Precise Buffer Overflow Detection via Model Checking

Sagar Chaki, Scott Hissam
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, USA
chaki@sei.cmu.edu, shissam@sei.cmu.edu

Abstract

Buffer overflows are the source of a vast majority of vulnerabilities in today's software. Existing solution for detecting buffer overflow, either statically or dynamically, have serious drawbacks that hinder their wider adoption by practitioners. In this paper we present an automated overflow detection technique based on model checking and iterative refinement. We discuss advantages, and limitations, of our approach with respect to today's existing solutions. We also describe how our approach may be implemented on top of a model checking technology being developed at the Software Engineering Institute (SEI).

Introduction

Software vulnerabilities [1, 2] are one of the major causes of concern in today's information centric world. For example it is estimated [3] that "Hacker attacks cost the world economy a whopping $1.6 trillion in 2000" and that "US virus and worm attacks cost $10.7 billion in the first three quarters of 2001". This problem is further highlighted by the increasing number of attacks that exploit such vulnerabilities. For example, "The CMU CERT Coordination Center reported 76,404 attack incidents in the first half of 2003, approaching the total of 82,094 for all of 2002 in which the incident count was nearly four times the 2000 total. If anything, the CERT statistics may understate the problem, because the organization counts all related attacks as a single incident. A worm or virus like Blaster or SoBig, a self-replicating program that can infect millions of computers, is but one event."

Buffer overflows are widely recognized [17] to be the prime source of vulnerabilities in commodity software. For example, the CodeRed[1] worm that caused an estimated global damage worth $2.1 billion in 2001 [3] exploited a buffer overflow in Windows. In addition, Wagner et al. [22] report, on the basis of CERT advisories, that "buffer overruns account for up to 50% of today's vulnerabilities, and this ratio seems to be increasing over time."

[1] http://www.cert.org/advisories/CA-2001-19.html

Broadly speaking, a buffer overflow occurs when a piece of data *D* is written to a buffer *B* such that the size of *D* is greater than the legally allocated size of *B*. In the case of a type-safe language or a language with explicit bound checking (such as Java), this leads to an exception being thrown. Unfortunately, the vast majority of commercial and legacy software is written in *unsafe* languages (such as C or C++). Such languages allow buffers to be overflowed with impunity.

Buffer overflows are typically used by attackers to execute arbitrary code (such as a *shell*) with administrative privileges. For example, a common strategy is to overwrite a program's activation record (commonly called a *stack smashing* attack) in order to redirect its control flow to any desired point. As such, buffer overflows are extremely dangerous and can lead to catastrophic system compromises and failures.

As mentioned before, the vast majority of commercial off-the-shelf (COTS)[2] and legacy software involves unsafe programming languages and is therefore particularly vulnerable to buffer overflows. Unfortunately, our critical infrastructure is becoming increasingly dependent on legacy and COTS systems. For instance, only about 23% of the software in the Joint Strike Fighter (JSF) consists of new automatically generated code that we may reasonably assume to be free of buffer overflows. However, the remaining 77% of the JSF software is an assembly of COTS components (12%), legacy code (30%), multi-used systems (23%) and manually written (12%) programs [3], developed in large part using languages that have no intrinsic safeguard against buffer overflows. Thus, an overwhelming majority of the JSF software is prone to buffer overflow vulnerabilities. Other instances of systems relying on COTS software include the voice switching systems of the White House and the NSANet [3]. More importantly, this situation is only going to worsen in the days to come. It is therefore imperative that we develop effective tools that can detect, and aid in fixing, buffers overruns in large software systems written in unsafe languages.

Buffer Overflow Detection: Existing Approaches

Given its significance, it is not surprising that considerable effort has been devoted toward the development of buffer overflow detection systems. Nevertheless, a satisfactory solution to this problem remains elusive. In this section we will discuss existing automated techniques for overflow detection. Manual approaches are inherently non-scalable, therefore we focus on procedures that involve a fair amount of automation.

A number of approaches that have been proposed to detect buffer overflows are *type-theoretic* [18] in nature. Such techniques require that programs be written in a type-safe language and are hence not applicable to the vast body of (legacy as well as in-production) systems that involve unsafe languages such as C or C++. Techniques based on simulation or testing are inexpensive and widely prevalent. However, they usually suffer from extremely low coverage and are typically unable to provide any reasonable degrees of assurance about critical software systems.

Yet other buffer overflow detection schemes advocate a run-time or *dynamic* [19, 20, 21] strategy. Such approaches incur performance penalties that are unacceptable in many

[2] Strictly speaking, our technique only applies to COTS components for which source code is also available.

situations. Even when performance is not a serious issue, it is often imperative that we be assured of the correctness of a system before it is deployed since any failure in real-life would be catastrophic. Such guarantees can only be obtained via *static* approaches.

More recently, a number of static approaches for buffer overflow detection have been proposed that rely on static analysis of programs. These approaches are usually based on converting the buffer overflow problem into a constraint solving problem, such as integer range checking [22] or integer linear programming [23]. Static analysis amounts, in principle, to a form of model checking [24] over the control flow graph of a program. However, a control flow graph is an extremely imprecise model. Therefore, in practice, static analysis is plagued by *false positives.* More specifically, every probable buffer overflow flagged by static analysis must be manually inspected to ensure that it corresponds to an actual problem and is not an artifact of the imprecise model which has no concrete realization.

In practice, three drawbacks seriously limit the effectiveness of static analysis as an approach for buffer overflow detection. First, each bug report must be manually verified. Second, a large majority of problems reported by static analysis turn out to be false alarms. Finally, there is no automated procedure for getting rid of false alarms by constructing more precise models.

Our buffer overflow detection technique is also static but is based on a paradigm called *iterative refinement* that overcomes each of the above shortcomings of static analysis. We will describe iterative refinement in detail shortly. But first we describe our model checking infrastructure on which we plan to develop the buffer overflow detection technology.

The PACC Initiative

The Predictable Assembly from Certifiable Components (PACC) [6] initiative at the SEI aims to predict the behavior of a component-based system prior to implementation, based on known properties of components. The vision of PACC is that software components have certified properties (for example, safety, reliability, and performance) and the behavior of systems assembled from components is predictable. To this end, PACC is developing a component specification language called Construction and Composition Language (CCL) [7], a run-time called Pin [10], and a set of reasoning frameworks [9], packaged together as a Prediction Enabled Component Technology (PECT) [8].

Currently, two reasoning frameworks are being developed by the PACC team. The performance family of reasoning frameworks [11] employs analytic theories such as rate monotonic analysis and real-time queuing theory for predicting run-time attributes related to system performance. Typical examples of such attributes are the average and worst-case latencies of a system under various distributions of the arrival rate and execution times of jobs. The ComFoRT [12] reasoning framework uses software verification technology based on model checking [4] and automated abstraction-refinement to prove claims related to system reliability and safety. A significant advantage of both these reasoning frameworks is that they are static, and do not require a system to be executed for making any kind of prediction about its run-time behavior.

In the remainder of this paper we will establish strong connections between the model checking technology being developed as part of the ComFoRT reasoning framework and buffer overflow detection. In particular, we will show how the model checking infrastructure developed as part of ComFoRT can be leveraged to develop an effective overflow detection system.

ComFoRT: Model Checking in PACC

Model checking is an automated approach for exhaustively analyzing whether systems satisfy specific behavioral claims that express safety and reliability requirements. Due to its exhaustive nature, model checking is especially attractive for analyzing concurrent systems where the number of possible interleaving between various components is quite large, and yet must be explored. A distributed safety critical software system is a typical example of such a system. The ComFoRT reasoning framework packages the effectiveness of state-of-the-art model checking in a form that enables software developers to apply the analysis technique without being experts in its use. The key ideas behind ComFoRT are:

- Safety and reliability claims about a target system are expressed using a state/event-based temporal logic. The logic, called SE-LTL [13], was developed particularly for the purpose of specifying properties of software systems. An SE-LTL claim essentially encodes desirable (or undesirable) sequences composed of a combination of constraints on data and occurrence of events. Such claims may equivalently be expressed as finite state machines.
- A CCL component specification, which also includes a relevant claim to be verified, is automatically interpreted into a concurrent program that can be input to a model checker. This program is written in a restricted form of ANSI-C along with finite state machine descriptions of certain library routines.
- The concurrent program resulting from the interpretation is input to a model checker, *Copper.* The output of Copper is either *yes,* which means that the claim holds, or *no,* which means that it does not. If the result is *no,* then Copper also returns a *counterexample,* which is an actual execution of the program that violates the claim. The counterexample is reverse-interpreted to the original CCL form and returned as diagnostic feedback.

Iterative Refinement

The ability of Copper to verify claims on concurrent, and in general infinite-state, C programs is based on the iterative refinement paradigm. As mentioned before, iterative refinement addresses the three critical drawbacks of conventional static analysis of programs. It achieves this via an iterative procedure that can be described (please see Figure 1 for a pictorial description), in the context of Copper, as follows:

1. Construct a finite state conservative concurrent model M of the target C program. Copper uses a technique called predicate abstraction to achieve this.
2. Model check M against the desired claim. If M satisfies the claim, and since it is a conservative model, then so does the original C program. In this case, Copper exits

with a *yes*. Otherwise let CE be a counterexample to the claim with respect to the model M.

3. Check if CE is also a counterexample with respect to the original C program. If so, then Copper exits with *no* and returns CE as the counterexample. Otherwise CE is said to be *spurious* since it is a behavior that does not belong to the original C program, but was only introduced in the model M by the abstraction process.
4. Construct a more precise model M using the spurious CE. The new model is guaranteed not to contain CE as an admissible behavior. Repeat from step 2 above.

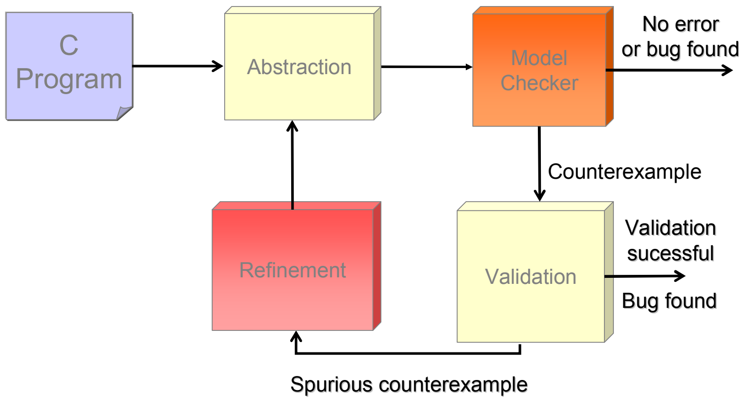

Figure 1: Iterative Refinment in ComFoRT

In summary, iterative refinement improves upon static analysis by enabling automated verification of counterexample for spuriousness, and automated model refinement to eliminate spurious counterexamples. It is therefore extremely suitable for detecting violations of safety conditions in large-scale software systems in an automated and scalable manner. In particular, the absence of buffer overflows is a perfect example of a safety claim that is amenable to iterative refinement.

Our Approach: PACC for Buffer overflow

In general, model checking is an extremely attractive choice as a foundation for buffer overflow detection technology. The main reason behind this is that buffer overflow is concerned with a program's control flow and simple relationships between its data. More specifically, in order to prove the absence of buffer overflow, we only need to show, for every control flow point where some data *D* is being written into a buffer *B,* that the allocated size of *B* is no less than the size of *D*. We do not need to be concerned about the properties of the data *D* itself. Such situations are most conducive for software model checking to succeed[3].

In contrast to dynamic approaches [29, 30, 31, 5], model checking is static. It therefore incurs no run-time overheads and requires no mechanisms for graceful recovery once an anomaly has been detected. This feature is particularly useful for mission-critical software whose correctness must be ensured *before* actual deployment. Finally, the capability of model checking to generate counterexamples is invaluable for producing

[3] Indeed, model checking has been applied with considerable success toward the detection of software bugs (including security bugs) [25, 26, 27] in recent times.

diagnostic feedback[4]. However, in order to be fruitfully applied to infinite state systems, such as software, model checking must be combined with a technique such as iterative refinement.

We therefore plan to apply iterative refinement for buffer overflow detection in C programs. More specifically, we will develop our overflow detection technology on top of the ComFoRT reasoning framework. Several features of ComFoRT make it a lucrative and advantageous choice for buffer overflow flaw detection in comparison to the tools and techniques presented above. First, ComFoRT includes powerful and automated abstraction-refinement techniques that allow it to model source code at the correct level of granularity. As mentioned before, all existing static overflow detection tools are limited by their ability to only model programs as their control flow graph. This makes them prone to an excessive number of false alarms and inhibits their wider adoption by practitioners.

In addition, ComFoRT allows the verification of concurrent systems. This will allow our approach to detect buffer overruns in multi-threaded and distributed systems. Such vulnerabilities are expected to become increasingly more frequent, and threatening, in the days to come. They are also virtually impossible to detect using present day tools and algorithms that can only analyze components in isolation.

Another important problem faced by existing buffer overflow detection systems is the inability to specify appropriate environments. This ultimately results in an increased number of false alarms. It is important to note that these false alarms are the result of imprecise modeling of environment as opposed to imprecise modeling of the program being analyzed. The ability to analyze concurrent systems will enable us to specify proper environments and eliminate this category of spurious counterexamples as well.

Certifying Buffer Overflow Freedom

For software systems that are mission-critical in nature, the ability to trust analysis results becomes imperative. This is no longer a trivial issue since the tools that analyze complex software have themselves become quite complicated. As part of both the performance and ComFoRT reasoning frameworks, we are developing validation techniques that enhance our ability to achieve increased confidence in our predictions.

In the context of the ComFoRT reasoning framework, we are investigating techniques for combining certifying model checking [15] and proof carrying code [14]. The goal is to enhance the existing ComFoRT framework so that it outputs a *proof certificate* along with *yes* if a desired claim is found to hold. The validity of the proof certificate can be checked separately to assure the correctness of positive answer returned by ComFoRT.

The power of this technique lies in the fact that it can be automated and applied to realistic software systems. In addition, certificates are tamper-proof. A valid proof certificate guarantees that the software system is provably secure, even if it was generated by an untrusted source and transmitted over an untrusted communication channel. We

[4] Interestingly, this feature of model checking has also been successfully used to generate attack graphs [28] for intrusion detection in large-scale networks.

plan to extend this technique to generate trusted certificates for software certified to be free of buffer overflows.

Challenges and Success Measures

It is also important to discuss some of the challenges that must be overcome in order to adapt and apply successfully model checking technology to buffer overflow detection. An important challenge is scalability. Model checking is known to be hampered by the *state-explosion* problem, in particular for concurrent systems whose number of reachable states increases exponentially with the number of components. We believe that powerful abstraction and compositional reasoning techniques, some of which were developed as part of ComFoRT, enable us to tackle this problem.

The predicate abstraction implemented as part of ComFoRT has limited support for pointers. On the other hand, pointers are an integral part of the buffer overflow problem. We must therefore add improved pointer support to ComFoRT as part of the development of our buffer overflow detection tool. While the theory behind this step is fairly well-know, its practical ramifications are yet to be completely understood. Finally, the abstraction refinement scheme implemented in ComFoRT is geared toward SE-LTL counterexamples. We must tailor this step for buffer overflows. Once again, this is theoretically straightforward but practically unchartered.

As the old saying goes, nothing succeeds like success. Thus, the ultimate success story would be the fruitful demonstration of the effectiveness of our tool on a wide selection of representative examples. Some additional achievements would be: (a) a better understanding of the applicability of model checking for buffer overflow detection, and (b) a qualitative measure of the relative advantages of iterative refinement over traditional static analysis methods in the context of buffer overflow detection.

Summary

In summary, the goal of this effort is to develop a buffer overflow infrastructure that:

- Models source code more *precisely*, yet *scalably*, than existing tools. This will reduce the amount of false alarms, and enable us to analyze larger program, fostering wider acceptance by practitioners. We believe that techniques such as iterative refinement and symbolic representations will help us in this direction.
- Detects buffer overflows that arise only due to the interaction of multiple components and specifies appropriate environments. This is impossible using existing buffer overflow detection tools that analyze components in isolation. We believe that or use of model checking will provide us with the vital capability of finding *distributed buffer overflows*.
- *Certifies* code to be free of buffer overflows. This is clearly an extremely powerful capability that is also lacking in existing tools. An important precondition for certification is that analysis must be conservative, i.e., if the analysis cannot find any problems, then there really are no problems. Fortunately, the abstraction techniques we use are conservative.

Conclusion

Society is becoming ever increasingly reliant on software to manage critical infrastructure. Buffer overflows remain, and continue to grow in importance as, the major source of vulnerabilities in such software systems. Despite considerable effort, a satisfactory solution to the buffer overflow detection problem remains elusive. In this paper we have presented a static buffer overflow detection scheme based on model checking and iterative refinement. In particular, we believe that the model checking technology being developed as part of the PACC initiative at the SEI can be adapted to develop such a buffer overflow detection tool. Our tool will not only be able to analyze concurrent systems but will also be able to generate certificates that guarantee that the target program is free from buffer overflows. We believe that such a tool will go a long way in enhancing the state-of-the-art in our buffer overflow detection and certification technology.

Reference:

[1] A Structured Approach to Classifying Buffer overflow Vulnerabilities, Robert C. Seacord, Allen Householder, Technical Note, Technical note, CMU/SEI-2005-TN-003.

[2] Formal Modeling of Vulnerability, William Fithen, Shawn Hernan, Paul O'Rourke, David Shinberg, Bell Labs Technical Journal 8, 4 (February 5, 2004) : 173-186.

[3] Systems, Networks and Information Integration Context for Software Assurance, Joe Jarzombek, Presentation, January 2004, http://www.sei.cmu.edu/products/events/acquisition/2004-presentations/jarzombek/jarzombek.pdf.

[4] Model Checking, Edmund Clarke, Orna Grumberg, Doron Peled, MIT Press, 2000.

[5] Bro: A System for Detecting Network Intruders in Real-Time, Vern Paxson.

[6] Predictable Assembly from Certifiable Components (PACC), http://www.sei.cmu.edu/pacc.

[7] Snapshot of CCL: A Language for Predictable Assembly, Kurt Wallnau, James Ivers, Technical report, CMU/SEI-2003-TN-025.

[8] Volume III: A Technology for Predictable Assembly from Certifiable Components (PACC), Kurt Wallnau, Technical report, CMU/SEI-2003-TR-009.

[9] Reasoning Frameworks, Len Bass, James Ivers, Mark Klein, Paulo Merson, Technical report, CMU/SEI-2005-TR-007.

[10] Pin Component Technology and its C Interface, Scott Hissam, James Ivers, Daniel Plakosh, Kurt Wallnau, Technical note, CMU/SEI-2005-TN-001, to appear.

[11] Performance Property Theories for Predictable Assembly from Certifiable Components (PACC), Scott Hissam, Mark Klein, John Lehoczky, Paulo Merson, Gabriel Moreno, Kurt Wallnau, Technical Report, CMU/SEI-2004-TR-017.

[12] Overview of ComFoRT: A Model Checking Reasoning Framework, James Ivers, Natasha Sharygina, Technical note, CMU/SEI-2004-TN-018.

[13] State/Event-based Software Model Checking, Sagar Chaki, Edmund Clarke, Joel Ouaknine, Natasha Sharygina, Nishant Sinha, Proceedings of 4th International Conference on Integrated Formal Methods (IFM), LNCS 2999, pages 128-147, 2004.

[14] George Necula, "Proof-carrying code," In Proc. 24th ACM Symposium on Principles of Programming Languages (POPL), New York, Jan. 1997 (106-119).

[15] Certifying Model Checkers, Kedar Namjoshi, Proceedings of the 13th International Conference on Computer Aided Verification (CAV), LNCS 2102, pages 2-13, 2001.

[16] Fred Schneider, “Enforceable Buffer overflow Properties,” in ACM Transactions on Information and System Buffer overflow, Vol. 3, No. 1, February 2000 (30-50).

[17] Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade, Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, Jonathan Walpole, Proceedings of the DARPA Information Survivability Conference and Expo (DISCEX), 1999.

[18] Detecting Format String Vulnerabilities with Type Qualifiers, Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, David Wagner, Proceedings of the 10th USENIX Buffer overflow Symposium, pages 210-220, 2001.

[19] A Practical Dynamic Buffer Overflow Detector, Olatunji Ruwase, Monica Lam, Proceedings of the Network and Distributed System Buffer overflow (NDSS) Symposium, pages 159--169, February 2004.

[20] Backwards-compatible bounds checking for arrays and pointers in C programs, Richard W M Jones, Paul H J Kelly, Proceedings of AADEBUG, Sweden, 1997.

[21] Using Program Transformation to Secure C Programs Against Buffer Overflows, Christopher Dahn and Spiros Mancoridis, Proceedings of the 10th Working Conference on Reverse Engineering (WCRE ’03), pages 323, Washington, DC, USA, 2003.

[22] A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities, D. Wagner, J. Foster, E. Brewer, and A. Aiken, Network and Distributed System Buffer overflow Symposium, San Diego, CA, February 2000.

[23] Buffer Overrun Detection using Linear Programming and Static Analysis, Vinod Ganapathy and Somesh Jha and David Chandler and David Melski and David Vitek, Proceedings of the 10th ACM conference on Computer and communications buffer overflow (CCS ’03), pages 345-354, Washington D.C., USA, 2003.

[24] Dataflow analysis is model checking of abstract interpretations. David A. Schmidt, Conference Record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego CA, January 1998.

[25] MOPS: An Infrastructure for Examining Buffer overflow Properties of Software, Hao Chen and David Wagner.

[26] Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code, Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf.

[27] Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions, Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem.

[28] Automated Generation and Analysis of Attack Graphs, Oleg Sheyner, Somesh Jha, Jeannette M. Wing.

[29] Dynamic Detection and Prevention of Race Conditions in File Accesses, Eugene Tsyrklevich, Bennet Yee.

[30] Dynamic Taint Analysis: Automatic Detection, Analysis, and Signature Generation of Exploit Attacks on Commodity Software, James Newsome, Dawn Song.

[31] A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors, R. Sekar, M. Bendre, D. Dhurjati, P. Bollineni.