Structural Comparison of Executable objects

Reverse Engineering changes between executable versions

Halvar Flake – DIMVA 2004

halvar.flake@sabre-security.com

Motivation

i. Reverse engineering multiple versions of essentially the same binary is oftentimes needed:
   i. Security review of sequential versions of the same piece of software
   ii. Analysis of multiple variants of the same high-level-language virus
   iii. Analysis of security updates (“patches”)

ii. Problem is asymetric:
   i. Changing a few lines of sourcecode and recompiling is comparatively easy
   ii. Reverse engineering is harder: Function names have to be recovered, then functions have to be read and the change detected
   iii. Both HLL-Virus authors and software vendors try to exploit this asymetry to their advantage

Structural Comparison

Diff'ing executables is difficult

Why not use something like DIFF ?

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 behind the new member
    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 ?

A structural approach

i. Standard source-code "diff"-techniques can't be applied:
   i. Register allocation can / will change
   ii. Arrangement of basic blocks and branch directions can change
   iii. Depending on optimization, compilers can decide to use different instructions

        (e.g. lea eax, [eax + 10] vs add eax, 10)
ii. Addresses of global symbols change
iii. Filtering unwanted changes requires very CPU-specific implementation

➔ A more general approach is presented in this talk, focusing primarily on structural properties of an executable

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 is encoded a callgraph with nodes being the individual function flowgraphs

ii. Comparing two executable based on these graphs will detect 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

iv. The comparison will not be false-negative-free:

   i. Switching signedness of a type or changing constants and buffer sizes will go undetected

   ii. This is neglectable in many cases

v. So how can two graphs of graphs be compared ?

An executable as "Graph of Graphs"

An executable consists of:

$$F := \{f_1, \ldots, f_n\}$$

which are nodes of a digraph, the *callgraph* of an executable (edges imply calls-to relation)

Every function $f_i \in F$ can itself be viewed as a digraph, the *function flowgraph*.

➔ Executable is a *graph of graphs*

An executable as "Graph of Graphs"

Statically generating a callgraph is not always trivial:

- Calls via function pointers can not be always statically resolved
- Calls via OS-dependent functions (e.g. CreateThread()) can not always be statically resolved
- Calls via indirection through compiler-generated structures such as *vtables* for virtual C++ methods can not always be resolved statically

Luckily, calls that cannot be resolved in one variant of the binary are unlikely to be resolved in the other

Structural Matching

Consider executables A and B and their callgraphs:

$$\mathfrak{A} := \{\{a_1, \ldots, a_n\}, \{a_1^e, \ldots, a_m^e\}\}$$
$$\mathfrak{B} := \{\{b_1, \ldots, b_l\}, \{b_1^e, \ldots, b_k^e\}\}$$

We want to construct an isomorphism

$$p : \{a_1, \ldots, a_n\} \rightarrow \{b_1, \ldots, b_m\}$$

In the general case, this isomorphism does not exist because the cardinalities of the two sets are not necessarily identical

Iterative construction of the partial isomorphism

An initial mapping is created:

$$A_1 \subset \{a_1, \ldots, a_{n'}\}$$
$$B_1 \subset \{b_1, \ldots, b_{n'}\}$$
$$p_1 : A_1 \rightarrow B_1$$

This mapping is used to create sequence of mappings: $p_2, \ldots, p_h$

$$A_1 \subset A_2 \subset \cdots \subset A_h$$
$$B_1 \subset B_2 \subset \cdots \subset B_h$$

A Simple matching heuristic

- Comparing individual flowgraphs initially would be too expensive
- Heuristic is used: Every function is associated with 3-tuple: $(\alpha_i, \beta_i, \gamma_i)$

α_i := Number of basic blocks

β_i := Number of edges in flowgraph

γ_i := Number of edges originating at this node in the callgraph

A Simple matching heuristic

The initial mapping is created by associating functions under the following conditions

- Both functions have the same 3-tuple
- No other functions with the same 3-tuple exist in both sets

(Additional initial matches can be generated by using the names of functions (if available, e.g. in the case of dynamically linked functions))

Improving the initial mapping (I)

Only a small number of functions will be mapped initially.

- Smaller functions are less likely to be mapped as the propability for a “collision” of the signature increases
- Smaller subsets to be matched by this heuristic will produce better matches as fewer collisions occur

Improving the initial mapping (II)

Construct better isomorphism p_i from p_{i-1}

1. Take a_i and $p_{i-1}(a_i)$
2. Let A'_i be the set of all functions that have are called by a_i and B'_i be the set of all functions that are called by $p_{i-1}(a_i)$
3. Construct $p'_i : A'_i \to B'_i$ from A'_i , B'_i in the same way p_1 was constructed from the larger sets
4. If $a_j \notin A_{i-1}$ and a new match was constructed, $p_i(a_j) := p'_i(a_j)$ otherwise let $p_i(a_j) := p_{i-1}(a_j)$ if $a_j \in A_{i-1}$
5. A_i and B_i are the domain and image of p_i

Practical implementation

- Based on IDA Pro as a plugin
- Can deal with x86, MIPS so far
- Additional platforms are normally simple to add (exception: Platforms with speculative execution)
- PPC and SPARC are planned
- Extra code for attempting to “highlight” changes in the graph (very broken heuristics though)
- Additional “heuristic” matches in the isomorphism: Treat 3-tuple as coordinates, if euclidian distance is smaller than threshold attempt to match as well

Practical results

IIS SSL/PCT parser

- Updated schannel.dll
- Information from the security bulletin:
  - Possible remote compromise
  - Flaw in PCT parsing (PCT is a legacy protocol obsoleted by TLS)
  - No technical details about whereabouts etc.
- Only one function with a “PCT” in the name changes

➔ Change is an added range check to prevent a simple stack overflow

Practical results

MSASN1.DLL bugs

- Information from the security bulletin:
  - Integer wrap leading to compromise
  - No technical details about whereabouts etc.
- Changes in ASN1DECAlloc, ASN1DecRealloc
- Prevent integer overflows in the allocation functions
- Additional changes to prevent memory leaks

Practical results

MSASN1.DLL bugs

- Information from the security bulletin:
  - Integer wrap leading to compromise
  - No technical details about whereabouts etc.
- Changes in ASN1DECAlloc, ASN1DecRealloc
- Prevent integer overflows in the allocation functions
- Additional changes to prevent memory leaks

Practical results

ISA Server H.323 library bugs

- Information from the security bulletin:
  - Parsing problems in H.323 code
  - No technical details about whereabouts etc.
- Added range check before PERDecZero CharStringNoAlloc()
- Prevent integer overflows in the allocation functions

➔ Disclosed unknown vulnerability in NetMeeting and H323MSP !

➔ Publication of the fix did more harm than good

Future improvements

Future improvements on the BinDiff:

- Add functions frame sizes and constant arguments of malloc()-calls to the functions identification (to detect changed buffer sizes)
- Add static strings as nodes to callgraph for improved matching and ambiguity resolution
- Add function flowgrapher to retrieve better flowgraphs for speculative execution architectures

Future/Related work ?

Treating executables as graphs of graphs opens up interesting opportunities:

- Ero Carrera (of F-Secure AV Research team) uses graphs of graphs to cluster new HLL virii together to identify “code sharing” between virus authors
- Identification of library functions (e.g. OpenSSL) in large embedded systems can aid in reverse engineering
- Identification of GPL’ed code fragments in closed-source software could be possible
- Identification (or debunking) of code theft claims (as in the SCO vs Linux case)

Questions ?