

Towards Seamless Prevention & Recovery from Application-Level Vulnerabilities

Benjamin Livshits*

Stanford University

livshits@cs.stanford.edu

Abstract

A great deal of research in the last several years has focused on securing sever-side software systems, which are a common target to buffer overruns, format string violations, and other similar types of attacks. A variety of techniques to protect server-side software have been suggested, ranging from hardware-level mechanisms [12, 36] to static analysis [23, 34, 37].

However, most of the code being written is *application software*, which has a totally different set of security requirements. In particular, application software is often written by programmers who not trained in security, suffer from a rushed development schedule, and exist in environment where features are emphasized over software quality. Automatic solutions such as static analysis may be too intrusive and out of place at organizations that lack a well-established development process. Moreover, rushed development schedules that favor features over robustness leave little time to ensure the security of the system, which is often considered “nice to have”, but not a necessity. As a result, it is often impractical to expect developers to take the responsibility for securing their applications. The reality of today’s application development suggests that fully automatic security solutions that relieve applications developers from the responsibility of securing their code are highly desirable.

In this paper we argue for a generalized approach to protecting application software from a variety of exploits with minimal programmer intervention.

1 Introduction

Buffer overruns and format string violations found in `wu-ftpd`, `1httpd`, Apache, and other widely deployed

software packages have motivated a great deal of research into techniques for detection and prevention of these types of vulnerabilities. A quick examination of vulnerability catalogs such as `securityfocus.com` reveals that most vulnerabilities reported these are contained in widely deployed software packages. However, there is no reason to believe that “custom-made” application software contains fewer vulnerabilities. Moreover, the recent shift from the client-server to the Web-based model for business applications has opened the door for a new breed of Web application vulnerabilities caused by insecure information flow. These vulnerabilities include SQL injections [1, 2, 13], cross-site scripting [7, 17, 35], path traversal attacks [28], etc.

While vulnerabilities in application software are rarely publicized, aggregate statistics are quite telling. According to a Gartner survey, 75% of all security attacks today are performed at the application level [20]. Moreover, 97% of more than 300 sites audited in the survey were vulnerable to Web application attacks. According to another survey done by the Computer Security Institute, the average financial loss from unauthorized access or information theft exceeds \$300,000 [8].

In this paper we argue that an automatic *runtime* approach is often superior to the most widely-used techniques for a range of common vulnerabilities.

2 Overview

We begin by giving an overview of the most common approaches to software security used today:

- **Testing for security.** There is been much interest in testing approaches to security problems, resulting in an increased commercial interest in penetration testing, both manual and automatic. These techniques rely on generating

*This work was supported by NSF Grant № 0326227.

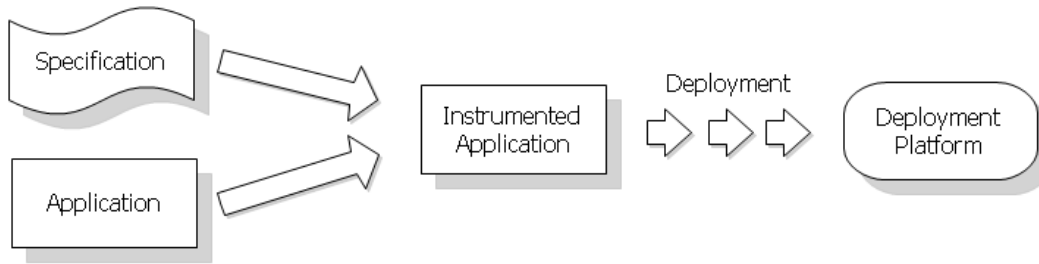


Figure 1: A typical binary rewriting architecture.

inputs in hopes of triggering a vulnerability or a fault within the application. While the problem of input generation to achieve complete path coverage is undecidable in general, even in practice these techniques are not very effective at finding vulnerabilities. This is not entirely surprising: a pretty rare event needs to be triggered in a black box fashion, i.e. without knowing *anything* about the application internals.

- **Code auditing.** While an effective approach overall, it is difficult to scale to programs containing hundreds of thousands lines of code. Moreover, it is nearly impossible to keep up with rapidly changing code bases that quickly have to go into production.
- **Static analysis tools.** Static analysis tools and techniques are more practical than code auditing in the long run. It has the additional benefit of finding more bugs than an approach that requires program execution because multiple runtime scenarios may be considered. Moreover, a *sound* static analysis approach can *guarantee* that the absence of warnings implies the absence of vulnerabilities. While an attractive approach in general, the adoption of static analysis techniques is hindered by the following:

Poor fit into the existing process. For static analysis tools to be widely adopted, they need to fit well into the existing development process. Large organizations have a set of established processes their developers tend to follow and a static analysis step fits naturally between development and testing.

However, smaller or more heterogeneous environments such as small IT companies and large diverse organizations do not have that in place. For instance, IT staff at a university is often responsible for supporting multiple applications written in languages ranging from Perl, Python, and Ruby to Java, C#, and ASP.NET. Even if

static analysis tools for security existed for each of these languages and platforms, proper integration would still be a challenge.

Soundness and precision is hard to scale.

While opinions on what sound static analysis is capable of may vary, most people would agree that it is difficult to scale precise and sound approaches to programs consisting more than several hundred thousand lines of code. Since analysis precision is paramount, existing scalable tools provide a best-effort attempt as finding the vulnerabilities and some may be missed.

As an alternative to compile-time sound tools, it makes sense to talk about *runtime-sound* approaches, i.e. whether a given dynamic approach detects *all* exploits all vulnerabilities of a particular kind at runtime. It is well-known that some solutions such as StackGuard [10] can be circumvented with trampolining, making StackGuard runtime-unsound. However, runtime soundness may be much easier to ensure in practice than compile-soundness and runtime solutions, while imposing an overhead, typically do not suffer from the lack of scalability.

3 Current Solutions

In this section we summarize the steps taken in the direction of seamless runtime protection and discuss the pros and cons of each of the approaches.

One way to separate existing techniques is to distinguish between approaches that use binary rewriting and approaches that rely on platform support.

3.1 Binary Rewriting for Security

Many projects have focused on rewriting techniques that address buffer overruns. Among them are program shepherding [21]

One of the first projects to focus on Web application vulnerabilities is WebSSARI [18].

[26] [4] [11] [5] [30]

System	Language	Rewriting-based?	Customizable?	Can recover?	Static component?	Runtime-sound?	Overhead (%)
perl - t	Perl						
Perl + Taint + CGI :: Untaint	Perl		✓				
Program shepherding	x86	✓					
Ruby	Ruby		✓				
WebSSARI	PHP						
PQL	JVM	✓	✓	✓	✓	✓	
AMNESIA	JVM						
Haldar et al.	JVM						
CSSE	JVM, PHP						
PHPrevent	PHP						

Figure 2: Summary of information about existing runtime systems for security. ~ indicates incomplete support.

3.2 Platform Support

4 Challenges

- Overhead
- Specification completeness
- Runtime soundness

5 Related Work

5.1 Server Software Protection

5.1.1 Improved Runtime Protection

PointGuard [9] [36] program shepherding [21] TIED, LibSafe, etc. [3] StackGuard [10]

5.1.2 Binary Rewriting

[26] [4] [11] [5] [30]

5.1.3 Hardware Protection Mechanisms

StackGhost [12] [39] [25]

5.1.4 Failure-Oblivious Computing

Failure-oblivious computing [33]

5.2 Application Software Protection

[19] [24] [16] [15] [14] [6] [29] [27]

5.3 Other

[22] [40] [31] [32]

6 Conclusions

References

- [1] C. Anley. Advanced SQL injection in SQL Server applications. http://www.nextgenss.com/papers/advanced_sql_injection.pdf, 2002.
- [2] C. Anley. (more) advanced SQL injection. http://www.nextgenss.com/papers/more_advanced_sql_injection.pdf, 2002.
- [3] K. Avijit, P. Gupta, and D. Gupta. TIED, LibsafePlus: Tools for runtime buffer overflow protection. In *USENIX Security Symposium*, pages 45–56, 2004.
- [4] D. Avots, M. Dalton, B. Livshits, and M. S. Lam. Improving software security with a C pointer analysis. In *Proceedings of the 27th International Conference on Software Engineering*, pages 332–341, 2005.
- [5] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, 2000.
- [6] S. Boyd and A. Keromytis. SQLrand: Preventing SQL injection attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, 2004.
- [7] CGI Security. The cross-site scripting FAQ. <http://www.cgisecurity.net/articles/xss-faq.shtml>.
- [8] Computer Security Institute. Computer crime and security survey. http://www.gocsi.com/press/20020407.jhtml?_requestid=195148, 2002.
- [9] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuardTM: Protecting pointers from buffer overflow vulnerabilities. In *Proc. of the 12th Usenix Security Symposium*, Aug. 2003.
- [10] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, Jan. 1998.
- [11] A. Dasari and P. Dasgupta. 2005.

- [12] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. pages 55–66.
- [13] S. Friedl. SQL injection attacks by example. <http://www.unixwiz.net/techtips/sql-injection.html>, 2004.
- [14] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for Java. In *21st Annual Computer Security Applications Conference*, Dec. 2005.
- [15] W. G. J. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005)*, Long Beach, CA, USA, Nov. 2005.
- [16] W. G. J. Halfond and A. Orso. Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks. In *Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)*, pages 22–28, St. Louis, MO, USA, May 2005.
- [17] D. Hu. Preventing cross-site scripting vulnerability. <http://www.giac.org/practical/GSEC/Deyu.Hu.GSEC.pdf>, 2004.
- [18] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web application code by static analysis and runtime protection. In *Proceedings of the 13th conference on World Wide Web*, pages 40–52, 2004.
- [19] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52, New York, NY, USA, 2004.
- [20] G. Hulme. New software may improve application security. <http://www.informationweek.com/story/IWK20010209S0003>, 2001.
- [21] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, Aug. 2002.
- [22] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: a basis for building self-protecting servers. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 213–222, New York, NY, USA, 2005.
- [23] B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. Submitted for publication, 2005.
- [24] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 365–383, 2005.
- [25] J. McGregor, D. Karig, Z. Shi, and R. Lee. A processor architecture defense against buffer overflow attacks, 2003.
- [26] J. Newsome, D. Brumley, and D. Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proceedings of the 13th Annual Network and Distributed Systems Security Symposium*, 2006. <http://www.cs.cmu.edu/~dbrumley/>.
- [27] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening Web applications using precise tainting. In *Proceedings of the Twentieth IFIP International Information Security Conference*, May 2005.
- [28] OWASP. Ten most critical Web application security vulnerabilities, 2004.
- [29] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection 2005 (RAID)*, 2005.
- [30] M. Prasad and T. cker Chiueh. A binary rewriting defense against stack based overflow attacks.
- [31] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. *SIGOPS Oper. Syst. Rev.*, 39(5):235–248, 2005.
- [32] F. Qin, J. Tucek, and Y. Zhou. Treating bugs as allergies: A safe method for surviving software failures. In *Proceedings of the USENIX 10th Workshop on Hot Topics in Operating Systems*, June 2005.
- [33] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebe. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, pages 303–316, 2004.
- [34] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 2001 Usenix Security Conference*, pages 201–220, 2001.
- [35] K. Spett. Cross-site scripting: are your Web applications vulnerable. <http://www.spidynamics.com/support/whitepapers/SPIcross-sitescripting.pdf>, 2002.
- [36] N. Tuck, B. Calder, and G. Varghese. Hardware and binary modification support for code pointer protection from buffer overflow. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 209–220, Washington, DC, USA, 2004.
- [37] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed Systems Security Symposium*, pages 3–17, 2000.
- [38] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336, New York, NY, USA, 2003.
- [39] J. Xu, Z. Kalbarczyk, S. Patel, and R. Iyer. Architecture support for defending against buffer overflow attacks, 2002.
- [40] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a range of attacks. 2005.