DPTrace: Dual Purpose Trace for Exploitability Analysis of Program Crashes

Rohit Mothe (@rohitwas)
Senior Security Researcher
rohit.mothe *noSPAM* intel.com

Rodrigo Rubira Branco (@BSDaemon)
Principal Security Researcher
rodrigo.branco *noSPAM* intel.com
Disclaimer

- We don’t speak for our employer (duh!). All the opinions and information presented here is our responsibility (actually no one has seen this talk before today).

  - IMPORTANT: No, we are *not* part of the Intel Security Group (McAfee)
Agenda

- Objectives
- Current state of Affairs or Security Today
- Taint Analysis Introduction
- Our approach – Dual Tracing
- Comparison with other ideas
- Demos
- Limitations
- Future
Objectives

• Contribute towards improving the state of the art in crash analysis

• Automate laborious/repetitive parts, but still requiring skilled exploit writer/analyst

• Discuss hybrid usage of techniques and the mixture of automation with manual analysis
Current State of Affairs

- Buggy programs deployed on critical servers
- Rapidly-evolving threats, attackers and tools (exploitation frameworks)
- Lack of developers training, resources and people to fix problems and create safe code
- That’s why we are here today, right?
tl; dr
Through our work we try to answer two fundamental questions:

- Are the input operands in the attacker’s control?
- And if so, is the forward execution providing a primitive that is good for an attacker?

Taint Analysis is one specific kind of program flow analysis and we use it to define the influence of external data (attacker’s controlled data) over the analyzed application.

Since the information flows, or is copied to, or influences other data there is a need to follow this influence in order to determine the control over specific areas (registers, memory locations). This is a requirement in order to determine exploitability.
History and Lore - Backward-Taint

- Original Motivation: Complex client-side vulnerability in a closed (at the time) file format

- Extended Motivation: Trying to better analyse hundreds of thousands of bugs in Microsoft Word (search for Ben Nagy, Coseinc)

- Initial version integrated with a fuzzer, only for Linux (showed in 2011 at Troopers)

- Ported version for Solaris to analyze a vulnerability released by Secunia in the same software RISE Security released a vulnerability a month before (also circa 2011)

- Thanks to Julio Auto’s parallel research in the same field, a Windows version was created (extended in this research)
Original Motivation: Triaging submissions in a vulnerability purchase program is hard. Many submissions lack a complete exploit but still might have real value.

Extended Motivation: Categorizing fuzzing crashes is a pain (NOT bang(!) exploitable categorizing)

Manual process includes lots of repetitive steps

Automation is key. Certain classes repeat themselves (such as UAF)

‘Prototyping Exploitation’ in such cases is both cost and time effective. Also a more reasonable and simpler ‘automatable’ problem than automating exploit writing for all classes of bugs. Prototype or GTFO!
Existing Solutions - What we aren’t

- exploitable
  - Tries to classify unique issues (crashes appearing through different code paths, machines involved in testing, and in multiple test cases). Group the crashes for analysis
  - Quickly prioritizes issues (since crashes appear in thousands, while analysis capabilities are VERY limited)
  - Classic, timeless!

- Spider Pig
  - Created by Piotr Bania
  - Not available for testing, but from the paper: It is much more advanced than the provided tool (but well, it is not available?)
    - Virtual Code Integration (or Dynamic Binary Rewriting)
    - Disputable Objects: Partially controlled data is analyzed using the parent data

- Taint Bochs
  - Used for tracking sensitive data lifecycle in memory
Existing Solutions - What we aren’t contd..

- **Taint Check**
  - Uses DynamicRIO or Valgrind
  - Taint Seed: Defining the tainted values (data coming from the network for example)
  - Taint Tracker: Tracks the propagation, Taint Assert: Alert about security violations
  - Used while testing software to detect overflow conditions, does not really help in the exploit creation

- **Bitblaze**
  - An amazing platform for binary analysis
  - Provides better classification of exploitability (Charlie Miller talk in BH)
  - Can be used as base platform for the provided solution (VINE)

- **Moflow Framework**
  - Cisco Talos. Tools built on CMU’s BAP framework.
  - sliceflow- post-crash graph back taint slicer
  - Post-crash forward symbolic emulator looking for more exploitable conditions
  - Pretty neat and advanced!
Case 1 (green): Format String
Case 2 and 3 (red and blue): buffer overflow
Case 4 (purple): unpredictable

Source:
Automatic Diagnosis and Response to Memory Corruption Vulnerabilities
Moving Backward

- Legitimate assumption:
  - To change the execution of a program illegitimately we need to have a value being derived from the attacker's input (which we call: controlled by the attacker)

- String sizes and format strings should usually be supplied by the code itself, not from external, untrusted inputs

- Any data originated from or arithmetically derived from un-trusted source must be inspected
Analyzing Taint

- Tainted data: Data from un-trusted source
- Keeps track of tainted data (from un-trusted source)
- Monitors program execution to track how tainted attribute propagates
- Detect when tainted data is used in sensitive way
When a tainted location is used in such a way that a value of other data is derived from the tainted data (like in mathematical operations, move instructions and others) we mark the other location as tainted as well.

The transitive relation is:
- If information A is used to derive information B:
  - A->t(B) -> Direct flow
- If B is used to derive information C:
  - B->t(C) -> Direct flow
  - Thus: A->t(C) -> Indirect flow

Due to the transitive nature, you can analyze individual transitions or the whole block (A->t(C))
Location

- A location is defined as:
  - Memory address and size
  - Register name (we use the register entirely, not partially -> thus %al and %eax are the same)
    - When setting a register, we set it higher (setting %al as tainted will also taint %eax)
    - When clearing a register, we clear it lower

- To keep track over bit operations in a register it is important to taint the code-block level of a control flow graph
  - This create extra complexity due to the existence of the flow graph and data flow dependencies graph
  - The dependencies graph represents the influence of a source data in the operation been performed
Flows

- Explicit flow:
  - `mov %eax, A`

- Implicit flow:
  - If `(x == 1) y=0;`

- Conditional statements require a special analysis approach:
  - In our case, we are analyzing the trace of a program (not the program itself, but only what was executed during the debugging section)
  - We have two different analysis step: tracing and analysis
Special Considerations

- Partial Tainting: When the untrusted source does not completely control the tainted data
- Tainting Merge: When there are two different untrusted sources being used to derive some data

Data
  - In Use: when it is referenced by an operation
  - Defined: when the data is modified
Inheritance problems

Problem: state explosion for binary operations!

Application

mov %eax ← A
mov B ← %eax
add %ebx ← D

Propagation Tracking

taint(%eax) = taint(A)
taint(B) = taint(%eax)
taint(%ebx) |= taint(D)

Inheritance Tracking

%eax inherits from A
B inherits from %eax
insert D into %ebx’s inherit-from list

Events

Rare
- e.g., malloc/free, system calls

Frequent
- e.g., memory access, data movement
Tracking Instructions

- Pure assignments: Easy to track
  - If a tainted location is used to define another location, this new location will be tainted

- Operations over strings are tainted when:
  - They are used to calculate string sizes using a tainted location
    - \(a = \text{strlen(tainted(string))}\);
    - Since the ‘string’ is tainted, we assume the attacker controls ‘a’
  - Search for some specific char using a tainted location, defining a flag if found or not found
    - \(\text{pointer} = \text{strchr(tainted(string), some_char)}\);
    - If (pointer) flag=1;
    - ‘flag’ is tainted if the attacker controls ‘string’ or ‘some_char’
Arithmetic instructions with at least one tainted data usually define tainted results

Those arithmetic instructions can be simplified to map to boolean operations and then the following rules applies
The eflags register can also be tainted to monitor flags conditions influencing in operations (and flow).

In the presented approach, conditional branches are taken care due to the trace generated by the WinDBG plugin (single-stepping).
Backward Taint Analysis

- Divide the analysis process in two parts:
  - A trace from a good state to the crash (incrementally dumped to a file) -> Gather substantial information about the target application when it receives the input data, which is formally named 'analysis'
  - Analysis of the trace file -> Formally defined as 'verification' step, where the conclusive analysis is done
To see what kind of primitives (read/write/calls) are available we ‘prototype’ input control and allocate a fake object structure in memory such that the program can continue from the point of the crash to other code paths.

The property of such fake memory structure should guarantee to a reasonable extent that any memory references (like virtual function tables or other object pointers) will be resolved including memory address references that are additive or subtractive to the faulting address (which is already assumed controllable).

In essence one could imagine it as simulating the reallocation of a fake object ‘within’ the debugger in a use-after-free situation and continuing the exception. Or allocating an adjacent object in an out of bounds access violation, etc.
Fake Memory Structure Sample

n - size of each object in bytes

d - depth/number of fake objects in the linked list chain
Forward Logic

- In the debugger you see a seemingly non exploitable read AV (access violation).
  - Example: `mov eax, [ecx];` (ecx is supposed here to be a pointer to attacker controlled memory.)
  - You allocate a chunk of memory within the process (preferably the size of the memory pointed to by ecx to mimic an accurate freed block control using heap massaging, feng-shui)

- The permissions of all memory blocks in a linked list chain are read-only. So any attempt to write/execute on any of the values within the memory blocks would cause an exception later and that shows evidence of exploitability

- Now manually change the ecx value in the crash to point to the address of the root of this linked list which is the root of the chain of memory blocks pointing to one another

- Continue the program execution and it will continue from the point of crash with the modified value of ecx.
Assembly instructions have explicit operands, which are easy to deal with, and sometimes implicit operands:

- Instruction: `push eax`

- Explicit operand: `eax`

- What it really does?
  - `ESP = ESP - 4` (a substraction)
  - `SS:[ESP] = EAX` (a move)

Here we have ESP and SS as implicit operands

Tks to Edgar Barbose for this great example!
Implementing the Tracer

- Instead of using an intermediate language, we play straight with the debugger interfaces (WinDBG). Windbg or GTFO!

- The tracer stores some useful information, like effective addresses and data values and also simplifies the instructions for easy parsing:

  - CMPXCHG r/m32, r32 -> 'Compare EAX with r/m32. If equal, ZF is set and r32 is loaded into r/m32. Else, clear ZF and load r/m32 into AL'

    - Such an instruction creates the need for conditional taints, since by controlling %eax and r32 the attacker controls r/m32 too.
Implementation Details

- Instead of using an intermediate language, we play straight with the debugger interfaces (WinDBG). Windbg or GTFO!

- Trace File Contains:
  - Mnemonic of the instruction and operands
  - Dependences for the source operand
    - Eg: Elements of an indirectly addressed memory
    - This creates a tree of the dataflow, with a root in the crash instruction

- The verification (GUI and cmdline program) step reads this file and:
  - Search this tree using a BFS algorithm

- Forward step uses the debugger interfaces for the memory allocation and forward execution
Program Execution Timeline

- Program start
- Initial Crash, AV
- Exception → Constraint 1
- Exception → Constraint n
- Exploitable Primitive?! Profit!
- OR march on, down to hell 🙃
Theoretical Example

- 1) mov edi, 0x1234 ; dst=edi, src=0x1234
- 2) mov eax, [0xABCD] ; dst=eax, src=ptr 0xABCD ; Note 0xABCD is evil addr
- 3) lea ebx, [eax+ecx*8] ; dst=ebx, src=eax, srcdep1=ecx
- 4) mov [edi], ebx ; dst=ptr 0x1234, src=ebx
- 5) mov esi, [edi] ; dst=esi, src=ptr 0x1234, srcdep1=edi
- 6) mov edx, [esi] ; Crash!!!
6-) Where does [esi] come from?
5-) [edi] is moved to esi, where edi comes from and what does exist in [edi]?
4-) [edi] receives ebx and edi is defined in 1-) from a fixed value
3-) ebx comes from a lea instruction that uses eax and ecx
2-) eax receives a value controlled by the attacker
... ecx is out of the scope here :}
Assumptions & Challenges

- Since we only use the trace information, if the crash input data does not force a flow, we can't see the influence of the input over this specific flow data.

- To solve that:
  - If a jmp is dependent of a flag, the attacker controls branch decision.
  - Control over a branch means tainted EIP.
  - To define the value of EIP, consider:
    - The address if the jump is taken.
    - The address of the next instruction (if the jump is not taken).
    - The value of the interesting flag register (0 or 1).
  - Then: `%eip <- (address of the next instruction) + value of the register flag * ( |address if jump is taken – address of the next instruction| )`
The method here was conceived originally to help determine whether crashes for potential UAF (Use-After-Free) bugs in browsers are exploitable or not.

UAFs in browsers or any significantly large programs for that matter are often hard to analyze for exploitability and typically involve following varied code paths in the control flow to find a write access violation/potential code redirection using indirect calls.

The idea is not just limited to UAFs though.

After input control (first part of the problem) has been determined, the next logical step is to gauge what can be done with it.
Command-line options

0:016> .load dptracer
0:016> !dptrace_help
Dual Purpose Tracer v1.0 Alpha - Copyright (C) 2000-2016
License: This software was created as companion to a Black Hat Presentation.
Developed by Rodrigo Rubira Branco (BSDemail) <rodrigo@kernelhacking.com> and Rohit Motshe <rohitwa@gmail.com> (alphabetical order of names)
Heavily based on VDT-Tracer by Julio Auto and Rodrigo Franco

!dptrace_trace <filename> - trace the program until a breakpoint or exception and save the trace
    in a file to be later consumed by the Visual Data Tracer GUI.
!dptrace_forward <n(required) - x(required) - p(OPTIONAL)>
    - forward analysis, either no arguments or all mandatory
!dptrace_analyzer <analyzer_filepath> <trace_filepath> <close_gui> <controlled_ranges> <instr_index>
!dptrace_analyzer_help - help to the !dptrace_run_analyzer command
!dptrace_forward_help - help to the !dptrace_forward command
!dptrace_help - this help screen

*** Error: Symbol file could not be found. Defaulted to export symbols for C:\WINDOWS\SYSTEM32\ntdll.dll
ntdll!DbgBreakPoint:    int    3
0:016> .load dptracer

0:016> !dptrace_trace C:\Desktop\LogFiles\Log.vdt |
Analyzer

```
0:000> !dptrace_analyzer """C:\\Users\\rrbranco\\Desktop\\Black Hat 2016\\DPTrace-BlackHat 2\\Debug\\DP TRACE-GUI.exe"
0:000> .unload dptrace-tracer
Unloading dptrace-tracer extension DLL
0:000> .load dptrace-tracer
0:000> !dptrace_analyzer """C:\\Users\\rrbranco\\Desktop\\Black Hat 2016\\DPTrace-BlackHat 2\\Debug\\DP TRACE-GUI.exe"

Opening file: C:\\Users\\rrbranco\\Desktop\\Black Hat 2016\\DPTrace-BlackHat 2\\Debug\\Sample_output\\dpt
Processing file...
Number of instrs (and instruction to check taint of): 124116 124115
Range Start: 0x80000 Range End: 0x81000
Range Start: 0x10000 Range End: 0x11000
Instruction: 775bb5b7 8b9264040000 mov edx,word ptr [edx+464h] ds:002b:00381464=000000

Dumping instruction taint information:
instr->Src tainted: *00381464
instr->SrcDep1 tainted: edx

Closing GUI
```
Forward

0:009> dptrace_forward help
dptrace_forward:

[*] Two options of running-
   a) Simple Run with no arguments
      dptrace_forward
      = > default number of objects n = 2, size s= 40, permissions are RW for first object, READONLY for all other

   b) If running with arguments to configure the run following rules apply -
      dptrace_forward - n(required) - s(required) - p(OPTIONAL)
      = > number of fake objects(required), size of each object in bytes(required), page permissions(OPTIONAL)(0:

Parameters n and s are parsed in decimal(base 10). however, if passing the 3rd parameter p, specify the const

Optional>For passing 'p' use the following map -
   PAGE_EXECUTE  0x10
   PAGE_EXECUTE_READ 0x20
   PAGE_EXECUTE_READWRITE 0x40
   PAGE_EXECUTE_WRITECOPY 0x80
   PAGE_NOACCESS  0x01
   PAGE_READONLY  0x02 // default protection if run with no arguments.
   PAGE_READWRITE 0x04
   PAGE_WRITECOPY  0x08

</Optional>

0:009> dptrace_forward 3 100 0x2

Allocated range is
3da0000-3da1000,3db0000-3db1000,3dc0000-3dc1000
WHAT IS DEAD MAY NEVER DIE
Sample Analysis 1

We did a bit of cheating to avoid huge traces (from that point on til the crash, we would have traced more than 10 million instructions)

- CVE-2010-0188 – Adobe Reader Libtiff TIFFFetchShortPair Stack-based Buffer Overflow
- TIFF file embedded in a PDF were the IFD Entry has Tag ID (0x0129, 0x0141, 0x0212 or 0x0150) and Tag Type 3 (short)
- The field data count of the TIFF file will be used as size (dc*2) to copy to a fixed buffer in stack
At the crash point, we check the trace to see if the pointer is Indeed controlled.
Dataflow information can be visualized in the GUI
We indeed control the values (coming from our input file)
Sample Analysis 2

CVE-2014-0282 IE8/9/10/11 ‘Cinput’ Use-After-Free (MS14-035)
Replace Freed object with the root of the fake object chain.
Continue from initial crash and trace each subsequent breakpoint/access violation.
Add the range of the fake allocated objects, so when we look for the taint information on the instruction of interest, we can confirm it is mapped to our controlled memory areas.
Program Control Immediately evident. We just need to make sure we can point indeed it to our fake structure
We see that the EIP value at time of crash comes from our fake object allocated at the previous crash.
Visualize it in the tracer and trace the program control (or directly in the command line of the debugger, shown later)
Taint source is confirmed also in the analyzer (visual here). Same thing can be obtained in the command line by `!dptrace_analyzer <analyzer_binary> <trace_file> <keep GUI open> <ranges> <index of instruction to check the taint of>`
Because the backward taint analysis demand tracing the process, so we can later construct the BFS analysis, it is important to use intelligently/diligently. In the case of this issue, we use to analyze a part of the execution, instead of the initial crash.
Contd...

First chance exceptions are reported before any exception handling.
This exception may be expected and handled.

First chance exceptions are reported before any exception handling.
This exception may be expected and handled.

A total of 9 instructions were traced and 6 were dumped to C:\Users\Yours\Desktop\FoCs\log_final2.vdt
Duration of this command in seconds: 0.800000
Contd...

```
Contd...

Allocation range is
5880000-5881000, 5890000-5891000, 58a0000-58a1000, 58b0000-58b1000

0:065> rd rsi=5880000
0:065> ex
```

A total of 9 instructions were traced and 6 were dumped to C:\Users\MacbookPro\Desktop\PoCa\log_final12.vtd
Duration of this command in seconds: 0.000800
Everything that was done using the GUI (setting the taint ranges, defining the instruction of interest and analyzing its taint information) is possible to do via the command-line of the debugger, as shown here:
Sample Analysis 3

Fake object chain of 4 objects of size 200. Precise size can be determined by manual analysis to figure out the freed/alloc’d function and checking the size of the root object.
‘Redefine’ the reference to freed reference (eax) with the first fake object. Continue the execution with !dptrace_trace and monitor the forward trace.
Add taint range and check to see if the source of an access violation can be traced back to controlled input.
Following another path by meeting a new constraint
More constraints
Checking the taint source again. This particular execution run leads us to uncertainty and we aren’t sure of an exploitable primitive yet.
So we carry on another execution while trying to meet some other constraints and hit an alternate code path this time.
We try another path this time by crafting some different values within the fake object, notably the value of 0x40000 in the dword @ fake_object+0x24. We also modify references to the same fake object in edi (CTreeNode *) and on the stack (esp+24). Hit a more interesting exception!
Preliminary analysis shows us that the MSHTML!!\texttt{report\_securityfailure} call was triggered due to a failed VTguard\_check (next figure)
Confirm taint control and we influence the pointer which is dereferenced to do the vtguard check. That there is code execution right after the vtguard_check can either be looked into the debugger or within IDA for more clarity as shown above.
Challenges & Limitations

- Determining the actual range of memory which needs to be traced. Determining this is easier for some cases (like file format bugs) whereas for browser based bugs this can be difficult (and sometimes unnecessary).
- Explosion and partial tainting (we assume full control when merging taint).
- Because the tracer outputs instruction information, it needs to understand the semantics of it (for example, source and destination operands):
  - It only supports the most basic x86 subset (no x87, MMX, XMM, etc) (future versions, also, helping is caring!)
Another limitation of the approach is covering conditional code paths that hit only on certain values expected to be in the memory address (checking of reference counters, object type tag or some other metadata that affects the control flow of the program after the crash point)

- Branch Explosion! Similar problems can arise with symbolic execution approach

- Manual analysis involves knowing where to break, where to start tracing, etc. The closer to the exception the better because of smaller traces and faster processing time by the analyzer

- Not a magic solution that works on its own without a skilled analyst. Not a one size fits all solution either. Meant to augment crash analysis.
We aren’t soothsayers. More like sooth-slayers \../

Please, read TODO.txt in the code trunk and send pull requests :p

Latest version of this presentation, paper, code and demos available at:

Acknowledgments

- Julio Auto for his previous work alongside one of the authors of this work (Rodrigo Branco) in implementing VDT (Vulnerability Data tracer) which was the original implementation of the backward taint tracing plugin.

- David D. Rude (@bannedit) and Kiran Bandla (@kb) for ideas and feedback regarding the initial prototype of the forward trace.

- All of the other researchers who contributed to this field!
Thanks!

Rohit Mothe (@rohitwas)
Senior Security Researcher
rohit.mothe *noSPAM* intel.com

Rodrigo Rubira Branco (@BSDaemon)
Principal Security Researcher
rodrigo.branco *noSPAM* intel.com