Vulnerability analysis, practical data flow analysis & visualization

Jeong Wook Oh
jeongoh@microsoft.com
Microsoft Malware Protection Center
Vulnerability analysis - what is it for?

To find the root cause of a vulnerability

- Find the relationship between user supplied data and the vulnerability

**Methodology**

- **Dynamic analysis**
  - Debuggers
    - Source code or symbol is a plus
  - Dynamic binary instrumentation

- **Static analysis**
  - Disassemblers
  - Source code review

- **Data format analysis**
  - Parse the analysis target using file or protocol specification
  - Analyze the target data for inconsistencies
Dynamic binary instrumentation

- Basically - perform user defined operations on executed instructions, basic blocks, functions and memory access operations.

- Callback model
  - One way to use DBI is by putting a callback function for each instruction, basic block, function and memory access.

- Implementations
  - Pin, DynamoRIO, etc.
  - Proprietary implementations

- We can trace a series of instructions that leads to a vulnerability
Why isn’t binary instrumentation enough?

- Why do we need additional instruction emulation above binary instrumentation?

Usually binary instrumentation provides callbacks to save information about instructions, basic blocks, functions, etc...

They don’t provide a means to evaluate the side effects of instruction execution.

- For example, which registers are affected, which registers are used for which instructions, etc...
The following assumptions are quite possible with existing public or private tools and the additional development of code:

- You have a nice binary instrumentation tool
  - You can save the execution log of a program to a file and retrieve all the run information (memory data, register value) as you need it
- You have CPU instruction emulation code
- The two pieces of code are working together to perform data flow analysis

These assumptions are not unrealistic and are totally possible to implement with reasonable effort.
FILE_HEADER file_header;

DWORD NumberOfBytesRead;

ReadFile(
    hFile,
    &file_header,
    sizeof( FILE_HEADER ),
    &NumberOfBytesRead,
    NULL
);
if( NumberOfBytesRead == sizeof( FILE_HEADER ) )
{
    // Integer Overflow (file_header.size is DWORD)
    size_t length = sizeof( FILE_HEADER ) + file_header.size;

    // length is used for malloc argument
    char *dst = ( char * ) malloc( length );
    if( dst )
    {
        memcpy( dst, &file_header, sizeof( FILE_HEADER ) );
        ...
    }
}
We trace back malloc argument.

Start of trace (start node)

Memory of 1st argument for malloc (memory node)

push eax is the argument pushing instruction from vulnerable code (instruction node)
Addition of 8 is happening, which implies integer overflow condition.
The data is eventually coming from user supplied data (ReadFile).
This implies the input for malloc is user-controllable.
For basic cases, data flow analysis is enough to determine the characteristics of the vulnerability.
Advanced Vulnerability Analysis Methodology

Case Of Control Flow Related Vulnerability
Taint analysis = Data flow analysis

- Works great for simple vulnerabilities, but has limited utility in analyzing complicated cases.
- For example, if it is a use-after-free vulnerability,
  - You can trace back to the origin of the data used in the crash instructions.
  - You get into somewhere that is inside freed memory.
  - You can determine this is an uninitialized memory access vulnerability.
  - But, data flow analysis will not tell you how it is happening.

A more advanced vulnerability analysis method is required

- How can you determine how the user-provided data is actually affecting the crash?
- User data changes not only data flow - it can also change control flow.
Data flow analysis

- From crash point (Ground 0)
  - Trace back to the cause of the crash
  - Most vulnerabilities can be analyzed using this method.

User supplied input → Corrupt data → Vulnerability
But, there are some vulnerabilities that are not straightforward enough to be traced by simple data flow analysis.

Those vulnerabilities usually involve control flow changes based on the user supplied data.

We need to find the control flow that made the vulnerability.
Compare the data flows:
- Acquire normal files that hit on same vulnerable code location
- Compare data flow from normal file and crash file
- Analyze the data flow differences
- Pinpoint the code part that is responsible for the data flow diversion.
Trace the data backward - find the first instruction that is not in a corrupt data flow, but in a good data flow.

- Good data
- Last part of good data flow
- Common data flow
- No crash
- Corrupt data
- Crash
Acquiring normal files

- You know the crash point
  - Code coverage test on the crash point (in basic block or function level)

- You have sample files
  - You have tons of files that are in the affected format (in this case, PDF files acquired from all places). We call them template files.

- Run the sample files using vulnerable software, performing code coverage testing on the vulnerable point
  - Find samples that hit the vulnerable point
  - These samples are used for data flow differential analysis
Control flow differential analysis

- Identifying key conditions for control flow difference
  - Perform control flow differential analysis between good/bad repro files that are used for data flow differential analysis
  - Deduce the conditions that create the control flow differences
    - Trace back to the user supplied data using data flow analysis
Control flow differential analysis

Common control flow

Conditional instructions

Good control flow

Good data

No crash

Corrupt control flow

Corrupt data

Crash

We find these instructions through control flow differential analysis
Visualization sometimes make things easier to understand

When we have long chains of data/control flow data, visualization can help researchers substantially with analysis
Advanced vulnerability analysis - methodology example

Control flow differential analysis

Visualization

Data flow analysis

Data flow differential analysis
EXAMPLE
CVE-2011-2462 (Adobe Reader)
- Used for highly targeted attack against US government & military companies
- The problem was inside U3D stream processing code

Why?
- Uninitialized memory access
  - Not a very typical memory corruption issue
  - Kind of tricky to debug
  - Taint tracking just tells you that some data comes from an uninitialized or previously freed memory area
- Public exploits were out, but no vulnerability analysis available except that provided to Adobe’s MAPP partners
  - We are not revealing any new vulnerabilities here
  - All information disclosed here is for research purposes and it’s not helping the development of new exploits in any way
Challenges

- Where is the freed memory coming from?
  - Can we find the code piece that is actually responsible for the vulnerability?
- Is this crash related to a value in the input file?
  - Can we identify the file position that is actually responsible for the vulnerability?
- What value range will make the thing happen?
  - Can we identify the condition where the vulnerability actually happens
We can get a very straightforward memory trace result.

- 3dife!E3DLLFunc+0x953f (2CAB7DF): mov eax, dword ptr [ecx]
- Memory:8E2E408(4) [B8 E5 E2 08 ]
- ntdll!RtlInterlockedPushEntrySList+0xc (7C902ABC): mov dword ptr [ebx], eax

Execution order

Memory transfer operations

Memory free operations (free)

Tracing backward
The pointer value is directly coming from the ntdll heap management function.

This vulnerability is a use-after-free or uninitialized memory access one.

If you take a look into the call stack, this is the part of heap free API (ntdll!RtlFreeHeap) which is eventually called from the msvcr80!free API.
Memory allocation operations (malloc)

Memory transfer operations

Execution order

Tracing backward

Memory allocation using the malloc API
We can conclude that the normal pointer should be pointing to a memory area allocated by using malloc API.

From the crash trace, the pointer was pointing to an area that is freed by free API.
Data flow differential analysis

Crash case

Memory transfer operations

Normal case

Memory allocation operations

Memory free operations

Memory transfer operations

Memory allocation operations
### Crash case vs Normal case

<table>
<thead>
<tr>
<th></th>
<th>Crash case</th>
<th>Normal case</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Address</strong></td>
<td>ntdll!RtlpInterlockedPushEntrySLlist+0xc</td>
<td>3difr!E3DLLFunc+0x7c08</td>
</tr>
<tr>
<td><strong>Instruction</strong></td>
<td>mov dword ptr [ebx], eax</td>
<td><strong>mov dword ptr [ecx+ebx*4h], eax</strong> (Key Instruction)</td>
</tr>
<tr>
<td><strong>Analysis</strong></td>
<td>Ntdll!RtlpInterlockedPushEntrySLlist is called from free API. So, ebx is a part of the heap management data structure, which means the crash comes from a use-after-free style vulnerability.</td>
<td>[ecx+ebx*4h] is the address of an array, the array base is ecx and ebx is the index of the array. Eax is coming from a malloc API. We need to figure out why this instruction is not hit and what is the closest hit around this key instruction.</td>
</tr>
</tbody>
</table>
Now we have a key instruction that creates the data flow differences.

We can do control flow differential analysis between the crash and normal files from this key instruction.

The purpose of this analysis is to find another key instruction that is creating the control flow difference.
Control flow differential analysis - Start from key instruction

- Identify the function that has the key instruction affecting data flow.
- Perform control flow analysis on the function using crash and normal files.
- Deduce differences of control flows in both cases.
- This should be repeated until any common control flow path is found.

**Key Instruction**

3difr!E3DLLFunc+0x7c08
mov dword ptr [ecx+ebx*4h], eax
Control flow differential analysis - Finding common instructions

- We can find the common control flow within the function itself.
- If it’s not found inside the same function, you need to trace up until common control flow is found.
The difference in the control flow is made by the test instruction at the red block.
Now you can trace the register eax value back to where it comes from.
Memory content is 0x00000000

The 4 bytes data is coming from the memcpy function.

If you track down more, you can see that this is coming from the file contents.
Now we can pinpoint the exact location of the data that is causing the vulnerability.

- Perform further analysis
  - Read specification document
  - Write parser
  - Do static and dynamic analysis upon the parsing code inside the binary
Data Flow Analysis
Find the source of data that is passed to the target variable

Control Flow Differential Analysis
Find the key instruction making the data flow differences

Data Flow Differential Analysis
Find the first node that is making the difference in the flow

Visualization

Strategy summary used here
This is an example of applying binary instrumentation & instruction emulation.

The detailed strategy and methodology should be applied according to each case.

Some hard cases can be cracked using this technology & application.

Comparing data flow & control flow with normal cases makes the vulnerability more visible.

Aside from data flow analysis which is used traditionally, we suggested using data flow differential analysis and control flow differential analysis method to help analyze vulnerabilities.