Learning and Experience in Computer Security Education

Matt Bishop
Department of Computer Science
University of California at Davis
Davis, CA 95616-8562
Email: mabishop@ucdavis.edu

***Abstract*—Computer security is a discipline highly dependent on the environment in which systems and sites are to be secured. But the practical experience needed to understand the limits of abstract knowledge in the field, and to mould that knowledge in a way that can be applied to specific situations arising in practice, is often not taught in academia. Non-academic institutions, including sites that use security to protect themselves and organizations and companies that develop security tools, technologies, and practices, can help close this gap in a way that benefits the organizations, the academic institutions, and the students. An example using the current lack of security and robustness in software shows how this might be done.**

I. Introduction

Computer security draws many of its most difficult problems from the realm of practice. The details of the practice create the problems, and they often arise from non-technical considerations. Perhaps the best example of this is the hierarchical public key infrastructure (PKI). Initially, a proposed hierarchy had a single root node. Conceptually, this makes the hierarchy a tree, and therefore (relatively) simple and clean. It also requires all certification authorities to be certified by that root, either directly or indirectly. This implies that all those CAs trust the root to some extent. But in reality, such an assumption is untenable. One need only look at the politics of the world to understand that some nation-states will never trust an entity not under their control. This led to the "forest" notion that currently predominates, with several root CAs. When appropriate, the root nodes certify one another (called "cross-certification"). Technically, a single trusted root CA suffices; in practice, no such root exists, requiring the development of alternate mechanisms.

Academic education focuses on principles, which by their nature are abstract. For example, the "principle of least privilege" says that a process should have the minimal set of privileges needed to carry out its tasks [12]. In a formal model, this is straightforward. Simply define rights that enable the subject to access the object as needed, and provide those rights to the subject (in an access control list or capability list). But in a real system, the architecture becomes critical. First, what is the granularity of the subject with respect to the resource? In a UNIX-like system, if the subject is not the owner of the object, it shares permissions with other subjects, so the permissions assigned would be the union of the permissions that each subject needed. This may be considerably more than what the subject in question needs. Second, how are the rights constrained by environmental considerations? In that same system, if the subject has (say) read rights over the object, it will not be able to exercise those rights unless that subject has search (execute) permission on the containing directory. Thus, the "clean" model in which least privilege can be applied exactly fails in this situation.

The result is a need to understand how security works in practice, so that we know how to apply existing models to improve security, and to improve the models to reflect practice better. In the PKI example, cross-certification fits naturally into existing models: the cross-certified root CA becomes a node in the hierarchy directly under the cross-certifying root CA. In the access example, the access control model must be augmented to take "groups" of subjects into account, as well as cascading permissions; or, the UNIX-like system must be augmented to handle rights on a per-object basis, and the incorporation of ACLs into most existing UNIX-like systems enables exactly that.

Incorporating practice into education helps bridge this gap between theory and practice. Giving students practical experience in which they can apply the abstract theories, principles, and analyses they learn in class brings those theories to life. In reverse, the students can take what they have done or are doing in practice, and incorporate them into new theories and models, or modify existing ones to reflect the practice better.

Working together, academic institutions and non-academic organizations can provide this combination of theory and practice. Further, working together may help compensate somewhat for the damage caused by the pervasive lack of resources in both non-academic organizations and academic institutions damages both. Indeed, such joint work may even provide a basis for seeking additional resources.

The goal of this paper is to examine ways that universities may interact with non-academic organizations to support education. First, we describe the goals of this interaction based upon the nature of the non-academic institutions; then we discuss the methods of interaction. We next use the notion of "secure programming" as an example of how this co-operation might work. We conclude with some thoughts on the benefits of academic institutions working with other organizations to enhance their educational programs.

II. Goals of Interactions

Academic institutions and other institutions (which we collectively call "organizations", and includes industrial, commercial, non-profit, governmental, and international organizations) can interact in several ways, depending on the goals. The benefits of working with an organization that is a "producer" of security products or services (for example, a company that sells anti-virus software or does penetration testing) are slightly different than those of working with a "consumer" of those products or services (for example, a company that hires a security firm to install a firewall, or that has purchased a firewall and maintains it without outside aid).

A. Goals for Consumer Organization

Consumer organizations span many disciplines, ranging from the simple (such as dentists' offices or small companies) to the critical (such as banks, hospitals, and many government agencies). Almost all these organizations are connected to the Internet; and like them, even those organizations not connected are concerned about security. So, one goal for students working with these industries is to learn about the environment in which the particular organization functions, and how that environment affects security.

As an example, a military organization wants control over the information visitors to its web site can access. So, if the organization determines that visitors to its web site may access some sensitive information, the web site can be deactivated until the nature and seriousness of the damage can be determined. But if the organization is an Internet vendor such as Amazon or Ebay, that organization wants visitors to come to the web site and buy something. The information on the web site is not sensitive to disclosure, because the business model of the organization is to disclose (in the hopes that someone will bid on or buy the product). The threat is not disclosure; it is integrity, where an adversary might alter the information about the item being sold. Similarly, the response to an attack is different because blocking access to the web site will prevent legitimate purchasers from buying their wares, and hence impact their revenue stream. So it is simply not acceptable. This is an example of why the phrase "securing cyberspace" is glib. What *exactly* does it mean? The answer depends on what type of "security" is of concern, and organizations in "cyberspace" have wildly differing notions of what "secure" means.

In addition to the context defining security, organizations may implement the same security goals very differently. Two sites want to have a web server available for the public, but also want to protect their internal systems. The traditional approach is to set up a DMZ. One way to do this is to have two firewalls, one between the Internet and the DMZ, and the other between the DMZ and the internal network — effectively, the DMZ is a buffer between the Internet and the internal network. A second way is to have a single firewall connected to the Internet, the DMZ, and the internal network. The firewall determines whether to route messages to the DMZ, the Internet, or the internal network based on its rules. Both designs meet the security requirement of isolating the web server (which is in the DMZ) from the internal network; each design has advantages and disadvantages [7].

Affecting all this are the human considerations — personal, organizational, political, and legal — that are typically mentioned but not explored in detail. Organizational structures can affect the effectiveness of security. The role of policies and procedures is often overlooked or misunderstood. For example, much of the work analyzing electronic voting systems focuses on the vulnerabilities of the system, without taking into account procedures that may mitigate or aggravate those vulnerabilities.

Finally, academic examples focus on parts of the implementation of systems. Systems change rapidly, relatively speaking, and students must be prepared to adapt to those changes. Also, academic education does not cover all systems, so students will be exposed to systems in the work place that they have not seen, and yet must manage.

To really understand these aspects of computer security, students must be placed in an environment where they can experience their effects. Academic institutions can teach about the effects, but in general students get this experience by *doing* — by actually dealing with these problems in real-world situations. Interacting with non-academic organizations provides such experience.

Summarizing, some goals of interacting with consumer organizations for enhancing education are:

1) To experience how the technical and non-technical considerations affect the security requirements;
2) To learn how those same considerations affect the implementation of those requirements; and
3) To put into practice the theory studied in school, whether the specifics of the systems being used were discussed in class.

Ideally, the interactions will provide students with experience in evaluating and remediating security threats holistically.

B. Goals for Producer Organizations

Producer organizations are also consumers of security, because they must protect their assets. So the above goals also apply to them. In addition, producer organizations develop, deploy, maintain, and retire software, hardware, tests, and procedures to enhance security. This provides additional opportunities for students to enhance their classroom and project knowledge.

Producer organizations must either identify or develop a market for their products and services. Market development relies as much on perception as fact; if something is *perceived* to be a threat, then the market for dealing with this threat grows in proportion to the spread of the perception. Thus, understanding what (current and prospective) customers consider threats is part of product development. Note that perceptions may or may not reflect the realistic nature of the threat, but even if the perception is misplaced, it may speak to the credibility of the organization's efforts to protect its assets. Credibility in public life is sometimes just as important. See for example

recent work in electronic voting systems that provides voters with the opportunity to check that their votes are recorded and counted correctly.

As part of identifying new threats, producer organizations must analyze new attacks on systems and client organizations. For example, anti-virus providers have quickly analyzed malware such as Stuxnet [8], Duqu [1], and, more recently, Flame [13]. Forensic groups examine detritus from attacks to determine how attackers got in, and what they did. Doing these analyses in an academic setting is excellent experience; doing it "in the field" adds additional dimensions such as complexity, time pressure, and the need to explain to non-technical people what happened. Also, academic exercises usually take place in a constrained environment, known to the students. Analysts elsewhere often work with incomplete information, and do not know specifically what is missing, so they must either work with victims or make assumptions. Few academic institutions can provide this experience without help from industry.

A third problem is the quality of the product or service. Most software is poorly written. The problems usually lie in the implementation of the design. A non-robust implementation, for example one that does not check inputs or makes assumptions that can be checked yet are not, can make the best-designed security tool a danger. Universities teach good programming practices in the first, and sometimes the second, programming class. After that, though, the students' programs are checked to see if they work, and few if any are rewarded for excellent code (or penalized for non-robust code). Producer organizations have been embarrassed when security-related software has been used to compromise systems due to non-robust programming. They often develop coding standards, and do quality reviews, before shipping a product. Undergoing such a review or being forced to adhere to specific programming standards, is experience that academic institutions all too rarely give students.

Summarizing, some goals of interacting with producer organizations for enhancing education are:

1–3. The goals of consumer organizations, above;
4. To develop requirements to meet specific market needs and pressures;
5. To identify and analyze new threats in an environment where time is critical; and
6. To design, implement, and deploy robust software.

III. Methods of Interactions

Academic institutions and organizations can work together in a number of ways, depending on the specific goals of the interaction. Each benefits from such an interaction, as do their students and customers. We consider three methods: internships, joint research, and adjunct or guest involvement.

Internships, or co-ops, are programs where students work with an organization as an employee or a trainee. A good internship will expose a student to the practice of the principles, concepts, and mechanisms they learn in the classroom. The students will also learn what they do not know, and are expected to know, when they work for such an organization. Undoubtedly the organization will also talk to the academic institution, especially the faculty supervising the academic aspects of the internship program, of deficiencies. The faculty can then take this into account when planning the institution's courses. In this way, the academic institution benefits from the internship program. The benefits to the organization are short-term and long-term. In the short term, they get a worker (the student) who is well educated in the academics of the job, usually at much less cost than for a normal employee. In the long term, the organization's managers can look over the intern and decide whether they wish to recruit her after graduation. More broadly, that organization can influence the way that the academic institution teaches relevant courses by providing examples and ideas related to the job.

A simple way to do this is to have organization members give guest lectures or help run laboratory exercises. The students do not get the full experience of seeing how the academic work translates into what is used in industry and government, but the non-academic lecturers often provide a view of the field and of the material that many faculty cannot provide. This has a second benefit. The approach of the organization can be critiqued, and through a dialogue with the students, the guest lecturer can discuss the reason for that approach. Possibly the students will offer ideas that the organization can use to improve their approach.

Having members of the organization run laboratory exercises allows the faculty of the academic institution to provide much more realistic exercises, and students enjoy seeing the application of their work. As an example, UC Davis was once given a set of electronic voting systems to test, and the election officials gave a guest lecture about the process in which those systems were used. They then helped guide the laboratory exercises of applying the Flaw Hypothesis Methodology [9], [15] to those voting systems. As a result, several potential vulnerabilities were found, including one that the voting process would not protect against (specifically, a denial of service attack enabling a voter to crash the e-voting system). When advised of the problem, the election officials changed the process to prevent the voter from launching the attack. The students were thrilled that a classroom project helped improve the security of such a critical civic process.

The limiting case of this interaction is having the guest lecturer teach a course. Such a course would presumably focus on practice, and provide the students with experience in applying the concepts learned in other classes. The critical aspects of this course are how the theory is integrated into the practice — from an academic point of view, that integration is critical — and how closely the practice reflects what is done by organizations and practitioners in the field.

More generally, having members of a non-academic organizations work with students in a classroom setting requires that the instructors make the link between the more abstract principles and concepts and the real-world practice.

Unlike internships, guest lectures and assisting with laboratory exercises will not achieve all the goals described in section II. Specifically, much of the immersion (goals 1, 4,

5, and 6) will be a simulation or a partial instantiation of the actual organization environment. This is often enough to capture student interest and provide experience that will prove useful later in their career.

A third type of interaction is joint research. The organization itself may fund the research, or the organization and academic institution may jointly apply to a funding agency (for example, a government department or a foundation). This is particularly appropriate for graduate education that focuses on research. Students and faculty involved in the research will interact with organization members to learn how the company or agency approaches and defines problems, and what limits it has on solutions. Having students involved in writing the proposal emphasizes these points to them even more. Like an internship, the organization members will see the student at work, and can decide whether to recruit her as an employee after graduation. Further, the organization reaps the fruits of the research while supporting education in general and building bridges to the academic institution. Producer organizations may find new algorithms or results that markedly improve their products; consumer organizations may devise new processes and procedures to enhance their security, or learn about alternate approaches to protecting themselves. The nature of the research project determines which goals are met.

A benefit to all these methods of interaction is that the students gain (varying degrees of) practical experience. If the students want to go into non-academic positions, that experience will help them get better jobs than had they not had the experience. If they want to go into academic positions, they can draw on their practical experience to enhance their courses. Presenting this experience to show how principles and concepts are translated into practice, and to show how non-technical considerations affect security, enriches students' understanding of the subject.

We now turn to a problem that is plaguing academic and non-academic institutions alike, and show how industry involvement in academia could provide a basis for ameliorating the problem.

IV. Robust Programming

Perhaps the most common complaint about software is its low quality. It does not do what the user wants, it is difficult to use, it is prone to crashes, or it is easy for attackers to exploit flaws in the software to obtain privileged access (or otherwise cripple the system). This complaint is not new; indeed, the psychologist Gerald Weinberg formulated his second law in the mid-1970s: "If builders built buildings the way programmers wrote programs, then the first woodpecker to come along would destroy civilization".

This is especially true of security. Except in unusual circumstances, we do not build systems that meet security requirements in all cases. They may meet the requirements for the vast majority of inputs, but the few inputs that cause the properties to not be satisfied cause the security policy to be violated. In normal programming, a flaw is called a "bug". When security is involved, a flaw that affects meeting the security properties is a "hole"—and an attacker need find only one to compromise the system.

Definition. *Robust programming* is a style of programming that prevents unexpected actions, including abnormal terminations.

A robust program (library) handles bad inputs (arguments) in a reasonable way. If an internal error occurs, the program (library) terminates gracefully, and provides enough information so a programmer can debug the program (library). It adheres to four principles [3]:

1) Be paranoid — check any data the program does not generate to be sure it is not malformed or incorrect;
2) Assume stupidity — handle incorrect, bogus, and malformed inputs and parameters, and return unambiguous and detailed error indicators;
3) Modularize — disallow access to anything expected to be unchanged across invocations; and
4) Never say "can't happen" — check that "impossible" conditions do not occur, because changes may make them possible.

Definition. *Secure programming* is a style of programming that satisfies (stated or implicit) security properties.

A secure program cannot violate specific properties, called "security properties". These properties may be explicit (such as a property that authentication always precede the granting of privileges) or implicit (such as the property that all buffer bounds are checked). Typically, the implicit properties include those required for a robust program.

In what follows, "good programming" produces robust, secure programs. "Poor programming" does not.

When students learn to program in introductory classes, they learn to write programs that are well structured. They learn to check inputs for problems, and to think of what can go wrong and build appropriate error checking into the program. For example, they learn to check that indices do not exceed the buffer length when dealing with buffers. Thus, they become aware of common errors. Further, part of the program grade includes style, so using good programming practices results in a higher grade than not using them.

Interestingly enough, few textbooks in introductory programming cover the principles of robust and secure programming; instructors must do so on their own, or through the use of supplementary material [10]. One way to do this is to gather examples of problems that arose through non-robust or non-secure programming, but often they are very complex and require sophistication to see how the non-robust practice causes the problem. Generally, the instructor must carefully craft examples — and this requires considerable knowledge of how the poor programming practices can be exploited.

Once students begin taking advanced courses, the goal of the programming becomes reinforcement of what is learned in class, or an exploration of details not covered in class. Grading focuses on these aspects — correctness with respect to the concepts or details — and often the program's style

is not examined. Thus, the practice of robust and secure programming is not reinforced, and it atrophies.

Simply saying that those practices must be reinforced begs the question of *how* they are to be reinforced. Here, non-academic organization can play a role both in imparting the knowledge of how to write robust or secure programs, and in showing students why this type of programming is important. Indeed, bringing in examples seen in practice of what non-robust, non-secure programs look like, and explaining exactly how the non-robust or non-secure feature led to a problem, would tie the need for good programming practices to specific class material. Demonstrating the effects of failing to check the length of an input in a system call, and the resulting buffer overflow corrupting the kernel stack, would emphasize the need for operating systems developers to use good programming style while writing the operating system in a way that an abstract discussion of possible consequences could not.

Examples and materials need to be grounded in problems and practices found in organizations. In addition to illustrating the principles, concepts, and methods discussed in lecture, such examples demonstrate the applicability of that material to current events. These problems and practices also help the instructor choose *which* principles to emphasize. They may be the ones tied to the problems and practices. Or, extrapolating from these current problems, they may be ones that would prove useful in the future.

Non-academic involvement enhances learning by providing such examples. For instance, most security classes spend little time discussing usability. But the complexity of configuring a program, a system, or a set of systems to provide the desired types and levels of security is great enough that users make mistakes that open the organization to attack. Worse, errors made by system administrators who install, configure, and maintain systems may leave the systems, and hence the organization, even more vulnerable to attack.

Because modern software is so complex, most organizations use special code-checking tools to enable analysts and programmers to check their code. Having industry-standard tools available for students to use ensures they are familiar with these tools when they graduate. It will also make students familiar with the limits of those tools, because they must learn to distinguish false positives from true positives. Alternatively, using languages and development environments in class that inhibit the use of non-robust, non-secure programming constructs leads to better programs, but these environments come with one danger. They hide the details of robustness and security. This is, ideally, what should be done; but should the programmer ever use a language or development environment that does not provide such support, the student still needs to know how to avoid non-robust, non-secure constructs.

Other types of tools will prove useful. Security problems arise because of failures to maintain a code base, or to update one software module and fail to integrate it properly, or other code development problems. Encouraging students to use industry-standard tools for software development and maintenance, such as source code control systems Subversion (SVN) [11] and Concurrent Versioning System (CVS) [14], and to adopt appropriate controls, will ameliorate this problem.

Human assessment of programs is critical. This requires someone who can review software (possibly with the help of code-checking tools) and point out examples of potential problems, so the students can then learn how to do the analyses themselves. The theory underlying the analyses can be taught in programming language, security, or other classes. The goal of the human assessment will be to ensure the students learn how to check for non-robust, non-secure constructs in practice.

One way to do this is to have a grader (or group of graders) assess the programming style of the software, and others examine it to see how well it fulfilled the requirements of the assignment. The instructor could then compute the grade using an appropriate combination of the two. A second is to provide a "secure programming clinic". The model here is to have clinicians to whom students can bring their programs, and who will work with the student to identify security and robustness problems. Such a clinic can be set up for a particular class, or for all students [4]. The advantage to these methods is that they do not require a separate class or extra lectures. The students learn by doing, and especially from the feedback on their assignments. These techniques can be used with any programming course, or (preferably) all of them.

Staffing issues abound. The analysts could be students with special skills and experience in analyzing code. However, few students have those skills. But non-academic organizations who practice this style of programming do have such personnel. So one way to begin such a clinic is to have those non-academic organizations work with the academic institution to "seed" such a clinic, and train the students who will work in the clinic while helping others detect problems with their programs. In this way, the organizations effectively provide mentors for the students. This benefits the students by giving them time with experienced practitioners. It benefits the mentors (and the organizations they work for) by enhancing their knowledge — paradoxically, the best way to improve one's knowledge is to teach it — and enabling them to identify students for possible future work with, or in, the company.

Many of the above ideas can be applied to non-computer science curricula where students program. Indeed, much software is written in other disciplines, ranging from physics to digital art, that suffers from similar problems. The interested reader is referred to the final report of the Summit on Education in Secure Software [5] for ideas about teaching robust, secure programming to a variety of audiences. Indeed, that report suggested many of the ideas elaborated here.

Students learning to write robust programs will *not* by itself cause the state of software and system security to improve dramatically. The problem is that software and systems depend on existing libraries and services. If these have vulnerabilities, so do their callers — this is the "supply chain" problem. For example, in 1999, the widely-used RSAREF2 library was found to have a buffer overflow vulnerability; as that standard cryptographic library is widely used in security-sensitive programs, those programs were at risk [6]. Further,

system security relies on systems being set up and configured *as required for the particular environment in which the system is used*. Writing robust software is not enough. But system and software security depends upon it.

V. Conclusion

Sir Richard Livingstone gave perhaps the best exposition of experience enhancing education:

> This truth was first brought home to me more than thirty years ago one December day, as I walked down the road from Argentières to Chamonix after a snowfall, and suddenly from the abyss of unconscious memory a line of Virgil rose into my mind and I found myself repeating
>
> *Sed iacet aggeribus niveis informis, et alto Terra gelu.*
>
> I had read the words at school and no doubt translated them glibly "the earth lies formless under snow-drifts and deep frost"; but suddenly, with the snow scene before my eyes, I perceived for the first time what Virgil meant by the epithet *informis*, "without form", and how perfectly it describes the work of snow, which literally does make the world formless, blurring the sharp outlines of roofs and eaves, of pines and rocks and mountain ridges,taking from them their definiteness of shape and form. Yet how many times before that day had I read the words without seeing what they really mean! It is not that the word *informis* meant nothing to me when I was an undergraduate; but it meant much less than its full meaning. Personal experience was necessary to real understanding. [2, p. 89]

Industries, foundations, governmental and other agencies, and practitioners have much to contribute to the next generation, who one day will need to provide society's tools, products, and services. Working with academic institutions, that contribution can be integrated with more theoretic work to provide a solid basis for improving today's practices, and advancing the state of the art of computer security.

In some ways, this process has begun. Large organizations (and, sometimes, smaller ones) have internships and co-ops. Several academic institutions have had success in working with organizations that saw benefit from class laboratories undertaking analyses; the voting machine exercise mentioned above is just one of many examples.

Many faculty see academic education as divorced from practice because the goal of academic education is *understanding*, not *implementing*. Yet achieving that understanding requires experience, as noted above. Those who feel that academic education is education in the "ivory tower" should understand that the nexus for working with non-academic organizations lies in the realm of giving students experience to cement their understanding, something any teacher will agree is critical.

Organizations must believe that working with academia and students will benefit them. The benefit varies with the organization's mission. Essentially, the organization must be willing to commit some resources in the hopes of getting a greater return. It is a gamble, with potentially high payoff.

For example, organizations understand the need for robust programming, but are unwilling to pay the additional cost that it requires in both development time and money. Academia also understands the need for it, but often sees it as an "implementation skill" not in high demand. Students do not see that experience in robust programming has any competitive advantage in the job market, and so are unconcerned about it.

There are no easy answers. Perhaps the best that can be said is that, unless institutions and organizations work together to solve the problem, the problem will render them obsolete and unresponsive to the needs of society, and of the students—the next generation of producers and consumers.

Acknowledgments

Thanks to Urko Zurutuza and Marco Prandini for helpful discussions. This work was supported by U.S. National Science Foundation awards CCF-0905530 and CNS-1039564. Any opinions expressed are those of the author.

References

[1] W32.duqu: The precursor to the next stuxnet. Technical report, Symantec Corporation, Mountain View, CA, USA, Oct. 2011.

[2] S. D. Alinsky. *Rules for Radicals*. Vintage Books, New York, NY, USA, 1989.

[3] M. Bishop and C. Elliott. Robust programming by example. In *Proceedings of the Seventh World Conference on Information Security Education*, pages 23–30, June 2011.

[4] M. Bishop and B. J. Orvis. A clinic to teach good programming practices. In *Proceedings of the Tenth Colloquium for Information Systems Security Education*, pages 168–174, June 2006.

[5] D. Burley and M. Bishop. Summit on education in secure software: Final report. Technical Report GW-CSPRI-2011-7, Cyber Security Policy and Research Institute, The George Washington University, Washington, DC, June 2011.

[6] CERT. Buffer overflows in SSH daemon and RSAREF2 library. certadv CA-1999-15, CERT, Pittsburgh, PA, USA, Dec. 1999.

[7] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley Professional, Boston, MA, USA, second edition, Mar. 2003.

[8] N. Falliere, L. O Murchu, and E. Chien. W32.stuxnet dossier version 1.3. Technical report, Symantec Corporation, Mountain View, CA, USA, Nov. 2010.

[9] R. R. Linde. Operating system penetration. In *Proceedings of the 1975 National Computer Conference*, AFIPS '75, pages 361–268, New York, NY, USA, May 1975. ACM.

[10] K. Nance, B. Hay, and M. Bishop. Secure coding education: Are we making progress? In *Proceedings of the 16th Colloquium for Information Systems Security Education*, June 2012.

[11] C. M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick. *Version Control with Subversion*. O'Reilly and Associates, Sebastopol, CA, USA, second edition, Sept. 2008.

[12] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sep. 1975.

[13] sKyWIper Analysis Team. sKyWIper (a.k.a. flame a.k.a. flamer): A complex malware for targeted attacks. Technical Report v1.05 (May 31, 2012), Laboratory of Cryptography and System Security (CrySyS Lab), Budapest University of Technology and Economics, Budapest, Hungary, May 2012.

[14] J. Vesperman. *Essential CVS*. O'Reilly and Associates, Sebastopol, CA, USA, second edition, Nov. 2006.

[15] C. Weissman. Security penetration testing guideline: A chapter of the handbook for the computer security certification of trusted systems. Technical Memorandum 5540:082A, Naval Research Laboratory, Washington, DC, USA, Jan. 1995.