

Software Security for Open-Source Systems

Debate over whether open-source software development leads to more or less secure software has raged for years. Neither is intrinsically correct: open-source software gives both attackers and defenders greater power over system security. Fortunately, several security-enhancing technologies for open-source systems can help defenders improve their security.



CRISPIN
COWAN
WireX
Communications

Some people have claimed that open-source software is intrinsically more secure than closed source,¹ and others have claimed that it's not.² Neither case is absolutely true: they are essentially flip sides of the same coin. Open source gives both attackers and defenders greater analytic power to do something about software vulnerabilities. If the defender does nothing about security, though, open source just gives that advantage away to the attacker.

However, open source also offers great advantages to the defender, giving access to security techniques that are normally infeasible with closed-source software. Closed source forces users to accept the level of security diligence that the vendor chooses to provide, whereas open source lets users (or other collectives of people) raise the bar on security as high as they want to push it.

This article surveys security enhancements that take advantage of the nature of open-source software. We'll concentrate on *software* security (mitigation of vulnerabilities in software), not *network* security (which is dealt with in terms of line protocols and is thus unaffected by whether other network components are open source). All the solutions that we'll consider apply to open-source systems, but they may not be entirely open source themselves.

Software security is fundamentally simple: just run perfect software. Being that perfect software is infeasible for non-trivial systems, we must find other means to ascertain that large, complex, probably vulnerable software does what it should do and *nothing else*. (See Ivan Arce's "Whoa, Please Back Up for One Second" at <http://online.securityfocus.com/archive/98/142495/2000-10-29/2000-11-04/2>.) We

classify methods that ensure and enforce the "nothing else" part into three broad categories:

- *Software auditing*, which prevents vulnerabilities by searching for them ahead of time, with or without automatic analysis
- *Vulnerability mitigation*, which are compile-time techniques that stop bugs at runtime
- *Behavior management*, which are operating system features that either limit potential damage or block specific behaviors known to be dangerous

Software auditing

The least damaging software vulnerability is the one that never happens. Thus, it is optimal if we can prevent vulnerabilities by auditing software for flaws from the start. Similarly, it is near optimal to audit existing applications for flaws and remove them before attackers discover and exploit them. Open source is ideal in this capacity, because it enables anyone to audit the source code at will and productively share the results of such audits with the world.

The problem with this approach is that auditing source code for correctness, or even for common security coding pathologies, is difficult and time-consuming. It follows that assuring a given piece of security-critical code has been audited, and by people competent enough to effectively detect vulnerabilities, is equally difficult.

The Sardonix project presented here was created to address the social problems of coaxing people to audit code and keep track of the results. Following are descrip-

Table 1. Software auditing tools.

TOOL	DOMAIN	EFFECT	RELEASE DATE	LICENSE	URL
BOON	C source analysis	Static source-code analysis to find buffer overflows	2002	BSD	www.cs.berkeley.edu/~daw/boon
CQual	C source analysis	Static type inference for C code to discover inconsistent usage of values	2001	GPL	www.cs.berkeley.edu/~jfofster/cqual
MOPS	C source analysis	Dynamically enforcing that C program conforms to a static model	2002	BSD	www.cs.berkeley.edu/~daw/mops
RATS	C, Perl, PHP, and Python source analysis	Uses both syntactic and semantic inspection of programs to find vulnerabilities	2001	GPL	www.securesw.com/rats
FlawFinder	C source analysis	Multiple syntactic checks for common C vulnerabilities	2001	GPL	www.dwheeler.com/flawfinder
Bunch	C source analysis	Program understanding and visualization to help software analyst understand program	1998	Closed-source freeware	http://serg.cs.drexel.edu/bunch
PScan	C source analysis	Static detection of printf format vulnerabilities	2002	GPL	www.striker.ottawa.on.ca/~aland/pscan
Sharefuzz	Binary program vulnerability detection	Stress-test programs looking for improperly checked inputs	2002	GPL	www.atstake.com/research/tools/index.html#vulnerability_scanning
ElectricFence	Dynamic memory debugger	Complains about various forms of malloc() and free() misuse	1998	GPL	http://perens.com/FreeSoftware
MemWatch	Dynamic memory debugger	Complains about various forms of malloc() and free() misuse	2000	Public domain	www.linkdata.se/sourcecode.html

tions of several static and dynamic software analysis tools (see Table 1 for a summary).

Sardonix

Sardonix.org provides an infrastructure that encourages the community to perform security inspection of open-source code and preserve the value of this effort by recording which code has been audited, by whom, and subsequent reports.

Sardonix measures auditor and program quality with a ranking system. The auditor ranking system measures quality by the volume of code audited and the number of vulnerabilities missed (as revealed by subsequent audits of the same code). Programs, in turn, are rated for trustworthiness in terms of who audited them. This ranking system encourages would-be auditors with something tangible to shoot for (raising their Sardonix rank) and use on their resumes.

Sardonix also helps novice auditors by providing a central repository of auditing resources—specifically, descriptions and links to auditing tools and how-to and FAQ documents.

Static analyzers

Static analyzers examine source code and complain about suspicious code sequences that could be vulnerable. Unlike compilers for “strongly typed” languages such as Java and ML, static analyzers are free to complain about code that might in fact be safe. However, the cost of exuberantly reporting mildly suspicious code is a high false-positive rate. If the static analyzer “cries wolf” too often, developers start treating it as an annoyance and don’t use it much. So *selectivity* is desirable in a source-code analyzer.

Conversely, *sensitivity* is also desirable in a source-code analyzer. If the analyzer misses some instances of the pathologies it seeks (false negatives), it just creates a false sense of confidence.

Thus we need *precision* (sensitivity plus selectivity) in a source-code analyzer. Unfortunately, for the weakly typed languages commonly used in open-source development (C, Perl, and so on), security vulnerability detection is often undecidable and in many cases requires exponential resources with respect to code size. Let’s look at some source-code analyzers that use various heuristics to function but that can never do a perfect job. Such tools are

Table 2. Vulnerability mitigation tools.

TOOL	DOMAIN	EFFECT	RELEASE DATE	LICENSE	URL
StackGuard	Protects C source programs	Programs halt when a buffer overflow attack is attempted	1998	GPL	http://immunix.org/stackguard.html
ProPolice	Protects C source programs	Programs halt when a buffer overflow attack is attempted	2000	GPL	www.trl.ibm.com/projects/security/ssp
FormatGuard	Protects C source programs	Programs halt when a <code>printf</code> format string attack is attempted	2000	GPL	http://immunix.org/formatguard.html

perfect for *assisting* a human in performing a source-code audit.

Berkeley. Researchers at the University of California at Berkeley have developed several static analysis tools to detect specific security flaws:

- BOON is a tool that automates the process of scanning for buffer overrun vulnerabilities in C source code using deep semantic analysis. It detects possible buffer overflow vulnerabilities by inferring values to be part of an implicit type with a particular buffer size.³
- CQual is a type-based analysis tool for finding bugs in C programs. It extends the type system of C with extra user-defined type qualifiers. Programmers annotate their program in a few places, and CQual performs qualifier inference to check whether the annotations are correct. Recently, CQual was adapted to check the consistency and completeness of Linux Security Module hooks in the Linux kernel.⁴ Researchers have also extended its use to type annotation to detect `printf` format vulnerabilities.⁵
- MOPS (MOdel checking Programs for Security) is a tool for finding security bugs in C programs and verifying their absence. It uses software model checking to see whether programs conform to a set of rules for defensive security programming; it is currently a research in progress.⁶

RATS. The Rough Auditing Tool for Security is a security-auditing utility for C, C++, Python, Perl, and PHP code. RATS scans source code, finding potentially dangerous function calls. This project's goal is not to find bugs definitively. Its goal is to provide a reasonable starting point for performing manual security audits. RATS uses an amalgam of security checks, from the syntactic checks in ITS4⁷ to the deep semantic checks for buffer overflows derived from MOPS.³ RATS is released under the GNU Public License (GPL).

FlawFinder. Similar to RATS, FlawFinder is a static source-code security scanner for C and C++ programs

that looks for commonly misused functions, ranks their risk (using information such as the parameters passed), and reports a list of potential vulnerabilities ranked by risk level. FlawFinder is free and open-source covered by the GPL.

Bunch. Bunch is a program-understanding and visualization tool that draws a program dependency graph to assist the auditor in understanding the program's modularity.

PScan. In June 2000, researchers discovered a major new class of vulnerabilities called "format bugs."⁸ The problem is that a `%n` format token exists for C's `printf` format strings that commands `printf` to write back the number of bytes formatted to the corresponding argument to `printf`, presuming that the corresponding argument exists and is of type `int *`. This becomes a security issue if a program lets unfiltered user input be passed directly as the first argument to `printf`.

This is a common vulnerability because of the (previously) widespread belief that format strings are harmless. As a result, researchers have discovered literally dozens of format bug vulnerabilities in common tools.⁹

The abstract cause for format bugs is that C's argument-passing conventions are type-unsafe. In particular, the `varargs` mechanism lets functions accept a variable number of arguments (such as `printf`) by "popping" as many arguments off the call stack as they wish, trusting the early arguments to indicate how many additional arguments are to be popped and of what type.

PScan scans C source files for problematic uses of `printf` style functions, looking for `printf` format string vulnerabilities. (See <http://plan9.hert.org/papers/format.htm>; www.securityfocus.com/archive/1/815656; and www.securityfocus.com/bid/1387 for examples.) Although narrow in scope, PScan is simple, fast, and fairly precise, but it can miss occurrences in which `printf`-like functions have been wrapped in user-defined macros.

Dynamic debuggers

Because many important security vulnerabilities are undecidable from static analysis, resorting to dynamic debugging often helps. Essentially, this means running the

program under test loads and seeing what it does. The following tools generate unusual but important test cases and instrument the program to get a more detailed report of what it did.

Sharefuzz. “Fuzz” is the notion of testing a program’s boundary conditions by presenting inputs that are likely to trigger crashes, especially buffer overflows and `printf` format string vulnerabilities. The general idea is to present inputs comprised of unusually long strings or strings containing `%n` and then look for the program to dump core.

Sharefuzz is a local `setuid` program fuzzer that automatically detects environment variable overflows in Unix systems. This tool can ensure that all necessary patches have been applied or used as a reverse engineering tool.

ElectricFence. ElectricFence is a `malloc()` debugger for Linux and Unix. It stops your program on the exact instruction that overruns or underruns a `malloc()` buffer.

MemWatch. MemWatch is a memory leak detection tool. Memory leaks are where the program `malloc()`’s some data, but never frees it. Assorted misuses of `malloc()` and `free()` (multiple `free`’s of the same memory, using memory after it has been freed, and so on) can lead to vulnerabilities with similar consequences to buffer overflows.

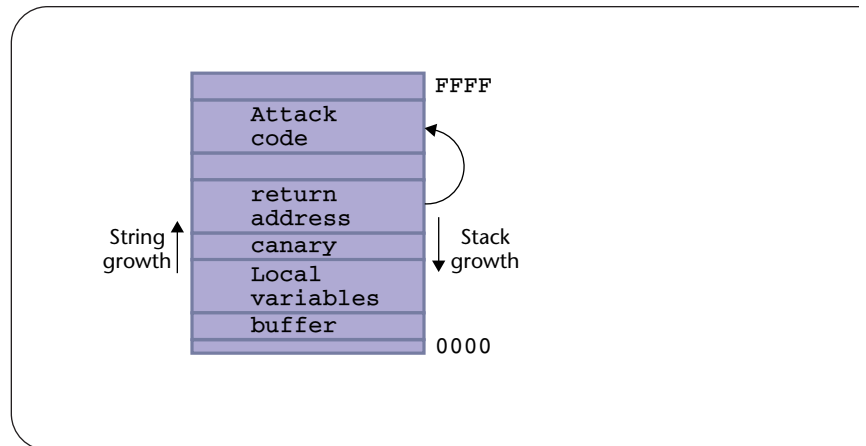
Vulnerability mitigation

All the software-auditing tools just described have a work factor of at least several hours to analyze even a modest-size program. This approach ceases to be feasible when faced with millions of lines of code, unless you’re contemplating a multiyear project involving many people.

A related approach—vulnerability mitigation—avoids the problems of work factor, precision, and decidability. It features tools (see Table 2 for a summary) that insert light instrumentation at compile time to detect the exploitation of security vulnerabilities at runtime. These tools are integrated into the compile tool chain, so programs can be compiled normally and come out protected. The work factor is normally close to zero. (Some tools provide vulnerability mitigation and also require the source code to be annotated with special symbols to improve precision.⁵ However, these tools have the unfortunate combination of the high work factor of source-code auditing tools and the late detection time of vulnerability mitigators.)

StackGuard

StackGuard appears to have been the first vulnerability mitigation tool.¹⁰ It is an enhancement to the GCC (the GNU Compiler Collection; <http://gcc.gnu.org>) C compiler that emits programs resistant to the “stack smashing”



variety of buffer overflows.¹¹

StackGuard detects stack-smashing buffer overflows in progress via integrity checks on the function calls’ activation records, introducing the “canary” method of integrity checking (see Figure 1). The compiler emits code that inserts canaries into activation records when functions are called and checks for them when those functions return. If a stack-smashing overflow occurs while the function is active, the canary will be smashed and the function return code will abort the program rather than jumping to the address indicated by the corrupted activation record.

StackGuard has been in wide use since summer 1998. Developers have used it to build complete Immunix Linux distributions based on Red Hat 5.2, 6.2, and 7.0. However, StackGuard 2 (the current release) is based on GCC 2.91.66, and, as of this writing, a port to GCC 3.2 is almost complete. StackGuard is released under the GPL.

ProPolice

ProPolice is an independent implementation similar to StackGuard. It adds several features, the most significant of which are

- Moving the canary. ProPolice places the canary between the activation record and the local variables, rather than in the middle of the activation record. StackGuard 3 will include this feature.
- Sorting variables. ProPolice sorts the variables in each function to put all the character arrays (strings) above other variables in memory layout. The goal is to prevent buffer overflows from corrupting adjacent variables. It has limited effectiveness, because ProPolice cannot move strings embedded within structures and because the attacker can still corrupt adjacent strings.

StackGuard injects instructions late in the GCC compiler’s RTL stage, taking great care to ensure that the compiler does not optimize away or otherwise defeat the canary checks. ProPolice, in contrast, injects the canary

Figure 1. The StackGuard defense against stack-smashing attack.

Table 3. Behavior management systems.

TOOL	DOMAIN	EFFECT	RELEASE DATE	LICENSE	URL
LSM	Linux 2.5 and 2.6 kernels	Enables behavior management modules to be loaded into standard Linux kernels	2002	GPL	http://lsm.immunix.org
Type Enforcement	Linux and BSD kernels	Map users to domains and files to types; mitigate domain/type access	1986	Subject to several patents; GPL	www.cs.wm.edu/~hallyn/dte
SELinux	Linux kernels supporting LSM modules	Type enforcement and role-based access control for Linux	2000	Subject to several patents; GPL	www.nsa.gov/selinux
SubDomain	Linux kernels	Confines programs to access only specified files	2000	Proprietary	www.immunix.org/subdomain.html
LIDS	Linux kernels	Critical files can only be modified by specified programs	2000	GPL	www.lids.org
Openwall	Linux 2.2 kernels	Prevents pathological behaviors	1997	GPL	www.openwall.com
Libsafe	glibc library	Plausibility checks on calls to common string manipulation functions	2000	LGPL	www.research.avayalabs.com/project/libsafe
RaceGuard	Linux kernels	Prevents temporary file race attacks	2001	GPL	http://immunix.org
Systrace	BSD kernels	Controls the system calls and arguments a process can issue	2002	BSD	www.citi.umich.edu/u/provos/systrace

checks into GCC's abstract syntax tree layer, introducing the risk that the compiler can disrupt the canary checks to the point of being ineffective.

FormatGuard

FormatGuard is similar to StackGuard in that it detects and halts exploitation of `printf` format string vulnerabilities in progress.¹² FormatGuard was the first vulnerability mitigation for `printf` format bugs. It uses CPP (C PreProcessor) macros to implement compile-time argument counting and a runtime wrapper around `printf` functions that match the expected number of arguments in the format string against the actual number of arguments presented. FormatGuard is available as a version of `glibc` under the LGPL.

Behavior management

Behavior management describes protections that function entirely at runtime, usually enforced by libraries or the operating system kernel. The Linux Security Modules project (LSM), presented here enables behavior management modules to be loaded into standard Linux 2.5 and 2.6 kernels. After that are some leading access control systems and some behavior management systems that are not exactly access control but that do provide effective security protection against various classes of software vulnerability. Table 3 summarizes the available behavior management tools.

LSM: Linux Security Modules

Linux's wide popularity and open-source code have made it a common target for advanced access control model research. However, advanced security systems remain out of reach for most people. Using them requires the ability to compile and install custom Linux kernels, a serious barrier to entry for users whose primary business is not Linux kernel development.

The Linux Security Modules (LSM) project¹³ was designed to address this problem by providing a common modular interface in the Linux kernel so that people could load advanced access control systems into standard Linux kernels; end users could then adopt advanced security systems as they see fit. To succeed, LSM must satisfy two goals:

- Be acceptable to the Linux mainstream. Linus Torvalds and his colleagues act as a de facto standards body for the Linux kernel. Adding an intrusive feature such as LSM requires their approval, which, in turn, requires LSM to be minimally intrusive to the kernel, imposing both small performance overhead and small source-code changes. LSM was designed from the outset to meet this goal.
- Be sufficient for diverse access controls. To be useful, LSM must enable a broad variety of security models to be implemented as LSM modules. The easy way to do this is to provide a rich and expressive application programming interface. Unfortunately, this directly con-

flicts with the goal of being minimally intrusive. Instead, LSM met this goal by merging and unifying the API needs of several different security projects.

In June 2002, Linus Torvalds agreed to accept LSM into Linux 2.5 (the current development kernel), so it will most likely be a standard feature of Linux 2.6 (the next scheduled production release).

Access controls

The Principle of Least Privilege¹⁴ states that each operation should be performed with the least amount of privilege required to perform that operation. Strictly adhered to, this principle optimally minimizes the risk of compromise due to vulnerable software. Unfortunately, strictly adhering to this principle is infeasible because the access controls themselves become too complex to manage. Instead, a compromise must be struck between complexity and expressability. The standard Unix permissions scheme is very biased toward simplicity and often is not expressive enough to specify desired least privileges.

Type enforcement and DTE. Type enforcement introduced the idea of abstracting users into domains, abstracting files into types, and managing access control in terms of which domains can access which types.¹⁵ DTE (Domain and Type Enforcement¹⁶) refined this concept. Serge Hallyn is writing an open-source reimplementation of DTE, ported to the LSM interface (see www.cs.wm.edu/hallyn/dte).

Intellectual property issues surrounding type enforcement are complex. Type enforcement and DTE are both subject to several patents, but implementations have also been distributed under the GPL. The ultimate status of these issues is still unclear.

SELinux. SELinux evolved from type enforcement at Secure Computing Corporation, the Flask kernel at the University of Utah, and is currently supported by NAI (<http://nai.com>) under funding from the US National Security Agency. SELinux incorporates a rich blend of security and access control features, including type enforcement and RBAC (Role-Based Access Control¹⁷). The SELinux team has been instrumental in the development of the LSM project and distributes SELinux exclusively as an LSM module.

SubDomain. SubDomain is access control streamlined for server appliances.¹⁸ It ensures that a server appliance does what it is supposed to and nothing else by enforcing rules that specify which files each program may read from, write to, and execute.

In contrast to systems such as DTE and SELinux, SubDomain trades expressiveness for simplicity. SELinux can express more sophisticated policies than SubDomain, and

should be used to solve complex multiuser access control problems. On the other hand, SubDomain is easy to manage and readily applicable. For instance, we entered an Immunix server (including SubDomain) in the Defcon Capture-the-Flag contest¹⁹ in which we wrapped SubDomain profiles around a broad variety of badly vulnerable software in a period of 10 hours. The resulting system was never penetrated. SubDomain is being ported to the LSM interface.

The Linux Intrusion Detection System. LIDS started out with an access model that would not let critical files be modified unless the process's controlling `tty` was the physical console. Because this severely limited anything other than basement computers, LIDS extended its design to let specified programs manipulate specified files similar to the SubDomain model. LIDS has been ported to the LSM interface.

Behavior blockers

Behavior blockers prevent the execution of certain specific behaviors that are known to (almost) always be associated with software attacks. Let's look at a variety of behavior-blocking techniques implemented for open-source systems.

Openwall. The Openwall project is a security-enhancing patch to the Linux kernel comprised of three behavior blockers:

- **Nonexecutable stack segment.** Legacy factors in the Intel x86 instruction set do not permit separate read and execute permissions to be applied to virtual memory pages, making it difficult for x86 operating systems to make data segments nonexecutable. Openwall cleverly uses the x86 segment registers (which do allow separate read and execute attributes) to remap the stack segment so that data on the stack cannot be executed.
- **Non-root may not hard link to a file it does not own.** This prevents one form of temporary file attack in which the attacker creates a link pointing to a sensitive file such that a privileged program might trample the file.
- **Root may not follow symbolic links.** This prevents another form of temporary file attack.

The latter two features have been ported to the LSM interface in the form of the OWLSM module. OWLSM—a pun on Openwall's abbreviation (OWL) and LSM—in turn has been augmented with a “no `ptrace` for `root` processes” policy, which defends against chronic bugs in the Linux kernel's `ptrace` handling.

Libsafe. Libsafe is a library wrapper around `glibc` standard functions that checks argument plausibility to prevent the `glibc` functions from being used to damage the calling program.¹⁹ Libsafe 2.0 can stop both

buffer overflows and `printf` format string vulnerabilities by halting the function if it appears to be about to overwrite the calling activation record. Its main limitation is that its protection is disabled if programs are compiled with the `-fomit-frame-pointer` switch, commonly used to give the GCC/x86 compiler one more register to allocate.

RaceGuard. Temporary file race attacks are where the attacker seeks to exploit sloppy temporary file creation by privileged (`setuid`) programs. In a common form of temporary file creation, a time gap exists between a program checking for a file's existence and the program actually writing to that file.²⁰ RaceGuard defends against this form of attack by transparently providing *atomic* detection and access to files—preventing the attacker from “racing” in between the read and the write.²¹ We provide efficient atomic access by using *optimistic locking*: we let both accesses go through but abort the second write access if it is mysteriously pointing to a different file than the first access.²² RaceGuard is being ported to the LSM interface.

Systrace. Systrace is a hybrid access control and behavior blocker for OpenBSD and NetBSD. Similar to SubDomain, it allows the administrator to specify which files each program can access. However, Systrace also lets the administrator specify which system calls a program can execute, allowing the administrator to enforce a form of behavior blocking.

Integrated systems

These tools all require some degree of effort to integrate into a system, ranging from RaceGuard and Openwall, which just drop in place and enforce security-enhancing policies, to the access control systems that require detailed configuration. Let's look at three products that integrate some of these tools into complete working systems.

OpenBSD

OpenBSD's core philosophy is the diligent application of manual best security practices (see www.openbsd.org). The entire code base went through a massive manual source-code audit. The default install enables few network services, thus minimizing potential system vulnerability in the face of random software vulnerabilities.

OpenBSD also features a `jail()` system, which is similar to the common `chroot()` confinement mechanism. More recently, Systrace was added to OpenBSD. The OpenBSD project also provided the open-source community with OpenSSH, an open-source version of the popular SSH protocol, recently upgrading OpenSSH to use “privilege separation” to minimize vulnerability due to bugs in the SSH daemon. OpenBSD is completely open-source software.

OWL: Openwall Linux

OWL is similar to OpenBSD in philosophy (audited source code and a minimal install/configuration) but is based on Linux instead of BSD. It uses the Openwall behavior blockers.

Immunix

To stay relatively current with fresh software releases, Immunix does not use any source-code auditing. Instead, it compiles all code with vulnerability mitigators (StackGuard and FormatGuard) and behavior management (SubDomain and RaceGuard) to block most vulnerabilities from being exploitable. WireX has an ongoing research program to develop new and improved software security technologies. Although many components are open source, the Immunix system is commercial software.

EnGarde

EnGarde is a commercial Linux distribution hardened with LIDS access controls.

Discussion points

All these tools were developed for or around open-source systems, but they are variably applicable to proprietary systems, requiring either access to application source code or modifications to the operating system kernel or libraries.

Vulnerability mitigation

Microsoft “independently innovated” the StackGuard feature for the Visual C++ 7.0 compiler.²³ In principle, this delivers the same protective value as in open-source systems, but in practice only the code's purveyors can apply the protection, because no one else has the source code to compile with.

Behavior managers

Several commercial vendors are now providing defenses marketed as “host intrusion prevention.” Two such vendors are Entercept (a behavior blocker that blocks a list of known vulnerabilities) and Okena (a profile-oriented mandatory access control system, similar to Systrace).

Here, open source's advantage is relatively weak: being able to read an application's source code can somewhat help in creating a profile for the application, but it is not critical. The main advantage of open source is that it is relatively easy for researchers and developers to add these kinds of features to open-source operating systems.

Open-source software presents both a threat and an opportunity with respect to system security. The threat is that the availability of source code helps the attacker create new exploits more quickly. The opportunity is that the available source code enables defenders to turn up their degree of security diligence arbitrarily high, independent of

vendor preferences.

The opportunity afforded by open-source systems to raise security arbitrarily high is revolutionary and not to be missed by organizations seeking high security. No longer locked into what a single vendor offers, users can choose to apply security-enhancing technologies to their systems or choose an open-source system vendor that integrates one or more of these technologies to produce higher-security systems that remain compatible with their unenhanced counterparts. □

Acknowledgments

DARPA contract F30602-01-C-0172 supported this work in part.

References

1. E.S. Raymond, *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O'Reilly & Assoc., 1999; www.oreilly.com/catalog/cb.
2. K. Brown, *Opening the Open Source Debate*, Alexis de Tocqueville Inst., 2002; www.adti.net/cgi-local/SoftCart.100.exe/online-store/scstore/p-brown_%1.html?L+scstore+llf68476ff0a810a+1042027622.
3. D. Wagner et al., "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," *Network and Distributed System Security*, 2000; www.isoc.org.
4. X. Zhang, A. Edwards, and T. Jaeger, "Using CQUAL for Static Analysis of Authorization Hook Placement," *Usenix Security Symp.*, Usenix Assoc., 2002, pp. 33–47.
5. U. Shankar et al., "Automated Detection of Format-String Vulnerabilities," *Usenix Security Symp.*, Usenix Assoc., 2001, pp. 201–216.
6. H. Chen and D. Wagner, "MOPS: An Infrastructure for Examining Security Properties of Software," *Proc. ACM Conf. Computer and Communications Security*, ACM Press, 2002.
7. J. Viega et al., "ITS4: A Static Vulnerability Scanner for C and C++ Code," *Ann. Computer Security Applications Conf. (ACSAC)*, Applied Computer Security Assoc., 2000; www.cigital.com/its4.
8. C. Cowan, "Format Bugs in Windows Code," *Vuln-dev mailing list*, 10 Sept. 2000; www.securityfocus.com/archive/82/81455.
9. S. Shankland, "Unix, Linux Computers Vulnerable to Damaging New Attacks," *Cnet*, 7 Sept. 2000; http://yahoo.cnet.com/news/0-1003-200-2719802.html?pt.yfin.cat_fin.txt.ne.
10. C. Cowan et al., "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *7th Usenix Security Conf.*, Usenix Assoc., 1998, pp. 63–77.
11. "Smashing the Stack for Fun and Profit," *Phrack*, vol. 7, Nov. 1996; www.phrack.org.
12. C. Cowan et al., "FormatGuard: Automatic Protection from printf Format String Vulnerabilities," *Usenix Security Symp.*, Usenix Assoc., 2001, pp. 191–199.
13. C. Wright et al., "Linux Security Modules: General Security Support for the Linux Kernel," *Usenix Security Symp.*, Usenix Assoc., 2002, pp. 17–31; <http://lsm.immunix.org>.
14. J.H. Saltzer and M.D. Schroeder, "The Protection of Information in Computer Systems," *Proc. IEEE*, vol. 63, Nov. 1975, pp. 1278–1308.
15. W.E. Bobert and R.Y. Kain, "A Practical Alternative to Hierarchical Integrity Policies," *Proc. 8th Nat'l Computer Security Conf.*, Nat'l Inst. Standards and Technology, 1985.
16. L. Badger et al., "Practical Domain and Type Enforcement for Unix," *Proc. IEEE Symp. Security and Privacy*, IEEE Press, 1995.
17. D.F. Ferraiolo and R. Kuhn, "Role-Based Access Control," *Proc. 15th Nat'l Computer Security Conf.*, Nat'l Inst. Standards and Technology, 1992.
18. C. Cowan et al., "SubDomain: Parsimonious Server Security," *Usenix 14th Systems Administration Conf. (LISA)*, Usenix Assoc., 2000, pp. 355–367.
19. A. Baratloo, N. Singh, and T. Tsai, "Transparent Run-Time Defense Against Stack Smashing Attacks," *2000 Usenix Ann. Technical Conf.*, Usenix Assoc., 2000, pp. 257–262.
20. M. Bishop and M. Digler, "Checking for Race Conditions in File Accesses," *Computing Systems*, vol. 9, Spring 1996, pp. 131–152; <http://olympus.cs.ucdavis.edu/bishop/scriv/index.html>.
21. C. Cowan et al., "RaceGuard: Kernel Protection from Temporary File Race Vulnerabilities," *Usenix Security Symp.*, Usenix Assoc., 2001, pp. 165–172.
22. C. Cowan and H. Lutfiyya, "A Wait-Free Algorithm for Optimistic Programming: HOPE Realized," *16th Int'l Conf. Distributed Computing Systems (ICDCS'96)*, IEEE Press, 1996, pp. 484–493.
23. B. Bray, *How Visual C++ .Net Can Prevent Buffer Overruns*, Microsoft, 2001.

Crispin Cowan is cofounder and chief scientist of WireX. His research interests include making systems more secure without breaking compatibility or compromising performance. He has coauthored 34 refereed publications, including those describing the StackGuard compiler for defending against buffer overflow attacks. He has a PhD in computer science from the University of Western Ontario. Contact him at WireX, 920 SW 3rd Ave., Ste. 100, Portland, OR 97204; crispin@wirex.com; <http://wirex.com/~crispin>.

**Members
save
25%**

on all conferences sponsored
by the IEEE Computer Society.

Not a member? Join online today!

computer.org/join/