Foundations for Security Aware Software Development Education[1]

Alec Yasinsac[1], J. Todd McDonald[2]
Florida State University, Tallahassee, FL, USA
yasinsac/mcdonald@cs.fsu.edu

Abstract

Most instances of software exploitation are really software failure. Even though we cannot eliminate vulnerability from modern information systems, we can reduce exploitable code long term with sound, robust development practices. We argue that the current hot topic of so-called "secure coding" represents commonly taught coding techniques that ensure robustness, rather than ensuring any commonly understood concept of security. Weaving the practice of rigorous coding techniques into curriculum is essential—coding for security is useless apart from fault-tolerant foundations. However, security-specific coding techniques need to be integrated pedagogically alongside robustness so that students can differentiate the two. We propose in this paper a shift in instructional methods based on this distinction to help future programmers, developers, and software engineers produce "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 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 system after system, usually after the fact and usually addressing only symptoms of a greater problem in code design and implementation.

While these fixes are labeled "security patches", they are actually 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. Most buffer overflow exploits use program flaws to gain access privileges they would otherwise not have, though the resulting mischief or malice are rarely directed at the vulnerable application itself.

So, what is termed software exploitation currently is merely a veiled manifestation of software failure. We can view security 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. From the robustness perspective, problems identified as security flaws can be seen more as problems stemming from poor programming practice than from a security threat. Without laying a proper foundation in good fault tolerant coding practices, integrating *security* goals into software has little value.

We propose a needed paradigm shift in educational environments that delineates and reinforces principles of *robust* and *secure* coding. Though distinct pedagogic concepts, security builds squarely upon robustness and both should be strategically incorporated into computer science curriculums. Strengthening the foundation of what and how we teach future programmers about robust practices will provide a necessary foundation to incorporate security specific goals.

[1] This material is based upon work supported in part by the U.S. Army Research Laboratory and the U.S. Army Research Office under grant number DAAD19-02-1-0235

[2] The views expressed in this article are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government

In section 2, we build our argument by considering necessary curriculum changes that highlight security-aware programming in both professional practice and educational patterns. In section 3 we address concepts that will build foundational support for *robust coding*. Section 4 details knowledge areas that are needed for *security-specific* requirements themselves. Section 5 gives our conclusions and summary of contribution.

2. Teaching security-aware programming

Hogland and McGraw [5] assert that "bad" software is the root of the security problem—and others agree security problems can only be fixed by building robust and survivable software. In order to influence the next generation of programmers toward security awareness, we believe that educational programs must teach the proper relationship and distinction between *robust* and *secure* coding practices. In professional practice, work to integrate security into software systems is well underway. We use process level maturity as a means to introduce our suggested patterns for education.

2.1. Classifying robustness

The systems security engineering capability maturity model (SSE-CMM)[3] 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]. Higher level managers such as chief information officers, administrators, and infrastructure planners have begun to see the value of integrating security into the entire lifecycle development process instead of dealing with it as an afterthought [8, 9]. In terms of education, specific techniques for coding need to be presented and taught in the light of CMM-related lifecycle concepts and security-related processes—early and as often as possible.

The amount of rigor a professional development organization puts into its software process is tied to the nature of the computer programs they develop. We consider three notional levels of rigor that are logically tied to a likely organizational level of capability and process level maturity (CMM level).

2.1.1. Disposable software. At the lowest level of required rigor, disposable software is "here today and gone tomorrow". Such programs gain less benefit from maintenance or lifecycle cost assessment. They are written to get a specific job done in a short amount of time—with no view for long term use. Robust coding practices can still be of benefit at this level (especially when short-term usage assumptions are wrong), but the primary focus of the system is on-time delivery or features—not necessarily quality.

2.1.2. Non-disposable software. While disposable software can be produced by any CMM-level 1 organization, most software systems are developed with expected longer term use. Rigor should be applied based on the budget, customer base, projected lifetime, and complexity of the system to be developed. Typical software development organizations strive for some level of process maturity to sustain non-disposable software development—striving for CMM-level 2 or 3 maturity to achieve minimum success.

2.1.3. Moon-shot software. The highest level of rigor we term the "moon-shot" system. This is software that is not only long term and requires organization al established discipline for success, but has 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). A fundamental property of moon-shot systems is that it is costly or impossible to test these systems to the level demanded by the application's criticality.

CMM dictates that organizations with development capabilities lower than CMM-level 3 not develop moon-shot systems. In fact, moon-shot systems represent our desire to use the very *best* software development processes in every lifecycle area. Disposable software may not require robust coding principles or integration of security requirements—though using such techniques provide inherent benefit. Moon-shot software, by nature, demands the highest level of development quality controls.

Non-disposable is the category where most software efforts fall. It is essential that present and future programmers understand their role in developing non-disposable software that is not only functionally correct and efficient, but that is robust and designed to prevent faults. Our educational environments, the specific quality techniques that are taught, and the concepts of development and security rigor all help reinforce a security-aware view of programming.

[3] See http://www.sse-cmm.org

2.2. Teaching robustness

Entry level programmers and computer science students 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 (robustness level 1). They should also understand the rigor necessary to develop critical applications such as embedded software in life support, surgical, or weapons systems (robustness level 3).

Presently, instructional emphasis is on robustness level 2 systems (all systems that are not level 1 or level 3) that classically categorize the 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.

There is a natural correlation between the rigor partitions in the previous section and standard professional programming curriculums. It turns out that the lower rigor levels are the most applicable (and understandable) to junior, entry level students and programmers. Conversely, the higher levels are more naturally incorporated into more advanced, theoretic programming and security courses. Table 1 depicts these robustness levels in terms of appropriate courses where these concepts need to be introduced and taught. We consider now the other aspect of security-aware programming which table 1 depicts: techniques that specifically enforce security.

2.3. Classifying software security

Just as categorizing systems allows us to select appropriate levels of rigor and robustness, we recognized that we may similarly categorize security conscious systems, coincidentally, in three levels. These levels are hierarchical and cumulative in the sense that systems that demand higher level security automatically need the lower level techniques. We term these techniques as rigorously robust software protection, threat protection required, and critical systems protection.

2.3.1. General software protection. The first security level includes systems that demand rigorous robustness guarantees. These are systems that do not require any classical security techniques, but where malfunction would be catastrophic. A, somewhat contrived example of such a system may be software the guides delicate instruments used in major organ surgery. While this software may execute in a closed environment, essentially with no security threat, software flaws are a matter of life and death. This example is contrived in the sense that it is likely that some level of security will be included in such a system. The essence of our argument is that software that is not "critically" robust, i.e. sufficiently robust for even the most high risk system, is not sufficiently robust for any security purpose.

2.3.2. Software-specific protection. The second security level is for systems with software-specific security requirements, such as systems that require protection for the data or code for privacy, integrity or availability. These systems range from low risk access control systems (such as for home network access) to medium risk systems where compromise could cause significant financial or business impact.

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

2.4. Teaching security

The software community is gradually acknowledging the need to incorporate security often and early into the development process—but now the educational community must begin to introduce robust and secure coding techniques often and early in the learning process. In spite of attention given to secure coding recently, programming curriculums still need to adapt principles of professional software practice into class-related objectives and goals.

		Course Type					
Hierarchical Rigor		Entry-level Programming	Follow-on Programming	Data Structures	Software Engineering	Programming Languages	Security
Robustness	1: Disposable SW	X			X	X	
	2: Non-disposable SW	X	X	X	X	X	X
	3: Moon Shot SW	X	X	X	X		X
Security	1: General protection		X	X	X	X	
	2: Software-specific		X	X	X	X	X
	3: Critical systems				X		X

Table 1. Integrating Robustness and Security Rigor into Software Development Curriculum

Table 1 summarizes our pedagogy based on hierarchical rigor. Several categories of standard computer science courses are listed across the top rows: entry-level and follow-on programming, data structures, software engineering, programming languages, and security. For each type of course we specify the level of both fault-tolerance and security rigor that should be covered in that course.

For entry-level courses, we recommend giving a general and high-level overview of fault-tolerant software development concepts across the spectrum of quality. In depth study of level two and three robustness rigors are addressed in follow-on programming courses, data structures, and software engineering. Security related courses address higher levels of robustness rigor in the context of other software security-related concerns. Programming language courses should give thorough examples of robustness techniques that support both efficient development for disposable software and longer-term code maintenance for non-disposable software.

In terms of instruction on different levels of security rigor, follow-on programming courses should expose students to the notion of general and -specific software security techniques. Software engineering should incorporate such foundations in context to the overall software lifecycle development plan. Upper level programming and security courses should be used to cover requirements and techniques for level three critical systems rigor and to lay practical foundations for using cryptographically strong protection mechanisms.

The next question to deal with is which robustness and security techniques to teach in these courses. We address this issue in section 3 by detailing sources of fault-tolerant vulnerabilities and avoidance techniques while we detail in section 4 instructional patterns for security-specific coding. Table 2 summarizes our recommendations for course integration of specific areas of professional practice discussed in sections 3 and 4. Instruction on these topics should follow other pedagogical lines concerning *how* to effectively teach programming and computer science related material.[4]

3. Coding for robustness

The cause of many common security faults can be traced to lack of software risk avoidance. Robustness, defined in [10], is the "degree to which a software component functions correctly in the presence of exceptional inputs or stressful environmental conditions". Exceptional program execution cases that malicious parties may exceed any notion of what has traditionally been considered. Intentionally malformed or irrationally long, short, or empty inputs, altered program control flow, dynamically linked or patched code segments, and memory corruption errors are faults that programmers must consider in modern applications.

Educational environments must foster the notion that secure programs are first and foremost reliable and safe programs. By safe we mean that programs clean up after themselves, police their own code and data space, and do not 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 programmer's mantra that the operating system (or other environmental mechanism) clears memory (before or after execution), provides containment, protects input streams, etc. must be changed.

The foundations, however, begin with how coding is presented above the basic functional and semantic levels. We consider several areas of fault-tolerant coding practice and the need for their integration into software development education.

	Course Type					
Programming Practice & Skill	Entry-level Programming	Follow-on Programming	Data Structures	Software Engineering	Programming Languages	Security
Defect Patterns	X	X		X		
Sources of Faults	X	X	X		X	
Testing Methods		X		X	X	
Risky Coding		X			X	
Robust Coding Techniques	X	X	X	X	X	
Secure Coding Techniques		X	X	X	X	X
Memory-Related Vulnerability			X		X	X
Secure Data Initialization			X		X	X
Operating System & Process Security				X	X	X
Template and Pattern Programming				X		X
Tamperproofing				X		X

Table 2. Integrating Robust and Secure Coding Practice into Software Development Curriculum

3.1. Risky defects

Research continues to identify coding weaknesses and vulnerabilities that are security related, *so-called* [1,2,5,8,11,12]. By definition, software vulnerability

[4] See http://www.pedagogicalpatterns.org/ where example patterns of computer science instruction are given

is a defect in either design or implementation of code—and 90% of vulnerabilities derive from exploiting known defect patterns in coding [11]. As table 2 depicts, practical defect avoidance by identifying failure causes such as defect patterns/sources of faults, and teaching appropriate remedies, including robust coding techniques, should be introduced early in programming coursework.

3.2. Sources of faults

Buffer overflows have been called the "nuclear bomb of all software vulnerabilities" [5]. They result from programming errors that allow memory to be corrupted. When data structures are not properly protected, data is written outside of memory boundaries. Once corruption occurs, a wide variety of attacks are possible including overwriting critical program information, changing global state, removing security restrictions, or disabling program controls.

In programming languages such as the C language, string handling routines that assume the presence of the NULL terminator provide a ripe environment for memory overflow attacks. This library design choice intended to simplify programming (i.e., you don't have to manage the size of a string yourself) has become a security nightmare. The real problem, however, is curriculums do not stress that such choices are invalid fault-tolerant assumptions.

Though security conferences, publications, and books call for security awareness and bring attention to software vulnerabilities, a more compelling issue may be programmer laziness. For example, it is not safe to assume a NULL terminator in a C program will always indicate the end of a string. Instead, programmers need to manage and verify string sizes for themselves. Furthermore, it is the programmer's job to make sure memory [buffer] operations stay within their bounds. Table 2 indicates that sources of such faults should be covered in introductory and follow-on programming courses as well as language-specific instruction.

Because data and memory locations are stored on the stack, redirecting program control flow can be simple once an appropriate weakness is discovered. Buffer overflows can be exploited to modify variable pointers or function return addresses on the stack. Such modifications can alter program behavior and application data or may execute viral code. Nonetheless, these results are problems that concern robustness, not security.

Traditional testing methods have been overhauled to address these software problems. Fault injection techniques [3,4,13], for example, have been used for quite some time to identify and root out buffer vulnerabilities. No matter how novel or comprehensive, testing can not guarantee absence of data vulnerabilities in a given program. Testing methods, especially for robustness, need to be introduced in programming and language specific courses and thoroughly reviewed in software engineering courses (see table 2).

Another class of vulnerabilities that should be covered by programming curriculums involves integer manipulation errors and truncation [11]. Code that performs numeric computations is naturally susceptible to underflow, overflow, signed numeric errors, and truncation of data bytes because of smaller data type capacities. These 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. Even whth disposable software, software engineering and programming languages courses should identify such vulnerabilities as robustness issues, while reinforcing the connection of these weaknesses with malicious exploitation.

Memory leaks are another source of problems that revolve more around survivable code than security. Leaks occur when programmers practice poor memory management, resulting in the operating system being unaware of memory that should be free and available. These hidden 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.

The risks described in this subsection are not security-centric in nature. They stem from failure to validate user input or prevent users from (mistakenly or intentionally) providing erroneous input or formatting strings to the program. Table 2 summarizes our recommendations for courses where these concepts should be introduced.

3.3. Risky coding techniques

An important aspect of modern programming languages is that they simplify the programmer's job by reducing the required knowledge of the underlying hardware environment. Still, powerful language features may not be helpful if they cause problems in stressful or abnormal environments where malicious parties can find vulnerability 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 take advantage of this facility for malicious purposes.

Aside from the security threat, polymorphism critics have 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. While such programming features are powerful, they are not conducive to reliable software. As such, curriculums need to promote the use of safer and more reliable programming techniques in lieu of powerful, but risky, language features like polymorphism.

As another example, consider dynamic memory allocation. It is heretical in computer science to regress to using static data structures. However, there is a strong case to be made that static allocation of program resources ultimately leads to more reliable code. Dynamic allocation is a powerful, but complex tool. Incorrect computations by the application programmer routinely result in both of the two major memory problems: overflows and leaks.

Dynamic memory allocation can be incompatible with both program predictability and is potentially non-deterministic—qualities that fault-tolerant software should avoid. Our education process again must change to teach not only the functionality of languages, but also the inherent reliability risk that comes with certain language features. Table 2 depicts our recommendations for which courses should cover risky coding techniques along with the specific practices that increase robustness, discussed next.

3.4. Techniques for robustness

Robust programming methods demand that programmers 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, introductory programming to high level software engineering, so that reliable coding becomes the foundational premise on which other, security-related, techniques can be built.

Type systems have been debated over the years in terms of whether they increase or decrease programmer productivity, code reliability, and reduce software faults. *Type systems* of programming languages can be characterized as strong or weak while *type checking* occurs statically or dynamically [16]. Strong typing dictates that that all types for variables and data structures must be defined at compile time. We agree with such findings in [17] that using strong typing leads 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 property of strongly-typed languages [16].

Another simple technique to increase robustness involves avoidance of variable length fields. Any data field whose length is determined dynamically reduces the verifiability and safety of a program. At a high level, teaching environments should encourage future programmers to verify as much of a program's data structure as possible before execution.

A third technique for robust programming reduces environmental assumptions: always filter input. Input validation to ensure 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 can reduce run time faults. As a side effect, security threats are also reduced.

No discussion of robustness and security would be complete without addressing testing. Extensive and systematic testing must be 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 programmers. Tools for checking code correctness need to be introduced at the same time that compiler features are 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 do not limit quality, reliability, and safety-specific efforts. 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 input format 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.

4. Coding for security

Some programs may not rely on 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 in security-aware practices.

It is easy to understand why it has taken so long for security issues to become incorporated into software practice, let alone education. 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[5]. 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 heterogeneous 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.

In terms of educational paradigms, these issues must be addressed and incorporated into the computer science learning process—where future analysts and programmers are birthed. A sense of both high quality and secure coding practice must be affirmed in programming curriculums, from beginning to end, if the tide is going to be turned. Table 2 summarizes our recommendations for incorporating security-related skills and practice for software development[6] and we discuss next the needed shift in educational philosophy by highlighting security-specific coding techniques.

4.1. Caveat emptor

While we have emphasized the importance of rigor in modern applications programming, we also point out that effective software security demands skeptical programmers. In modern applications, a wide variety of forces determine what is normal and abnormal. Educational processes need to foster a healthy but realistic view of program security. Programming students must be taught to think like security specialists: be skeptical, question the simplest of assumptions, resist depending on uncontrolled factors, and verify, verify, verify.

We propose that introducing security-related programming practice across a wide spectrum of courses will reinforce the idea that programmers are the front-line of defense against software exploitation. Future programmers and analysts must be keenly aware of the forces that cause security and robustness to be overlooked, while becoming practitioners of security-relevant coding.

---

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

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

Table 2 lists several “good” professional practices that have security related impact: reducing memory-related vulnerabilities, secure data initialization, operating system and process security, template and pattern programming, and tamperproofing. Next, we discuss each of these concepts to expose their educational relevance; table 2 correlates appropriate courses where these practices are best integrated.

4.2. Garbage collecting

One of the easiest places to implement controlled skepticism is through aggressive garbage collection. Items left over from program execution can offer 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 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 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 feature).

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 intruders to reveal residual data or other malicious intent.

4.3. 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.

4.4. 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.

4.5. Preventing hidden features

We now make a decided shift to address an issue has been at the forefront of many programming

discussions: preventing 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 software features are trap doors and penny shaving. Trap doors are mechanisms that allow the programmer system access outside the normal authorization mechanisms. Trap door access is intended to be undetectable and to provide high priority and broad access levels.

Penny shaving involves applications that manage some valued 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, per se. Rigorous use of well-designed templates can improve chances of detecting malicious "features", but 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.

Coding practice can contribute to protecting against malicious features 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, allowing detection and removal of malicious (or other non-specified) functionality.

4.6. 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 may be used 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 a general program obfuscation approach remains elusive [24, 25]. Still, approaches based on complex program control flow [21] and others on homomorphic encryption [22] reflect progress in this area.

4.7. 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 security requirements of all shapes and sizes. Cryptographic systems and approaches to provide privacy, integrity, authentication, non-repudiation and combinations therein are interesting and applicable to this discussion, but are omitted here for lack of space.

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.

5. Conclusion

In this paper we present an approach for analyzing, measuring, and teaching programming rigor that result in robust and secure systems. Our approach is based on 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 software 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.