Diff, Navigate, Audit

Three applications of graphs and graphing for security

Halvar Flake – Blackhat Briefings Las Vegas 2004
Overview
The talk consists of three parts:

i. Structural comparison of executable objects
   i. Porting symbolic information between disassemblies
   ii. Differing security updates

ii. Navigating through unknown binaries
   i. Restructuring of graphs for better visualisation
   ii. Navigation using “echo”
   iii. Heap visualisation

iii. Interprocedural aliased dataflow graphs
   i. Improving callgraph generation
   ii. Memory leaks and double-free’s
   iii. Integer arithmetic
Structural Comparison

Introduction

i. Security Patch Analysis became interesting to the mainstream with the first LSD DCOM bug
   i. Special situation: Existence of critical security problem public, but no details available

ii. Few organisations patch in the first 10-14 days
    i. Window of exposure between publication of security problem and fixing

iii. “Nonpublication of exploit details buys the customer extra time, because reverse engineering of security updates is hard”
Structural Comparison
Asymmetry between source recompilation and RE

i. Making a minor change and recompiling a program is easy

ii. The general assumption is that reverse engineering these changes is hard
   i. Reverse Engineer has to re-do all work, because he lost all results from previous disassembly
   ii. Reverse Engineer then has to compare the function’s logic, as many instructions will have changed due to optimisation

iii. Software Vendors and HLL-Virus Authors try to exploit this asymmetry
   i. The first want to buy time for customers
   ii. The second can create variants quickly/easily
Structural Comparison

Diff’ing executables is difficult

i. Small changes in the source code can trigger significant changes in the executable:
   i. Adding a structure member will change immediate offsets for all accesses to structure members after the change
   ii. Adding a few lines of code can produce radically different register assignments and lead to differing instructions
   iii. Changed sizes of basic blocks in one function can lead to code in unrelated functions changing (because of branch inversion)

ii. The overwhelming majority of changes in the binary are irrelevant
   i. Classical trade-off: More false positives or running the risk of a false negative?
Structural Comparison
Viewing a program as graph of graphs

i. Primarily one is interested in changes to program logic

ii. A program can be viewed by looking at two graphs:
   i. The callgraph which contains all functions and their relationships (A calls B etc.)
   ii. The individual function flowgraphs which represent the basic blocks of every function and how they are linked by conditional or unconditional branches

iii. The program logic is more or less encoded in these two graphs
   i. Adding a single if( ) in any function will trigger a change in it’s flowgraph
   ii. Changing a call to strcpy to a call to strncpy will change the callgraph
Structural Comparison
Detecting changes by comparing graphs

i. Program logic can be viewed as a callgraph with nodes representing the individual flowgraphs

ii. Comparing two executable based on these graphs will detect all logic changes

iii. The comparison should be false-positive-free:
   i. Only “real” changes to program logic should be detected
   ii. Compilers don’t usually change the program logic
   iii. Modern compilers can inline entire functions and do many more crazy things (thus the “should be” instead of “is”)

iv. The comparison will not be false-negative-free:
   i. Switching signedness of a type or changing constants and buffer sizes will go undetected

v. So how can two graphs of graphs be compared?
Structural Comparison
Comparing the graphs

i. Checking if two undirected graphs are isomorphic is NP -- Math-speak for: Finding out if two graphs are the same is damn expensive

ii. Problem is a lot less problematic if one can find “fixed points” – two nodes in the two graphs that are definitely the same

iii. When analyzing programs, entry points and names for imported functions are available, yielding a first set of “fixed points”

iv. More “fixed points” would be desirable – so what would be a decent heuristic to generate more of them?
Structural Comparison
Heuristic signatures for the functions

Simplistic signature for every function:
- Number of basic blocks
- Number of links
- Number of functions called from this function
Structural Comparison
Heuristic signatures for the functions: Basic Blocks

5 Nodes
Structural Comparison

Heuristics signatures for the functions: Links

6 Links
Structural Comparison
Heuristic signatures for the functions: Subcalls

6 subcalls

Signature: (5, 6, 6)
Structural Comparison
Finding more fixedpoints (1)

Signature Set of Binary A

Signature Set of Binary B
Structural Comparison

Finding more fixedpoints (2)

Signature Set of Binary A

12::19::7
2::2::1
11::17::2
5::7::0
1::0::1
2::2::1
22::36::4
39::66::33

Signature Set of Binary B

2::2::1
12::19::7
18::32::23
3::3::2
1::0::2
14::25::19
2::2::5
22::36::4

39::66::33
16::28::19
5::7::0
11::17::2
18::32::23
3::3::2
1::0::1
Structural Comparison

Finding more fixedpoints (3)

Signature Set of Binary A

Signature Set of Binary B
Structural Comparison
Finding more fixedpoints (4)

Signature Set of Binary A

12::19::7
2::2::1
11::17::2
5::7::0
1::0::1
2::2::1
22::36::4

2::2::5
3::3::2
14::25::19
1::0::2
1::0::1
1::0::2

Signature Set of Binary B

2::2::1
12::19::7
1::0::1

2::2::5
16::28::19
5::7::0

2::2::1
11::17::2
Structural Comparison
Finding more fixedpoints (5)

Signature Set of Binary A

12::19::7
2::2::1
5::7::0
1::0::1
2::2::1
22::36::4

2::2::5
3::3::2
14::25::19
1::0::2
1::0::2

Signature Set of Binary B

2::2::1
12::19::7
1::0::1

16::28::19
5::7::0

22::36::4
2::2::5
Structural Comparison
Finding more fixedpoints (6)

Signature Set of Binary A
- 2::2::1
- 14::25::19
- 2::2::1

Signature Set of Binary B
- 2::2::1
- 16::28::19
- 2::2::1
Structural Comparison

Heuristic signatures for the functions: More fixedpoints

In the next step, signatures that occur more than once in both binaries are eliminated by using the callgraph:
Structural Comparison
Heuristic signatures for the functions: More fixedpoints

In the next step, signatures that occur more than once in both binaries are eliminated by using the callgraph:
Navigating Binaries
Road Maps as analogy for programs

i. When driving in a car, a frequent problem is to get from the current location of the car to another location

ii. Similar problems have to be solved in program analysis:
   i. A patch for a problem changes a function somewhere deep in program logic and fixes a security problem
   ii. A (static) analysis tool detects a problem somewhere deep in program logic

iii. In both cases, the problem of finding a way from point A to point B has to be solved
   i. Why use textual representations such as “road A leads to road B” etc. for program analysis?
Navigating Binaries
The importance of visualisation

i. Programs are huge and not easily understood

ii. Most of the human brain is built to react to visual stimulus
   i. Humans are more suited to recognize food than to keep large graphs in their head
   ii. “Recognition” tasks are a lot faster than “memory” tasks, meaning that reading code dependencies is significantly slower than “seeing” them

iii. Many problems are data visualisation problems

iv. Good ways to visualise function dependencies can yield better understanding
Navigating Binaries
The differences between road maps and programs

i. Roads crossings have an outdegree of less than 4 usually
   i. Many functions call much more than just 4 subfunctions

ii. Road crossings have an indegree of less than 4 usually
   i. Many functions get called from a lot more than 4 subfunctions

iii. When driving with a car, one usually does not drive down many dead ends
   i. A subfunction call that does not lead to the desired location immediately will nonetheless be executed

⇒ Useful analogy, but use with care
Navigating Binaries
Restructuring callgraphs

i. Library functions (such as malloc()) tie together logically independent parts of the binary in the graph

ii. Removal of library functions should clean up the graph significantly

   i. Library functions are called from many locations, thus adding a significant number of edges

   ii. Library functions have callee’s grouped closer together even though no obvious logical connection exists

iii. Removing via blacklists is a bad idea

   i. Not generic enough

   ii. “Wrappers” will remain

Æ Removal of nodes dependent on in/outdegree
Navigating Binaries
Restructuring callgraphs
Navigating Binaries
Restructuring callgraphs
Navigating Binaries
Navigating using “echo”

i. Same concept as presented at BH Vegas 2002

ii. Set BPX on individual nodes (remove upon hit)

iii. Visualize the results in a graph
   i. Highlight all functions that have been hit
   ii. Highlight all basic blocks in a function that have been hit
   iii. Set breakpoints on all basic blocks that can not lead to a target location
   iv. Playback the path the program takes as a “movie”

⇒ Navigation can be significantly improved
Navigating Binaries
Heap Visualisation

i. Memory blocks can be interpreted as a directed graph, too
   i. An allocated block is a node
   ii. A pointer to another allocated block is an edge

ii. Many data structures can be easily visually identified:
   i. Linked lists, doubly linked lists etc.
   ii. Tree’s, BTree’s etc…
Interprocedural Dataflow Graphs

Introduction

i. Two of the most important questions when reading code are:
   i. Where does this value come from?
   ii. Where can this value go?

ii. Questions are hard to answer …
   i. … especially in an environment with pointers
   ii. … especially in an environment without types

→ We have neither in the binary

• Sample problems that can be tackled are:
  • Improving OOP callgraphs
  • Detecting memory leaks and double-free’s
  • Generating expressions to describe integer calculations
Interprocedural Dataflow Graphs

Improving OOP callgraphs

i. Virtual method calls are done via a vtable of function pointers

ii. The instructions will be something like

```
    mov    eax, [ecx] ; Load vtable ptr from “this”-pointer
    call   [eax + 0x0C] ; Call method
```

iii. If we can answer what values [ecx + 0] can contain, we can answer the question “what function is called” statically

iv. If we want a complete callgraph (e.g. for better navigation), we cannot ignore this question
Interprocedural Dataflow Graphs

Detecting possible memory leaks

Definition: A block of memory that has been allocated “leaks” iff all references to it are lost/overwritten without free() being called

→ This can be reformulated as a graph-theoretical question: Does every path through the data flow graph go through free() at least once?

→ Graph needs to be complete and dataflow analysis reasonably accurate
Interprocedural Dataflow Graphs
Detecting possible double free()'s

Definition: A “double-free” occurs iff a block of memory is passed into the function “free()” more than once

→ This can be reformulated as a graph-theoretical question: Is there a path through the dataflow graph that passes through free() more than once?

→ Graph needs to be complete and dataflow analysis reasonably accurate