Using dynamic binary analysis for tracking pointer data

John Galea* and Mark Vella

University of Malta
Department of Computer Science

1 Introduction

The examination and monitoring of binaries during runtime, referred to as dynamic binary analysis, is a widely adopted approach, especially in the field of security and software vulnerabilities. Fundamentally, it provides one with a means to understand and reason about binary executions. There are various applications of dynamic binary analysis, including vulnerability analysis, malware analysis, and Web security.

One technique typically employed to perform dynamic analysis is taint analysis, which revolves around inspecting interesting information flows [3]. In this approach, taint marks are associated with values that are (1) introduced via defined sources and (2) propagated to other values to keep track of information flow. Marks may also be removed (untainted) once a defined sink has been reached. In addition, taint checking is also carried out in order to determine whether or not certain runtime behaviours of the program occur. The properties describing how taint analysis is performed, i.e taint introduction, propagation and checking, are specified by a set of rules referred to as a taint policy. One convenient way to define taint rules is in the form of operational semantics rules, as it avoids ambiguity issues. Rule 1 specifies the general form of a taint rule used in this paper. Given the current machine context of the program $\triangle$ and a statement, the rule specifies the end result, after the computation has been carried out.

$$\frac{\text{computation}}{\langle\triangle\rangle\langle\text{taintstate}\rangle\text{instruction} \leadsto \langle\text{taintstate'}\rangle}$$

Rule 1: General Rule

2 A taint policy for UAF vulnerability detection

The main aim of the overall research is to detect Use-After-Free (UAF) vulnerabilities, which are caused by dereferences of dangling pointers (pointers that point to freed memory). Based on the work carried out by J. Caballero et al. [2], this work focuses on using taint analysis to monitor flows that introduce, move and delete pointers referring to heap memory. Ongoing contributions include defining a clear taint policy for tracking heap pointers and implementing an online prototype for detecting UAF vulnerabilities.

As an example, consider the taint introduction rule (rule 2). It defines the taint source as calls to heap memory allocation functions. The pointer m, which stores the returned

* The research work disclosed in this publication is partially funded by the *Master it!* Scholarship Scheme (Malta).

value v of the function call, is associated with a taint label structure t_2 that holds information about the allocated memory. The link between the pointer and the taint label are established by making use of two global maps, namely a forward map τ and a reverse map π. More specifically, τ is responsible for mapping pointers to their respective taint labels, whilst π maps taint labels to a list of shared pointers, which refer to the same memory block. The benefit of the latter map is that a list of pointers associated with the same taint label can be efficiently retrieved without the need to iterate over the forward map.

$$\frac{\triangle, \tau \vdash ret \Downarrow \langle m, t_1 \rangle \quad \pi'' = \pi[t_1 \hookrightarrow m] \qquad t_2{=}\langle Ptr{,}v{,}\triangle(pc)\rangle \quad \tau'{=}\tau[m{\leftarrow}t_2] \quad \pi'{=}\pi''[t_2{\hookleftarrow}m]}{\triangle,\, \tau,\, \pi,\, alloc\_call(dst{,}v{,}ret) \;\rightsquigarrow\; \tau',\, \pi'}$$

Rule 2 - Pointer Introduction Rule

Taint labels propagate upon the execution of mov instructions, as defined in the rule 3. The computation of the rule mainly involves associating the taint label t_2 mapped by the source address m_2 with the destination address m_1.

$$\frac{\triangle, \tau \vdash dst \Downarrow \langle m_1, t_1 \rangle \quad \pi'' = \pi[t_1 \hookrightarrow m_1] \qquad \triangle{,}\tau \vdash src \Downarrow \langle m_2{,}t_2 \rangle \quad \tau'{=}\tau[m_1{\leftarrow}t_2] \quad \pi'{=}\pi''[t_2{\hookleftarrow}m_1]}{\triangle,\, \tau,\, \pi,\, mov(dst{,}src) \;\rightsquigarrow\; \tau',\, \pi'}$$

Rule 3 - Move Propagation Rule

3 Implementation using Dynamic Binary Translation

A prototype that implements the taint policy for pointer tracking has been built as a client using the DynamoRIO tool [1]. Since instrumentation on binary instructions is required to carry out taint analysis during runtime, the DynamoRIO tool [1] was found to be ideal, as it allows one to monitor, as well as manipulate, executed binary instructions without the need to modify the code of the application under examination. In essence, the DynamoRIO client acts as an intermediary and is placed between the application and the underlying operation system. By copying an application's original instructions to a code cache one block at a time, changes to instructions can be easily conducted. The DynamoRIO prototype monitors for calls to memory allocation and deallocation functions. Moreover, instructions such as mov and $push$ are also examined in order to carry out taint propagation as described in rule 3.

References

1. Dynamorio - dynamic instrumentation tool platform. http://dynamorio.org. [Online; accessed 3-Oct-2014].
2. J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012.
3. T. A. Edward J. Schwartz and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy (SP '10)*, 2010.