

Dynamic Program Analysis and Software Exploitation

From the crash to the exploit code

Rodrigo Rubira Branco (BSDaemon)

Founder Dissect || PE – Now the Qualys Vulnerability & Malware Research Lab

rodrigo *noSPAM* kernelhacking.com

<http://twitter.com/bsddaemon>

Agenda

- Objectives
- History
- Introduction

- Concepts of Taint Analysis
 - Taint Sources
 - Intermediate Languages and Tainted Sources
 - Explosion of Watched Data

- Backward Taint Analysis
 - From the crash to the exploit code

- Existent solutions and comparisions

- Future

Objectives

- Explain my latest Phrack Article
- Demonstrate how vulnerability finding works (or is supposed to work)
- Give some concepts about program analysis for vulnerability exploitation
- Explain the challenges the exploit writer faces nowadays
- Be fun?

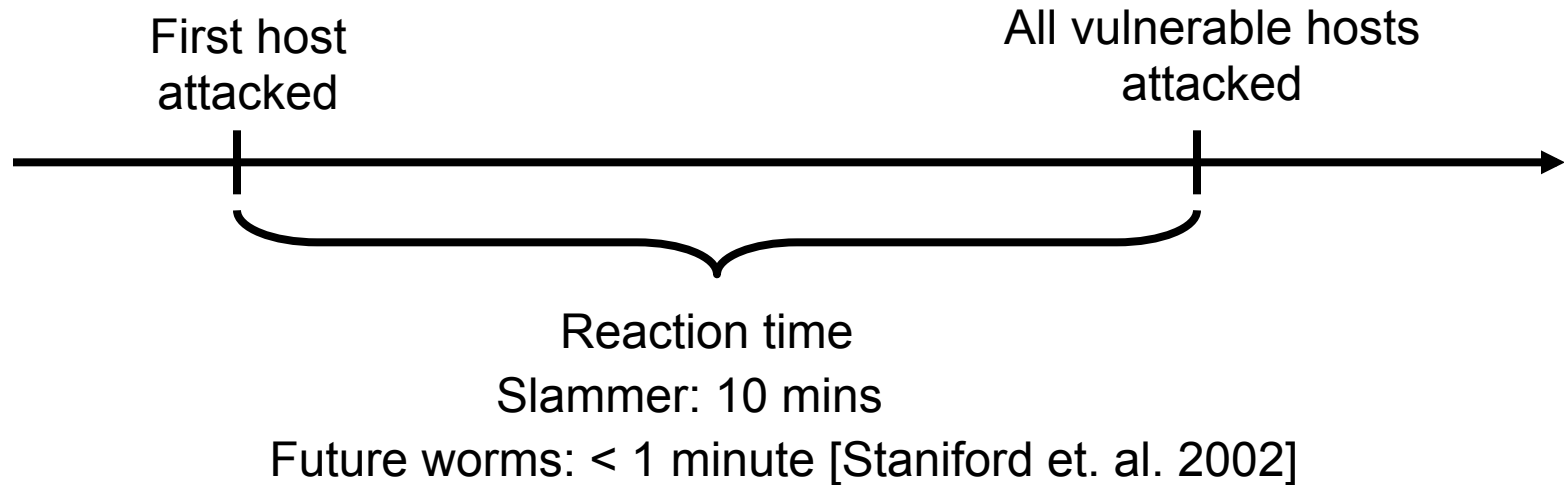
Security nowadays (yeap, again the same slides)

- Buggy programs deployed on critical servers
- Rapidly-evolving threats, attackers and tools (exploitation frameworks)
- Lack of developers training, resources and people to fix problems and create safe code
- **That's why we are here today, right?**

Sorry, really sorry

- Usually I start from the end and here I was supposed to show an 0day vulnerability in Excel
- Everything is ready to be presented using the tool that I'll explain in the presentation
- The problem: Microsoft did not issue the patch yet -> Well, they delayed it (it was supposed to be released in March, now only in April)
 - I'm not blaming Microsoft, they've been very supportive

Security nowadays – 0day challenge



"0day Statistics

Average 0day lifetime:

348 days

Shortest life:

99 days

Longest life:

1080 (3 years)"

- Justine Aitel

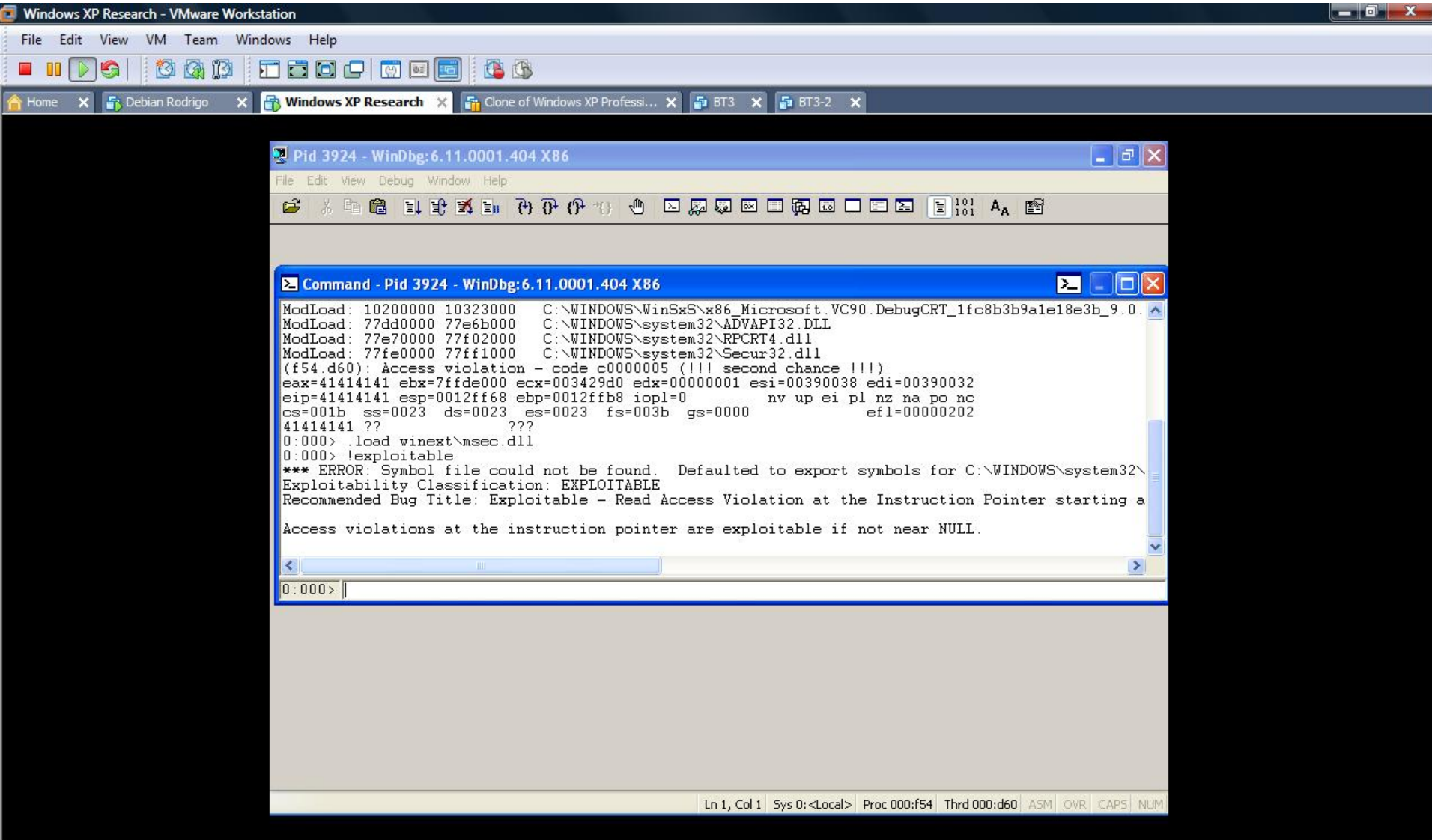
History

- Original Motivation: Complex client-side vulnerability in a closed (at the time) file format
- Extended Motivation: Trying to better analyse hundred thousands of bugs in word (search for Ben Nagy, Coseinc)
- Initial version integrated with a fuzzer, only for Linux (showed past year here in Troopers)
- Ported version for Solaris to analyze a vulnerability released by Secunia in the same software RISE Security released a vulnerability some time before
- Thanks to Julio Auto parallel research in the same field, we created together the WinDBG version presented here

Introduction – What is program analysis for us?

- Make a computational system reason automatically (or at least with little human assistance) about the behaviour of a program and draw conclusions that are somehow useful
- Help us to determine exploitability of vulnerabilities, or to rapidly develop an exploit code
- The most widely known solution for the exploitability determination is given by Microsoft: !exploitable

!exploitable



```
Pid 3924 - WinDbg:6.11.0001.404 X86
File Edit View Debug Window Help
ModLoad: 10200000 10323000 C:\WINDOWS\WinSxS\x86_Microsoft_VC90.DebugCRT_1fc8b3b9a1e18e3b_9.0.
ModLoad: 77dd0000 77e6b000 C:\WINDOWS\system32\ADVAPI32.DLL
ModLoad: 77e70000 77f02000 C:\WINDOWS\system32\RPCRT4.dll
ModLoad: 77fe0000 77ff1000 C:\WINDOWS\system32\Secur32.dll
(f54.d60): Access violation - code c0000005 (!!! second chance !!!)
eax=41414141 ebx=7ffde000 ecx=003429d0 edx=00000001 esi=00390038 edi=00390032
eip=41414141 esp=0012ff68 ebp=0012ffb8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
41414141 ??                ???
0:000> .load winext\nsec.dll
0:000> !exploitable
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\WINDOWS\system32\
Exploitability Classification: EXPLOITABLE
Recommended Bug Title: Exploitable - Read Access Violation at the Instruction Pointer starting a
Access violations at the instruction pointer are exploitable if not near NULL.
0:000>
```

To direct input to this VM, click inside or press Ctrl+G.

!exploitable

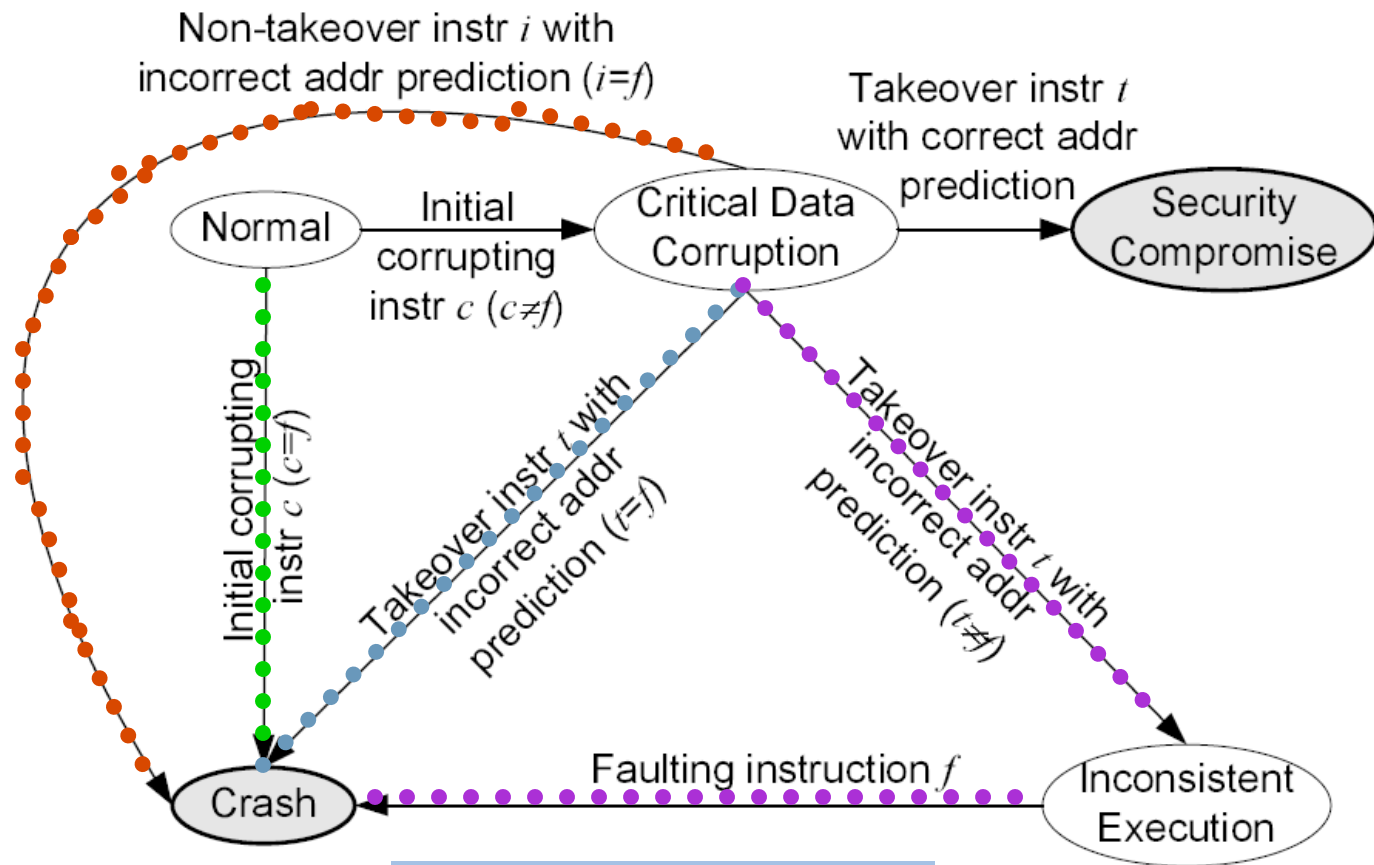
- This is incorrectly classified as EXPLOITABLE because the tool always assume that the attacker has control over all the input operands
- In this presentation, we are going to try to answer the question: Are the input operands in the attacker's control?

Concepts of Taint Analysis

- Taint Analysis is one kind of program flow analysis and we use it to define the influence of external data (attacker's controlled data) over the analyzed application
- Since the information flows, or is copied to, or influence other data there is a need to follow this influence in order to determine the control over specific areas (registers, memory locations). This is a requirement for determine exploitability

State Transition for Memory Corruption

- Case 1 (green): Format String
- Case 2 and 3 (red and blue): buffer overflow
- Case 4 (purple): unpredictable



c: corrupting instruction
t: takeover instruction
f: faulting instruction

Source:
Automatic Diagnosis and
Response to Memory
Corruption Vulnerabilities

So, what?

- Legitimate assumption:
 - To change the execution of a program illegitimately we need to have a value being derived from the attacker's input (which we call: controlled by the attacker)
- String sizes and format strings should usually be supplied by the code itself, not from external, un-trusted inputs.
- Any data originated from or arithmetically derived from un-trusted source must be inspected.

Taint Analysis

- Tainted data: Data from un-trusted source
- Keep track of tainted data (from un-trusted source)
- Monitors program execution to track how tainted attribute propagates
- Detect when tainted data is used in sensitive way

Taint Propagation

- When a tainted location is used in such a way that a value of other data is derived from the tainted data (like in mathematical operations, move instructions and others) we mark the other location as tainted as well
- The transitive relation is:
 - If information A is used to derive information B:
 - » $A \rightarrow t(B) \rightarrow$ Direct flow
 - If B is used to derive information C:
 - » $B \rightarrow t(C) \rightarrow$ Direct flow
 - » Thus: $A \rightarrow t(C) \rightarrow$ Indirect flow
- Due to the transitive nature, you can analyze individual transitions or the whole block ($A \rightarrow t(C)$)

Location

- A location is defined as:
 - Memory address and size
 - Register name (we use the register entirely, not partially -> thus %al and %eax are the same)
 - » When setting a register, I set it higher (setting %al as tainted will also taint %eax)
 - » When clearing a register, I clear it lower
- To keep track over bit operations in a register it is important to taint the code-block level of a control flow graph
 - This create extra complexity due to the existence of the flow graph and data flow dependencies graph
 - The dependencies graph represents the influence of a source data in the operation been performed

Taint Sources

- Any information in the control of the attacker is tainted (remember the transitive relation of the tainted data)
- The more tainted information, the bigger the propagation and the required resources in order to keep track of that
- Tainted data is only deleted when it receives an assignment from an untainted source or an assignment from a tainted source resulting in a constant value not controlled by the attacker

Flows

- **Explicit flow:**
 - `mov %eax, A`
- **Implicit flow:**
 - `If (x == 1) y=0;`
- **Conditional statements require a special analysis approach:**
 - In our case, we are analyzing the trace of a program (not the program itself, but only what was executed during the section that generated the crash)
 - We have two different analysis step: tracing and analysis

Special Situations

- Partial Tainting: When the untrusted source does not completely control the tainted data
- Tainting Merge: When there are two different untrusted sources being used to derive some data
- Data
 - In Use: when it is referenced by an operation
 - Defined: when the data is modified

Inheritance problems

Problem: state explosion for binary operations !

Application

```
mov %eax ← A  
mov B ← %eax
```

```
add %ebx ← D
```

Propagation Tracking

```
taint(%eax) = taint(A)  
taint(B)     = taint(%eax)
```

```
taint(%ebx) |= taint(D)
```

Inheritance Tracking

%eax inherits from A
B inherits from %eax

insert D into %ebx's
inherit-from list

Events

Rare

e.g., malloc/free, system calls

Frequent

e.g., memory access,
data movement

Tracking Instructions

- Pure assignments: Easy to track
 - If a tainted location is used to define another location, this new location will be tainted
- Operations over strings are tainted when:
 - They are used to calculate string sizes using a tainted location
 - » `a = strlen(tainted(string));`
 - » Since the 'string' is tainted, I assume the attacker controls 'a'
 - Search for some specific char using a tainted location, defining a flag if found or not found
 - » `pointer = strchr(tainted(string), some_char);`
 - » `If (pointer) flag=1;`
 - » 'flag' is tainted if the attacker controls 'string' or 'some_char'

Tracking Instructions

- Arithmetic instructions with at least one tainted data usually define tainted results
- Those arithmetic instructions can be simplified to map to boolean operations and then the following rules applies

OR truth table

~~XOR truth table~~

X	Y	X or Y
0	0	0
0	1	1
1	0	1
1	1	0

Arithmetics with Tainted Data

- OR Operand
 - If the untainted data is 1, the result is untainted
 - If the untainted data is 0, the result is tainted
- AND Operand
 - If the untainted data is 0, the result is untainted
 - If the untainted data is 1, the result is tainted
- XOR Operand
 - If it is an xor against itself, the result is untainted
 - Otherwise, the result is tainted

Eflags and Flow Information

- The eflags register can also be tainted to monitor flags conditions influencing in operations (and flow)
- In the presented approach, conditional branches are taken care due to the trace generated by the WinDBG plugin (single-stepping)

Backward Taint Analysis

- Divide the analysis process in two parts:
 - A trace from a good state to the crash (incrementally dumped to a file) -> Gather substantial information about the target application when it receives the input data, which is formally named 'analysis'
 - Analysis of the trace file -> Formally defined as 'verification' step, where the conclusive analysis is done

The need for intermediate languages...

- Assembly instructions have explicit operands, which are easy to deal with, and sometimes implicit operands:
 - Instruction: `push eax`
 - Explicit operand: `eax`
 - What it really does?
 - » `ESP = ESP - 4` (a subtraction)
 - » `SS:[ESP] = EAX` (a move)
 - » Here we have `ESP` and `SS` as implicit operands
- Tks to Edgar Barbosa for this great example!

The tracing step

- Instead of using an intermediate language, I play straight with the debugger interfaces (WinDBG)
- The tracer stores some useful information, like effective addresses and data values and also simplifies the instructions for easy parsing:
 - `CMPXCHG r/m32, r32` -> 'Compare EAX with r/m32. If equal, ZF is set and r32 is loaded into r/m32. Else, clear ZF and load r/m32 into AL'
 - » Such an instruction creates the need for conditional taints, since by controlling `%eax` and `r32` the attacker controls `r/m32` too.

Tracing File

- **Contains:**
 - Mnemonic of the instruction
 - Operands
 - Dependences for the source operand
 - » Eg: Elements of an indirectly addressed memory
 - » This creates a tree of the dataflow, with a root in the crash instruction
- **The verification step reads this file and:**
 - Search this tree using a BFS algorithm

Theoretical Example

- 1-) `mov edi, 0x1234` ; dst=edi, src=0x1234
- 2-) `mov eax, [0xABCD]` ; dst=eax, src=ptr 0xABCD ;
Note 0xABCD is evil addr
- 3-) `lea ebx, [eax+ecx*8]` ; dst=ebx, src=eax,
srcdep1=ecx
- 4-) `mov [edi], ebx` ; dst=ptr 0x1234, src=ebx
- 5-) `mov esi, [edi]` ; dst=esi, src=ptr 0x1234,
srcdep1=edi
- 6-) `mov edx, [esi]` ; Crash!!!

Theoretical Example – The Tree

- 6-) Where does [esi] come from?
- 5-) [edi] is moved to esi, where edi comes from and what does exist in [edi]?
- 4-) [edi] receives ebx and edi is defined in 1-) from a fixed value
- 3-) ebx comes from a lea instruction that uses eax and ecx
- 2-) eax receives a value controlled by the attacker
- ... ecx is out of the scope here :)

Limitation of the approach

- Since I only use the trace information, if the crash input data does not force a flow, I can't see the influence of the input over this specific flow data
- To solve that:
 - If a jmp is dependent of a flag, the attacker controls branch decision
 - Control over a branch means tainted EIP
 - To define the value of EIP, consider:
 - » The address if the jump is taken
 - » The address of the next instruction (if the jump is not taken)
 - » The value of the interesting flag register (0 or 1)
 - » Then: $\%eip \leftarrow (\text{address of the next instruction}) + \text{value of the register flag} * (|\text{address if jump is taken} - \text{address of the next instruction}|)$

Existent Solutions and Comparisions

■ !exploitable

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

■ Spider Pig

- Created by Piotr Bania
- Not available for testing, but from the paper: It is much more advanced than the provided tool (but well, it is not available?)
 - » Virtual Code Integration (or Dynamic Binary Rewriting) -> Discussed in my previous year presentation about Fuzzers here in Troopers
 - » Disputable Objects: Partially controlled data is analyzed using the parent data

■ Taint Bochs

- Used for tracking sensitive data lifecycle in memory

Existent Solutions and Comparisions

■ Taint Check

- Uses DynamicRIO or Valgrind
- Taint Seed: Defining the tainted values (data coming from the network for example)
- Taint Tracker: Tracks the propagation
- Taint Assert: Alert about security violations
- Used while testing software to detect overflow conditions, does not really help in the exploit creation
 - » In the article I also provided a heap analysis tool for Embedded Linux Architecture (ARM) since the Memcheck plugin for Valgrind is not available on this architecture

■ Bitblaze

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

How it works (or is supposed to)

```
ModLoad: 75da0000 75e5d000 C:\WINDOWS\system32\SXS.DLL
(ac.594): Break instruction exception - code 80000003 (first chance)
eax=7ffdd000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
eip=7c81a3e1 esp=009bffc0 ebp=009bfff4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\
ntdll!DbgBreakPoint:
7c81a3e1 cc                int     3
0:003> bp kernel32!CreateFileW
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\
0:003> g
```

```
*BUSY* | Debuggee is running...
```

Start tracing

```
0:003> .load vdt-tracer
0:003> !vdt_help
Visual Data Tracer v1.0 Alpha - Copyright (C) 2008-2010
License: This software was created as companion to a Phrack Article.
Developed by Rodrigo Rubira Branco (BSDaemon) <rodrigo@risesecurity.org> and
Julio Auto <julio@julioauto.com>

!vdt_trace <filename>           - trace the program until a breakpoint or
                                in a file to be later consumed by the Vis
!vdt_help                       - this help screen
0:003> !vdt_trace excel_phrack.vdt
```

Find something from your input to search for in memory

XVI32 - FIL573.XLS

File Edit Search Address Bookmarks Tools XVIscript Help

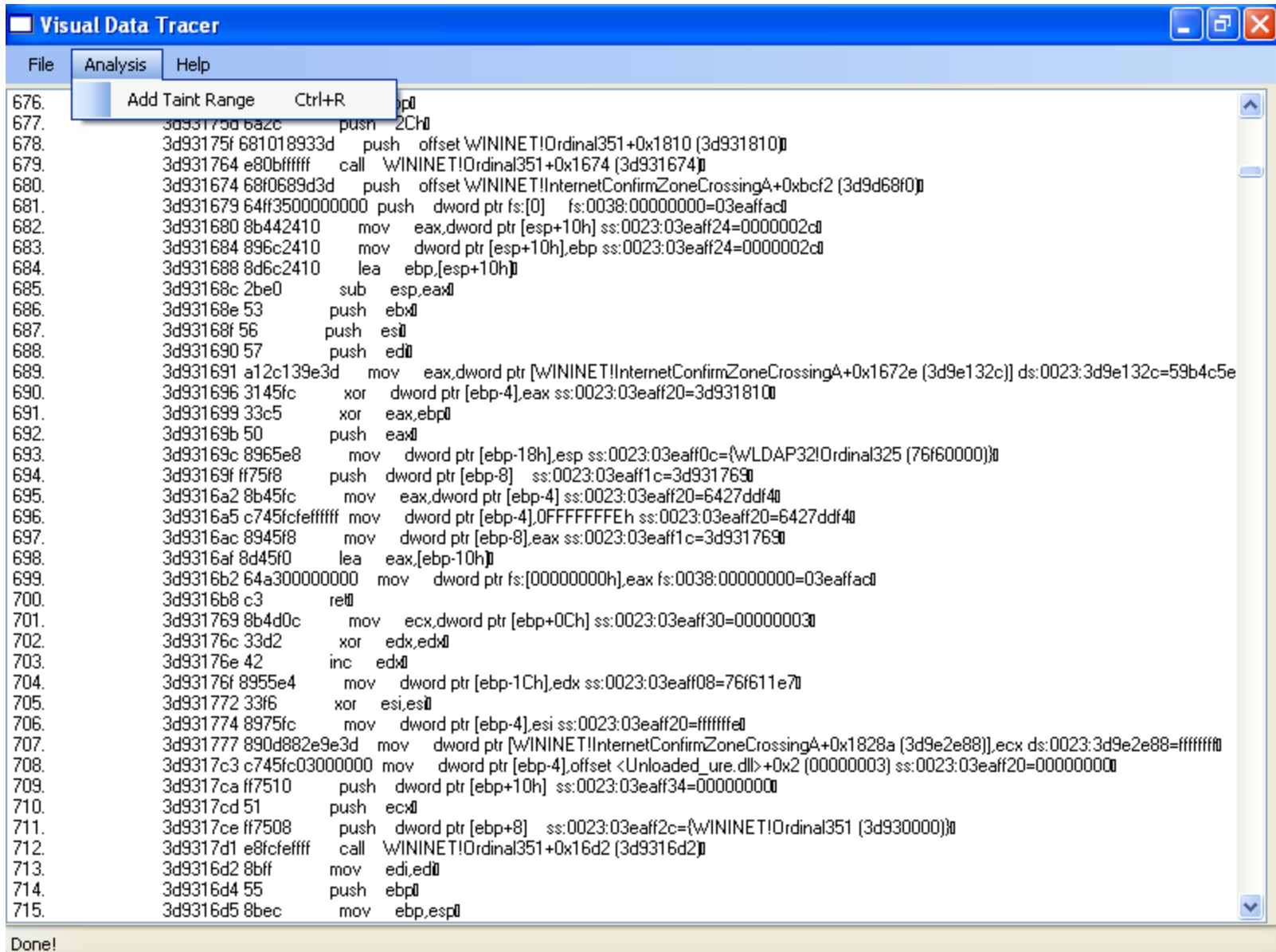
Address	Hex Data	ASCII
0	09 04 06 00 78 CC 02 3D 40 39 03 00 09 02 04 00 60 03 FF	J - x î 7 = @ 9 L 7 J ' L ŷ
13	3E 00 09 40 5F C2 02 C0 88 40 00 0B C4 02 73 00 41 40 0C	> @ _ Â 7 À ^ @ ð Ä 7 s A @ □
26	01 C3 08 36 00 3E 02 12 00 B4 B9 40 49 00 00 C0 72 C2 01	Ä 6 > 7 î ' ' @ I À r Ä
39	01 01 A0 80 3D 82 03 C0 00 1D 00 0F 00 03 41 5B 98 00 00	e = , L À % L A [~
4C	01 00 19 40 02 06 EF C0 01 FA 00 80 1F 00 C4 89 C6 0C 3F	† @ 7 - i À ú e Ä % E □ ?
5F	02 7F 10 7F 10 97 7F 10 7F 10 75 10 FE A0 7D 05 01 C1 75	7 † † - † † u † p } Á u
72	03 2B 08 A0 4C 00 E0 85 9F F2 F9 00 4F 68 10 AB 91 08 00	L + □ L à ... Ÿ ò ù Oh † « ' □
85	2B D0 27 B3 D9 30 40 04 C0 60 00 21 64 D5 E1 03 48 60 01	+ ð ' ' Û 0 @ J À ' ! d Õ á L H '
98	04 60 00 50 60 00 61 02 6A 60 E0 00 12 60 00 7C 60 00 E1	J ' P ' a 7 j ' à † ' ' á
AB	5B 94 AB E0 00 81 56 A0 E0 00 0D 60 00 AC 60 00 5A 13 60	[" « à V à ' - ' z !! '
BE	00 B8 60 00 21 0E E4 80 07 1E 05 60 01 07 60 00 53 42 6C	' ' ! ß ä € • ' • ' S B L
D1	61 63 18 6B 00 00 E1 01 61 08 6D 61 72 00 63 65 6C 6F 2E	a c † k á a □ mar celo .
E4	67 75 61 00 7A 7A 65 6C 6C 69 00 65 05 62 03 10 60 05 4D	gua z z e l l i e b L † ' M
F7	69 63 72 6F 80 73 6F 66 74 20 45 78 80 04 02 00 A2 50 C6	i c r o € s o f t E x € J 7 e P E
10A	27 B2 71 83 C3 02 01 61 01 80 FB 39 14 61 1A 01 63 01 00	' ' q f Ä 7 a € û 9 ¶ a + c
11D	0F 66 64 86 A8 C3 FE 01 E1 8B 40 07 5F 00 3F 04 3F 04 3F	% f d + " Ä p á < @ • _ ? J ? J ?
130	04 3F 04 FF 3F 04 3F 04 3F 04 3F 04 3F 04 3F 04 3F 04 3F	J ? J ŷ ? J ? J ? J ? J ? J ? J ?
143	04 FF 3F 04 3F 04 3F 04 3F 04 3F 04 3F 04 3F 04 3F 04 FF	J ŷ ? J ? J ? J ? J ? J ? J ? J ?
156	3F 04 3F 04 3F 04 1F 02 1F 01 1F 01 1F 01 1F 01 FF 1F 01	? J ? J ? J 7 ŷ

Adr. dec: 232 Char dec: 122 Overwrite

Locate the input in the program's memory

```
0:000> s -[w1]a 0x0 L?800000000 "zzelli"  
0x001393ce  
0x001717e0  
0x30862168
```

Open the tracing file



The screenshot shows the Visual Data Tracer application window. The title bar reads "Visual Data Tracer" and includes standard window controls. The menu bar contains "File", "Analysis", and "Help". The "Analysis" menu is open, showing "Add Taint Range" (Ctrl+R). The main window displays assembly code with the following lines:

```
676. 3d93175d 6a2c      push  2Ch
677. 3d93175f 681018933d    push  offset WININET!Ordinal351+0x1810 (3d931810)
678. 3d931764 e80bffff      call  WININET!Ordinal351+0x1674 (3d931674)
679. 3d931674 68f0689d3d    push  offset WININET!InternetConfirmZoneCrossingA+0xbc2 (3d9d68f0)
680. 3d931679 64ff3500000000 push  dword ptr fs:[0] fs:0038:00000000=03eaffac
681. 3d931680 8b442410      mov   eax,dword ptr [esp+10h] ss:0023:03eaff24=0000002c
682. 3d931684 896c2410      mov   dword ptr [esp+10h],ebp ss:0023:03eaff24=0000002c
683. 3d931688 8d6c2410      lea  ebp,[esp+10h]
684. 3d93168c 2be0         sub   esp,eax
685. 3d93168e 53          push  ebx
686. 3d93168f 56          push  esi
687. 3d931690 57          push  edi
688. 3d931691 a12c139e3d    mov   eax,dword ptr [WININET!InternetConfirmZoneCrossingA+0x1672e (3d9e132c)] ds:0023:3d9e132c=59b4c5e
689. 3d931696 3145fc      xor   dword ptr [ebp-4],eax ss:0023:03eaff20=3d931810
690. 3d931699 33c5       xor   eax,ebp
691. 3d93169b 50          push  eax
692. 3d93169c 8965e8      mov   dword ptr [ebp-18h],esp ss:0023:03eaff0c={WLDAP32!Ordinal325 (76f60000)}
693. 3d93169f ff75f8      push  dword ptr [ebp-8] ss:0023:03eaff1c=3d931769
694. 3d9316a2 8b45fc      mov   eax,dword ptr [ebp-4] ss:0023:03eaff20=6427ddf4
695. 3d9316a5 c745fcfeffff mov   dword ptr [ebp-4],0FFFFFFFh ss:0023:03eaff20=6427ddf4
696. 3d9316ac 8945f8      mov   dword ptr [ebp-8],eax ss:0023:03eaff1c=3d931769
697. 3d9316af 8d45f0      lea  eax,[ebp-10h]
698. 3d9316b2 64a300000000 mov   dword ptr fs:[00000000h],eax fs:0038:00000000=03eaffac
699. 3d9316b8 c3         ret
700. 3d931769 8b4d0c      mov   ecx,dword ptr [ebp+0Ch] ss:0023:03eaff30=00000003
701. 3d93176c 33d2       xor   edx,edx
702. 3d93176e 42         inc  edx
703. 3d93176f 8955e4      mov   dword ptr [ebp-1Ch],edx ss:0023:03eaff08=76f611e7
704. 3d931772 33f6       xor   esi,esi
705. 3d931774 8975fc      mov   dword ptr [ebp-4],esi ss:0023:03eaff20=ffffffe1
706. 3d931777 89d882e9e3d mov   dword ptr [WININET!InternetConfirmZoneCrossingA+0x1828a (3d9e2e88)],ecx ds:0023:3d9e2e88=fffffffd
707. 3d9317c3 c745fc03000000 mov   dword ptr [ebp-4],offset <Unloaded_ure.dll>+0x2 (00000003) ss:0023:03eaff20=00000000
708. 3d9317ca ff7510      push  dword ptr [ebp+10h] ss:0023:03eaff34=00000000
709. 3d9317cd 51         push  ecx
710. 3d9317ce ff7508      push  dword ptr [ebp+8] ss:0023:03eaff2c={WININET!Ordinal351 (3d930000)}
711. 3d9317d1 e8fcfefff      call  WININET!Ordinal351+0x16d2 (3d9316d2)
712. 3d9316d2 8bff       mov   edi,edi
713. 3d9316d4 55         push  ebp
714. 3d9316d5 8bec       mov   ebp,esp
```

At the bottom of the window, the status bar displays "Done!".

Add the taint range

The screenshot shows the Visual Data Tracer application window. The main window displays a list of assembly instructions with their addresses and disassembled forms. A dialog box titled "Add Taint Range" is open in the foreground, allowing the user to specify a range of memory addresses to be tracked for taint analysis. The dialog box has two input fields for "Start" and "End" addresses, both set to 0x001393ce and 0x001717e0. There are "Add", "Remove", and "Close" buttons. The background assembly code includes instructions like pop, push, call, mov, lea, sub, xor, and retd.

Visual Data Tracer

File Analysis Help

676. 3d931757 5d pop ebp
677. 3d93175d 6a2c push 2Ch
678. 3d93175f 681018933d push offset WININET!Ordinal351+0x1810 (3d931810)
679. 3d931764 e80bffff call WININET!Ordinal351+0x1674 (3d931674)
680. 3d931674 68f0689d3d push offset WININET!InternetConfirmZoneCrossingA+0xbcf2 (3d9d68f0)
681. 3d931679 64ff3500000000 push dword ptr fs:[0] fs:0038:00000000=03eaff24
682. 3d931680 8b442410 mov eax,dword ptr [esp+10h] ss:0023:03eaff24=0000002d
683. 3d931684 896c2410 mov edx,dword ptr [esp+10h] ds:0023:03eaff24=0000002d
684. 3d931688 8d6c2410 lea edi,[esp+10h] ds:0023:03eaff24=0000002d
685. 3d93168c 2be0 sub esp,edi
686. 3d93168e 53 push ebx
687. 3d93168f 56 push esi
688. 3d931690 57 push edi
689. 3d931691 a12c139e3d mov ebx,dword ptr [00139e3d] ds:0023:3d9e132c=59b4c5e
690. 3d931696 3145fc xor dw,ebx
691. 3d931699 33c5 xor eax,ecx
692. 3d93169b 50 push eax
693. 3d93169c 8965e8 mov ecx,dword ptr [00139e58] ds:0023:3d9e132c=76f60000
694. 3d93169f ff75f8 push dword ptr [00139e58] ds:0023:3d9e132c=76f60000
695. 3d9316a2 8b45fc mov ebx,dword ptr [00139e58] ds:0023:3d9e132c=76f60000
696. 3d9316a5 c745fcffffff mov ecx,dword ptr [00139e58] ds:0023:3d9e132c=76f60000
697. 3d9316ac 8945f8 mov ecx,dword ptr [00139e58] ds:0023:3d9e132c=76f60000
698. 3d9316af 8d45f0 lea eax,[00139e58] ds:0023:3d9e132c=76f60000
699. 3d9316b2 64a300000000 mov eax,dword ptr [00139e58] ds:0023:3d9e132c=76f60000
700. 3d9316b8 c3 retd
701. 3d931769 8b4d0c mov ebx,dword ptr [0017170c] ds:0023:3d9e132c=76f60000
702. 3d93176c 33d2 xor edx,edx
703. 3d93176e 42 inc edx
704. 3d93176f 8955e4 mov ecx,dword ptr [0017170c] ds:0023:3d9e132c=76f60000
705. 3d931772 33f6 xor esi,esi
706. 3d931774 8975fc mov dword ptr [ebp-4],esi ss:0023:03eaff20=fffffffa
707. 3d931777 890d882e9e3d mov dword ptr [WININET!InternetConfirmZoneCrossingA+0x1828a (3d9e2e88)],ecx ds:0023:3d9e2e88=fffffffa
708. 3d9317c3 c745fc03000000 mov dword ptr [ebp-4],offset <Unloaded_ure.dll>+0x2 (00000003) ss:0023:03eaff20=00000003
709. 3d9317ca ff7510 push dword ptr [ebp+10h] ss:0023:03eaff34=00000000
710. 3d9317cd 51 push ecx
711. 3d9317ce ff7508 push dword ptr [ebp+8] ss:0023:03eaff2c={WININET!Ordinal351 (3d930000)}
712. 3d9317d1 e8fcffff call WININET!Ordinal351+0x16d2 (3d9316d2)
713. 3d9316d2 8bff mov edi,edi
714. 3d9316d4 55 push ebp
715. 3d9316d5 8bec mov ebp,esp

Done!

Add Taint Range

Start End

0x001393ce 0x001717e0 Add

Start End

0x001393ce 0x001717e0 Remove

Close

Analyze

The screenshot shows the Visual Data Tracer application window. The title bar reads "Visual Data Tracer" and includes standard window controls. The menu bar contains "File", "Analysis", and "Help". The main area displays a list of assembly instructions with their addresses and hex values. Instruction 697 is highlighted in blue, and a context menu is open over it, showing "Check Taint Of" (with a right-pointing arrow) and "Scroll To Item". The register "eax" is visible in the menu. The status bar at the bottom left shows "Done!".

Address	Hex	Instruction
676.	3d931757 5d	pop ebp
677.	3d93175d 6a2c	push 2Ch
678.	3d93175f 681018933d	push offset 'WININET!Ordinal351+0x1810 (3d931810)'
679.	3d931764 e80bffff	call 'WININET!Ordinal351+0x1674 (3d931674)'
680.	3d931674 68f0689d3d	push offset 'WININET!InternetConfirmZoneCrossingA+0xbc2 (3d9d68f0)'
681.	3d931679 64ff3500000000	push dword ptr fs:[0] fs:0038:00000000=03eaffac
682.	3d931680 8b442410	mov eax,dword ptr [esp+10h] ss:0023:03eaff24=0000002c
683.	3d931684 896c2410	mov dword ptr [esp+10h],ebp ss:0023:03eaff24=0000002c
684.	3d931688 8d6c2410	lea ebp,[esp+10h]
685.	3d93168c 2be0	sub esp,eax
686.	3d93168e 53	push ebx
687.	3d93168f 56	push esi
688.	3d931690 57	push edi
689.	3d931691 a12c139e3d	mov eax,dword ptr ['WININET!InternetConfirmZoneCrossingA+0x1672e (3d9e132c)'] ds:0023:3d9e132c=59b4c5e
690.	3d931696 3145fc	xor dword ptr [ebp-4],eax ss:0023:03eaff20=3d931810
691.	3d931699 33c5	xor eax,ebp
692.	3d93169b 50	push eax
693.	3d93169c 8965e8	mov dword ptr [ebp-18h],esp ss:0023:03eaff0c={'LDAP32!Ordinal325 (76f60000)'}'
694.	3d93169f ff75f8	push dword ptr [ebp-8] ss:0023:03eaff1c=3d931769
695.	3d9316a2 8b45fc	mov eax,dword ptr [ebp-4] ss:0023:03eaff20=6427ddf4
696.	3d9316a5 c745fcfeffff	mov dword ptr [ebp-4],0FFFFFFEh ss:0023:03eaff20=6427ddf4
697.	3d9316ac 8945f8	mov dword ptr [ebp-4],eax ss:0023:03eaff20=3d931769
698.	3d9316af 8d45f0	lea eax,[ebp-4] ss:0023:03eaff20=3d931769
699.	3d9316b2 64a300000000	mov dword ptr [eax],eax ss:0023:03eaff20=03eaffac
700.	3d9316b8 c3	ret
701.	3d931769 8b4d0c	mov ecx,dword ptr [ebp+0Ch] ss:0023:03eaff30=00000003
702.	3d93176c 33d2	xor edx,edx
703.	3d93176e 42	inc edx
704.	3d93176f 8955e4	mov dword ptr [ebp-1Ch],edx ss:0023:03eaff08=76f611e7
705.	3d931772 33f6	xor esi,esi
706.	3d931774 8975fc	mov dword ptr [ebp-4],esi ss:0023:03eaff20=ffffffe0
707.	3d931777 89d882e9e3d	mov dword ptr ['WININET!InternetConfirmZoneCrossingA+0x1828a (3d9e2e88)'],ecx ds:0023:3d9e2e88=ffffffe0
708.	3d9317c3 c745fc0300000000	mov dword ptr [ebp-4],offset <Unloaded_ure.dll>+0x2 (00000003) ss:0023:03eaff20=00000000
709.	3d9317ca ff7510	push dword ptr [ebp+10h] ss:0023:03eaff34=00000000
710.	3d9317cd 51	push ecx
711.	3d9317ce ff7508	push dword ptr [ebp+8] ss:0023:03eaff2c={'WININET!Ordinal351 (3d930000)'}'
712.	3d9317d1 e8fcfefff	call 'WININET!Ordinal351+0x16d2 (3d9316d2)'
713.	3d9316d2 8bff	mov edi,edi
714.	3d9316d4 55	push ebp
715.	3d9316d5 8bec	mov ebp,esp

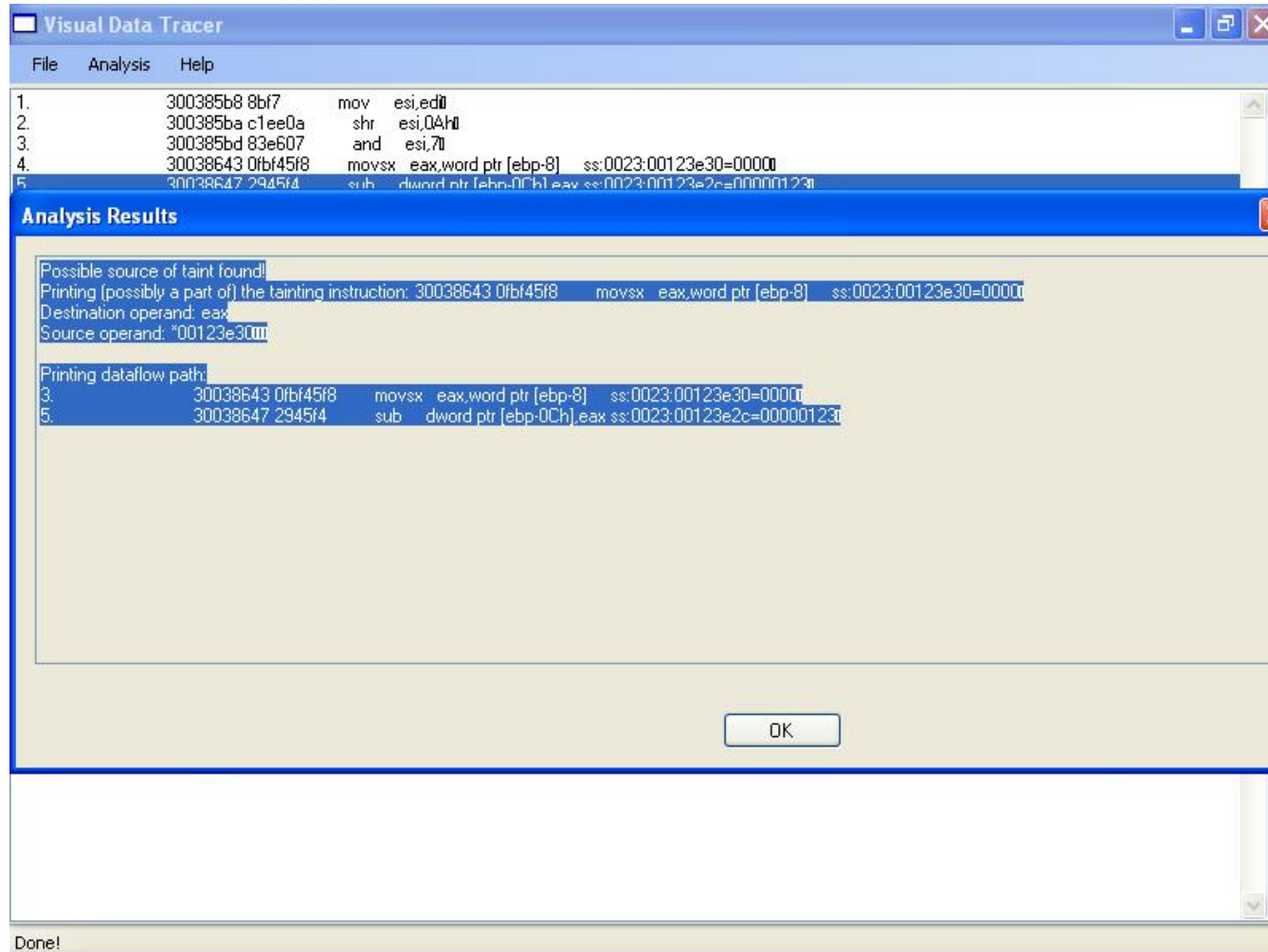
Analyze

The screenshot shows the Visual Data Tracer application window. The title bar reads "Visual Data Tracer" and includes standard window controls. The menu bar contains "File", "Analysis", and "Help". The main area displays assembly code with columns for instruction address, hex offset, instruction, and comment. Instruction 682 is highlighted in blue. A context menu is open over this instruction, showing "Check Taint Of" (with a right-pointing arrow) and "Scroll To Item". To the right of the menu, a small panel shows "*03eaff24" and "esp".

Address	Offset	Instruction	Comment
676.	3d931757 5d	pop ebp	
677.	3d93175d 6a2c	push 2Ch	
678.	3d93175f 681018933d	push offset \WININET!Ordinal351+0x1810 (3d931810)	
679.	3d931764 e80bffff	call \WININET!Ordinal351+0x1674 (3d931674)	
680.	3d931674 68f0689d3d	push offset \WININET!InternetConfirmZoneCrossingA+0xbcf2 (3d9d68f0)	
681.	3d931679 64ff3500000000	push dword ptr fs:[0] fs:0038:00000000=03eaffa	
682.	3d931680 8b442410	mov eax,dword ptr [esp+10h] ss:0023:03eaff24=0000002c	
683.	3d931684 896c2410	mov dword ptr [esp+10h],	
684.	3d931688 8d6c2410	lea ebp,[esp+10h]	
685.	3d93168c 2be0	sub esp,eax	
686.	3d93168e 53	push ebx	
687.	3d93168f 56	push esi	
688.	3d931690 57	push edi	
689.	3d931691 a12c139e3d	mov eax,dword ptr [\WININET!InternetConfirmZoneCrossingA+0x1672e (3d9e132c)] ds:0023:3d9e132c=59b4c5e	
690.	3d931696 3145fc	xor dword ptr [ebp-4],eax ss:0023:03eaff20=3d931810	
691.	3d931699 33c5	xor eax,ebp	
692.	3d93169b 50	push eax	
693.	3d93169c 8965e8	mov dword ptr [ebp-18h],esp ss:0023:03eaff0c=(\LDAP32!Ordinal325 (76f60000))	
694.	3d93169f ff75f8	push dword ptr [ebp-8] ss:0023:03eaff1c=3d931763	
695.	3d9316a2 8b45fc	mov eax,dword ptr [ebp-4] ss:0023:03eaff20=6427ddf4	
696.	3d9316a5 c745fcffff	mov dword ptr [ebp-4],FFFFFFFFEh ss:0023:03eaff20=6427ddf4	
697.	3d9316ac 8945f8	mov dword ptr [ebp-8],eax ss:0023:03eaff1c=3d931763	
698.	3d9316af 8d45f0	lea eax,[ebp-10h]	
699.	3d9316b2 64a300000000	mov dword ptr fs:[00000000h],eax fs:0038:00000000=03eaffa	
700.	3d9316b8 c3	ret	
701.	3d931769 8b4d0c	mov ecx,dword ptr [ebp+0Ch] ss:0023:03eaff30=00000003	
702.	3d93176c 33d2	xor edx,edx	
703.	3d93176e 42	inc edx	
704.	3d93176f 8955e4	mov dword ptr [ebp-1Ch],edx ss:0023:03eaff08=76f611e7	
705.	3d931772 33f6	xor esi,esi	
706.	3d931774 8975fc	mov dword ptr [ebp-4],esi ss:0023:03eaff20=ffffffe	
707.	3d931777 890d882e9e3d	mov dword ptr [\WININET!InternetConfirmZoneCrossingA+0x1828a (3d9e2e88)],ecx ds:0023:3d9e2e88=ffffff	
708.	3d9317c3 c745fc03000000	mov dword ptr [ebp-4],offset <Unloaded_ure.dll>+0x2 (00000003) ss:0023:03eaff20=00000000	
709.	3d9317ca ff7510	push dword ptr [ebp+10h] ss:0023:03eaff34=00000000	
710.	3d9317cd 51	push ecx	
711.	3d9317ce ff7508	push dword ptr [ebp+8] ss:0023:03eaff2c=(\WININET!Ordinal351 (3d930000))	
712.	3d9317d1 e8fcef	call \WININET!Ordinal351+0x16d2 (3d9316d2)	
713.	3d9316d2 8bf	mov edi,edi	
714.	3d9316d4 55	push ebp	
715.	3d9316d5 8bec	mov ebp,esp	

Done!

Analyze



Future

- I can't foresee the future!
- Hope more researchers will contribute in the future
- The code needs immediate support for extended coverage of x86 instructions, speed enhancements, introduction of heuristical detection over user input (so you don't need to specify memory ranges to watch)

Special Thanks

- To the Troopers Staff, for trusting me once again... This conference is awesome
- Prime Security Team, specially Filipe Balestra
- RISE Security Group, yeah, we still exist, but now everybody works
- Special thanks to Julio Auto who developed everything with me (and besides me, lots of patience I know...)

End! Really !?

Rodrigo Rubira Branco (BSDaemon)

Founder Dissect || PE – Now the Qualys Vulnerability & Malware Research Lab

rodrigo *noSPAM* kernelhacking.com

<http://twitter.com/bsddaemon>