversals

- **Chaining of traversals by function composition**
  - Input and output domain identical
  - Complex queries based on chain of simple traversals

- **Idea:** Construct utility traversals for vulnerability discovery
  - Modelling of common analysis steps

- **Example:** A match traversal
  - All AST nodes below nodes $X$ satisfying predicate $p$
  - $\text{Match}_p(X) = \text{Filter}_p \circ \text{TNodes}(X)$
Example: LibSSH2 Bug

- **Implementation as chain of traversals**
  - Extract all function calls & find all calls to `malloc`
  - Extract all arguments & find all *first* arguments
  - Check for math operations in first argument

- **Gremlin code for querying the code property graph**

  ```java
  getArguments('malloc', '0').astNodes().filter{it.type == 'AdditiveExpression'}
  ```

  *NOT BAD.*
Example: Overflows in Linux Kernel

› Query for hunting overflows in write handlers
  › Find all calls to `memcpy` with argument called `count`
  › Find all unsanitized data flows from variable `count`

› Gremlin code for this traversal

```plaintext
getFunctionASTsByName('*_write*')
  .getArguments('memcpy', '2')
  .unsanitized(
      { it._().or(
          _().isCheck('.*' + paramName + '.*'),
          _().codeContains('.*alloc.*' + paramName + '.*'),
          _().codeContains('.*min.*')
      )
    )
  .param( '.*c(ou)?nt.*' )
```

Source: variable from userspace  
Sink: `memcpy` in write functions

Check for sanitization
Are code property graphs and traversals really helpful?

Evaluation with security expert from industry
- Nico Golde from Qualcomm
- Audit of internal and Linux kernel code

Analysis of past Linux vulnerabilities
- Joint design of five traversals for common flaws
- Types: buffer overflows \(2\), zero-byte allocation \(1\), memory mapping bugs \(1\), memory disclosure \(1\)
### Coverage of Vulnerabilities

<table>
<thead>
<tr>
<th>Vulnerability types</th>
<th>Code representations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AST</td>
</tr>
<tr>
<td>Memory Disclosure</td>
<td></td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>(✓)</td>
</tr>
<tr>
<td>Resource Leaks</td>
<td></td>
</tr>
<tr>
<td>Design Errors</td>
<td></td>
</tr>
<tr>
<td>Null Pointer Dereference</td>
<td></td>
</tr>
<tr>
<td>Missing Permission Checks</td>
<td>✓</td>
</tr>
<tr>
<td>Race Conditions</td>
<td></td>
</tr>
<tr>
<td>Integer Overflows</td>
<td></td>
</tr>
<tr>
<td>Division by Zero</td>
<td>✓</td>
</tr>
<tr>
<td>Use After Free</td>
<td></td>
</tr>
<tr>
<td>Integer Type Issues</td>
<td></td>
</tr>
<tr>
<td>Insecure Arguments</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Analysis of all vulnerabilities in the Linux kernel in 2012**

- 10 out of 12 types covered by coder property graphs
### 18 unknown vulnerabilities in the Linux kernel

<table>
<thead>
<tr>
<th>Type</th>
<th>Location</th>
<th>Developer Feedback</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer Overflow</td>
<td>arch/um/kernel/exitcode.c</td>
<td>Fixed</td>
<td>CVE-2013-4512</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>drivers/staging/ozwpan/ozcdev.c</td>
<td>Fixed</td>
<td>CVE-2013-4513</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>drivers/s390/net/qeth_core_main.c</td>
<td>Fixed</td>
<td>CVE-2013-6381</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>drivers/staging/wlags49_h2/wl_priv.c</td>
<td>Fixed</td>
<td>CVE-2013-4514</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>drivers/scsi/megaraid/megaraid_mm.c</td>
<td>Fixed</td>
<td></td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>drivers/infiniband/hw/ipath/ipath_diag.c</td>
<td>Fixed</td>
<td></td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>drivers/infiniband/hw/qib/qib_diag.c</td>
<td>Fixed</td>
<td></td>
</tr>
<tr>
<td>Memory Disclosure</td>
<td>drivers/staging/bcm/Bcmchar.c</td>
<td>Fixed</td>
<td>CVE-2013-4515</td>
</tr>
<tr>
<td>Memory Disclosure</td>
<td>drivers/staging/sb105x/sb_pci_mp.c</td>
<td>Fixed</td>
<td>CVE-2013-4516</td>
</tr>
<tr>
<td>Memory Mapping</td>
<td>drivers/video/au1200fb.c</td>
<td>Fixed</td>
<td>CVE-2013-4511</td>
</tr>
<tr>
<td>Memory Mapping</td>
<td>drivers/video/au1100fb.c</td>
<td>Fixed</td>
<td>CVE-2013-4511</td>
</tr>
<tr>
<td>Memory Mapping</td>
<td>drivers/uio/uio.c</td>
<td>Fixed</td>
<td>CVE-2013-4511</td>
</tr>
<tr>
<td>Memory Mapping</td>
<td>drivers/staging/.../drv_interface.c</td>
<td>Fixed</td>
<td></td>
</tr>
<tr>
<td>Memory Mapping</td>
<td>drivers/gpu/drm/i810/i810_dma.c</td>
<td>Fix underway</td>
<td></td>
</tr>
<tr>
<td>Zero-byte Allocation</td>
<td>fs/xfs/xfs_ioclt1.c</td>
<td>Fixed</td>
<td>CVE-2013-6382</td>
</tr>
<tr>
<td>Zero-byte Allocation</td>
<td>fs/xfs/xfs_ioclt32.c</td>
<td>Fixed</td>
<td>CVE-2013-6382</td>
</tr>
<tr>
<td>Zero-byte Allocation</td>
<td>drivers/net/wireless/libertas/debugfs.c</td>
<td>Fixed</td>
<td>CVE-2013-6378</td>
</tr>
<tr>
<td>Zero-byte Allocation</td>
<td>drivers/scsi/aacraid/commctrl1.c</td>
<td>Fixed</td>
<td>CVE-2013-6380</td>
</tr>
</tbody>
</table>

... and an email from Linus Torvalds.
18 unknown vulnerabilities in the Linux kernel

<table>
<thead>
<tr>
<th>Type</th>
<th>Location</th>
<th>Developer Feedback</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer Overflow</td>
<td>arch/um/kernel/exitcode.c</td>
<td>Fixed</td>
<td>CVE-2013-4512</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>drivers/staging/ozwpan/ozcdev.c</td>
<td>Fixed</td>
<td>CVE-2013-4513</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>drivers/s390/net/qeth_core_main.c</td>
<td>Fixed</td>
<td>CVE-2013-6381</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>drivers/staging/wlags49_h2/wl_priv.c</td>
<td>Fixed</td>
<td>CVE-2013-4514</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>drivers/scsi/megaraid/megaraid_mm.c</td>
<td>Fixed</td>
<td></td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>drivers/infiniband/hw/ipath/ipath_diag.c</td>
<td>Fixed</td>
<td></td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>drivers/infiniband/hw/qib/qib_diag.c</td>
<td>Fixed</td>
<td></td>
</tr>
<tr>
<td>Memory Disclosure</td>
<td>drivers/staging/bcm/Bcmchar.c</td>
<td>Fixed</td>
<td>CVE-2013-4515</td>
</tr>
<tr>
<td>Memory Disclosure</td>
<td>drivers/staging/sb105x/sb_pci_mp.c</td>
<td>Fixed</td>
<td>CVE-2013-4516</td>
</tr>
<tr>
<td>Memory Mapping</td>
<td>drivers/video/au1200fb.c</td>
<td>Fixed</td>
<td>CVE-2013-4511</td>
</tr>
<tr>
<td>Memory Mapping</td>
<td>drivers/video/au1100fb.c</td>
<td>Fixed</td>
<td>CVE-2013-4511</td>
</tr>
<tr>
<td>Memory Mapping</td>
<td>drivers/uio/uio.c</td>
<td>Fixed</td>
<td>CVE-2013-4511</td>
</tr>
<tr>
<td>Memory Mapping</td>
<td>drivers/staging/.../drv_interface.c</td>
<td>Fixed</td>
<td></td>
</tr>
<tr>
<td>Memory Mapping</td>
<td>drivers/gpu/drm/i810/i810_dma.c</td>
<td>Fix underway</td>
<td></td>
</tr>
<tr>
<td>Zero-byte Allocation</td>
<td>fs/xfs/xfs_ioctl1.c</td>
<td>Fixed</td>
<td>CVE-2013-6382</td>
</tr>
<tr>
<td>Zero-byte Allocation</td>
<td>fs/xfs/xfs_ioctl32.c</td>
<td>Fixed</td>
<td>CVE-2013-6382</td>
</tr>
<tr>
<td>Zero-byte Allocation</td>
<td>drivers/net/wireless/libertas/debugfs.c</td>
<td>Fixed</td>
<td>CVE-2013-6378</td>
</tr>
<tr>
<td>Zero-byte Allocation</td>
<td>drivers/scsi/aacraid/commctrl1.c</td>
<td>Fixed</td>
<td>CVE-2013-6380</td>
</tr>
</tbody>
</table>

... and an email from Linus Torvalds.
Conclusions and Outlook
Finding vulnerabilities challenging and demanding
- Automatic approaches often fail due to complexity

Idea: Assisted discovery of vulnerabilities
- Guided auditing: Suggest interesting code to analyst
- Better modeling: Design tools specifically for bug hunting

Example: Code property graphs
- Blend of classic code analysis and graph mining
- Basis for finding 40 vulnerabilities in popular software
Next Steps

› Extension of code property graphs
  › Automatic construction of traversals (*IEEE S&P 2015*)
  › Incorporation of other programming languages, e.g. PHP
  › Incorporation of external resources, e.g. network data
  › Multi-layered view: *source code ~ IR ~ binary code*

› Open-source project “Joern”
  › Developed by Fabian Yamaguchi and Alwin Maier
  › [http://www.mlsec.org/joern](http://www.mlsec.org/joern)
Thanks! Questions?
Gremlin Examples

Code property graph

Graph traversals

> g.v(1).out('knows')
===>v[2]
===>v[4]
> g.v(1).out('knows').filter{it.age < 30}
===>v[2]
> g.v(1).out('knows').filter{it.age < 30}.name
===>vadas
> g.v(1).out.loop(1){it.loops < 3}
===>v[5]
===>v[3]
Defensive vs. Offensive Security

Defensive security

- Detection of attacks
- Analysis of attacks
- Prevention of attacks
Defensive vs. Offensive Security

**Defensive security**
- Detection of attacks
- Analysis of attacks
- Prevention of attacks

**Offensive security**
- Discovery of vulnerabilities
- Analysis of vulnerabilities
- Exploiting of vulnerabilities
Defensive vs. Offensive Security

Defensive security
- Detection of attacks
- Analysis of attacks
- Prevention of attacks

Offensive security
- Discovery of vulnerabilities
- Analysis of vulnerabilities
- Exploiting of vulnerabilities
Defensive vs. Offensive Security

Defensive security
- Detection of attacks
- Analysis of attacks
- Prevention of attacks

Offensive security
- Discovery of vulnerabilities
- Analysis of vulnerabilities
- Exploiting of vulnerabilities

Diagram illustrating the processes of defensive and offensive security.