Modeling and Discovering Vulnerabilities with Code Property Graphs

Fabian Yamaguchi, Nico Golde (Qualcomm), Daniel Arp, and Konrad Rieck
Security & Privacy 2014
Implementation Vulnerabilities

» Implementation vulnerabilities
  » Subtle mistakes in large code bases
  » Can have devastating effects
  » Landscape as diverse as code is

» Goal of our research
  » Develop effective methods to assist analysts in practice

» Let’s begin with an illustrative example.
Esser’s libssh2 Bug (Syscan’13)

Store attacker controlled data in 32 bit integer

```c
 [...] 1
 if (channelp) {
   /* set signal name (without SIG prefix) */
   uint32_t namelen =
     _libssh2_ntohu32(data + 9 + sizeof("exit-signal"));
   channelp->exit_signal =
     LIBSSH2_ALLOCP(session, namelen + 1);
   [...] 7
   memcpy(channelp->exit_signal,
           data + 13 + sizeof("exit_signal"), namelen);
   channelp->exit_signal[namelen] = '\0';
   [...] 10
 } 14
 [...] 14
```
Esser’s libssh2 Bug

Summation inside allocation

```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';
    […]
}
[…]```

Heap-based buffer overflow

```c
[...]
if (channelp) {
    /* set signal name (without SIG prefix) */
    uint32_t namelen =
        _libssh2_ntohu32(data + 9 + sizeof("exit-signal"));
    channelp->exit_signal =
        LIBSSH2_ALLOCF(session, namelen + 1);
    [...]
    memcpy(channelp->exit_signal,
           data + 13 + sizeof("exit_signal"), namelen);
    channelp->exit_signal[namelen] = '\0';
    [...]
}
[...]```
Q: How did he find the vulnerability?
   - Whitebox fuzzer enhanced with symbolic execution?
   - Machine learning approach for anomaly detection?
   - Theorem proving? Model checking?
Q: How did he find the vulnerability?
   
   » Whitebox fuzzer enhanced with symbolic execution?
   
   » Machine learning powered anomaly detector?
   
   » Theorem proving? Model checking?

A: regular expression for grep.

`'ALLOC[A-Z0-9_]\s*\([\^,]*,[\^;]*[\^+-][\^>][\^;]*\)\s*;'`
What I think this should tell us

» If used right, even primitive tools like grep are powerful

  *Allow expert knowledge to guide the analysis!*

» Query is a simple model making expert knowledge explicit

» High false positive rates are often tolerated in practice
This talk – Modeling and Discovering Vulnerabilities

» Design of a code mining system for vulnerability discovery

» Modeling vulnerabilities with queries for the system

» Bringing together code analysis and graph databases
Aspects of code a query should be able to model

» Approach: studied all vulnerabilities resulting in CVEs for the Linux Kernel in 2012

» We want to encode…
  » syntax (“What does it look like?”)
  » control flow (“Can we get from one statement to another?”)
  » data flow (“where is attacker-controlled data used?”)

» Compiler construction deals with all of these aspects. Let’s see how.
Language awareness: Abstract Syntax Trees (ASTs)

- Fine-grained, hierarchical decomposition of code into its language constructs

- **Nodes:** “Call”, “Declaration”, “AdditiveExpression”, ...

- **Edges:** nesting of language constructs
Statement interplay: Control Flow Graphs (CFGs)

- **Nodes**: statements and conditions
- **Edges**: conditional control flow

“Which sequences of statements can be executed in order?”

```c
void foo()
{
    int x = source();
    if (x < MAX)
    {
        int y = 2 * x;
        sink(y);
    }
}
```
Dependencies: Program Dependence Graph (PDG)

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

» **Nodes**: statements and conditions

» **Control dependency edges**: “What conditions need to hold for a statement to be executed?”

» **Data dependency edges**: “Where is the variable defined? Where is it used?”
A combined representation

» Typical query:

Find a call to foo in a statement where data flow exists to a statement that contains a call to bar.

» Creates the need to transition from one graph representation to the other. (AST -> PDG -> AST in the example)

» Idea: merge AST, CFG and PDG at statement nodes
» Use a property graph:
A property graph is a graph with labeled edges and key-value pairs attached to nodes and edges
» Vulnerabilities can be described as “traversals” in this graph
Graph Databases and Traversals

» Informal: A traversal is a description of how to walk the graph from a set of start nodes to reach nodes that you are interested in

Example:

```plaintext
>> {1}.out('knows')
vadas, josh
>> {1}.out('knows').filter{ it.age > 30}
josh
```

Image from: http://markorodriguez.com/2011/08/03/on-the-nature-of-pipes/
Formally

\[ \mathcal{T} : \mathcal{P}(V) \to \mathcal{P}(V) \]

\[ \text{OUT}(X) = \bigcup_{v \in X} \{ u : (v, u) \in E \} , \]

\[ \text{FILTER}_p(X) = \{ v \in X : p(v) \} \]
Traversals can be chained via function composition!

A chain of traversals is again a traversal

Idea: build utility traversals for vulnerability discovery

Build search ‘templates’ from utilities

Image from: http://markorodriguez.com/2011/08/03/on-the-nature-of-pipes/
Example: Syntax-only Vulnerability Descriptions

- Description based entirely on (what used to be) the AST

- Primitive traversals:
  \[ \text{TNodes}(V) \]
  - All nodes of the subtrees rooted at any \( v \) in \( V \).

  \[ \text{Match}_p(V) \]
  - All nodes matching predicate \( p \) contained in a subtree rooted at any \( v \) in \( V \).

\[ \text{Match}_p(V) = \text{Filter}_p \circ \text{TNodes}(V) \]
Example: Summations inside arguments to malloc

» A chain of $\text{MATCH}_p(V)$ traversals:
  » For all function nodes, traverse to all calls to malloc
  » For nodes in resulting set, traverse to all arguments
  » From there, traverse to all additive expressions

» Example query:

```javascript
getArguments('malloc', '0').astNodes().filter{it.type == 'AdditiveExpression'}
```
A syntax-only description:

- Provide a set of match traversals that MUST match!
- Provide a set of match traversals that MUST NOT match!

Insufficient for the description of most vulnerabilities

- No data flow description
- No control flow description
"Unsanitized paths to"-primitive: $\text{UNSANITIZED}_M(X)$

Starting from a sink, find unsanitized paths in the control flow graph from a source.

$$\text{ARG}^{1}_{\text{get\_user}} \circ \text{UNSANITIZED}\{T_s\} \circ \text{ARG}^{3}_{\text{memcpy}}$$

Syntax Only Description for source
Syntax Only Descriptions for sanitizers
Syntax Only Description For sink
Example: Query for buffer overflows in write handlers

```python
query1 = """

getFunctionASTsByName("*_write*"
    .getArguments('((copy_from_user OR memcpy)', '2')
    .filter{ it.code.matches('c(ou)?nt') }

    .unsanitized(
        { it._().or(
            _().isCheck('.*' + paramName + '.*'),
            _().codeContains('.*alloc.*' + paramName + '.*'),
            _().codeContains('.*min.*')
        )}
    )

    .param( '.*c(ou)?nt.*' )
"""

» Profit: 4 previously unknown buffer overflows in 5 hits!
Practical Evaluation

» Get somebody working in the industry to actually try this!
  » Found about 100 issues in an internal audit at Qualcomm with 9 expert queries

» Evaluation for paper: audit of the Linux Kernel Mainline
  » Buffer overflows (2 queries)
  » Zero-byte allocation (1 query)
  » Memory mapping bugs (1 query)
  » Memory disclosure (1 query)
### Identified (previously unknown) Vulnerabilities

<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/commctrl.c</td>
<td>Fixed</td>
<td>CVE-2013-6380</td>
</tr>
</tbody>
</table>

» 18 vulnerabilities, acknowledged /fixed by developers
### Analysis: All Linux Kernel Vulnerabilities 2012

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

10 of 12 of these groups can be expressed well as traversals in the code property graph.
» Introduced a system to mine code bases for vulnerabilities

» Brings together classic intermediate representations from compiler construction and graph databases

» Traversals in the code property graph as search queries

» Demonstrated to be effective in practice
Thank you! Questions?

» Implementation
  » [http://mlsec.org/joern](http://mlsec.org/joern) (webpage + documentation)
  » [http://github.com/fabsx00/joern](http://github.com/fabsx00/joern) (main system)
  » [http://github.com/fabsx00/python-joern](http://github.com/fabsx00/python-joern) (traversals)

» Fabian Yamaguchi
  » fabs@goesec.de
  » [http://codeexploration.blogspot.de](http://codeexploration.blogspot.de) (my webpage)
  » Twitter: @fabsx00