

# DPTTrace

Dual Purpose Trace for Exploitability Analysis of Program Crashes

**Rohit Mothe (th1rm0)**

Senior Security Researcher

Intel Corporation - Security Center of Excellence

**Rodrigo Rubira Branco (BSDaemon)**

Principal Security Researcher

Intel Corporation – Security Center of Excellence

**\* Authors name in random order**

**v1.1**

**For an updated version of this paper, please check the repository of the talk in  
github:**

**<https://github.com/rrbranco/BlackHat2016>**

# Table of Contents

- [0] Abstract
- [1] Background and Motivation
- [2] Dichotomy of the 'Exploitability' Problem
- [3] Approach and Ideas
- [4] Solution = Backward Trace (!?)
- [5] Solution + = Forward Taint (!)
- [6] Challenges & Limitations
- [7] Code & Implementation
- [8] Demos and real life usages
- [9] Other existing solutions
- [10] Acknowledgements
- [11] References

## [0] Abstract

This research focuses on determining the practical exploitability of software issues by means of crash analysis. The target was not to automatically generate exploits, and not even to fully automate the entire process of crash analysis; but to provide a holistic feedback-oriented approach that augments a researcher's efforts in triaging the exploitability and impact of a program crash (or fault). The result is a semi-automated crash analysis framework that can speed-up the work of an exploit writer (analyst).

Fuzzing, a powerful method for vulnerability discovery keeps getting more popular in all segments across the industry - from developers to bug hunters. With fuzzing frameworks becoming more sophisticated (and intelligent), the task of product security teams and exploit analysts to triage the constant influx of bug reports and associated crashes received from external researchers has increased dramatically. Exploit writers are also facing new challenges: with the advance of modern protection mechanisms, bug bounties and high-prices in vulnerabilities, their time to analyze a potential issue found and write a working exploits is shrinking.

Given the need to improve the existing tools and methodologies in the field of program crash analysis, our research speeds-up dealing with a vast corpus of crashes. We discuss existing problems, ideas and present our approach that is in essence a combination of backward and forward taint propagation systems. The idea here is to leverage both these approaches and to integrate them into one single framework that provides, at the moment of a crash, the mapping of the input areas that influence the crash situation and from the crash on, an analysis of the potential capabilities for achieving code execution.

We discuss the concepts and the implementation of two functional tools developed by the authors (one of which was previously released) and go about the benefits of integrating them. Finally, we demonstrate the use of the integrated tool (DPTrace – to be released as open-source at Black Hat) with public vulnerabilities (zero-days at the time of the released in the past), including a few that the authors themselves discovered, analyzed/exploited and reported.

## [1] Background and Motivation

Determining the exploitability of a program crash not only helps the bug hunter/exploit writer who is rummaging her/his way through tens of thousands of crashes to find some gold but also a product security response team in the initial triaging phases to understand the real impact of a bug submission before allocating resources for the development of a fix. It helps enterprises to make informed decisions on patch prioritization as well if, of course, they have some way of accessing and testing the bug trigger. At the core of it though, these seemingly different problems are essentially the same - given a large set of crashes (as a result of fuzzing or obtained by bug submissions by researchers all across the world) how can one weed out the non-exploitable issues? This is quite a complex problem in itself to completely automate with so many different code paths, conditional constraints and the continuously varying states of the target program.

Traditionally, this process of determining exploitability relies on a lot of manual work of debugging the program at, and just before, the time of the crash and figuring out the kind of error that occurs. While analyzing submitted (or found on the wild) materials (crashes), it is also necessary to triage back to the input location that triggered the issue. Product security response teams are often tasked in analyzing reported issues without access to source-code, thus facing similar challenges as external researchers. To triage issues, one must have an understanding about the program itself, the internals of the program state and other specialized knowledge. In the Windows OS, the most readily and popular available solution to analyze such crashes is provided by Microsoft in the form of a WinDBG plugin called “!exploitable” (aka “bang exploitable”) [1]. This plugin conducts a preliminary analysis of the crash point and among a variety of other things, analyzes the crash and determines an exploitability rating that can span 4 different labels - ‘exploitable’, ‘probably exploitable’, ‘not exploitable’ or ‘unknown’. This is inferred based on a variety of parameters, like the kind of crash (READ/WRITE based access violation), call stack related information etc. It obviously is meant for a quick checkup of what the crash is but if one is really interested in digging through the crash and find out the true exploitability then !exploitable is not a reliable mechanism. It’s an easy and simplistic way of classifying crashes but does not provide any real value to address the problem of exploitability. The goal then becomes to improve upon this and propose a better system. But first, we intend to clearly define the problem and explain the intricacies involved with it.

## [2] Dichotomy of the ‘Exploitability’ Problem

There can be multiple solutions to a particular problem and every solution is unique in the sense that in the mind of each creator/inventor/designer of the solution, the problem itself is perceived differently. Hence it is imperative that we detail our vision of the problem of exploitability determination in crash analysis. The way we see it, there are two dimensions to this problem. One is determining control over the cause of the exception/crash in the program and the other is determining the actual program execution control that can be achieved from that.

Both are tied into each other and a lack of information about either of them translates to an incomplete understanding of the exploitability of the program. A program is truly exploitable if the two following conditions are met:

- a) Whatever is causing the exception/crash is related and to some extent controllable through the input that crashes a program.
- b) Either at the time of the crash or in some code path after the crash, there is evidence of program control that can be achieved by directly/indirectly manipulating the program input.

Of course a) and b) go hand in hand. If you can't control the input then b) doesn't even make sense and the crash is not exploitable. Similarly if you have program control but you can't leverage that to corrupt something in memory for instance to gain program execution control, then that is also moot and the crash is non exploitable.

## [3] Approach towards a Solution

As noted above, our aim is to solve the two fundamental problems defined above which, in a terse form are: a) Input control and b) Program Execution Control.

The problem of input control is automating the process of checking whether the attacker controls the input in a way that influences a crash and also the associated code traces to reach that crash point. The technique used here is Backward Taint Analysis. It is essentially tracking a particular memory area which contains controllable input fed to the program initially as taint and track/trace it throughout the program execution. We dump and analyze the trace to help a researcher check back if the crash was influenced by the initially defined memory area containing the controllable input or not. Furthermore the verification process also helps identify what modifications were performed to this input leading to the crash.

The idea for this is an extension of the original idea presented previously as VDT (Vulnerability Data Tracer) [2] in the form of a Windbg plugin and an analysis program.

After determining the input control and the influence of the input over the program crash, we want to go further and try to solve the actual program control problem. The technique we propose here is of forward taint propagation and the idea is pretty similar to the backward taint (even though the implementation is completely different, since we are not interested in the instruction tracing). Assuming input control does exist at the moment of the program crash; we create a fake memory structure within the debugger at the time of the crash, assign certain memory permissions to these allocated blocks of

memory and manually change the invalid illegal access of memory to point to the block of specially allocated memory. This simulates attacker control of the invalid memory that was accessed and once the execution continues with the fake memory block pointer, we trace the execution path and analyze the potential of program control.

The idea primarily is that the way the memory permissions are set up on the fake memory structure, any attempts to access a region of memory within this block will trigger an immediate exception and prove that the attacker controlled data can be utilized for program control later in the execution of the program after the crash. With this a researcher can quickly prototype an actual exploitation attempt and determine the actual exploitability of the program crash.

The final goal of this work is to release the tool as a debugger extension (WinDBG for now as the tool targets Windows environments only), that when run with a program that crashes on a particular input:

- Traces the program execution until it crashes
- Analyzes the trace backwards (from the point of the crash to the initial loading of the program)
  - Dumps this backward trace to help manual analysis as well
- Determine input influence and if positive then automatically invokes the next stage
- Creates fake memory structures of chosen depth/size, modifies the program execution by modifying the invalid memory access (through CPU registers changes) and continues execution
- Provides a trace of this post-crash execution to help manual analysis

Sample run of the plugin (we call it DPTrace as in Dual Purpose Trace and pronouncing Deep Trace) would be as follows:

```
0:018> .load dptracer
0:018> !dptrace_help
Dual Purpose Tracer v1.0 Alpha - Copyright (C) 2008-2016
License: This software was created as companion to a Black Hat Presentation.
Developed by Rodrigo Rubira Branco (BSDaemon) <rodrigo@kernelhacking.com> and Rohit Mothe <rohitwas@gmail.com> (alphabetical order of names)
Heavily based on VDT-Tracer by Julio Auto and Rodrigo Branco

!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) - s(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
```

\*Switches are various controlling parameters for the tool including but not limited to running the forward taint tracer unconditionally, dumping the traces on potential failures, modifying the default memory permissions of the fake memory structures during forward trace, get notifications of untaken paths (that could influence exploitability), etc.

## [4] Backward Taint Analysis = half the problem

Backward Taint Analysis is an inverse approach to the natural taint analysis flow. Basically, instead of getting all the input, mark it as tainted and track it during the program execution, what it does is to get the crash, validate what is of interest (which led to the application crash) and trace back to see if it comes from the input and, if so, what modifications were performed.

This avoids the explosion of tainted data, since most of the input is considered not tainted (and usually it is legitimate). To do so, the process of backward tainting is divided 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 its input data
- Analysis of the trace file -> a verification step, where the conclusive analysis of the trace is performed

The trace step stores some useful information, like effective addresses and data values (later used to determine what is been copied to where and how it is been affected). Note that:

- This is done using a WinDBG extension
- It only supports the basic x86 instructions (so, no MMX and SSE)
- Simplification of the instructions to make the next step easier

The simplification deals with many instruction specifics, for example:

- `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' [3]

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

The alternative taints list is also generated, in the form of `srcdep{1,2,3}`.

Since the trace step generates a file to be loaded by the next step, this file contains:

- Mnemonic of the instruction
- Operands
- Dependences for the source operand

Dependences for an operand are for example, elements of an indirectly addressed memory. This will create a tree of the dataflow, with a root in the crash instruction.

The verification step receives the address ranges that have the attacker data (input to the DPTrace) and then does the automatic analysis to determine the control over the elements involved at crash time, activating the next phase of the analysis (the forward analysis). The verification is performed by a standalone tool called by the plugin (also provided as open-source). Since the dataflow is available in a tree rooted in the crash instruction, the analysis step will just search in this tree, using a BFS [4] algorithm.

## [5] The Other Half = Forward Taint

The second phase is fully implemented inside the debugger plugin itself, and is activated based on the results from the backward taint analysis. The idea behind it is: Given a controllable access violation how do you determine if that control can be leveraged later in the code path to cause an exploitable access violation.

The method here was conceived originally to help determine whether crashes for potential UAF (Use-After-Free) bugs in browsers are exploitable or not. This was very helpful in browser based (IE/Chrome/Firefox) UAF crash analysis when there is a huge corpus of crashes. The main motivation behind this was that 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 (call eax) etc.

The idea is not just limited to UAFs though and this generic pattern of self-referential (kind of) memory is, in our opinion, applicable to other bug classes as well. Fundamentally, this type of fake memory structure guarantees to a certain 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 assumed controllable). So even out of bounds (OOB) access, underflows/overflows, type confusion (ultimately leading to some form of an OOB access) will all be under the plugin's scope.

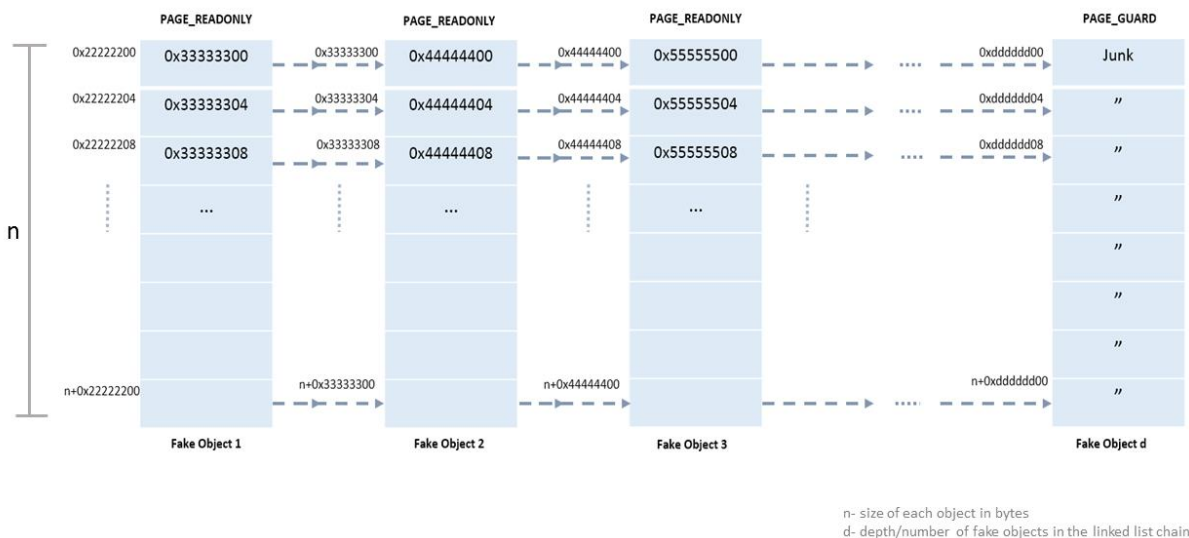
### **Logic/Implementation overview:**

The basic idea is to automate most of the following manual process in this phase:

- a) In the debugger you see a seemingly non exploitable read AV (access violation). For 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 spray/feng shui).
- b) Let's call the memory block initialized in step a) as M1. Now allocate multiple memory blocks M2, M3,M4...Mn within the process such that dword at M1 contains the address of the first dword of M2 (basically first 4 bytes of M1 point to the address of the first 4 dword of M2) , second dword of M1 points to second dword of M2.... And so on.  
Similarly the first dword of M2 points to first dword of M3, second dword M2 to second dword of M3 and so on and on....  
The last memory block Mn just contains garbage data.
- c) The permissions of all memory blocks from M1, M2, M3, M4...Mn 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.
- d) Now manually change the ecx value in the crash above to point to the address of M1 which is the root of the chain of memory blocks pointing to one another.
- e) Continue the program execution and it will continue from the point of crash with the modified value of ecx.

Figure 1 has the layout of the configured memory with the fake entries, for better understanding.





The problem here of course is that certain code areas might not be reachable even after multiple runs of the tool. For example there might be checks on the fake memory values and a certain code path with exploitable primitives only reached if the values are a certain number or within a certain range. For this reason, the taint analyzes documents these checkpoints and provides this info in the dump so that the analyst can further leverage this information in debugging and making sure to cover all the code paths. This is also a potential limitation that is meant to be addressed in future work by involving code coverage strategies.

## [6] Challenges & Limitations

The primary challenges with the backward trace are 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.

Currently this requires manual effort from the researcher's side and for cases like browser based bugs it's harder to gauge input control just by tracking input taint since the HTML/JS is interpreted and the difficulty of mapping the input code in memory prior to interpretation by the HTML/JS engines to the resulting execution is a more challenging problem that requires a deeper knowledge of the inner workings of each different browser.

This problem we believe is better solved by a focused analysis of each browser and its internals which would make it dependent not only on the browser product but also the version and the OS it runs on.

This aspect is out of scope of our current work and our suggestion is to augment this directly via manual research/analysis. Having said that, the backward taint analysis system in our implementation does offer some respite in the sense that you don't need a precise range of addresses in the memory you need to track the taint.

On the other hand, some immediately obvious limitations of the forward taint approach are 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. Navigating them is a maze and currently our solution doesn't completely solve this. But the helpful aid here is that we do mark these so called checkpoints and in the resulting trace a researcher can see which points to further break at and resume debugging or do another run of the plugin to skip these checks(if at all they can be skipped) etc. Another potential problem that we don't address is that for certain classes of bugs you need the precise object size to imitate a program control. There is an option in the plugin for inputting the actual size of each block within the memory structure but this has to be manually inputted by the researcher for now after some basic debugging and initial triaging of the crash.

## [7] Code & Implementation

This new plugin we releasing is a combination of two separate functioning pieces of tools that can be seamlessly combined into one integrated framework. One of the tools is already public as noted below and the other one is currently privately held. The intention is to release the combined tool to the public just after Black Hat 2016 after more tests of the integration.

The implementation of the final tool is a C++ based WinDBG plugin as we personally see the most benefit in implementing this idea on a standard windows debugger. Future work could involve porting it to Linux (GDB) based on feedback and interest from the community.

The final tool combines elements from above implementations and provide a single cohesive debugger (WinDBG plugin only for now) plugin that should enhance a researcher's crash analysis workflow for determining exploitability. The reason for choosing to make it a debugger plugging is also to simplify the integration of this tool into any other fuzzing framework because all the logic is triggered within the debugger. The specific fuzzing framework can then implement functionality to use the results of applying this plugin to the crashes on the fly and classify into buckets of exploitability based on the results.

The code used in this paper can be downloaded at [5]:

<https://github.com/rrbranco/BlackHat2016>

## [8] Demos and real life usages

In this section we demonstrate the analysis of three different issues. We start from the simplest to the most complicated, trying to not only walking the reader through the process, but also clearly pointing out the limitations of the approach.

### **CVE-2010-0188 Adobe Reader Libtiff TIFFFetchShortPair Stack-Based Buffer Overflow (APSB10-07)**

A stack-based buffer overflow vulnerability in the libtiff used by Adobe Reader. The vulnerability affects the AcroForm.api, in the function TIFFFetchShortPair and is related to the element DataCount of a TIFFDirEntry.

“Tag Image File Format (TIFF) is a file format used primarily for storing digital images, including photographs and line art. TIFF is a popular format for high color depth images, along with JPEG and PNG”. The vulnerability affected libtiff and years later it was found that Adobe was still using the vulnerable library in its products.

Looking into the TIFF format, we have:

Offset	Size	Description
0x0000	2	Byte Order
0x0002	2	Constant Identifier
0x0004	4	Offset of the first IFD table (T)
...	...	...
T	2	Number of IFD tables (M)
T+0x02	12	IFD Entry 1
T+0x0E	12	IFD Entry 2
T+0x1A	12	IFD Entry 3
...		
T+0x02+12*M		Offset to the next IFD until value is 0
...	...	...

And for each IFD entry:

Offset	Size	Description
0x0000	2	Tag ID
0x0002	2	Tag Type
0x0004	4	Data Count (dc)
0x0008	4	Value (if dc <= 4) or Offset (if dc > 4)

PDF files can embed TIFF image files. Those files are parsed by the libtiff inside Adobe and treated as normal TIFF files. They are usually base64 encoded inside the PDF.

The problem here is with an embedded TIFF file that has TAG ID 0x0129 (known as PageNumber), TAG ID 0x0141 (known as HalftoneHints), TAG ID 0x0212 (known as YCbCrSubSampling) or TAG ID 0x0150 (known as DotRange). What happens is for those TAG IDs, if their Tag Type is defined as SHORT (3), the parser uses the value of the Data Count (dc) field to get the size of the source data buffer, as in:

size = data count \* 2

That amount of data (size) is read and copied in a fixed stack buffer, leading to a stack-based buffer overflow.

We did a bit of cheating in this one, since we had a working exploit and no crashes. The working exploit was crashing in some targets (OS versions), but with an obvious exploitable condition. We modified it to generate a violation in the stack copying (copying too much data to the point to overwrite illegal values). We also tried to avoid to trace too many instructions (from the breakpoint to the crash point we traced more than 10 million instructions). The trick is to instead let the program go under the trace, to create more conditions for the point where you start tracing. We leave that process as an exercise to the reader since it is not directly related to our analysis.

This is the instruction of interest and its offset location for analysis:

```

Breakpoint 0 hit
eax=05bf3c38 ebx=00000400 ecx=05db7260 edx=05bf3c33 esi=002be28c edi=00000000
eip=638038d5 esp=002be174 ebp=002be1a4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
AcroForm!DllUnregisterServer+0x1bd752:
638038d5 8b01          mov     eax,dword ptr [ecx]  ds:0023:05db7260=63bd8d68

```

At the crash point, if we analyze the trace for backward taint data we have:

```

0:000> !dptrace_analyzer ^" "C:\Users\rrbranco\Desktop\Black Hat 2016\DPTrace-BlackHat 2016\Debug\DPTRACE-GUI.exe" ^"C:\Users\rrbranco\Desktop\Black Hat 2016\DPTrace-BlackHat 2016\Debug\DPTRACE-GUI.exe" ^"C:\Users\rrbranco\Desktop\Black Hat 2016\DPTrace-BlackHat 2016\Sample_output\dptrace-test2.vdt"
Opening file: C:\Users\rrbranco\Desktop\Black Hat 2016\DPTrace-BlackHat 2016\Sample_output\dptrace-test2.vdt
Processing file...

Instruction: 651c35ed 8b01          mov     eax,dword ptr [ecx]  ds:0023:062d9300=65591260

Dumping instruction taint information:

instr->Src tainted: *062d9300
instr->SrcDepl tainted: ecx

```

Meaning that there is control information coming directly from our input. In the GUI, we can see that the dataflow indeed exists:

The screenshot shows the DPTrace GUI with the following analysis results:

```

Possible source of taint found:
Printing (possibly a part of) the tainting instruction. 651c35ea 8b4e1c  mov     ecx,dword ptr [esi+1Ch] ds:0023:062d920c=062d9300
Destination operand: ecx
Source operand: *062d920c

Printing dataflow path:
651c35ea 8b4e1c  mov     ecx,dword ptr [esi+1Ch] ds:0023:062d920c=062d9300
651c35ed 8b01  mov     eax,dword ptr [ecx]  ds:0023:062d9300=65591260

```

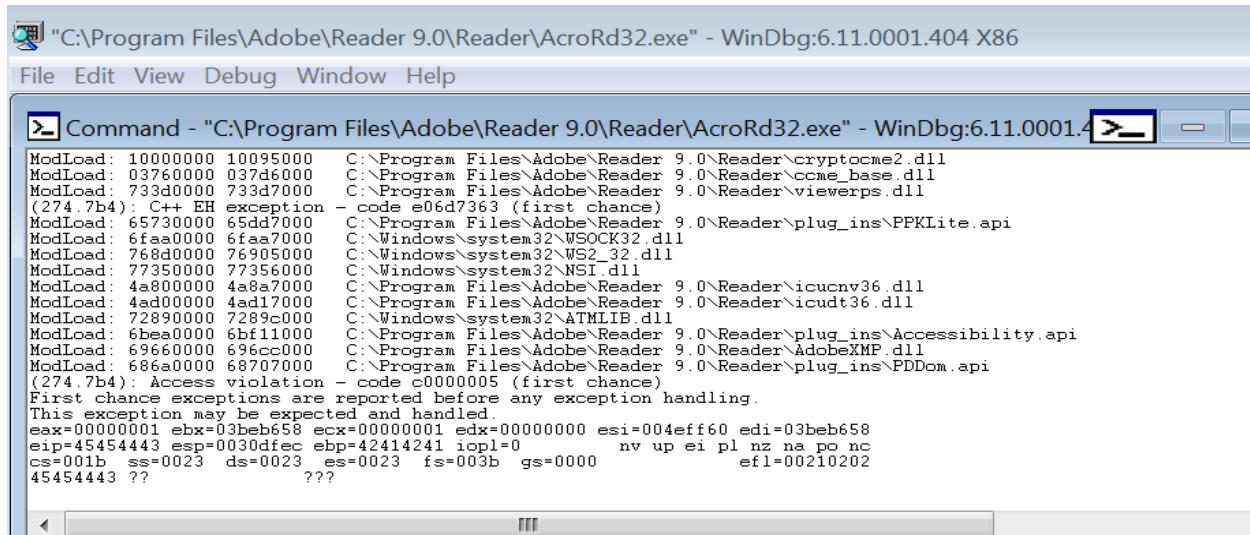
Modifying our input to create a more distinguishable pattern, we see:

```

0:000> g
(62c.180): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=424144b7 ebx=00000400 ecx=42414241 edx=00000002 esi=002be28c edi=00000276
eip=638038d5 esp=002be044 ebp=002be074 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210206
AcroForm!DllUnregisterServer+0x1bd752:
638038d5 8b01                mov     eax,dword ptr [ecx]  ds:0023:42414241=????????

```

And by having identified the location in our input, we can finally demonstrate flow control:



The screenshot shows the WinDbg interface with the command window open. The title bar reads "C:\Program Files\Adobe\Reader 9.0\Reader\AcroRd32.exe" - WinDbg:6.11.0001.404 X86. The command window displays the following text:

```

Command - "C:\Program Files\Adobe\Reader 9.0\Reader\AcroRd32.exe" - WinDbg:6.11.0001.404 X86
ModLoad: 10000000 10095000 C:\Program Files\Adobe\Reader 9.0\Reader\cryptocme2.dll
ModLoad: 03760000 037d6000 C:\Program Files\Adobe\Reader 9.0\Reader\ccme_base.dll
ModLoad: 733d0000 733d7000 C:\Program Files\Adobe\Reader 9.0\Reader\viewerps.dll
(274.7b4): C++ EH exception - code e06d7363 (first chance)
ModLoad: 65730000 65dd7000 C:\Program Files\Adobe\Reader 9.0\Reader\plug_ins\PPKLite.api
ModLoad: 6faa0000 6faa7000 C:\Windows\system32\WSOCK32.dll
ModLoad: 768d0000 76905000 C:\Windows\system32\WS2_32.dll
ModLoad: 77350000 77356000 C:\Windows\system32\NSI.dll
ModLoad: 4a800000 4a8a7000 C:\Program Files\Adobe\Reader 9.0\Reader\icucnv36.dll
ModLoad: 4ad00000 4ad17000 C:\Program Files\Adobe\Reader 9.0\Reader\icudt36.dll
ModLoad: 72890000 7289c000 C:\Windows\system32\ATMLIB.dll
ModLoad: 6bea0000 6bf11000 C:\Program Files\Adobe\Reader 9.0\Reader\plug_ins\Accessibility.api
ModLoad: 69660000 696cc000 C:\Program Files\Adobe\Reader 9.0\Reader\AdobeXMP.dll
ModLoad: 686a0000 68707000 C:\Program Files\Adobe\Reader 9.0\Reader\plug_ins\PDdom.api
(274.7b4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=03beb658 ecx=00000001 edx=00000000 esi=004eff60 edi=03beb658
eip=45454443 esp=0030dfec ebp=42414241 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210202
45454443 ??          ???

```

Because this case is a simple, classical stack-based buffer overflow and our crash was so near the exploitable condition, the analysis is quite simplified and did not require real forward analysis. We included it here to be very clear on the limitations (even for a simple stack-based buffer overflow we needed a bit of cheating and a lot of manual work). The next example is more elaborate to demonstrate the capabilities of the tool.

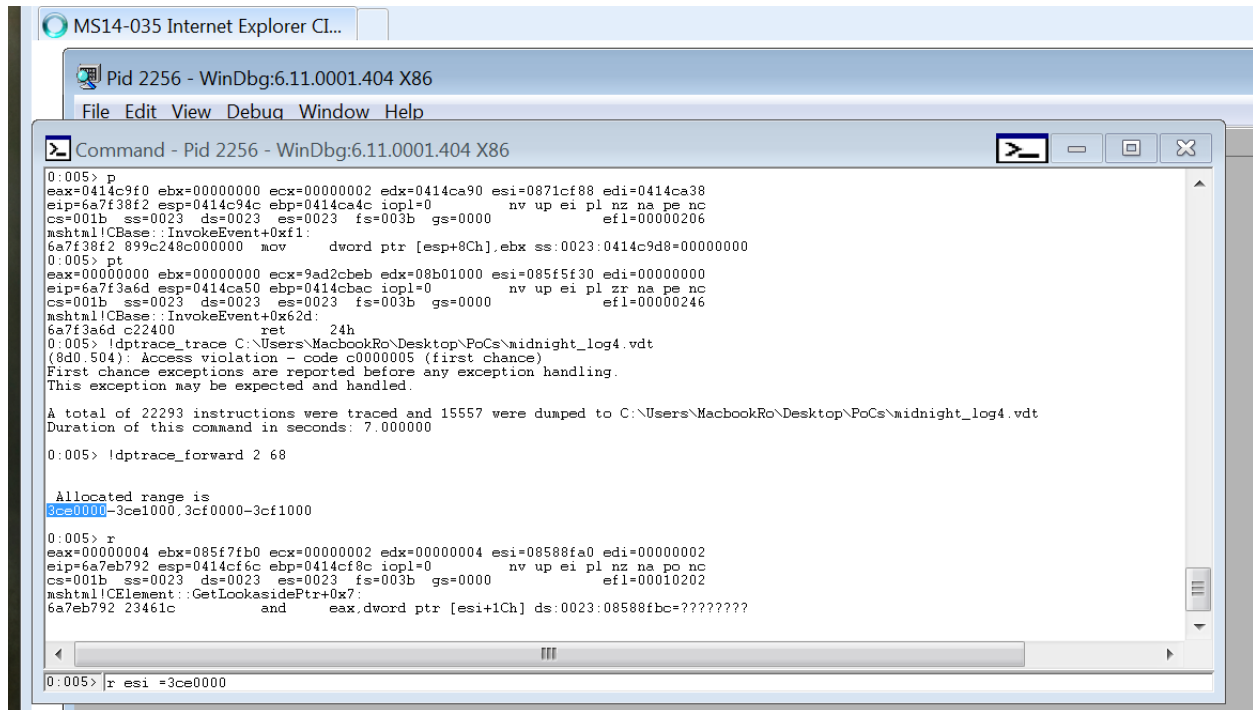
### CVE-2014-0282 IE8/9/10/11 'Cinput' Use-After-Free (MS14-035)

This is a Cinput Object Use after Free that was reported by ZDI researcher Simon Zuckerbraun and patched by Microsoft in June 2014. According to Microsoft it affects all versions of IE from 6 to 11. The PoC Trigger is available on exploit-db since August 2014 (<https://www.exploit-db.com/exploits/33860/>) and recently there have been public write-ups on how to convert the PoC to a "calc-popping" exploit\* on different versions of IE(8 and 11) albeit on Windows XP SP3 (exploits rely on the non-aslr msvcr71.dll binary for the information leak). These can be easily ported to other versions of IE and Windows utilizing other info leaks like hxds.dll (Windows7 and Microsoft Office 2007 and 2010. Windows 8 has ForceASLR kernel flag☺)

\* <http://www.cybersphinx.com/exploiting-cve-2014-0282-ms-035-cinput-use-after-free-vulnerability/>

\* <https://www.nccgroup.trust/globalassets/our-research/uk/whitepapers/2015/12/cve-2014-0282pdf/>

The analysis below was done on Windows 7 SP1 32 bit with IE 11 and page heap enabled for iexplore.exe (full)



The screenshot shows the WinDbg interface with the following components:

- MS14-035 Internet Explorer CL...** (Process Name)
- Pid 2256 - WinDbg:6.11.0001.404 X86** (Process ID and Architecture)
- File Edit View Debug Window Help** (Menu Bar)
- Command - Pid 2256 - WinDbg:6.11.0001.404 X86** (Command Window Title)
- Command Window Content:**

```
0:005> p
eax=0414c9f0 ebx=00000000 ecx=00000002 edx=0414ca90 esi=0871cf88 edi=0414ca38
eip=6a7f38f2 esp=0414c94c ebp=0414ca4c iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
mshtml!CBase::InvokeEvent+0xf1:
6a7f38f2 899c248c00000000  mov     dword ptr [esp+8Ch],ebx  ss:0023:0414c9d8=00000000
0:005> pt
eax=00000000 ebx=00000000 ecx=9ad2cbeb edx=08b01000 esi=085f5f30 edi=00000000
eip=6a7f3a6d esp=0414ca50 ebp=0414cbac iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CBase::InvokeEvent+0x62d:
6a7f3a6d c22400          ret     24h
0:005> !dptrace_trace C:\Users\MacbookRo\Desktop\PoCs\midnight_log4.vdt
(8d0.504): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.

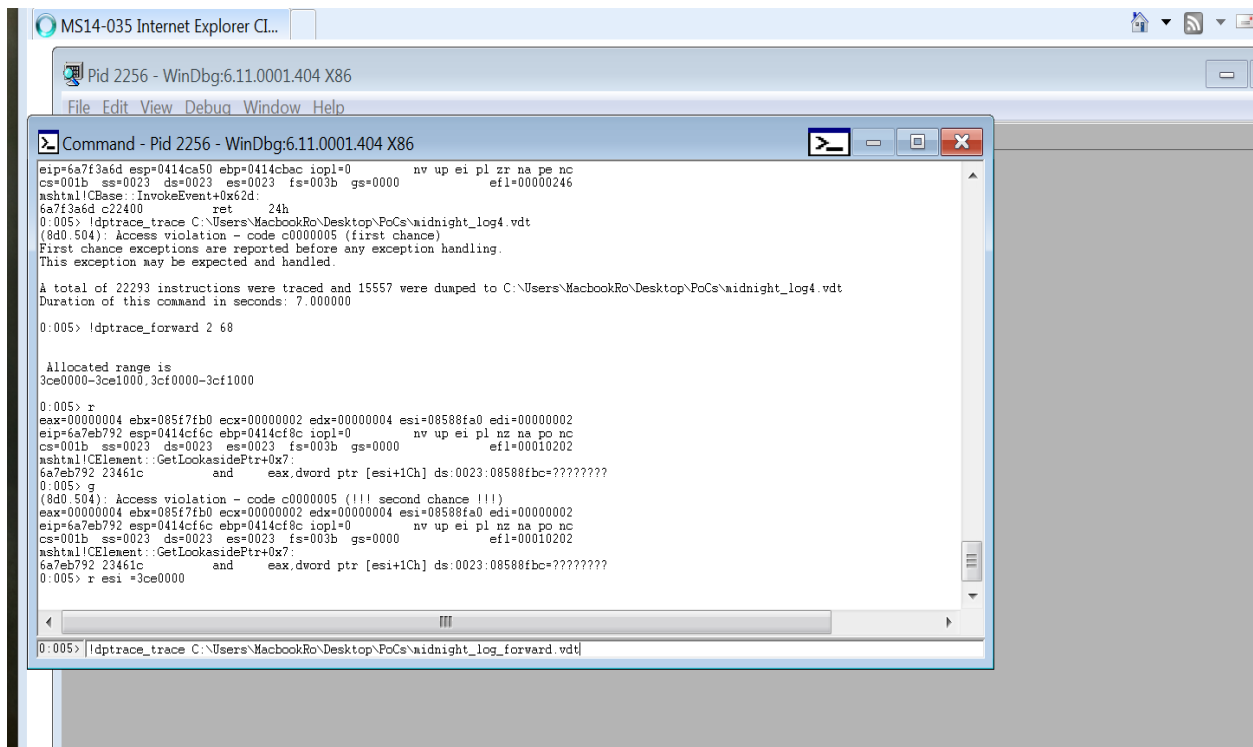
A total of 22293 instructions were traced and 15557 were dumped to C:\Users\MacbookRo\Desktop\PoCs\midnight_log4.vdt
Duration of this command in seconds: 7.000000

0:005> !dptrace_forward 2 68

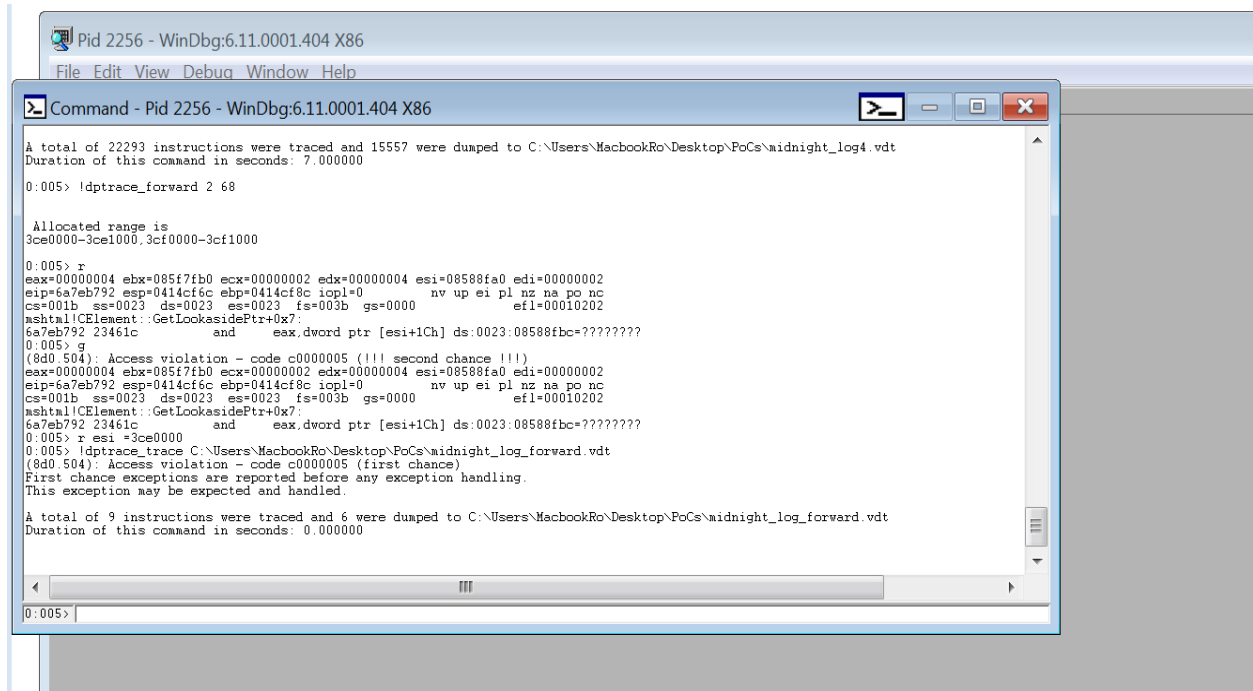
Allocated range is
3ce0000-3ce1000,3cf0000-3cf1000

0:005> r
eax=00000004 ebx=085f7fb0 ecx=00000002 edx=00000004 esi=08588fa0 edi=00000002
eip=6a7eb792 esp=0414cf6c ebp=0414cf8c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
mshtml!CElement::GetLookasidePtr+0x7:
6a7eb792 23461c          and     eax,dword ptr [esi+1Ch]  ds:0023:08588fbc=????????
0:005> r esi =3ce0000
```

The above figure illustrates the tracing file being generated (!dptrace\_trace) and the forward analysis allocation prepared (!dptrace\_forward).

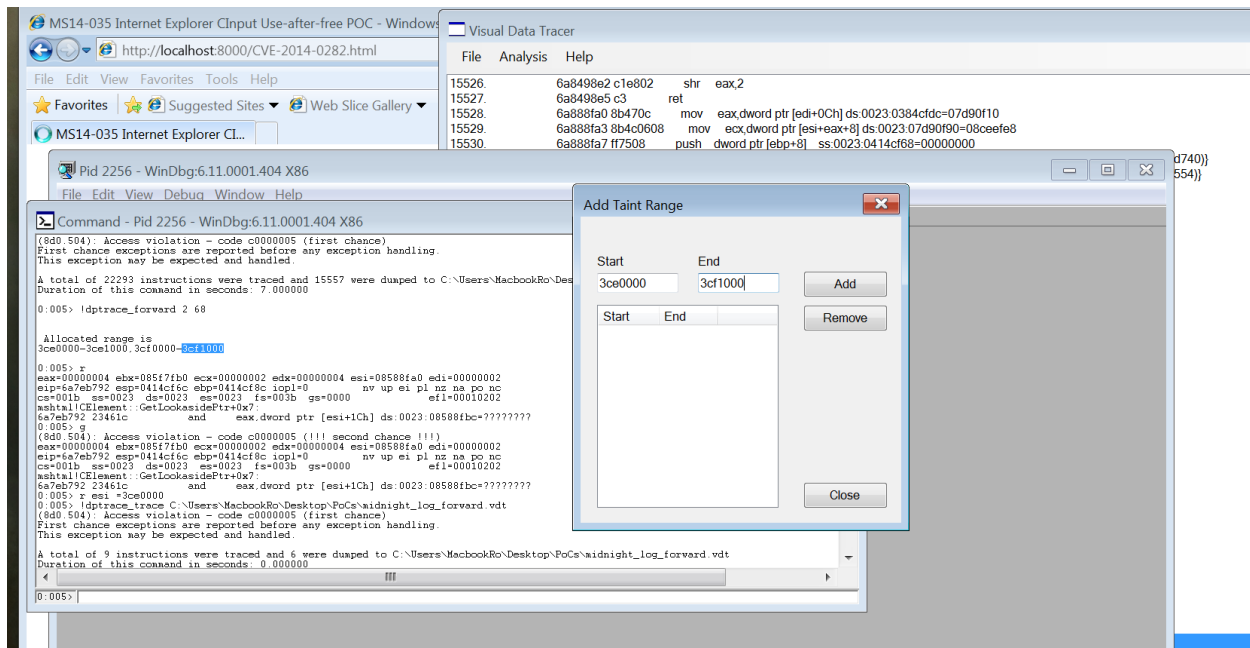


When the crash is triggered, the analyst can set the invalid pointer to the location returned by the forward allocations (r esi= command). Also note that the first triggering of the bug already sets the exception, meaning that one must set the value after letting it go. For this issue, we are replacing the freed object with the root of the fake object chain.

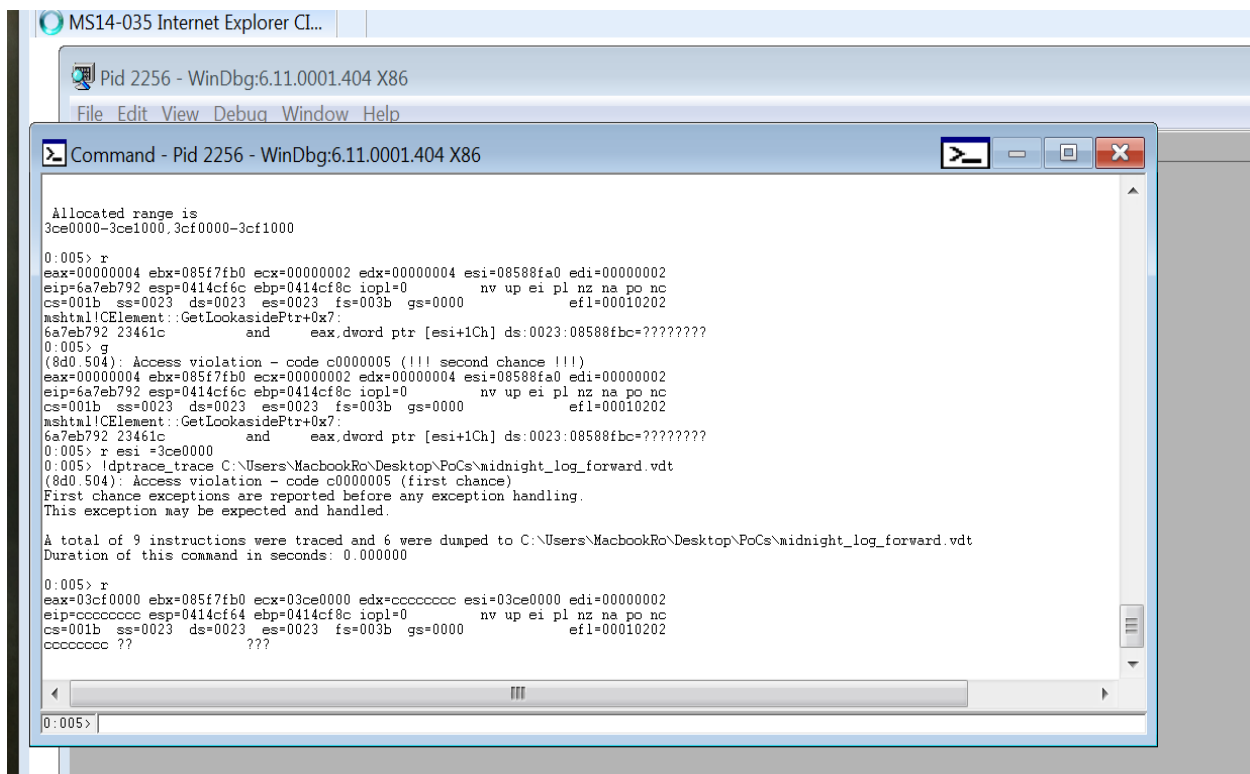




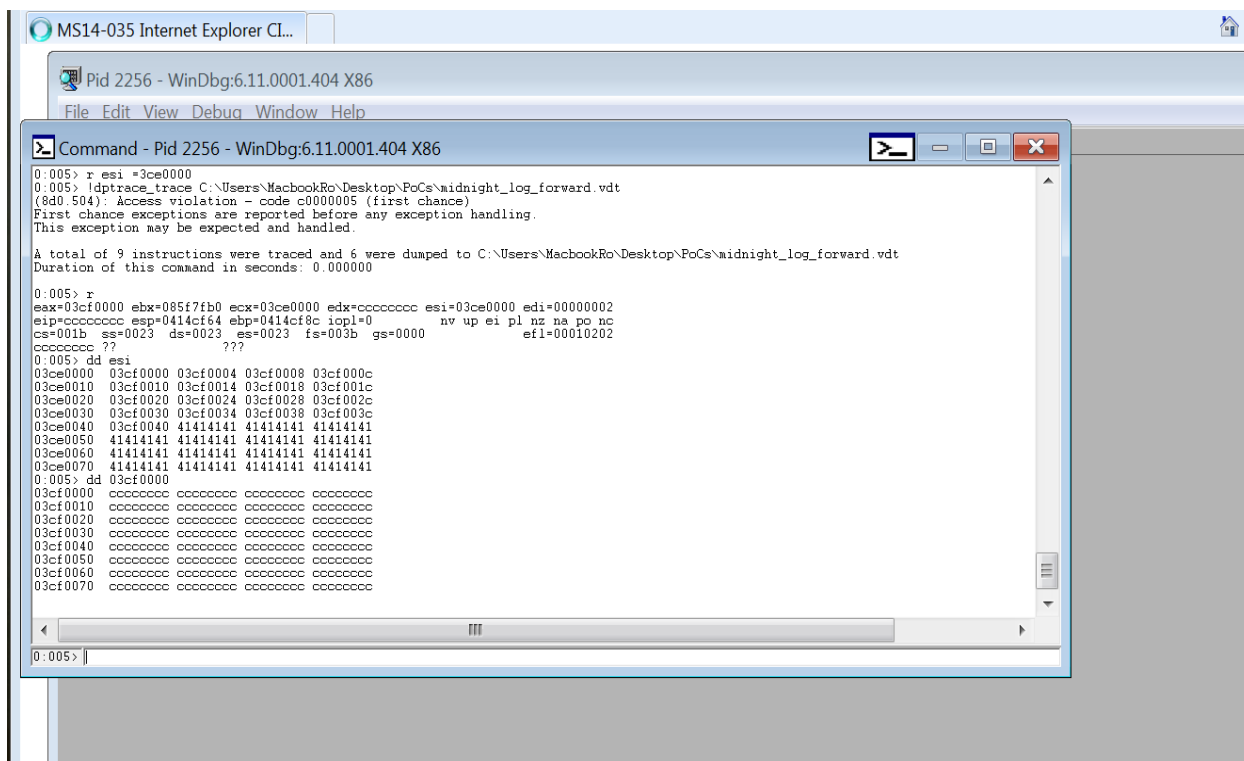
We let the execution continue, and trace until the subsequent breakpoint/access violation. To be sure that the freed object is indeed controlled, we can use the analyzer to see the tainted value. We first define the range of interest:



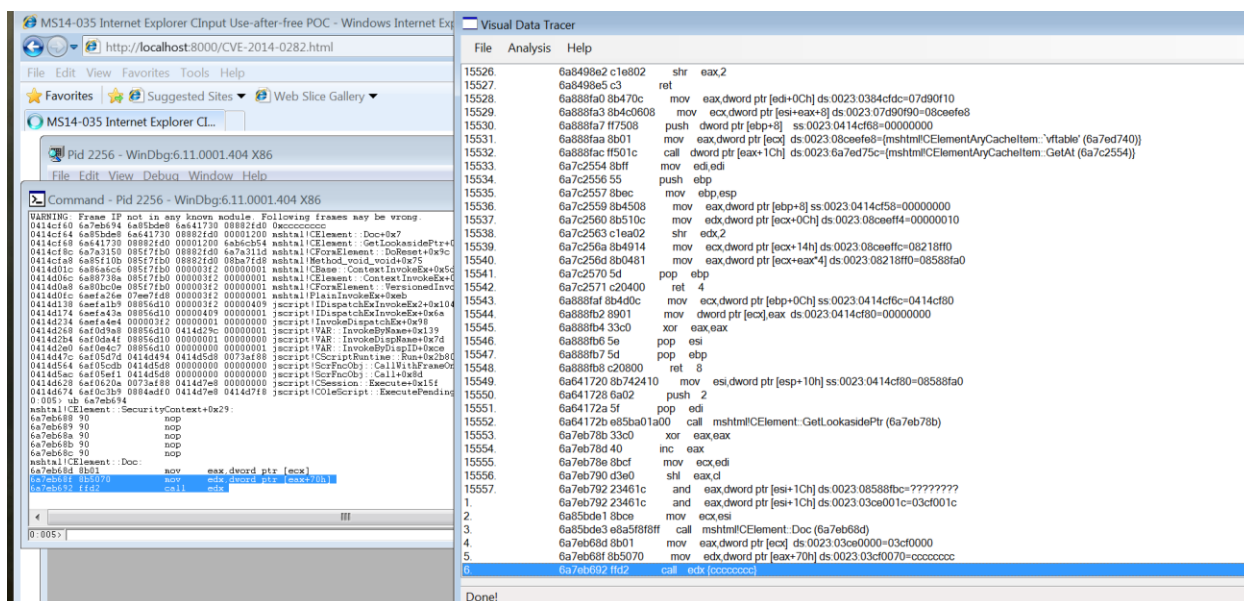
We let the execution go in the debugger:



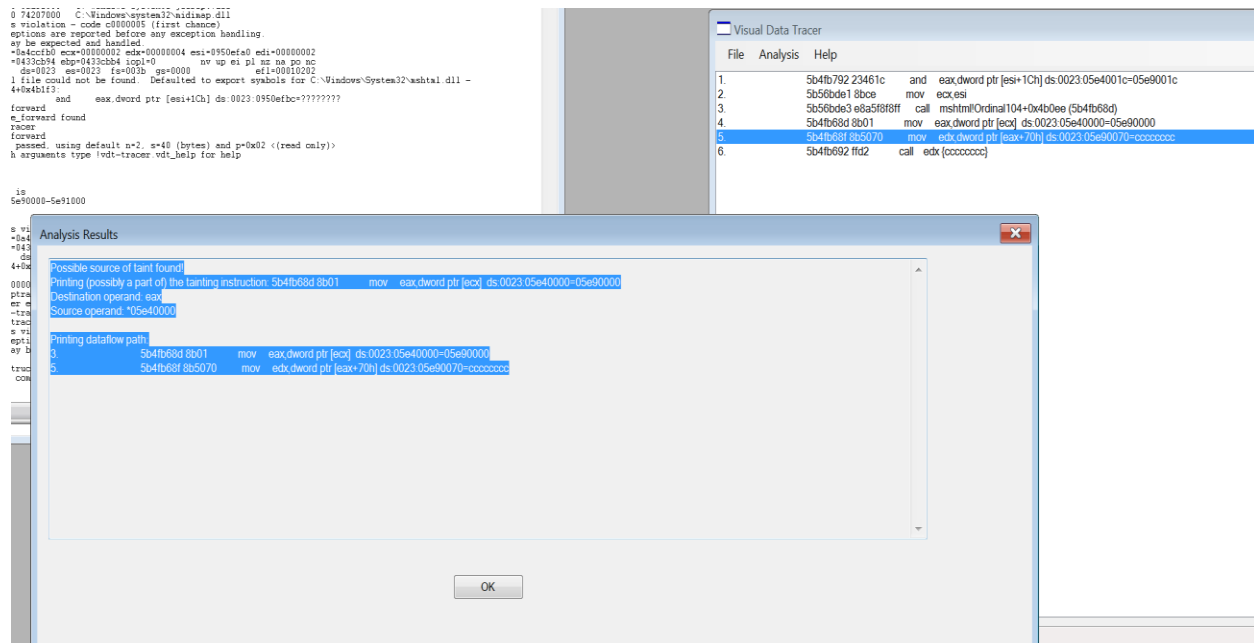
If we look the crash moment and our fake forward structures, we have:



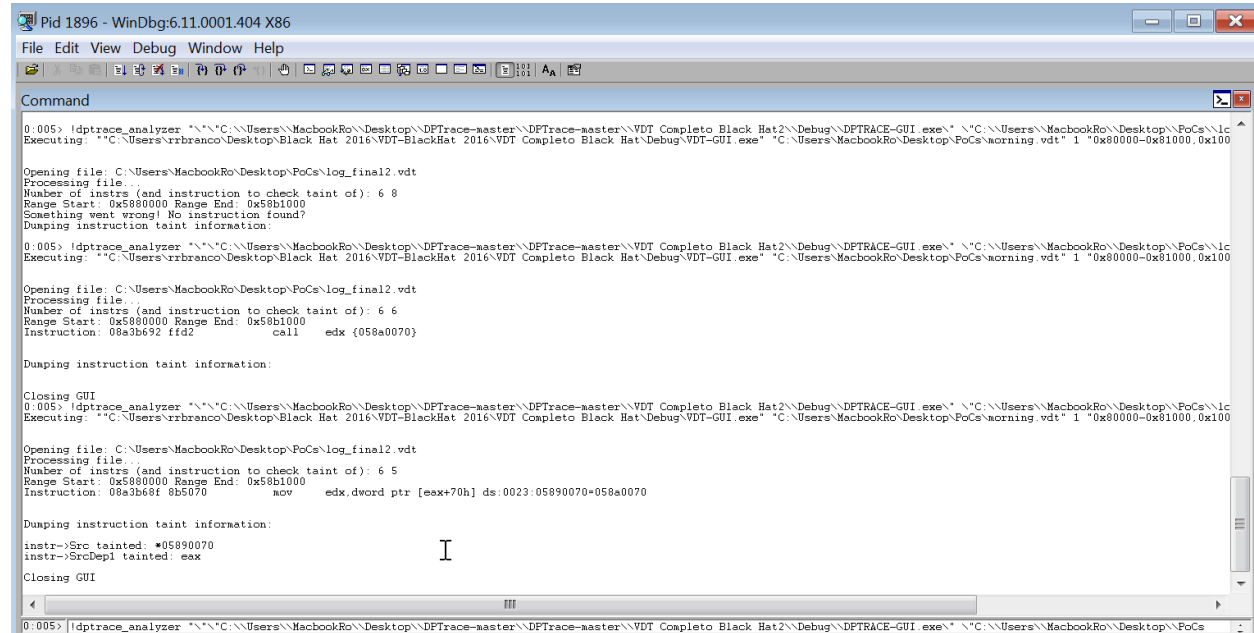
Program control seems evident from the forward analysis view (EIP comes from our faked structure), so we need to be sure we control the initial violation:



And finally if we look the first crashing point:



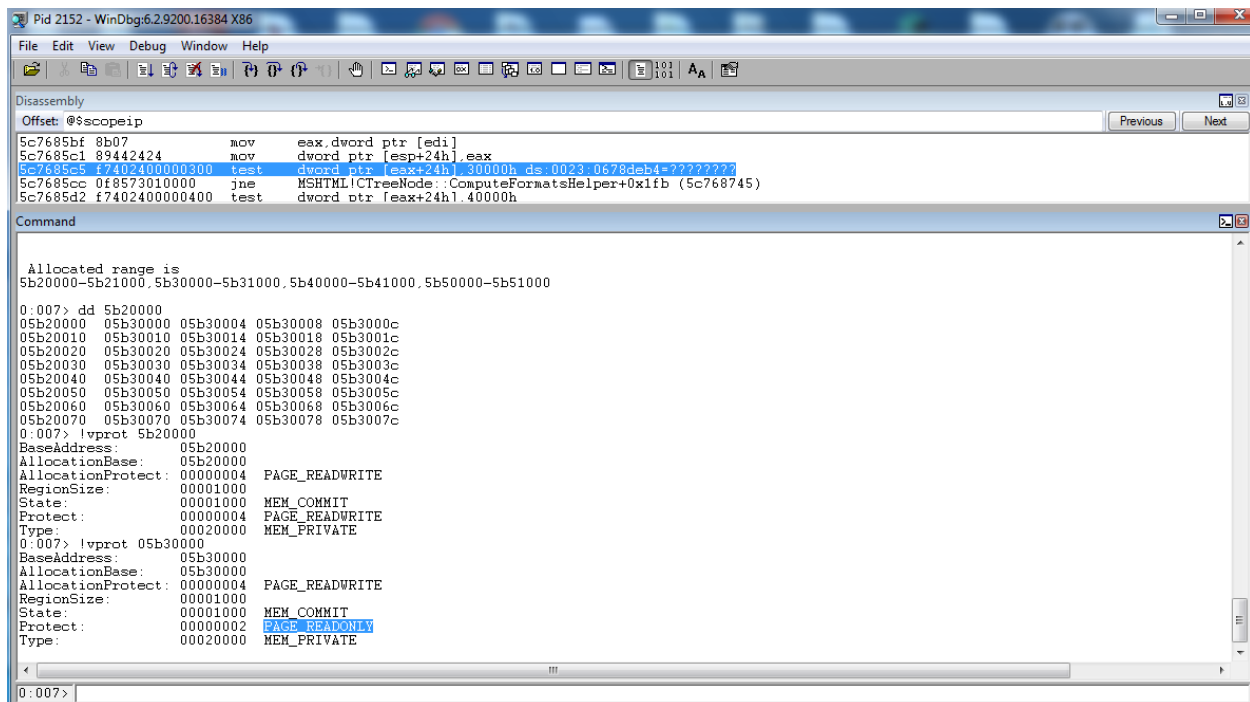
Taint source is confirmed. This is shown here using the GUI, but can also be done directly from the debugger (adding the ranges and collecting the taint information before even proceeding to the forward analysis):



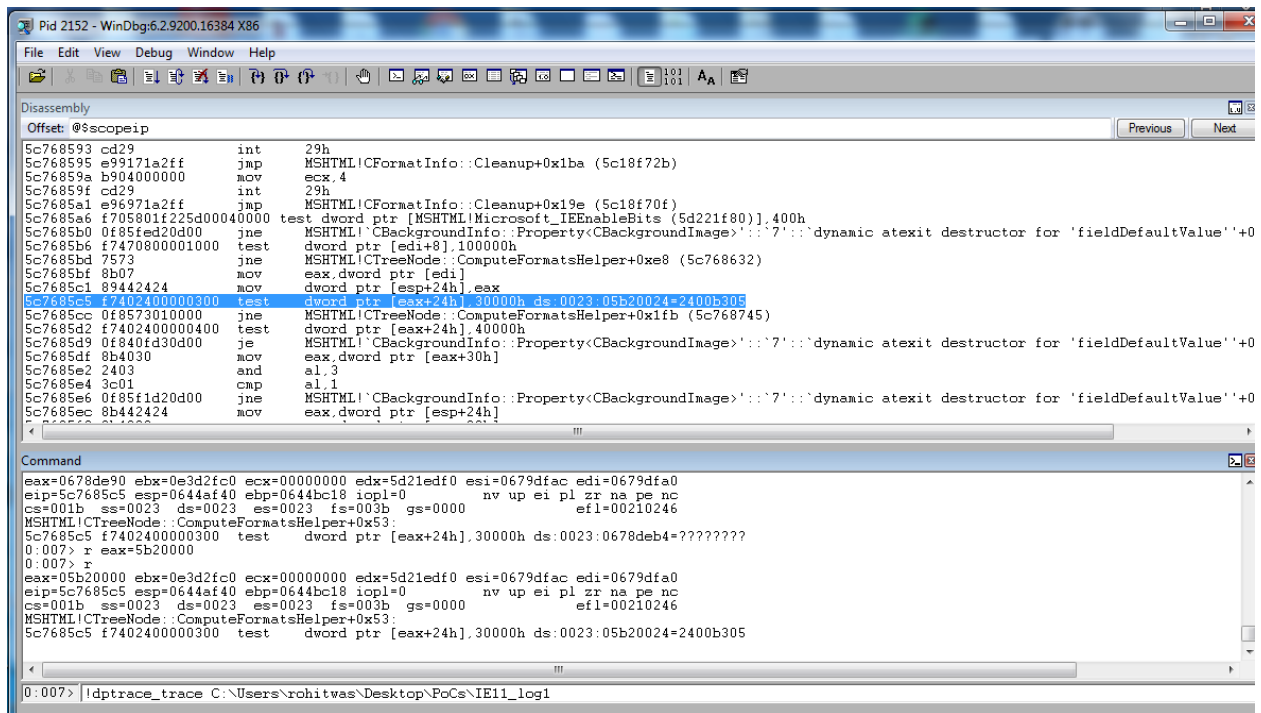
This is a CObjectElement Use-After-Free that occurs in the CTreeNode::ComputeFormatsHelper function in MSHTML which was reported by Moritz Jodeit of Blue Frost Security in August 2015 and patched by Microsoft in December 2016 in MS15-124 bulletin. There is publicly available PoC that crashes vulnerable version of IE 11\* but no public exploit code at the time of writing this.

The analysis below was done on Windows 7 SP1 32 bit with IE 11, Memory Protection Feature turned off in the Registry setting and page heap enabled (full).

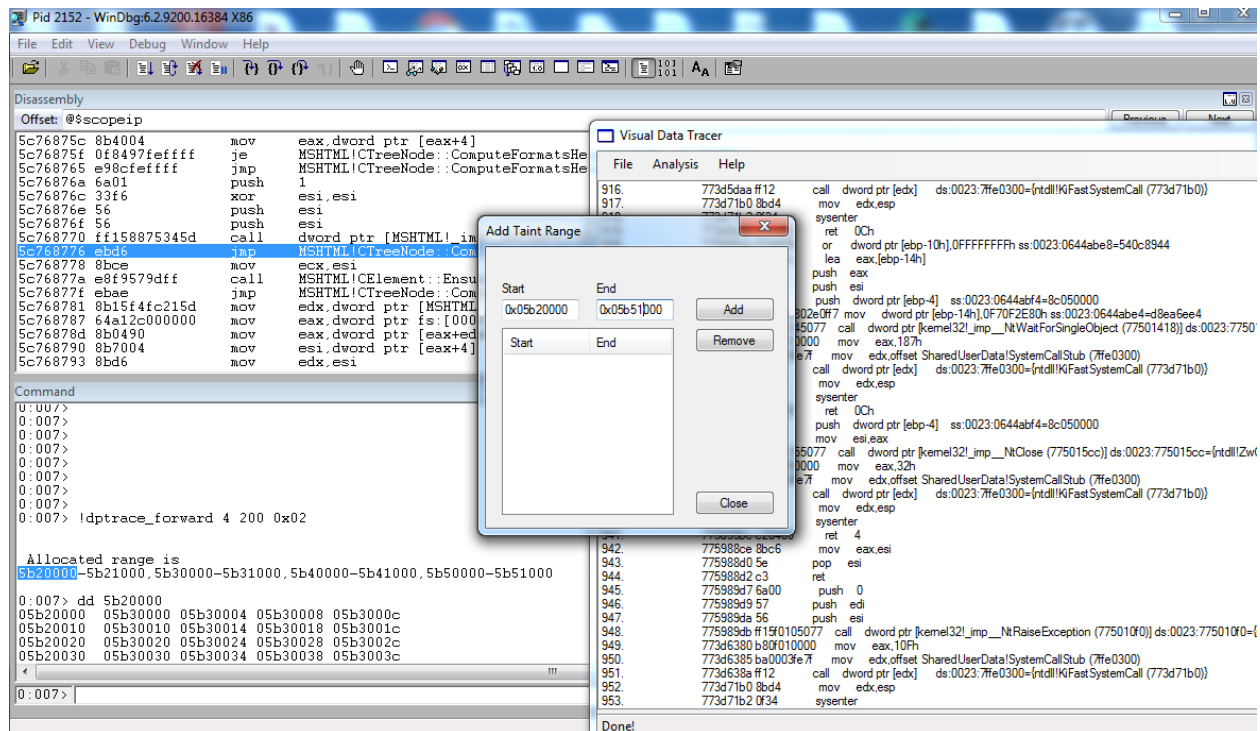
The above figure illustrates the forward analysis preparation (ldptrace\_forward) at the moment of the crash. In this case, we proceed with the forward analysis since IE issues are sometimes complex for tracing from the beginning (number of instructions, JITed code, etc).



We selected a fake object chain of 4 objects of size 200 (this is not precise, just a rough estimate). This is enough as any attempts to access outside the range will be caught anyway. Precise size can be determined by manual analysis to figure out the freed/alloc'd function and checking the size of the root object. Memory permissions are set as READ\_ONLY for all the other objects except the first one (as seen in the picture).



We define the offending register to point to our fake object.



Since we got the ranges from the forward tracer, we define them in the GUI as the ones we are interested. We want to know if the source of an access violation can be traced back to controlled input.

```
J:007> ed eax+24 300
J:007> r
eax=05ba0000 ebx=0ea4bfc0 ecx=00000000 edx=5d36edf0 esi=0616ffac edi=0616ffa0
eip=5c8b85c5 esp=05e1b320 ebp=05e1bfe8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
MSHTML!CTreeNode::ComputeFormatsHelper+0x53:
5c8b85c5 f7402400000300 test     dword ptr [eax+24h],30000h ds:0023:05ba0024=00030000
```

This is an example of other paths of execution, we can manually force them as above (a limitation of the tool is we do not do symbolic execution, neither make multiple execution attempts).

Disassembly

Offset: @\$scopeip

```

5c18f7ed 83e4e0    and     esp,0FFFFFFE0h
5c18f7f0 83ec78    sub     esp,78h
5c18f7f3 a1acf8215d mov     eax,dword ptr [MSHTML!_security_cookie (5d21f8ac)]
5c18f7f8 33c4     xor     eax,esp
5c18f7fa 89442474  mov     dword ptr [esp+74h],eax
5c18f7fe 8bc1     mov     eax,ecx
5c18f800 56       push    esi
5c18f801 8b750c    mov     esi,dword ptr [ebp+0Ch]
5c18f804 57       push    edi
5c18f805 8b4824    mov     ecx,dword ptr [eax+24h] ds:0023:3f800024=????????
5c18f808 8b7d08    mov     edi,dword ptr [ebp+8]
5c18f80b 89442410  mov     dword ptr [esp+10h],eax
5c18f80f 897c241c  mov     dword ptr [esp+1Ch],edi
5c18f813 f7c100003000 test    ecx,30000h
5c18f819 0f858b7a5d00 jne     MSHTML!CElement::ComputeFormats+0x55b (5c7672aa)
5c18f81f f7c100004000 test    ecx,40000h
5c18f825 0f84dc215600 je      MSHTML!CElement::ComputeFormats+0x5eb (5c6f1a07)
5c18f82b 8b4030    mov     eax,dword ptr [eax+30h]

```

Command

```

0:007> r eax=5b20000
0:007> !dptrace_trace C:\Users\rohitwas\Desktop\PoCs\IE11_log3
(868.54c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.

A total of 26 instructions were traced and 23 were dumped to C:\Users\rohitwas\Desktop\PoCs\IE11_log3
Duration of this command in seconds: 0.000000

0:007> r
eax=3f800000 ebx=0e3d2fc0 ecx=3f800000 edx=00000000 esi=0644ac5c edi=0644ac5c
eip=5c18f805 esp=0644aea0 ebp=0644af24 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210202
MSHTML!CElement::ComputeFormats+0x1d:
5c18f805 8b4824    mov     ecx,dword ptr [eax+24h] ds:0023:3f800024=????????

```

As we analyze the binary, more constraints appear and can be manually forced to be true. We can always check the taint source to make sure we could satisfy it:

Disassembly

Offset: @\$scopeip

```

5c18f7ed 83e4e0    and     esp,0FFFFFFE0h
5c18f7f0 83ec78    sub     esp,78h
5c18f7f3 a1acf8215d mov     eax,dword ptr [MSHTML!_security_cookie (5d21f8ac)]
5c18f7f8 33c4     xor     eax,esp
5c18f7fa 89442474  mov     dword ptr [esp+74h],eax
5c18f7fe 8bc1     mov     eax,ecx
5c18f800 56       push    esi
5c18f801 8b750c    mov     esi,dword ptr [ebp+0Ch]
5c18f804 57       push    edi
5c18f805 8b4824    mov     ecx,dword ptr [eax+24h] ds:0023:3f800024=????????
5c18f808 8b7d08    mov     edi,dword ptr [ebp+8]
5c18f80b 89442410  mov     dword ptr [esp+10h],eax
5c18f80f 897c241c  mov     dword ptr [esp+1Ch],edi
5c18f813 f7c100003000 test    ecx,30000h
5c18f819 0f858b7a5d00 jne     MSHTML!CElement::ComputeFormats+0x55b (5c7672aa)
5c18f81f f7c100004000 test    ecx,40000h
5c18f825 0f84dc215600 je      MSHTML!CElement::ComputeFormats+0x5eb (5c6f1a07)
5c18f82b 8b4030    mov     eax,dword ptr [eax+30h]

```

Command

```

0:007> r eax=5b20000
0:007> !dptrace_trace C:\Users\rohitwas\Desktop\PoCs\IE11_log3
(868.54c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.

A total of 26 instructions were traced and 23 were dumped to C:\Users\rohitwas\Desktop\PoCs\IE11_log3
Duration of this command in seconds: 0.000000

0:007> r
eax=3f800000 ebx=0e3d2fc0 ecx=3f800000 edx=00000000 esi=0644ac5c edi=0644ac5c
eip=5c18f805 esp=0644aea0 ebp=0644af24 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210202
MSHTML!CElement::ComputeFormats+0x1d:
5c18f805 8b4824    mov     ecx,dword ptr [eax+24h] ds:0023:3f800024=????????

```

Visual Data Tracer

File	Analysis	Help
1	5c76875c 8b4004    mov     eax,dword ptr [eax+4] ds:0023:05b20004=0400e305	
2	5c7687f6 8b80b8000000 mov     eax,dword ptr [eax+08h] ds:0023:05b300bc-bc00b405	
3	5c7687fc 8b400c    mov     eax,dword ptr [eax+0Ch] ds:0023:05b400c8-41414141	
4	5c7687ff 8b07    mov     ecx,dword ptr [edi] ds:0023:0644ac5c=0000803f	
5	5c768801 8984249400000000 mov     dword ptr [esp+94h],eax ss:0023:0644af30-e85e7406	
6	5c768808 8d84249000000000 lea     eax,[esp+90h]	
7	5c76880f 57       push    edi	
8	5c768810 50       push    eax	
9	5c768811 89bc249800000000 mov     dword ptr [esp+98h],edi ss:0023:0644af30-906f0005	
10	5c768818 e8cb71a2ff    call    MSHTML!CElement::ComputeFormats (5c18f7e8)	
11	5c18f7e8 8b07    mov     edi,edi	
12	5c18f7ea 55       push    ebp	
13	5c18f7eb 8bec     mov     ebp,esp	
14	5c18f7ed 83e4e0    and     esp,0FFFFFFE0h	
15	5c18f7f0 83ec78    sub     esp,78h	
16	5c18f7f3 a1acf8215d mov     eax,dword ptr [MSHTML!_security_cookie (5d21f8ac)] ds:0023:5d21f8ac-f3362f42	
17	5c18f7f8 33c4     xor     eax,esp	
18	5c18f7fa 89442474  mov     dword ptr [esp+74h],eax ss:0023:0644af1c=00000000	
19	5c18f7fe 8bc1     mov     eax,ecx	
20	5c18f800 56       push    esi	
21	5c18f801 8b750c    mov     esi,dword ptr [ebp+0Ch] ss:0023:0644af30-5cac4406	
22	5c18f804 57       push    edi	
23	5c18f805 8b4824    mov     ecx,dword ptr [eax+24h] ds:0023:3f800024=????????	

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.

```

Pid 3204 - WinDbg:6.2.9200.16384 X86
File Edit View Debug Window Help
Disassembly
Offset: @$scopeip
5c8b85b6 f7470800001000 test dword ptr [edi+8],100000h
5c8b85bd 7573 jne MSHTML!CTreeNode::ComputeFormatsHelper+0xe8 (5c8b8632)
5c8b85bf 8b07 mov eax,dword ptr [edi]
5c8b85c1 89442424 mov dword ptr [esp+24h],eax
5c8b85c5 f7402400000300 test dword ptr [eax+24h],30000h ds:0023:0e858eb4=????????
5c8b85cc 0f8573010000 jne MSHTML!CTreeNode::ComputeFormatsHelper+0x1fb (5c8b8745)
5c8b85d2 f7402400000400 test dword ptr [eax+24h],40000h

Command
ModLoad: 73600000 7360a000 C:\Windows\system32\ddrawex.dll
ModLoad: 5e870000 5e957000 C:\Windows\system32\DDRAW.dll
ModLoad: 73470000 73476000 C:\Windows\system32\DCIMAN32.dll
(c84.fc8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0e858e90 ebx=0f776fc0 ecx=00000000 edx=5d36edf0 esi=0e85cfac edi=0e85cfa0
eip=5c8b85c5 esp=05d8b500 ebp=05d8c1d0 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010246
MSHTML!CTreeNode::ComputeFormatsHelper+0x53:
5c8b85c5 f7402400000300 test dword ptr [eax+24h],30000h ds:0023:0e858eb4=????????
0:008> g
(c84.fc8): Access violation - code c0000005 (!!! second chance !!!)
eax=0e858e90 ebx=0f776fc0 ecx=00000000 edx=5d36edf0 esi=0e85cfac edi=0e85cfa0
eip=5c8b85c5 esp=05d8b500 ebp=05d8c1d0 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010246
MSHTML!CTreeNode::ComputeFormatsHelper+0x53:
5c8b85c5 f7402400000300 test dword ptr [eax+24h],30000h ds:0023:0e858eb4=????????
0:008> !dprtrace_forward 4 200

Allocated range is
55e0000-55e1000,55f0000-55f1000,65f0000-65f1000,6600000-6601000

```

We start similarly at the crash point and allocate fake objects.

```

Pid 3204 - WinDbg:6.2.9200.16384 X86
File Edit View Debug Window Help
Disassembly
Offset: @$scopeip
5c8b85b6 f7470800001000 test dword ptr [edi+8],100000h
5c8b85bd 7573 jne MSHTML!CTreeNode::ComputeFormatsHelper+0xe8 (5c8b8632)
5c8b85bf 8b07 mov eax,dword ptr [edi]
5c8b85c1 89442424 mov dword ptr [esp+24h],eax
5c8b85c5 f7402400000300 test dword ptr [eax+24h],30000h ds:0023:055e0024=00000400
5c8b85cc 0f8573010000 jne MSHTML!CTreeNode::ComputeFormatsHelper+0x1fb (5c8b8745)
5c8b85d2 f7402400000400 test dword ptr [eax+24h],40000h

Command
(c84.fc8): Access violation - code c0000005 (!!! second chance !!!)
eax=0e858e90 ebx=0f776fc0 ecx=00000000 edx=5d36edf0 esi=0e85cfac edi=0e85cfa0
eip=5c8b85c5 esp=05d8b500 ebp=05d8c1d0 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010246
MSHTML!CTreeNode::ComputeFormatsHelper+0x53:
5c8b85c5 f7402400000300 test dword ptr [eax+24h],30000h ds:0023:0e858eb4=????????
0:008> !dprtrace_forward 4 200

Allocated range is
55e0000-55e1000,55f0000-55f1000,65f0000-65f1000,6600000-6601000
0:008> r eax=55e0000
0:008> dd edi 11
0e85cfa0 0e858e90
0:008> ed 0e85cfa0 0e85cfa0
0:008> ed 0e85cfa0 55e0000
0:008> esp+24 55e0000
0:008> eb eax+30 1
0:008> eb eax+24 40000 1
^ Overflow error in 'eb eax+24 40000 1'
0:008> eb eax+24 40000
^ Overflow error in 'eb eax+24 40000'
0:008> ed eax+24 40000

```

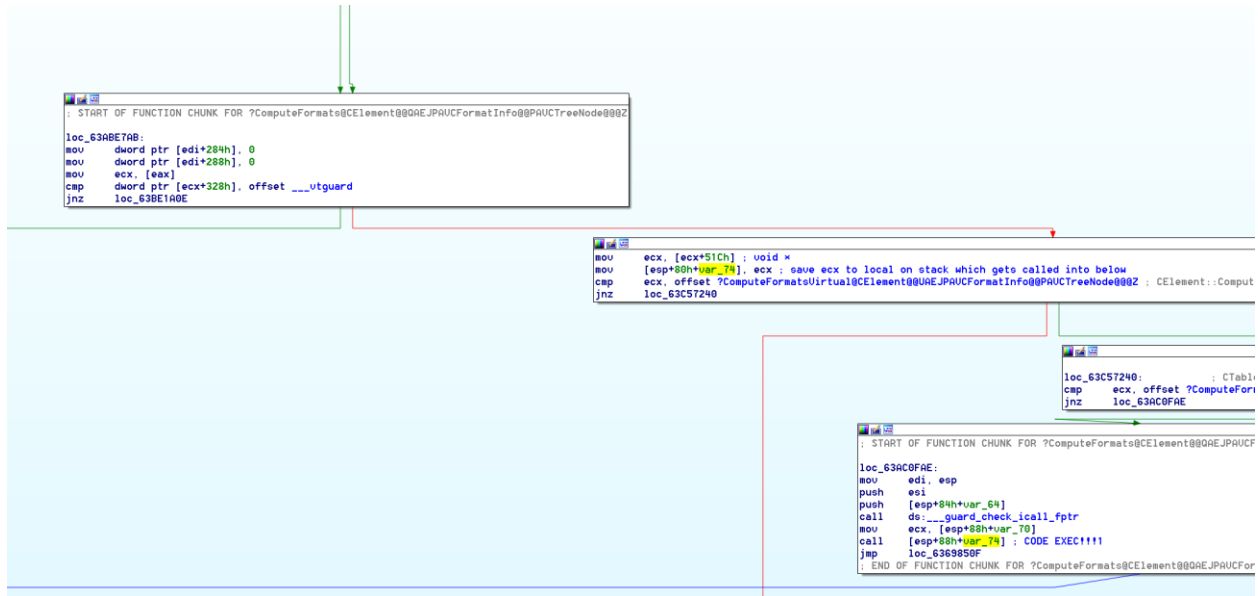


[illegible]

File	Analysis	Help
67.	7f597e0c1e902	shr ecx,2
68.	7f597e9f3ab	rep stos dword ptr [edi]
69.	7f597e9f3ab	rep stos dword ptr [edi]
70.	7f597e9f3ab	rep stos dword ptr [edi]
71.	7f597e9f3ab	rep stos dword ptr [edi]
72.	7f597e9f3ab	rep stos dword ptr [edi]
73.	7f597e9f3ab	rep stos dword ptr [edi]
74.	7f597e9f3ab	rep stos dword ptr [edi]
75.	7f597e9f3ab	rep stos dword ptr [edi]
76.	7f597e9f3ab	rep stos dword ptr [edi]
77.	7f597e9f3ab	rep stos dword ptr [edi]
78.	7f597e9f3ab	rep stos dword ptr [edi]
79.	7f597e9f3ab	rep stos dword ptr [edi]
80.	7f597e9f3ab	rep stos dword ptr [edi]
81.	7f597e9f3ab	rep stos dword ptr [edi]
82.	7f597e9f3ab	rep stos dword ptr [edi]
83.	7f597e9f3ab	rep stos dword ptr [edi]
84.	7f597f3b442408	mov eax,dword ptr [esp+8] ;ds:00000000
85.	7f59777f3	pop edi
86.	7f59788c3	ret
87.	5c2dfb933cd	xor eax,eax
88.	5c2df8b83cd	add esp,0Ch
89.	5c2df8c689442460	mov dword ptr [esp+60h],eax
90.	5c2df8ca89442464	mov dword ptr [esp+64h],eax
91.	5c2df8ce89442468	mov dword ptr [esp+68h],eax
92.	5c2df8d4c7442460ffffff	mov dword ptr [esp+60h],0FFFFFFFh
93.	5c2df8dc84424480	lea eax,[esp+60h]
94.	5c2df8de87442468ffffff	mov dword ptr [esp+64h],0FFFFFFFh
95.	5c2df8ef87442468ffffff	mov dword ptr [esp+68h],0FFFFFFFh
96.	5c2df8f0898788020000	dword ptr [edi+288h],eax
97.	5c2df8fe84442420	lea eax,[esp+20h]
98.	5c2df8fa898784020000	dword ptr [edi+284h],eax
99.	5c2df9003b442410	mov eax,dword ptr [esp+10h]
100.	5c2df9048b08	mov ecx,dword ptr [eax];ds:00230000
101.	5c2fb10a5e01	push 1
102.	5c30410108bf52e0d0	call MSHTML!_report_security_failure
103.	5c3040308ff	mov edi,edi
104.	5c3040255	push ebp

We can trace back the call and confirm that this is indeed controlled by our taint 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 below:



Obviously the thing to note here is that in order to get code execution via the call instruction into a value controlled by us, the vtguard\_check which would necessitate an information leak within MSHTML.

Note that the other two constraints seen above i.e  
cmp ecx, CElement::ComputeFormatsVirtual(CFormatInfo \*,CTreeNode \*) and  
cmp ecx, CTableCell::ComputeFormatsVirtual(CFormatInfo \*,CTreeNode \*) can be conveniently skipped as the path to call [esp+88+var\_74] which calls a pointer @ 0x51c bytes into the tainted object which we control.

This particular case was detailed to demonstrate the difficulty of analysis in some relatively complex cases and certain pitfalls in our approach in getting a definitive answer without a lot of manual intervention and analysis.

## [9] Other existing solutions

Many researchers had similar ideas, or ideas towards the same problem. This does not intend to be a comprehensive list, but instead to give the reader some pointers to understand other approaches and their potential limitations/benefits.

The order is not chronological nor of importance.

### **[9.1] ! exploitable**

This WinDBG plugin by Microsoft [1] is widely known and referenced and is one of the precursors of taint analysis for exploitability determination. It tries to classify unique issues (crashes appearing through different code paths, machines involved in testing and in multiple different test cases) but still with the same root cause. It groups those crashes for analysis and quickly prioritizes issues (since crashes appear in thousands, while analysis capabilities are very limited).

The main drawback of the approach is that it assumes the attacker has full control at the elements in the crash point, and then do a forward analysis to verify if with that, there are clearly exploitable conditions. That leads to lots of false negatives (limitations on defining an exploitable condition), as well with false positives. It provides a lot of value though as an easy initial triage.

### **[9.2] Spider Pig**

Created by Piotr Bania, the tool is not available for testing/evaluation but has details in a published paper. It is much more advanced than the tool provided here (but well, it is not available):

- Has Virtual Code Integration (Dynamic Binary Rewriting)
- Disputable Objects: Partially controlled data is analyzed using the parent data

### **[9.3] Taint Bochs**

Used for tracking sensitive data lifecycle in memory, the objectives are quite different than the ones on crash analysis. Nonetheless worth having a deeper look.

### **[9.3] Taint Check**

Uses DynamicRIO or Valgrind and provides:

- Taint Seed: Defining the tainted values (data coming from the network for example)
- Taint Tracker: Tracks the propagation
- Taint Assert: Alert about a security violation

It is quite interesting to use while testing software to detect potential security issues, but not really useful for the exploit creation process itself.

### **[9.4] Bitblaze**

Interesting platform for binary analysis, provides better classification of exploitability (accordingly to Charlie Miller talk in Black Hat). Can be used as a base platform for VINE. It is a moflow framework.

## [9.5] Cisco Talos Moflow Framework

They've recently released a bunch of analysis tools based on the CMU's BAP framework. The tools perform symbolic execution (therefore eliminating one of our limitations related to finding paths that are not traced). The drawback is potential performance problems for complex paths (like the ones we analyze here), which would be quite slow with the full approach. They are by far the most worthy checking by anyone interested in this kind of research.

The tools perform post-crash graph back taint slicer and forward symbolic emulation (looking for more exploitable conditions).

## [10] Acknowledgements

<In no particular order>

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

David D. Rude and Kiran Bandla for ideas and feedback regarding the initial prototype of the forward trace.

## [11] References

This paper is quite weak in references, basically because of laziness – we cite many tools in Section 9 but do not provide references to them. They are all very easy to find in any search engine ☺

[1] Microsoft !exploitable. Link: <https://msecdbg.codeplex.com/>

[2] BSDaemon. "Dynamic Program Analysis and Software Exploitation: From the crash to the exploit code", Volume 0x0e, Issue 0x43, Phrack. Link: <http://phrack.org/issues/67/10.html>

[3] Intel Corporation. Link:

[http://www.intel.com/software/products/documentation/vlin/mergedprojects/analyzer\\_ec/mergedprojects/reference\\_olh/mergedProjects/instructions/instruct32\\_hh/vc42.htm](http://www.intel.com/software/products/documentation/vlin/mergedprojects/analyzer_ec/mergedprojects/reference_olh/mergedProjects/instructions/instruct32_hh/vc42.htm)

[4] BFS algorithm. Link: [http://en.wikipedia.org/wiki/Breadth-first\\_search](http://en.wikipedia.org/wiki/Breadth-first_search)

[5] GitHub Repository for this Paper/Presentation: Link: