Wireless Applications: Middleware Security

Mourad Debbabi, Mar-André Laverdière, Azzam Mourad and Syrin Tlili
Computer Security Laboratory, CIISE
Concordia University, Montreal, Canada

Keywords: *(5-10 keywords)*

Definition: *Wireless*

Abstract
In this chapter, we survey the security of wireless applications and middleware. Nowadays, we are, more and more, relying on the use of mobile and wireless systems such as PDAs, cell-phones, pagers, etc., for communication, work and entertainment. The popularity of these devices is increasing day after day. These devices now offer increased bandwidth and processing power, so that great variety of applications can be executed on them. Alongside, the security issues typically associated with wired computing have now migrated to the mobile systems' world. Many middleware frameworks (e.g. J2ME), that have been adapted to respect the resource limitations of such type of devices, offer high level security features. In this context, we examined the security objectives of applications and middleware, the vulnerabilities they face, the detection techniques used to uncover them and the hardening practices performed to remedy them.

1 Introduction
As traditional software met the world of wireless systems, and that they started offering networking capabilities that enabled features previously available only to wired systems, they brought security vulnerabilities along that threaten both devices and networks. These systems have proven vulnerable software-related attacks common to PCs [36, 39]. BlueSnarfing, for instance, is a method using buffer overflows on certain phones allowing to steal information remotely. Furthermore, high-level security, such as the principle of least privilege, is lacking on handheld computers and cell phones, giving full capabilities to any piece of code executing on it, malicious or not.

The security of wireless communication itself has been the focus of a lot of study, which resulted in strong medium security technologies [23] that are now used in the industry, some security concerns remain in the integration of mobile systems into traditional network infrastructure such as the Internet [43, 55]. Middleware frameworks are of great help, as they offer interoperable features that often have been built with security in mind. Other useful features offered by such technologies include fault tolerance, standardized communication mechanisms, adaptability, scalability and resource sharing. They can allow developers to create richer applications and systems in a shorter time, while not sacrificing security to do so. Although middleware is mostly known in the field of distributed system in classical environments with technologies such as CORBA, .NET and J2EE, it is nevertheless present in mobile devices via lightweight variants, such as increasingly popular J2ME.

Overall, mobile and wireless systems of all kinds are now facing the same security challenges as in traditional computing environments, and must solve them by taking in consideration limited resources. Traditional protection methods, such as firewalls and execution monitors, are not always realistic options due to such resource constraints or unfitting system architecture. Furthermore, the traditional code-exploit-fix cycle is unrealistic for devices and systems that rarely offer automatic updating. Economic pressures force shorter time-to-market constraints on

product development and force the reuse of existing software. In short, the realities of mobile systems require security improvements of the operating system, middleware and applications both before and after deployment. The process used must be simple, systematic, rapid and effective, in order to be used in depth during development and maintenance.

We refer to such improvements as security hardening and will show how vulnerabilities can be automatically detected both statically and dynamically, and how security engineers are able to specify security hardening plans to remedy both low-level and high-level security weaknesses.

In this chapter, we will examine how software vulnerabilities can be found and corrected. We will look at the related work in Section 2, and then briefly survey software vulnerabilities in Section 3. Afterwards, we will study automatic detection of low-level security vulnerabilities in Section 4. We then move on to Section 5 to determine how existing mobile applications and middleware can be hardened in a rigorous and systematic manner in response to the discovered vulnerability. We finally bring concluding remarks.

2 Related Works

Traditional middleware has been shown to be imperfect solutions when directly ported to wireless systems, and significant research has been invested in adapting those frameworks for the mobile world, as well as into building new solutions that were more adapted to platform requirements for both nomadic and ad-hoc systems [40]. With such improvements, middleware has been shown to be useful in enabling new ways to interact with enterprise information systems [55]. Adaptation middleware (e.g. Odyssey, Mobiware and Puppeteer), mobile agent systems (e.g. Telescript), service discovery (e.g. UPnP, Jini, SLP, Salutation and Ninja), tuple spaces (e.g. Lime and L2imbo), data replication (e.g. Xmiddle), Object-request brokers (e.g. LW-IOP) and transaction support (e.g. Kangaroo) are some of the middleware services for mobile systems in the literature [4, 40].

The study of middleware security has typically been related to the enforcement of access control in distributed systems [10, 24], although smart card support via middleware has been demonstrated [54]. Some systems offer more security services, such as .NET and J2EE that both offer a five-prong approach to security: code-based access control, role-based access control, secure code verification and execution, secure communication mechanisms and secure code and data protection [24].

The field of both low and high level security vulnerabilities has been extensively studied for classical applications, and taxonomies have emerged [7, 8, 16, 13, 19, 34]. Explanations of low-level vulnerabilities abound [60, 46, 57, 34], and a few authors detail how to exploit them ([18, 5, 30, 51] to name a few).

Secure programming has also been studied especially for C and C++ programs, a technology popular for both traditional and embedded systems. The authors [15, 46, 57, 25, 58, 34, 14, 53] usually explain what kind of code creates security vulnerabilities and how such vulnerabilities should be prevented by better coding.

Many techniques external to flawed programs have been developed. The reader will find many libraries for C/C++, some specifically compensating C memory manipulation vulnerabilities [56, 9], compiler extensions [35, 6], as well as operating system extensions that complicate the exploitation vulnerabilities [11, 59, 41].

To complement the previous contributions, authors have developed automatic tools to statically detect software vulnerabilities, mostly for C/C++ programs [49, 52, 28, 37, 20, 45, 22, 62, 26, 27].

Others developed tools that perform code instrumentation to detect security vulnerabilities at runtime [33, 47].

3 Security Requirements and Vulnerabilities

High-level security features are of tremendous importance for a sustainable mobile environment. Sadly, the implementation of software is never free of errors, and some lower-level errors can cause vulnerabilities that defeat the existence of security mechanisms. Each middleware has its own models, paradigms and techniques that provide its security capabilities and services. In this context, the authentication, authorization, confidentiality, integrity and non-repudiation are mainly the high level security requirements that are enforced by the security components of the major wireless middleware such as J2ME, CORBA, .NET etc. In the sequel, we present the definition of these requirements provided by the ISO standard 7498-2 [2]:

- Authentication. Corroboration of the identity of an entity or source of information.
- Authorization. Restricting access to resources to privileged entities.
- Confidentiality. Keeping information secret from all but those who are authorized to see it.
- Integrity. Ensuring information has not been altered by unauthorized or unknown means.
- Non-Repudiation. Preventing the denial of previous commitments or actions.

Other requirements exist in the security literature such as availability, anonymity, auditing, certification, privacy, revocation, timestamping, etc.

Many vulnerabilities could be created during the development phase of the software allowing security threats that can compromise the security capabilities of the applications and middleware. Flaws in the security models or in any of their components could be the cause of such vulnerabilities. For instance, if an application is using a particular API or protocol implementation in order to provide one of the aforementioned requirements, and this API or protocol contains some flaws, then the application as a whole will be considered as non-secure. Implementation flaws may lead to dangerous security vulnerabilities that are called low level security or safety vulnerabilities. Their exploitation is considered as dangerous threats because they can affect high level security capabilities provided by the applications and middleware.

The low level security vulnerabilities are extremely dependent on the programming language and platform. In this context, the C/C++ programming language has a bad reputation in the security world because it was designed for maximal performance, at the expense of some safety-enhancing techniques. Its memory management left to the programmer and the lack of type safety are the major causes of security flaws. Such flaws do not exist in many other technologies, such as Java applications and middleware because it offers better built-in security options: type verification and garbage collection in the virtual machine. In the sequel, we list the major safety vulnerabilities that are introduced in the source code during the implementation and we discuss their impacts on the applications and middle security:

3.1 Buffer Overflow

Buffer Overflows (BoF) exploit common programming errors that arise mostly from weak or nonexistent bounds checking of input being stored in memory buffers. Attacks that exploit these vulnerabilities are considered as one of the most dangerous security threats since it can compromise the integrity, confidentiality and availability of the target system often via code injection. Buffers on both the stack and the heap can be corrupted [60, 34]. The following are the common causes of buffer overflows: boundary conditions errors, input validation errors, assumption of Null-termination and improper format string.

3.2 Integer Overflow

Integer security issues arise on the conversion (either implicit or explicit) of integers from one type to another, and because of their inherently limited range [46]. C compilers distinguish between signed and unsigned integer types and silently perform operations such as implicit casting, integer promotion, integer truncation, overflows and underflows. Such silent operations are typically overlooked, which can cause various security vulnerabilities. Integer vulnerabilities may be used to write to an unauthorized area of memory. A first instance is the allocation of less memory than thought, allowing writing to unwanted parts of the heap. Another instance is to access invalid memory areas with a negative index or memory copying operation. In some cases, if the access is to an invalid page, the result will be a denial of service via an application crash. They can also cause other security problems by bypassing preconditions and expected protocol values that are specific to the program exploited. The following are the common causes of these vulnerabilities: converting between signed and unsigned, signedness errors, truncation errors, overflow and underflow.

3.3 Memory Management

The C programming language allows programmers to dynamically allocate memory for objects during program execution. C memory management is an enormous source of safety and security problems. The programmer is in charge of pointer management, buffer dimensions, allocation and de-allocation of dynamic memory space. Thus, memory management functions must be used with precaution in order to prevent memory corruption, unauthorized access to memory space, buffer overflow, etc. The following are the major errors caused by improper memory management in C: using un-initialized memory, accessing freed memory, freeing unallocated memory and memory leaks.

3.4 File Management

File management problems occur when access or modification of a restricted file happens. Some problems are closely related to race conditions. File Management errors can lead to many security vulnerabilities such as data disclosure, data corruption, code injection and denial of service. The following are two major sources of vulnerabilities in file management: Unsafe Temporary File and Improper File Creation Access Control Flags [57].

4 Security Evaluations

The main security mechanisms to consider for applications and middleware security are access control, authentication, message protection and audit. So far, the focus is on security policies and security protocols. However, there are many issues related to the implementation of these mechanisms. Growing applications and middleware security requirements have raised the stakes on software security. Building a secure software focuses on techniques and methodologies of designing and implementing a software in order to avoid exploitable security vulnerabilities. The C language is considered the de facto standard for system programming. Most of the existent middleware are written in C. These include Core ORB, J2ME, and HPCM. C fulfills performance, flexibility, strong support and portability requirements for these middleware. However, security features are either absent or badly supported in C programming. Lack of type safety and memory management left to the programmer's discretion are source of many critical security vulnerabilities such as buffer overflows and format strings. These vulnerabilities enable an intruder to take a complete control over the target machine and to circumvent all deployed security mechanisms for middleware. Despite the huge availability of books and documents that guide programmers in writing secure code, implementation errors still exist in C source code. Therefore, automated tools for vulnerabilities detection are very helpful for programmers in detecting and fixing errors in source code. There are different techniques and approaches for verification and validation of software security requirements:

4.1 Security Code Inspection

Code inspection as introduced by Michael Fagan [29], is widely recognized as an effective technique for finding software defects and bugs. It is carried out by a technically competent team of four persons: moderator, the designer, the coder and the tester. The moderator, experienced software engineer but preferably not involved in the project, is the key person and plays the role of the coach in the inspection process. The Fagan inspection process is based on the following five steps: (1) Overview: the designer describes what he has designed to all the participants. (2) Preparation: Each member, using the work documentation, individually tries to understand the design, its intent and logic. (3) Inspection: All the participants get together as a group and attempt to find as many defects as possible. (4) Rework: The designer or the coder resolves all the errors or problems noted in the inspection report. (5) Follow-up: The moderator checks the quality of the work and determines if the component needs to be re-inspected.

4.2 Static Analysis

The source code is examined statically without any execution of the source code [42]. The security properties that hold during static analysis are supposed to hold true for all executions of the analyzed source code. The objective of static analysis is to detect ahead of time the vulnerabilities in the code. There are four main approaches for static analysis: abstract interpretation, type systems, flow analysis and model checking. Each of these approaches has its advantages 11 and its drawbacks. The abstract interpretation has a strong formal semantics foundation with more than 30 years old. PolySpace [45] and AbsInt [3] are the two prominent commercial tools based on abstract interpretation. These tools can detect run-time errors that others static analysis tools fail to. PolySpace has served many clients in Telecommunications industries such as France Telecom, Siemens, LG Electronics and Samsung. Coverity [22] is another commercial tool that uses flow based static analysis to detect security vulnerabilities in source code. Flow analysis does not have strong formal foundation as abstract interpretation but it does scale to large programs. Telecommunication industries such as NOKIA, NOMADIX and ShoreTel have used Coverity tools to verify the security of their software. There are many other academic tools that can be used to assess the security of open source middleware. Some of these tools focus on C pointers and type management for being the main causes of security violations. We cite a representative set of these tools: CCured [31], Cyclone [32], and Fail-Safe [61]. Their approach is based on type based static analysis. In other words, the standard C type is extended with additional type annotation and type check to tackle statically at compile time execution errors. Most of these tools resort to dynamic analysis by inserting run-time checks when it is not possible to decide statically about the safety of an operation in the program. Type based analysis has also been used to check authorization and authentication properties which are the main security requirements of middleware. In fact, Cqual [48] has been used to check the placement of security hooks in Linux Security Modules that are used to enforce access control security policies. The intent is to make sure that each security critical operation is performed after being authorized by the security policy. Hence, Cqual can also be used the implementation of access control policies in middleware. Model checking is also an appealing approach for checking security properties of middleware. There are many success stories of verifying protocols by model checking. Therefore, the approach can easily be adopted for checking the safe behavior of authentication, access control and cryptographic protocols of middleware. Microsoft uses SLAM model checker to check temporal properties of its drivers and interfaces. Since middleware are acting as an interface between application and protocol stack, it is very useful to consider using a tool such SLAM [50] to ensure a safe behavior of the middleware. There is also many open source model checker that can be used to check temporal properties for C source code. The most stable model checker for that purpose is MOPS [21].

4.3 Dynamic Analysis

The source code is examined during its execution. Dynamic analysis is done through code instrumentation to collect information on the program as it runs. In contrast to static analysis, the properties derived from dynamic analysis hold for the current execution of the analyzed program and may not be generalized for all other possible executions. Software testing is the most

common approach for dynamic analysis. Basically, the underlying principle of software testing is to exercise the system/software under test using a subset of its input domain, called, test cases, in order to unveil the defects and bugs. An appealing approach is to use aspect oriented program to insert check into the source code. The complexity and efficiency varies with respect to the nature (centralized, distributed) of the application under test. As for static analysis, there are many tools that use dynamic analysis to check security properties of software. Parasoft offers a wide range of testing framework that can automatically detect errors in source code. Parasoft SOAtest is intended for Service Oriented Architectures which are the essence of middleware. Parasoft [44] has been used by IBM to check security and interoperability features of their web services. Aspect Oriented Programming (AOP) can be considered as a promising approach for middleware. The company nearInfinity uses AOP for auditing J2EE middleware and many API such as Servlet, JNDI and EJB.

5 Security Hardening

Security hardening is a relatively unknown term in the current literature and, as such, we first provide a definition for it. We also propose taxonomy of security hardening methods that refer to area to which the solution is applied. We established our taxonomy by studying the solutions of software security problems in the literature. Even though our reading included a significant bias towards C [14, 46, 34, 57], we believe that our taxonomy is language independent. We also investigated the security engineering of applications at different levels, including specification and design issues [12, 17, 34]. From this information on how to correctly build new programs, and some hardening advice existing in the literature, we were able to draw out a classification for software hardening methodologies.

We define software security hardening as any process, methodology, product or combination thereof that is used to increase the security of existing software. In this context, the following constitutes the detailed classification of security hardening methods:

- Code-Level Hardening Changes in the source code in a way that prevents vulnerabilities without altering the design.
- Software Process Hardening Replacement of the development tools, the use of stronger implementation of libraries and the use of code weaving tools in a way that does not change the original code.
- Design-Level Hardening Reengineering of the application in order to integrate security features that were absent or insufficient.
- Operating Environment Hardening Improvements to the security of the execution context (network, operating systems, libraries, utilities, etc.) that is relied upon by the software.

Code-Level Hardening

Code-Level Hardening is the closest topic to what has been covered extensively in the literature. Code level hardening constitutes of removing the programming-related vulnerabilities in a systematic way by implementing the proper coding standards that were not enforced originally.

Software Process Hardening

Software Process Hardening refers to the inclusion of hardening practices within the software development process, notably on the matter of choosing appropriate platforms, statically-linked libraries, compilers, etc. that result in increased security. It is possible that a more secure implementation of the same library is used, instead of modifying the underlying source code. This will allow externalizing the security issues, keeping the solution independent of the using software, maximizing code reuse and facilitating the hardening of multiple applications relying

on this library. It is also possible to use compilers and aspects that add some protections in the object code, which were not specified in the source code, and that prevent or complicates the exploitation of vulnerabilities existing in the program. In all cases, the original source code is not modified.

Design-Level Hardening
Design-Level Hardening refers to changes in the application design and specification. Some security vulnerabilities cannot be resolved by a simple change in the code or by a better environment, but are due to a fundamentally flawed design or specification. Changes in the design are thus necessary to solve the vulnerability or to ensure that a given security policy is enforceable. Moreover, some security features need to be added for new versions of existing products. This category of hardening practices naturally target more high-level security issues. In this context, best practices, known as security design patterns [17], can be used to guide the redesign effort. Although such patterns are targeting the security engineering of new systems, such approach can also be redirected and mapped to cover deploying security into existing software.

Operating Environment Hardening
The Operating Environment Hardening refers to practices that impact the security of the software in a way that is unrelated to the program itself. This addresses the hardening of the operating system, the protection of the network layer, the configuration of middleware, the use of security-related operating system extensions, the normal system patching, etc [1, 57]. Many security appliances can be deployed and integrated into the operating environment in a way that provides some high-level security services. These hardening practices fall within the scope of proper management of an IT department and, as much as they can prevent exploitation of vulnerabilities, they do not remedy them.

5.1 Hardening of High Level Security
Many mobile wireless devices lack high level security properties that make them ideal backdoors to compromise an organization's network. Consumer reacted to this situation increasingly demanding to security features for new and existing systems. In this context, many wireless middleware provide the security capabilities needed to prevent some of these attacks. Other security features such as privilege isolation can only be implemented at the operating system level, making middleware a useful although incomplete solution. Thus, applying low level and high level security become a synergic approach for overall security in the wireless computing universe.

When we talk about high level security, we usually refer to authentication, authorization, confidentiality, availability, non-repudiation and integrity. These are the major security objectives and properties that need to be achieved. For instance, we can choose between password-based or certificate-based authentication, RBAC or Multilevel authorization, Kerberos or SSL for confidentiality, Checksum or Atomic transaction for integrity, etc. Typically, the lack of security at that level is mainly caused by either the absence of those features or the improper implementation of the mechanisms enforcing them. We focus in this subsection on the first case since the later one belongs to low level security hardening.

Hardening of high level security consists of configuration changes at the relevant levels, changes in the application source up to a complete redesign, or integrating new components or libraries to provide the required features. The aforementioned options can be applied either manually or systematically by using AOP. In order for this approach to be truly relevant, the threats should be determined first, so that only the needed countermeasures are put in place. Once the threat is well identified and categorized, it is possible to determine the appropriate technique(s) to mitigate it. In the literature, it is possible to find mapping between categories of threats and

known counter-measures addressing them. Choosing the best techniques will be mostly based on the state of the art of weaknesses and mitigation methods as well as security patterns. For instance, in [34], the authors provide a list of mitigation techniques for each category of threats of their classification. Table 1 provides an excerpt of this mapping.

Threat Type	**Mitigation Techniques**
Spoofing Identity	Appropriate Authentication, Protect Secret Data
Tampering with Data	Appropriate Authorization, Hashes, Message Authentication Codes, Digital Signatures
Repudiation	Digital Signatures, Timestamps, Audit Trails
Information Disclosure	Authorization, Encryption, Protect Secrets
Denial of Service	Appropriate Authentication and Authorization, Filtering, Throttling, Quality of Service
Elevation of Privilege	Run with Least Privilege

Table 1: Mapping Between Threats and Mitigation

The hardening of countermeasures should be specified by the security architects into security hardening plans. These plans determine what the security improvements are, where they should be applied and how they should be applied. These plans offer a separation of concerns between the specification of the hardening and its implementation. The detail steps needed to perform the hardening are the responsibility of security experts. In many cases, they are encapsulated in patterns, libraries and middleware frameworks. For instance, security patterns encapsulate expert knowledge in the form of proven solutions to common problems. Catalog of patterns offer a structured repository of solutions that are inherently formulated in a form similar to the weakness-mitigation mapping we detailed previously. Many works have been published in this domain, which we previously surveyed in [38].

5.2 Hardening of Low Level Security

Based on our classification, deploying security at that level is mainly categorized as Code-Level and Software Process hardening. As such, this type of hardening will be extremely dependent on the programming language and the platform. In this context, we discuss in this subsection the hardening techniques used to remedy the low level security vulnerabilities in C programs.

5.2.1 Hardening for Buffer Overflow Vulnerabilities

Many APIs and tools have been deployed to solve the problem of buffer overflow or to make its exploitation harder [60, 11, 41]. More methods for secure coding can be found in [34]. In this context, the following are some design and programming tips and assumption that can help to solve the buffer overflows problem [13]:

- Always assume that input may overflow a buffer and design the program in a way that provides proper input validation conditions.
- Use functions that respect buffer bounds such as fgets, strncpy, strncat.
- Ensure NULL-termination of strings, even if using those functions.
- Invalidate stack execution, since stack-based buffer overflows are the easiest to exploit.
- The number of arguments of printing functions must be checked to make sure that the format string argument is explicitly specified.

Table 2 summarizes the security hardening solutions for buffer overflows.

Hardening Level	Product/Method
Code	Bound-checking, memory manipulation functions with length parameter, nulltermination, ensuring proper loop bounds
Software Process	Compile with canary words, inject boundchecking Aspects
Design	Input validation, input sanitization
Operating Environment	Disable stack execution, use libsafe, enable stack randomization

Table 2: Hardening for Buffer Overflows

5.2.2 Hardening for Integer Vulnerabilities

Integer vulnerabilities can be solved using sound coding practices. The generalized use of unsigned integers can simplify things for the programmer, and the addition of range checking before sensitive operations can avoid unexpected results. Some compilers provide built-in support for detection of integer issues, and it is possible to replace integer operations with safer calls [46]. The security hardening solutions that we described above are summarized in Table 3.

Hardening Level	Product/Method
Code	Use of functions detecting integer overflow/ underflow, migration to unsigned integers, ensuring integer data size in assignments/ casts
Software Process	Compiler option to convert arithmetic operation to error condition-detecting functions
Design	
Operating Environment	

Table 3: Hardening for Integer Vulnerabilities

5.2.3 Hardening for Memory Management Vulnerabilities

There are no known API or library solutions solving memory management problems as a whole. However, hardened memory managers can prevent multiple freeing vulnerabilities. Other than that, only improvements in programming practices can be useful in hardening against such problems. The following are some hints and best practices: initialize each declared pointer and make it point to a valid memory location, do not allow a process to dereference or operate on a freed pointer, and apply error checking on memory allocation calls. The security hardening solutions that we described above are summarized in Table 4.

Hardening Level	Product/Method
Code	NULL assignment on freeing and initialization, error handling on allocation
Software Process	Using aspects to inject error handling and assignments, compiler option to force detection of multiple-free errors

Design	
Operating Environment	Use a hardened memory manager (e.g. dmalloc, phkmalloc)

Table 4: Hardening for Memory Management Vulnerabilities

5.2.4 Hardening for File Management Vulnerabilities

Vulnerabilities related to unsafe temporary file management and creation can be minimized by using secure library calls [57]. In some cases, we can redesign the application to use inter-process communication instead of temporary files. File creation mask vulnerabilities, in UNIX-like systems, can be resolved using proper file creation-related system calls and specifying appropriate access rights. The security hardening solutions that we described above are summarized in Table 5.

Hardening Level	**Product/Method**
Code	Use proper temporary file functions, default use of restrictive file permissions, setting a restrictive file creation mask, use of ISO/IEC TR 24731 functions
Software Process	Set a wrapper changing file creation mask
Design	Refactor to avoid temporary files
Operating Environment	Restricting access rights to relevant directories

Table 5: Hardening for File Management Vulnerabilities

6 Conclusions

In response to the growth of mobile system use and the need to improve their security, we provided in this chapter a survey on the security of wireless applications and middleware. We first presented the related work in Section 2, and then briefly surveyed software vulnerabilities in Section 3. Afterwards, we studied automatic detection of low-level security vulnerabilities in Section 4. In Section 5, we determined how existing mobile applications and middleware can be hardened in a rigorous and systematic manner in response to the discovered vulnerability.

References

1. Bastille linux, 2006. http://www.bastille-linux.org/.
2. ISO/IEC 7498-2:1989. Information processing systems – open systems interconnection – basic reference model – part 2: Security architecture, 1989.
3. AbsInt. Advanced Compiler Technology for Embedded Systems. http://www.absint.com/.
4. Frank Adelstein, Sandeep K.S. Gupta, Golden G. Richard III, and Loren Schwiebert. *Fundamentals of Mobile and Pervasive Computing*. McGraw-Hill, 2005.
5. Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996. http://www.phrack.org/phrack/49/P49-14.
6. K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes, May 2002. In IEEE Symposium on Security and Privacy, Oakland, California.
7. T. Aslam. A taxonomy of security faults in the unix operating system, 1995.
8. T. Aslam, I. Krsul, and E. H. Spafford. Use of a Taxonomy of Security Faults. In *Proce.* 19*th NIST-NCSC National Information Systems Security Conference*, pages 551–560, 1996.
9. Avaya Labs Research. Libsafe. http://www.research.avayalabs.com/project/libsafe.

10. Jean Bacon, Ken Moody, and Walt Yao. Access control and trust in the use of widely distributed services. In *Middleware 2001 : IFIP/ACM International Conference on Distributed Systems Platforms*. Springer, 2001.
11. S Bhatkar, DC DuVarne, and R Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, 2003.
12. M. Bishop. *Computer Security: Art and Science*. Addison-Wesley Professional, 2002.
13. M. Bishop and D. Bailey. A Critical Analysis of Vulnerability Taxonomies. Technical Report CSE-96-11, Department of Computer Science, University of California at Davis, 1996.
14. Matt Bishop. How Attackers Break Programs, and How to Write More Secure Programs. http://nob.cs.ucdavis.edu/Ÿbishop/secprog/sans2002/index.html.
15. Matt Bishop. How to Write a Setuid Program. Technical Report Technical Report 85.6, Research Institute for Advanced Computer Science, Moffett Field, May 1985.
16. Matt Bishop. A Taxonomy of Unix System and Network Vulnerabilities. Technical Report CSE-9510, Department of Computer Science, University of California at Davis, May 1995.
17. B. Blakley and C. Heath. Security design patterns. Technical Report G031, Open Group, 2004.
18. Blexim. Basic integer overflows. *Phrack Magazine*, 0x0b(0x3c), 2002. http://www.phrack.org/phrack/60/p60-0x0a.txt.
19. John P. McDermott Carl E. Landwehr, Alan R. Bull and William S. Choi. A Taxonomy of Computer Program Security Flaws. *ACM Comput. Surv.*, 26(3):211–254, 1994. http://doi.acm.org/10.1145/185403.185412.
20. Hao Chen and David Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 235–244, Washington, DC, november 2002.
21. Hao Chen and David A. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. Technical Report UCB/CSD-02-1197, EECS Department, University of California, Berkeley, 2002.
22. Coverity. Coverity Prevent for C and C++. http://www.coverity.com/main.html.
23. Subir Das, Farooq Anjum, Yoshihiro Ohba, and Apostolis K. Salkintzis. Security issues in wirless ip networks. In Apostolis K. Salkintzis, editor, *Mobile Internet: Enabling Technologies and Services*, chapter 9. CRC Press, 2004.
24. Steven Demurjian, Keith Bessette, Thuong Doan, and Charles Phillips. Concepts and capabilities of middleware security. In Qusay H. Mahmoud, editor, *Middleware for Communications*, chapter 9. John Wiley & Sons, Ltd., 2004.
25. Takahiro Shinagawa Dept. Implementing A Secure Setuid Program.
26. Dawson Engler and Ken Ashcraft. Racerx: Effective, Static Detection of Race Conditions and Deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252. ACM Press, 2003. http://doi.acm.org/10.1145/945445.945468.
27. David Evans. Static detection of dynamic memory errors. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 44–53, New York, NY, USA, 1996.ACM Press.
28. David Evans, John V. Guttag, James J. Horning, and Yang Meng Tan. LCLint: A Tool for Using Specifications to Check Code. In *Symposium on the Foundations of Software Engineering*, December 1994.
29. M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Syst. J.*, 38(2-3):258–287, 1999.
30. Steve Friedl's. SQL Injection Attacks by Example, 2005. http://www.unixwiz.net/techtips/sql-injection.html.
31. George C. Necula, Scott McPeak, andWestley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.
32. Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *PLDI '02: Proceedings of the ACM*

SIGPLAN 2002 Conference on Programming language design and implementation, pages 282–293, New York, NY, USA, 2002. ACM Press.
33. R. Hasting and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136, January 2002.
34. Michael Howard and David E. Leblanc. *Writing Secure Code*. Microsoft Press, Redmond, WA, USA, 2002.
35. IMMUNIX. Stackguard: Protecting Systems From Stack Smashing Attacks. http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/.
36. Jazilah Jamaluddin, Nikoletta Zotou, Reuben Edwards, and Paul Coulton. Mobile phone vulnerabilities: a new generation of malware. In *2004 IEEE International Symposium on Consumer Electronics*, pages 199–202, 2004.
37. David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *10th USENIX Security Symposium*, pages 177–190. University of Virginia, Department of Computer Science, USENIX Association, August 2001.
38. M-A Laverdière, A. Mourad, A. Hanna, and M. Debbabi. Security design patterns: Survey and evaluation. In *IEEE Canadian Conference on Electrical and Computer Engineering*, 2006.
39. Neal Leavitt. Malicious code moves to mobile devices. *Computer*, 33(12):16–19, December 2000.
40. Cecilia Mascolo, Licia Capra, andWolfgang Emmerich. Principles of mobile computing middleware. In Qusay H. Mahmoud, editor, *Middleware for Communications*, chapter 11. John Wiley & Sons, Ltd., 2004.
41. J. McCormick. Openbsd declares war on buffer overruns. *TechRepublic*, 2003. http://techrepublic.com.com/5100-1035_11-5034831.html.
42. Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
43. Pekka Nikander and Jari Arko. Secure mobility in wireless ip networks. In Apostolis K. Salkintzis, editor, *Mobile Internet: Enabling Technologies and Services*, chapter 8. CRC Press, 2004.
44. Parasoft. Parasoft AEP: Automated Error Prevention Solutions for Business and Information Technology Initiatives. http://www.parasoft.com/.
45. PolySpace. Automatic Detection of Run-Time Errors at Compile Time. http://www.polyspace.com/.
46. R. Seacord. *Secure Coding in C and C++*. SEI Series. Addison-Wesley, 2005.
47. Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, Anaheim, California, USA, April 2005.
48. U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th USENIX Security Symposium, 2001.*, pages 201–220, 2001.
49. Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In 10*th USENIX Security Symposium*, August 2001.
50. SLAM. A Symbolic Model Checker for Boolean Programs , note = http://research.microsoft.com/slam/,.
51. Twitch. Taking advantage of non-terminated adjacent memory spaces. *Phrack*, 56, May 2000. Avaliable from http://www.phrack.com.
52. J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. ITS4: A Static Vulnerability Scanner for C and C++ code. In *ACSAC '00: Proceedings of the 16th Annual Computer Security Applications Conference*, page 257. IEEE Computer Society, 2000.
53. John Viega and Matt Messier. *Secure Programming Cookbook For C and C++*. O'Reilly Media, Inc., 2003.
54. Harald Vogt, Michael Rohs, and Roger Kilian-Kehr. Middleware for smart cards. In Qusay H. Mahmoud, editor, *Middleware for Communications*, chapter 15. John Wiley & Sons, Ltd., 2004.

55. Guijin Wang, Alice Chen, Surya Sripada, and Changzhou Wang. Application of middleware technologies to mobile enterprise information services. In Qusay H. Mahmoud, editor, *Middleware for Communications*, chapter 12. John Wiley & Sons, Ltd., 2004.
56. Gray Watson. Debug Malloc Library, October 2004. http://dmalloc.com/docs/5.4.2/online/dmalloc_toc.html.
57. D. Wheeler. *Secure Programming for Linux and Unix HOWTO – Creating Secure Software v3.010*. 2003. http://www.dwheeler.com/secure-programs/.
58. David A. Wheeler. Secure programmer: Prevent race conditions, 2004. http://www-128.ibm.com/developerworks/library-combined/l-sprace.html.
59. J. Xu, Z. Kalbarczyk, and RK Iyer. Transparent runtime randomization for security. In *22nd International Symposium on Reliable Distributed Systems*, 2003.
60. Y. Younan, W. Joosen, and F. Piessens. Code injection in c and c++: A survey of vulnerabilities and countermeasures. Technical Report CW386, Department of Computer Science, Katholieke Universiteit Leuven, July 2004.
61. Oiwa Yutaka, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. Fail-safe ansi-c compiler: An approach to making c programs secure: Progress report. In *ISSS*, pages 133–153, 2002.
62. Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using cqual for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, pages 33–48. USENIX Association, 2002.