A Discussion of Performance Optimizations for Compiler-Based Buffer Overflow Instrumentation

Timothy Tsai, *Sun Microsystems*

***Abstract*—This paper discusses an optimization to existing compiler-based buffer overflow solutions to reduce the impact of software instrumentation on performance. The most common form of buffer overflow attack is the *stack-smashing attack*, in which arbitrary attack code is executed by overflowing stack buffers and overwriting return addresses. The most popular tools to address this type of attack utilize compiler modifications to insert instrumentation code that is executed at run-time. This code typically inserts *canary* or protection words that signal when a return address has been modified by a buffer overflow. Although the protection can be very effective, the instrumentation exacts a performance cost. This paper estimates the potential impact of a specific performance optimization and discusses specific scenarios that require special care.**

***Index Terms*—Buffer overflow, stack-smashing, compiler, performance optimization**

I. Introduction

Buffer overflow vulnerabilities are programming errors that form the basis for many security alerts [1]. Among the solutions for detection of buffer overflows are compiler-based solutions such as StackGuard [2,3] and SSP/ProPolice [4] that automatically insert run-time instrumentation. This instrumentation places and monitors *canaries* (memory guard words whose compromise signal a buffer overflow) to protect return addresses on the stack of a process.

The effectiveness of the canary approach has been demonstrated not only in published papers but also by its inclusion and use in popular software distributions (including OpenBSD, FreeBSD, various flavors of Linux, Solaris) and tools (gcc, Visual C++[5]). Many, albeit not all, stack-smashing attacks are accurately detected with virtually no false positives. In fact, one implementation (StackGuard) was reported to have successfully survived most security attacks in a contest held at a well-known security conference [6].

Perhaps the most significant concern that limits wider utilization is the incurred run-time overhead. As reported by the authors of StackGuard and SSP, expected overheads can range from low single-digit percentage overheads to double-digit percentages or even more [2,4]. To address the performance issue, the authors of StackGuard have suggested a significant optimization. The performance savings is derived from selectively omitting canary protection for certain stack frames. Since canaries protect against buffer overflows, if no buffers are allocated in a particular stack frame, then no overflow can ostensibly occur in that stack frame. Furthermore, if a buffer does exist but it is not accessed by any statement (either in that same function or elsewhere), then that buffer cannot be overflowed. [2]

The authors of SSP make a similar observation. However, they suggest a more aggressive form of optimization that instruments only functions that contain string buffers. The justification is predicated on the idea that most buffer overflow vulnerabilities target string buffers. [4]

II. Estimated Overhead Reduction

	Total Function Calls Made	***Optimizable Functions***	***Estimated Overhead Reduction***
Apache 2.2.0	503	42 (95.45%)	97.02%
Sendmail 8.13.6	1499	27 (79.41%)	73.32%

Table 1: Count of Functions Benefiting from Optimization

An accurate estimation of the expected reduction in overhead due to the optimization suggested above requires a profiling study that covers several applications of interest. Table 1 shows a preliminary study based on profiling analysis of two server applications, Apache and Sendmail. These two programs are server programs that are often executed with root permission. Thus, a buffer overflow in the context of either program could form the basis for a critical, remotely exploitable security attack. The selected programs were executed on a Solaris 10 system and profiled using DTrace [7], which is able to perform profiling of functions without the need for sampling. The data collected by DTrace includes a count of the number of times each function is called. The source code of each function was then analyzed to identify those functions that allocate local buffers and therefore require canary protection. Since the source code analysis was performed via human inspection, only functions statically linked to the main executable (i.e., not including dynamically linked libraries) were considered. Table 1 shows both the number of optimizable functions (the number of source code function that do not allocate a local buffer and therefore do not require canary protection) as well as the estimated overhead reduction after considering the number of times each function was actually called based on the profiling data. The results suggest that the overhead may be reduced significantly.

It should be noted that this analysis is somewhat simplistic because the profiling study is based on limited workloads for a small set of programs. However, the results show that the optimizations suggested by the authors of StackGuard and SSP may provide significant reductions in performance overhead, specifically that most of the overhead, which is a constant time overhead per function call, can be eliminated with the optimization. This is an encouraging sign.

III. Setjmp()/Longjmp() Issue

For proper implementation of the suggested optimizations, care should be exercised in one particular situation. The authors of StackGuard correctly point out that the location of the vulnerable

stack buffer is more important than the location of the code that accesses that buffer. The offending code that contains the bounds checking flaw can exist in either the same function or a different function. If the flaw exists in a different function, then the overflow can only be detected when the program executes a return from the function that originally allocated the overflowed buffer, so it makes sense to insert the canary protection for that particular function.

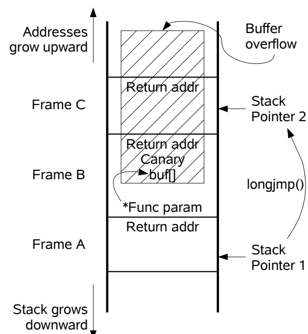

Figure 1: Setjmp()/longjmp() Vulnerability" Return Address is Overwritten

However, there is one case where the above reasoning does not hold. Other researchers (e.g., [8]) have pointed out vulnerabilities associated with setjmp()/longjmp(). However, if canary protection is selectively omitted for performance reasons as described above, then an additional vulnerability exists. If a program utilizes setjmp()/longjmp() to return the control flow to a function that is not the direct caller for the current function, then several stack frames, along with their return addresses and associated canaries can be popped off the stack. Consider the situation in Figure 1. The current context is in Frame A, which has a function parameter that contains a pointer to a buffer (buf[]) in Frame B. A buffer overflow for buf[] occurs, and then a longjmp() instruction is executed, which pops Frames A and B off the stack without returning explicit control to the corresponding stack frames. Because the epilog code that contains the canary check for the function corresponding to Frame B is never executed, the canary check for that frame is never executed; hence, the overflow is not detected. Furthermore, if the overflow extends to Frame C beyond Frame B, and Frame C is not protected by a canary because it doesn't contain a vulnerable buffer, then the return address from Frame C can be compromised without detection.

The solution is to ensure that all functions that call setjmp() are protected via a canary. However, one further caution must be

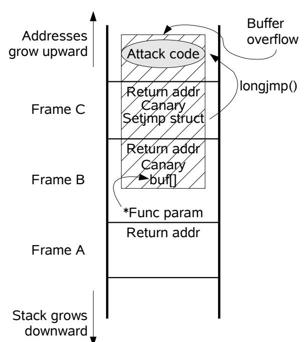

Figure 2: Setjmp()/longjmp Vulnerability: Setjmp()Structure is Overwritten

heeded. Consider the situation in Figure 2, which shows a scenario similar to that in Figure 1. However, in Figure 2, the structure containing the setjmp() state is allocated on the stack in the same function containing the call to setjmp(). If the overflow from Frame B corrupts the contents of Frame C, then the setjmp() state structure can be overwritten. Since this structure contains a pointer to the address where execution will be resumed after the corresponding longjmp(), the structure must be protect in much the same way as the return address. That is, the structure must itself be protected by a canary to guard against corruption. Note that the need to protect the setjmp() structure with a canary is not affected by any reordering of local variables that SSP might perform, since the the SSP reordering only protects local variables from being overwritten by overflows of buffers in the same stack frame.

Implementation of canary protection for the setjmp() state structure requires instrumentation of the calls to setjmp() and to longjmp(). Upon encountering a setjmp(), a canary must be placed before the setjmp() state structure. When a longjmp() call is encountered, verification of the appropriate canary must be performed before the transfer of control flow to the longjmp() target. Fortunately, the required instrumentation can be done by the compiler in much the same way as the original canary instrumentation. The only change is the addition of additional code in the epilog of setjmp() and the prolog of longjmp().

IV. Conclusion

This paper suggests that omitting canary protection for selected functions can greatly reduce the incurred run-time overhead of compiler-inserted instrumentation. Although calls to setjmp()/longjmp() do not necessarily occur often, their possible presence must be accounted for when implementing the optimizations discussed in this paper.

Bibliography

[1] Arash Baratloo, Navjot Singh, and Timothy Tsai, "Transparent Run-Time Defense Against Stack Smashing Attacks," in *Proc. Of the 2000 USENIX Technical Conference*, San Diego, California, USA, June 2000.

[2] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang, "Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," in *Proc. of the 7th USENIX Security Symposium*, San Antonio, Texas, USA, January 1998, pages 63-78, <http://citeseer.ist.psu.edu/cowan98stackguard.html>.

[3] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen, "Protecting Systems from Stack Smashing Attacks with StackGuard," in *Linux Expo*, Raleigh, North Carolina, USA, May 1999.

[4] "GCC Extension for Protecting Applications from Stack-Smashing Attacks," <http://www.research.ibm.com/trl/projects/security/ssp/>.

[5] "/GS (Buffer Security Check) (C++)," <http://msdn2.microsoft.com/en−us/library/8dbf701c(VS.80).aspx>.

[6] "Nearly 100 Hackers Fail to Crack WireX Immunix Server; WireX Software Staves Off Toughest Attacks, Earns 2nd Place in DefCon 10's 'Capture the Flag' Games," <http://www.businesswire.com/cgi−bin/f_headline.cgi?bw.082602/222380413>.

[7] "Solaris Dynamic Tracing Guide," <http://docs.sun.com/app/docs/doc/817-6223>.

[8] John Wilander and Mariam Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention," in *Proc. 10th Network and Distributed System Security Symposium (NDSS)*, San Diego, California, USA, February 2003, pp. 149-162.