## **Programming for Security\***

## M. E. Kabay, PhD, CISSP Assoc. Prof. Information Assurance Program Director, MSIA Division of Business and Management, Norwich University

Gregory E. Borter, Systems Coordinator of Silver Springs Alfa SmartParks, Inc., wrote to me with an interesting series of questions:

Message text written by "Gregory E. Borter"

>I've been reading about security problems with the various OS components, both Windows and Linux, and the problems with security with applications software. Where is the best place to start implementing system security?<

It seems to me that security should be integrated into the requirements analysis, the design stage for software, the operating system security kernel, in corporate policy development and in human awareness, training and education programs.

>Should security start with the computer programming languages themselves, or their support libraries?<

PASCAL uses strong typing and requires full definition of data structures, thus making it harder to access data and code outside the virtual machine defined for a given process. In contrast, C and C++ allow programmers to access any region of memory at any time the operating system permits it.

There are several sets of security utilities available for programmers; for example, RSA has a number of cryptographic toolkits. Some textbooks (e.g., Schneier's \_Advanced Cryptography\_) include CD-ROMs with sample code.

>Are there any computer languages that have security features built-in to the language itself?<

Not to my knowledge, but I'm not an expert in languages.

>With so many PCs linked via networks and the Internet, shouldn't all programs be coded with the assumption that the programs will be operating in an environment where they may very probably be subject to hostile attack?<

Yes, but the difficulty in testing for security is that there are so many possible ways to generate security holes in code. Buffer overflows, for example, are the most common form of security

<sup>\*</sup> An earlier version of this article was published in five parts in Network World Fusion in 2001.

exploit, but clearly the programmers never thought to impose length restrictions on the input strings being handled by Web server software.

>Do any current computer programming languages give programmers tools with which to implement security best practices in their code?<

All computer languages allow you to write code as well as you can <smile>. I think that strongly-typed languages may offer better constraints on programmers, but the essential issue is that the programmers continue to think about security as they design and implement code. Java does include provisions for limiting access to resources outside the "sandbox" reserved for a process, as described in the books by Felten and McGraw.

>Is there any such thing as security best practices for computer programmers?<

In a sense, though not, as far as I know, in any codified form. There are recommendations on security-related aspects of programming in most general security textbooks; see for example Stallings.

In addition to designing security into a system from the start, I can think of some obvious guidelines that can apply:

- Impose strong identification and authentication for critical and sensitive systems in addition to the I&A available from the operating-system; ideally, use token-based or biometric authentication as part of the initialization phase of your application.
- Document your code thoroughly, including using data dictionaries for full definition of allowable input and output to functions and allowable range and type of values for all variables.
- Use local variables, not global variables, when storing sensitive data that should be used only within a specific routine; i.e., use the architecture of the process stack to limit inadvertent or unauthorized access to data in the stack.
- Re-initialize temporary storage immediately after the last legitimate use for the variable, thus making scavenging harder for malefactors.
- Limit functionality in a specific module to what is required for a specific job; e.g., don't use the same module for supervisory functions and also for routine functions carried out by clerical staff.
- Define views of data in databases that conform to functional requirements and limit access to sensitive data; e.g., the view of data from a medical-records database should exclude patient identifiers when the database is being used for statistical aggregation by a worker in the finance department.

- Use strong encryption (NOT home-grown encryption) that has industry-standard routines to safeguard sensitive and critical data on disk. Locally developed, home-grown encryption is generally NOT as safe.
- Disallow access by programmers to production databases.
- Randomize or otherwise mask sensitive data when generating test subsets from production data.
- Use test-coverage monitors to verify that all sections of source code are in fact exercised during quality assurance tests; investigate the functions of code that never gets executed.
- Integrate logging capability into all applications for debugging work, for data recovery after crashes in the middle of a transaction, and also for security purposes such as forensic analysis.
- Create log-file records that include a cryptographically-sound message authentication code (MAC) that itself includes the MAC of the preceding record as input for the algorithm; this technique ensures that forging a log file or modifying it will be more difficult for a malefactor.
- Log all process initiations for a program and log process termination; include full details of who loaded the program or module.
- Log all modifications to records and optionally provide logging for read-access as well.
- Use record-level locking to prevent inadvertent overwriting of data on records that are accessed concurrently. Be sure to unlock a sequence of locks in the inverse order of the lock sequence to prevent deadlocks (thus if you lock resource A, B and C in that order, unlock C, then B, then A).
- Sign your source code using digital signatures.
- Use checksums in production executables to make unauthorized modifications more difficult to conceal.

My friend and colleague (and former manager at AtomicTangerine) Mike Gerdes, contributed the following suggestions and comments.

• Might I suggest that you recommend the readers adopt a practice of designing code in a more holistic fashion? A common practice is to write and test routines in a way that verifies the code processes the data in the way intended. To avoid the effects of malicious code and data input attacks, the programmer must also write code which

deals with what is NOT supposed to be processed. A more complete design methodology would also include testing of all inbound information to ensure exclusion of any data which did not fit the requirements for acceptable data. This method should be applied to high risk applications and those with an extremely arduous test cycle and will eliminate many of the common attack methods used today.

- Establish the criteria for determining the sensitivity level of information contained in, or processed by the application and subroutines.
- If they are not already present, consider implementing formal control procedures in the software programming methodology to ensure all data is reviewed during QA processes to be sure it is classified and handled appropriately for the level assigned.
- Identify and include any mandatory operating system and network security characteristics for the production system in the specifications of the software. In addition to providing the development and QA teams some definition of the environment the software is designed to run in, giving the administrator and end users an idea of what your expectations were when you created the code can be extremely useful in determining where software can, or cannot, be used.
- Where appropriate, verify the digital signatures of routines that process sensitive data when the code is being loaded for execution.
- If you include checksums on executables for production code, include routines which verify the checksums at every system restart.

Finally, Reader Sasha Romanosky of Morgan Stanley sent me a stimulating letter as a followup to the series on programming and security; he has very kindly allowed me to share it with readers. The following is an edited version of his original letter.

I came across your articles on programming for security and thought of an additional resource your readers. Recently, SecurityPortal published a review by Razvan Peteanu < <u>http://securityportal.com/articles/designpatterns20010611.html</u> > of a paper entitled, "Architectural Patterns for Enabling Application Security," by Joseph Yoder and Jeffrey Barcalow (1998) < <u>http://www.joeyoder.com/papers/patterns/Security/appsec.pdf</u> > (also available in MS-Word, RTF and PostScript from Yoder's Web site at < <u>http://www.joeyoder.com/papers/patterns/</u>>).

The authors took the premise of OO design patterns and applied it to security. They introduced the following patterns:

• Single Access Point: preventing back doors by forcing a single entry point to code.

- Check Point: Organizing security checks and the repercussions of security violations.
- Roles: Organizing role-based security to define security privileges for different job functions.
- Session: Localizing global information about users, their privileges, resources in use and application states (e.g., locking).
- Limited View: Allowing users to see only the functions and fields that they can access.
- Full View with Errors: Showing users a full view of all functions fields (but not contents) with disabled functions and inaccessible fields clearly marked.
- Secure Access Layer: Integrating application security with low-level security such as encryption, firewalls, and authentication methods.

The paper excited me because it seemed like a great way to organize the concepts and practices that should make up a good application-security policy. One takes security existing or desired practices and formulates them into security patterns. When one needs to implement a new application, host or network, one can quickly identify the security patterns from this collection of best-practice implementations and apply them to the new application design.

Collecting and formalizing known security principles in this way is of great value in developing and applying good security measures.

Typically, security measures seem to focus on network security and rarely tackle security at the application level. The authors, I believe, attempt to fill this gap. I'll note, however, that many of these patterns can (and happily do) apply to both network and applications.

In addition to my thanks to Mike Gerdes for the ideas included above, I thank our friend and colleague Edwin Blackwell, also formerly at AtomicTangerine, for his helpful comments on the original text of this article

## **For Further Reading**

Felten, E. & G. McGraw (1999). Securing Java: Getting down to business with mobile code. John Wiley & Sons (New York). Also free and unlimited Web access from <a href="http://www.securingjava.com">http://www.securingjava.com</a>

McGraw, G. & E. W. Felten (1997). *Java Security: Hostile Applets, Holes and Antidotes --What Every Netscape and Internet Explorer User Needs to Know.* Wiley (New York). ISBN 0-471-17842-X. xii + 192. Index.

RSA Data Security < <u>http://www.rsasecurity.com/products/</u> >

Schneier, B. (1995). *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition.* John Wiley & Sons (New York). Hardcover, ISBN 0-471-12845-7, \$69.95; Softcover, ISBN 0-471-11709-9. xviii + 618. Index.

Stallings, W. (1995). *Network and Internetwork Security: Principles and Practice*. Prentice Hall (Englewood Cliffs, NJ). ISBN 0-02-415483-0. xiii + 462. Index.