Finding Security Vulnerabilities Before Evil Doers Do

David Evans
evans@cs.virginia.edu
University of Virginia, Department of Computer Science
Charlottesville, Virginia USA

Abstract

Most security attacks exploit instances of well-known classes of implementation flaws. Many of these flaws could be detected and eliminated before software is deployed. This paper describes open source tools that programmers can use to identify likely security vulnerabilities in programs before they are released.

An analysis of any vulnerability database quickly reveals that most software vulnerabilities are not the result of clever attackers discovering new classes of software flaws. Instead, the vast preponderance of vulnerabilities stem from repetitive instances of well-known problems. An analysis of entries in the Common Vulnerabilities and Exposures database found that found that 29% of reported vulnerabilities involved buffer overflows or string format flaws [EL02]. Wagner et. al., found that buffer overflow vulnerabilities account for approximately 50% of CERT advisories [WFBA00].

Even conscientious programmers can overlook security issues, especially when security issues rely on undocumented assumptions about procedures and datatypes.

Tools that detect these vulnerabilities automatically will find problems that even the most vigilant and dedicated programmers would overlook.

1. Approaches

More promising approaches for reducing the damage caused by software flaws can be grouped into two categories – mitigate the damage flaws can cause or eliminate some of the flaws before the software is deployed. Techniques that limit the damage software flaws may cause include modifying program binaries to insert run-time checks or running applications in restricted environments that limit what they may do. In addition, several projects have developed safe libraries [BST00] and compiler modifications [CP98] specifically for addressing classes of buffer overflow vulnerabilities. These approaches all reduce the risk of security vulnerabilities while requiring only minimal extra work from application developers. Damage-limitation approaches do not eliminate the flaw, but replace it with a denial-of-service vulnerability since it is usually not possible to recover from a detected problem without terminating the program.

Techniques for detecting and correcting software flaws include human code reviews, testing, and static analysis. Human code reviews are time-consuming and expensive, but can find the types of conceptual problems it would be impossible to find automatically. They are likely to miss, however, more mundane problems that even extraordinarily thorough people will overlook. Testing is necessary, but typically not effective in finding security vulnerabilities.

Static analysis tools analyze the source code directly which enables them to make claims about all possible executions of a program instead of just the particular execution observed in a test case. From a security viewpoint, this is a significant advantage. There is a wide range of static analysis techniques, offering a tradeoff between the effort required to use them and the complexity of the analyses they are able to perform. Standard compilers perform type checking and other simple program analyses. At the other extreme, full program verifiers attempt to prove complex properties about programs. They typically require a complete formal specification and use automated theorem provers. These techniques are nearly always too expensive and cumbersome to use on even security-critical programs.

The simplest way to detect security vulnerabilities is to produce a warning whenever a potentially dangerous library functions is used. For example, the `gets` function is *always* vulnerable to a buffer overflow attacks. Other library functions, such as `strcpy` may be used safely, but are often the source of buffer overflow vulnerabilities. Several tools report uses of dangerous functions including Flawfinder [W03] and the Rough Auditing Tool for Security (RATS) [SS01].

2. Splint

Splint, the successor to LCLint, is an open source, lightweight static analysis tool for ANSI C. that fits between traditional compilers and program verifiers. It is designed to be as fast and easy to use as a compiler. It is able to do checking no compiler can do, however, by exploiting annotations added to libraries and

programs that document assumptions and intents. Splint checks that source code is consistent with the properties implied by annotations.

Annotations are denoted using stylized C comments identified by a `@` character following the `/*` comment marker. Annotations can be associated syntactically with function parameters and results, global variables and structure fields. For example, the annotation `/*@notnull@*/` can be used in a pointer declaration syntactically like a type qualifier. In a parameter declaration, the notnull annotation documents an assumption that the value passed for this parameter is not `NULL`. Splint would report a warning for any call site where the actual parameter might be `NULL`. Failure to handle possible `NULL` values (especially those returned by memory allocation procedures) can be exploited in denial of service attacks, and is often not detected in normal testing.

Splint provides annotations for documenting many different properties including ownership of storage, sizes of allocated storages, information hiding and aliasing. Programmers can also design their own annotations and associated checking rules to enforce properties. If parameters and return values are annotated with information about the sizes of allocated objects, Splint can detect many types of buffer overflow vulnerabilities.

There are both theoretical and practical limits on what can be analyzed statically. Precise analysis of most interesting properties of arbitrary C programs depends on several undecidable problems including reachability and determining possible aliases. Since our goal is to do as much useful checking as possible, we choose to

allow checking that is both unsound and incomplete. This means Splint produces both false positives and false negatives. Warnings are intended to be as useful as possible to the programmer, but there is no guarantee that all messages indicate real bugs or that all bugs will be found. Splint provides many options that make it possible for users to configure checking to suppress particular messages and weaken or strengthen checking assumptions.

Using Splint is an iterative process. Running Splint produces warnings that lead to either changes in the code or annotations. Then, Splint is run again to check the changes and propagate the newly documented assumptions. This process continues until no warnings are produced. Since Splint checks approximately 1000 lines per second, the need to re-run it is not burdensome.

Splint (and its precursor, LCLint) has been used to detect a range of problems not specifically focused on security including data hiding [EGHT94]; memory leaks, uses of dead storage, and null dereferences [Eva96] on programs comprising hundreds of thousands of lines of code. We have used Splint to detect both known and previously unknown buffer overflow vulnerabilities in wu-ftpd, a popular ftp server, and BIND, libraries and tools that comprise the reference implementation of DNS (see [LE01] for details), and for checking wu-ftpd for format bugs (see [EL02] for details).

3. Conclusion

The vast majority of security attacks exploit vulnerabilities in software that are well understood and can be eliminated. Lightweight static analysis is a promising

technique for detecting likely vulnerabilities so they can be fixed before software is deployed, not patched after attackers have exploited them. Several open source tools are available that can be used to improve the quality of programs before they are released.

No tool will eliminate all security risks – but lightweight static analysis should increasingly become part of the development process for security-sensitive applications.

References

[BST00] Arash Baratloo, Navjot Singh and Timothy Tsai. *Transparent Run-Time Defense Against Stack-Smashing Attacks*. 9th USENIX Security Symposium, August 2000.

[CP98] Crispin Cowan, Calton Pu, David Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle and Qian Zhang. *Automatic Detection and Prevention of Buffer-Overflow Attacks*. 7th USENIX Security Symposium, January 1998.

[EGHT94] David Evans, John Guttag, Jim Horning and Yang Meng Tan. *LCLint: A Tool for Using Specifications to Check Code*. SIGSOFT Symposium on the Foundations of Software Engineering. December 1994.

[Eva96] David Evans. *Static Detection of Dynamic Memory Errors*. SIGPLAN Conference on Programming Language Design and Implementation. May 1996.

[EL02] David Evans and David Larochelle. *Improving Security Using Extensible Lightweight Static Analysis*. In IEEE Software, Jan/Feb 2002.

[LE01] David Larochelle and David Evans. *Statically Detecting Likely Buffer Overflow Vulnerabilities*. 10th USENIX Security Symposium, August 2001.

[SS01] Secure Software Solutions. *Secure Software Announces Initial Release of RATS*. http://www.securesw.com/rats/. May 2001.

[WFBA00] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. *A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities*. Network and Distributed System Security Symposium. February 2000.

[W03] David Wheeler. Flawfinder Home Page. http://www.dwheeler.com/flawfinder/. 2003.