NOV 30 2005

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** 22.Nov.05	**3. REPORT TYPE AND DATES COVERED** MAJOR REPORT

4. TITLE AND SUBTITLE FOUNDATIONS FOR SECURITY AWARE SOFTWARE DEVELOPMENT EDUCATION.	**5. FUNDING NUMBERS**
6. AUTHOR(S) MAJ MCDONALD JEFFREY T	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) FLORIDA STATE UNIVERSITY	**8. PERFORMING ORGANIZATION REPORT NUMBER** CI04-1706
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) THE DEPARTMENT OF THE AIR FORCE AFIT/CIA, BLDG 125 2950 P STREET WPAFB OH 45433	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION AVAILABILITY STATEMENT Unlimited distribution In Accordance With AFI 35-205/AFIT Sup 1	**12b. DISTRIBUTION CODE**

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

13. ABSTRACT *(Maximum 200 words)*

14. SUBJECT TERMS	**15. NUMBER OF PAGES** 10
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**

Foundations for Security Aware Software Development Education

Abstract

Software vulnerability is part and parcel of modern information systems. Even though eliminating all vulnerability is not possible, reducing exploitable code can be accomplished long term by laying the right programming foundations. We argue that the current hot topic of so-called "secure coding" represents commonly taught coding techniques to ensure robustness, rather than any commonly understood concept of security. In this paper, we show how rigorous coding techniques should be woven into the fabric of computer science curriculum and ultimately should be distinguished from requirements-driven security techniques. Coding for security is useless without rigorous fault-tolerant coding techniques. However, the two are best separated pedagogically with a reinforcing theme in educational environments. This instructional paradigm shift will ultimately produce programmers, developers, and software engineers who habitually create security aware software.

1. Introduction

As the saying goes, the best defense is a good offense. Defensive coding practices, which are termed by many as "secure coding" [1, 2], are intended to offensively counter the growing threat of software vulnerability exploitation. Buffer overflows [3,4] have been a hot topic of secure coding because they remain commonplace in diverse attack schemes where malicious code is injected and then executed on a victim system. As Hogland and McGraw state, buffer overflows are the "whipping boy of software security" because of all the hype and fear they generate [5]. Fixes to buffer overflow problems emerge in software system after system, usually after the fact and usually addressing only symptoms of a greater problem inherent in the design and implementation of the code itself.

While these fixes are labeled "security patches", they actually are less related to security than they are to poor programming practice. Many programming errors can result in security vulnerability. Buffer overflows occur because programmers do not follow well-known programming practices, resulting in software delivered with unresolved faults. Buffer overflow repair has little to do with security of the application being repaired; in fact, most buffer overflow exploits use the vulnerability to gain access privileges they would otherwise not have, though the resulting mischief or malice are rarely directed at the vulnerable application itself.

So, most of the instances that we see today advertised as software exploitation is nothing more than software failure. Security can be seen under the larger umbrella of software assurance—which covers failure from both malicious and non-malicious interactions. Whether failure comes from execution of unintentionally buggy programs or the malicious exploitation of a weakness inherent in code—software faults cause loss of productivity and subsequently loss of revenue.

A primary goal of software assurance is to engineer robustness from the ground up. When robustness is in view, problems identified as security flaws can be seen as problems that stem more from poor programming practices than security threats. Without laying a proper foundation in "good" coding practices for fault tolerant software to begin with, integrating *security* goals into software becomes useless. We present our view of a needed paradigm shift in educational environments that focuses on teaching principles of coding for robustness and security. Though they are two distinct pedagogic concepts, security builds squarely upon robustness and both should be incorporated strategically into computer science curriculums. Strengthening the foundation of what and how we teach future programmers about robust practices will provide the necessary foundation to incorporate security specific goals.

Initial work has already been accomplished to identify how security can be integrated into software systems. The systems security engineering capability maturity model (SSE-CMM) was launched in the late 1990's to address the need for holistic integration of security in the software development lifecycle [6]. The SSE-CMM provides a framework to reason about security needs, guidance, vulnerabilities, assessments, and effectiveness of security mechanisms themselves. McGraw states that to practice good software security you must leverage good software engineering practices and start early in the development life cycle [7]. Capability maturity evaluation criteria and the subsequent security solutions advocated by professionals in the field have been aimed at higher level entities such as chief information officers, administrators, and infrastructure planners [8, 9].

More recent work has begun to express practical guidelines for programmers and developers and thus get to the root of the problem. Hogland and McGraw [5] assert strongly that "bad" software is the root of the security problem while Ghost *et al.* [8] express that security problems are only fixed at the core by building robust and survivable software. The tide is beginning to turn towards fixing security at its root source—where programmers actually create lines of code and build software components.

The remainder of this paper is organized along these pedagogic lines as well. In section 2, we address what should be considered foundational coding principals that support robustness. We discuss the educational paradigm shift that should occur to support this change: clear identification of fault-tolerance risks and coding techniques to mitigate those risks. Section 3 details coding principals and mechanisms to support security specific requirements themselves. Section 4 gives our conclusions and recommendations for future work.

2. Coding for Robustness

The root cause of many common security faults can be traced back to lack of risk avoidance for software faults in general. Robustness, defined in [10], is the "degree to which a software component functions correctly in the presence of exceptional inputs or stressful environmental conditions". When talking about exceptional cases for program execution, we easily come into the modern day realm where programs are used in ways that they were never intended to be used—with malformed inputs, altered program control flow, dynamically linked or patched code segments, and possible memory corruption.

Educational environments must begin to foster the notion that secure programs are first and foremost reliable and safe programs. Safe, in the loose sense of the word, meaning that programs clean up after themselves, police their own code and data space, and don't assume anything from outside their environment without verifying first. It is truly the ounce of prevention that far outweighs the "tons" of cure later down the road. The foundations, however, begin with how coding is presented above the basic functional and semantic levels.

2.1. Process Maturity

The amount of rigor (and therefore robustness or security) that a development organization puts into its software process is tied to the nature of the computer programs they develop. We notionally consider three levels of rigor, which can be logically tied to a likely level of capability and process level maturity (CMM level). At the lowest level of required rigor, disposable software is "here today and gone tomorrow". Such programs have no need for maintenance or lifecycle cost assessment. They are written to get a specific job done in a very short amount of time—with no view for long term use (maintenance nightmares occur when this assumption is violated).

While disposable software can be produced by any CMM-level 1 organization, most software systems are developed with expected longer term survivability. More rigor is required depending on the budget, customer base, projected lifetime, and complexity of the system to be developed. Typical software development organizations strive for some level or process maturity to sustain the 80% category of non-disposable software—roughly striving for at least CMM-level 2 or 3 maturity for minimum success.

The highest level of rigor we term the "moon-shot" system. This is software which is not only long term and requires established disciplines in the development organization for success, but it is software with very stringent requirements based on safety (human life is at stake), monetary value (multiple billions might be at risk), large user base, or high maintenance cost factors (a satellite sent into space). We would hopefully not rely on organizations with development capabilities lower than CMM-level 3 to undertake such development efforts. In fact, such software systems represent our desire to use the very *best* processes for software development that can be used—in every area of the lifecycle.

Robustness and security can be seen in similar terms as rigor. Disposable software may not require or benefit from either robust coding principals or integration of security requirements. Non-disposable software, on the other hand, is more ubiquitous than either the moon-shot or disposable variants. Because most software falls into this category, we want future programmers to understand their role in contributing to software that is not only functionally correct and efficient, but that is robust and designed to prevent faults. We consider next the source of vulnerabilities from both common defects and dangerous coding techniques. We consider the risks that can occur from simple but commonly overlooked programmer choices.

2.2. Risky Defects

Much work has been done to identify coding weaknesses and vulnerabilities that are security related, *so-called* [1, 2, 5, 8, 11, 12]. By definition, software vulnerability is a defect in either design or implementation of code—and 90% of vulnerabilities derive from exploiting known defect patterns in coding [11]. Practical avoidance of these defects—by identifying causes of failure in code and appropriate remedies to fix them—must be introduced early into our curriculums.

2.3. Sources of Faults

Buffer overflows have been called the "nuclear bomb of all software vulnerabilities" [5]. They are defined as simple programming errors that allow memory corruption to take place—data is written outside of preordained boundaries of some data structure in memory. Once corruption takes place, a multitude of very nasty attacks can be leveraged through the software—overwriting of critical program information, changing global state, removing security restrictions, or disabling program controls. In languages like C, string handling routines that assume the presence of the NULL terminator create a ripe environment for such attacks.

What was originally a library design choice to make the life of the programmer easier (you don't have to manage the size of a string yourself) has now turned into an incredible security nightmare. The real problem, however, is that programmers are not taught early on that such assumptions are not valid from a fault-tolerant point of view. Even though fault injection techniques [3, 4] have been developed to identify and root out these failures, but do not guarantee their absence in a given program.

It is no longer safe to assume that a NULL terminator in a C program will always indicate the end of a string. Instead, the programmer should manage and verify string sizes for themselves. Curriculums must establish that "normal" programs always check and verify memory operations-i.e., it is the programmer's job to make sure buffer operations stay within their bounds.

Because data buffers and memory locations are both used on the stack, redirecting program control flow becomes rather simple once a given weaknessis discovered. Though security conferences, publications, and books alike are trumpeting the need for security awareness and calling attention to these flaws, lack of attention to detail on the part of the programmer is really the major issue. True, buffer overflows can be exploited to modify a variable or data pointer or even the return address of a function that is sitting on the stack. Such modifications can alter program behavior, application data, or execute viral code. Nonetheless, these end results are problems that concern robustness, not security.

Though security conferences, publications, and books alike are trumpeting the need for security awareness and calling attention to these flaws, the major issues derive from lack of attention to detail on the part of the programmer. Traditional testing methods have required overhaul to account for this new source of software weakness. Fault injection techniques [3, 4, 13] have been used for quite some time to identify and root out these failures, but do not guarantee their absence in a given program. Because data buffers and memory locations are both used on the stack, redirecting program control flow becomes rather simple once a given weakness is discovered. True, buffer overflows can be exploited to modify a variable or data pointer or even the return address of a function that is sitting on the stack. Such modifications can alter program behavior, application data, or execute viral code. Nonetheless, these end results are problems that concern robustness, not security.

Another class of vulnerabilities derives from integer manipulation errors and truncation [11]. Code that performs numeric computations by nature has the possibility for underflow, overflow, signed numeric errors, and truncation of data bytes because of smaller data type capacities. Such errors occur because ranges are not checked on variables or results, integer operations are not bounded, and variables are wrongly cast from larger types to smaller types. These flaws again are attributed to poor programming practices that do not consider robustness—even though they can be exploited for malicious purposes. Such errors occur not because security was in view to start with—but because good programming practices for fault tolerance were not encouraged.

Memory leaks are another source of problems that revolve more around survivable code than security. Leaks occur when programmers do not manage memory properly—thus the operating system is not aware of memory that should be free and available. Allocated memory locations can be read and exploited by adversaries and exploited to reveal program data. Memory leaks can also occur around function calls when parameters are altered by use of adversary-controlled formatting strings.

Yet again, these risks are not security-centric in nature. They stem from failure to validate user input or prevent users from (mistakenly or on purpose) providing erroneous input or formatting strings to the program. Good fault-tolerant coding practices would eliminate such vulnerabilities.

2.4. Risky Coding Techniques

The marvel of modern programming languages is that they have greatly simplified the job of the programmer to take software requirements and translate them into executable programs—all with little knowledge of the underlying hardware environment. However, even the best selling book of all time can instruct us in this regard: "all things are lawful for me, but not all things are helpful" (1 Cor 10:23). Though powerful language features are well within a programmer's right to use, certain features can cause undue problems in stressful or abnormal environments—environments where malicious parties can wreak havoc when certain language features are used.

Bertrand Meyer was one of many to recognize the inherent dangers that come with powerful language features [14]. He notes that a language design can be considered *bad* when "the programmer is presented with a wealth of facilities, and left to figure out when to use each, when not, and which to choose when more than one appears applicable." Take for example polymorphism and the ability to dynamically bind classes at run time. Polymorphism gives the dangerous facility for a subclass to change the operations or intentions of its superclass. When dynamic binding is allowed, an adversary can easily take advantage of this facility for malicious purposes.

Aside from the security threat, however, advocates against polymorphism in other languages have traditionally pointed to the decrease in reliability and fault-tolerance that such features introduce [15]. The inherent risk of using dynamic binding is not primarily from malicious parties but rather that the end-user or run time environment will not properly execute the decision of which method to invoke. This reveals a deeper fault tolerant problem—that of ensuring dynamic code is locatable and loadable—way before issues of Byzantine faults come into view. In general, these programming features are powerful, but not conducive to reliable software. As such, curriculums need to promote the use of safer and more reliable programming techniques in lieu of certain language features like polymorphism.

As another example, consider dynamic memory allocation. It is almost heretical to suggest that we could live without such a feature in programs, but static allocation of program resources ultimately leads to more reliable code. How many errors in modern software, not even related to security or intruder activities, are related to improper memory management on the part of the programmer? Dynamic memory allocation can be incompatible with both program predictability and is non-deterministic by nature—qualities that fault-tolerant software should avoid. Our education process again must change to not only teaching the functionality of languages, but also the inherent risk to program reliability that comes when certain language features are used.

2.5. Techniques for Robustness

Robust programming methods establish that programmers should expect and code for the unexpected. We mention several methods here for completeness and affirm that these practices need to be established in computer programming courses of all levels—including high level software engineering—so that reliable coding becomes the foundational premise on which other, more security related, techniques can be built. These principles can be taught and enforced in the ground floor of programming level curriculum to solidify the importance of robust code.

Type systems have been the subject of much debate over the years in terms of whether they increase programmer productivity or code reliability and reduce software faults. *Type systems* of programming languages can be characterized as strong or weak while *type checking* can be termed static or dynamic [16]. Strong typing simply dictates that that all types for variables and data structures must be defined and known at compile time. We would agree with such findings in [17] that using strong typing does in fact lead to more reliable code and an overall reduction in defect-induced software faults. In the case of RoboX which was implemented on two different platforms and coded by different yet equally skilled teams, a sixteen-fold increase in quality was noted and attributed to the memory safety induced by strongly-typed language [16].

Regardless of arguments for or against, *strong typing* forces detection of type errors. This practice should be introduced early in programming curriculums as a general principle of robust coding. The more errors that can be handled or prevented before software execution, the more safe and reliable code will be. Both strong and static typing provides proof that *some* aspect of a program executes correctly—but they can not prove the absence of all run time errors.

Another simple technique to increase robustness would involve the avoidance of variable length fields. Any data field whose length is determined dynamically reduces the verifiability and safety of a program automatically. Education should focus programmers on the risk reduction techniques that will reduce run time faults—which includes verifying as much of a program's data structure as possible before hand.

A third technique for robust programming solves the problem of making environmental assumptions: always filter input. Validation of input data so that only legal values are permitted should be discussed in programming curriculum alongside the functional aspects of how to get data into a program. This includes basic, good practices such as checking integer ranges in code and using safe operations on untrusted data. Size validation of input data must guarantee that it does not exceed the size of its storage buffer—a basic quality coding practice that reduces run time faults significantly. As a great side effect to these techniques, of course, security threats are consequently reduced.

Most importantly, extensive and systematic testing should be considered normal and common practice for programmers and no longer relegated in academic curriculum to specialized courses. Source code auditing and reviews need to be integrated as part of traditional language courses to establish that rigor is no longer an option for non-disposable software systems. Static and dynamic analysis techniques and the proper development of testing suites must also take forefront in the way academic institutions present programming to future professionals in the field. Tools for checking the correctness of program code need to be introduced at the same time that compiler features are being taught.

By the intermediate programming level, most programming students have been introduced to graceful degradation techniques. The notion is that when unexpected program termination is unavoidable, programmers reduce the impact to the system and to the end user.

In addition to these, there are a multitude of other practical techniques that fall under the category of *good* and *safe* programming rules. With the increase of processor capability, CPU cycles tend not to be the greatest driving factor in software systems these days—quality, reliability, and safety may be however. Among suggestions provided by Plakosh in [11], it is a good idea to use unsigned types for variables which should never have negative values. Programmers should consider that letting a user control the format of input is usually a bad idea. String constants tend to be better for both formatting and output.

Plakosh also points out that numerous ANSI C standard library functions are susceptible to buffer overflow attacks. The use of these may endanger programs needlessly to faulty logic and runtime errors from unexpected input. A better alternative may be to use string functions where maximum number of bytes can be specified in the operation. C++ string functions and other "safe" string libraries also exist. In many cases, using a language that performs runtime boundary checking is a way to mitigate poor programming skills—but the better solution is to change the way we educate.

To conclude this train of thought, much of what is touted currently as "secure" coding techniques are really nothing more than programming principles that support robust and reliable software. Our educational paradigms must shift to introduce these concepts at the same time that functional aspects of programming languages are taught. When rigor is demanded in software development, programmers must be familiar with standard coding practices that support safe, reliable, and efficient software. The burden rests on the educational establishment to instill this notion early, consistently, and continuously in its academic programs. Once this foundation is present, coding for security specific threats is not only possible, but can be taught from a distinctly different pedagogic framework.

3. Coding for Security

Programs may not rely upon or need protection because either there is low risk of malicious or mischievous behavior or there is less sensitivity to environmental influences. The extensive, and still expanding, business reliance on the Internet is a major driving force into security-aware practices.

It is easy to understand why it has taken so long for security issues to become incorporated into software practices. Programming-in security is not cheap. First and foremost, for software to be secure, programmers must apply their maximum level of rigor to ensure that their software is essentially flawless. Any routine programming error injects vulnerability into the system[1]. The cost of the additional rigor necessary to reduce errors coupled with the increasing pressure to be the first to the market often leaves security as a second class citizen...and a lot of money has been made based on this business model. The sins of the past are now catching up with us, the innocent observers.

The Internet itself was not designed with security in mind; rather the early (and lingering) focus was on connecting computers in a heterogenous environment. Security was left to the application or to the next generation (e.g. IP v6) and was not a primary concern because business application was not driving the development. Security was simply an afterthought.

Still, it would be nice if security were transparent to users. Unfortunately, often only the user knows what security is need, though they are often do not understand the vulnerability. This is nowhere better illustrated than with Internet browser security mechanisms, where only rare users have a clue about "restricted or "trusted" sites, let alone what the impact of allowing ".net" or "authenticode".

Thus, if applications are to be secure, analysts must be able to recognize security requirements during early development phases, and these requirements must trigger an appropriate response from designers and coders. The following discussion addresses security threats to software[2] and identifies some specific responses that are appropriate for applications where security requirements are identified.

3.1. Caveat Emptor

Most security specialists are naturally skeptical. They question even the simplest assumptions and verify ad infinitum. While extreme responses are only necessary in extreme circumstances, an elevated level of skepticism is necessary when programming software for systems with security requirements. Ideas that project managers driven by deadlines and functionality baselines see as extreme, are seen as routine by security specialists.

The question becomes where between the "beta test and out the door" and "zero defect" approaches does responsible software practice lie? We earlier proposed a three tiered hierarchy to reflect the level of rigor necessary to prevent program faults. For security sensitive software, only the most rigorous fault prevention is suitable, since any error injects security vulnerability into the software.

Moreover, other security vulnerabilities occur because programmers fail to be sufficiently skeptical. The following are security-specific techniques that are sometimes taught simply as good programming practices, but that have a direct impact on software security.

3.2. Garbage Collecting

One of the easiest places to implement controlled skepticism is through aggressive garbage collection. Items left over from program execution can offer a sophisticated intruder information free of charge and with little effort, depending only upon the operating system procedures for terminating programs and the ingenuity of the adversary.

One of the easiest items for a programmer to clean up is memory. When a memory location is no longer needed by a program, it should be cleaned (overwritten) and released. When a program completes its task normally, it should clean and release any remaining memory resources. This may mean executing a loop that overwrites a character at a time, or utilizing a programming language construct that accomplishes the same function, as long as the action is overt (not assumed by some unproven or non-standard feature).

Of course sophisticated adversaries may circumvent this process by causing a program's abnormal termination before cleansing occurs. We posit that such abnormal termination is only possible through programming errors and again emphasize techniques for graceful degradation prevented in the previous section.

Memory is not the only resource where sensitive residuals may reside. Communication connections are vulnerable to data interception, message injection, and session hijacking. Thus, connections that pass sensitive data must be carefully protected, using direct security techniques of strong authentication and encryption. These techniques are recognized as being employed in classical security systems.

Multi-process or multi-threaded systems are notorious for loosing track of or leaving subordinate processes unguarded when the main program terminates. If left unguarded, these processes may be hijacked by sophisticated intruders in much the same way as connections. Such "ghost" processes may be used by

[1] Here is a clear illustration of the relationship of proper programming practices to security. Sloppy or less rigorously written programs are rarely secure.

[2] Here, we take a *caveat emptor* approach and suggest actions that programmers can take in addition to (possibly overlapping) operating system protection.

intruders to reveal residual data or other malicious intent.

3.3. Starting with a Clean Slate

One of the first things that entry-level programmers are taught is how to initialize data structures. They are aided in this elementary task by language and architectural approaches to data initialization. However, the need to initialize data structures by clearing out all residual data is often not recognized by programmers eager to exercise their new-found skills to produce highly functional programs. For security sensitive programs, proper initialization is essential; else data may be injected into a process from an unrelated process that previously utilized the memory location.

Again applying the caveat emptor principle, one approach to addressing memory related vulnerability is for the application programmer to manage their own memory, where possible. This entails a programmer establishing a memory management process that requests memory in bulk, then manages the allocation during execution.

Under this paradigm, the entire memory allocation can be cleared when it is acquired and increments can be cleared when they are returned internally for redistribution. The internal memory manager can also clear the entire allocation before releasing it to the operating system just prior to program termination.

3.4. Cleaning Temporary Storage Areas

We briefly digress to address an issue that is not under the control of the application programmer, but that reflects a similar security principle, that of clearing temporary storage areas. Operating systems and input-output systems frequently utilize temporary storage locations such as caches, swap spaces, and print spools for synchronization, performance, or efficiency optimization. Not only are the operations themselves outside the control of the programmer, the storage areas themselves are not directly or legally accessible to the programmer.

While caching may be out of their control, application programmers may be able to reduce vulnerability injected by temporary storage operations. Compartmentalizing operations so that data is used immediately after it is required and the data structures are destructed promptly can minimize data exposure to swap spaces. Encrypting data before it is sent to storage can reduce (or eliminate) exposure of data to caches. In some environments these operations are redundant because exposure in temporary storage areas is prevented by the operating system, but skeptical programmers need not rely on that.

3.5. Preventing Hidden Features

We now make a decided shift to address an issue has been at the forefront of many programming discussions. That is, how can we prevent programmers from incorporating unwanted, possibly malicious, features into programs that they are assigned to write? Thus, we are talking about programming techniques to protect clients FROM programmers.

Two examples of malicious features are trap doors and penny shaving. Trap doors are mechanisms that allow the programmer access to the system outside the normal authorization mechanisms. Trap door access is intended to be undetectable and to provide the highest priority and broadest levels of access.

Penny shaving involves applications that manage some resource. Programmers may enter code that allows them to divert a very small [micro] portion of the resource from each transaction for their personal use or redemption. Of course, this code is intended to be unidentifiable and to operate covertly.

Coding techniques cannot prevent excess features such as trap doors and penny shaving. The best chance for this is presently entwined in rigorous development processes that couple a structured review process and incorporate verification tools with software coding.

Where coding practice can contribute to security in this area is by making functionality more evident from the program's static representation. Standardization based upon templates and patterns can help make deviations stand out during the review process, thus allowing detection and removal of malicious (or other non-specified) functionality.

3.6. Tamper-proof software

Protecting programs from illegitimate use is a classic problem in computer science [18, 19, 20, 21, 22, 23], both as a matter of program security and of digital rights management.

Tamper-proof techniques have been considered as a means to protect software that executes on remote hosts. It turns out to be a very difficult problem to protect program execution, manipulation, and copying in an environment that is controlled by a sophisticated adversary.

Program obfuscation is one approach to tamper-proofing, though systematically strong obfuscation is generally considered to be impossible [24, 25]. Still, approaches based on complex program control flow [21] and others on homomorhpic encryption [22] reflect some progress in this area.

3.7. Security Systems

We intentionally left this class of techniques until last. Information security is a discipline in itself dealing with the study of mechanisms for meeting all shapes and sizes of security requirements. Cryptographic systems and approaches to provide privacy, integrity, authentication, nonrepudiation and combinations therein are interesting and applicable to this discussion, but are omitted here for lack of space.

Suffice it to say that the basics of information security are essential for any comprehensive computing science curriculum. These basics include the fundamentals of cryptography, cryptographic protocols, encryption systems, information assurance, principles of privacy, legal and ethical issues, and physical security.

4. Teaching Security-Aware Programming

In spite of the attention that has been given to secure coding recently, little has changed in programming curriculum to reduce vulnerability in our software systems. Here we propose concrete steps that can be taken to influence entry level programmers to develop software that has more positive security and reliability properties.

4.1. Partitioning Rigor

The foundation of our approach is to partition applications into six classes consisting of three levels of rigor as seen from each the security and reliability perspectives. Entry level programmers should be taught the difference between disposable software (robustness level 1) and "moon shot" (robustness level 3) systems. They should be taught mechanisms for development expediency, so that prototype and proof of concept implementations can be quickly and efficiently built. They should also understand the rigor necessary to develop critical applications such as embedded software in life support, surgical, or weapons systems.

Presently, the instructional emphasis has been on robustness level 2 systems (all systems that are not level 1 or level 3) that classically categorize the vast majority of all information systems. That balance is changing. The impact of buffer overflow problems has begun to convince developers that systems once considered level 2 are actually level 3 simply because errors in them can impact other (possibly level 3) applications, multiplied by the scale of the Internet.

Similarly, we posit that security conscious systems reflect three basic levels. The first level is that effected by application of rigorous program protection techniques (such as those described in sections 3.2 and 3.3). These are techniques that do not require any special security training, but that do provide a security specific function (such as privacy protection through garbage collection).

Our second level of security rigor is for systems that have software-specific security consideration, such as systems where the code is vulnerable to modification or copying or where the system is particularly vulnerable to software that may contain hidden, malicious functionality (sections 3.4 and 3.5)

Systems that require level 3 security rigor contain sensitive information whose compromise may result in high impact consequences, such as loss of life or significant resources. These systems demand cryptographically strong techniques for authentication and access control.

4.2. Security Aware Programming in a Computing Science Curriculum

There is a natural correlation between the rigor partitions presented in the previous section and a standard professional programming curriculum. It turns out that the lower rigor levels are the most applicable (and understandable) to junior, entry level programmers. Conversely, the higher levels are more naturally incorporated into more advanced, theoretic programming and security courses. Table 1 summarizes our pedagogy based on the hierarchical rigor model.

Course	Robustness Rigor	Security Rigor
Entry level	1,2,3 (overview)	
2nd course	2, 3	1, 2
Data Structures	2, 3	1, 2
SWE	1, 2, 3	1, 2, 3
PL	1, 2, 3	1, 2
Security	3	2, 3

Table 1. Development and Security Rigor

Specifically, we propose to introduce the three hierarchical robustness levels at the entry programming course. More in depth study of level two and three rigor follow in the second programming course. Programming constructs and foundations are evaluated in the programming languages course and their place in the development process is addressed in the software engineering course.

Security rigor is investigated primarily in the upper level programming courses and in the security course. Programming practices required for secure software are considered in all programming courses and their importance in this area is emphasized. Discussion of advanced topics is reserved for programming languages, software engineering, and security courses.

5. Conclusion

In this paper we present a model for analyzing, measuring, and teaching programming rigor. Our model is based on a hierarchical partitioning of software rigor categories for robustness and security. These categories form the basis of a new approach to teaching security-aware programming or coding techniques.

We give an approach for teaching appropriate security-aware concepts in a professional programmer curriculum and map the skills and concepts to specific courses.

Software vulnerability is second only to identity theft as the main security problem of the modern Internet. We propose an approach to reversing the trend that is inexpensive and consistent with existing and known successful programming practice.

6. References

[1] Howard, M. and LeBlanc, D., *Writing Secure Code*, Microsoft Press, Seattle, WA, 2002.

[2] Viega, J. and McGraw, G., *Building Secure Software: How to Avoid Security Problems the Right Way*, Addison-Wesley, Boston, MA, 2002.

[3] Ghosh, A. and O'Connor, T., "Analyzing Programs for Vulnerability to Buffer Overrun Attacks", *Proc. of the 21st NIST-NCSC National Information Systems Security Conference*, 1998.

[4] Haugh, E. and Bishop, M., "Testing C Programs for Buffer Overflow Vulnerabilities", *Proc. of the 2003 Symposium on Networked and Distributed System Security (SNDSS 2003)*, Feb. 2003.

[5] Hoglund, G. and McGraw, G., *Exploiting Software: How to Break Code*, Addison-Wesley, Boston, MA, 2004.

[6] Cheetham, C. and Ferraiolo, K., "The Systems Security Engineering Capability Maturity Model", *21st National Information Systems Security Conference*, October 5-8, 1998, Arlington, Virginia, USA.

[7] McGraw, G., "Software Security", IEEE Security and Privacy, vol. 2, no. 2, March/April 2004, 80-83

[8] Ghosh, A, Howell, C., and Whittaker, J., "Building Software Securely from the Ground Up," IEEE Software, vol. 19, no. 1, January/February 2002, 14-16.

[9] Lee, Y., Lee, J., and Lee, Z., "Integrating Software Lifecycle Process Standards with Security Engineering", Computers and Security, vol. 21, no. 4, 2002, 345-355.

[10] IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology.

[11] Plakosh, D., "Coding Flaws That Lead to Security Failures", 2nd Annual Hampton University Information Assurance Symposium. April 2005.

[12] Peteanu, R., "Best Practices for Secure Development", citeseer.ist.psu.edu/peteanu01best.html, June 2005.

[13] Ghosh, A. and Voas, J., "Inoculating software for survivability", Communications of the ACM, vol. 42, no. 7, 1999, 38-44.

[14] Meyer, B., "Principles of language design and evolution", Proc. of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoar, Millenial Perspectives in Computer Science, 2002, 229-246.

[15] Schwartz, J., "Object Oriented Extensions to Ada: A Dissenting Opinion", Proc. of the Conference on TRI-ADA '90, Baltimore, Maryland, December 03-06, 1990, 92-94.

[16] Lehrmann-Madsen, O., Magnusson, B., and Möller-Pedersen, B.,"Strong Typing of Object-Oriented Languages Revisited", Proc. OOPSLA and ECOOP, ACM Press, New York, NY, October 1990, 140–150.

[17] Tomatisa, N., Brega, R., Rivera, G., and Siegwart, R., "May You Have a Strong (-Typed) Foundation: Why Strong-Typed Programming

Languages Do Matter", Proc. of the International Conference on Robotics and Automation, New Orleans, April 2004

[18] David Aucsmith, "Tamper Resistant Software: An Implementation", Proceedings of the First International Workshop on Information Hiding, Pages: 317-33, 1996, LNCS 1174

[19] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In Proceedings of the 9 Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOSIX) , pages 169--177, November 2000.

[20] David Lie, John Mitchell, Chandramohan A. Thekkath, Mark Horowitz", Specifying and Verifying Hardware for Tamper-Resistant Software", 2003 IEEE Symposium on Security and Privacy May 11 - 14, 2003 Berkeley, CA. p. 166

[21] Toshio Ogiso ,Yusuke Sakabe,Masakazu Soshi,and Atsuko Miyaji, "Software Tamper Resistance Based on the Difficulty of Interprocedural Analysis", WISA 2002, Cheju Island, Korea, August 28-30, 2002

[22] Sander, T., and Tschudin, C.F., "Protecting mobile agents against malicious hosts", 'Mobile Agents and Security', Lecture Notes in Computer Science, Vol. 1419, SpringerVerlag, 1997, pp. 44-61.

[23] T. Sander, and C. Tschudin, "Towards mobile cryptography." Proceedings of the 1998 IEEE Symposium on Security and Privacy, Los Alamitos, CA, USA: IEEE Comput. Soc, 1998. p.215-24.

[24] [NAL] L. D'Anna, B. Matt, A. Reisse, T. van Vleck, S. Schwab, P. LeBlanc. "Self-Protecting Mobile Agents Obfuscation Report". Network Associates Laboratories, Technical Report 03-015 (final), June 30, 2003.

[25] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, K. Yang. "On the (Im)possibility of Obfuscating Programs". In Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology. LNCS, v. 2139, pp. 1-18. 2001.