Detecting Overflow Vulnerabilities Using Automated Verification

Brian A. Malloy, Murali Sitaraman, Jason O. Hallstrom
School of Computing
Computer Science Division
Clemson University
Clemson, SC, USA 29634-0974
{malloy, murali, jasonoh}@cs.clemson.edu

1. INTRODUCTION

Maintaining the security of our computer systems has become one of the dominant aspects of the war on terror and many researchers and developers predict that the next attack against the United States will be a computer attack [4, 8, 10]. The Department of Homeland Security has established a partnership with industry, the United States Computer Readiness Team (US-CERT), to protect the nation's Internet infrastructure [1]. The US-CERT sponsors an automated, web-based repository of standards based vulnerability data (NVD), which includes a database of security related software flaws [2].

One of the most frequently occuring attacks reported at NVD, and other vulnerability databases, is remotely exploitable buffer overflow, which consume as much as 25% of the reported attacks [2, 6]. Moreover, buffer overflow attacks have been occuring at increasing rates. The bar chart in Figure 1 captures data, gathered from the NVD web site, about the number of buffer overflow vulnerabilities reported from the year 2000 until November of 2008. The bar on the left side of Figure 1 lists the number of buffer overflow vulnerabilities reported in the year 2000 as 202, and the bar on the right side of the figure lists the number of buffer overflow vulnerabilities reported in the year 2007 as 822. There have already been 537 buffer overflow vulnerabilites reported during 2008, as illustrated by the bar at the far right of Figure 1, which indicates that the buffer overflow vulnerability remains an important security problem.

There has been much research directed at the buffer overflow vulnerability. Wagner et al. frame buffer overflow vulnerabilities as an integer range analysis problem, and they present a static scalable approach toward detection of the problem [9]; however, the solution described by Wagner et al. is not precise and cannot handle heap-based vulnerabilities. Haugh and Bishop extend Wagner et al. using a dynamic approach to detection of buffer overflow, exploiting an extended form of testing [5]; however, the Haugh and Bishop approach is limited by the coverage provided by the test suite and failed to detect some known vulnerabilites in the applications under test. Cooprider et al. describe an instrumentation-based approach to detecting buffer overflow vulnerabilities in sensor network systems [3]; the contribution is specific to TinyOS. The approach relies on language extensions that make buffer boundaries manifest in program source code, which are used to automate the insertion of appropriate runtime bounds checks. The approach is not extensible to arbitrary data types, is not provably correct, and is inherently incomplete due to its dynamic nature.

In this paper, we apply lightweight formal methods to address the problem of vulnerabilities in software applications. Our approach bridges the gap between formal methods verification and security problems because most security constraints can be formulated as an integer range constraint problem [9]. We leverage an existing tool set and an automated verifier to provide a solution to the buffer overflow vulnerability problem that is both precise and scalable. We describe an implementation of our approach that has proven successful in the test cases that we consider. In the next section, we provide an overview of our approach. In the full paper, we provide a complete description of the methodology and the results of our case study.

2. OVERVIEW

Figure 2 provides an overview of our verification approach. To begin, the system developer follows a standard development process: no formal specifications are included in the source code. This is illustrated in the figure using a circled "1." The source code may, however, leverage a number of library components, which may include partial specifications. In our approach, these specifications are expressed using the RESOLVE specification language [7][1]. A key motivation for selecting this notation is its extensibility — in particular, RESOLVE is extensible to new mathematical definitions, models, and theories, including general-purpose theorems and lemmas that can be proven off-line. This extensibility is essential for specifying software components, which can be arbitrarily complex, both in terms of state and behavior. The component library, partially specified in RESOLVE, is illustrated at the top of the figure.

[1]RESOLVE is an integrated specification *and* implementation language. For our purposes, we do not require use of the implementation component of RESOLVE.

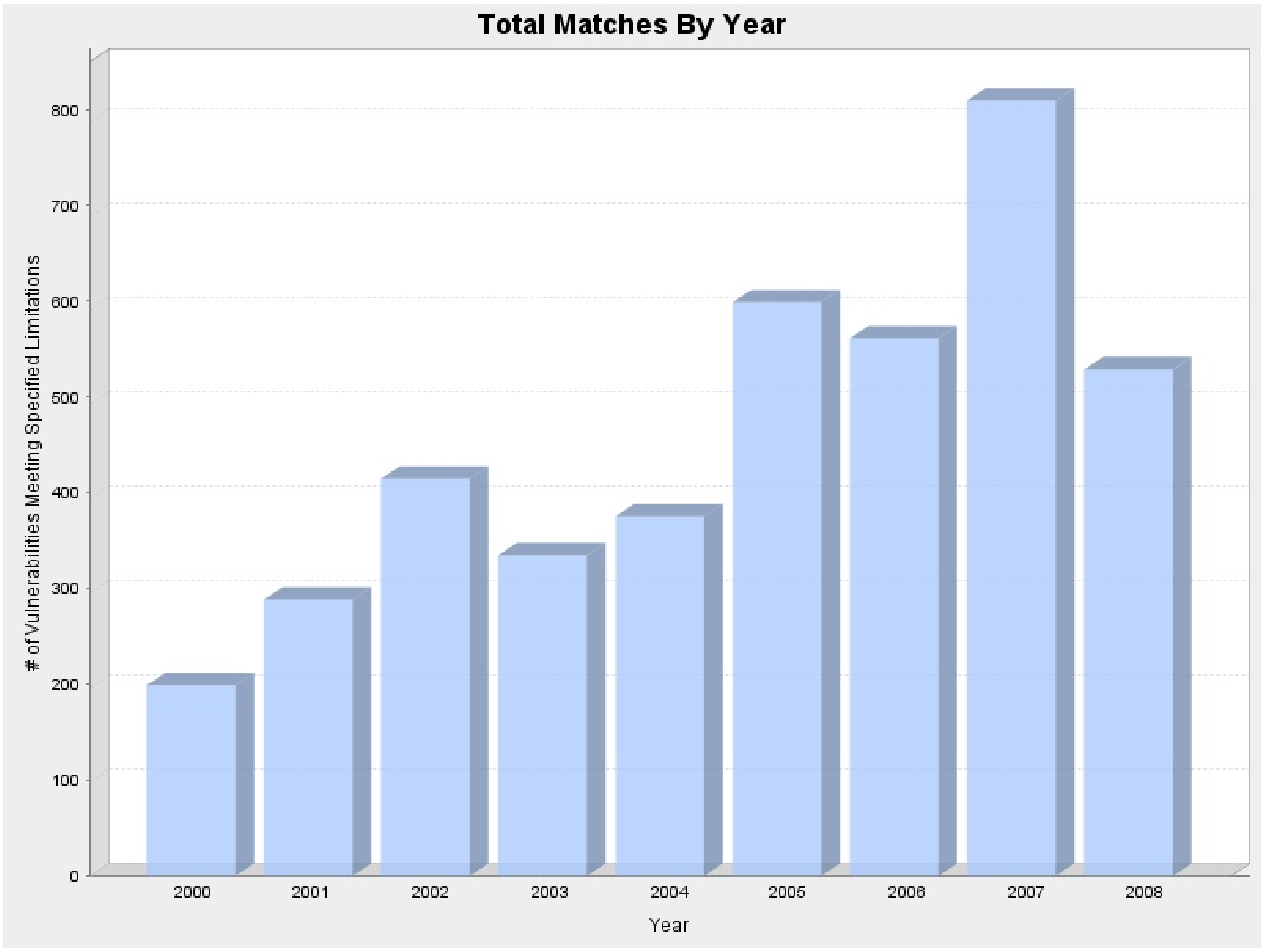

Figure 1: Buffer Overflow Vulnerabilities. *The bar chart in this figure illustrates information about the number of buffer overflow vulnerabilities reported on the NVD database from January, 2000 until November, 2008.*

The user component and all referenced library components serve as input to our modular *verification system*, illustrated in the center of the figure. As noted, the verification system is composed of two sub-elements. The first element is a *verification condition generator*, responsible for enumerating the mathematical assertions that must be proved to ensure program correctness. While the enumeration task may involve complex assertions, it is largely mechanical: Consider a user component U that implements a single operation $O()$. Assume further that $O()$ is implemented using operations provided by library components $L1$ and $L2$, both of which include partial specifications. To show that $O()$ is implemented correctly, the post-condition specified for $O()$ must be proven to hold upon termination of the operation. In effect, the verifier must ultimately prove an implication that states that the pre-condition of $O()$ combined with the effects of calls to $L1$ and $L2$ imply the post-condition of $O()$. The task of the verification condition generator is to enumerate the assertions that appear in the antecedent of this implication. Namely, the assertions that capture the impact of calls to $L1$ and $L2$ — the assertions described by their respective post-conditions[2].

This is of course not the only implication that defines correctness for $O()$. If, for example, in the implementation of $O()$, a developer were to violate a pre-condition of $L1$, the behavior specified by the corresponding post-condition need not hold. Hence, the verifier must also show that the pre-conditions of each operation invoked from within O are satisfied based on the post-conditions of the statements that preceded the call. The verification condition generator is responsible for generating these implications as well. Hence, the output of the verification condition generator is a set of implications which, if proven correct, show that $O()$ is correct with respect to its specification, assuming of course, that its dependent components are correct. The proof rules that form the foundation for this process, together with the proofs of soundness and completeness, are described in references [?].

[2]Here we omit consideration of component invariants, loop invariants, and progress metrics, all of which must, in the general case, be specified to ensure provable correctness. They are supported by our tools.

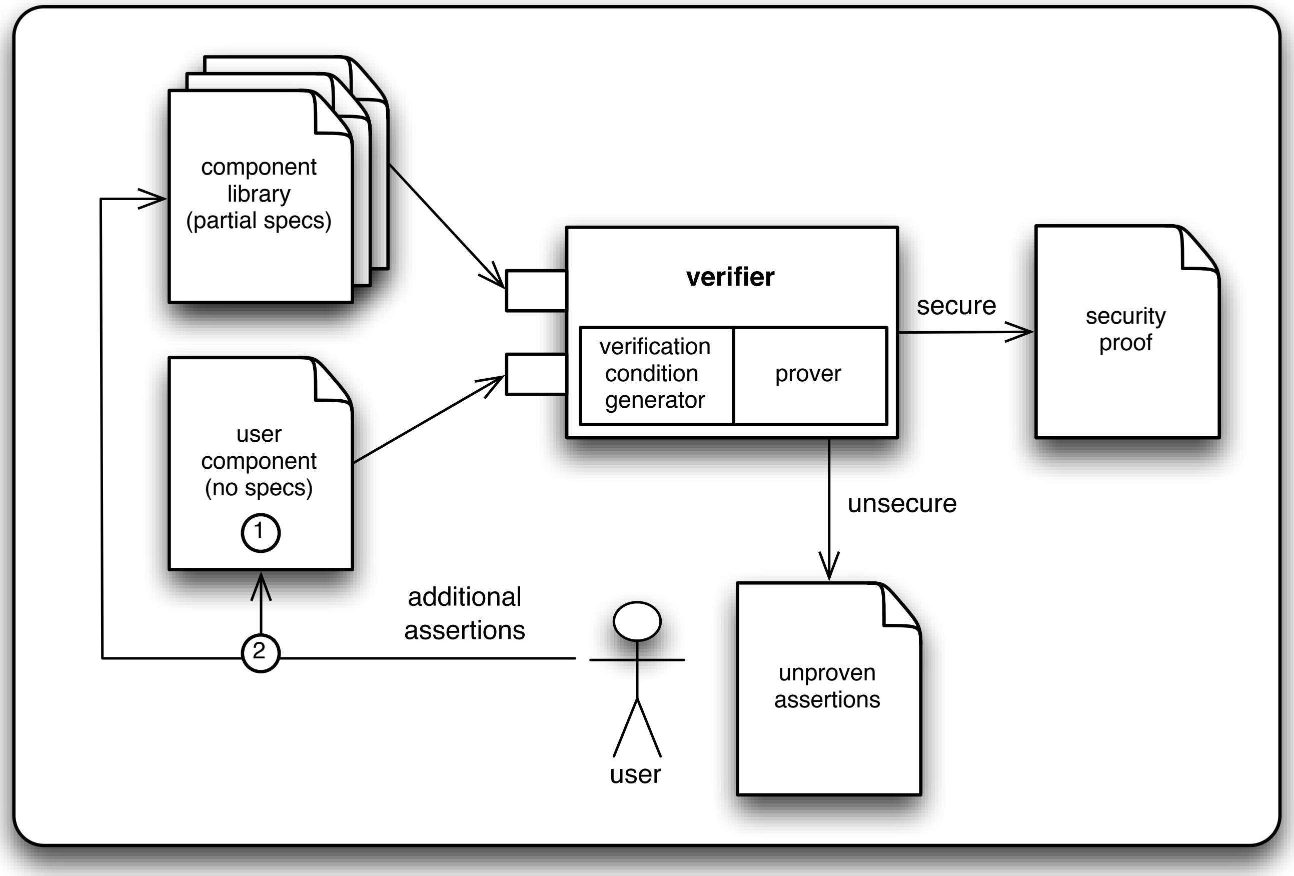
component library (partial specs)
verifier
verification condition generator
prover
secure
security proof
user component (no specs)
1
unsecure
additional assertions
2
user
unproven assertions

Figure 2: System Overview. *This figure provides an overview of our verification system.*

While the process is mechanical, the resulting set of assertions is likely to be cognitively burdensome, especially for developers who are not accustomed to mathematical specification and reasoning techniques. Hence, it is necessary to automate the proofs of these assertions if the approach is to be broadly applicable. For this automation step, we rely on the *Isabelle* prover [], a general-purpose proof assistant for use in mathematics. The assertions generated by the verification condition generator are translated to the Isabelle syntax. These assertions are then provided as input to Isabelle, which we run in *full automation* mode. While it is certainly true that Isabelle cannot, in a fully automated way, prove every assertion, the types of assertions that we're considering here are relatively basic. Isabelle is particularly well-suited to handling simple arithmetic expressions and conditions on integer elements — the cornerstone of integer range analysis, and ultimately, elimination of buffer overflow vulnerabilities.

There are two possible outcomes. If the verifier is able to prove the correctness of the generated assertions, the developer's component implementation is guaranteed to be free of buffer overflow vulnerabilities. In this case, there is no additional specification or verification work to be completed. The verified component is simply used in the application of interest, and is likely added to the library of proven components for later use. In addition to providing a positive response, the prover can yield the corresponding Isabelle proof, which we refer to as a *security proof*, illustrated on the right side of the figure. This proof may be stored as part of the documentation repository.

The second possible outcome of the verification process is the negative case, where the verification system is unable to generate a security proof. Or, stated more precisely, this case involves one or more generated assertions that could not be proven correct. These unproven assertions are presented to the user using standard mathematical syntax. The user's task is to determine why the assertions could not be proven. Consider a single unproven assertion for operation $O()$. There are again two cases to consider. In the first case, the assertion is legitimately false; it represents a pre-condition violation, or a violation of $O()$'s post-condition. In the second case, the assertion cannot be proven because the antecedent is too weak. Or stated otherwise, the verification condition generator was unable to generate a sufficiently strong antecedent because one or more operations invoked from within $O()$ are under-specified. In this case, the verification system can highlight the unproven assertions and the particular operations involved in those assertions. This gives the user an opportunity to incrementally enhance their specifications, both for the user component and any referenced library components. This step in the process is illustrated in the figure using a circled "2." Although not shown in the figure, this process repeats iteratively until the component is proven correct, or is shown to contain a buffer overflow vulnerability.

It is worth emphasizing that the incremental nature of this process is key to our approach. Over time, the component library will become more fully specified. As a result, when using the verifier, less effort will be required since fewer iterations through the verification process will be necessary. In effect, the system allows developers to amortize the (significant) expense of specification and verification effort across *all* of the systems they develop.

Full behavior verification, which is ultimately necessary to ensure complete security, requires full behavioral specification. Even as researchers continue their efforts on that front [**?**, ?, ?], our goal in this paper is to apply similar methods towards achieving a more modest goal: To detect a class of common errors that arise from buffer overflows in secure software development. Such overflows may occur either in accessing statically-allocated arrays or dynamically-allocated heap elements. In our current approach, we consider only array accesses; but the approach is fully extensible to dynamically-allocated memory. An array access operation (e.g., `get`, `set`) has an implicit pre-condition that the location of access is within array bounds. Unfortunately, checking that the bounds hold at run-time is both expensive and too late in the process. Our approach is to specify the pre-conditions explicitly and prove, using verification, that every array access is within bounds, statically. Such verification, in general, may not be trivial because nested expressions are often used to determine array access locations. However, it is much simpler than the general verification problem, because simple invariants for loops, for example, may be sufficient to detect such violations. Achieving this goal requires an initial specification of all operations on standard types used in programming languages, such as array accesses and operations to manipulate integers. Further specification can be incorporated in the process as necessary, for example, when an array is indexed with a user-defined function.

Proving the absence of overflow vulnerabilities in a dynamic memory context is a more challenging problem, but it can be handled using a similar approach in which standard operations on pointer variables are specified formally and used in verification.

3. CONCLUSION

We have described a verification system for ensuring the absence of buffer overflow vulnerabilities in component-based software. The system relies on existing mathematical formalisms for specifying software correctness and existing tools for automating the verification process — in particular, the Isabelle prover []. Unlike previous approaches to handling the buffer overflow problem, our approach is provably sound and complete, while at the same time supporting amortization of specification effort over the lifetime of the component library being developed. We believe our approach can safeguard system security in the presence of one of the most pernicious and frequent software attacks.

Acknowledgments

This work is supported by the National Science Foundation through awards DUE-0633506, CNS-0745846, DMS-0701187, and CCF-0811748.

4. REFERENCES

[1] *The National Strategy to Secure Cyberspace*. February 2003.

[2] *The National Vulnerability Database*. December 2008.

[3] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for tinyos. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 205–218, New York, NY, USA, 2007. ACM.

[4] D. E. Denning. Is cyber terror next?, November 2001.

[5] E. Haugh and M. Bishop. Testing c programs for buffer overflow vulnerabilities. In *Network and Distributed System Security Symposium (NDSS*, 2003.

[6] G. Helmer. *Incomplete list of Unix vulnerabilities.*

[7] M. Sitaraman, D. P. Gandi, W. Küchlin, C. Sinz, and B. B. Weide. DEET for component-based software. In *Proceedings of the 2004 SAVCBS Workshop, ACM SIGSOFT 2004/FSE-12*, pages 95–104, Newport Beach, CA, Oct. 2004.

[8] A. C. Trembly. The next terrorist attack: Coming soon to a computer screen near you? October 2001.

[9] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[10] C. Wilson. Computer attack and cyber terrorism: Vulnerabilities and policy issues for congress. 2003.