Graph-Based Binary Analysis

Drawing pictures from code

Blackhat Briefings 2002

Halvar Flake
Reverse Engineer
Blackhat Consulting – http://www.blackhat.com
Graph-Based Binary Analysis

Overview (I)

The speech consists of four parts:

- **Part 1: Introduction**
  - What is a Graph?
  - Why Graphs?

- **Part 2: Simple Flowgraphing**
  - Problems with Microsoft Optimized Binaries
  - Flowgraph reduction for manual decompilation
  - FUZZ coverage analysis

- **Part 3: Structure and Object Reconstruction**
  - Pointer Control Graphing
  - Vtable parsing

- **Part 4: Variable Control Graphing**
  - Buffer Definition Graphing
Graph-Based Binary Analysis

Speech Background

- Reverse Engineering as main subject
  - Not security-centered
  - No new vulnerabilities
  - Why this is relevant at a security conference?

- Part 2: Code understanding & Manual Decompilation
  - Manual Binary Audits
  - Decompilation of tools only available in the binary

- Part 3: Structure and Object Reconstruction
  - Speeds up manual binary audits by a large factor
  - “Groundwork” for more sophisticated automated analysis

- Part 4: Inverse Variable Tracking
  - Speeds up manual audits a bit further
  - Allows advances in automated binary auditing
Introduction
What are Graphs?
Why Graphs?

- Graphs make code understanding easier
- Graphs make complex issues more clear than sequential code
- The only valid abstraction for computer code (single-threaded) is a directed Graph
- Graphs have been extensively studied in abstract mathematics
  - Many efficient algorithms for Graph Manipulation exist
- Graphs are fairly easy to generate
- Graphs can be displayed using off-the-shelf tools

→ Structuring Code as directed Graphs is beneficial for both manual analysis and automated tools
Simple Flow Graphs

Applications

- Simplify Code understanding
- Clarify Code interdependences
- Allow for gradual manual decompilation
- Can be used as basic blocks from which to build more sophisticated analysis tools

IDA 4.17 and higher include a built-in flowgraphing plugin
- Output is only provided in a file (not as data structure)
- The file is temporary and hard to find 😊
Simple Flow Graphs

Building a function flowgraph

Creating a flowgraph from the disassembly is trivial:

• Begin by tracing the code downwards
• If a local branch is encountered, “split” the graph and follow both branches
• Continue until a node with no further downlinks is encountered
• Heuristically scan for “switch”-constructs and handle them (special case)
Simple Flow Graphs
Microsoft Binary Optimization (I)
Microsoft optimizes memory footprints & page-fault-behaviour by re-arranging functions:
Simple Flow Graphs
Microsoft Binary Optimization (II)

The “less-trodden”-path is moved to a different page → Only relevant code stays on this page:

Begin
↓
Regular Code
↓
Return

Error Handler

Regular Flow

Side-Effect: IDA’s built-in Flowgrapher cannot cope with non-contiguous functions: (Demonstration)
Simple Flow Graphs
Graph Coloring & Reduction

• Manual Decompilation is tedious:
  – Reverse Engineers burn out easily
  – Small mistakes get back to you
  – Hard to keep track of progress

• Graphs can be used as visual aid
  – Step 1: Color the covered code
  – Step 2: Remove outer-layer loops & branches

• Graphs will keep track of progress
  – It’s good to see that you’re getting somewhere
RtlFreeHeap (II)

Checks if the pointer to the block is Non-NULL
RtlFreeHeap (III)

mov al, 1

mov ecx, [ebp + var_10]
mov large ptr fs:0, ecx
pop edi
pop esi
pop ebx
leave
retn
Simple Flow Graphs
Graph Coloring & Reduction

RtlFreeHeap(/* snip */ void *blk)
{
    if(blk == NULL)
        return(TRUE);
    return(TRUE);
}
RtlFreeHeap (IV)

```
mov    ebx, [ebp + arg_4]
or    ebx, [edi + 10h]
test   ebx, 7D030F60h
jnz   loc_77CBA96

push   edx
push   ebx
push   edi
call   _RtlFreeHeapSlowly
jmp    loc_77FCB6E4
```
RtlFreeHeap(HEAP *hHeap, DWORD flags, void *blk)
{
    if(blk == NULL)
        return(TRUE);

    if((flags | hHeap->flgs) & FLAGMASK)
        return(
            RtlFreeHeapSlowly(hHeap,
                flags | (hHeap->flgs),
                blk)
        );
}
RtlFreeHeap (V)
RtlFreeHeap (VI)
RtlFreeHeap (VII)
Graph Coloring & Reduction

- Graph Coloring helps ...
  - ... to see progress (Motivation boost 😊)
  - ... to keep track of covered code
  - ... to ensure no codebranch is missed
  - ... to “show results” to management

- Graph Reduction helps
  - ... to clarify complex situations
  - ... to see progress (“Only 5 Nodes left!”)
  - ... to make sure nothing is missed
RtlFreeHeap (VIII)
RtlFreeHeap (IX)
RtlFreeHeap (X)
RtlFreeHeap (XI)
FUZZ coverage analysis

• FUZZ-testing is highly inefficient:
  – Minor desynchronisation between protocols leads to not fuzzing at all
  – Undocumented features cannot be fuzzed
  – Hard to impossible to estimate how good a certain fuzz testing program is

• Analogy: Shooting Bats in a dark apartment

• Graphs can be used as a visual aid again!
  – Step 1: Generate Flow Graph
  – Step 2: Load into a debugger, set breakpoints
  – Step 3: FUZZZ the program, color touched nodes
Simple Flow Graphs
FUZZ coverage analysis

- Major advantages to conventional FUZZ:
  - Percentage of covered code can be measured
  - Fuzzing mechanisms/scripts can be dynamically improved to improve coverage
  - Quality of existing FUZZ-tools can be compared
- Analogy: Still shooting Bats in a dark appartment, but now we know that we’ve been in every room
- Demonstration
Simple Flow Graphs

Any questions concerning this part?
Structure/Class Reconstruction

- All information about structures and their layout gets lost in the compilation process.
- If we look for buffer overruns, we need to know buffer sizes.
- Manual structure reconstruction is an incredibly tedious, repetitive and annoying process!

→ Specialized Graphs might help.

Pointer Control Graphs
Structure/Class Reconstruction

• Identifying a pointer to a structure in the binary is usually trivial:

```
    mov    edi, [ebp + arg_0]
    mov    eax, [edi + 03Ch]
```

• If we can follow a pointer through the code, we can find all offsets which are added to it.
Pointer Control Graphs

Pointer Control Graphs are best suited for this:

• Start tracing code at a location, tracking a specific register/stack variable

• Trace code downwards until
  • A (local) branch is encountered
  • A write access to our variable is encountered
  • A read access to our variable is encountered
  • (Optional: A far branch (subfunction call) is encountered)
As soon as any of the above situations are encountered, do the following:

- **In case of a local branch:**
  - Behave as if we’re building a flowgraph → “split” the path and follow both codepaths downwards

- **In case of a register/variable write**
  - Abort the tracing as our register/variable has been overwritten

- **In case of a register/variable read**
  - “Split” the path and follow the codepaths for both the new and the old register/variable

- **In case of a non-local branch (optional)**
  - Trace into subfunctions and follow possible argument passing (tricky on x86 due to argument passing in both registers and stacks variables)
Example:

A simple Constructor for the IIS-Internal HTTP_REQUEST – Object:

- Visual C++ compiled code: `this` - Pointer in ECX
- Every move of ECX into another register needs to be tracked
- Every move of ECX into a stack variable needs to be tracked
- Tracking has to be recursive: Other registers are to be treated like ECX

- Demonstration
Pointer Control Graphs
Class Reconstruction

NodeBegin: 65f2a1ec
NodeEnd: 65f2a1ef
Register: ecx

NodeBegin: 65f2a1f1
NodeEnd: 65f2a20b
Register: ecx

NodeBegin: 65f2a267
NodeEnd: 65f2a268
Register: esi

NodeBegin: 65f2a267
NodeEnd: 65f2a26a
Register: eax

NodeBegin: 65f2a1f1
NodeEnd: 65f2a265
Register: esi
Example:
A simple Constructor for the IIS-Internal HTTP_REQUEST – Object:

- Single Functions do usually not access all structure members
- C++ Inheritance can lead to calling multiple Constructors subsequently
- **Subcall recursion and tracking of registers through subcalls** is needed for decent structure reconstruction

- Demonstration
Pointer Control Graphs
Class Reconstruction
Vtable Parsing:

- Virtual Methods are arranged in a “VTable”
- All Methods operate on the same data structure
- Very accurate reconstruction of classes by parsing this table
Summary:

- Structure data layouts can be automatically reconstructed from the binary by constructing & parsing pointer control graphs.
- Class data layouts can be automatically reconstructed from the binary by constructing & parsing pointer control graphs from vtables.
- Larger graphs can be too complex to display 😊.
- RPC interfaces (such as COM/COM+/DCOM) help us by publically exporting vtables for certain objects.

- Structure/Class reconstruction speeds up the binary analysis process by a large factor!
- *(TODO: Automatic type reconstruction from known library calls)*
Class/Structure Reconstruction
Any questions concerning this part?
Buffer Definition Graphs
Finding buffer definitions

Problem:
– Many problematic functions are not dangerous if the target buffer is big enough to hold all data
– These functions work on char *, which do not tell me the size of their buffers
– Tracking down where a char * came from is slow, boring, tedious and annoying
– In complex situations (multiple recursive functions etc.) it is quite easy to get lost and miss definitions

→ Specialized Graphs might help
Buffer Definition Graphs
Inverse Variable Tracking

Trace code upwards and track a variable/register until

• The current instruction was target of a branch
• The current register is written to from another register/variable
• The current register is loaded with something
• The current register is a return value from a function
Buffer Definition Graphs
Inverse Variable Tracking

• The current instruction was target of a branch
  – “Multi-Split” the graph (there can be more than 2 references) and trace further upwards

• The current register is written to from another register/variable
  – Follow this new register/variable, no need for splitting

• The current register is loaded with something
  – Analyze the situation, color the node blue for success and red for failure (ALPHA CODE)

• The current register/variable is manipulated in a way that we cannot cope with
  – Color the node red (ALPHA CODE)
Buffer Definition Graphs

Example Graphs
Buffer Definition Graphs

Any questions?

OBJRec and x86Graph available at:
