Programming Education in the Era of the Internet: A Paradigm Shift

W. Scott Harrison, Nadine Hanebutte, Jim Alves-Foss
University of Idaho
Center for Secure and Dependable Systems
Moscow, ID, 84844-1008
{harrison,hane,jimaf}@cs.uidaho.edu

Abstract

Over the last several years, the Computer Science (CS) community has put a great deal of effort in to the area of security research, and have made great advances. Counterintuitively, however, the number and severity of cyber threats is not declining, and further, the overall security of computer systems is not improving.

Because of the magnitude of this problem, computer security is an issue that concerns everyone who works with computers. However, computer security training is only given in designated classes to a small set of computer systems users.

This problem needs to be addressed at its core: in the educational system. Insecure code is written by people who do not know the implications of their coding techniques, and current texts and instruction do little to rectify this situation.

Thus, the goal of this paper is to show the need to incorporate basic information assurance knowledge into general CS classes.

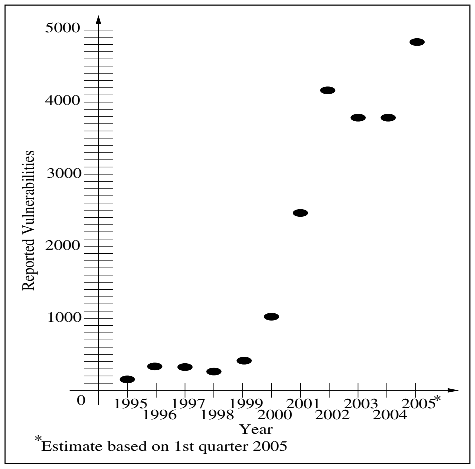

Figure 1: CERT-Number of Reported Security Vulnerabilities Since 1995 [2]

1. Introduction

Computer Science (CS) has evolved a great deal since its inception. And yet, while computer systems have, over the last 15 years, become more connected, accessible, and ubiquitous, the argument can be made using statistics such as the number of reported vulnerabilities (Figure 1.), that system security education has not kept up with this development. As an example, a recent poll [1] showed that from 2003-2004, 42% of a set of surveyed companies have seen an increase, not a decrease[1], in electronic crime. Further, the total number of incidents have also been increasing from 6 in 1988 to over 82,000 in 2003 (Table 1). Why is this the case?

There are a multitude of reasons for the causes that may have led to this development, such as the steady increase in the number of networked computers and services from 1,313,000 in 1993 to 317,646,084 in 2005 [3], which increases the "attack surface," that is the number of potential targets for cyber attacks. In addition, information about vulnerabilities and descriptions on how to exploit them are readily available.

It should be the case that modern educators are teaching students programming "in the era of the Internet," incorporating the knowledge of the change in program execution environment, yet we do not believe this is truly happening. Although it cannot be denied that security has gained immense importance over the last several years, it is still relegated to specific "security courses" and is not, well-integrated into the general computer science curriculum.

The issue at hand is the notion that basic programming classes are taught very much as they would have been taught many years ago, without a recognition that the world of computing has changed and that there is a basic awareness that programmers need to have now that was not es-

[1]A decrease was seen by only 6.2%.

Table 1: CERT-Number of Reported Security Incidents Since 1988

Year	1988	1989	1990	1991	1992	1993	1994	1995
Incidents	6	132	252	406	773	1,334	2,340	2,412
Year	1996	1997	1998	1999	2000	2001	2002	2003
Incidents	2,573	2,134	3,734	9,859	21,756	52,658	82,094	137,529

sential even a few years ago.

At the same time the argument can be made that "the basics have not changed," and this, to some extent, the case. C was written in 1973 and operating systems today are still being written with it. If/then/else statements work as they always have, and Boolean logic has not changed since Boole created it in the early nineteenth century.

What has changed, however, is the magnitude of the law of "unintended consequences." While educators focus their attention on making programs work correctly, very little focus is given to in-depth understanding of the ramifications of the code that is being written.

Students tend to repeat what they have learned first and done the most. What we are creating is a generation of coders, who essentially code like their teachers, using insecure code without necessarily even realizing that it is insecure.

As educators, we fail to teach security research and implementation techniques in a timely manner. This observation is borne out by the simple fact that up to 50% of all code vulnerabilities are due to simple buffer overruns [4] (not enforcing the boundary of an array). However, buffer overrun vulnerabilities are known since 1988, when the "Morris Worm" brought parts of the Internet to a grinding halt [5]. It could be the case that not learning security-aware coding from the beginning is a major cause of these types of statistics.

Security is not a stand-alone problem. Building a system, whether software or hardware, without well-defined security goals will eventually lead to an intrusion or failure of such a system. However, the people who actually implement such systems are generally not security experts. Security, if considered at all, is generally thought of as an afterthought, and often takes second place to other issues, such as ship dates or efficiency. Further the implementors of these system could easily have never taken any security courses of any kind (as most universities do not require this as part of a general curriculum). As such, even when the best intentions exist, one cannot build what one does not understand.

Future CS experts must have a basic knowledge of security issues to build secure and usable systems. Unfortunately, the CS community has thus far failed to properly incorporate this field into the general CS curriculum. By teaching reliability, performance, dependability, and so forth, we teach our students that such attributes must be employed to build quality software. And, to this extent, these topics are represented in both required specialized courses as well as interspersed throughout the basic curriculum; students are early on, generally in a basic programming course, told that a shell sort is more efficient than a bubble sort (as well as being taught why this is the case), that you should not convert a set of ASCII characters to an integer without checking the return value of the conversion function, and so on. Unfortunately, we have not yet added security to the set of base knowledge.

For example, the Accreditation Board for Engineering and Technology (ABET) does not consider security education as a required component of a computer science curriculum in the *Criteria for Accrediting Computing Programs* [6]. The *IEEE/ACM Computing Curricula 2001 for Computer Science* [7], does, acknowledge the necessity to integrate security into the general curriculum. It suggests a set of security classes in addition to the inclusion of security topics as a component of existing classes, for example, Operating Systems. However, a look at *Operating Systems Concepts* [8], one of the standard textbooks used for this course, shows that security is directly emphasized in 6 lines of text only.

An informal poll taken among programming teachers also showed that, unless their field of expertise is security, no security related knowledge is presented in their programming classes. If security is mentioned at all within a non-security-oriented course, it is generally only in terms of available tools and services, such as code scanners and network auditing tools, or in terms of software to avoid using due to insecurities, e.g., *wu-ftpd* [9] and *sendmail* [10]. Specific issues and problems within code and systems, including what code constructs are exploitable, are usually not addressed.

Essentially, the results from security research lack distribution among non-security researchers and educators. The implementation and application of results from security or trustworthy computing research should span all courses within CS (as well as other areas that rely on computer-based technology). In fact, a large number of computer system vulnerabilities, which are often targets of computer crime, were introduced into a system inadvertently [11]; often due to lack of security awareness of the system implementer. While computer system architects and system administrators are very knowledgeable about how to create and maintain a running system, they often do not fully comprehend the security implications of their choices.

The following section discusses different code-level security improvement activities and their importance to the CS curriculum. Sections 3. and 4. discover reason why security education is not part of the general CS education and how teaching programming languages is currently approached. A set of examples from textbooks and related code vulnerabilities are shown in section 5. in order to demonstrate, which problematic issues exist. The paper is concluded with a set of suggestions and finding that might help to improve the CS curriculum and adjust to account for the problems of coding in the Internet era.

2. Current Topics within Code Security

Security issues during the development life cycle can be roughly divided divided in three areas:

1. Security audits
2. Security features
3. Secure implementation

These areas have very distinctive goals within the software development cycle. All three of these principles, applied to the development process in a proper manner will ensure that the system developed is significantly more secure than a system with the same functionality that was built without security considerations.

Textbooks, [12], for example, and articles discuss these issues in parallel. In order to emphasize the importance of what can and should be taught in general programming classes and what should be left to specialized classes, we will discuss in detail each of these focus areas.

2.1. Security Audits

Security audits are performed to check a section of code for the existence of code-level security problems. They can be seen as code auditing with a narrow focus. Audits are used to either verify the results of an implementation process or check third party or legacy code for it compliance to standards. Security audits can be performed in the same manner as a manual audit. In addition, a variety of audit tools, such as Splint [13], RATS [14], BooN [4], and so forth, are available. These tools can detect instances of potentially exploitable code. It is then up to the programmer and his knowledge about security problems to annotate the code, to decide, and to mitigate such instances of code.

Security audits are often added by development companies to address security issues. Audits, if done as an afterthought, are not a replacement for the inclusion of security awareness in the entire development process. Since, security issues can be introduced in all stages of the software development cycle, code audits alone are not sufficient. As is common software engineering knowledge, problems should always be dealt with at the stage of introduction. This includes avoidance, testing, and analysis. This also implies that there should not be a dedicated security audit stage in the development cycle, but rather that security concerns must be addressed at all phases, and this includes the coding stage.

2.2. Security Features

A software system can contain special features that were added because the system use policy requires such features, for example, the use of data encryption over plain text data storage, or ensuring that positive authentication is enforced before system access is granted. A security feature can also be the availability of system setup options to allow enabling or disabling certain functionality in order to adapt the system for a specific usage environment.

From a software system design perspective this means that the developer needs to understand which alternatives are available, for example, which encryption algorithms should be used. It also requires the implementer to understand how a selected feature is to be properly implemented. If, for example, an authentication mechanism can be bypassed or fails on certain input, the mechanism is obsolete or, even worse, it leaves the user in a false sense of security.

Students who are unfamiliar with security research tend to mistake security features for secure development. Security features, in terms of software engineering are features just like an implemented sorting algorithm or the ability for a program to display results in a color chart. Further, in a similar way to other features, they can be implemented correctly or not. If, for example, if no type checking is performed when numerical input is read by a program, the result of sorting this “numerical” data is undefined.

2.3. Secure Implementation

When a system is designed and implemented, there is always more than one way to implement the same functionality. For example, the code `printf(cmd)` will print the content of the string `cmd` to the screen, as will the code `printf("%s", cmd)`. However, the first option, can allow the code to be misused due to the fact that the explicit formatting directive has been omitted.

There are many security issues that are rooted in the inappropriate use of code constructs. For example, some common security faults are:

- *Buffer Overflows* are caused be insufficient input length restrictions.
- *Format String Attacks* are possible if the formatting directives do not match the formatted data in size and type.
- *Race Conditions* are based on the violation of access atomicity and task serialization.
- *Access Validation Vulnerabilities* may be avoided when conditional clauses are designed deny by default.
- *Domain and Input Validation Vulnerabilities* are introduced when the execution environment, such as the trustworthiness of input, is not appropriately implemented.

Each of these problems can come in many versions, with many facets that can be illustrated by examining examples from programming textbooks. For example, in "Problem Solving, Abstraction and Design using C++" [15], students learn how write data to external files. The system call used to create and access the file is listed as: `fileName.open("SomeName")`. However, what the book does not mention, for example, is that the default protection settings on such a file are to allow at least group or group and world read/write access to this file (in Unix, the permissions would be 0660 or 0666, respectively). It is, however, possible to set the file permissions directly in C++ upon file creation. While it is probably not necessary to explain all details of file permissions, simple knowledge about the existence of default file permissions and their ramifications could probably have prevented many access violations within commercial code.

2.4. Code Security Knowledge

Security audits and the design of security features require an in-depth knowledge, such as available tools, how to set them up, how to annotate code, or how encryption algorithms work and the associated strength of a specific algorithm. Such knowledge has to be presented in specialized security classes for students attempting to become security administrator or security researchers.

Secure coding is the proper use of a given programming language in order to avoid code level security problems, and it must be taught as part of the general curriculum since coding is something that all CS students will do as part of their career.

3. Why Secure Coding is Not Emphasized

Historically, it was perfectly acceptable and common practice to write code that simply implemented a set of specified functionalities. Software was written for a well-defined audience to run on a computer that are not connected to an external network. Within the infrastructure that has been developed over the last decade, this somewhat benevolent environment has been replaced by an one in which where essentially every computer is connected to the Internet and the intent of people that have potential remote access to a computer is largely unknown. In light of this, we must reconsider our view on software development.

We have learned that code has to written to be reliable and robust because our code will be presented with incorrect input and run on insufficient hardware by untrained users. We don't make the assumption that every user knows exactly what he is doing and will always do everything in a correct manner. The assumption made, however, is that the input is non-malicious and to some extend reasonable. From a software user perspective we are accustomed to software failure due to some input or event. We, therefore, tend to a void a flawed functionality; trying to figure out "work-arounds" until a patch becomes available. Software faults tend to be seen as a minor problem as long as most of the software functions according to specification. In a malicious environment, all it takes is one exploitable fault for a hacker to potentially damage the entire system.

We need to understand that maliciousness has become a significant part of the computing environment over the last 15 years. We know from experience (Whom's system has not been hacked yet?) and empirical evidence (e.g., CERT, SecurityFocus and ISC) that there are criminal and malicious users that will use software with criminal or other at least questionable intents. It therefore is necessary to ensure that security is be built into every system, hence every programmer needs to understand code security implications.

4. Changing the Current Focus

In [16] Piessens points out that:
"Software developers tend to spend a lot of time thinking about how to make things *possible*. From a security point of view, it is important to spend time thinking about how to make certain things *impossible*."

The focus of modern textbooks and classes that teach students how to develop code is primarily on how to implement the desired features. The students learn how to identify the required features and then learn to decide upon a way of implementing them, given a specific programming language. The specification, design, and implementation of features is one aspect of the software engineering process. The aspect that is less often addressed is the analysis of *constraints*, which is defining what software should *not* do.

Thus, constraints specify what should be *impossible*. In terms of security engineering, a student needs to know which additional "features" a specific algorithm or code sequence choice yields. As a result, the student has to understand that some feature implementation solution cannot be used because they will lead to a violation of constraints.

For example, one would expect a constraint on using the `printf()` function (in standard C) not to alter program behavior beyond printing data to an output device, and this is the general knowledge of this function.

However, in order to satisfy this constraint, a student must understand the difference between using (for example): `printf(cmd)` and `printf("%s,cmd")`. Both implement the same feature, specifically, printing the content of the string `cmd`. Both lines of code will compile without an error. However, the first statement (due to the way the stack interacts with `printf()`) allows exploitation by giving control flow of the program to a user via a format string attack.

The following quote, taken from the SysAdmin, Audit, Network, Security (SANS) Institute's list of the most exploited problems, illustrates the need for this type of knowledge [17]:

"Most web servers support Common Gateway Interface (CGI) programs to provide interactivity in web pages, such as data collection and verification. Many web servers come with sample CGI programs installed by default. Unfortunately, many CGI programmers fail to consider ways in which their programs may be misused or subverted to execute malicious commands."

5. Programming Textbooks

In order to illustrate the problem we will show and discuss a series of code examples taken from C++ programming textbooks as they are presented to our students together with exploits that take advantage of such constructs. C++, is alongside JAVA the most commonly first taught programming language. We limit the following textbooks examples to C++, but similar issues can be found for almost every programming language. The presented problems are a subset of existing issues and far exhaustive.

5.1. Buffer Overflows and String Termination

Buffer Overruns are the single most exploited code level vulnerability [4]. To cite some examples, the "Blaster" Worm [18] is due to a buffer overrun vulnerability within Microsoft's (MS) Distributed Component Object Model Remote Procedure Call. The "Slammer" Worm enters a system through a buffer overrun in MS-SQL [19]. "Wu-ftp" (a common FTP server) was the subject of four CERT[2] Advisories due buffer overruns in the code (Advisories: 2001-33,2000-13,1999-13,1999-03). The "Sasser" worm spreads via a buffer overrun in the MS Local Security Authority Subsystem Service (LSASS) [20]. Ironically, a buffer overrun vulnerability can be very simply avoided through proper array length enforcement.

For example, in [15, p.473], strings as arrays of characters are discussed. One of the suggested methods to read the content of a character array is:

```
char flower[10];
cin >> flower;
```

These statements are presented without any indication of the need for boundary control. In its form this code construct can lead to a buffer overrun if the input string is greater than 9 characters. Only when talking about C specific statements such as `strcpy()` do the authors demonstrate the difference of this statement with `strncpy()`, even though C++ provides means for adding boundary restriction to the above code. The authors do mention the the possibility of memory overwrites with `strcpy()` but not discuss the severity of such an event.

Additionally, it should be made clear to students that the `string` class in C++ can only handle strings of limited length, otherwise the program behavior will be undefined. The chance of overrun is not limited only to strings handling functions. Even if a "secure" function, such as `strncpy(char *a,char *b,int n)` is used overrun can occur if the code to determine `n` contains an integer overrun, as for example in [21].

"Programming in C++" by D'Orazio [22, p.593–605] on the other hand, does a very good job of preparing students for the pitfalls of character array handling. The author points out the possibility of overwriting memory and suggests methods to prevent this from happening. This book also points out the different C and C++ statements that should be used for proper string handling. The importance of buffer termination for code level security, however, is not mentioned.

Other textbooks discuss this topic with similar detail or lack thereof.

The approach in [22] presented along with an indication about how this may violate data integrity could prepare students to understand how improper string storage might lead to security violations. Without pronouncing the importance of proper input validation, a student might think that if an array is just "reasonably large", there will be no problem with the code. Students must understand that input to a program that they write might not necessarily be made by "reasonable" users.

[2]http://www.cert.org

5.2. Input Validation

Input validation faults occur when user input and the input data type don't match. The issue that has a great impact on security is what happens when the input is incorrect? How do we go about handling the different cases of invalid input? When the student learns to write an input handling function, one thing he needs to understand is the "deny by default" principle. This problem is exemplified by Howard in [12, p.65]. The idea is to check if the *allow* condition is fulfilled and deny if not. Otherwise, if it is checked if specific error conditions are present, some, not explicitly stated, error condition might have been omitted. Even though an error occurred the program would continue in a situation where functionality execution should have been aborted.

Further, we need to move away from the notion that "input comes from files and keyboards." Input is not limited to user input from the keyboard, but from many (possibly uncontrolled) origins such as network traffic, files, calling functions, shared memory, message passing, and so on. As such, the importance of input verification cannot be overstated. The C++ textbooks usually introduce input using a statement along the lines of [15, p.60]:

```
float miles;
cin >> miles;
```

What is going to happen if the input to `miles` is not a number?

An input validation problem where the input is actually a packet from the network is motive for the *teardrop* Denial-of-Service [23] attack, where an attacker sends invalid alignment data to be used for packet reassembly. Since some reassembly routines do not perform a check if this data is actually valid, the result is that a negative integer that is copied into memory allocated for an unsigned integer. This is then interpreted as a very large integer, and, as this number is the size parameter to a memory allocation routine, the allocation process fails and may result in the service or even operating system to crash.

5.3. File Access Control

In [15, p.398] students learn how write data to external files. The code example to create and access the file are constructed similar to `fileName.open("SomeName")`. What the book does not mention is that the default protection settings are world-readable and world-writable when a file is opened this way (although this may differ from system to system). In fact, this text does not even mention the notion of default protection settings or how to set or check file permissions. Other books (e.g, [22, p.138], [24, p.198]) omit the importance of this issue as well.

Microsoft Windows 2000 Insecure Default File Permissions Vulnerability [25] exists because of insufficient default file permissions. Lax permissions on a screensaver executable is the reason for the Microsoft Windows Logon Screensaver Local Privilege Escalation Vulnerability [26], which is run with *SYSTEM* privileges and can result in local users gaining these privileges.

Further, when we teach students about handling file and data streams, the issue of atomicity and serialization must be discussed. Race conditions (in particular, Time-of-check-time-of-use) are a result of algorithms that do not implement proper checks of atomicity and/or serialization. A race condition in Yahoo's Messenger Server allows access to personal information of other users [27]. A race condition in McAfee's Internet Security Suite allows attackers to disable this security software [28]. Textbooks examples, such as (shortened from [15, p.399]):

```
ofstream outs;
outs.open(inFile);
outs.fail() {
 . . . }
copyLine(ins.outs);
```

leaves students with the impression that the `.fail()` method will take care of everything that can go wrong, and that it is not possible for the file to be deleted or linked between creation, check, and access. However, we cannot teach proper file handling without discussing race conditions if we expect our students to not make the same faux pas as the Yahoo and McAfee programmer (and many more). As such when we discuss files and streams the pre- and post condition of the file access step should be emphasized.

6. Conclusions

None of the textbooks we looked addresses security explicitly in conjuncture with the presented material. One book provided some basic security terminology. Some authors do a better job then others to emphasize potential problems and constraint on code constructs. D'Orazio [22] discussed several potential problems in detail that may lead to security problems, implicitly preparing students for potential exploitable pitfalls.

Not all security, trustworthy computing, or information assurance findings can and should be taught to all CS students. Howard [12], for examples, gives plenty of examples how little programmers actually know about code level security and that employers currently training their coders to update their security knowledge.

Institution that provide formal training in coding, need to follow this trend, since it is our goal to prepare students properly for their professional career. We do not need to teach them specific algorithm, auditing tools, or solutions. The decision which to use is usually specific to the employer. What we need to teach them is the basic understanding for the environment that their code will execute in.

Adding an emphasis on security awareness to formal CS training could be done by requiring a secure coding class as part of the curriculum or to add security topics to existing classes. The first approach yields the problem that we essentially teach the same topic twice, once without once with security emphasis. Besides redundancy, the issue here is that the students will have to revise "wrong" or incomplete knowledge, which might lead to confusion. So why not teaching it right the first time?

Adding attention to security issues to general programming classes is necessary and in some textbook we can find attempts to draw attention to common programming pitfalls. These text passages, extended through integration of lectures and examples from secure coding books, such as [12] and [29] could be one way to go about improving programming classes and associated textbooks.

In order to avoid that the next generation of programmers will make the same mistakes as the current generation and/or their teachers, a revision of how programming classes are done is needed to prepare students for the 21st century programming and code execution environment.

References

[1] CSO. 2004 e-crime watch survey. `http://www.csoonline.com/releases/052004129_release.html` [June 2005].

[2] CERT/CC. CERT/CC Statistics 1988-2005. `http://www.cert.org/stats/cert_stats.html` [August 2005].

[3] ISC. Internet domain survey, jan 2005. `http://www.isc.org/index.pl?/ops/ds/` [June 2005].

[4] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed Systems Security Symposium (NDSS 2000)*, 2000.

[5] E. Spafford. Presentation: The internet worm + 10 years: Lessons learned and not learned. `http://www.cerias.purdue.edu/homes/spaf/presents/Andersen.pdf` [June 2005], 1998.

[6] ABET. Criteria for accrediting computing programs 2004–2005. `http://www.abet.org/LinkedDocuments-UPDATE/CriteriaandPP/05-06-CACCriteria.pdf` [June 2005].

[7] IEEE/ACM. Computing curricula 2001. `http://www.computer.org/education/cc2001/cc2001.pdf` [June 2005].

[8] A. Silberschatz, G. Gagne, and P. Baer Galvin. *Operating System Concepts*. Wiley; 6th edition, 2002.

[9] CERT/CC. CERT advisory CA-2001-33-Multiple Vulnerabilities in WU-FTPD. `http://www.cert.org/advisories/CA-2001-33.html` [June 2005].

[10] CERT/CC. CERT advisory CA-2003-07-Remote Buffer Overflow in Sendmail. `http://www.cert.org/advisories/CA-2003-07.html` [June 2005].

[11] C. Landwehr, A. Bull, J. McDermott, and W. Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys*, 3(26), 1994.

[12] M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press: Redmond WA, 2003.

[13] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1), 2002.

[14] SecureSoftware. RATS. `http://www.securesoftware.com/download_rats.htm` [June 2005].

[15] F. Friedman and E. Koffman. *Problem Solving, Abstraction and Design Using C++*. Pearson / Addison-Wesley, (4th Edition), 2004.

[16] F. Piessens, B. De Decker, and B. De Win. Developing secure software - a survey and classification of common software vulnerabilities. In *Proceedings of the Fourth Working Conference on Integrity, Internal Control and Security in Information Systems (IICIS 2001)*, 2001.

[17] SANS. How to eliminate the ten most critical internet security threats. `http://www.sans.org/top20/top10.php` [June 2005].

[18] SANS. Sans malware faq: What is w32/blaster worm? `http://www.sans.org/resources/malwarefaq/w32_blasterworm.php` [June 2005].

[19] SANS. MS-SQL Slammer. `http://www.sans.org/resources/malwarefaq/ms-sql-exploit.php` [June 2005].

[20] eEye Digital Security Advisory. Windows local security authority service remote buffer overflow. `http://www.eeye.com/html/Research/Advisories/AD20040413C.html` [June 2005].

[21] Citadel. Citadel's top hpux remedies nr: 6. multiple vendor calloc() implementation integer overflow vulnerability. `https://hercules.citadel.com/hpux.html` [June 2005].

[22] T. D'Orazio. *Programming in C++*. McGraw Hill, 2004.

[23] David Hoggan. The internet book- section: Teardrop fragmentation attack. `http://www.camtp.uni-mb.si/books/Internet-Book/IP_TeardropAttack.html` [June 2005].

[24] B. Overland. *C++ Without Fear*. Pearson / Prentice Hall Professional Technical Reference, 2005.

[25] ISS. Windows 2000 weak system partition permissions. `http://xforce.iss.net/xforce/xfdb/9779` [June 2005].

[26] SecurityFocus. Microsoft windows logon screensaver local privilege escalation vulnerability. `http://www.securityfocus.com/bid/11711/info` [June 2005].

[27] SecuriTeam. Yahoo! messenger server race condition vulnerability. `http://www.securiteam.com/windowsntfocus/5IP0I20FPO.html` [June 2005].

[28] SecuriTeam. McAfee internet security suite race condition vulnerability. `http://www.securiteam.com/windowsntfocus/5TP0C2KFFG.html` [June 2005].

[29] J. Viega and G. McGraw. *Building Secure Software*. Addison-Wesley (Proessional Computing Series), 2002.