# UMLpac: An Approach for Integrating Security into UML Class Design

Matthew J. Peterson
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24060
mjpeters@vt.edu

John B. Bowles
University of South Carolina
Columbia, SC 29208
bowles@engr.sc.edu

Caroline M. Eastman
University of South Carolina
Columbia, SC 29208
eastman@engr.sc.edu

## Abstract

*One of the biggest goals in software engineering is to create secure software. This process must begin in the design phase of the software development life cycle. While the Unified Modeling Language (UML) exists to aid engineers in designing software systems, it lacks features to integrate security aspects into that design. This paper presents an extension of UML, UMLpac, which bridges the gap between software class design and the security techniques required for that design. Security packages accomplish this goal by keeping a level of abstraction between the system class diagram and its security features. This design technique preserves the original system diagram, while maintaining in depth security features for all aspects of the system.*

## 1. Introduction

Software security is like a picket fence; it is only as good as its weakest post. With advances in network and physical access security, software has become the weakest post. A major reason for this is a lack of consideration for designing secure software systems from a developer's perspective. In the past software security was added only after the system was complete. Security vulnerabilities were discovered (normally by an attacker) and were then patched by system developers. This mindset, known as the penetrate-and-patch approach, was ineffective as evidenced by numerous security failures in industry.

The solution to ensure secure software is to include security in every aspect of the software development life cycle, most importantly, in the design phase. Many companies, such as Microsoft, use security teams that work with developers and managers in the design of systems to help improve software security [10]. This has created an entirely new set of problems with how software design should be represented. Consider the analogy of a software system to a building design. In the past, software was a simple room layout created by developers to meet basic functional needs. With the addition of security requirements, that room layout has turned into a complete building blue print, equipped with electricity, water, and air conditioning. This increase in complexity causes a major loss of coherence of the system design. The focus of this paper is to demonstrate how security decisions can be represented in the design phase of the software development life cycle thereby simplifying the entire project design for developers, managers, and security analysts.

Presently, the Unified Modeling Language (UML) is the industry standard for designing software systems but it contains minimal capabilities for representing the security aspects of a system. This paper presents an extension of UML, UMLpac, which enables security features to be easily integrated into UML class diagrams. This extension makes it possible for security teams to layout security features directly onto the UML class diagram of a system while keeping a level of abstraction between the two. This approach offers many advantages, a few of which are summarized below:

1. Separation between the system class diagram and the system security features through use of security packages creates a level of abstraction between the class diagram and the security features making it easier to focus on just the class diagram, just the security features, or the entire system (both).
2. Security threats are defined through catalog entries and the corresponding prevention techniques are laid out in the design. This helps improve clarity and avoids overlooking possible threats.
3. Inclusion of a risk factor on security packages makes it possible to see a basic overview of system threat areas when looking at the class diagram.
4. Providing features for showing coding rules in UML design gives security specialists the ability to place secure coding practice reminders in a design for developers. (This is done through the use of rule security descriptors.)
5. Compartmentalization of security features in system design is improved by breaking down the security requirements into a collection of categories.

### 1.1. Related Work

We are not, at present, aware of any other ways of integrating of security requirements into UML system class diagrams that keep a level of abstraction between the class diagram and the security requirements. However, several other techniques for incorporating security requirements

into UML are easily accommodated by UMLpac. For example, SecureUML [6] which focuses on representing authorization and access control in UML can be integrated into UMLpac through the use of a security package having a principle security descriptor for SecureUML. The principles of SecureUML can then be applied to define the access control for the system in the security tile. Another example is UMLsec [5] which presents a way to implement secure-systems in UML. This can be used very effectively with UMLpac to lay out security features in security tiles.

## 2. Methodology

In constructing the UML extension UMLpac, 5 basic steps were followed. First, was the selection of an appropriate modeling language to extend. UML was chosen since it is presently the industry standard for modeling software systems and because it is not programming language specific. Second, was the consideration of how to keep a level of abstraction between the system code and the security features. Multiple concept ideas were constructed and analyzed. The best of these was the security package/tile approach because it seemed to work well when dealing with an object-oriented system. Third, was the categorization of security descriptors. Categories were constructed based off previous security works [1][3]. Fourth, combinations and sets were added to the extension to improve read and write ability. Finally, UMLpac was tested on basic systems for functionality.

## 3. Security Packages

To represent security features in a class diagram, UMLpac uses a stereotype construct called a *security package*. A security package represents each security aspect of a system. Each security package has three stereotype attributes:

- Risk Factor
- Security Tile
- Security Descriptor.

### 3.1 Risk Factor

This attribute estimates the probability of an attack on the given security package. The primitive type of the value used is left to the designer. Some examples are using an exact integer (e.g., 1-10), a string range (e.g., not likely – very probable) or a numeric probability value (e.g., 0.8). The goal of this attribute is to make the high and low risk areas of a system easily visible in the design.

### 3.2 Security Tile

A security tile defines the security descriptor (described next) of a security package. This is accomplished through a combination of: a UML (or UML extension) class diagram of the internal security system, a catalog entry for the security feature, UML notes, or more security packages as in the case of combinations and sets (discussed in Section 3). A catalog entry is a definition of the security aspect being considered. Barnum and McGraw provide some guidelines for creating functionally complete catalog entries in [1]. The security tile is where a security analyst will specify how a given security package will protect parts of a system. The goal of this feature is to separate the specific security details of each security package from the system UML class diagram.

It is recommended that guidelines be considered when creating security tiles. First, it is important to limit information on a security tile so it only represents the implementation of its corresponding security package. Limiting information can become difficult when dealing with a system that is tightly coupled since security tiles might become intertwined with one another. Second, be sure to keep order and layout of information on security tiles consistent throughout your system. Finally, be sure you are absolutely complete in covering all possible risks relating to the security descriptor. For example, if you descriptor is environment variables, be sure to consider every environment variable, not just a few specific selections.

By following these guidelines and using security tiles, people working with a system diagram can choose to see basic security information by looking at the security package and gain specific details of how the security is implemented by looking at the security tiles. By separating the security details from the class diagram, the diagram does not lose its coherence by becoming overwhelmed with security information. (Figure 3 at the end of this paper shows several example security tiles)

### 3.3 Security Descriptor

The security descriptor outlines the specific security categories that protect a given part of the system. It makes it possible to see which security topics were taken into consideration for protecting which parts of a system class diagram. The type of security descriptor identifies what information is in the security tile. Seven possible stereotype attributes for the security descriptor are listed below. These categories are based primarily on some of the ideas in [1]; others can be added as appropriate.

- Principles
- Guidelines
- Survivability
- Attack Patterns
- Accountability
- Third Party Software
- Rules

**3.3.1. Principles.** A principle is "a statement of general security wisdom derived from experience" [1]. The purpose of this category is to define computer security principles and show how their implementations are covered in the UML class diagram. The security tile for a principle will normally contain a catalog entry defining the principle and a UML class diagram of how the principle protects parts of the system.

**3.3.2. Guidelines**. A guideline is "a recommendation for things to do or avoid during software development, described at the semantic level" [1]. The purpose of this category is to layout the security guidelines that have been agreed upon by analysts. The security tile then shows how these guidelines are met for each aspect of the system.

**3.3.3. Survivability**. Survivability is the ability of a system to work consistently until failure at which time recovery to the most recent working state is feasible. This category focuses on describing how parts of the system are backed up and recovered. The security tile for this usually is a class diagram depicting any hardware or software in the system.

**3.3.4. Attack Patterns**. An attack pattern is "developed by reasoning over large sets of software exploits" [1] and shows the ways a system is attacked. The purpose of an attack pattern in the security package is to define a set of system classes that have risks and show (in the security tile) how to prevent those risks. The security tile for attack patterns should consist of information on how to prevent this type of attack either through something as precise as a class diagram or as simple as providing tips on coding practices. To make it easier to identify and categorize various attack patterns, ten possible stereotype sub-attributes, based on information in [3], are described in Table 1.

**3.3.5. Third Party Software**. Third party software consists of any out of company software that is encapsulated within the system. The major reason for including third party software in the security tile is that any security flaws in the third party software must be taken into consideration so as to not compromise the larger system.

**3.3.6. Accountability**. Accountability is the ability of a system to log events and to assess what or who is responsible for those events. This descriptor is for any security that focuses on logging system events.

**3.3.7. Rules**. A rule is "a recommendation for things to do or to avoid during software development, described at the syntactic level" [1]. The major purpose of the rule attribute is to give security analysts a place to put coding practice reminders in the class design for developers. The goal is to make developers aware of common coding mistakes and

**Table 1. Attack pattern attributes.**

| Attribute | Description |
| --- | --- |
| Environment Variables | Environment variables are "variables that encapsulate information that does not change across executions of a program" [3]. The goal of this attack pattern is to take advantage of the fact that when a parent program executes a child program, it can control the child's environment variables. |
| Buffer Overflow | A buffer overflow is an attack by which a memory stack is overflowed so a system will execute information in memory outside of the stack. All the varieties of buffer overflows are represented in this category. |
| Data/Script Injections | The goal of this attack is to take advantage of a system running inputs given to it without checking the validity of the inputs. By passing scripts or incorrect information as input, a user can perform various tasks such as running remote processes, gaining sensitive information, or crashing the system. |
| Numeric Overflows | This attack focuses on giving a system some value that exceeds the bounds that the system can handle, with the intention of causing a system crash or worse. This is normally performed on some primitive type that has a known range (such as an integer). |
| Race Conditions | Race condition attacks are performed by getting a system to execute information out of order or at the same time; this can result in such problems as a system crash or data corruption. |
| Network Exposures | This attribute is for anytime there is a possibility of an attack against network-based applications. |
| Operational Misuse | Operational misuse can occur any place where the system is easily used incorrectly resulting in a security breach. One common example of this is places in the design where untrained users are present resulting in misuse. |
| Default Settings | This category is compromised of the collection of default settings that software products come with "out-of-the-box" that are insecure if not configured correctly. |
| Programmer Backdoors | This is the idea of developers working on the project intentionally or unintentionally leaving ways to access the system once it is complete. This attribute is for any instance in system design where analysts might try to take preventive action against this occurring. |
| Other | This attribute allows a designer to define an attack pattern that might not be as common as the ones above. It is included for the (likely) possibility of new attack patterns arising in the future. |

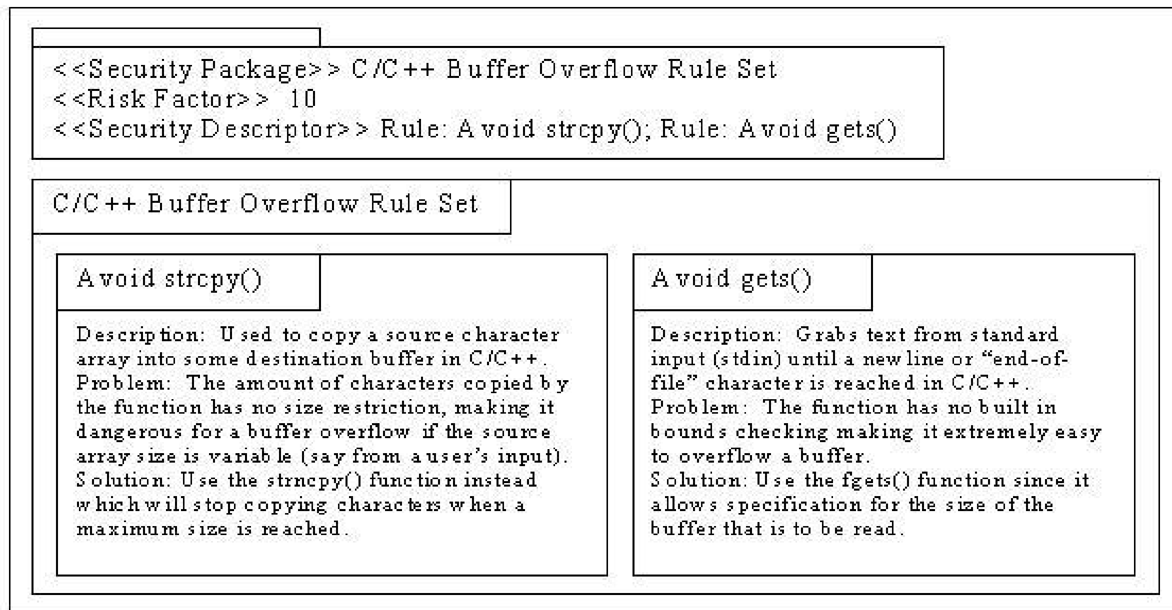


**Figure 1. Shows a brief example of a rule set.**

encourage them to remember and avoid these types of errors. This is extremely important since many software vulnerabilities are caused from insecure coding practice.

## 4. Security Package Combinations and Sets

As security packages are added to a class diagram some areas will have more of a security focus than others. To make the class diagram easier to read in these areas, it is good practice to use security package combinations and sets. A combination is a security package that contains a collection of different security descriptors that are all shared by the classes the security package protects. A set is a security package that contains a collection of the same security descriptors that are shared by the classes the security package protects. A *root security package* is the concept that makes combinations and sets possible. A root security package is a security package that has multiple security descriptors which means that its security tile contains security details for each descriptor. A good example of this is the creation of a *rule set*. A rule set is a security package set for the rule security descriptor so it combines a collection of similar rules. A simple example of this is shown in Figure 1, which shows part of a rule set for avoiding buffer overflows in C/C++. Note that because of space this example is not functionally complete and is shown simply for understanding the concept of a rule set. A security package combination is shown in the example in Figures 2 and 3 at the end of the paper. Overall, the goal of security package combinations and sets is to abstract a collection of related security ideas into a root security package to simplify the appearance of the system class diagram.

## 5. Connecting Security Packages and Security Tiles to Classes

Once you have defined a security package, it is necessary to show what parts of the system class diagram are protected by the package. To accomplish this, we use a new type of association with the stereotype <<protects>>. This shows which specific parts of a system are protected by each security package. Security packages that connect to nothing are assumed to protect all aspects of the system.

Another important detail is the difficult task of how UML diagrams in security tiles connect to the system class diagram (if needed). There are two basic situations where this can occur. The first situation is if the class diagram in the security tile is taking some input from the system diagram and then producing some output to the system diagram. The solution is to place the security package's <<protects>> stereotype to connect between the classes giving the input and the ones receiving the output. The security tile's class diagram should then depict where the input is taken in and the output is given out. The second situation is if the class diagram in the security tile has multiple connections throughout the system class diagram. The solution to this is to use the stereotype <<connects>> with a class name to show what and where the class diagram needs to connect. Examples of both of these situations can be seen in Figures 2 and 3.

## 6. A Brief Example

Let's now look at a brief example to show how UMLpac integrates onto a UML class diagram. Consider an extremely basic UML representation for an ATM that can

dispense cash and show an account balance to a user. A third party GUI package provides the interface for the ATM. A database, used as a backend, provides user account information. Figure 2 shows an example of a possible design for this system integrated with UMLpac for possible security considerations. Figure 3 shows the security tiles for the given security packages. Note that for both simplicity and space, the class diagram and security solutions are tremendously elementary and are not considered up to industry standard. The goal is to show how UML is integrated with UMLpac.

Several important features of Figures 2 and 3 should be noted.

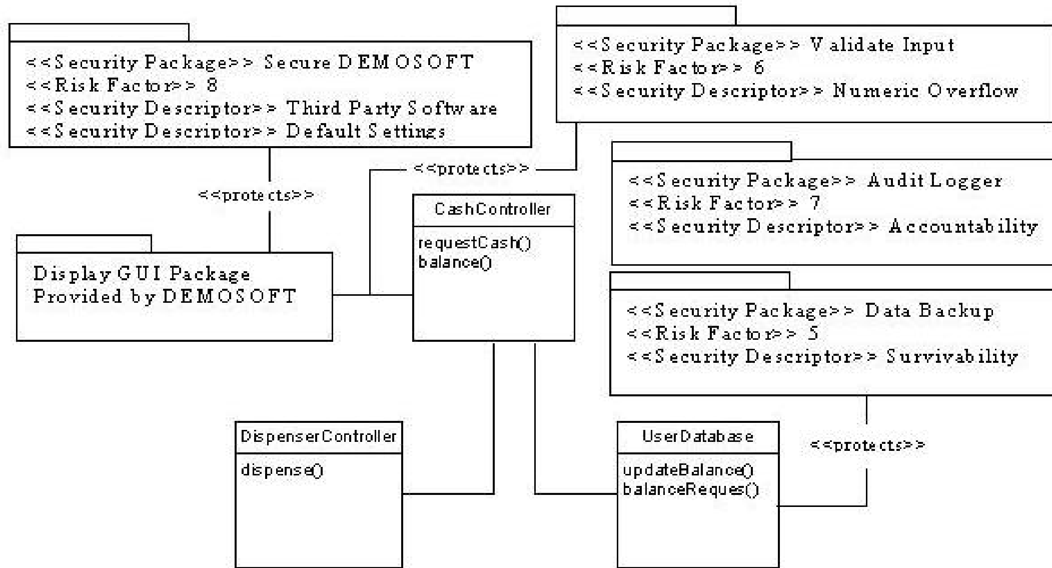


**Figure 2. Shows UML class diagram integrated with UMLpac for security features.**

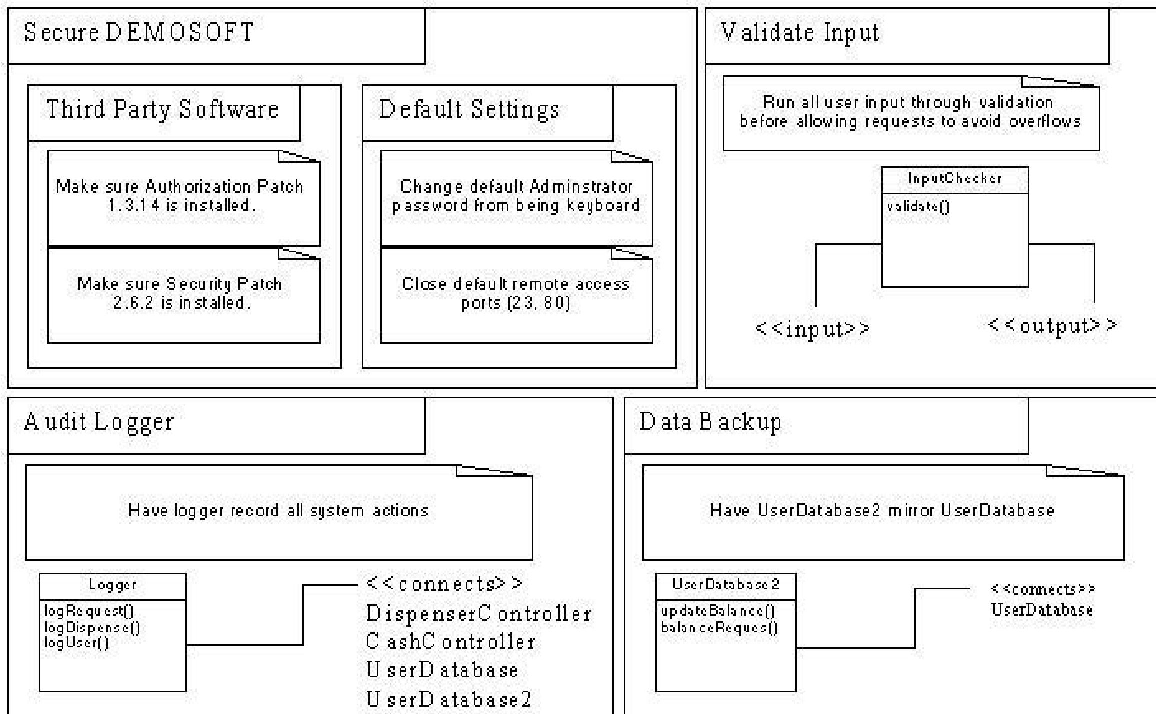


**Figure 3. Shows some example security tiles for the security packages in Figure 2.**

First, the idea of a security package combination is in the Secure DEMOSOFT security package. Second, the Validate Input security package shows the concept of representing a security tile that takes input and produces output. Finally, the idea of security tiles that connect to remote system elements is seen in the Audit Logger and Data Backup security packages.

## 7. Conclusions

This paper introduces an approach to improve the integration of security details into UML class diagrams using stereotypes and associated packages. The major goal of the approach, called UMLpac, is to keep a level of abstraction between the system class diagram and its security features so that the design continues to have simplicity. This goal is accomplished through the use of security packages that aid in separating basic security information from the in depth security implementation. This accomplishment was made even more efficient by the introduction of security package combinations and sets. UMLpac also succeeds in integrating security into the design by defining security threats to improve clarity through catalog descriptions, laying out high and low risk areas in design through use of a risk factor, and giving security analysts the ability to remind developers of secure coding practices through the use of rule security descriptors. These advantages make UMLpac an extremely useful approach and an ideal way to design secure software in the future.

Two additions to UMLpac will help to standardize and expand its use. The first of these is the creation of rule sets for various coding practices and the acceptance of these rule sets by industry. This will make it easy for designers to simply define standard rule sets in their UML class diagrams that will in turn remind developers of the secure coding practices they should follow when implementing a particular system. The second is the creation of a tool to help integrate security features into a class diagram using UMLpac. Along with this, the creation of complete guideline, rule, and principle catalogs as discussed in [1] will benefit the creation of security tiles that need a catalog entry. We expect that the approach taken in UMLpac will continue to expand as software security grows and develops in the future

## 8. Acknowledgement

This work was done at the University of South Carolina as part of the Research Experiences for Undergraduates in Multidisciplinary Computing project supported in part by National Science Foundation Award # 0353637.

## 9. References

[1] Barnum, S. and McGraw, G. “Knowledge for Software Security,” *IEEE Security & Privacy*, IEEE Computer Society Press, March/April, 2005, 74-78.

[2] Davis, N., Humphrey, W., Redwine, S. T. Jr., Zibulski, G. and McGraw, G. “Processes for Producing Secure Software: Summary of US National Cybersecurity Summit Subgroup Report,” *IEEE Security & Privacy*, IEEE Computer Society Press, May/June, 2004, 18-25.

[3] Gilliam, D. P., Wolfe, T. L., Sherif, J. S., and Bishop, M. “Software Security Checklist for the Software Life Cycle,” *Proceedings of the 12th IEEE WETICE*, IEEE Computer Society Press, 2003.

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

[5] Jurjens, J. “UMLsec: Extending UML for Secure Systems Development,” *Proc. of the 5th International Conference on the Unified Modeling Language*, 2002.

[6] Lodderstedt, T., Basin, D., and Doser, J. “SecureUML: A UML-Based Modeling Language for Model-Driven Security,” *Proc. of the 5th International Conference on the Unified Modeling Language*, 2002.

[7] Potter, B. and McGraw, G. “Software Security Testing,” *IEEE Security & Privacy*, IEEE Computer Society Press, September/October, 2004, pp. 81-85.

[8] Saltzer, J. and Schroeder, M. “The Protection of Information in Computer Systems,” *Proceedings of the IEEE,* volume 9, number 63, 1975, 1278-1308.

[9] Viega, J. and McGraw, G. *Building Secure Software*, Addison-Wesley, 2001.

[10] Whittaker, J. A. and Howard, M. “Building More Secure Software with Improved Development Processes,” *IEEE Security & Privacy*, IEEE Computer Society Press, November/December, 2004, 63-65.

[11] Whittaker, J. A. and Stytz, M. “Why Secure Applications Are Difficult to Write,” *IEEE Security & Privacy*, IEEE Computer Society Press, March/April, 2003, 81-83.