Design of a Process for Software Security

David Byers Nahid Shahmehri

Department of computer and information science
Linköpings universitet, SE-58183 Linköping, Sweden
E-mail: {davby,nahsh}@ida.liu.se

Abstract

Security is often an afterthought when developing software, and is often bolted on late in development or even during deployment or maintenance, through activities such as penetration testing, add-on security software and penetrate-and-patch maintenance. We believe that security needs to be built in to the software from the beginning, and that security activities need to take place throughout the software lifecycle. Accomplishing this effectively and efficiently requires structured approach combining a detailed understanding on what causes vulnerabilities, and how to prevent them.

In this paper we present a process for software security that is based on vulnerability cause graphs, a formalism we have developed for modeling the causes of software vulnerabilities. The purpose of the software security process is to evolve the software development process so that vulnerabilities are prevented. The process we present differs from most current approaches to software security in its high degree of adaptability and in its ability to evolve in step with changing threats and risks. This paper focuses on how to apply the process and the criteria that have influenced the process design.

1. Introduction

Vulnerabilities – security-related flaws – in software affect us almost daily, have forced us to change how we use computers, and are at the center of some of the most spectacular and costly computer failures in recent years. For example, the total cost of the Code Red worm has been estimated at $2.6 billion, and the Nachi worm affected operations at Air Canada and CSX railroad. Both exploited buffer overflows, a class of vulnerabilities that has been known since at least 1988. Efforts are being made to reduce vulnerabilities in software, but the industry clearly has a long way to go.

There are a number of tools and techniques for building secure software [8, 12, 13, 15, 16], but they are typically applied in an ad-hoc fashion; their benefits and limitations are rarely fully understood; they do not take risk assessments into account; and there are no methods for keeping up with changing threats and risks. These issues limit the applicability of current methods.

We are developing a process for systematic and continuous improvement of software security throughout the software lifecycle, that is suitable for industrial adoption, and focuses on preventing vulnerabilities in all phases of software development. In contrast to nearly all existing approaches, our process can be used in conjunction with any software development process, adapts to the needs of the business (rather than the other way around), and takes risk into account.

Our process [2] consists of three major steps: vulnerability modeling [5], vulnerability cause mitigation [1], and process component definition. The process is described in section 2.

Through collaboration with companies that want to improve software security in their products, we can empirically validate our work and collect criteria related to applicability and adoption issues. These issues are discussed in sections 3 and 4, and are the focus of this paper.

2. Process Overview

Our security process is a software process improvement (SPI) process: it runs parallel to the software lifecycle, throughout the entire lifecycle, and its output is used to improve the software development process. It consists of three main steps: vulnerability modeling, vulnerability cause mitigation, and process component definition. Although a single iteration of the process is predominately linear, steps may be revisited as required to refine the result.

2.1. Process Steps

Vulnerability modeling [5] is a process similar to root cause analysis, and results in a structure called a vulnerability cause graph (VCG), that models how causes (conditions or events that may contribute to the presence of a vulnerability; e.g. "Use of strcat") may contribute to the presence of software vulnerabilities. Non-leaf nodes in the VCG represent causes, and the single root node, labeled with the vulnerability that the VCG models, is used to calculate the model semantics. Figure 1 shows the VCG for a buffer overflow vulnerability in MySQL [18].

The intuition behind the VCG is that in order to prevent a cause *C*, then either *C* itself must be directly mitigated through activities in the software lifecycle, or all the predecessors of *C* must be prevented. Preventing a vulnerability becomes a matter of preventing a subset of its causes.

Vulnerability modeling is performed by first conducting a deep analysis of the vulnerability being modeled. The vulnerability must be understood completely before its causes are determined. This analysis is typically sufficient to determine the direct causes (which are usually found in the source code) of the vulnerability. After determining direct causes, modeling progresses to indirect causes; the modeling process calls for attempting to identify causes in all phases of software the software lifecycle. This process typically requires consulting external data sources (e.g. requirements and design documents, version control history, process documentation etc) in order to identify causes.

The final step of vulnerability modeling is a validation phase, in which the model is validated by an independent analyst. We have found that this step significantly increases the quality and consistency of models.

Vulnerability cause mitigation [1] is the process of determining how to prevent individual causes that are present in some VCG. This is modeled using structure called a security activity graph (SAG), that shows how activities in the software lifecycle combine to prevent a cause or vulnerability. Leaf nodes in the SAG represent activities; these are connected using logic (*and* and *or*) gates to form the complete graph. Figure 2 shows the SAG for a single cause

Activities are fully documented, concrete, activities in the software lifecycle. They include complete information on how to implement them, and how to verify that they are successful. The process of vulnerability cause mitigation calls for identifying activities in all phases of the software lifecycle. Causes related to e.g. source code need not be prevented by implementation-related activities; it is sometimes more efficient to prevent them earlier in the development process.

The SAG for a cause shows how to prevent that particular cause. The SAG for a vulnerability, which shows how to prevent that vulnerability, is computed automatically by combining the SAGs for each cause in the vulnerability's VCG, according to the structure of the VCG. The SAG for a typical vulnerability can be very large and complex, but has a structure that lends itself to automatic processing.

Process Component Definition is the process of selecting activities to implement from a set of security activity graphs, in order to prevent software vulnerabilities. Activities should be chosen that are suitable to the product, development process, and organization. To accomplish this, activities are assigned weights, and best weight valid set of activities is selected.

The weight of an activity depends on a number of factors, such as its fit to the development process, product, staff, and organization; direct costs associated with performing the activity; and cost of required tools or training. For example, an activity that staff already knows how to perform will be cheaper (from a training point of view) than one that requires extensive staff training. There may also be dynamic effects, such as synergism or conflicts between activities, that affect the overall weight of a set of activities. For example, there is synergy between activities that use the same tool. Weight estimation needs to be made for each individual project, and must be maintained as controlling factors change.

We are currently in the process of specifying how to determine and assign weights to activities. Given a SAG (or set of SAGs) and activity weights, the least expensive set of activities that prevent the vulnerability is selected. This set is then transformed, manually at the moment, to whatever documentation or other artifacts the organization requires for changing their processes.

2.2. The vulnerability analysis database

All information that is collected or generated during the process is stored in a shared repository, known as a vulnerability analysis database (VAD). Using this database during analysis and modeling promotes re-use of existing results, which in turn speeds up the process, and promotes consistency in modeling. Re-use is central to the process as it satisfies some of our more important design criteria (see section 4.

Ideally the VAD would be a shared resource, available to anyone using the security process. Input from our partners has clearly shown that this is not a tenable position, as some of the information in the VAD cannot be shared with others. For example, some information may relate to confidential methods or tools, or may be restricted due to contractual or legal obligations. We are considering architectures for the VAD that allow sharing of partial information.

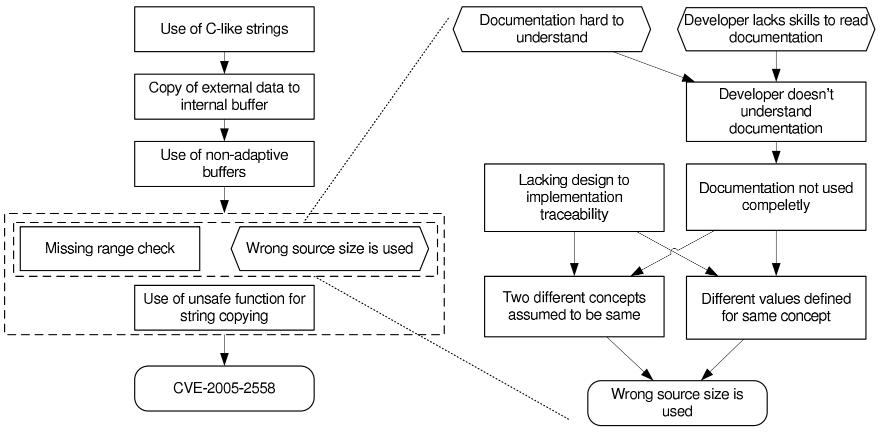

Use of C-like strings
Copy of external data to internal buffer
Use of non-adaptive buffers
Missing range check
Wrong source size is used
Use of unsafe function for string copying
CVE-2005-2558
Documentation hard to understand
Developer lacks skills to read documentation
Developer doesn't understand documentation
Lacking design to implementation traceability
Documentation not used compeletly
Two different concepts assumed to be same
Different values defined for same concept
Wrong source size is used

Figure 1. Vulnerability cause graph for CVE-2005-2558, with a compound node expanded

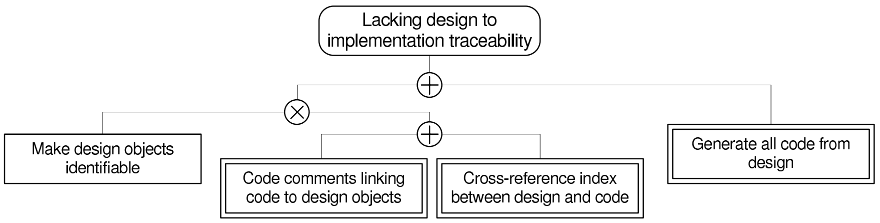

Lacking design to implementation traceability
Make design objects identifiable
Code comments linking code to design objects
Cross-reference index between design and code
Generate all code from design

Figure 2. The security activity graph for a cause of CVE-2005-2558

The VAD contains five major sets of data:

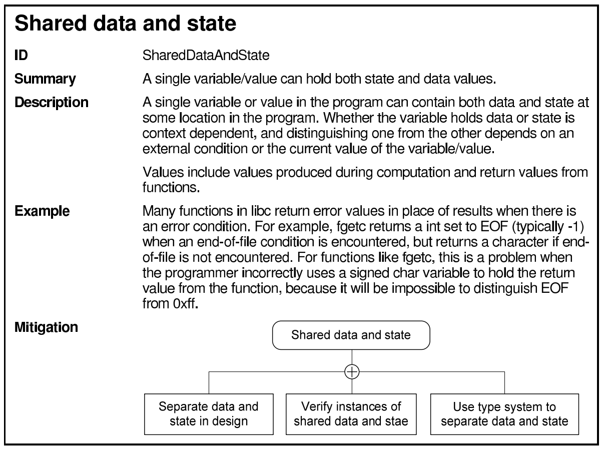
Vulnerabilities Every vulnerability that has been analyzed is entered into the VAD. This includes ID, title, summary and description of the vulnerability, together with an in-depth analysis, optional references or excerpts of code, references to external sources (e.g. CERT advisories or NVD entries), references to exploit code and a reference to the VCG for the vulnerability.

Vulnerability cause graphs Every vulnerability cause graph that has been created is stored in the VAD. Graphs can be visualized, and are linked to the vulnerabilities they represent, and to the causes they contain.

Causes Every cause present in some VCG has an ID, title, summary, in-depth description, code examples, references to relevant vulnerabilities, and a SAG. Figure 3 shows a typical cause in our current database.

Security activity graphs Every SAG that has been created is stored in the VAD. Graphs can be visualized, and are linked to the causes they represent, and to the activities they contain.

Activities Every activity present in some SAG has at least an ID, a title, an implementation procedure, a verification procedure, a set of constraints, and estimated weight. This section of the VAD is the collected knowledge of security-related activities in the software lifecycle. Figure 4 a typical activity in our current database.



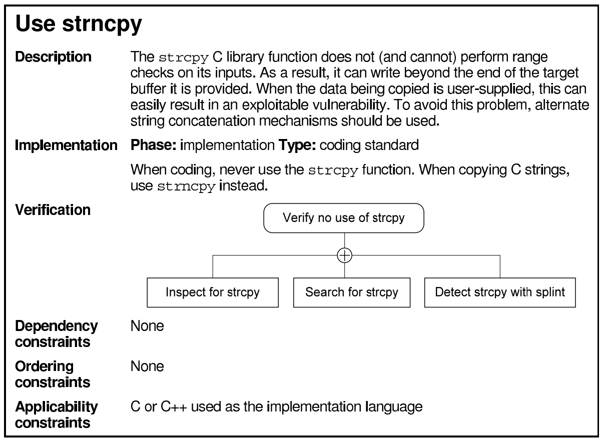
Shared data and state

ID	SharedDataAndState
Summary	A single variable/value can hold both state and data values.
Description	A single variable or value in the program can contain both data and state at some location in the program. Whether the variable holds data or state is context dependent, and distinguishing one from the other depends on an external condition or the current value of the variable/value. Values include values produced during computation and return values from functions.
Example	Many functions in libc return error values in place of results when there is an error condition. For example, fgetc returns a int set to EOF (typically -1) when an end-of-file condition is encountered, but returns a character if end-of-file is not encountered. For functions like fgetc, this is a problem when the programmer incorrectly uses a signed char variable to hold the return value from the function, because it will be impossible to distinguish EOF from 0xff.
Mitigation	

Shared data and state
Separate data and state in design
Verify instances of shared data and stae
Use type system to separate data and state

Figure 3. A typical cause

We have developed a very simple prototype VAD, accessible through a web browser, sufficient for the current needs of our research. We are in the process of developing a full-fledged VAD, which will be used in field trials of the process.



Use strncpy

Description	The `strcpy` C library function does not (and cannot) perform range checks on its inputs. As a result, it can write beyond the end of the target buffer it is provided. When the data being copied is user-supplied, this can easily result in an exploitable vulnerability. To avoid this problem, alternate string concatenation mechanisms should be used.
Implementation	**Phase:** implementation **Type:** coding standard When coding, never use the `strcpy` function. When copying C strings, use `strncpy` instead.
Verification	
Dependency constraints	None
Ordering constraints	None
Applicability constraints	C or C++ used as the implementation language

Figure 4. A typical activity

2.3. Tool Support

Although our process can be applied without the use of any tools, tool support is expected to be required for *effective* application of the process. We are currently in the process of developing these tools.

Vulnerability modeling will require visualization and model editing tools, as well as tools to search the VAD for re-usable model elements. To support the analysis process that precedes modeling, integration between the modeling tool and e.g. source code control systems, defect management systems, documentation, and so on would be useful.

Vulnerability cause mitigation will require visualization model editing tools, as well as tools to search the VAD for re-usable model elements. Additionally, integration with e.g. internal process documentation would be useful.

Process component definition will require tools to estimate weights of activities (and maintain such estimates), visualize VCGs and SAGs, make activity selections, visualize activity selections, and generate process documentation. Integration with systems used to document software development processes would also be useful.

3. Practical Application

Application of our process in a project should be fairly straightforward since it has been designed to cause minimal disruption to the business and to the development process on introduction (see section 4). It can be introduced incrementally, or on a trial basis, to further reduce its initial impact on the business.

For this process, much like other processes, to be implemented successfully, it is necessary to have management support and buy-in from those affected by the process; without this, the likelihood of success is small [9].

3.1. Staffing

The process requires two individuals who are trained in vulnerability modeling and cause mitigation analysis (two individuals are required to perform model validation). There must also be an individual or team with the mandate to alter the organization's development processes; this individual or team is a good candidate for ownership of the overall process.

It is possible for a single individual to implement our process without model validation. This may be appropriate during a trial of the process.

Our empirical work has shown that vulnerability modeling requires a mindset that is initially foreign to most developers: they tend to think in terms of fixing problems, rather than in terms of their causes. This matches our own experience from developing the process. We have also found that the more experience an individual has, the more varied causes they identify.

Our recommendation is that senior developers with experience from all phases of software development are trained in vulnerability modeling and cause mitigation analysis.

3.2. Input to the Process

The input to our process is essentially problem reports: reports of vulnerabilities or potential vulnerabilities in the software. Problems need to be reported in all phases of the software lifecycle, including problems uncovered (and possibly fixed) during development. If problem reports are limited to problems uncovered in test or after deployment, there is a significant risk that flaws similar to those discovered during development still exist in the software.

We do not mandate a particular process for collecting problem reports, or their contents. A simple problem description is sufficient for our process, as the first stage is an in-depth analysis of the problem.

3.3. When is the Process Executed

Vulnerabilities and potential vulnerabilities can be discovered at any time, so the process we are developing must exist throughout the software lifecycle. The process is iterative in nature: at any time in the software lifecycle when conditions mandate it, an iteration of the process is initiated, possibly leading to changes to the software development process. Figure 5 illustrates the

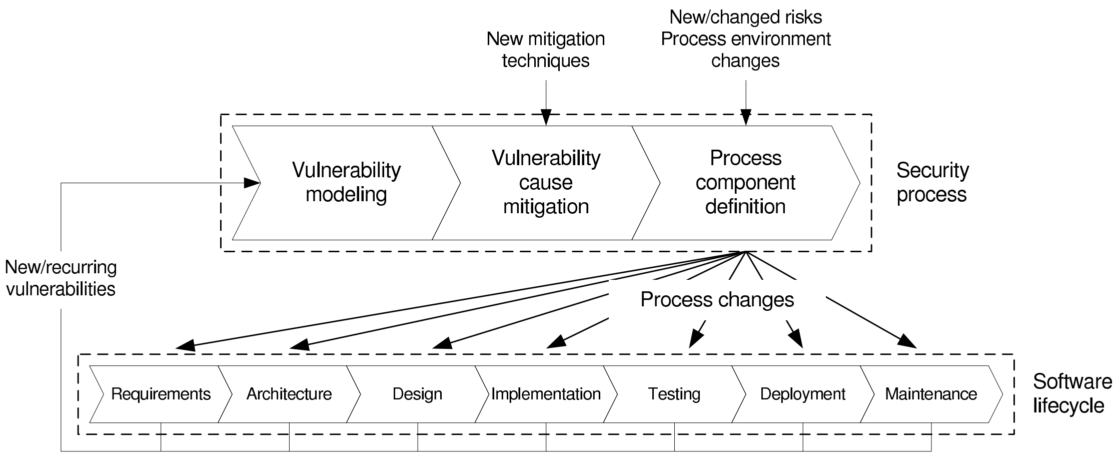

New mitigation techniques
New/changed risks Process environment changes
Vulnerability modeling
Vulnerability cause mitigation
Process component definition
Security process
New/recurring vulnerabilities
Process changes
Requirements
Architecture
Design
Implementation
Testing
Deployment
Maintenance
Software lifecycle

Figure 5. Security process in the context of the software lifecycle

security process in the context of the software lifecycle.

Conditions mandate initiating a new iteration of the process any time a something that was considered in previous iterations of the process changes. For example:

- When a new vulnerability (or potential vulnerability) is discovered, a new iteration of the process is initiated in order to prevent the vulnerability (or recurrences thereof), possibly resulting in changes to existing vulnerability or activity models.

- When a known vulnerability recurs, this indicates that the existing vulnerability or activity models are incomplete. The existing models are augmented to reflect the cause of the recurrence, so further recurrences are prevented, possibly resulting in changes to other vulnerability or activity models.

- If new mitigation techniques become known, all existing models are revised to include the new mitigation technique, where appropriate. If an existing mitigation technique changes, all models containing the mitigation technique are reviewed, to ensure that the continued use of the mitigation technique is appropriate. Since the set of mitigation techniques (and their properties) affect process component definition, that step is always revisited, to see if the selection of activities should change.

- When a new risk is identified or a known risk changes, new vulnerability models may be created to model potential vulnerabilities implied by the risk. Since risk influences process component definition (for example, a user of the process may elect not to prevent vulnerabilities with low risk, but reconsider when that risk increases), that step is revisited regardless of whether new vulnerabilities are modeled.

- If criteria that influence process component definition change (e.g. development process, staffing, tools, or some other basis for weight estimation), process component definition is performed again, as the selection of activities may change.

Events relating to new or recurring vulnerabilities, and new or changed risks are the most critical, and should result in immediate action. New mitigation techniques or issues related to weight are not critical from a security perspective. Immediate action is typically not required (unless mandated by business needs), but the affected models should eventually be revisited.

4. Process Design Criteria

Since we are designing this process for commercial adoption, it is important to examine the factors that influence adoption decisions, and attempt to address them in the process design. In this work, we rely a great deal on the needs of our three commercial partners. They share the view that software security is a key issue for their businesses, but otherwise represent a wide spectrum of potential adopters in terms of size (SME through multinational), primary customer groups (civilian through military), business model (product-oriented, customer-oriented, service-oriented) and development processes (from requirements-driven to agile).

In the terminology of Keen [11], a process must be salient (a key process for the business), and an asset (returns more money than it costs) to the business that considers it for adoption. Our partners in this project see a software security process as a priority process; it should be the same for others that depend on delivering secure software to their customers. For these businesses, the salience is clear. Our process is does not directly generate value, unless sold as a product (which at least one of our partners may consider). It is both an option-enabling (by e.g. speeding up response to new threats) and value-preserving process (it is intrinsic to the value generating process of developing software); hence it is clear that our process is an asset process. Determining the value of the process a particular business is beyond the scope of this discussion.

In addition to these aspects, process adoption is affected by a number of factors related to the people. In particular, process adoption is more likely to be successful when the people involved are highly motivated, than when they are not [9, 7, 14].

Baddoo and Hall have studied the factors that motivate [3] and de-motivate [4] practitioners' support for software process improvement (SPI). Their results on de-motivators are summarized in table 1. Others have also identified several of these issues [9].

De-motivators	Cited (%)		
	Devel-opers	Project mgrs	Senior mgrs
Commercial pressures	19	25	25
Cumbersome processes	24	13	8
Inadequate communication	5	13	17
Inertia	43	25	50
Lack of overall support	5	13	50
Negative/bad experience	5	13	33
Time pressure/constraints	62	44	58
Lack of resources	0	31	67
Lack of evidence	0	38	8
Budget constraints	24	0	25
Inexperienced staff	5	0	25
Personality clashes	10	0	8
Imposition	14	6	0

Table 1. Common de-motivators across practitioner groups (from Baddoo and Hall, 2003 [4])

Of these, we attempt to address cumbersome processes, inertia, time pressure, lack of resources, lack of evidence, and imposition. Commercial pressures are not an issue for us: our commercial partners all cite commercial pressures as their main reason for engaging in this project. Communication, support, and personality issues cannot be addressed by this process.

Baddoo and Hall identified a large set of motivators, with significant differences in kind between different groups. We feel that motivators for project and senior management are particularly important, as fulfillment of these can address a number of de-motivators. Project managers identify their top motivators as visible success, resources, process ownership, empowerment, easy processes, maintainable processes, knowledgeable team leaders, reward schemes. Senior management identifies visible success, meeting targets, cost beneficial, process ownership, resources, and reward schemes as their top motivators. Developers identify visible success, bottom-up initiatives, resources, and top-down commitment as their top motivators.

Through our project partners we have also identified a number of factors that can be expected to affect the adoption of our process (most apply to any SPI process), the most important of which are cost efficient, evidence of benefits, minimal impact on existing processes, minimal investment, and minimal cost of maintenance.

Based on this, we have identified a number of design criteria for the process.

4.1. Leverage Existing Security Know-How

There is already a considerable body of proven security know-how, and individual organizations have developed know-how related to their specific products, processes and other circumstances. Since this know-how represents a considerable investment, businesses would naturally be reluctant to adopt a process that reduces the value of this investment. Furthermore, from a security point of view, it makes more sense to leverage existing effective methods, tools, and techniques, than to ignore them.

We integrate existing know-how in our process as activities in the security activity graphs. Indeed, existing know-how is critical to the success of vulnerability cause mitigation.

This criterion addresses the inertia issue, by allowing existing practices to continue (if they provide value to the business by contributing to security); lack of evidence (to some extent), by utilizing proven security activities; inexperienced staff, since staff should already have experience with existing practices; and cost/resource issues, by exploiting existing investments.

4.2. Decreasing Cost, Increasing Benefit

Since cost and resource issues are clearly defined as important issues (both as de-motivators and motivators, and among our partners), a new process should have greater potential for success if it is inexpensive than if it is expensive. A process that decreases in cost and increases in benefit over time, should have an even greater chance of long-term viability, as this not only increases the asset value of the process over time, but also minimizes financial risk by avoiding hidden costs in the future.

In our process, we address this issue through re-use of analysis and modeling results. When new models are developed, parts of existing models (for which vulnerability cause mitigation is already completed) can often be reused, thereby significantly reducing the cost of modeling and mitigating the new vulnerability. We have found that even after modeling only a few vulnerabilities, significant re-use occurs.

This criterion addresses cost and resource issues by reducing resource requirements over time; cumbersome/easy process issues, since the process becomes more streamlined over time; and cost/benefit issues by providing increased benefits and reduced costs over time.

4.3. Development Process Agnostic

Process change of any kind is known to be difficult and costly. The more intrusive a new process is, by requiring changes to other processes, the greater the obstacles are to its introduction, since changes to other processes are costly, go against inertia, may be seen as intrusive, and so forth. Because of this, a new process should stand a greater chance of success if it adapts to the business in which it is to be used, rather than require the business to adapt to the process.

We have chosen to design our security process in such a way that it is entirely independent of the software development process. Instead of intruding in the development process, it is an add-on that directly impacts only a small number of people. Our impact on the development process itself is limited to changes required to prevent vulnerabilities, and the process ensures that those changes are as tailored to the business' needs as possible.

This criterion addresses the inertia issue, by allowing existing processes to continue; cost and resource issues, by requiring only minimal re-investments in other processes; the inexperienced staff issue, since staff can continue with the practices they are experienced in; and the imposition issue, by minimizing the limiting changes to the development process to those that are required to prevent vulnerabilities, and allowing these changes to be adapted to the people directly affected.

4.4. Straightforward Application

Cumbersome processes (which also tend to be expensive and resource intensive), and processes that must be imposed by management are known to be difficult to introduce. A process that is perceived as easy stands a greater chance of adoption than a cumbersome one. Furthermore, a process that can be introduced gradually in a bottom-up way faces fewer obstacles than one that must be imposed top-down in the organization.

We have designed our process to be as simple as possible, while remaining effective. Our empirical evidence indicates that vulnerability modeling requires less than an hour of training to perform; we expect that vulnerability cause mitigation will require even less, as it is closer to how developers think naturally. Understanding VCGs and SAGs takes hardly any training at all. As a result, it is possible to trial our process with minimal investment. It can even be done by a single developer, to guide personal development practices.

However, despite not being cumbersome, vulnerability modeling and vulnerability cause mitigation are time-consuming activities when applied to real vulnerabilities. We doubt that this will be a serious obstacle to adoption, as every part of the process contributes to the ultimate goal of preventing vulnerabilities.

This criterion addresses the cumbersome/easy process issue, and by implication some of the cost and resource issues. It also supports bottom-up initiatives.

4.5. Process Validation

All our sources identify visible success and clear evidence of benefits as top priorities for successful adoption. Although processes can occasionally be introduced without first proving their worth, the evidence is clear that this is not an easy path. It is better to show the worth of the process before attempting adoption.

Our process design can only address this to a small extent. Similar processes, such as root cause analysis, have proven successful both in software and in other technology areas.

In order to address this issue, we plan for validation of our methods in a commercial setting. We have begin validating key properties of each step of the process, and will later deploy the entire process at one of more of our partners, hoping to show that it leads to a reduction in vulnerabilities (or recurrences thereof). Our results to date indicate that the vulnerability modeling phase, which is the basis for the entire process, works as intended.

4.6. Sustainable Security

Security is not something that can be achieved and then forgotten. It is generally accepted in the security field that in order to sustain a given level of security, it is necessary to continuously react to new threats and vulnerabilities. This criterion is related to the salience of the process; a security process that does not adapt to new threats and vulnerabilities is not viable in the long term, and hence not salient to the business.

Our process is designed to exist through the entire software lifecycle. There is no point at which it is "complete"; it is always present in case new threats or vulnerabilities are discovered. It is also designed to accept advances in mitigation techniques, and to adapt to changing conditions within the business. Thus, by design, our process supports the idea of sustainable security.

5. Related Work

Currently, most approaches to software security are based on experience and best practices [8, 12, 13, 15, 16]. There is no doubt that these methods are valuable, as evidence clearly shows that they do prevent vulnerabilities. Nevertheless, they typically have several drawbacks.

One of the most important drawbacks of current methods is that they are inflexible and provide little or no support for evolution. For example, CLASP [16], while comprehensive and practical, can only be used with the Rational Unified Process [10], and evolution is accomplished by waiting for the next revision of the manual. In contrast, our approach supports any process, adapts to each user's needs and evolves to meet new threats through the continuous revision of vulnerability models and their SAGs. Flexibility and evolution are two of the primary goals of our work.

Another common drawback is lack of specificity. Many proposals for secure software development offer high-level recommendations, but little in the way of practical guidance. They achieve flexibility by sacrificing specificity. For example, Howard [8] recommends that a central security team is created, but does not discuss alternatives, concrete benefits of having such a team, or consequences of not having one. Many other proposals for secure software development have the same deficiencies. In contrast, our approach always shows alternatives, benefits and consequences (through SAGs and through the relationship between activities and causes), and offers a very flexible framework for selecting concrete activities. The high-level recommendations found in current literature may appear as activities or consequences of activities in our method.

Essentially all current methods tend to be centered on best practices in one way or another. Our approach is not different in that respect: many security activities in our approach are best practices, and a thorough knowledge of best practices is essential to successful cause mitigation analysis. Our approach does go beyond best practices by clarifying the security consequences of issues that may not normally be security-related (e.g. management or personnel issues). Our approach also reduces some of the obstacles to implementation of best practices [17], by clarifying the benefits of each practice.

Our overall approach to software security is related to root cause analysis (RCA) and defect causal analysis (DCA) [6]. There are significant differences: RCA and DCA are typically most concerned with root causes, while we are equally concerned with contributing causes (in RCA terminology). DCA uses fishbone diagrams to organize causes; we have found them too limiting. We provide more guidance on actions (the work presented in this paper) than do either RCA or DCA.

6. Conclusions

In this paper we have presented an overview of a process designed to help software development organizations prevent vulnerabilities in the software they develop. The process is flexible, straightforward and well adapted to the needs of most businesses. We have described key components (vulnerability modeling and cause mitigation analysis) in earlier work; this paper focuses on how to apply the process and the criteria that have influenced the process design.

The process is applied throughout the entire software lifecycle as an adjunct to the software development process. It can be used together with any software development process – from requirements-driven to agile. As new risks and vulnerabilities are identified, or the development organization changes, the software security process is used to adjust and adapt the development process to these new conditions.

The design criteria we have identified are all aimed at improving the applicability of the process and maximizing the likelihood of successful adoption. The criteria include those focused on the business (costs, benefits, and salience) as well as those focused on factors relating to successful adoption (e.g. integration of existing practices and impact on developers). We have also discussed how our process satisfies each criterion.

One of the key properties of our process is that it supports the idea of sustainable security. It is designed not only to meet the challenges we know of today, but to be adaptable to future challenges as well. It is designed to help its users continuously meet new threats and vulnerabilities as they evolve.

Another key property of our process is its flexibility. Since we make no assumptions about the development process, our software security process can be used in conjunction with any development process. Nor does our process make assumptions about how specific problems are to be mitigated; it provides a framework in which users of the process can determine which combinations of activities can potentially prevent vulnerabilities, and select the set of activities that best suits their situation. In this way existing security practices can be integrated into our process, which clarifies the exact benefits of each practice, with respect to the vulnerabilities the user of the process is concerned with.

Currently, we have completely defined two of the steps of the software security process, with the third well under way. We have empirically validated key aspects of the first step, and are in the process of validating the second. We plan on performing a full-scale field test of the process, together with our commercial partners, in the near future.

References

[1] S. Ardi, D. Byers, C. Duma, and N. Shahmehri. A cause-based approach to preventing software vulnerabilities. (submitted).

[2] S. Ardi, D. Byers, and N. Shahmehri. Towards a structured unified process for software security. In *Proceedings of the ICSE 2006 Workshop on Software Engineering for Secure Systems (SESS06)*, 2006.

[3] N. Baddoo and T. Hall. Motivators of software process improvement: an analysis of practitioners' views. *The Journal of Systems and Software*, 2002(62):85–96, 2002.

[4] N. Baddoo and T. Hall. De-motivators for software process improvement: an analysis of practitioners' views. *The Journal of Systems and Software*, 2003(66):23–36, 2003.

[5] D. Byers, S. Ardi, N. Shahmehri, and C. Duma. Modeling software vulnerabilities with vulnerability cause graphs. In *Proceedings of the International Conference on Software Maintenance (ICSM06)*, 2006.

[6] D. N. Card. Learning from our mistakes with defect causal analysis. *IEEE Software*, 15(1), 1998.

[7] J. D. Herbsleb and D. R. Goldenson. A systematic survey of CMM experience and results. In *Proceedings of the 18th International Conference on Software Engineering*, pages 323–330, Berlin, Germany, March 1996. IEEE.

[8] M. Howard. Building more secure software with improved development processes. *Security & Privacy Magazine*, 2(6):63–65, Nov-Dec 2004.

[9] W. S. Humphrey. Why don't they practice what we preach. *Annals of Software Engineering*, 1998(6):201–222, 1998.

[10] I. Jacobson, G. Booch, and J. Rumbaugh. *Unified Software Development Process*. Addison-Wesley, 1999.

[11] P. G. W. Keen. *The Process Edge: Creating Value Where it Counts*. Harvard Business School Press, 1997.

[12] S. B. Lipner. The trustworthy computing security development lifecycle. In *Proceedings of the 20th Annual Computer Security Applications Conference*, pages 2–13. IEEE Computer Society, December 2004.

[13] G. McGraw. Software security. *Security & Privacy Magazine*, 2(2):80–83, Mar-Apr 2004.

[14] M. Niazi, D. Wilson, and D. Zowghi. A model for the implementation of software process improvement: A pilot study. In *Proceedings of the Third International Conference on Quality Software (QSIC'03)*, pages 196–203. IEEE, 2003.

[15] S. T. Redwine and N. Davis. *Processes to Produce Secure Software*, appendix B. Task Force on Security Across the Software Development Lifecycle, 2004.

[16] Secure Software, Inc. The CLASP application security process. `http://www.securesoftware.com/` (accessed April 2006).

[17] R. Turner. Seven pitfalls to avoid on the hunt for best practices. *IEEE Software*, 20(1), 2003.

[18] US-CERT/NIST. Vulnerability summary CVE-2005-2558. National Vulnerability Database. `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2005-2558`.