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/bsdaemon
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 comparisons

- 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

First host attacked

All vulnerable hosts attacked

Reaction time
Slammer: 10 mins
Future worms: < 1 minute [Staniford et. al. 2002]

“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`
 Kernel Hacking: If you really know, you can hack! – http://www.kernelhacking.com/rodrigo

```c
int main() {
    asm {
        mov eax, 0x41414141
        call eax
    }
}
```
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

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

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

A location is defined as:
- Memory address and size
- Register name (we use the register entirely, not partially -> thus %al and %eax are the same)
  » When setting a register, 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!

<table>
<thead>
<tr>
<th>Application</th>
<th>Propagation Tracking</th>
<th>Inheritance Tracking</th>
</tr>
</thead>
</table>
| mov %eax ← A
mov B ← %eax | \( taint(\%eax) = taint(A) \)
\( taint(B) = taint(\%eax) \) | %eax inherits from A
B inherits from %eax |
| add %ebx ← D | \( taint(\%ebx) |= taint(D) \) | 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
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 substraction)
    » 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!!!
Theorical 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 <- (address of the next instruction) + value of the register flag \* ( |address if jump is taken – address of the next instruction| )
Existential Solutions and Comparisons

- **!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 Comparisons

- **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=009bfffcc ebp=009bffff4 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>
!vdt_help
0:003> !vdt_trace excel_phrack.vdt
```

- trace the program until a breakpoint or
- in a file to be later consumed by the Vis
- this help screen
Find something from your input to search for in memory
Locate the input in the program’s memory

0:000> s -[w1]a 0x0 I?80000000 "zzelli"
0x001393ce
0x001717e0
0x30862168
Open the tracing file
Add the taint range

Visual Data Tracer

File Analysis Help

Add Taint Range

Start End
0x001393ce 0x001717e0
Add

Start End
0x001393ce 0x001717e0
Remove

Close

Done!
Analyze

Visual Data Tracer

File  Analysis  Help

676.  3d31757 5d  pop  ebp
677.  3d3175d 6a2c  push  2Ch
678.  3d31757 68101893d  push  offset WININET1Ordinal351+0x1810 (3d31810)
679.  3d31764 88bfff  call  WININET1Ordinal351+0x1674 (3d31764)
680.  3d31764 680689d3  push  offset WININET1InternetConfirmZoneCrossingA+0xcbf2 (3d9d8f0)
681.  3d31769 64f350000000  push  dword ptr fs:0[0]  fs:0038:00000000=03eaffac
682.  3d31600 8b442410  mov  eax,dword ptr [esp+10h]  ss:0023:03eaff24=0000002d
683.  3d31684 89ec2410  mov  dword ptr [esp+10h],ebp  ss:0023:03eaff24=0000002d
684.  3d31688 8d6c2410  lea  ebp,[esp+10h]
685.  3d3168c 2be0  sub  esp,eax
686.  3d3168e 53  push  ebx
687.  3d3168f 56  push  esi
688.  3d31690 57  push  edi
689.  3d31691 312c13e3d  mov  eax,dword ptr [WININET1InternetConfirmZoneCrossingA+0x1672e (3d9e132c)]  ds:0023:3d9e132c=59b45e
690.  3d31696 3145cf  xor  dword ptr [ebp-4],eax  ss:0023:03eaff0f=3d31810
691.  3d31699 33c5  xor  eax,ebp
692.  3d3169b 50  push  eax
693.  3d3169c 8965e8  mov  dword ptr [ebp-18h],esp  ss:0023:03eaff0e=\WLDAP321Ordinal325 (76f60000)
694.  3d3169f 677518  push  dword ptr [ebp-8]  ss:0023:03eaff1c=3d31769
695.  3d316a2 8b445c  mov  eax,dword ptr [ebp-4]  ss:0023:03eaff20=6427dd4f
696.  3d316a5 c745cfef17  mov  dword ptr [ebp-4],0FFFFFFFEh  ss:0023:03eaff20=6427dd4f
697.  3d316ac 89d418  mov  dword ptr [ebp+1Ch],edi  ss:0023:03eaff08=76611e7f
700.  3d316b2 e830000000  mov  dword ptr [ebp+18h],edi  ss:0023:03eaff0c=03eaffac
701.  3d316b8 3c  ret
702.  3d3176d 8b4d0c  mov  ecx,dword ptr [ebp+0Ch]  ss:0023:03eaff30=00000003
703.  3d3176c 33d2  xor  edx,edi
704.  3d3176e 42  inc  edi
705.  3d3176f 8955e4  mov  dword ptr [ebp+1Ch],edx  ss:0023:03eaff08=76611e7f
706.  3d31772 33f6  xor  esi,edi
707.  3d31777 89d82e9e3d  mov  dword ptr [WININET1InternetConfirmZoneCrossingA+0x1828a (3d9e2e88)],ecx  ds:0023:3d9e2e88=ffffffb
708.  3d31777 c745cf300000  mov  dword ptr [ebp-4],offset Unloaded_uere.d1b=0x2 (00000003)  ss:0023:03eaff20=00000000
709.  3d31777 7f510  push  dword ptr [ebp+10h]  ss:0023:03eaff34=00000000
710.  3d31777 51  push  ecx
711.  3d31777 ff7508  push  dword ptr [ebp+8]  ss:0023:03eaff2c=\WININET1Ordinal351 (3d930000)
712.  3d31777 e8cfeffff  call  WININET1Ordinal351+0x16d2 (3d9316d2)
713.  3d316d2 8bf  mov  edi,edi
714.  3d316d4 55  push  ebp
715.  3d316d5 8bec  mov  ebp,esp

Done!

Kernel Hacking: If you really know, you can hack! – http://www.kernelhacking.com/rodrigo
Analyze

Kernel Hacking: If you really know, you can hack! – http://www.kernelhacking.com/rodrigo
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/bsdaemon