LETTER

On the Deployment of Dynamic Taint Analysis for Application Communities*

Hyung Chan KIM[†a)], ***Member and*** **Angelos KEROMYTIS**[†], ***Nonmember***

SUMMARY Although software-attack detection via dynamic taint analysis (DTA) supports high coverage of program execution, it prohibitively degrades the performance of the monitored program. This letter explores the possibility of collaborative dynamic taint analysis among members of an application community (AC): instead of full monitoring for every request at every instance of the AC, each member uses DTA for some fraction of the incoming requests, thereby loosening the burden of heavyweight monitoring. Our experimental results using a test AC based on the Apache web server show that speedy detection of worm outbreaks is feasible with application communities of medium size (*i.e.,* 250–500).

key words: *Dynamic taint analysis, 0-day attack detection, application community, software security*

1. Introduction

Dynamic taint analysis (DTA) [3]–[5] is a technique for tracking information flow within a software application. DTA can be used to detect 0-day (previously unknown) attacks or information leakage. A DTA tool tracks information flow by supervising execution of program instructions. Such instrumentation granularity supports substantial coverage of program execution. However, it also means that the instrumented program can suffer prohibitive performance degradation.

Widely used homogeneous applications form *de facto* software monocultures [8]. This is viewed as detrimental to security, since a single vulnerability can kead to compromise of all instances of that application. However, such homogeneity offers the chance to distribute the workload of heavyweight security monitoring, such as DTA, to spot 0-day attacks and share the attack information among members of such "application communities" (AC) [9].

In this letter, we describe an implementation of a DTA tool and performance results of deploying the tool with the Apache web server. Our approach is based on loosening the request coverage of each AC member so that they can achieve acceptable performance. Our analysis shows that medium-sized ACs (250–500 members) can feasibly use our tool to form a community for collaborative defense.

Manuscript received January 1, 2008.
Manuscript revised January 1, 2008.
Final manuscript received January 1, 2008.

[†]The authors are with the the Department of Computer Science, Columbia University, NY 10027, USA.
a) E-mail: hckim@cs.columbia.edu

*This material is based on research sponsored in part by the Air Force Research Laboratory under agreement number FA8750-06-2-0221, NSF Grant 06-27473, with additional support from Google and Intel. The opinions expressed herein reflect those of the authors, and not of the U.S. Government. This work is supported in part by the Korea Research Foundation Grant funded by the Korean Government(MOEHRD) (KRF-2007-357-D00240).

2. SeeC: A Dynamic Taint Analysis Tool

This section describes our implementation of DTA tool.

2.1 Implementation

The goal of the basic architecture [Fig. 1] is to keep track of the association between a taint tag in shadow memory and memory/registers handled by program instructions. For example, if an instruction causes information to flow directly (*e.g.,* via memory copying) or indirectly (*e.g.,* as part of an arithmetic operation that uses a tainted memory location as an operand) from one memory location to the other, the DTA tool reflects the flow by marking taint tags to corresponding locations in the shadow memory.

Depending on the data source, specific information flow can be dealt with. Assuming that we track untrustworthy data and supervise the taint markings propagated with the information flow directed by program execution, while tracking taint tags, a detector can raise an alert if some tainted memory or registers are used in a certain execution context. For example, for *overwrite attack* detection, incoming network data or keyboard inputs can be initially marked and, if propagated markings are found in some memory place that corresponds to a target address of return or branch instructions, the detector raises a warning that the memory place is overwritten and the control of program execution is about to be subverted.

The implementation can be realized with two different view points; (1) whole system approach [6], [7] based on virtual machine monitors or system emulators, and (2) application program approach [4], [5] based on dynamic binary instrumentation (DBI) frameworks such as PIN [1] or Valgrind [2]. Our implementation of the DTA tool is based on PIN.

Our tool (SeeC) supports initial taint markings from network and file sources by hooking system calls and tagging incoming data. We instrument data

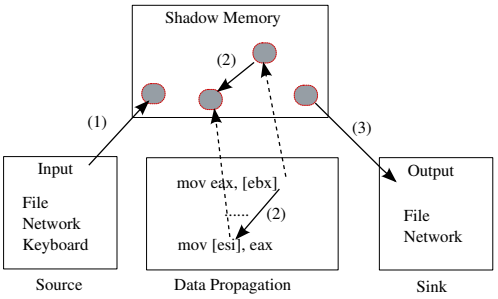

Fig. 1 Dynamic taint analysis architecture: (1) initial marking, (2) propagation, and (3) assertion

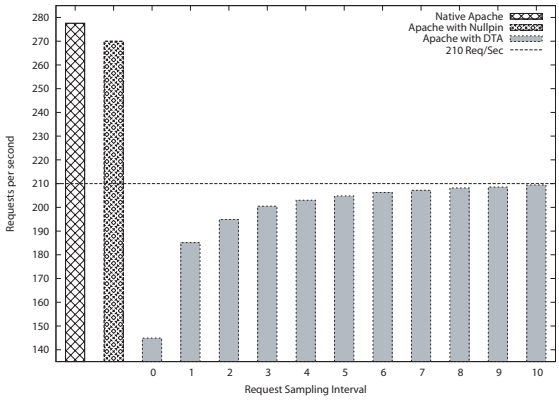

Fig. 2 Request throughput of a single process Apache under DTA monitoring

movement (*e.g., mov, push, pop*), string (*e.g., movs*) and arithmetic instructions (*e.g., add, sub*) for taint mark propagation according to the policy: if at least one source operand is tainted, then the destination operands should be also tainted. We also apply a clearance policy: if all inputs to an operation are clear, the destination operands are also cleared. The propagation policy includes implicit operands such as the *esp* register and memory indicated by that register when *push/pop* instructions are used. We check taint marks at branch/call instructions such as *ret, call*, and *jmp* to observe whether the memory or register containing the target location or the target itself is tainted.

Table 1 is an example basic block that demonstrates how SeeC performs taint propagations. In the table, $tag(x)$ is the taint tag value in shadow memory associated with native memory address or register x. $tag(constant)$ yields 0. Assuming that the register *ebp* is tainted in the initial context, data bytes in $ss[ebp]$ are also tainted by simple induction with the line 1 and 6. For other dependencies, if bytes in $ds[ebx+0x034e8]$ in line 2 or $gs[0x18]$ in line 5 is tainted, then eax is tainted in line 7. If data from those memory locations came from a network stream, the control flow decision at line 7 is unduly influenced from outside the process (or possibly the host).

In SeeC, each byte of application memory is mapped to 1 bit in shadow memory. In addition, we track the 8 general registers of the x86 architecture.

2.2 Attack Detection

To validate the ability of overwrite attack detection in SeeC, we have tested with a software vulnerability testbed [10] that enables us to check against 18 types of buffer overflow attacks targeting return address, base pointer, function pointer, and *longjmp* buffer. Our tool successfully blocks all the attacks in the testbed. For real-world applications, we have also tested with some vulnerable server applications shown in Table 2 and those attacks are also correctly detected with SeeC.

2.3 Performance of SeeC

To evaluate the performance overhead of our tool, we have tested with a CPU-bound application, `gzip` compressing 261MB of Linux kernel source code in a lightly loaded machine. The incurred overhead was approximately 28.11X with SeeC compared to 1.28X with Nullpin tool (that only instruments instructions and does nothing for analysis). For a desktop application, `FireFox` version 1.5 with SeeC took 38.8X overhead to render 792KB of randomly generated web pages [13].

3. Deployment of DTA tool for Application Communities

DTA has been considered very effective way to detect 0-day attacks and many recent works have illuminated the approach. However, it is basically relying on heavyweight instrumentation of program instructions; thus, it can incur application slowdown at least over two orders of magnitude without serious optimization (as shown in Section 2.3).

Our approach to mitigate the performance impact is to distribute heavyweight monitoring among members of an application community (AC). Specifically, we try to loosen temporal workload: each member does not fully monitor all the incoming requests. Instead, a member processes only a single request out of every r requests (request coverage: $1/r$). In this section, we present (1) experimental results of finding appropriate r at which the performance could be the best it can be, and (2) our preliminary analysis about the AC size with the given request coverage. For our prototype, we assume an AC consisting of instances of the Apache web server [19] version 1.3.31.

3.1 Performance Experience of DTA with Apache

Our performance observation is on the request throughput, that is, the number of requests that a single

Table 1 Example taint propagation in a basic block

	Instruction	Propagation
1	lea edx, ptr [ebp-0x78]	$tag(edx) = tag(ebp)$
2	mov eax, dword ptr ds[ebx+0x34e8]	$tag(eax) = tag(ds[ebx + 0x034e8])$
3	mov dword ptr ss[esp+0x4], 0x0	$tag(ss[esp + 0x4]) = tag(0x0)$
4	ror eax, 0x9	$tag(eax) = tag(eax)\|tag(0x9)$
5	xor eax, dword ptr gs[0x18]	$tag(eax) = tag(eax)\|tag(gs[0x18])$
6	mov dword ptr ss[esp], edx	$tag(ss[esp]) = tag(edx)$
7	call eax	

Table 2 Overwrite/taint-based attacks tested with SeeC

Application	Application Type	Attack Type	CVE/Bugtraq ID
Apache 1.3.31	Web server	Local buffer overflow	CVE-2004-0940 [14]
Atphttpd-0.4b	Web server	Remote buffer overflow	CVE-2002-1816 [16]
ShoutCast 1.9.4	MP3 media server	Remote format string	CVE-2004-1373 [15]
CoreHTTP 0.5.3 alpha	Web server	Remote buffer overflow	CVE-2007-4060 [17]
PHP Agenda	Web application	Input validation error	Bugtraq 30034 [18]

Apache process can handle per second, serving a 100KB HTML file. To overload HTTP requests, we use two httperf clients [20] in a Gigabit network environment. We could selectively process requests by turning on and off SeeC operation instrumenting `ap_read_request()` and `ap_process_request()` function calls in the main request processing loop of Apache. Figure 2 shows the throughput results. In the figure, request sampling interval means the number of requests skipped between two consecutive DTA processing sessions; thus, 0 means all requests are monitored by SeeC.

The apache with Nullpin tool incurs a relatively small overhead. The third bar in the figure shows that only half of requests can be processed with full DTA monitoring compared to the native case. As the request sampling interval increases, the throughput quickly converges to a constant rate (210 req/sec). After reaching a sampling interval 10, there is no further throughput improvement. Therefore, we conclude that processing a request out of every 10 or 11 requests represents the best performance/coverage tradeoff. Increasing the interval further will only drop request coverage without any throughput gain.

The possible best throughput (210 req/sec) was 75% of the native case. This is due to the flag-checking overhead to do selective DTA processing; all instrumented instructions check a boolean flag to determine whether they need to propagate taint marks or not. The ideal implementation is to move the on/off flag-checking at the function granularity instead of doing for every instructions; check on/off flag just before a function, and call native or Just-In-Time (JIT) instrumented version of the function accordingly. Unfortunately, the current version of PIN does not support dynamic mode change between JIT and native.

3.2 Analysis

Let f_i be the probability of an attack occurrence on a i-th member of a AC. This factor can be varied dependent on the position of monitoring in the network or the attack volume. For example, a rapidly spreading worm such as SQL Slammer may raise this value toward 1. The temporal attack coverage of an individual member is $f_i \times r_i$, where r_i is the temporal coverage rate of the i-th member. Thus, the probability of at least one of member detecting the attack is

$$P(AC) = \sum_{i=1}^{N} f_i \times r_i \tag{1}$$

To determine the appropriate size for the AC, we may set f_i from statistics of previous attack-spreading data or results from a simulation of worm propagation. For example, according to the simulation result of Zou *et al.* [11], 1% to 2% of the vulnerable population was infected in the slow start phase of the Code Red worm, when the infection rate is relatively low. Therefore, using that assumption, to prevent an unknown attack that has a similar spreading pattern we set f_i to be 2/100 for all members uniformly. All members configure SeeC to do DTA processing for one out of every 10 requests to achieve their best throughput, as shown in Section 3.1, thus r_i will be 1/10. In that case, an AC with 500 members will detect the attack within the slow-start phase with high probability.

Note that, in the above example, we assume that there is enough time to disseminate alert information among the AC members. In the simulation of Zou *et al.*, the time of Code Red's slow start phase of 2% population infection was 223 minutes, therefore, a small alert message (at around 1–2KB) could be shared in a few seconds through the 500 members, thereby reducing threats for remaining 98% population. For the reference, we performed a test of disseminating 4KB alert messages with 500 members in a centralized configuration (100bps) in Deterlab testbed [21]. It took 41 sec in average. Therefore, it would be sufficient to immunize against Code Red style worms within the AC.

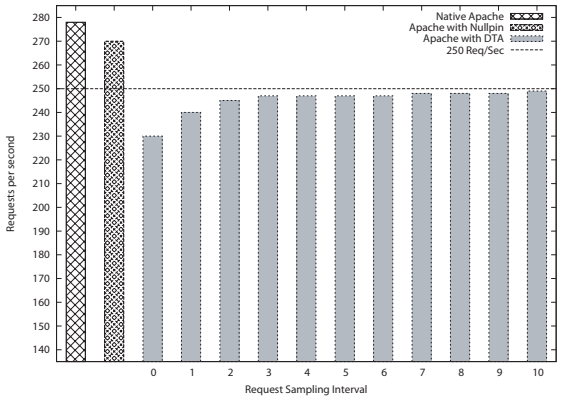

Fig. 3 Performance of a single process Apache under DTA monitoring, with taint path profiling information

Moreover, it would be also effective against Slammer style worms as the time for 1% population infection is 45 sec [11].

However, a hit-list worm [12] can infect a large portion of the population in a very short time, depending on hit-list entry size. Even though an initial member can detect the attack, much of the remaining population could face it without the alert report from the member because of the dissemination delay.

3.3 Further Improvement

From equation (1), it is evident that we can realize a smaller AC if members can increase the temporal coverage rate. In the experiment of Section 3.1, we identified the flag-checking overhead. As for possible performance improvements, we have tried *taint path profiling*. First, we run Apache process with SeeC in full monitoring. During the running, we collect information about functions involved in taint propagation: if any of the instructions of a function propagate taint, we include that function in the monitoring list. We fed several types of HTTP requests to trigger various taint paths. In selective monitoring, we only process functions in the concerned list. Figure 3 shows our improved throughput result. This time the best throughput is even more quickly converged to 250 req/sec, which is 90% of the native case, compared to 210 req/sec. With this improvement, we may increase temporal coverage rate to 1/5 thereby requiring a smaller AC.

4. Conclusions

We have presented an implementation of a dynamic taint analysis tool (SeeC) and experimental performance results focusing on the deployment of SeeC to a web server application community. We are encouraged by our results showing significant performance gains through DTA sampling, while achieving adequate (and tunable) detection speeds. We are now working toward implementing a full proof-of-concept application community: a framework to share attack information among AC members. Moreover, we will explore the analytical model on the relationship between worm propagation and alert dissemination in varying AC sizes.

References

[1] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), pp. 190–200, 2005.

[2] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," SIGPLAN Not., Vol. 42, No. 2, pp. 89–100, 2007.

[3] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," Proc. of the 12^{th} Symposium on Network and Distributed System Security (NDSS), 2005.

[4] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou and Y. Wu, "LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks," Proc. of the 39^{th} Annual IEEE/ACM International Symposium on Microarchitecture, pp. 135–148, 2006.

[5] J. Clause, W. Li and A. Orso, "Dytan: a generic dynamic taint analysis framework," Proc. of the International Symposium on Software Testing and Analysis, pp. 196–206, 2007.

[6] G. Portokalidis, A. Slowinska and H. Bos, "Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation," Proc. of the 1^{st} ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys), pp. 15–27, 2006.

[7] H. Yin, D. Song, M. Egele, C. Kruegel and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," Proc. of the 14^{th} ACM Conf. on Computer and Communications Security (CCS), pp. 116–127, 2007.

[8] G. Goth, "Addressing the monoculture," IEEE Security & Privacy, Vol. 1, No. 6, pp. 8–10, 2003.

[9] M. Locasto, S. Sidiroglou, and A. D. Keromytis, "Software Self-Healing Using Collaborative Application Communities," Proc. of the 13^{th} Symposium on Network and Distributed System Security (NDSS), pp. 95–106, 2006.

[10] J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention," Proc. of the 10^{th} Symposium on Network and Distributed System Security (NDSS) pp. 149–162, 2003.

[11] C. Zou, L. Gao, W. Gong, and D. Towsley, "Monitoring and early warning for internet worms," Proc. of the 10^{th} ACM Conf. on Computer and Communications Security (CCS), pp. 190–199, 2003.

[12] S. Staniford, V. Paxson, and N. Weaver, "How to own the internet in your spare time," Proc. of the 11^{th} USENIX Security Symposium, pp. 149–167, 2002.

[13] http://scragz.com/tech/mozilla/test-rendering-time.php

[14] http://nvd.nist.gov/nvd.cfm?cvename=CVE-2004-0940

[15] http://nvd.nist.gov/nvd.cfm?cvename=CVE-2004-1373

[16] http://nvd.nist.gov/nvd.cfm?cvename=CVE-2002-1816

[17] http://nvd.nist.gov/nvd.cfm?cvename=CVE-2007-4060

[18] http://www.securityfocus.com/bid/30034

[19] http://www.apache.org/

[20] http://www.hpl.hp.com/research/linux/httperf/

[21] http://www.isi.deterlab.net/