CHRIS WYSOPAL

putting trust in software code

Chris Wysopal is director of development at Symantec Corporation, where he leads research on how to build and test software for security vulnerabilities.

chris_wysopal@symantec.com

TWENTY YEARS AGO, KEN THOMPSON, the co-father of UNIX, wrote a paper about the quandary of not being able to trust code that you didn't create yourself. The paper, "Reflections on Trusting Trust,"[1] details a novel approach to attacking a system. Thompson inserts a back door into the UNIX login program when it is compiled and shows how the compiler can do this in a way that can't be detected by auditing the compiler source code. He writes:

"You can't trust code that you did not totally create yourself. No amount of source-level verification or scrutiny will protect you from using untrusted code. In demonstrating the possibility of this kind of attack, I picked on the C compiler. I could have picked on any program-handling program such as an assembler, a loader, or even hardware microcode."

Twenty years after Thompson's seminal paper was published, developments in the field of automated binary analysis of executable code are tackling the problem of trusting code you didn't write. Binary analysis can take on a range of techniques, from building call trees and looking for external function calls to full decompilation and modeling of a program's control flow and data flow. The latter, which I call deep binary analysis, works by reading the executable machine code and building a language-neutral representation of the program's behavior.

This model can be traversed by automated scans to find security vulnerabilities caused by coding errors and to find many simple back doors. A source code emitter can then take the model and generate a human-readable source code representation of the program's behavior. This enables manual code auditing for design-level security issues and subtle back doors that will typically escape automated scans.

The steps of the decompilation process are as follows:

1. Front end decodes binary to intermediate language.
2. Data flow transformer reconstructs variable lifetimes and type information.
3. Control flow transformer reconstructs loops, conditionals, and exceptions.
4. Back end performs language-specific transformation and exports high-level code.

To be useful the model must have a query engine that can answer questions for security scanning scripts:

- What is the range of a variable?
- Under what conditions is some code reachable? Any path, all paths?
- What dangerous actions does this program perform?

Security scanning scripts can then ask questions such as:

- Can the source string buffer size of a particular unbounded string copy be larger than the destination buffer size?
- Was the return value from a security-critical function call tested for success before acting on the results of the function call?
- Is untrusted user input used to create the file name passed to a create-file function?

By building meaning from the individual instructions that are executed by the CPU, deep binary analysis understands program behavior that is inserted by the compiler. Thompson's back-door code can't hide from the CPU, and it can't hide from deep binary analysis. More important for real-world software security is that vulnerabilities and back doors can't hide in the static and dynamic libraries that a program links to and for which source code is not always available.

The programmer productivity benefits of using off-the-shelf software components are well known, but not much is said about the risks of using binary components, which are common on closed source operating systems. When developing enterprise applications, programmers frequently concentrate on writing business logic and leave presentation, parsing, transaction processing, encryption, and much more to commercial libraries for which they often have no source code. These libraries typically come from OS vendors, database vendors, transaction-processing vendors, and development framework vendors. Often the programmers dutifully audit the 20% of the application they wrote and ignore the 80% they cannot audit. Yet it is the entire program that is exposing the organization running it to risk.

Deep binary analysis can follow the data flows between the main program and the libraries in use to find issues that arise from the interaction between the main program and a library function. Often any security vulnerabilities discovered can be worked around by putting additional constraints on the data passed to the library function. Sometimes, however, the function call will need to be replaced. Using the tools we have developed at @stake (now Symantec), we have found buffer overflows in the string functions of a popular C++ class library and poor randomness being used in a cryptolibrary function. The cryptolibrary function was using a random number generated by `rand()`, and `srand()` was seeded with zero. No doubt there was a comment in the C code stating that the random number generation needed to be replaced in the future, but this being binary analysis we couldn't tell.

An interesting use of binary analysis is the differential analysis of two binaries that have small differences between them. Perhaps you are engaged in incident response and have discovered an altered binary on the system for which the attacker has not left any source code behind. Differential binary analysis can be used to see what behavior has been added or removed from the binary. A use that has important security implications is to reverse engineer the details of a vulnerability by determining the differences between a vulnerable program and one that has a vendor security patch applied. If black hats can easily determine the cause of a vulnerability simply by looking at the patch, then there is little to be gained from vendors withholding vulnerability details, and there is increased urgency to patch vulnerable systems quickly.

Halvar Flake has developed tools for binary diffing and gave a presentation on his techniques at Black Hat Windows 2004.[2] Todd Sabin has developed different techniques for differential binary analysis which he calls "Comparing Binaries with Graph Isomorphisms."[3] The field is rapidly evolving, so as with so many topics in the security arena, defenders are urged to stay apprised of developments, because attackers surely will.

I am hopeful that the field of binary analysis will evolve to a point where third-party testing labs will be able to perform repeatable, consistent security testing on closed source software products without the cooperation of software vendors. Much as *Consumer Reports* is able to verify automobile vendor claims of performance and carry out their own safety testing, a software testing lab would be able to quantify the number of buffer overflows, race conditions, script injections, and other issues in a program under automated test. A security quality score could be generated from the raw test results. It will undoubtedly be imperfect—there will always be issues missed and some false positives—but on a coarse scale, say, ranking program security from A to F, it would be very useful to consumers.

Today there is little useful information by which to rate software security quality except for a particular product's security patch record. This record is somewhat useful but typically only exists for the most popular products, which garner the bulk of the attention of security researchers. Another source of security information is common criteria evaluations. But with products that have received common criteria EAL 4 certification still being subject to monthly patches of critical severity, there is clearly a need for additional ways of rating security quality.

Deep binary analysis stands to revolutionize the software security space not only for developers and businesses but for consumers, too. It's an exciting future, one in which we don't have to trust the compiler manufacturers, third-party driver and library providers, or application and operating system vendors. Software developers can use binary analysis tools to discover and remediate the vulnerabilities in their own software, and consumers can verify that their vendors have performed due diligence and are not delivering shoddy code.

REFERENCES

1. Ken Thompson, "Reflections on Trusting Trust," Communications of the ACM, vol. 27, no. 8 (August 1984), reprinted at http://www.acm.org/classics/sep95.

2. Halvar Flake, "Automated Binary Reverse Engineering": http://www.blackhat.com/presentations/win-usa-04/bh-win-04-flake.pdf.

3. Todd Sabin, "Comparing Binaries with Graph Isomorphisms": http://www.bindview.com/Support/RAZOR/Papers/2004/comparing_binaries.cfm.