COSEINC®

Solid Security.Verified.

Taint Analysis

Edgar Barbosa

H2HC 2009

São Paulo

Who am I?

- Currently working for COSEINC, a Singapore based security company.
- Old rootkit.com contributor (opc0de)
- Discovered the KdVersionBlock trick (used by Windows memory forensic analysis tools)
- One of the developers of BluePill, a hardware-based virtualization rootkit.

Outline

- Concepts
- Taint analysis on the x86 architecture
- Taint objects and instructions
- Advanced tainting
- References

Motivation

- The motivation for this research came from the following questions:
  - Is it possible to **measure** the level of “influence” that external data have over some application? E.g. network packets or PDF files.

Taint Analysis

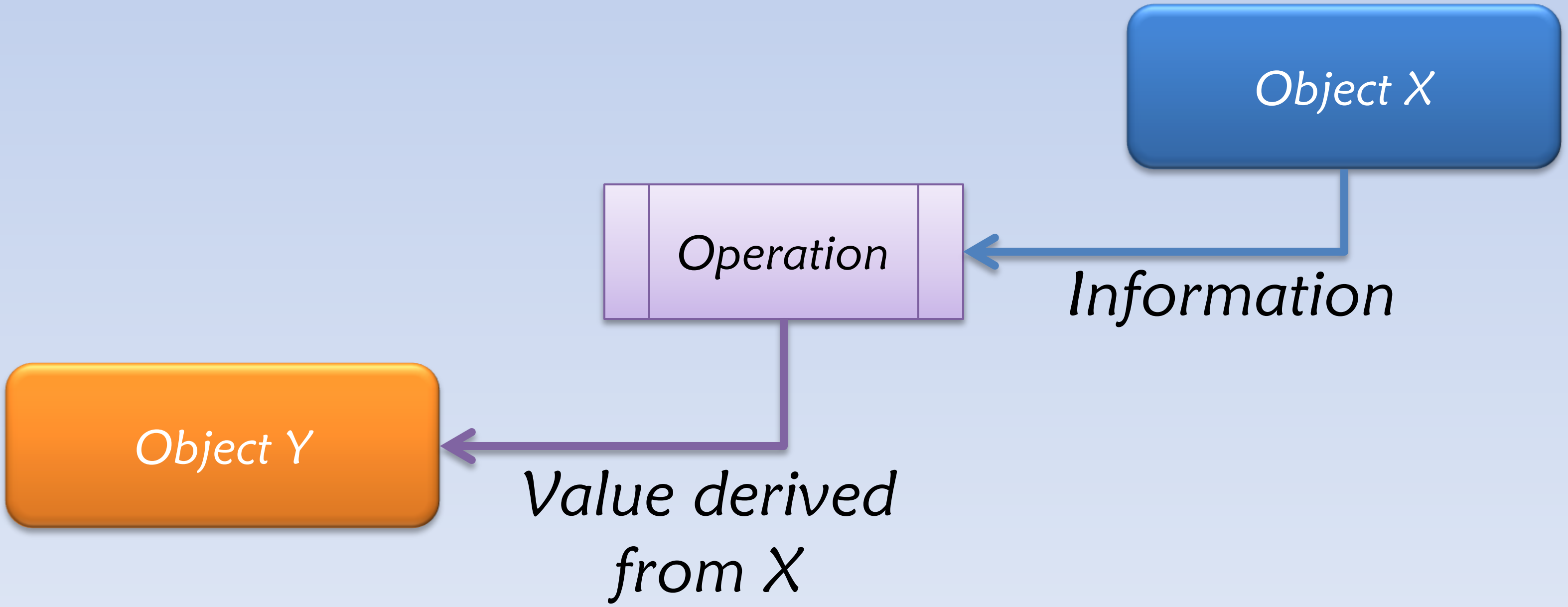
CONCEPTS

Information flow

- Follow any application inside a debugger and you'll see that data information is being copied and modified all the time. In another words, *information is always moving.*
- Taint analysis can be seen as a form of Information Flow Analysis.
- Great definition provided by Dorothy Denning at the paper "*Certification of programs for secure information flow*":
  - *"Information* ***flows*** *from object x to object y, denoted x→y , whenever information stored in x is transferred to, object y."*

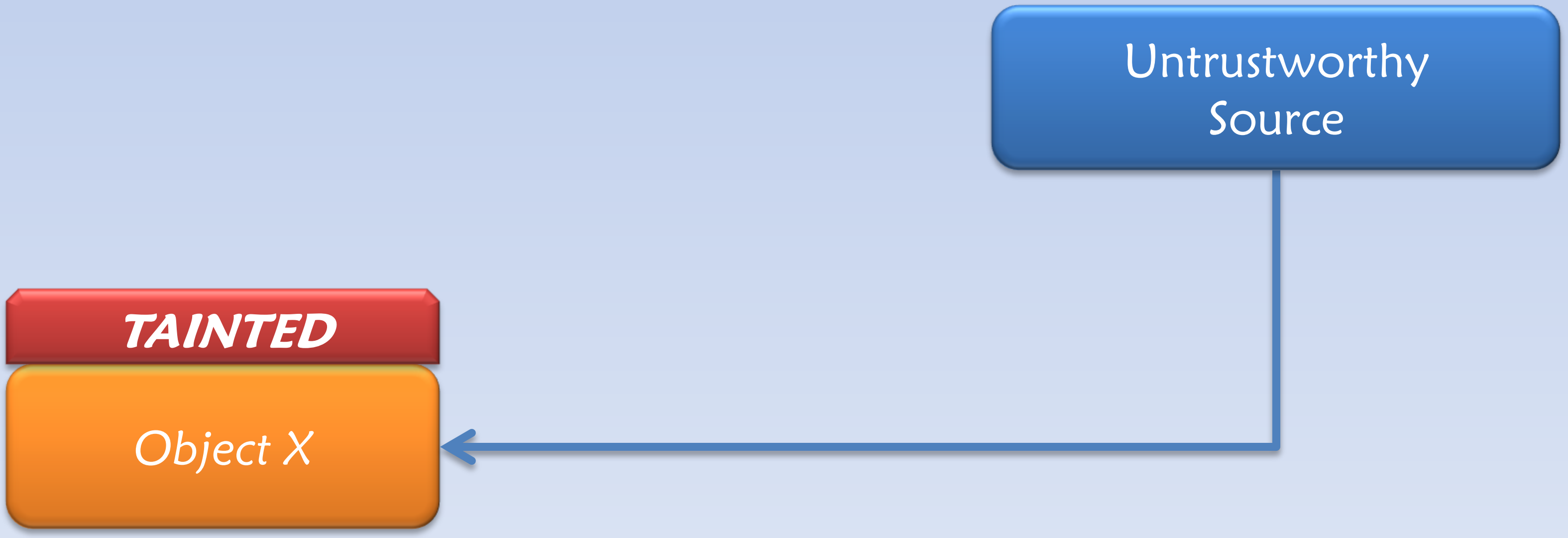
Flow

- *"An operation, or series of operations, that uses the value of some object, say x, to derive a value for another, say y, causes a* ***flow*** *from x to y."* [1]



Tainted objects

- If the **source** of the value of the object *X* is **untrustworthy**, we say that *X* is **tainted**.



Taint

- To “taint” user data is to insert some kind of tag or **label** for each object of the user data.
- The tag allow us to track the influence of the tainted object along the execution of the program.

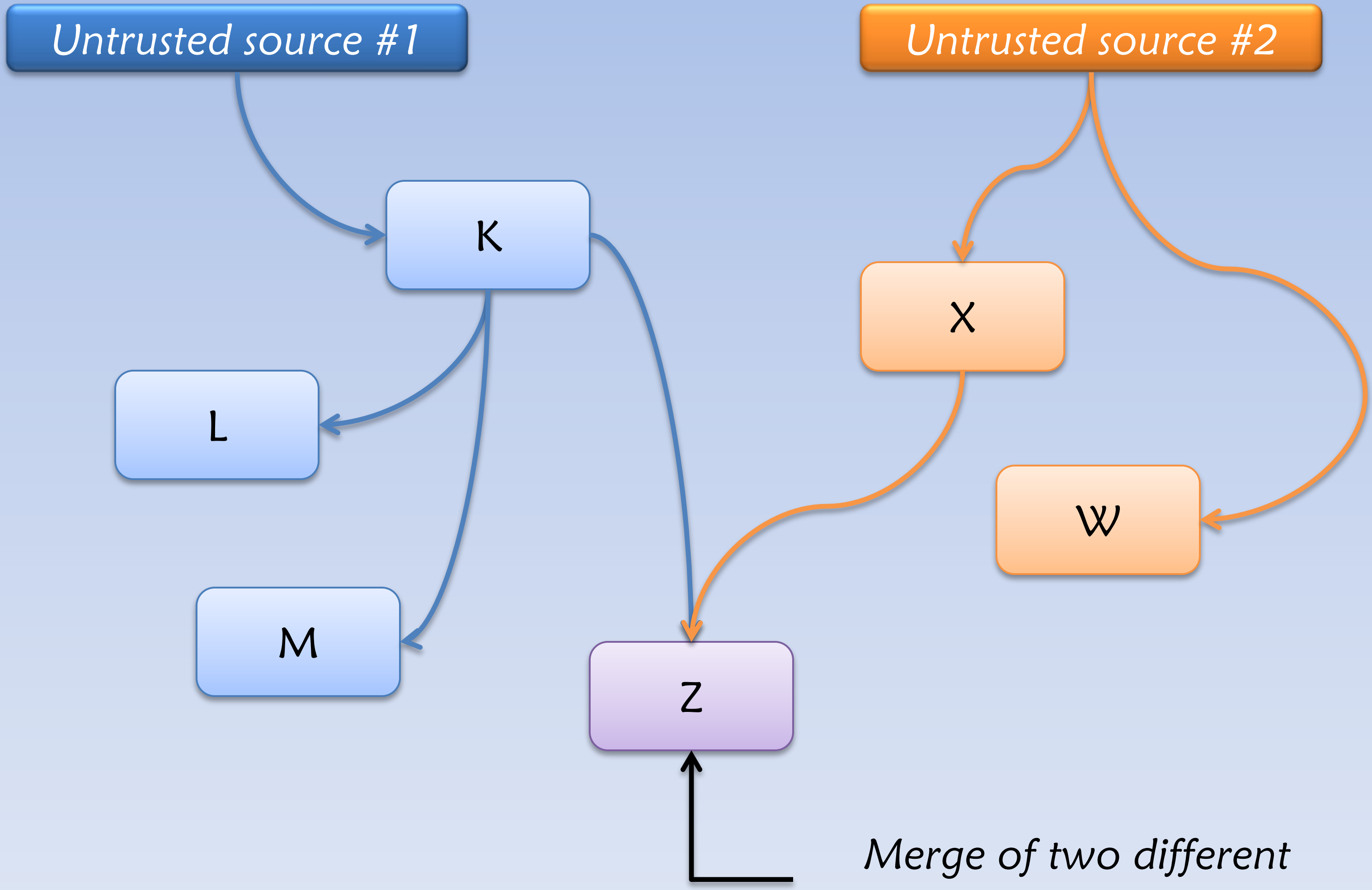
Taint sources

- Files (*.mp3, *.pdf, *.svg, *.html, *.js, …)
- Network protocols (HTTP, UDP, DNS, ...)
- Keyboard, mouse and touchscreen input messages
- Webcam
- USB
- Virtual machines (Vmware images)

Taint propagation

- If an operation uses the value of some **tainted** object, say X, to derive a value for another, say Y, then object Y becomes **tainted.** Object X tainted the object Y
- Taint operator ***t***
- ***X*** → ***t(Y)***
- Taint operator is transitive
  - X → t(Y) and Y → t(Z), then X → t(Z)

Taint propagation



Merge of two different tainted sources

Applications

- Exploit detection
  - If we can track user data, we can detect if non-trusted data reaches a privileged location
  - SQL injection, buffer overflows, XSS, …
  - Perl tainted mode
  - Detects **even** unknown attacks!
  - Taint analysis for web applications
- Before execution of any statement, the taint analysis module checks if the statement is tainted or not! If tainted issue an attack alert!

Applications

- Data Lifetime analysis
  - Jin Chow – “Understanding data lifetime via whole system emulation” – presented at Usenix’04.
  - Created a modified Bochs (TaintBochs) emulator to taint sensitive data.
  - Keep track of the lifetime of sensitive data (passwords, pin numbers, credit card numbers) stored in the virtual machine memory
  - Tracks data even in the kernel mode.
  - Concluded that most applications doesn’t have any measure to minimize the lifetime of the sensitive data in the memory.

Taint Analysis

TAINT ANALYSIS ON THE X86 ARCHITECTURE

Languages

- There are taint analysis tools for C, C++ and Java programming languages.
- In this presentation we will focus on tainted analysis for the x86 assembly language.
- The advantages are to not need the source code of applications and to avoid to create a parser for each available high-level language.

x86 instructions

- A taint analysis module for the x86 architecture must at least:
  - Identify all the operands of each instruction
  - Identify the type of operand (source/destination)
  - Track each tainted object
  - Understand the **semantics** of each instruction

x86 instructions

- A typical instruction like **mov eax, 040h** has 2 **explicit** operands like eax and the immediate value 040h.
- The destination operand:
  - eax
- The source operands are:
  - eax (register)
  - 040h (immediate value)
- Some instructions have **implicit** operands

x86 instructions

- **PUSH EAX**
- Explicit operand → EAX
- Semantics:
  - `ESP←ESP-4` (subtraction operation)
  - `SS:[ESP]→EAX` (move operation)
- Implicit operands → ESP register
  → SS segment register
- How to deal with implicit operands or complex instructions?

Intermediate languages

- **Translate** the x86 instructions into an Intermediate language!
- VEX language → Valgrind
- VINE IL → BitBlaze project
- REIL →Zynamics BinNavi

Intermediate languages

- With an intermediate language it becomes much more easy to parse and identify the operands.
- Example:
  - REIL → Uses only 17 instructions!
  - For more info about REIL, see Sebastian Porst presentation today
  - sample:
    - `1006E4B00: str edi, , edi`
    - `1006E4D00: sub esp, 4, esp`
    - `1006E4D01: and esp, 4294967295, esp`

Taint Analysis

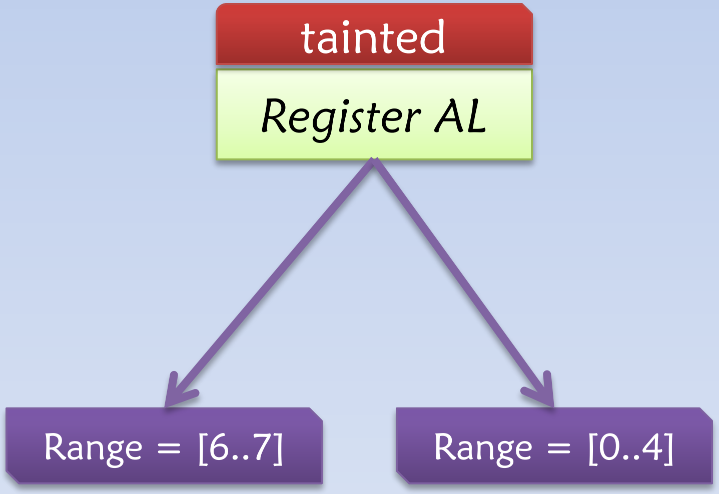
TAINT OBJECTS AND INSTRUCTIONS

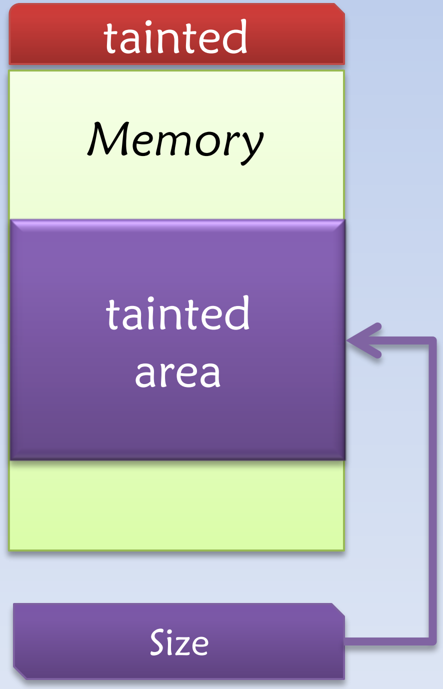
Taint objects

- In the x86 architecture we have 2 possible objects to taint:
  1. Memory locations
  2. Processor registers
- Memory objects:
  - Keep track of the initial address of the memory area
  - Keep track of the area size
- Register objects:
  - Keep track of the register identifier (name)
  - Keep a bit-level track of each bit

Taint objects

- The tainted objects representation presented here keeps track of each **bit**.
- Some tools uses a **byte**-level tracking mechanism (Valgrind TaintChecker)



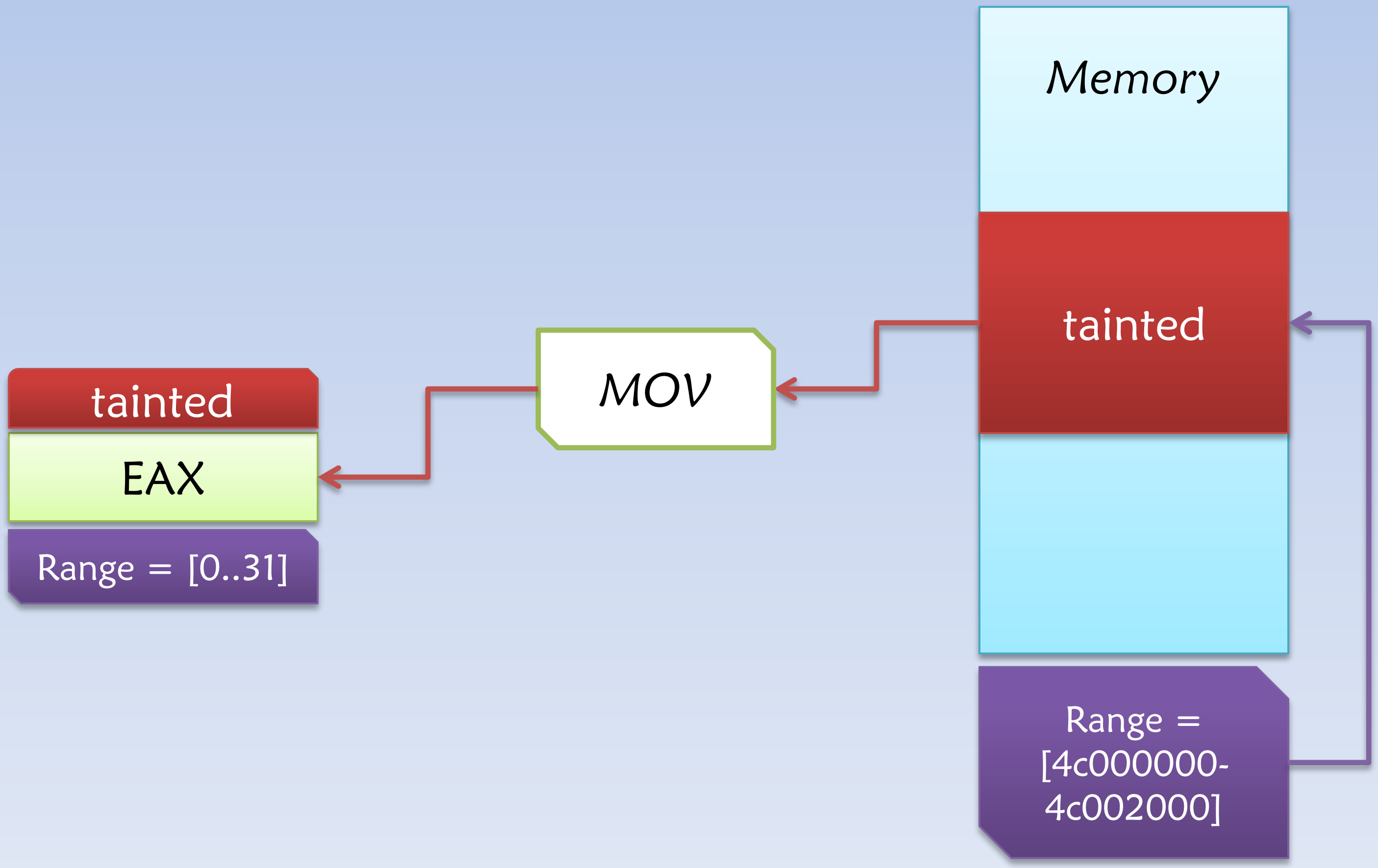


Instruction analysis

- The ISA (Instruction Set Architecture) of any platform can be divided in several categories:
  - Assignment instructions (load/store → mov, xchg, …)
  - Boolean instructions
  - Arithmetical instructions (add, sub, mul, div,…)
  - String instructions (rep movsb, rep scasb, …)
  - Branch instructions (call, jmp, jnz, ret, iret,…)

Assignment instructions

- *mov eax, dword ptr [4C001000h]*



Boolean

- Taint analysis of the most common boolean operators.
  - AND
  - OR
  - XOR
- The analysis must consider if the result of the boolean operator depends on the value of the tainted input.
- Special care must be take in the case of both inputs to be the **same** tainted object.

Boolean operators

- AND truth table

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

- If A is tainted
  - And B is equal 0, then the result is **UNTAINTED** because the result doesn't depends on the value of A.
  - And B is equal 1, then the result is **TAINTED** because A can control the result of the operation.

Boolean operators

- OR truth table

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

- If A is tainted
  - And B is equal 1, then the result is **UNTAINTED** because the result doesn't depends on the value of A.
  - And B is equal 0, then the result is **TAINTED** because A can control the result of the operation.

Boolean operators

- OR truth table

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

- If A is tainted
  - And B is equal 1, then the result is **UNTAINTED** because the result doesn't depends on the value of A.
  - And B is equal 0, then the result is **TAINTED** because A can control the result of the operation.

Boolean operators

- XOR truth table

A	**B**	**A xor B**
0	0	0
0	1	1
1	0	1
1	1	0

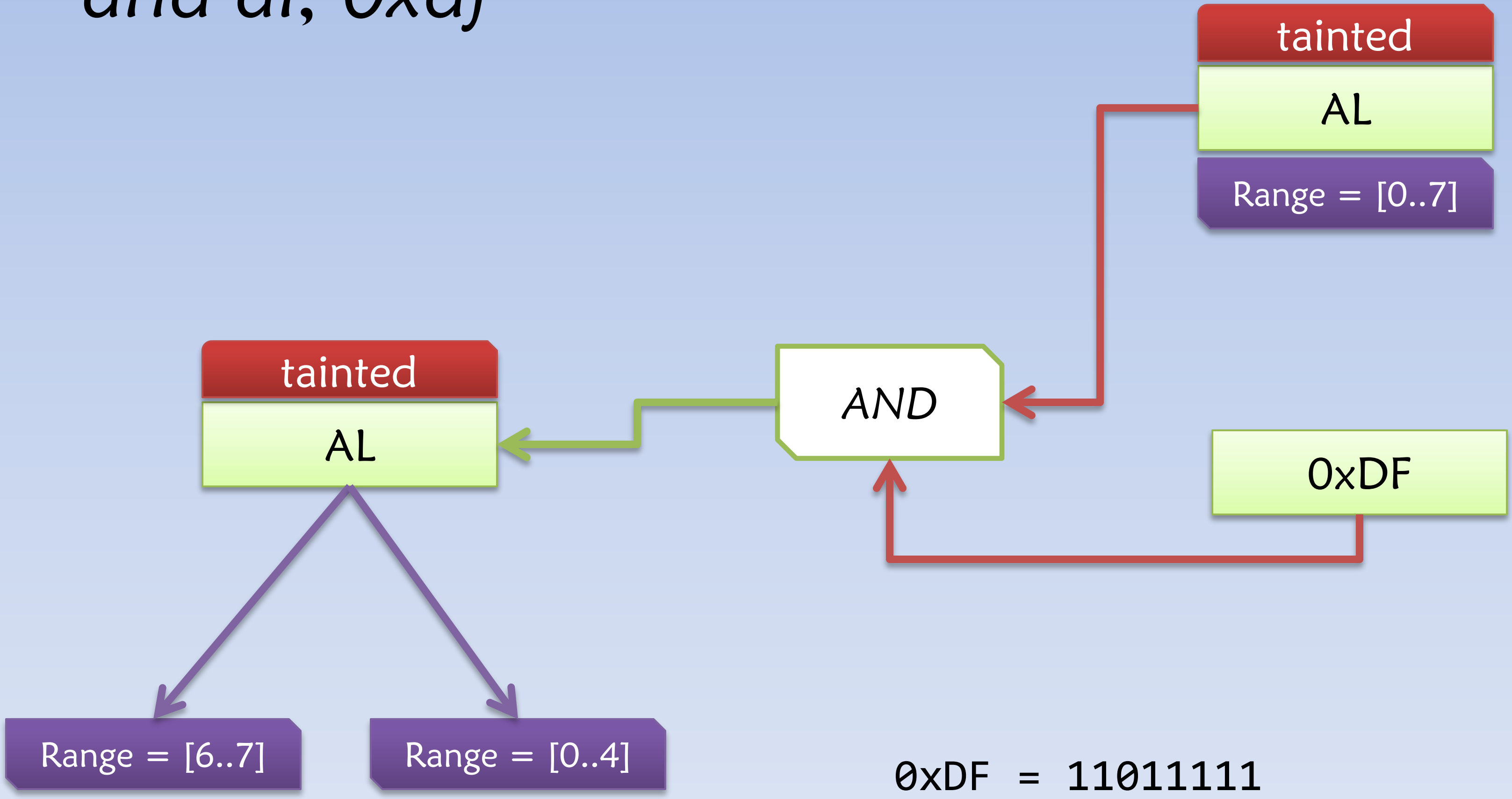
- If A is tainted,then **all** possible results are **TAINTED** indepently of any value of B.
- Special case → A XOR A

Boolean operators

- For the **tautology** and **contradiction** truth tables the result is always **UNTAINTED** because none of the inputs can can influentiate the result.
- In general operations which always results on *constant* values produces untainted objects.

Boolean operators

- *and al, 0xdf*

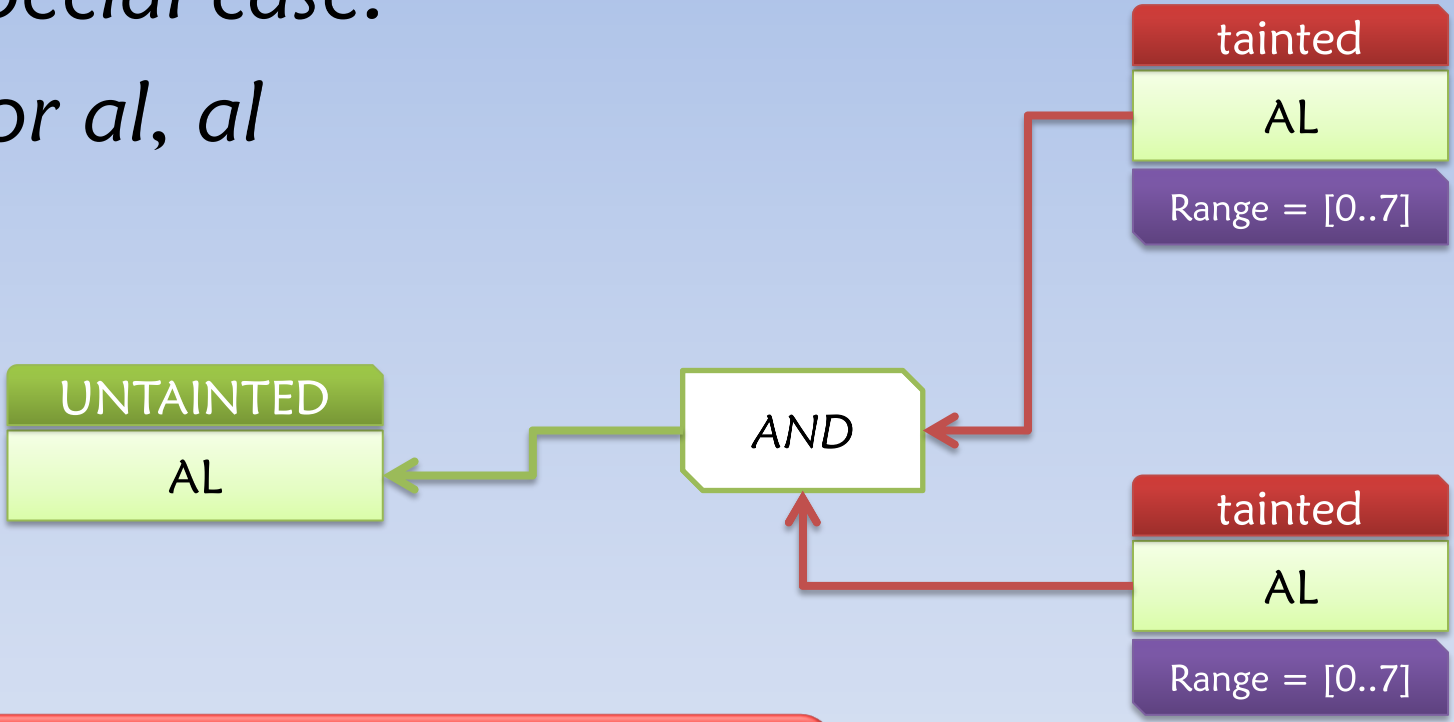


0xDF = 11011111

Boolean operators

- *Special case:*

  xor al, al



A XOR A → 0 (constant)

Arithmetical instructions

- add, sub, div, mul, idiv, imul, inc, dec
- All arithmetical instructions can be expressed using boolean operations.
- ADD expressed using only AND and XOR operators.
- Generally if one of the operands of an arithmetical operation is tainted, the result is also tainted.
- The affected flags in the EFLAGS register are also tainted.

String instructions

- Strings are just a linear array of characters.
- x86 string instructions – scas, lods, cmps, …
- As a general rule any string instruction applied to a tainted string results in a tainted object.
- String operations used to:
  - calculate the string size → Tainted
  - search for some specific char and set a flag if found/not found → Tainted

Lifetime of a tainted object

- Creation:
  - Assignment from an unstruted object
    - mov eax, userbuffer[ecx]
  - Assignment from a tainted object
    - add eax, eax
- Deletion:
  - Assignment from an untainted object
    - mov eax, 030h
  - Assignment from a tainted object which results in a constant value.
    - xor eax, eax

Taint Analysis

ADVANCED TAINTING

Level of details

- Some taint-based tools does not taint every object which is affected by a tainted object.
- For example, TaintBochs doesn`t taint comparison flags (eflags zf, cf, of,…). Others taint at a byte-level.
- This sometimes provides easy ways to bypass these tools.
- This section deals with more ‘agressive’ taint methods.

Optional taint objects

- Bit-level tracking instead of a byte-level.
- Conditional branch instructions tainting the EIP register and all the flag affect in the eflags register.
- Taint the code execution **time**.
- Taint at the code-block level of a control flow graph (CFG).

Comparison instructions

- x86 instructions → cmp, test
- *CMP EAX, 020h*

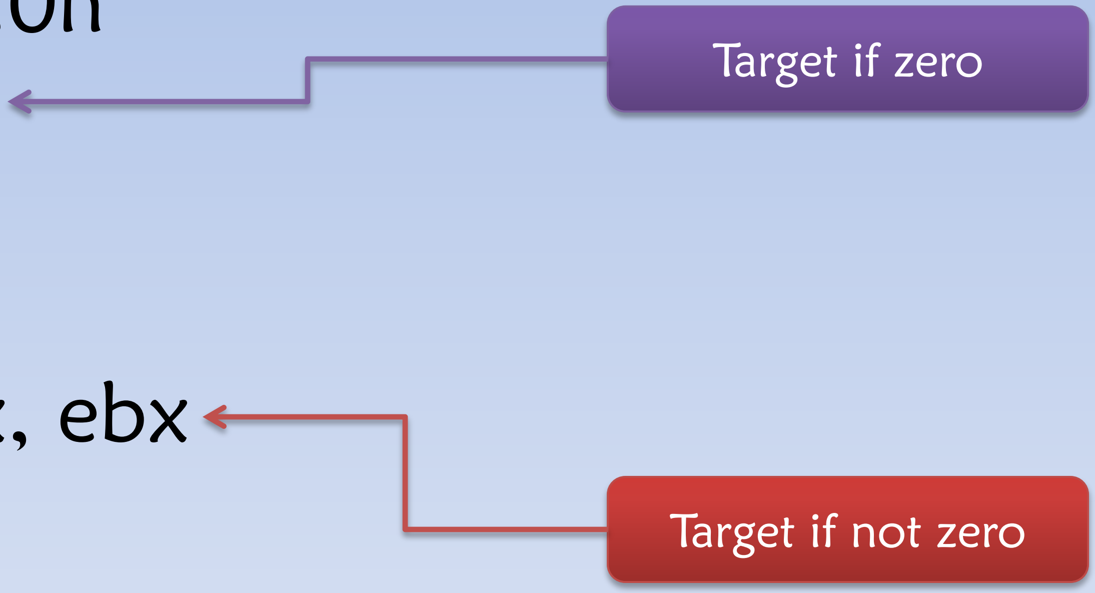
  pseudo-code:

  ```
      temp = eax – 20h
      set_eflags(temp)
  ```

- Lots of flags (Carry, Zero, Parity, Overflow,…)

Conditional branch instructions

-
  ```
  0100h: cmp eax, 020h
  0108h: jnz 0120h
  010dh: inc eax
  …
  …
  0120h: xor ebx, ebx
  ```



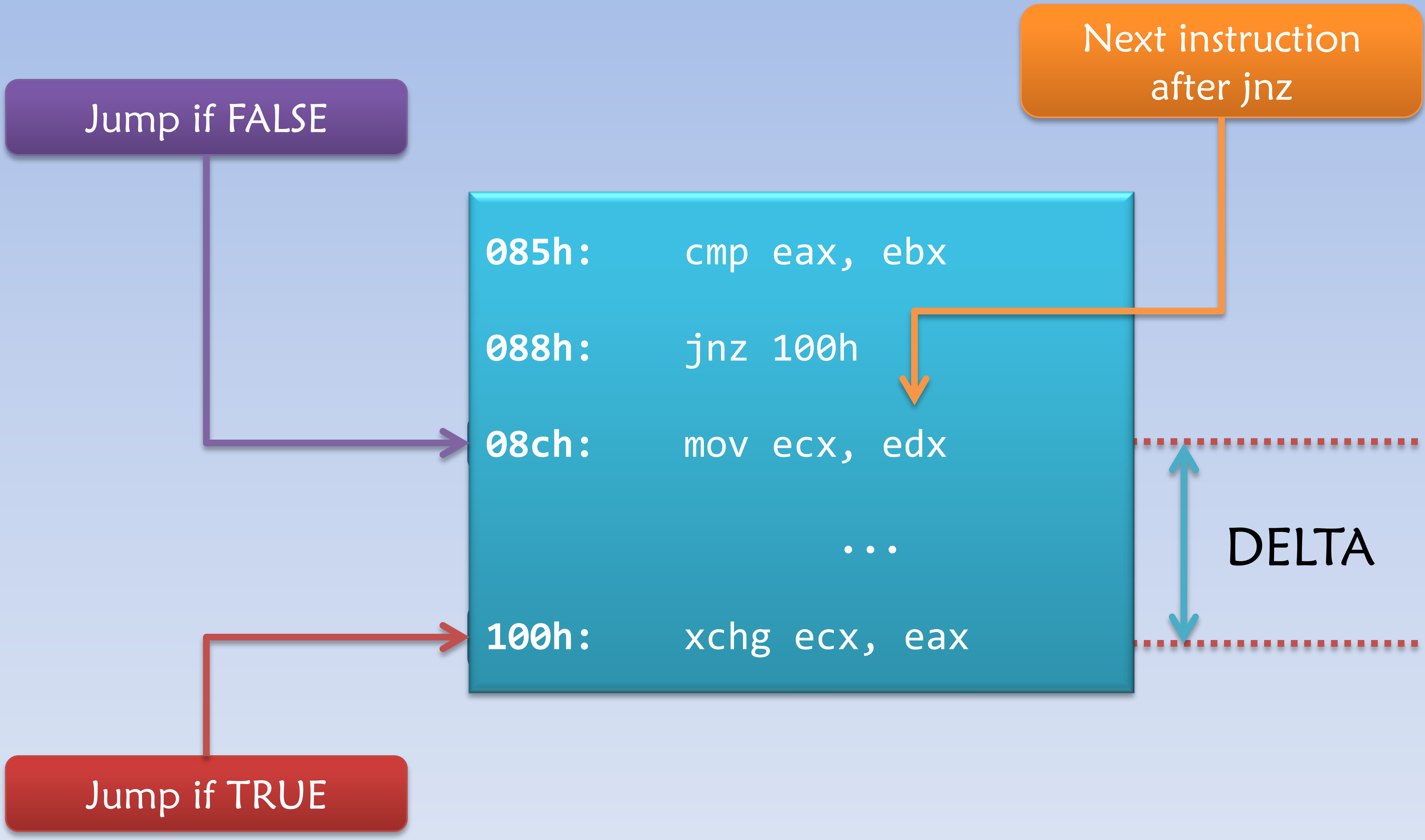
Target if zero

Target if not zero

Conditional branch instructions

- We already taint comparison flags like the Zero Flag.
- Branch instructions affects the EIP register.
- If a jump is dependent of the flag value, then the EIP must be **tainted**.
- How to express in a intermediate language the conditional jump to show relationship between the EIP and the ZF?

Tainted EIP



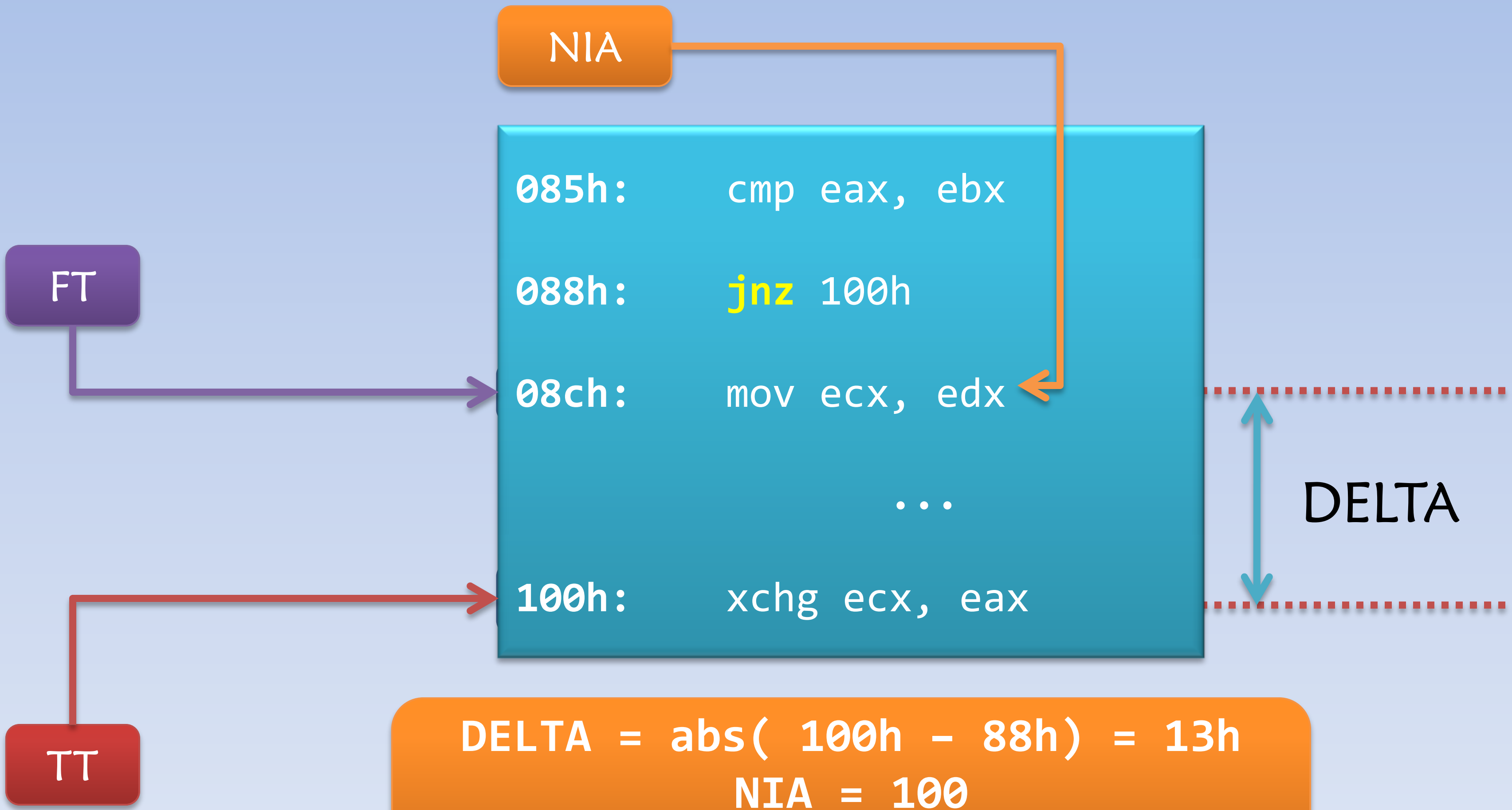
```
085h:     cmp eax, ebx
088h:     jnz 100h
08ch:     mov ecx, edx
                 ...
100h:     xchg ecx, eax
```

Formula for conditional jumps

- NIA → Next instruction **address** after the conditional jump
- TT → True Target (address of the target address if comparison is evaluated to TRUE)
- FT → Jump If False Target (008Ch)
- B → Flag value (always Boolean)
- D → Delta = abs (JITT - JIFT)

- We can now express EIP: ***EIP = NIA + BD***

Tainted EIP



```
085h:     cmp eax, ebx
088h:     jnz 100h
08ch:     mov ecx, edx
                   ...
100h:     xchg ecx, eax
```

DELTA = abs(100h – 88h) = 13h
NIA = 100
EIP → 8Ch + ZF * 13h

Tainted EIP

- What is the consequence of Tainted(EIP) = TRUE?
- The target code blocks of the Control Flow Graph are TAINTED!
- We can also use taint analysis to solve reachability problems!
  - Can I create a mp3 file which will make Winamp to execute the code block #357 of the function playSound()?

Full control

- A tainted EIP is not SUFFICIENT condition to define a vulnerability. It is necessary that the contents of the memory pointed by EIP to also be tainted:
- IF IsVulnerable() = TRUE then

```
    (IsTainted(EIP)     = TRUE)
                  AND
     (IsTainted(*EIP) = TRUE)
```

Algorithmic complexity attacks

- First published by Scott Crosby and Dan Wallach from Rice University at USENIX
- “Denial of Service via Algorithmic Complexity Attacks”
- Based on the fact that lots of algorithms have the worst-case.
- Creates special input which will direct the execution of the algorithm to the worst-case performance eventually causing a Denial of Service (DoS) attack.

Algorithmic complexity analysis

- If the time of the execution of a hash-algorithm depends of unstrusted data, then we can also **taint time!**
- *Tainted Time Analysis* (TTA) is generally more complex due to the use of more advanced mathematical analysis methods normally found on books about Analysis of algorithms.
- I applied a very basic TTA to detect the BluePill rootkit presented at SyScan`07.
- OS X Kernel Mach-O File Loading Denial of Service Vulnerability
  - http://www.securityfocus.com/bid/13222

QUESTIONS?

Thank you for allowing me to **taint** your precious **time**! ☺

References

1. “Certification of programs for secure information flow” – Dorothy E. Denning and Peter J. Denning. 1977 – Communication of the ACM
2. “A lattice model for secure information flow” – Dorothy E. Denning – 1976 – Communication of the ACM.
3. “Dytan: A generic dynamic taint analysis framework” – James Clause, Wanchun Li, and Alessandro Orso. Georgia Institute of Technology.
4. “Understanding data lifetime via whole system emulation” – Jim Chow, Tal Garfinkel, Kevi Christopher, Mendel Rosenblum – USENIX – Stanford University
5. “LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks” - Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan zhou, Youfeng Wu - University of Illinois at Urbana-Champaign
6. “BitBlaze: A New Approach to Computer Security via Binary Analysis” - Dawn Song
7. “Denial of Service via Algorithmic Complexity Attacks” - Scott A. Crosby

Taint Analysis

EXTRA SLIDES

Dynamic taint analysis

- To implement a Dynamic taint analysis tools it is necessary:
  - Debuggers → Paimei is great for prototypes. Necessary to insert breakpoint at the functions which provides interface to tainted sources like fopen, fread, …
  - For performance the amazing Rafal`s UMSS available at Avert Labs. It is around 100x faster than any debugger. Lazy evaluation of the affected flags of the eflags register also helps a lot.
  - Tainted objects tracking – tree/graph algorithms.