Unifying Disparate Tools in Software Security

Greg Morrisett
Harvard University

How can you trust software? In particular, when you install a piece of code, such as a video game or a device driver for a new camera, how can you ensure that the code won't do something bad when executed, such as deleting all of your files or installing a key-stroke logger that captures your passwords? How can you ensure that the software doesn't contain coding bugs or logic errors that might leave a security hole?

Traditional approaches to software security have assumed that users could easily determine when they were installing code, and whether or not software was trustworthy for a particular context. This assumption was reasonable when computers controlled few things of real value, when only a small number of people (typically experts) installed software, and the software itself was relatively small and simple. But the security landscape has been drastically altered by technology advances, such as the explosive growth of the Internet, the increasing size and complexity of software, and new business practices, such as out-sourcing and open source development. Furthermore, as we grow more reliant on software to control critical systems, from phone systems and airplanes to banks and armies, we desperately need new mechanisms to ensure software is truly trustworthy.

The Attacker

Today, malicious hackers have a serious financial incentive to break into and control your personal and business computers. If they can break into your machine,

then they can gather personal information, such as passwords, credit cards, and social security numbers and use this information to clean out bank accounts, apply for credit in your name, or gain access to other machines on the same network. Furthermore, if attackers can break into and control machines, then they can use them to send "spam" (unsolicited email advertisements), to store illicit digital goods (e.g., pirated music or pornography), and to launch denial-of-service attacks against other computing systems.

One technique that an attacker can use to gain a foothold on a machine is to trick the user into thinking a piece of code is trustworthy. For example, an attacker may send an email that contains a malicious program, but with a forged return address from someone the user trusts. Alternatively, they may set up a web site with a name that appears to be trustworthy, such as "www.micr0s0ft.com" (where the letter "o" is replaced with the numeral "0") and send an email with a link to the sight, suggesting that the user needs to download and install a new security update.

Even sophisticated users can be fooled: A clever attacker might contribute "Trojan Horse" code to a popular open source project, making an otherwise useful program (e.g., a compiler [cite Thompson lecture]) into undetected mal-ware. In short, it is relatively easy for an attacker to gain the trust of a user and get them to install code.

Today, we use a combination of digital signatures and virus scanners to try to stop these attacks. But digital signatures only tell us who produced the code, not whether it is trustworthy. Because virus scanners operate at the syntactic level, looking for snippets of previously identified mal-ware, they are easily defeated through automated obfuscation techniques. And of course, such scanners will never detect new attacks.

What we have today are relatively weak tools that tell us something about the *provenance* of the software, and its *syntax*, when what we require is a tool to validate the software's *semantics.*

Even Good Guys Can't Be Trusted

A second technique attackers use to gain control of a machine is to find a bug in an already installed program that communicates with the outside world. The classic example is a *buffer overrun* in some service, such as a login daemon, or some network-based application, such as a web-browser. A buffer overrun occurs when a program allocates *n* bytes of memory to hold an input (e.g., a password), but the attacker provides more than *n* bytes. A properly written program will check and reject any input that is too long. Unfortunately, many programmers fail to insert appropriate checks, in part because commonly used programming languages (C and C++) make it easy to forget those checks, and in part because programmers often think, "no one will have a password of more than a thousand characters."

When programmers fail to put in the check and the input is too long, the extra bytes overwrite whatever data happened to be stored next to the input buffer. In many situations, the buffer is allocated on the control stack near a return address for a procedure---a location where the program will transfer control after the input routine is complete. A clever attacker will enter an input long enough to overwrite the return address with a new value, thereby controlling what code will be executed when the input routine finishes. In the ideal attack, the rest of the input contains executable instructions

and the attacker causes control to transfer to this newly injected code. In this fashion, the attacker can cause the program to execute arbitrary code.[1]

Attacks based on buffer overruns are surprisingly prevalent, and at one point, accounted for over half of the security vulnerabilities reported by the Computer Emergency Response Team [cite CERT statistics]. But the failure to check input lengths is only one of a large number of coding errors that attackers can use to gain a foothold on a machine or extract private information. Other examples include integer overflows, format string attacks, script injection attacks, race conditions, use of bad pseudo-random number generators, and improper use of cryptographic primitives.

Thus, even well meaning software vendors cannot seem to produce trustworthy code. Part of the problem is that to sell new versions of the software, a vendor must produce new features, which simply adds complexity to the code base. Part of the problem is that current development practices, including design, coding, review, and testing are not adequate to rule out simple errors, such as buffer overruns, much less deeper problems such as the inappropriate use of cryptographic primitives or covert information channels.

Preventing Bugs through Rewriting

One approach to defending against attacks is to build tools that automatically insert checks at program locations where a policy violation might occur. For example, we might insert extra code at each buffer update to ensure that we do not write beyond its bounds.

---

[1] For a more detailed explanation of these attacks, see http://www.phrack.org/archives/49/P49-14

Compilers for high-level, type-safe languages, such as Java and C#, already insert checks to ensure that a wide class of language-level errors, including buffer overruns, are prevented. Unfortunately, the vast majority of systems and application software, including security-critical operating systems code, is still written in C or C++ where the compiler does not have enough information to insert appropriate checks. It is simply too expensive to rewrite the millions of lines of code that make up big software systems (e.g., Windows or Oracle) in a high-level language, and doing so is likely to introduce as many bugs as it eliminates. Furthermore, certain services, such as device drivers and real-time embedded software, need control over memory layout and timing, and are thus not suited to high-level languages.

Some effective tools for rewriting low-level legacy code have started to emerge. For example, the StackGuard tool [Cow98] and the Microsoft "/gs" compiler option, rewrite C/C++ code to insert a secret "cookie" next to buffers allocated on the control-flow stack, and to check that the cookie has been preserved when a procedure is about to jump to a return address. In this fashion, a buffer overrun can often be detected and stopped with relatively low overhead. Unfortunately, the approach does not protect against other forms of attack, such as overflowing a buffer allocated on the heap.

Another example of automated rewriting is the Ccured compiler [Nec02]. Ccured provides a strong type-safety guarantee by inserting extra meta-data and run-time checks into C code. The meta-data makes it possible to determine for instance, the size of a buffer at run-time, and the checks ensure that *all* buffer boundaries and typing constraints are satisfied. Compared to StackGuard, the overheads for Ccured can be greater, but the security guarantees are much stronger. However, StackGuard can be applied to almost

any C program without change, whereas Ccured requires a number of changes to the source code. In practice, a tool that is easily used gets used first.

There are many other tools that try to enforce security policies on legacy code through rewriting, including software-based fault isolation [Wah93,Mcc06], control-flow isolation [Abd05], and in-lined reference monitors [Sch00], among others. Each of these tools strikes a different balance with respect to the expressiveness of the policies that can be enforced, the code base to which the techniques are applicable, and the run-time overheads introduced.

Preventing Bugs through Static Analysis

Security-conscious companies are starting to use better tools that either prevent or detect a wide-class of coding errors at compile time. For example, Microsoft uses a static analysis tool called Prefast (based on an early tool called Prefix [Bus00]) that scans C and C++ code, looking for common errors such as buffer overruns. Companies such as Fortify and Coverity produce similar tools that can analyze software written in a variety of languages, and that support customizable rule-sets for finding new classes of bugs as they arise.

To a first approximation, a static analysis tool attempts to symbolically execute all paths in the code, looking for potential bugs. Of course, a program with n conditional statements can have 2^n distinct paths, and a program with loops or recursion can have an unbounded number of paths, making this naïve approach infeasible. Furthermore, the analysis does not know the actual values of inputs to the program, so it cannot always determine the values of variables or data structures.

Consequently, these tools construct models of the program that abstract details, and reduce the domains of reasoning to something finite. For example, instead of tracking the actual values integer variables might take on, an analysis might only track upper and lower bounds. To ensure termination, the analysis might only consider a few iterations of a loop. More sophisticated techniques, based on abstract interpretation [cite abstract interpretation] make it possible to determine an approximation of the behavior for all iterations.

Though automated static analysis has made tremendous strides, especially in the past few years, it still suffers from a number of problems. One problem, introduced by the necessary approximation, is that we may indicate a potential error where there is none (i.e., a false positive), or may fail to report an error (i.e., a false negative). If the approximations are constructed so that the analysis is *sound* (i.e., no false negatives), then unfortunately, programmers tend to be flooded with false positives, making it difficult to find and fix the real bugs in the program. Therefore, today, few of these analyses are sound and thus there is the potential that a bug will sneak through. For example, a recently-found buffer overrun in the Windows Vista custom cursor animation code went undetected by Prefast.

Looking to the Future:

One problem with static analysis tools is that, like an optimizing compiler, they tend to be large, complicated programs. Thus, a bug in the analysis can mean that a security-relevant error goes undetected. Furthermore, most tools operate on source code (or bytecode) and thus the tool must make assumptions about how the compiler will

translate the program to machine code. Consequently, an inconsistent assumption or a bug in the compiler can result in a security hole.

An ideal static analysis would:

- operate at the machine-code level (to avoid having to reason about the compiler),
- be small, simple, and ideally, formally verified,
- be sound (i.e., not let bugs go un-reported), and
- be complete (i.e., not suggest code has bugs when it does not).

Although these goals seem unattainable, the *proof-carrying code* (PCC) architecture, suggested by Necula and Lee [Nec97], comes remarkably close to satisfying them.

The idea behind PCC is to require that executable programs come equipped with a formal, machine-checkable "proof" that the code does not have bugs. By "proof", we mean a formal, logical argument that the code, when executed, will not violate a pre-specified (and formalized) policy on behavior. The key insight is that constructing a proof is hard (generally undecidable), but verifying a proof is easy. After all, a proof is meant to be mechanically checkable evidence. Furthermore, assuming we have a sound and *relatively* complete logic for reasoning about program behavior, we can, in principle, accept precisely those programs that are *provably* safe.

The really nice thing about PCC is that as a framework, it is applicable to the user, who is worried about whether or not it is safe to install a program downloaded from the Internet, as well as the software producer, who has out-sourced development of modules.

No longer are we dependent on *who* produced the code, but rather what the code will actually do.

In practice, the problem with PCC is that someone, namely the code producer, still has to come up with the proof that the code satisfies a given policy. Thus, none of the hard problems really go away. We still must use other techniques, such as static analysis and rewriting, to find an effective way to construct the proof. What changes is that we no longer need to trust the tools or compiler. In this fashion, PCC helps to minimize the size of the trusted computing base.

One way to construct a proof that machine code satisfies a policy is through the use of a *proof-preserving compiler*. In particular, if we start with source code and a proof that the source satisfies a policy, then a proof-preserving compiler can simultaneously transform the code and proof to the machine-code level. *Type-preserving compilers* [Nec98,Mor99,Col00] are a particular instance of this framework, where the policy is restricted to a form of type-safety.

Next-generation programming languages are incorporating increasingly sophisticated type systems that allow programmers to capture stronger safety and security properties through automated type checking. For example, the Jif dialect of Java lets programmers specify secrecy requirements on data, and the type system ensures that secret inputs (i.e., private fields) cannot flow to public outputs [Mye99]. Thus, when combined with a type-preserving compiler, the Jif type system makes it possible for a third party to verify that a program will not disclose information that is intended to be private.

At an extreme, it is possible to capture arbitrary policies with a *dependent type system*, as found in emerging languages such as ATS [Xi03], Concoqtion [Fog07], Epigram [McB04], and HTT [Nan06]. These languages make it possible to specify everything from simple typing properties up to full correctness. The price, of course, is that type checking is only semi-automatic. Simple properties can be discharged using a combination of static analysis, constraint solving, and automated theorem proving, but ultimately, programmers are required to construct explicit proofs for deeper properties. To a large degree, the success of these languages will be determined by how much can truly be automated. Nevertheless, for safety and security-critical software systems, these languages, in conjunction with proof-preserving compilers and the proof-carrying code architecture provide a compelling research vision.

References

[Abd05] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. *In Proceedings of the 12th ACM Conference on Computer and Communications Security* (CCS'05), Alexandria, VA, November 2005.

[Bus00] William Bush and Jonathan Pincus and David Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Software – Practice and Experience* 30(7):775-802, June 2000.

[Col00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Ken Cline, Mark Plesko. A Certifying Compiler for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, British Columbia, Canada, June 2000.

[Cou77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.

In Proceedings of the Symposium on Principles of Programming Languages, pages 238--252, Los Angeles, California, January 1977.

[Cow98] 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 *Proceedings of the 7th Usenix Security Symposium*, San Antonio, Texas, January 1998.

[Fog07] Seth Fogarty, Emir Pasalic, Jeremy Siek, and Walid Taha. Concoqtion: indexed types now! In *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 112-121, Nice, France, January 2007.

[McB04] Connor McBride. Practical Programming with Dependent Types. In Varmo Vene, Tarmo Uustalu (Eds.): Advanced Functional Programming, 5th International School, AFP 2004, Tartu, Estonia, August 14-21, 2004, Revised Lectures. pages 130-170, Published as *Lecture Notes in Computer Science* 3622 Springer 2005, ISBN 3-540-28540-7.

[McC06] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC Architecture. *In Proceedings of the 15th Usenix Security Symposium,* Vancouver, British Columbia, August 2006.

[Mor99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems,* 21(3):527-568, 1999.

[Mye99] Andrew C. Myers. Practical Mostly-Static Information Flow Control. *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 228-241, San Antonio, Texas, January 1999.

[Nan06] Aleksander Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and Separation in Hoare Type Theory. In *Proceedings of the ACM International Conference on Functional Programming*, pages 62-73, Portland, Oregon, September 2006.

[Nec97] George C. Necula. Proof-Carrying Code . In *Proceedings of the ACM International Symposium on Principles of Programming Languages*, Paris, France, January 1997

[Nec98] George C. Necula and Peter Lee. The Design and Implementation of a Certifying Compiler. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Montreal, Canada, 1998.

[Nec02] George C. Necula, Scott McPeak, Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. *In Proceedings of the ACM Symposium on Principles of Programming Languages*, London, January 2002.

[Sch00] Fred B. Schneider. Enforceable Security Policies. *In ACM Transactions on Information Systems Security*, 3(1), p. 30--50, 2000.

[Wah93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Software-Based Fault Isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203-216, 1993.

[Xi03] Hongwei Xi, Applied Type Systems (extended abstract), in *TYPES'03*, pages 394-408, Published as *Lecture Notes in Computer Science* 3085, Springer-Verlag, 2004.