Dynamic Diluted Taint Analysis for Evaluating Detected Policy Violations

Maksim Bakulin

ISP RAS

Dynamic taint analysis

3 set of rules:

1. When to mark data as tainted.
2. How to propagate tainted data during the execution
3. When to raise a warning

Existing dynamic taint analysis systems

- ARGOS - manual instrumentation. Guest system is used as a honeypot.
- DECAF - TCG-level instrumentation.
- PANDA - Intermediate and helper code is translated to LLVM bitcode and the result is instrumented.
- TaintBosch

Typical causes of under- and overtainting

- Address dependencies
- Control flow dependencies
- One-way functions
- Incorrect taint propagation for some instructions

Fixed overtaint and undertaint causes in DECAF

1. DMA transactions
2. SYSENTER, SYSEXIT, IRETD instructions
3. SETcc
4. SBB Reg, Reg
5. OR and AND instructions (Fixed in newer version of DECAF)
6. Data transfer through floating point registers. TODO: XMM registers

Even after the fixes, warnings for benign files were sometimes triggered

Symbolic execution?

Idea: build path predicate during the execution, combine with security predicate and use solver to perform checks when symbolic data reaches EIP or is used as code

+ More detailed than dynamic taint analysis
+ Potentially lower number of false positives
- Much slower
- Solver is not always capable of providing the solution

Diluted taint: middle ground between regular taint and symbolic execution

- Each tainted byte contains a value, that represents its threat
- Data straight from the source has the highest value (0xFF)
- Each operation between tainted and untainted data results in a lower taint value. This reflects the idea that an attacker has less control over the result
- When a warning is raised, taint value is used to estimate the potential threat of a found detection
- Lower values (0x01) are used for results, that are affected by input, but do not possess major threat, e.g. sign-extended higher parts or shift operations, when shift count is tainted

Dilute function

$$dilute(\bar{a}, \bar{b}) := \begin{cases} \bar{a} \text{ if } \bar{a} = \bar{b} \\ \max(\bar{a}, \bar{b}) - 1 \text{ if } \bar{a} > 1 \text{ or } \bar{b} > 1 \\ 1 \text{ otherwise} \end{cases}$$

1. If taint values are equal, attacker has equal control over the variables, taint value is preserved.
2. If one value is more tainted than the other, the maximum is decreased by one
3. If taint value is already 1, it is not decreased further

Diluted taint example

Trace	**Taint of AL after instruction**
`MOV EAX, DWORD PTR[X]`	0xFF
`ADD EAX, 10`	0xFE
`SUB EAX, EBX`	0xFD
`MOV DWORD PTR[Y], EAX`	0xFD
`XOR EAX, EAX`	0
`...`	
`MOV EBX, DRORD PTR[Y]`	0
`...`	
`MOV EAX, EBX`	0xFD

Detection results

Application	Behaviour	Max EIP taint	Max instruction taint
Notepad++	calc.exe startup	0xFF	0xFF
AllPlayer	calc.exe startup	0 (0xFF*)	0 (0xFF*)
Calavera	calc.exe startup	0xFF	0xFF
AlReader	calc.exe startup	0xFF	0xFF
Adobe 9	Application crash	0xFF	0xFF
Adobe 11 benign 1	Normal execution	0xd0	0xd0
Adobe 11 benign 2	Normal execution	0x01	0x01

* After manual fix of undertainting due to address dependencies

Slowdown

Execution mode	Taint Files?	XP, s	PNG, s	x264, s
Native	-	-	7	22
QEMU	-	43	68	508
Regular taint	-	124	423	2259
	+	-	435	2393
Diluted taint	-	150	503	2824
	+	-	558	3802

Problems

- Not applicable to malicious programs
- Byte-level granularity causes problems with shift operations
- One instruction can cause stronger taint decrease than intended due to TCG-level instrumentation.

Future work

- Detecting sensitive data leaks?
- More penalty for complex operations (multiplication, division, floating point)
- Applying the concept to address dependencies: use the highest bit as flag to show the source of the taint
- Implementing (diluted) taint propagation through XMM registers and for x86-64 guest

Questions?