Dynamic Invariant Detection
Robust Signatures for Kernel Data Structures

Outline

1. Dynamic Invariant Detection
   - Concepts
   - Daikon

2. Robust Signatures for Kernel Data Structures
   - Background
   - Robust Signatures

what is program invariant?

- refer to a *property* that holds at a certain point or points in a program
  - being constant $x = a$, non-zero $x \neq 0$, ordering $x \leq y$, being in a range $a \leq x \leq b$, containment $x \subseteq y$
  - linear relationships $y = ax + b$, functions from a library $x = f(y)$
  - ...
- often used in assert statements, documentation and formal specifications
- also useful in program understanding and a host of other applications

example

```plaintext
i, s := 0, 0;
do i \neq n ->
i, s := i + 1, s + b[i]
od

Precondition: $n \geq 0$
Postcondition: $s = (\sum j : 0 \leq j < n : b[j])$
Loop invariant: $0 \leq i \leq n$ and $s = (\sum j : 0 \leq j < i : b[j])$
```
Dynamic Invariant Detection

- Expecting programmers to fully annotate code with invariants but programmers do not usually explicitly annotate or document code with invariants.

Daikon

- An implementation of dynamic detection of likely invariants.
- Uses machine learning techniques.
- Can detect invariants in C, C++, Java and Perl programs.
- Input is the program and output is a set of likely program invariants.

Robust Signatures for Kernel Data Structures

- Background
- Robust Signatures

Outline

1. Dynamic Invariant Detection
   - Concepts
   - Daikon
2. Robust Signatures for Kernel Data Structures

Architecture

- The original program
- The instrumenters

Architecture: original program

architecture: instrumenters

- Add instructions to the target program so that it will output the values of variables.
- Language dependent (C/C++, Java and Perl etc.)
architecture: instrumented program

Original program Instrumented program Data trace database Detect Invariants

print b, n;
i,s := 0,0;
do i != n ->
print i, s, n, b[i];
i,s := i + 1, s + b[i]
od

architecture: trace file

Original program Instrumented program Data trace database Invariants

ENTER
B = 92 56 -96 -49 76 92 -3 -88, modified
N = 8, modified
LOOP
B = 92 56 -96 -49 76 92 -3 -88, unmodified
N = 8, unmodified
I = 0, modified
s = 0, modified
LOOP
B = 92 56 -96 -49 76 92 -3 -88, unmodified
N = 8, unmodified
I = 1, modified
s = 92, modified
....

architecture: inference engine

Original program Instrumented program Data trace database Detect Invariants

read the trace data and produces likely invariants
adopt optimizations to reduce redundant invariant checking

equal variables: if
x = y, then for any invariant f, f(x) =⇒ f(y)
dynamically constant variables: if
x = 5 =⇒ odd(x)
and x ≥ 5

variable hierarchy: for two program points A and B, if all samples for B also appear at A =⇒ (invariants true at A =⇒ true at B)
suppression of weaker invariants: if
x > y =⇒ x ≥ y

architecture: invariants

Original program Instrumented program Data trace database Detect Invariants

1) n ≥ 0
2) s = SUM(B)
3) i ≥ 0

Java example

Java class StackAr

```
Object[] theArray; // Array that contains the stack elements, not final/abstract!
int topOfStack; // Index of top element. -1 if stack is empty.
```

methods:
void push(Object x) // Insert x
void pop() // Remove most recently inserted item
Object top() // Return most recently inserted item
Object topAndPop() // Remove and return most recently inserted item
boolean isEmpty() // Return true if empty; else false
boolean isFull() // Return true if full; else false
void makeEmpty() // Remove all items

output (1)

Object invariants for StackAr

```
this.theArray != null
this.theArray.getClass() == java.lang.Object[].class
this.topOfStack >= -1
this.topOfStack <= this.theArray.length - 1
this.theArray[0..this.topOfStack] elements != null
this.theArray[this.topOfStack+1..] elements == null
```
### Dynamic Invariant Detection

#### Concepts
- Daikon

### Robust Signatures for Kernel Data Structures

#### Background
- Robust Signatures

### Other Tools
- Agitator product by Agitar
- DIDUCE tool for Java programs
- IODINE tool for hardware designs
- SPIN model checker which checks if two variables are related by =, <, >, ≤ or ≥; the output is a graph with variables at the nodes and edges labeled by the comparison relations

### Summary
- A full-featured, scalable, robust tool for dynamic invariant detection
- Supports multiple programming languages
- Likely invariants that it produces have been used from refactoring to testing to data structure repair etc.

---

### Outline

#### Dynamic Invariant Detection
- Concepts
- Daikon

#### Robust Signatures for Kernel Data Structures
- Background
- Robust Signatures
Dynamic Invariant Detection

- Robust Signatures for Kernel Data Structures

**attacks**
- objective: conceal presence of an object from the user
- hooking: modify code, make the APIs lie
- DKOM: manipulate kernel data structures instead of code

**signature scans**
- some new tools (PTFinder, Volatility) use signature scans to find hidden objects in memory
- identify a set of invariants (signatures) for the object
- perform a linear scan through memory
- report regions of data where the fields match your invariants (signatures)

**signature evasion**
- assume attackers can read and modify any kernel data (like DKOM)
- attackers may modify some fields to hide objects from scanners without OS crashing
- false negative for a signature

**Outline**

1. Dynamic Invariant Detection
   - Concepts
   - Daikon
2. Robust Signatures for Kernel Data Structures
   - Background
   - Robust Signatures

**architecture**
- develop robust signatures that resist signature evasion
  - feature selection:
    - profile the data structure to determine which fields are most commonly accessed by the operating system
    - fuzz the most commonly accessed fields to determine which can be modified without OS crashing
  - signature generation based on the selected fields
Data structure profiling

**Idea:** Watch the structure as OS executes. If a field is never accessed, attackers can control it. Monitor OS execution and build a histogram of how frequently each field is accessed.

**EPROCESS profile**

- Always
- Never

Fuzzing

- Modify data fields and see if OS crashes.
  - If OS consistently crashes, attackers probably can't modify that field.
- During fuzz testing, failures are indications that a field is robust.

Fuzzing methodology

- For each field, replace its contents with test data from one of several classes:
  - **Zero**
  - **Random**
    - 24601
    - 15151
    - 7669
    - 9272
    - 2
    - 159
  - **Random Primitive**
  - **Random Aggregate**

Profiling is not enough

- Don't know if functionality depends on a given member being accessed.
- Being accessed is a necessary condition, but not sufficient.
- Ensure that the field cannot be arbitrarily modified.
Dynamic Invariant Detection

Robust Signatures for Kernel Data Structures

so far...

original signatures

<table>
<thead>
<tr>
<th>Field</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pdb.Header.Type</td>
<td>0x001</td>
</tr>
<tr>
<td>Pdb.Header.Size</td>
<td>0x18</td>
</tr>
<tr>
<td>ThreadListHead.Plink</td>
<td>Kernel</td>
</tr>
<tr>
<td>ThreadListHead.Blink</td>
<td>Kernel</td>
</tr>
<tr>
<td>Pdb.DirectoryTableBase[]</td>
<td>Aligned</td>
</tr>
<tr>
<td>WorkingSetLock.Event.Header.Type</td>
<td>0x01</td>
</tr>
<tr>
<td>WorkingSetLock.Event.Header.Size</td>
<td>0x04</td>
</tr>
<tr>
<td>AddressCreationLock.Event.Header.Type</td>
<td>0x01</td>
</tr>
<tr>
<td>AddressCreationLock.Event.Header.Size</td>
<td>0x04</td>
</tr>
</tbody>
</table>

after profiling...

<table>
<thead>
<tr>
<th>Field</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pdb.Header.Type</td>
<td>0x001</td>
</tr>
<tr>
<td>Pdb.Header.Size</td>
<td>0x18</td>
</tr>
<tr>
<td>ThreadListHead.Plink</td>
<td>Kernel</td>
</tr>
<tr>
<td>ThreadListHead.Blink</td>
<td>Kernel</td>
</tr>
<tr>
<td>Pdb.DirectoryTableBase[]</td>
<td>Aligned</td>
</tr>
<tr>
<td>WorkingSetLock.Event.Header.Type</td>
<td>0x01</td>
</tr>
<tr>
<td>WorkingSetLock.Event.Header.Size</td>
<td>0x04</td>
</tr>
<tr>
<td>AddressCreationLock.Event.Header.Type</td>
<td>0x01</td>
</tr>
<tr>
<td>AddressCreationLock.Event.Header.Size</td>
<td>0x04</td>
</tr>
</tbody>
</table>

after fuzz...

<table>
<thead>
<tr>
<th>Field</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pdb.Header.Type</td>
<td>0x001</td>
</tr>
<tr>
<td>Pdb.Header.Size</td>
<td>0x18</td>
</tr>
<tr>
<td>ThreadListHead.Plink</td>
<td>Kernel</td>
</tr>
<tr>
<td>ThreadListHead.Blink</td>
<td>Kernel</td>
</tr>
<tr>
<td>Pdb.DirectoryTableBase[]</td>
<td>Aligned</td>
</tr>
<tr>
<td>WorkingSetLock.Event.Header.Type</td>
<td>0x01</td>
</tr>
<tr>
<td>WorkingSetLock.Event.Header.Size</td>
<td>0x04</td>
</tr>
<tr>
<td>AddressCreationLock.Event.Header.Type</td>
<td>0x01</td>
</tr>
<tr>
<td>AddressCreationLock.Event.Header.Size</td>
<td>0x04</td>
</tr>
</tbody>
</table>

signature generation

- dynamic invariant detection
- infer constraints (invariants) on robust fields
- input: for each robust field, extract a list of the values for all processes found in the memory images (uninfected)
- output: constraints (invariants)

a list of constrains

<table>
<thead>
<tr>
<th>Field</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pdb.Header.Type</td>
<td>0x001</td>
</tr>
<tr>
<td>Pdb.Header.Size</td>
<td>0x18</td>
</tr>
<tr>
<td>ThreadListHead.Plink</td>
<td>Kernel</td>
</tr>
<tr>
<td>ThreadListHead.Blink</td>
<td>Kernel</td>
</tr>
<tr>
<td>Pdb.DirectoryTableBase[]</td>
<td>Aligned</td>
</tr>
<tr>
<td>WorkingSetLock.Event.Header.Type</td>
<td>0x01</td>
</tr>
<tr>
<td>WorkingSetLock.Event.Header.Size</td>
<td>0x04</td>
</tr>
<tr>
<td>AddressCreationLock.Event.Header.Type</td>
<td>0x01</td>
</tr>
<tr>
<td>AddressCreationLock.Event.Header.Size</td>
<td>0x04</td>
</tr>
</tbody>
</table>

results

<table>
<thead>
<tr>
<th>Signature</th>
<th>Image Type</th>
<th>FP</th>
<th>FN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clean</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Malicious</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Clean</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Malicious</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
conclusions

- existing signatures are vulnerable to evasion
- new systematic method for selecting features for data structure signatures
- constrains attackers' ability to evade signatures

references