DPTrace

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

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 makes 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.

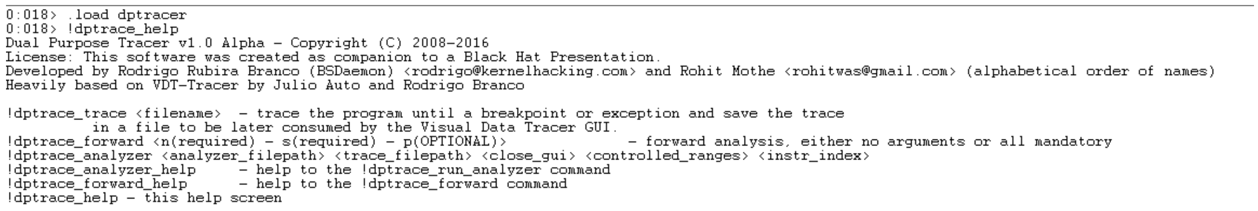
After determining the input control and the influence of the input over the program crash, we want to go further and try to solve the actual program control problem. The technique we propose here is of forward taint propagation and the idea is pretty similar to the backward taint (even though the implementation is completely different, since we 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).

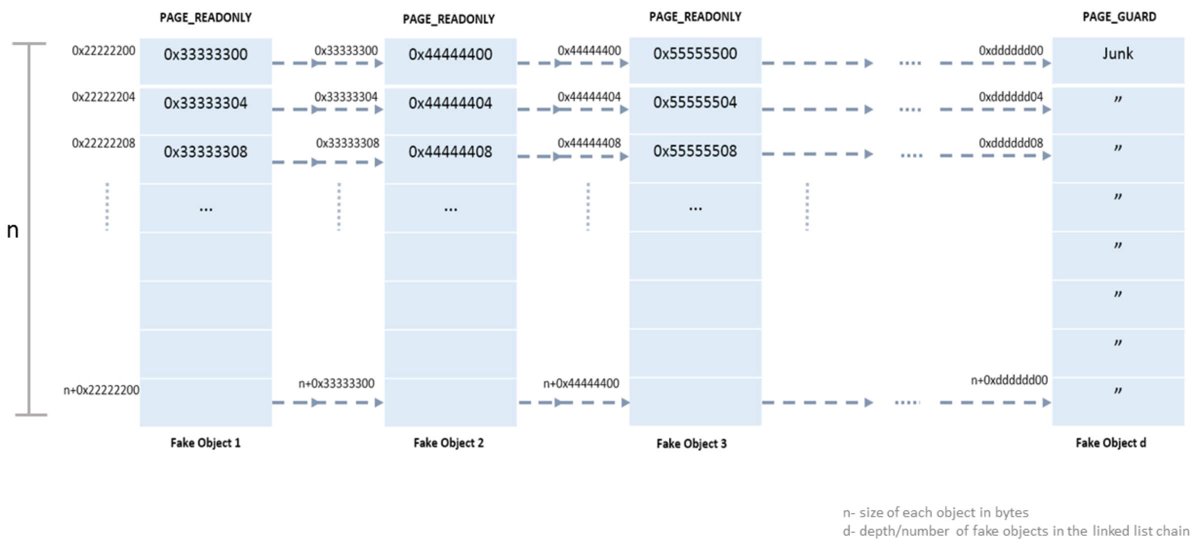
b) Let's call the memory block initialized in step a) as M1. Now allocate multiple memory blocks M2, M3,M4…Mn within the process such that dword at M1 contains the address of the first dword of M2 (basically first 4 bytes of M1 point to the address of the first 4 dword of M2) , second dword of M1 points 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

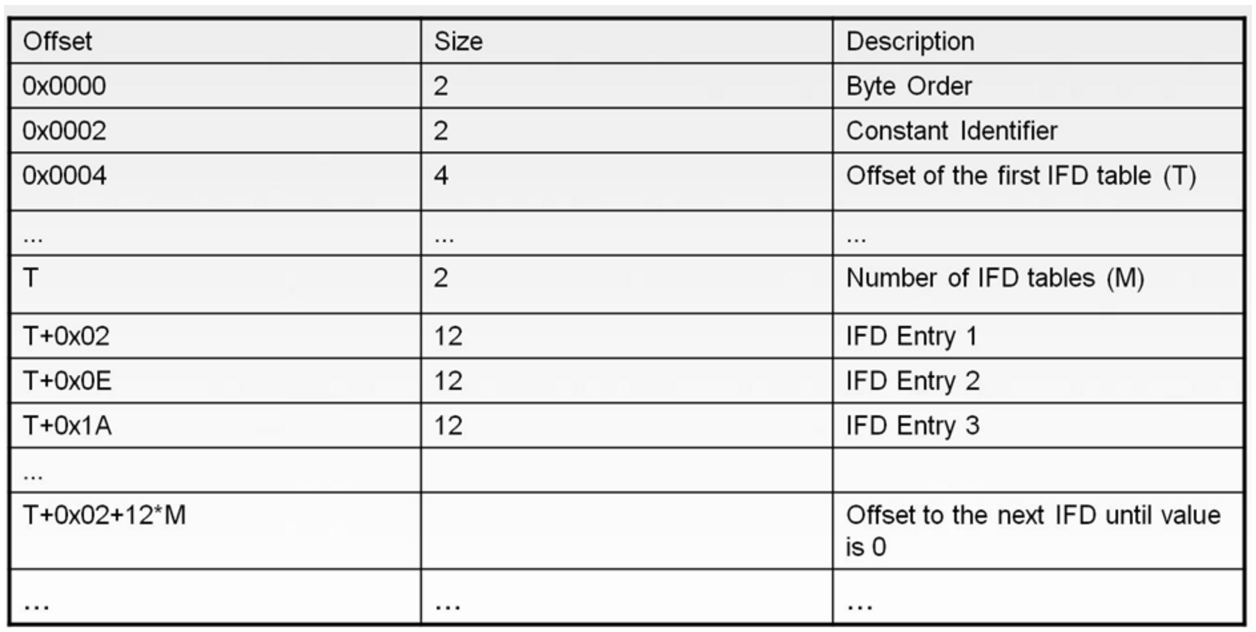
In this section we demonstrate the analysis of three different issues. We start from the simplest to the most complicated, trying to not only walking the reader through the process, but also clearly pointing out the limitations of the approach.

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.

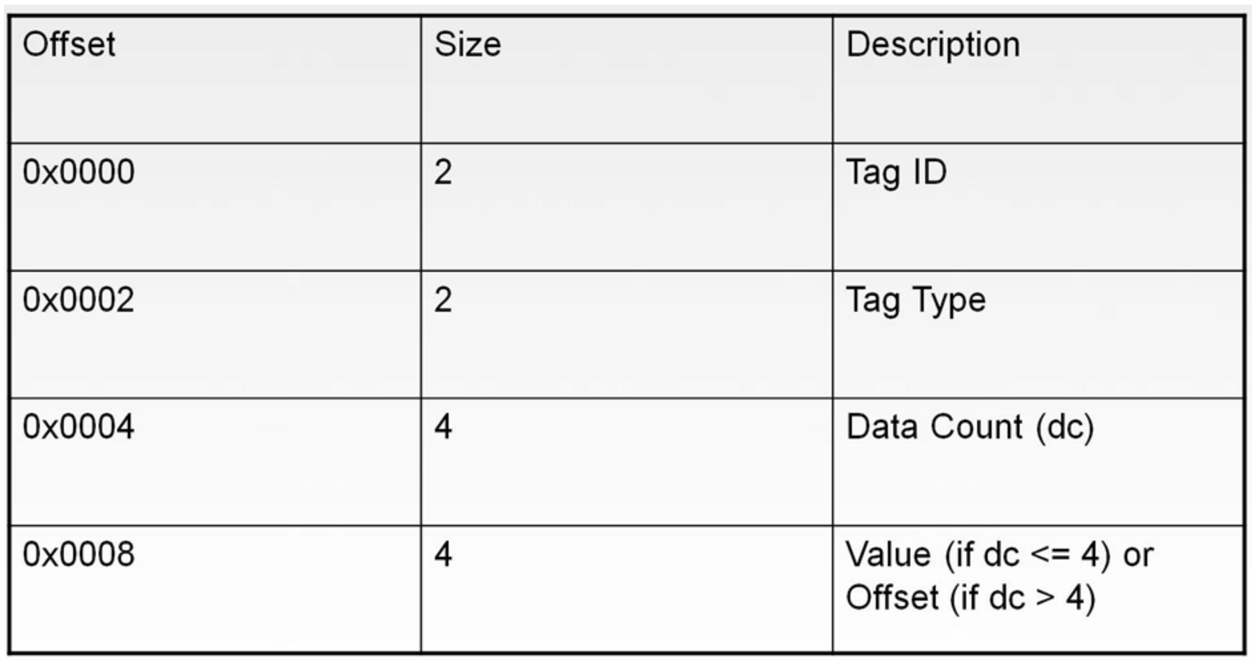
“Tag Image File Format (TIFF) is a file format used primarily for storing digital images, including photographs and line art. TIFF is a popular format for high color depth images, along with JPEG and PNG”. The vulnerability affected libtiff and years later it was found that Adobe was still using the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

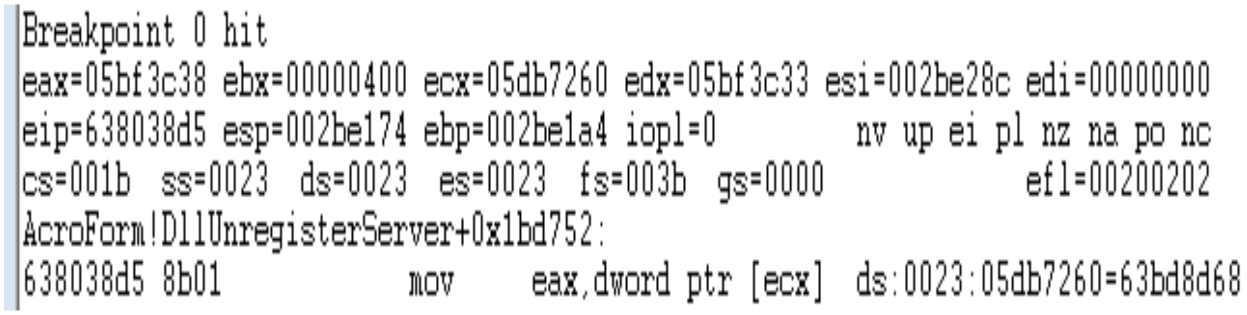
The problem here is with an embedded TIFF file that has TAG ID 0x0129 (known as PageNumber), TAG ID 0x0141 (known as HalftoneHints), TAG ID 0x0212 (known as YCbCrSubSampling) or TAG ID 0x0150 (known as DotRange). What happens is for those TAG IDs, if their Tag Type is defined as SHORT (3), the parser 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.

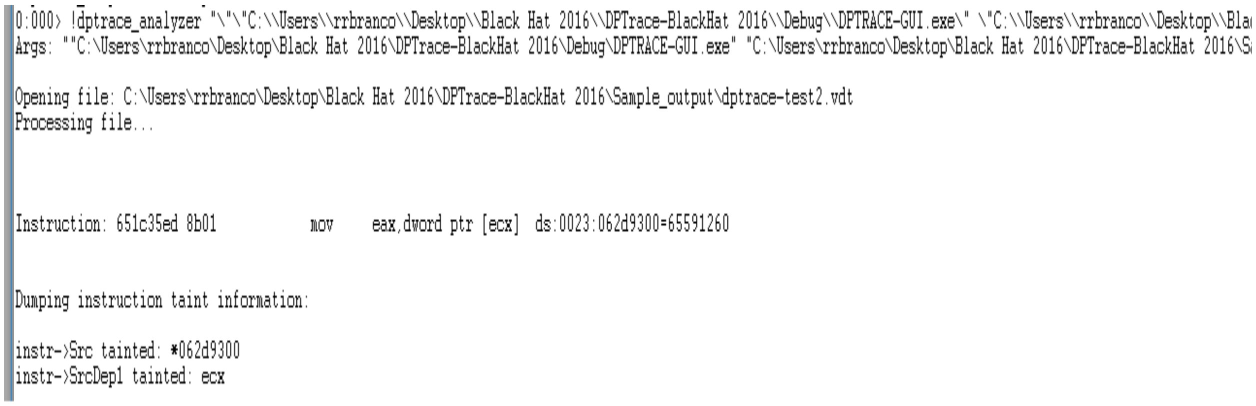
We did a bit of cheating in this one, since we had a working exploit and no crashes. The working exploit was crashing in some targets (OS versions), but with an obvious exploitable condition. We modified it to generate a violation in the stack copying (copying too much data to the point to overwrite 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:



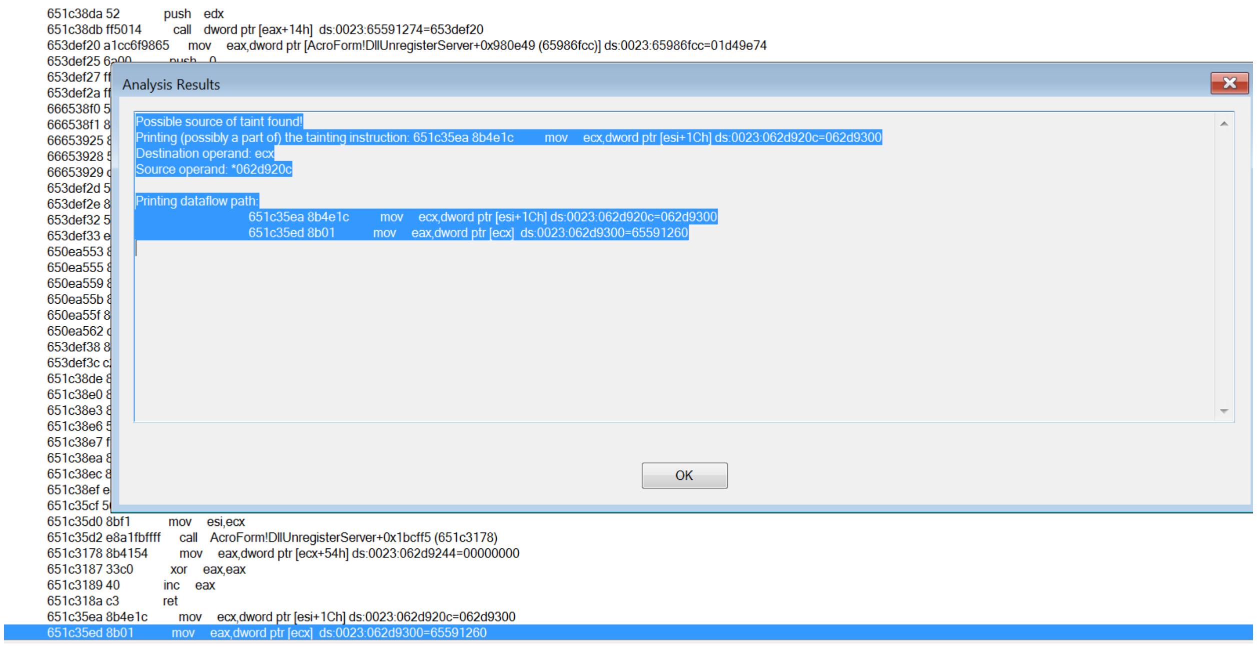
```
Breakpoint 0 hit
eax=05bf3c38 ebx=00000400 ecx=05db7260 edx=05bf3c33 esi=002be28c edi=00000000
eip=638038d5 esp=002be174 ebp=002be1a4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
AcroForm!DllUnregisterServer+0x1bd752:
638038d5 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\\Bla
Args: ""C:\Users\rrbranco\Desktop\Black Hat 2016\DPTrace-BlackHat 2016\Debug\DPTRACE-GUI.exe" "C:\Users\rrbranco\Desktop\Black Hat 2016\DPTrace-BlackHat 2016\S

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->SrcDep1 tainted: ecx
```

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

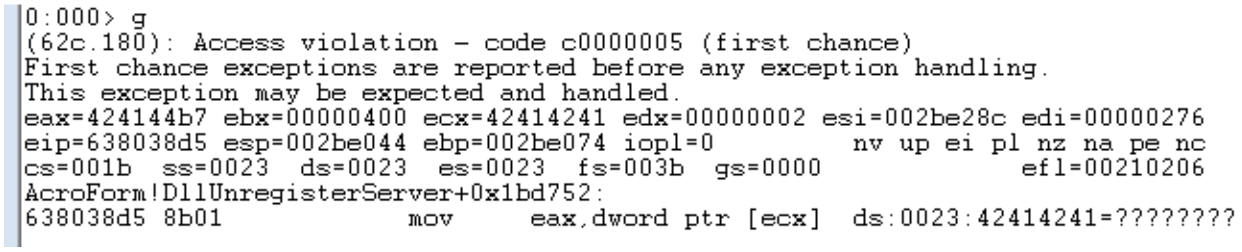


```
651c38da 52          push    edx
651c38db ff5014        call    dword ptr [eax+14h]  ds:0023:65591274=653def20
653def20 a1cc6f9865      mov     eax,dword ptr [AcroForm!DllUnregisterServer+0x980e49 (65986fcc)] ds:0023:65986fcc=01d49e74
653def25 6a00          push    0
653def27 ff
653def2a ff
666538f0 5
666538f1 8
66653925 8
66653928 5
66653929 c
653def2d 5
653def2e 8
653def32 5
653def33 e
650ea553 8
650ea555 8
650ea559 8
650ea55b 8
650ea55f 8
650ea562 c
653def38 8
653def3c c
651c38de 8
651c38e0 8
651c38e3 8
651c38e6 5
651c38e7 f
651c38ea 8
651c38ec 8
651c38ef e
651c35cf 5
651c35d0 8bf1          mov     esi,ecx
651c35d2 e8a1fbffff      call    AcroForm!DllUnregisterServer+0x1bcff5 (651c3178)
651c3178 8b4154        mov     eax,dword ptr [ecx+54h] ds:0023:062d9244=00000000
651c3187 33c0         xor     eax,eax
651c3189 40          inc     eax
651c318a c3          ret
651c35ea 8b4e1c        mov     ecx,dword ptr [esi+1Ch] ds:0023:062d920c=062d9300
651c35ed 8b01         mov     eax,dword ptr [ecx]  ds:0023:062d9300=65591260
```

Analysis Results

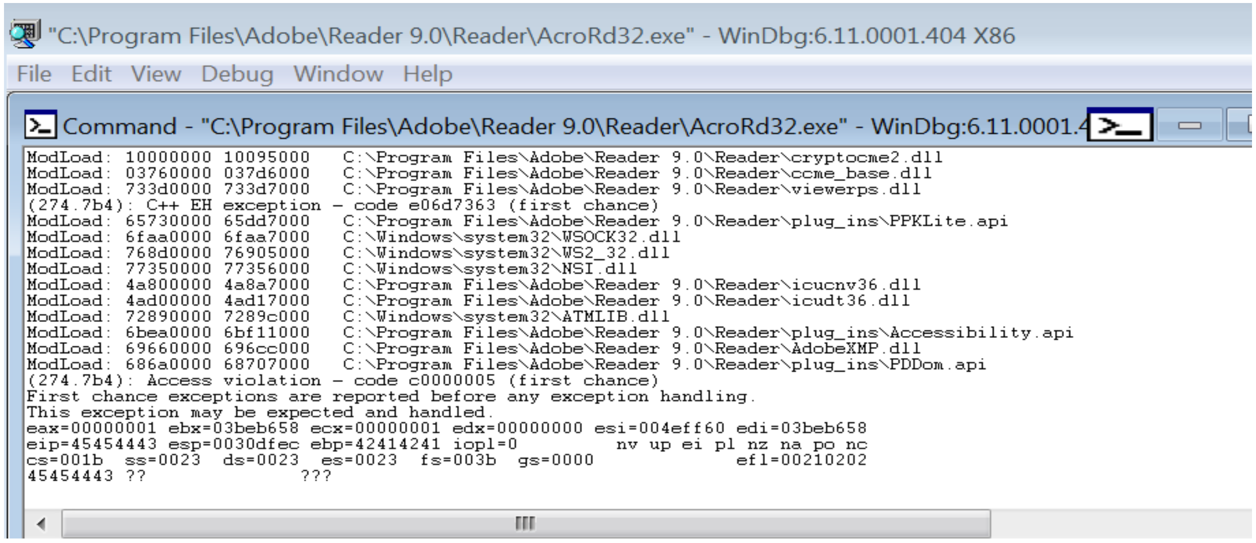
```
Possible source of taint found!
Printing (possibly a part of) the tainting instruction: 651c35ea 8b4e1c        mov     ecx,dword ptr [esi+1Ch] ds:0023:062d920c=062d9300
Destination operand: ecx
Source operand: *062d920c

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:


"C:\Program Files\Adobe\Reader 9.0\Reader\AcroRd32.exe" - WinDbg:6.11.0001.404 X86
File Edit View Debug Window Help
Command - "C:\Program Files\Adobe\Reader 9.0\Reader\AcroRd32.exe" - WinDbg:6.11.0001.4

```
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 ??              ???
```

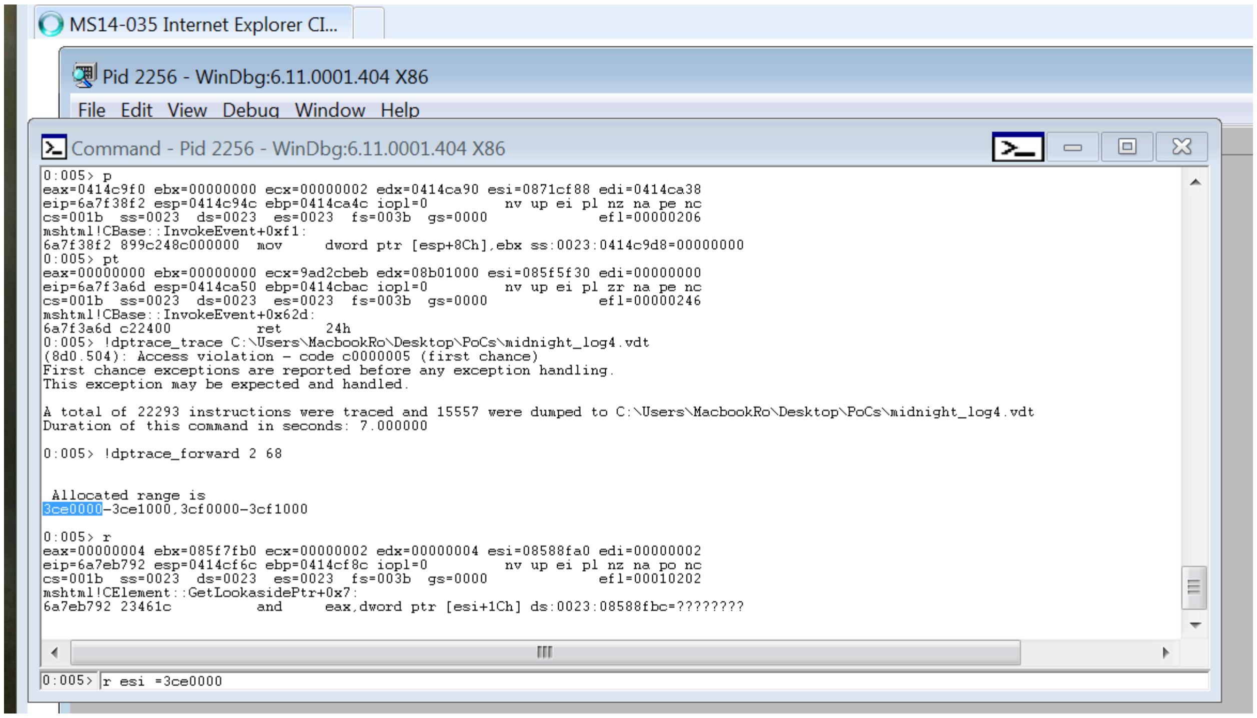
Because this case is a simple, classical stack-based buffer overflow and our crash was so near the exploitable condition, the analysis is quite simplified and did not require real forward analysis. We included it here to be very clear on the limitations (even for a simple stack-based buffer overflow 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☹)

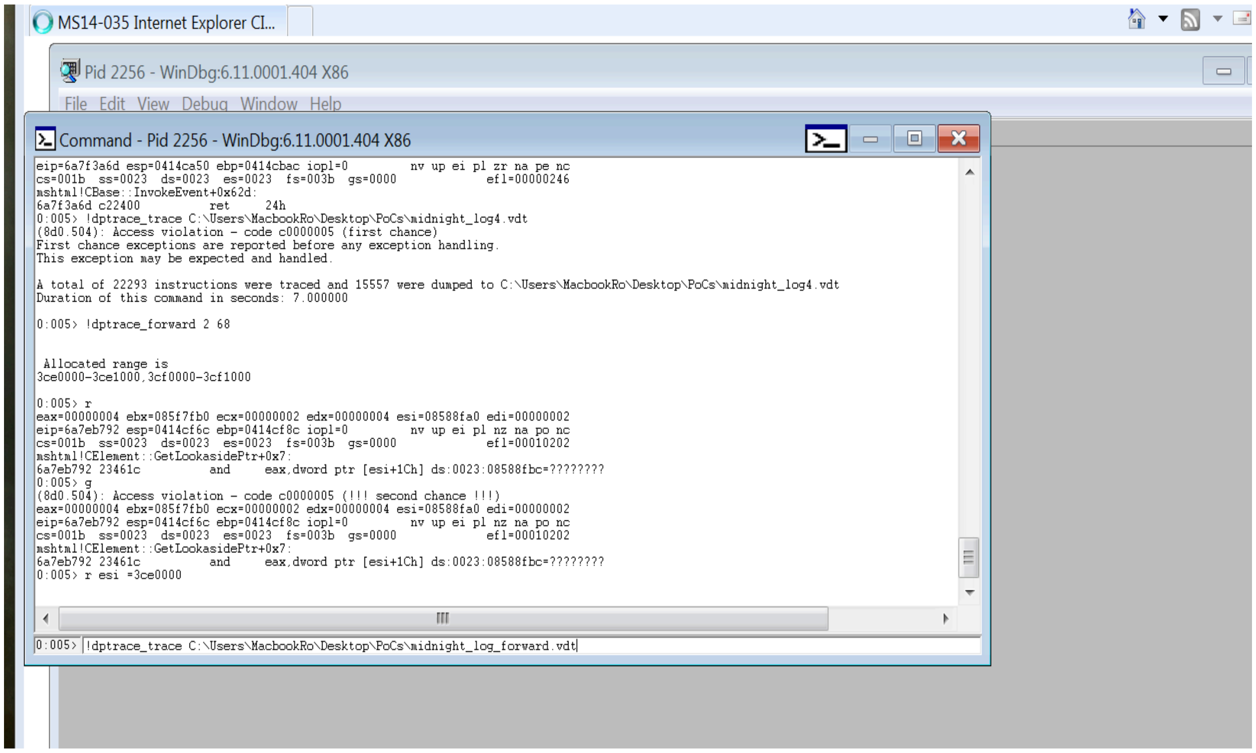
* http://www.cybersphinx.com/exploiting-cve-2014-0282-ms-035-cinput-use-after-free-vulnerability/
* https://www.nccgroup.trust/globalassets/our-research/uk/whitepapers/2015/12/cve-2014-0282pdf/

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


MS14-035 Internet Explorer Cl...
Pid 2256 - WinDbg:6.11.0001.404 X86
File Edit View Debug Window Help
Command - Pid 2256 - WinDbg:6.11.0001.404 X86

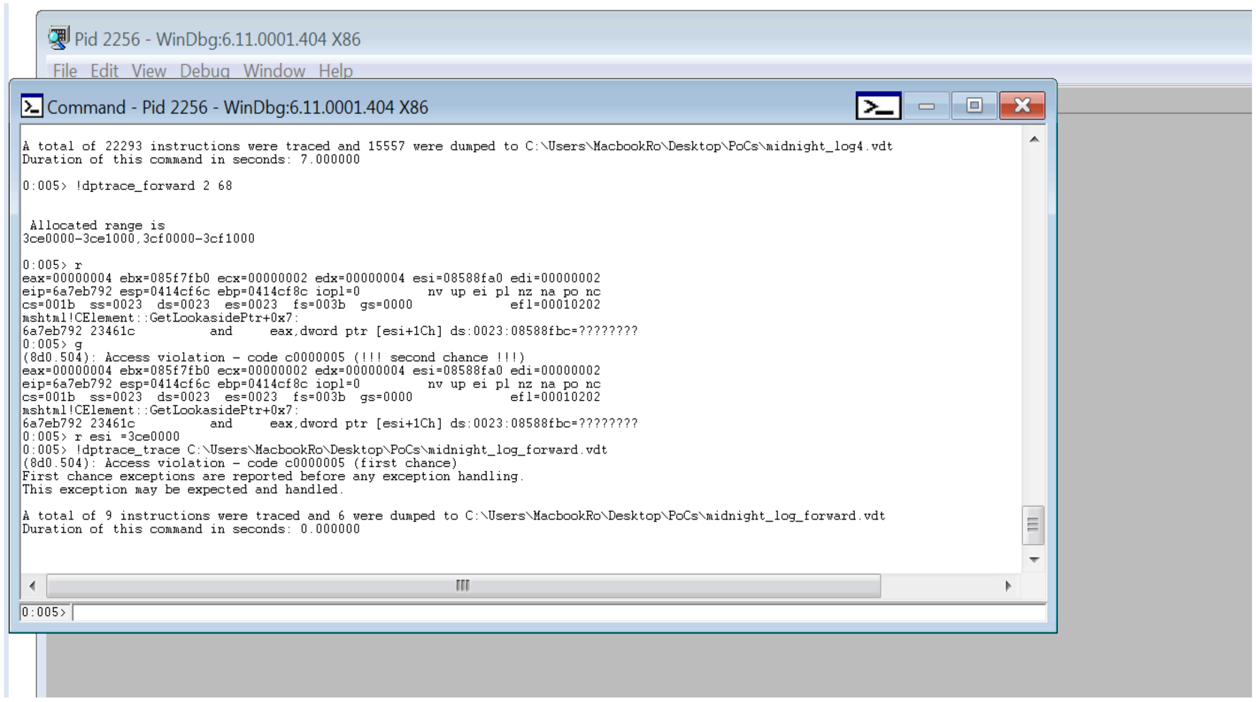
```
0:005> p
eax=0414c9f0 ebx=00000000 ecx=00000002 edx=0414ca90 esi=0871cf88 edi=0414ca38
eip=6a7f38f2 esp=0414c94c ebp=0414ca4c iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
mshtml!CBase::InvokeEvent+0xf1:
6a7f38f2 899c248c000000  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).


MS14-035 Internet Explorer Cl...
Pid 2256 - WinDbg:6.11.0001.404 X86
File Edit View Debug Window Help
Command - Pid 2256 - WinDbg:6.11.0001.404 X86

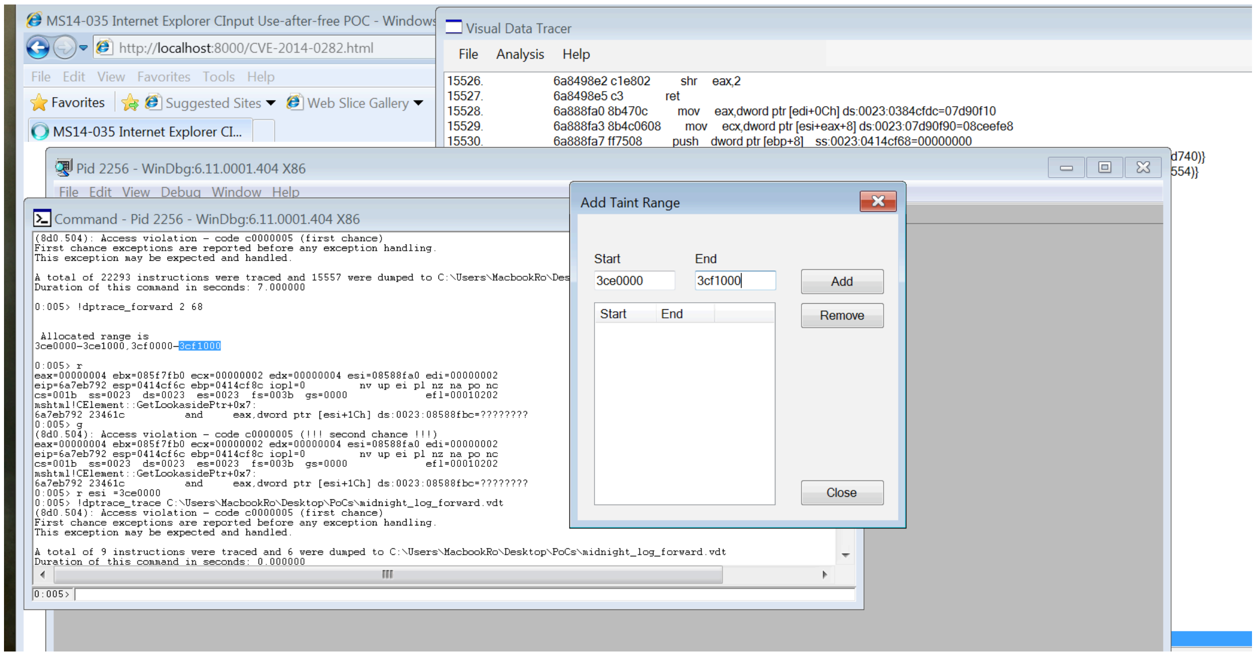
```
eip=6a7f3a6d esp=0414ca50 ebp=0414cbac iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CBase::InvokeEvent+0x62d:
6a7f3a6d c22400          ret     24h
0:005> !dptrace_trace C:\Users\MacbookRo\Desktop\PoCs\midnight_log4.vdt
(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> g
(8d0.504): Access violation - code c0000005 (!!! second chance !!!)
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

0:005> !dptrace_trace C:\Users\MacbookRo\Desktop\PoCs\midnight_log_forward.vdt
```

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.


Pid 2256 - WinDbg:6.11.0001.404 X86
File Edit View Debug Window Help
Command - Pid 2256 - WinDbg:6.11.0001.404 X86

```
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> g
(8d0.504): Access violation - code c0000005 (!!! second chance !!!)
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
0:005> !dptrace_trace C:\Users\MacbookRo\Desktop\PoCs\midnight_log_forward.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 9 instructions were traced and 6 were dumped to C:\Users\MacbookRo\Desktop\PoCs\midnight_log_forward.vdt
Duration of this command in seconds: 0.000000

0:005>
```

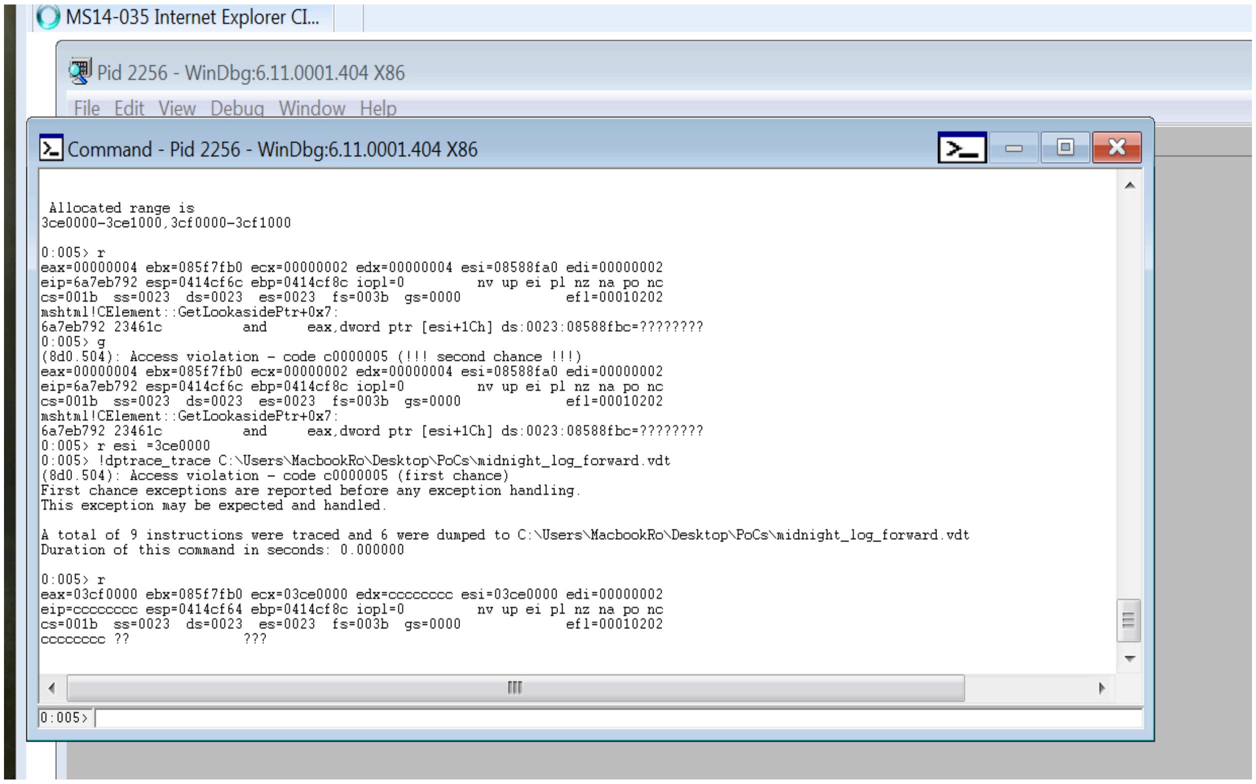
We let the execution continue, and trace until the subsequent breakpoint/access violation. To be sure that the freed object is indeed controlled, we can use the analyzer to see the tainted value. We first define the range of interest:


```
15526.          6a8498e2 c1e802       shr    eax,2
15527.          6a8498e5 c3          ret
15528.          6a888fa0 8b470c       mov    eax,dword ptr [edi+0Ch] ds:0023:0384cfdc=07d90f10
15529.          6a888fa3 8b4c0608      mov    ecx,dword ptr [esi+eax+8] ds:0023:07d90f90=08ceefe8
15530.          6a888fa7 ff7508       push   dword ptr [ebp+8]    ss:0023:0414cf68=00000000
```

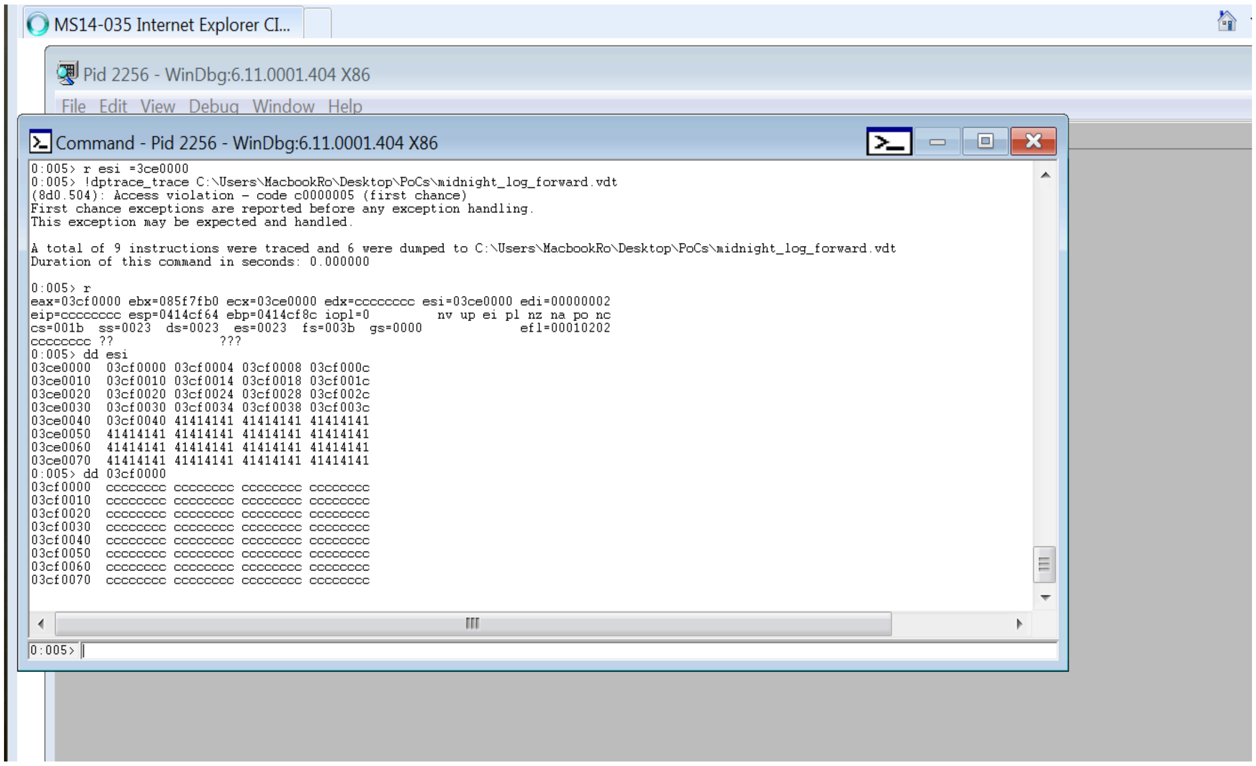
Add Taint Range — Start: 3ce0000 End: 3cf1000

```
(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\Des
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> g
(8d0.504): Access violation - code c0000005 (!!! second chance !!!)
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
0:005> !dptrace_trace C:\Users\MacbookRo\Desktop\PoCs\midnight_log_forward.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 9 instructions were traced and 6 were dumped to C:\Users\MacbookRo\Desktop\PoCs\midnight_log_forward.vdt
Duration of this command in seconds: 0.000000
0:005>
```

We let the execution go in the debugger:

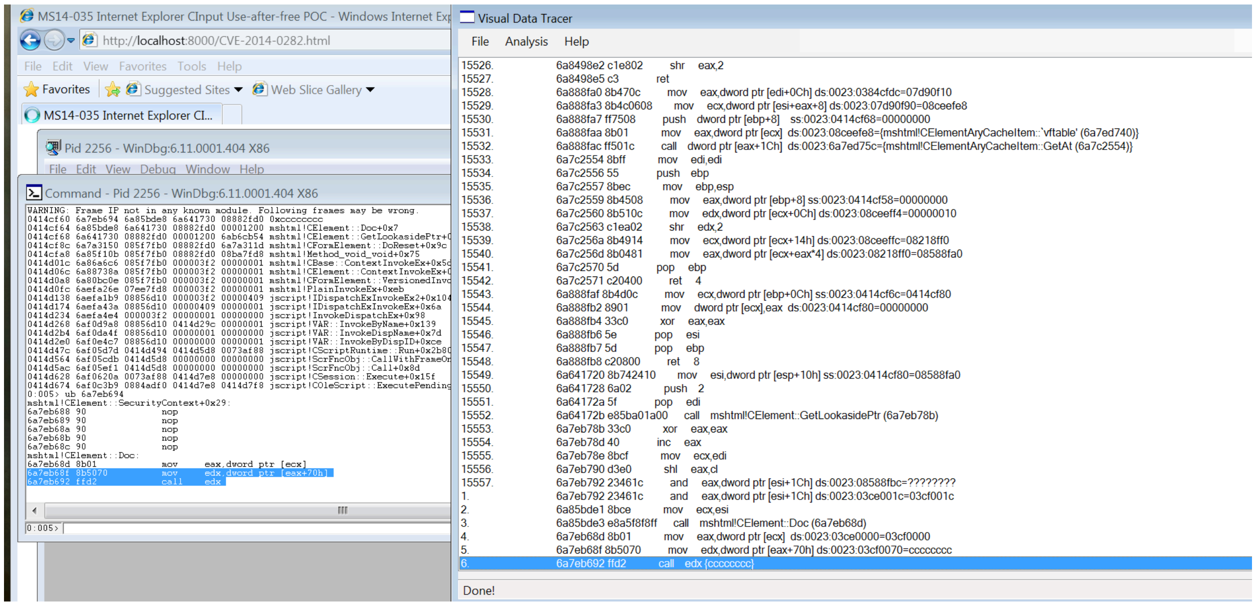

```

 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> g
(8d0.504): Access violation - code c0000005 (!!! second chance !!!)
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
0:005> !dptrace_trace C:\Users\MacbookRo\Desktop\PoCs\midnight_log_forward.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 9 instructions were traced and 6 were dumped to C:\Users\MacbookRo\Desktop\PoCs\midnight_log_forward.vdt
Duration of this command in seconds: 0.000000

0:005> r
eax=03cf0000 ebx=085f7fb0 ecx=03ce0000 edx=cccccccc esi=03ce0000 edi=00000002
eip=cccccccc esp=0414cf64 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
cccccccc ??              ???
0:005>
```

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



```
0:005> r esi =3ce0000
0:005> !dptrace_trace C:\Users\MacbookRo\Desktop\PoCs\midnight_log_forward.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 9 instructions were traced and 6 were dumped to C:\Users\MacbookRo\Desktop\PoCs\midnight_log_forward.vdt
Duration of this command in seconds: 0.000000

0:005> r
eax=03cf0000 ebx=085f7fb0 ecx=03ce0000 edx=cccccccc esi=03ce0000 edi=00000002
eip=cccccccc esp=0414cf64 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
cccccccc ??              ???
0:005> dd esi
03ce0000  03cf0000 03cf0004 03cf0008 03cf000c
03ce0010  03cf0010 03cf0014 03cf0018 03cf001c
03ce0020  03cf0020 03cf0024 03cf0028 03cf002c
03ce0030  03cf0030 03cf0034 03cf0038 03cf003c
03ce0040  03cf0040 41414141 41414141 41414141
03ce0050  41414141 41414141 41414141 41414141
03ce0060  41414141 41414141 41414141 41414141
03ce0070  41414141 41414141 41414141 41414141
0:005> dd 03cf0000
03cf0000  cccccccc cccccccc cccccccc cccccccc
03cf0010  cccccccc cccccccc cccccccc cccccccc
03cf0020  cccccccc cccccccc cccccccc cccccccc
03cf0030  cccccccc cccccccc cccccccc cccccccc
03cf0040  cccccccc cccccccc cccccccc cccccccc
03cf0050  cccccccc cccccccc cccccccc cccccccc
03cf0060  cccccccc cccccccc cccccccc cccccccc
03cf0070  cccccccc cccccccc cccccccc cccccccc
0:005>
```

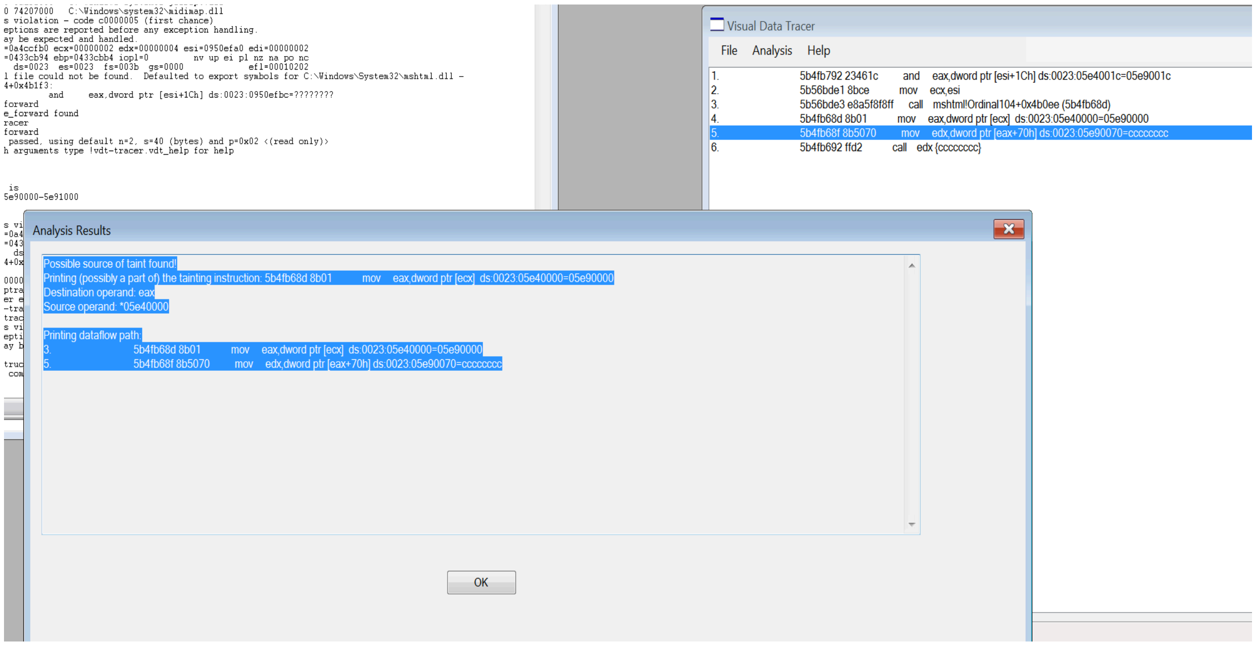
Program control seems evident from the forward analysis view (EIP comes from our faked structure), so we need to be sure we control the initial violation:



```
WARNING: Frame IP not in any known module. Following frames may be wrong.
0414cf60 6a7eb694 6a85bde8 6a641730 08882fd0 0xcccccccc
0414cf64 6a85bde8 6a641730 08882fd0 00001200 mshtml!CElement::Doc+0x7
0414cf68 6a641730 08882fd0 00001200 6ab6cb54 mshtml!CElement::GetLookasidePtr+0
0414cf8c 6a7a3150 085f7fb0 08882fd0 6a7a311d mshtml!CFormElement::DoReset+0x9c
0414cfa8 6a85f10b 085f7fb0 08882fd0 08ba7fd8 mshtml!Method_void_void+0x75
0414d01c 6a86a6c6 085f7fb0 000003f2 00000001 mshtml!CBase::ContextInvokeEx+0x5d
0414d06c 6a88738a 085f7fb0 000003f2 00000001 mshtml!CElement::ContextInvokeEx+0
0414d0a8 6a80bc0e 085f7fb0 000003f2 00000001 mshtml!CFormElement::VersionedInvo
0414d0fc 6aefa26e 07ee7fd8 000003f2 00000001 mshtml!PlainInvokeEx+0xeb
0414d138 6aefa1b9 08856d10 000003f2 00000409 jscript!IDispatchExInvokeEx2+0x104
0414d174 6aefa43a 08856d10 00000409 00000001 jscript!IDispatchExInvokeEx+0x6a
0414d234 6aefa4e4 000003f2 00000001 00000000 jscript!InvokeDispatchEx+0x98
0414d268 6af0d9a8 08856d10 0414d29c 00000001 jscript!VAR::InvokeByName+0x139
0414d2b4 6af0da4f 08856d10 00000001 00000000 jscript!VAR::InvokeDispName+0x7d
0414d2e0 6af0e4c7 08856d10 00000000 00000001 jscript!VAR::InvokeByDispID+0xce
0414d47c 6af05d7d 0414d494 0414d5d8 0073af88 jscript!CScriptRuntime::Run+0x2b80
0414d564 6af05cdb 0414d5d8 00000000 00000000 jscript!ScrFncObj::CallWithFrameOn
0414d5ac 6af05ef1 0414d5d8 00000000 00000000 jscript!ScrFncObj::Call+0x8d
0414d628 6af0620a 0073af88 0414d7e8 00000000 jscript!CSession::Execute+0x15f
0414d674 6af0c3b9 0884adf0 0414d7e8 0414d7f8 jscript!COleScript::ExecutePending
0:005> ub 6a7eb694
mshtml!CElement::SecurityContext+0x29:
6a7eb688 90              nop
6a7eb689 90              nop
6a7eb68a 90              nop
6a7eb68b 90              nop
6a7eb68c 90              nop
mshtml!CElement::Doc:
6a7eb68d 8b01            mov     eax,dword ptr [ecx]
6a7eb68f 8b5070          mov     edx,dword ptr [eax+70h]
6a7eb692 ffd2            call    edx
0:005>
```

```
15526.   6a8498e2 c1e802     shr   eax,2
15527.   6a8498e5 c3         ret
15528.   6a888fa0 8b470c     mov   eax,dword ptr [edi+0Ch] ds:0023:0384cfdc=07d90f10
15529.   6a888fa3 8b4c0608   mov   ecx,dword ptr [esi+eax+8] ds:0023:07d90f90=08ceefe8
15530.   6a888fa7 ff7508     push  dword ptr [ebp+8]  ss:0023:0414cf68=00000000
15531.   6a888faa 8b01       mov   eax,dword ptr [ecx]  ds:0023:08ceefe8={mshtml!CElementAryCacheItem::`vftable' (6a7ed740)}
15532.   6a888fac ff501c     call  dword ptr [eax+1Ch]  ds:0023:6a7ed75c={mshtml!CElementAryCacheItem::GetAt (6a7c2554)}
15533.   6a7c2554 8bff       mov   edi,edi
15534.   6a7c2556 55         push  ebp
15535.   6a7c2557 8bec       mov   ebp,esp
15536.   6a7c2559 8b4508     mov   eax,dword ptr [ebp+8] ss:0023:0414cf58=00000000
15537.   6a7c2560 8b510c     mov   edx,dword ptr [ecx+0Ch] ds:0023:08ceeff4=00000010
15538.   6a7c2563 c1ea02     shr   edx,2
15539.   6a7c256a 8b4914     mov   ecx,dword ptr [ecx+14h] ds:0023:08ceeffc=08218ff0
15540.   6a7c256d 8b0481     mov   eax,dword ptr [ecx+eax*4] ds:0023:08218ff0=08588fa0
15541.   6a7c2570 5d         pop   ebp
15542.   6a7c2571 c20400     ret   4
15543.   6a888faf 8b4d0c     mov   ecx,dword ptr [ebp+0Ch] ss:0023:0414cf6c=0414cf80
15544.   6a888fb2 8901       mov   dword ptr [ecx],eax  ds:0023:0414cf80=00000000
15545.   6a888fb4 33c0       xor   eax,eax
15546.   6a888fb6 5e         pop   esi
15547.   6a888fb7 5d         pop   ebp
15548.   6a888fb8 c20800     ret   8
15549.   6a641720 8b742410   mov   esi,dword ptr [esp+10h] ss:0023:0414cf80=08588fa0
15550.   6a641728 6a02       push  2
15551.   6a64172a 5f         pop   edi
15552.   6a64172b e85ba01a00 call  mshtml!CElement::GetLookasidePtr (6a7eb78b)
15553.   6a7eb78b 33c0       xor   eax,eax
15554.   6a7eb78d 40         inc   eax
15555.   6a7eb78e 8bcf       mov   ecx,edi
15556.   6a7eb790 d3e0       shl   eax,cl
15557.   6a7eb792 23461c     and   eax,dword ptr [esi+1Ch] ds:0023:08588fbc=????????
1.       6a7eb792 23461c     and   eax,dword ptr [esi+1Ch] ds:0023:03ce001c=03cf001c
2.       6a85bde1 8bce       mov   ecx,esi
3.       6a85bde3 e8a5f8f8ff call  mshtml!CElement::Doc (6a7eb68d)
4.       6a7eb68d 8b01       mov   eax,dword ptr [ecx]  ds:0023:03ce0000=03cf0000
5.       6a7eb68f 8b5070     mov   edx,dword ptr [eax+70h] ds:0023:03cf0070=cccccccc
6.       6a7eb692 ffd2       call  edx {cccccccc}
Done!
```

And finally if we look the first crashing point:

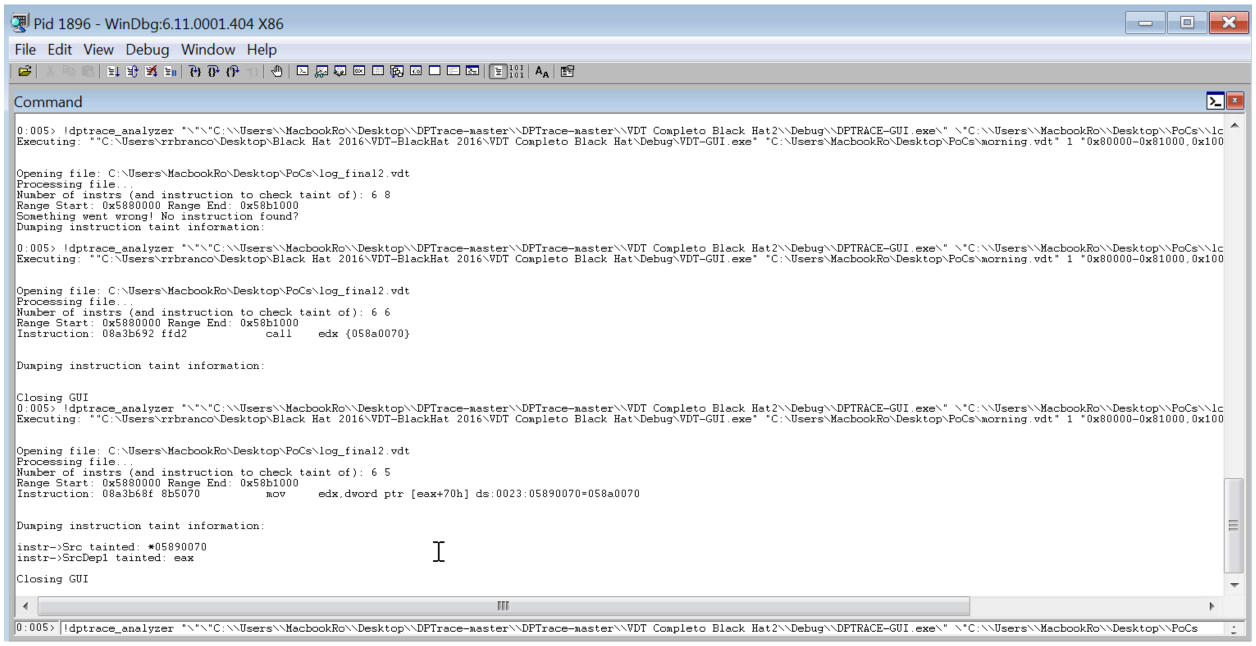


```
0 74207000   C:\Windows\system32\midimap.dll
s violation - code c0000005 (first chance)
eptions are reported before any exception handling.
ay be expected and handled.
=0a4ccfb0 ecx=00000002 edx=00000004 esi=0950efa0 edi=00000002
=0433cb94 ebp=0433cbb4 iopl=0         nv up ei pl nz na po nc
  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
l file could not be found.  Defaulted to export symbols for C:\Windows\System32\mshtml.dll -
4+0x4b1f3:
         and     eax,dword ptr [esi+1Ch] ds:0023:0950efbc=????????
forward
e_forward found
racer
forward
 passed, using default n=2, s=40 (bytes) and p=0x02 <(read only)>
h arguments type !vdt-tracer.vdt_help for help


 is
5e90000-5e91000
```

```
1.    5b4fb792 23461c     and    eax,dword ptr [esi+1Ch] ds:0023:05e4001c=05e9001c
2.    5b56bde1 8bce       mov    ecx,esi
3.    5b56bde3 e8a5f8f8ff   call   mshtml!Ordinal104+0x4b0ee (5b4fb68d)
4.    5b4fb68d 8b01       mov    eax,dword ptr [ecx]  ds:0023:05e40000=05e90000
5.    5b4fb68f 8b5070     mov    edx,dword ptr [eax+70h] ds:0023:05e90070=cccccccc
6.    5b4fb692 ffd2       call   edx {cccccccc}
```

```
Possible source of taint found!
Printing (possibly a part of) the tainting instruction: 5b4fb68d 8b01       mov    eax,dword ptr [ecx]  ds:0023:05e40000=05e90000
Destination operand: eax
Source operand: *05e40000

Printing dataflow path:
3.          5b4fb68d 8b01       mov    eax,dword ptr [ecx]  ds:0023:05e40000=05e90000
5.          5b4fb68f 8b5070     mov    edx,dword ptr [eax+70h] ds:0023:05e90070=cccccccc
```

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):



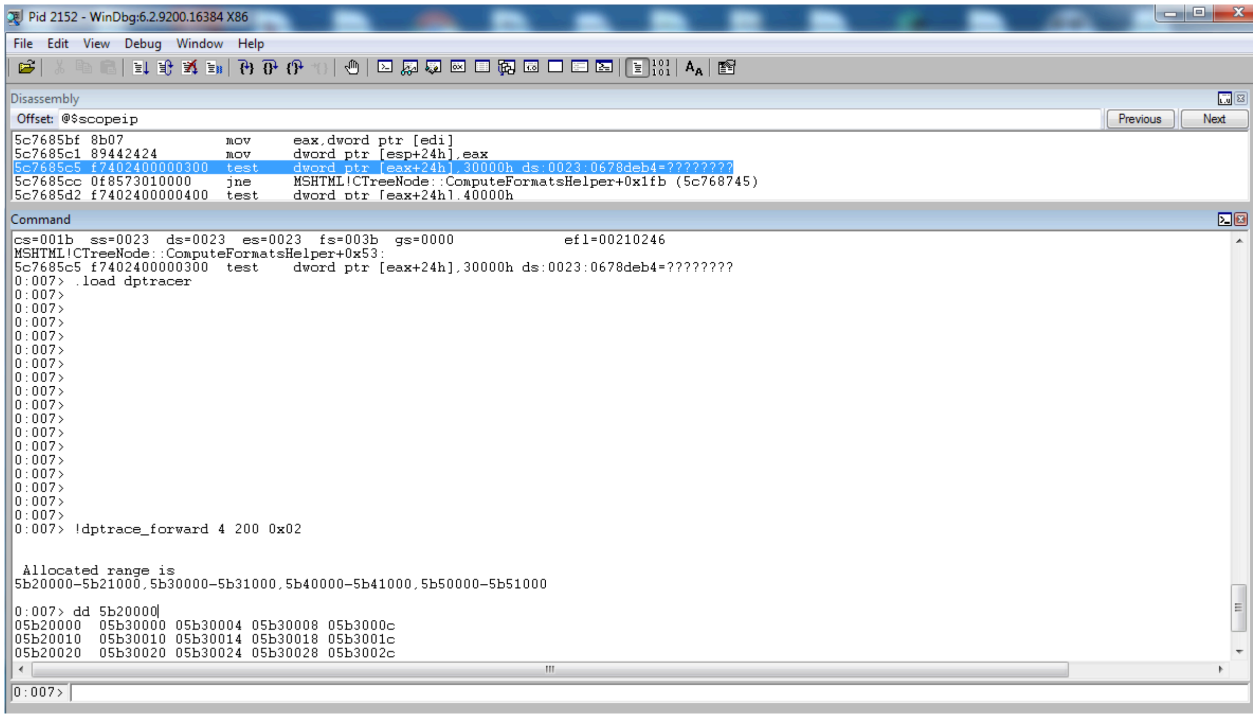
Pid 1896 - WinDbg:6.11.0001.404 X86
File Edit View Debug Window Help
Command

```
0:005> !dptrace_analyzer "\"\"C:\\Users\\MacbookRo\\Desktop\\DPTrace-master\\DPTrace-master\\VDT Completo Black Hat2\\Debug\\DPTRACE-GUI.exe\" \"C:\\Users\\MacbookRo\\Desktop\\PoCs\\lc
Executing: ""C:\Users\rrbranco\Desktop\Black Hat 2016\VDT-BlackHat 2016\VDT Completo Black Hat\Debug\VDT-GUI.exe" "C:\Users\MacbookRo\Desktop\PoCs\morning.vdt" 1 "0x80000-0x81000,0x100


Opening file: C:\Users\MacbookRo\Desktop\PoCs\log_final2.vdt
Processing file...
Number of instrs (and instruction to check taint of): 6 8
Range Start: 0x5880000 Range End: 0x58b1000
Something went wrong! No instruction found?
Dumping instruction taint information:

0:005> !dptrace_analyzer "\"\"C:\\Users\\MacbookRo\\Desktop\\DPTrace-master\\DPTrace-master\\VDT Completo Black Hat2\\Debug\\DPTRACE-GUI.exe\" \"C:\\Users\\MacbookRo\\Desktop\\PoCs\\lc
Executing: ""C:\Users\rrbranco\Desktop\Black Hat 2016\VDT-BlackHat 2016\VDT Completo Black Hat\Debug\VDT-GUI.exe" "C:\Users\MacbookRo\Desktop\PoCs\morning.vdt" 1 "0x80000-0x81000,0x100


Opening file: C:\Users\MacbookRo\Desktop\PoCs\log_final2.vdt
Processing file...
Number of instrs (and instruction to check taint of): 6 6
Range Start: 0x5880000 Range End: 0x58b1000
Instruction: 08a3b692 ffd2            call    edx {058a0070}


Dumping instruction taint information:


Closing GUI
0:005> !dptrace_analyzer "\"\"C:\\Users\\MacbookRo\\Desktop\\DPTrace-master\\DPTrace-master\\VDT Completo Black Hat2\\Debug\\DPTRACE-GUI.exe\" \"C:\\Users\\MacbookRo\\Desktop\\PoCs\\lc
Executing: ""C:\Users\rrbranco\Desktop\Black Hat 2016\VDT-BlackHat 2016\VDT Completo Black Hat\Debug\VDT-GUI.exe" "C:\Users\MacbookRo\Desktop\PoCs\morning.vdt" 1 "0x80000-0x81000,0x100


Opening file: C:\Users\MacbookRo\Desktop\PoCs\log_final2.vdt
Processing file...
Number of instrs (and instruction to check taint of): 6 5
Range Start: 0x5880000 Range End: 0x58b1000
Instruction: 08a3b68f 8b5070          mov     edx,dword ptr [eax+70h] ds:0023:05890070=058a0070


Dumping instruction taint information:

instr->Src tainted: *05890070
instr->SrcDep1 tainted: eax

Closing GUI

0:005> !dptrace_analyzer "\"\"C:\\Users\\MacbookRo\\Desktop\\DPTrace-master\\DPTrace-master\\VDT Completo Black Hat2\\Debug\\DPTRACE-GUI.exe\" \"C:\\Users\\MacbookRo\\Desktop\\PoCs
```

CVE-2015-6152 IE 11 CObjectElement Use-After-Free

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.
* https://www.exploit-db.com/exploits/38972/

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).

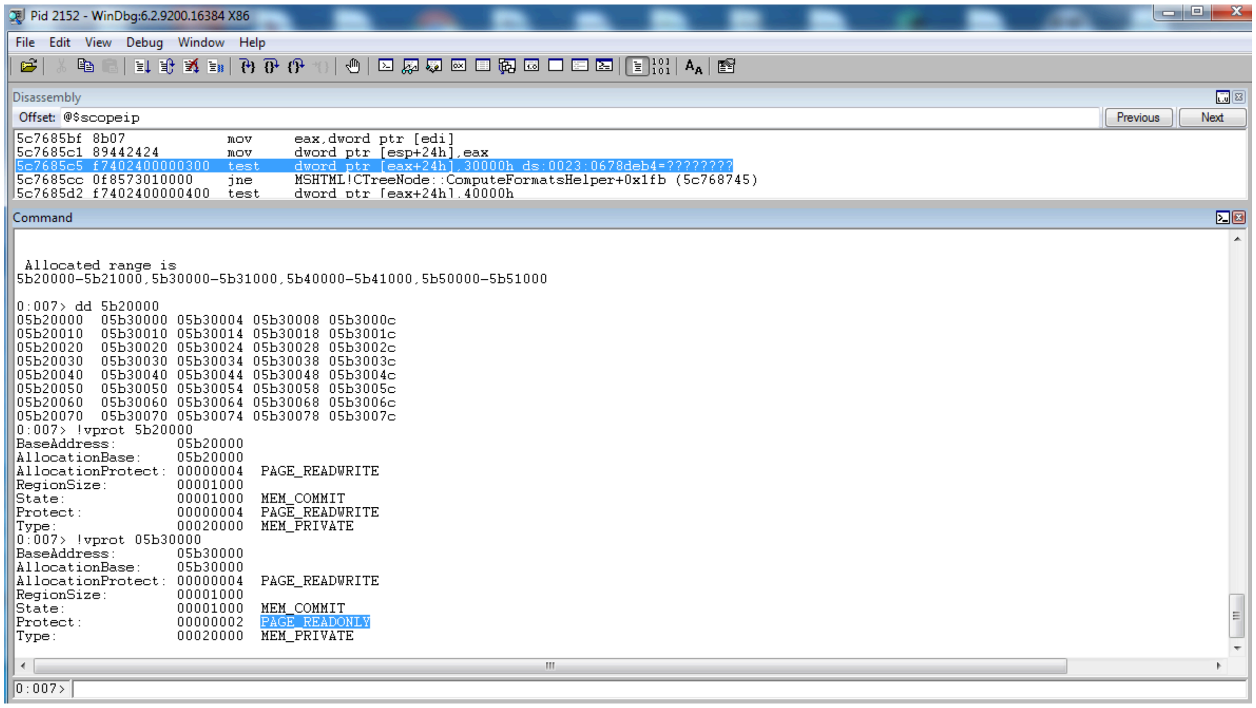

Pid 2152 - WinDbg:6.2.9200.16384 X86
File Edit View Debug Window Help
Disassembly
Offset: @$scopeip
Previous
Next
Command

```
5c7685bf 8b07            mov     eax,dword ptr [edi]
5c7685c1 89442424        mov     dword ptr [esp+24h],eax
5c7685c5 f7402400000300  test    dword ptr [eax+24h],30000h ds:0023:0678deb4=????????
5c7685cc 0f8573010000    jne     MSHTML!CTreeNode::ComputeFormatsHelper+0x1fb (5c768745)
5c7685d2 f7402400000400  test    dword ptr [eax+24h],40000h
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210246
MSHTML!CTreeNode::ComputeFormatsHelper+0x53:
5c7685c5 f7402400000300  test    dword ptr [eax+24h],30000h ds:0023:0678deb4=????????
0:007> .load dptracer
0:007>
0:007>
0:007>
0:007>
0:007>
0:007>
0:007>
0:007>
0:007>
0:007>
0:007>
0:007>
0:007>
0:007>
0:007>
0:007>
0:007>
0:007>
0:007> !dptrace_forward 4 200 0x02


 Allocated range is
5b20000-5b21000,5b30000-5b31000,5b40000-5b41000,5b50000-5b51000

0:007> dd 5b20000
05b20000  05b30000 05b30004 05b30008 05b3000c
05b20010  05b30010 05b30014 05b30018 05b3001c
05b20020  05b30020 05b30024 05b30028 05b3002c
```

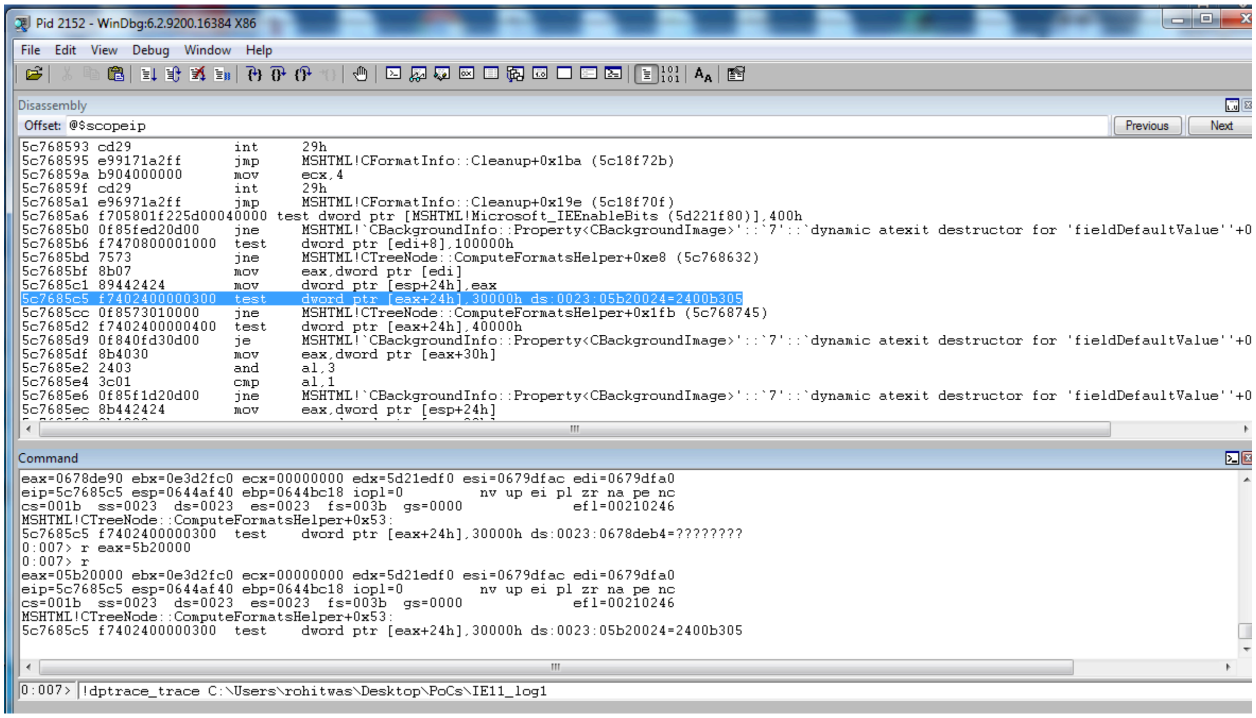
```
0:007>
```

The above figure illustrates the forward analysis preparation (!dptrace_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).



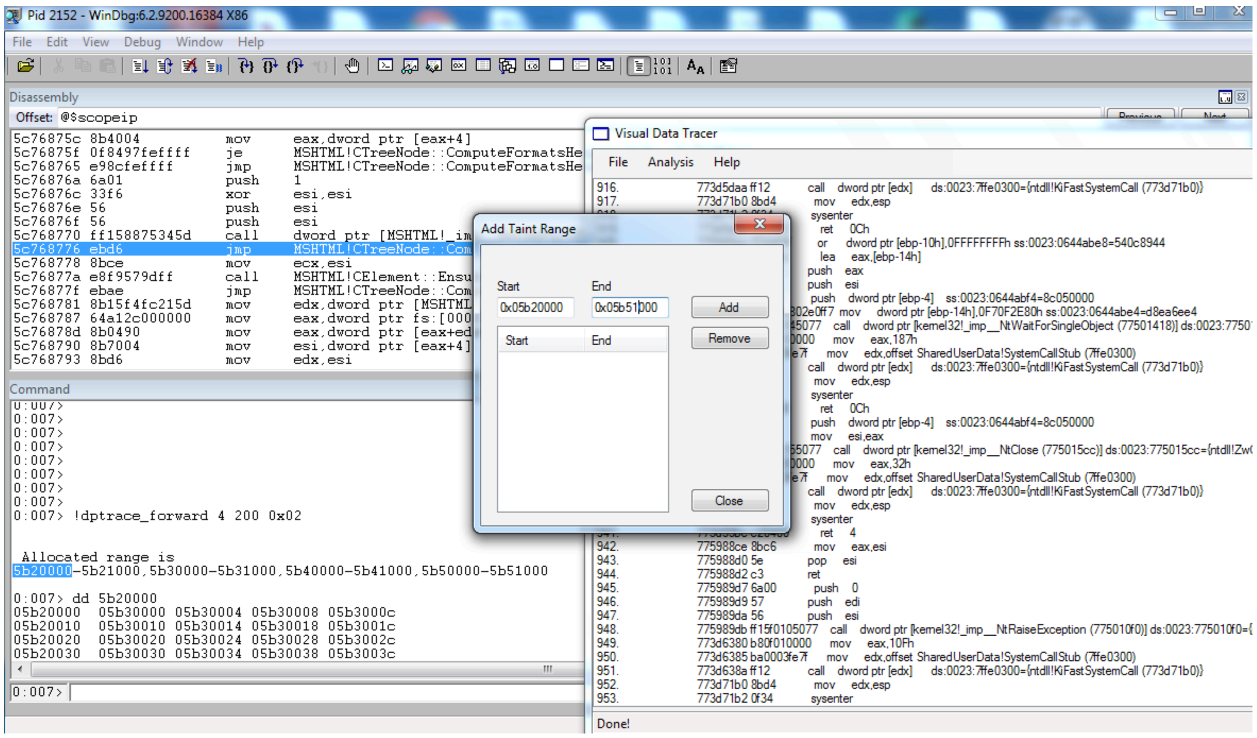
```
Pid 2152 - WinDbg:6.2.9200.16384 X86

Disassembly
Offset: @$scopeip
5c7685bf 8b07             mov     eax,dword ptr [edi]
5c7685c1 89442424         mov     dword ptr [esp+24h],eax
5c7685c5 f7402400000300   test    dword ptr [eax+24h],30000h ds:0023:0678deb4=????????
5c7685cc 0f8573010000     jne     MSHTML!CTreeNode::ComputeFormatsHelper+0x1fb (5c768745)
5c7685d2 f7402400000400   test    dword ptr [eax+24h],40000h

Command

 Allocated range is
5b20000-5b21000,5b30000-5b31000,5b40000-5b41000,5b50000-5b51000

0:007> dd 5b20000
05b20000  05b30000 05b30004 05b30008 05b3000c
05b20010  05b30010 05b30014 05b30018 05b3001c
05b20020  05b30020 05b30024 05b30028 05b3002c
05b20030  05b30030 05b30034 05b30038 05b3003c
05b20040  05b30040 05b30044 05b30048 05b3004c
05b20050  05b30050 05b30054 05b30058 05b3005c
05b20060  05b30060 05b30064 05b30068 05b3006c
05b20070  05b30070 05b30074 05b30078 05b3007c
0:007> !vprot 5b20000
BaseAddress:       05b20000
AllocationBase:    05b20000
AllocationProtect: 00000004  PAGE_READWRITE
RegionSize:        00001000
State:             00001000  MEM_COMMIT
Protect:           00000004  PAGE_READWRITE
Type:              00020000  MEM_PRIVATE
0:007> !vprot 05b30000
BaseAddress:       05b30000
AllocationBase:    05b30000
AllocationProtect: 00000004  PAGE_READWRITE
RegionSize:        00001000
State:             00001000  MEM_COMMIT
Protect:           00000002  PAGE_READONLY
Type:              00020000  MEM_PRIVATE

0:007>
```

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 seem in the picture).



```
Pid 2152 - WinDbg:6.2.9200.16384 X86

Disassembly
Offset: @$scopeip
5c768593 cd29             int     29h
5c768595 e99171a2ff       jmp     MSHTML!CFormatInfo::Cleanup+0x1ba (5c18f72b)
5c76859a b904000000       mov     ecx,4
5c76859f cd29             int     29h
5c7685a1 e96971a2ff       jmp     MSHTML!CFormatInfo::Cleanup+0x19e (5c18f70f)
5c7685a6 f705801f225d00040000 test dword ptr [MSHTML!Microsoft_IEEnableBits (5d221f80)],400h
5c7685b0 0f85fed20d00     jne     MSHTML!`CBackgroundInfo::Property<CBackgroundImage>'::`7'::`dynamic atexit destructor for 'fieldDefaultValue''+0
5c7685b6 f7470800001000   test    dword ptr [edi+8],100000h
5c7685bd 7573             jne     MSHTML!CTreeNode::ComputeFormatsHelper+0xe8 (5c768632)
5c7685bf 8b07             mov     eax,dword ptr [edi]
5c7685c1 89442424         mov     dword ptr [esp+24h],eax
5c7685c5 f7402400000300   test    dword ptr [eax+24h],30000h ds:0023:05b20024=2400b305
5c7685cc 0f8573010000     jne     MSHTML!CTreeNode::ComputeFormatsHelper+0x1fb (5c768745)
5c7685d2 f7402400000400   test    dword ptr [eax+24h],40000h
5c7685d9 0f840fd30d00     je      MSHTML!`CBackgroundInfo::Property<CBackgroundImage>'::`7'::`dynamic atexit destructor for 'fieldDefaultValue''+0
5c7685df 8b4030           mov     eax,dword ptr [eax+30h]
5c7685e2 2403             and     al,3
5c7685e4 3c01             cmp     al,1
5c7685e6 0f85f1d20d00     jne     MSHTML!`CBackgroundInfo::Property<CBackgroundImage>'::`7'::`dynamic atexit destructor for 'fieldDefaultValue''+0
5c7685ec 8b442424         mov     eax,dword ptr [esp+24h]

Command
eax=0678de90 ebx=0e3d2fc0 ecx=00000000 edx=5d21edf0 esi=0679dfac edi=0679dfa0
eip=5c7685c5 esp=0644af40 ebp=0644bc18 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210246
MSHTML!CTreeNode::ComputeFormatsHelper+0x53:
5c7685c5 f7402400000300  test    dword ptr [eax+24h],30000h ds:0023:0678deb4=????????
0:007> r eax=5b20000
0:007> r
eax=05b20000 ebx=0e3d2fc0 ecx=00000000 edx=5d21edf0 esi=0679dfac edi=0679dfa0
eip=5c7685c5 esp=0644af40 ebp=0644bc18 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210246
MSHTML!CTreeNode::ComputeFormatsHelper+0x53:
5c7685c5 f7402400000300  test    dword ptr [eax+24h],30000h ds:0023:05b20024=2400b305

0:007> !dptrace_trace C:\Users\rohitwas\Desktop\PoCs\IE11_log1
```

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



```
5c76875c 8b4004          mov     eax,dword ptr [eax+4]
5c76875f 0f8497feffff    je      MSHTML!CTreeNode::ComputeFormatsHe
5c768765 e98cfeffff      jmp     MSHTML!CTreeNode::ComputeFormatsHe
5c76876a 6a01            push    1
5c76876c 33f6            xor     esi,esi
5c76876e 56              push    esi
5c76876f 56              push    esi
5c768770 ff158875345d    call    dword ptr [MSHTML!_im
5c768776 ebd6            jmp     MSHTML!CTreeNode::Com
5c768778 8bce            mov     ecx,esi
5c76877a e8f9579dff      call    MSHTML!CElement::Ensu
5c76877f ebae            jmp     MSHTML!CTreeNode::Com
5c768781 8b15f4fc215d    mov     edx,dword ptr [MSHTML
5c768787 64a12c000000    mov     eax,dword ptr fs:[000
5c76878d 8b0490          mov     eax,dword ptr [eax+ed
5c768790 8b7004          mov     esi,dword ptr [eax+4]
5c768793 8bd6            mov     edx,esi
```

```
0:007>
0:007>
0:007>
0:007>
0:007>
0:007>
0:007>
0:007>
0:007> !dptrace_forward 4 200 0x02


 Allocated range is
5b20000-5b21000,5b30000-5b31000,5b40000-5b41000,5b50000-5b51000

0:007> dd 5b20000
05b20000  05b30000 05b30004 05b30008 05b3000c
05b20010  05b30010 05b30014 05b30018 05b3001c
05b20020  05b30020 05b30024 05b30028 05b3002c
05b20030  05b30030 05b30034 05b30038 05b3003c
0:007>
```

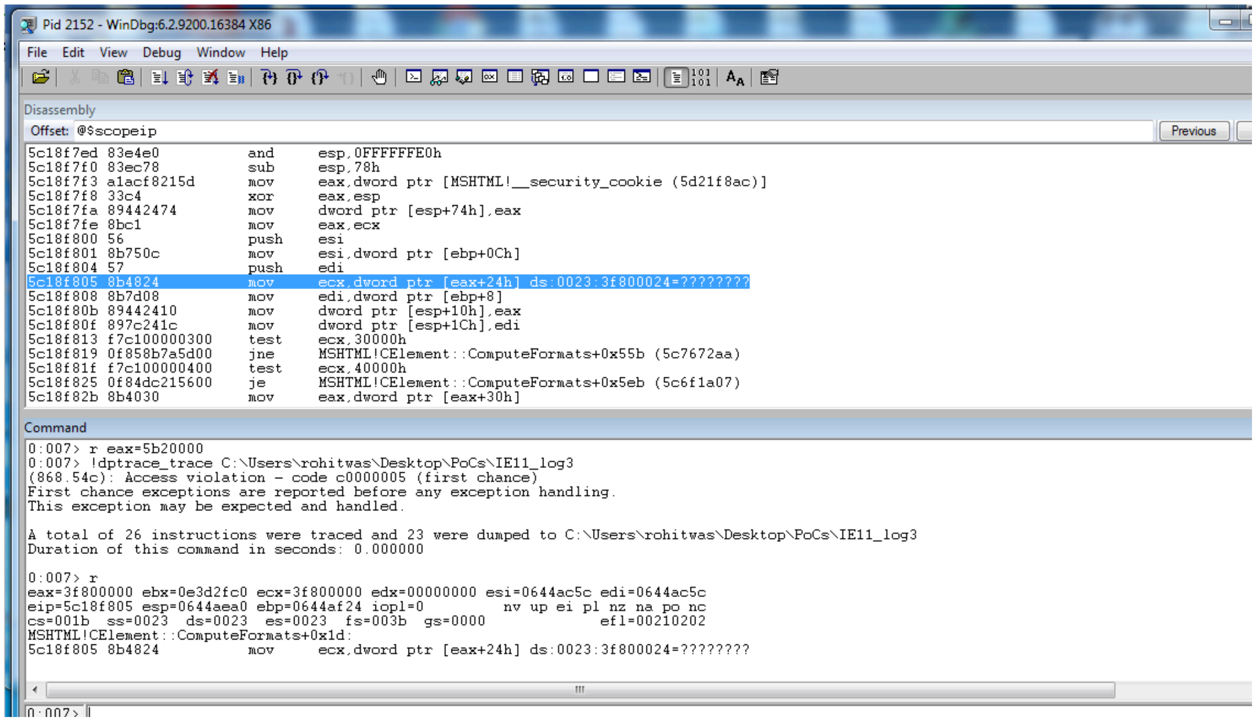
Start: 0x05b20000   End: 0x05b51000

```
916.      773d5daa ff12       call   dword ptr [edx]    ds:0023:7ffe0300={ntdll!KiFastSystemCall (773d71b0)}
917.      773d71b0 8bd4        mov   edx,esp
                               sysenter
                                 ret   0Ch
                                 or    dword ptr [ebp-10h],0FFFFFFFFh ss:0023:0644abe8=540c8944
                                 lea   eax,[ebp-14h]
                               push  eax
                               push  esi
                               push  dword ptr [ebp-4]   ss:0023:0644abf4=8c050000
          302e0ff7 mov   dword ptr [ebp-14h],0F70F2E80h ss:0023:0644abe4=d8ea6ee4
          45077   call   dword ptr [kernel32!_imp__NtWaitForSingleObject (77501418)] ds:0023:7750
          0000     mov   eax,187h
          e7f      mov   edx,offset SharedUserData!SystemCallStub (7ffe0300)
                  call   dword ptr [edx]    ds:0023:7ffe0300={ntdll!KiFastSystemCall (773d71b0)}
                   mov   edx,esp
                  sysenter
                    ret   0Ch
                  push  dword ptr [ebp-4]   ss:0023:0644abf4=8c050000
                  mov   esi,eax
          55077   call   dword ptr [kernel32!_imp__NtClose (775015cc)] ds:0023:775015cc={ntdll!Zw
          0000     mov   eax,32h
          e7f      mov   edx,offset SharedUserData!SystemCallStub (7ffe0300)
                  call   dword ptr [edx]    ds:0023:7ffe0300={ntdll!KiFastSystemCall (773d71b0)}
                   mov   edx,esp
                  sysenter
                    ret   4
942.      775988ce 8bc6        mov   eax,esi
943.      775988d0 5e         pop   esi
944.      775988d2 c3         ret
945.      775989d7 6a00        push  0
946.      775989d9 57         push  edi
947.      775989da 56         push  esi
948.      775989db ff15f0105077  call   dword ptr [kernel32!_imp__NtRaiseException (775010f0)] ds:0023:775010f0={
949.      773d6380 b80f010000   mov   eax,10Fh
950.      773d6385 ba0003fe7f   mov   edx,offset SharedUserData!SystemCallStub (7ffe0300)
951.      773d638a ff12       call   dword ptr [edx]    ds:0023:7ffe0300={ntdll!KiFastSystemCall (773d71b0)}
952.      773d71b0 8bd4        mov   edx,esp
953.      773d71b2 0f34        sysenter
```

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.

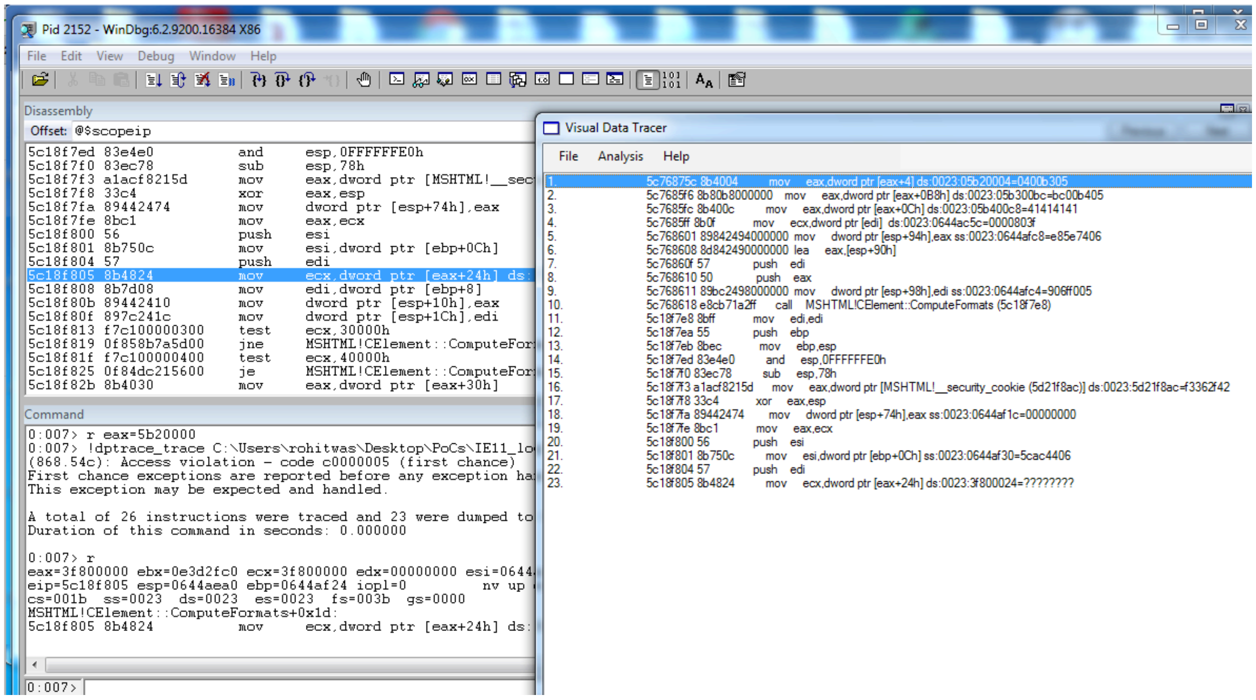
```
):007> ed eax+24 300
):007> r
eax=05ba0000 ebx=0ea4bfc0 ecx=00000000 edx=5d36edf0 esi=0616ffac edi=0616ffa0
eip=5c8b85c5 esp=05e1b320 ebp=05e1bfe8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
MSHTML!CTreeNode::ComputeFormatsHelper+0x53:
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 f7c100000300    test    ecx,30000h
5c18f819 0f858b7a5d00    jne     MSHTML!CElement::ComputeFormats+0x55b (5c7672aa)
5c18f81f f7c100000400    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 nz 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=????????

0:007>
```

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

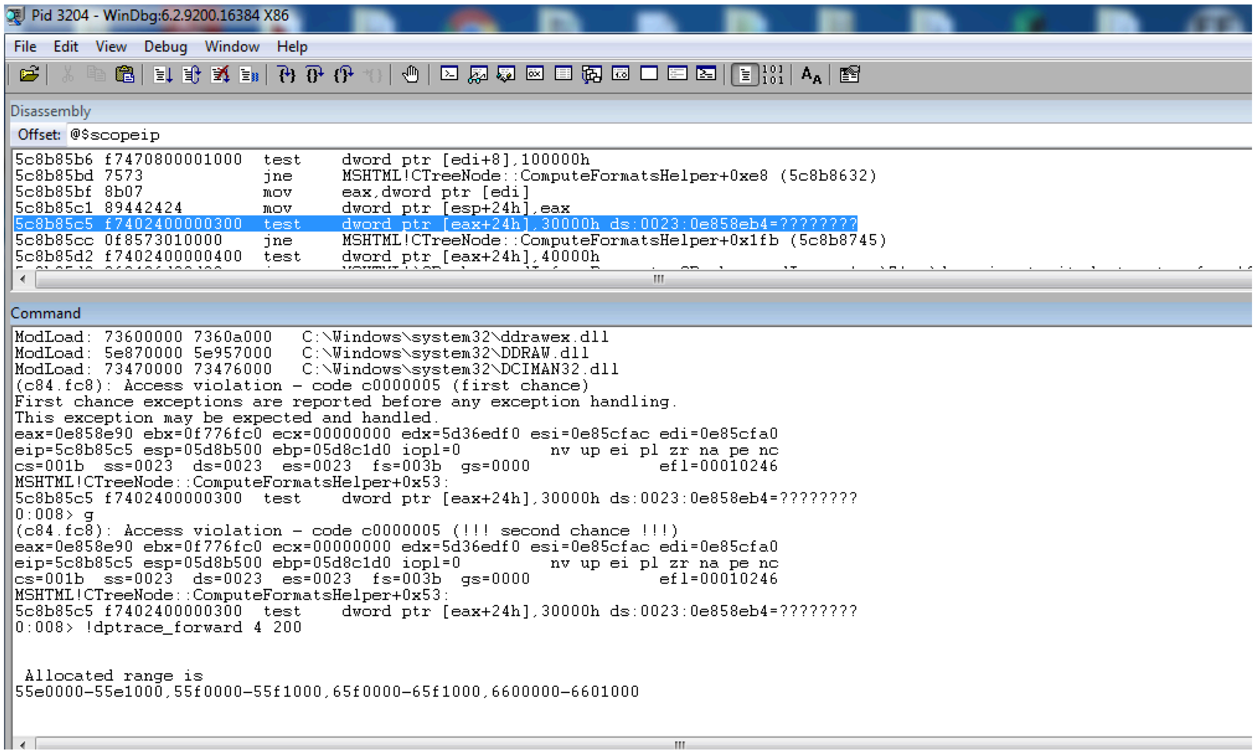


```
Disassembly
Offset: @$scopeip
5c18f7ed 83e4e0          and     esp,0FFFFFFE0h
5c18f7f0 83ec78          sub     esp,78h
5c18f7f3 a1acf8215d      mov     eax,dword ptr [MSHTML!__sec
5c18f7f8 33c4            xor     eax,esp
5c18f7fa 89442474        mov     dword ptr [esp+74h],eax
5c18f7fe 8bc1            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:
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 f7c100000300    test    ecx,30000h
5c18f819 0f858b7a5d00    jne     MSHTML!CElement::ComputeFor
5c18f81f f7c100000400    test    ecx,40000h
5c18f825 0f84dc215600    je      MSHTML!CElement::ComputeFor
5c18f82b 8b4030          mov     eax,dword ptr [eax+30h]

Command
0:007> r eax=5b20000
0:007> !dptrace_trace C:\Users\rohitwas\Desktop\PoCs\IE11_lo
(868.54c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception ha
This exception may be expected and handled.

A total of 26 instructions were traced and 23 were dumped to
Duration of this command in seconds: 0.000000

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

0:007>
```

```
Visual Data Tracer
1.   5c76875c 8b4004          mov   eax,dword ptr [eax+4] ds:0023:05b20004=0400b305
2.   5c7685f6 8b80b8000000    mov   eax,dword ptr [eax+0B8h] ds:0023:05b300bc=bc00b405
3.   5c7685fc 8b400c          mov   eax,dword ptr [eax+0Ch] ds:0023:05b400c8=41414141
4.   5c7685ff 8b0f            mov   ecx,dword ptr [edi]  ds:0023:0644ac5c=0000803f
5.   5c768601 89842494000000  mov   dword ptr [esp+94h],eax ss:0023:0644afc8=e85e7406
6.   5c768608 8d842490000000  lea   eax,[esp+90h]
7.   5c76860f 57              push  edi
8.   5c768610 50              push  eax
9.   5c768611 89bc2498000000  mov   dword ptr [esp+98h],edi ss:0023:0644afc4=906ff005
10.  5c768618 e8cb71a2ff      call  MSHTML!CElement::ComputeFormats (5c18f7e8)
11.  5c18f7e8 8bff            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.

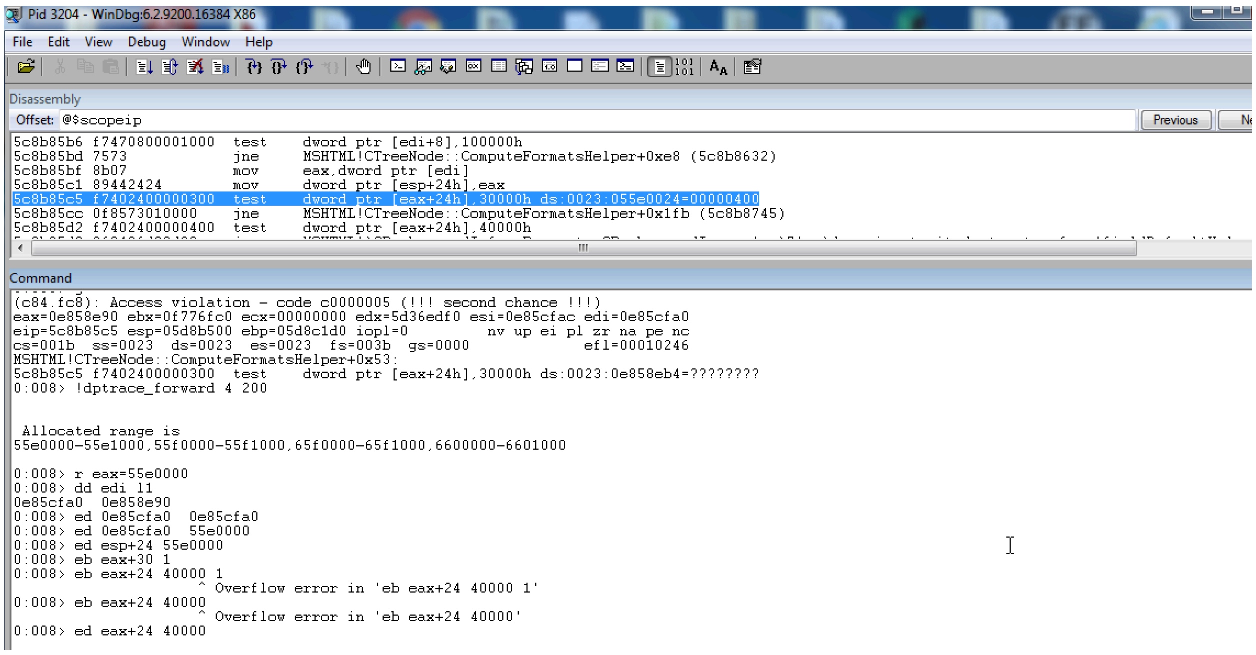
So we carry on another execution while trying to meet some other constraints and hit an alternate code path this time.


Pid 3204 - WinDbg:6.2.9200.16384 X86
File Edit View Debug Window Help
Disassembly
Offset: @$scopeip
Command

```
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
```

```
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> !dptrace_forward 4 200


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

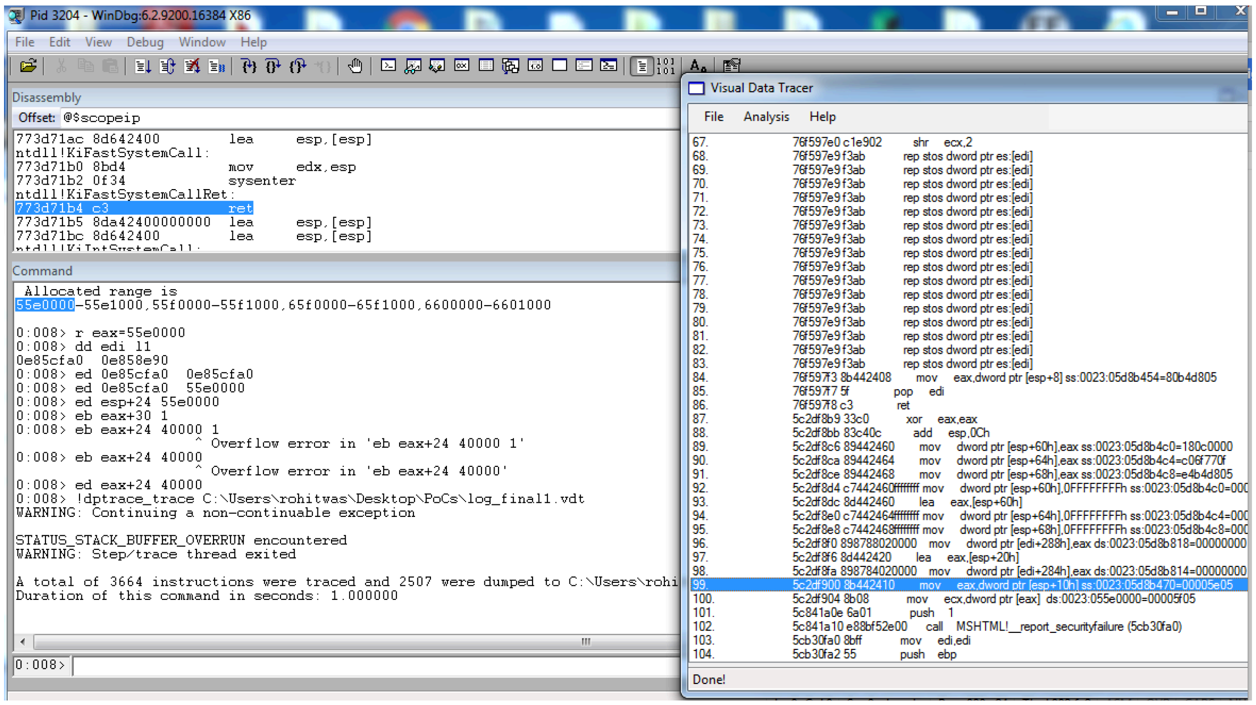
We start similarly at the crash point and allocate fake objects.


Pid 3204 - WinDbg:6.2.9200.16384 X86
File Edit View Debug Window Help
Disassembly
Offset: @$scopeip
Previous
Command

```
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
```

```
(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> !dptrace_forward 4 200


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

0:008> r eax=55e0000
0:008> dd edi l1
0e85cfa0  0e858e90
0:008> ed 0e85cfa0  0e85cfa0
0:008> ed 0e85cfa0  55e0000
0:008> ed 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
```

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 references to the same fake object in edi (CTreeNode *) and on the stack (esp+24).

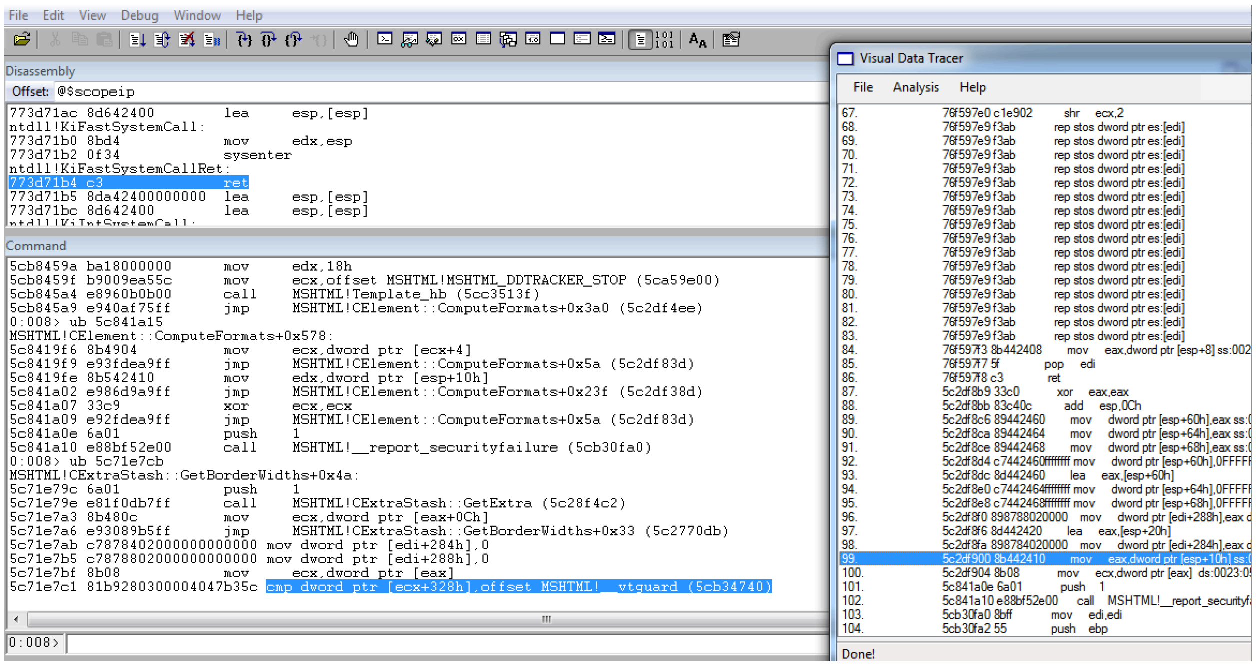


```
773d71ac 8d642400        lea     esp,[esp]
ntdll!KiFastSystemCall:
773d71b0 8bd4            mov     edx,esp
773d71b2 0f34            sysenter
ntdll!KiFastSystemCallRet:
773d71b4 c3              ret
773d71b5 8da42400000000  lea     esp,[esp]
773d71bc 8d642400        lea     esp,[esp]
```

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

0:008> r eax=55e0000
0:008> dd edi l1
0e85cfa0  0e858e90
0:008> ed 0e85cfa0  0e85cfa0
0:008> ed 0e85cfa0  55e0000
0:008> ed 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
0:008> !dptrace_trace C:\Users\rohitwas\Desktop\PoCs\log_final1.vdt
WARNING: Continuing a non-continuable exception

STATUS_STACK_BUFFER_OVERRUN encountered
WARNING: Step/trace thread exited

A total of 3664 instructions were traced and 2507 were dumped to C:\Users\rohi
Duration of this command in seconds: 1.000000
0:008>
```

```
67.    76f597e0 c1e902       shr    ecx,2
68.    76f597e9 f3ab         rep stos dword ptr es:[edi]
69.    76f597e9 f3ab         rep stos dword ptr es:[edi]
70.    76f597e9 f3ab         rep stos dword ptr es:[edi]
71.    76f597e9 f3ab         rep stos dword ptr es:[edi]
72.    76f597e9 f3ab         rep stos dword ptr es:[edi]
73.    76f597e9 f3ab         rep stos dword ptr es:[edi]
74.    76f597e9 f3ab         rep stos dword ptr es:[edi]
75.    76f597e9 f3ab         rep stos dword ptr es:[edi]
76.    76f597e9 f3ab         rep stos dword ptr es:[edi]
77.    76f597e9 f3ab         rep stos dword ptr es:[edi]
78.    76f597e9 f3ab         rep stos dword ptr es:[edi]
79.    76f597e9 f3ab         rep stos dword ptr es:[edi]
80.    76f597e9 f3ab         rep stos dword ptr es:[edi]
81.    76f597e9 f3ab         rep stos dword ptr es:[edi]
82.    76f597e9 f3ab         rep stos dword ptr es:[edi]
83.    76f597e9 f3ab         rep stos dword ptr es:[edi]
84.    76f597f3 8b442408     mov    eax,dword ptr [esp+8] ss:0023:05d8b454=80b4d805
85.    76f597f7 5f           pop    edi
86.    76f597f8 c3           ret
87.    5c2df8b9 33c0         xor    eax,eax
88.    5c2df8bb 83c40c       add    esp,0Ch
89.    5c2df8c6 89442460     mov    dword ptr [esp+60h],eax ss:0023:05d8b4c0=180c0000
90.    5c2df8ca 89442464     mov    dword ptr [esp+64h],eax ss:0023:05d8b4c4=c06f770f
91.    5c2df8ce 89442468     mov    dword ptr [esp+68h],eax ss:0023:05d8b4c8=e4b4d805
92.    5c2df8d4 c7442460ffffffff mov    dword ptr [esp+60h],0FFFFFFFFh ss:0023:05d8b4c0=000
93.    5c2df8dc 8d442460     lea    eax,[esp+60h]
94.    5c2df8e0 c7442464ffffffff mov    dword ptr [esp+64h],0FFFFFFFFh ss:0023:05d8b4c4=000
95.    5c2df8e8 c7442468ffffffff mov    dword ptr [esp+68h],0FFFFFFFFh ss:0023:05d8b4c8=000
96.    5c2df8f0 898788020000 mov    dword ptr [edi+288h],eax ds:0023:05d8b818=00000000
97.    5c2df8f6 8d442420     lea    eax,[esp+20h]
98.    5c2df8fa 898784020000 mov    dword ptr [edi+284h],eax ds:0023:05d8b814=00000000
99.    5c2df900 8b442410     mov    eax,dword ptr [esp+10h] ss:0023:05d8b470=00005e05
100.   5c2df904 8b08         mov    ecx,dword ptr [eax]  ds:0023:055e0000=00005f05
101.   5c841a0e 6a01         push   1
102.   5c841a10 e88bf52e00   call   MSHTML!__report_securityfailure (5cb30fa0)
103.   5cb30fa0 8bff         mov    edi,edi
104.   5cb30fa2 55           push   ebp
```

This time we hit a more interesting exception! Preliminary analysis shows us that the MSHTML!!report_securityfailure call was triggered due to a failed VTguard_check as shown below:



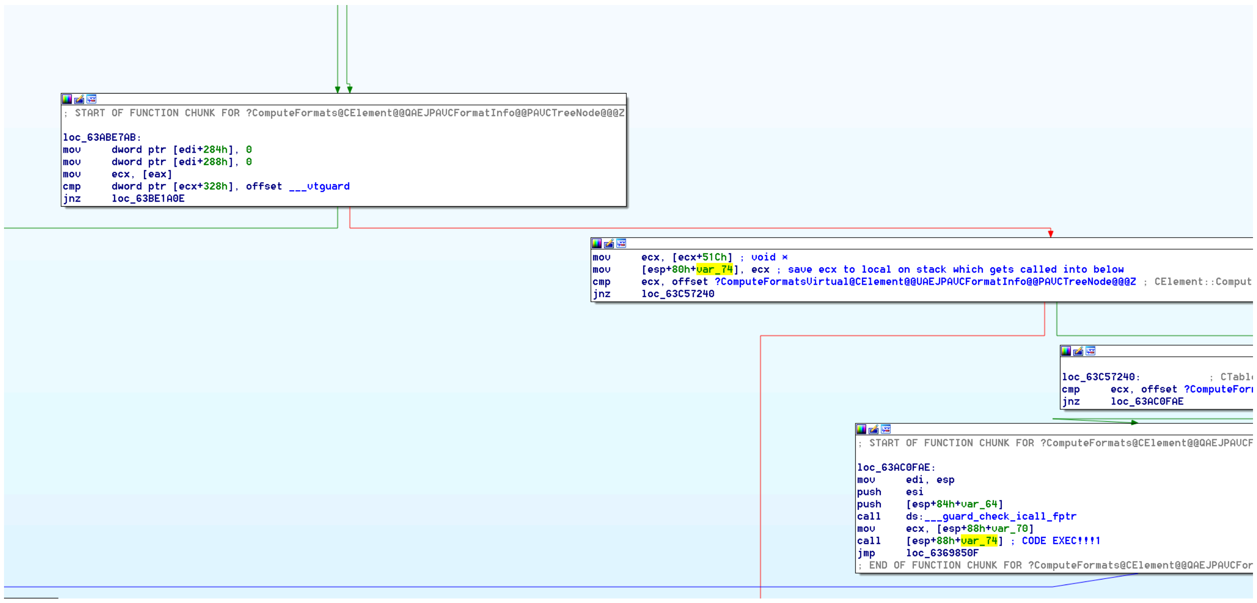
```
773d71ac 8d642400        lea     esp,[esp]
ntdll!KiFastSystemCall:
773d71b0 8bd4            mov     edx,esp
773d71b2 0f34            sysenter
ntdll!KiFastSystemCallRet:
773d71b4 c3              ret
773d71b5 8da42400000000  lea     esp,[esp]
773d71bc 8d642400        lea     esp,[esp]
```

```
5cb8459a ba18000000      mov     edx,18h
5cb8459f b9009ea55c      mov     ecx,offset MSHTML!MSHTML_DDTRACKER_STOP (5ca59e00)
5cb845a4 e8960b0b00      call    MSHTML!Template_hb (5cc3513f)
5cb845a9 e940af75ff      jmp     MSHTML!CElement::ComputeFormats+0x3a0 (5c2df4ee)
0:008> ub 5c841a15
MSHTML!CElement::ComputeFormats+0x578:
5c8419f6 8b4904          mov     ecx,dword ptr [ecx+4]
5c8419f9 e93fdea9ff      jmp     MSHTML!CElement::ComputeFormats+0x5a (5c2df83d)
5c8419fe 8b542410        mov     edx,dword ptr [esp+10h]
5c841a02 e986d9a9ff      jmp     MSHTML!CElement::ComputeFormats+0x23f (5c2df38d)
5c841a07 33c9            xor     ecx,ecx
5c841a09 e92fdea9ff      jmp     MSHTML!CElement::ComputeFormats+0x5a (5c2df83d)
5c841a0e 6a01            push    1
5c841a10 e88bf52e00      call    MSHTML!__report_securityfailure (5cb30fa0)
0:008> ub 5c71e7cb
MSHTML!CExtraStash::GetBorderWidths+0x4a:
5c71e79c 6a01            push    1
5c71e79e e81f0db7ff      call    MSHTML!CExtraStash::GetExtra (5c28f4c2)
5c71e7a3 8b480c          mov     ecx,dword ptr [eax+0Ch]
5c71e7a6 e93089b5ff      jmp     MSHTML!CExtraStash::GetBorderWidths+0x33 (5c2770db)
5c71e7ab c78784020000000000000 mov dword ptr [edi+284h],0
5c71e7b5 c78788020000000000000 mov dword ptr [edi+288h],0
5c71e7bf 8b08            mov     ecx,dword ptr [eax]
5c71e7c1 81b9280300004047b35c cmp dword ptr [ecx+328h],offset MSHTML!__vtguard (5cb34740)
0:008>
```

```
67.    76f597e0 c1e902       shr    ecx,2
68.    76f597e9 f3ab         rep stos dword ptr es:[edi]
69.    76f597e9 f3ab         rep stos dword ptr es:[edi]
70.    76f597e9 f3ab         rep stos dword ptr es:[edi]
71.    76f597e9 f3ab         rep stos dword ptr es:[edi]
72.    76f597e9 f3ab         rep stos dword ptr es:[edi]
73.    76f597e9 f3ab         rep stos dword ptr es:[edi]
74.    76f597e9 f3ab         rep stos dword ptr es:[edi]
75.    76f597e9 f3ab         rep stos dword ptr es:[edi]
76.    76f597e9 f3ab         rep stos dword ptr es:[edi]
77.    76f597e9 f3ab         rep stos dword ptr es:[edi]
78.    76f597e9 f3ab         rep stos dword ptr es:[edi]
79.    76f597e9 f3ab         rep stos dword ptr es:[edi]
80.    76f597e9 f3ab         rep stos dword ptr es:[edi]
81.    76f597e9 f3ab         rep stos dword ptr es:[edi]
82.    76f597e9 f3ab         rep stos dword ptr es:[edi]
83.    76f597e9 f3ab         rep stos dword ptr es:[edi]
84.    76f597f3 8b442408     mov    eax,dword ptr [esp+8] ss:002
85.    76f597f7 5f           pop    edi
86.    76f597f8 c3           ret
87.    5c2df8b9 33c0         xor    eax,eax
88.    5c2df8bb 83c40c       add    esp,0Ch
89.    5c2df8c6 89442460     mov    dword ptr [esp+60h],eax ss:(
90.    5c2df8ca 89442464     mov    dword ptr [esp+64h],eax ss:(
91.    5c2df8ce 89442468     mov    dword ptr [esp+68h],eax ss:(
92.    5c2df8d4 c7442460ffffffff mov    dword ptr [esp+60h],0FFFFF
93.    5c2df8dc 8d442460     lea    eax,[esp+60h]
94.    5c2df8e0 c7442464ffffffff mov    dword ptr [esp+64h],0FFFFF
95.    5c2df8e8 c7442468ffffffff mov    dword ptr [esp+68h],0FFFFF
96.    5c2df8f0 898788020000 mov    dword ptr [edi+288h],eax d
97.    5c2df8f6 8d442420     lea    eax,[esp+20h]
98.    5c2df8fa 898784020000 mov    dword ptr [edi+284h],eax d
99.    5c2df900 8b442410     mov    eax,dword ptr [esp+10h] ss:(
100.   5c2df904 8b08         mov    ecx,dword ptr [eax]  ds:0023:0!
101.   5c841a0e 6a01         push   1
102.   5c841a10 e88bf52e00   call   MSHTML!__report_securityfa
103.   5cb30fa0 8bff         mov    edi,edi
104.   5cb30fa2 55           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:



```
; START OF FUNCTION CHUNK FOR ?ComputeFormats@CElement@@QAEJPAVCFormatInfo@@PAVCTreeNode@@@Z

loc_63ABE7AB:
mov     dword ptr [edi+284h], 0
mov     dword ptr [edi+288h], 0
mov     ecx, [eax]
cmp     dword ptr [ecx+328h], offset ___vtguard
jnz     loc_63BE1A0E
```

```
mov     ecx, [ecx+51Ch] ; void *
mov     [esp+80h+var_74], ecx ; save ecx to local on stack which gets called into below
cmp     ecx, offset ?ComputeFormatsVirtual@CElement@@UAEJPAVCFormatInfo@@PAVCTreeNode@@@Z ; CElement::Comput
jnz     loc_63C57240
```

```
loc_63C57240:                  ; CTabl
cmp     ecx, offset ?ComputeFor
jnz     loc_63AC0FAE
```

```
; START OF FUNCTION CHUNK FOR ?ComputeFormats@CElement@@QAEJPAVCF

loc_63AC0FAE:
mov     edi, esp
push    esi
push    [esp+84h+var_64]
call    ds:___guard_check_icall_fptr
mov     ecx, [esp+88h+var_70]
call    [esp+88h+var_74] ; CODE EXEC!!!1
jmp     loc_6369850F
; END OF FUNCTION CHUNK FOR ?ComputeFormats@CElement@@QAEJPAVCFor
```

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 group those crashes for analysis and quickly prioritizes issues (since crashes appears 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 comming 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
[4] BFS algorithm. Link: http://en.wikipedia.org/wiki/Breadth-first_search
[5] GitHub Repository for this Paper/Presentation: Link: