Association for Information Systems
AIS Electronic Library (AISeL)

AMCIS 2005 Proceedings

Americas Conference on Information Systems (AMCIS)

2005

# Secure Software Design Principles: A Systems Approach

Art Conklin
*The University of Texas at San Antonio*, art.conklin@utsa.edu

Glenn Dietrich
*The University of Texas at San Antonio*, glenn.dietrich@utsa.edu

# Secure Software Design Principles: A Systems Approach

**Art Conklin**
The University of Texas at San Antonio
art.conklin@utsa.edu

**Glenn Dietrich**
The University of Texas at San Antonio
glenn.dietrich@utsa.edu

**ABSTRACT**

The fact that security was often neglected in the design and construction of computer software has led to significant system changes in an attempt to add desired security functionality after the fact. Four methods of implementing security functionality, from augmentation through integration, are examined with respect to implementation strategy and efficacy of the desired security functionality. Using system theory, an examination of the issues associated with complex systems as applied to the addition of security functionality demonstrates the weaknesses of these approaches and the need to design security in from the beginning of a project. The application of system theory, the concepts of equifinality, feedback, control theory and the law of requisite variety assist in the understanding of the outcomes of the differing approaches to adding security to a design. The implications of understanding the foundational effects of adding security functionality will enable developers to properly invoke security in their designs.

**Keywords (Required)**

Security, secure software design, software engineering.

**INTRODUCTION**

Using systems theory to examine the problems and issues suggests that most fixes will not be successful in the general case. Systems theory also provides insight into effective methods of incorporating security into a computer system application. The purpose of this research is to compare four methods of incorporating security into a software application. These four methods, augmentation, interruption, incorporation and integration will be examined using system theory with respect to the level of variety they can accommodate.

A complex system can be described in a variety of ways. A system decomposition of parts is one method, a functional review is yet another. The point of reference used to describe a system acts as a framework for both understanding and manipulation of the system. Different points of view result in different reference frameworks and different levels of understanding of a system and the problems and solutions it delivers. For simple systems with limited functionality, the difference between these levels of understanding is minimal and of little consequence. With complex systems, such as most software today, the range of operational options is so vast that a complete understanding of all the interactions is impossible. In this case the viewpoint forms a reference that describes only a subset of the total system, including issues and problems. Operating outside the bounds of the defining framework may well be within the capability of the system, but it can result in unexpected behaviors and outcomes due to unforeseen interactions.

A key element in analyzing software functionality is the context within which it is employed. Most software involves interactions with users, making these users an integral part of the complete system. Just as poorly trained and low skilled users can have issues with software functionality, so too can highly trained and highly experienced users. The difference in problems comes from different elements and is a result of a lack of consideration of the other party's viewpoint with respect to functionality issues. An example of this can be seen in today's virus laden email environment. A common business problem, emails can contain viruses and worms and it is incumbent upon a business to develop a process to deal with the issue and protect their business resources from this malicious source of attack. Poorly trained users are susceptible to email viruses as they may open attachments without a clear understanding of their origin and purpose. To combat this system vulnerability software solutions have been deployed into email systems. Industry accepted practices include a software component, anti-virus software, awareness and training component for end users, and lastly a policy component that describes permissible uses of the email system. Together these are supposed to protect a network from outside virus and worm attacks. Yet this solution routinely fails in some form or another every day across the computing world.

Asking why it failed leads typically to blaming some component for a specific failure that allowed the undesired outcome to occur. While this may be true on occasion, it is more typically the case of an unintended interaction between elements of the

system that allowed the end result to occur. Take an email virus case. To prevent a threat that involves the attachment of zip files, the firm takes the position of scanning all zip files attached to emails and automatically destroying any executable files within them. End users run into issues as they may indeed have a legitimate need to use zip files to move large data sets around. They ask for special permission and are denied – management citing security as paramount in importance. But for the person needing to work, this creates a dilemma – they may not be able to complete their work in a satisfactory manner and it is doubtful that management will issue a new set of standards and directly address productivity issues in any meaningful manner. A highly skilled user is useful in that they can adapt their behavior to changing business conditions to keep their productivity high and their value to the business in good standing. This same behavioral trait is used to find an alternative to the now blocked business process – they will find a way around the zip file blockage.

The workers learn that encrypting the zip files prevents the software from flagging them and destroying them. The worm writer learns this as well and adapts his worm to encrypt the zip and provide enticing direction on how to unzip the file upon arrival in the inbox. Management responds by blocking all zip files. Users respond by manipulating the zip files so they pass the new restrictions. This battle goes on and on, with users using outside mail services such as hotmail and a variety of other methods to avoid the security restrictions. The user's effort is not intended to be insecure but to achieve the business purpose of the computer system. Security becomes a casualty in the war between productivity and management reactions to real business threats. The failure is typically seen as one of training and awareness, yet this is not the case at all. Ask a user to justify their action and they will explain them in terms of their job performance and productivity, issues measured without respect to the changes in security rules and enforcement of new policies. They will further add that they have taken measures to be sure that they are acting in a safe manner, citing that they know what they are doing and that their files are not corrupted.

## SYSTEMS THEORY

Examining the email example in terms of a system model, we have three main components; a detector, a comparator and an activator. The variable being monitored is the content of email attachment files. A detector exists in the system that finds and examines attachment files. A comparator determines the presence or absence of a zip format file for a given attachment. The activator removes the attachment if it is of zip file format type. Regulation exists in this system in its ability to detect and remove a specific type of file. System science refers to this type of control mechanism in terms of a reduction in variety.(Ashby 1963) The specific level of control that can be exerted by this system is bound by the law of requisite variety; "only variety in R can force down variety in D." (Ashby 1963) This role of variety with respect to control has the implication that the larger the variety of actions available to a control system, the larger the variety of perturbations it is able to compensate. In the email case, the regulator is only able to remove zip files, and when confronted with encrypted zip files, it is unable to handle this new level of variety. A point solution that is good for the specific case, but easily circumvented by an adapting environment. To have a robust system, one needs one that is able to adapt to changes in its environment and continue appropriate levels of regulation.

The true culprit is a lack of understanding of the complete nature of the email system and its role in daily business operations. Actions by administrators charged with protecting the network results in unexpected and uncorrected shifts in work processes. Users react to achieve their desired goals, and although it is not diametrically opposed to the administrator's goals, the differences in system level understanding results in crossed purposes. The blame lies not with the actors in the system, but their lack of complete understanding of the system and complete modeling of outcomes from actions with the system. Systems science has for years stressed the need to understand and describe systems with respect to their purpose and function, not their structure.(Churchman 1968) Yet the solution to block all zip files is rooted in a simple level of understanding structure and ignores effects on the purpose and function of the total system. And the complex system, which includes trained users as a component, adapts to changes and finds a new way to pursue the desired function and business purpose, changing structural aspects as needed to achieve this goal.

## SYSTEM MODELS

Examining the software engineering process, one sees a variety of methods by which security is applied to a system. A range of options exist, from being built in from the ground up as a strict requirement of the system to an add-on after the fact or even neglected in its entirety. The level of integration is typically a function of the significance of the system and its data, including severity of compromise and data integrity issues. Regardless of when it is put into a system, a key determinant in the effectiveness of security functionality stems directly from how it is implemented in the system. Security functionality can be added in four distinct fashions; it can be added as an augmentation to the system, an interruption to the process flow, incorporated into the process flow or fully integrated into the process flow. (Smith 2004)

### Augmentation

Taking a system and adding security functionality through augmentation is best described as the application of security functionality around the basic business functionality of a process. The security functionality exists separately from the business functionality. This can be done at any part of the development process, from the beginning of the design phase, through maintenance patches to existing systems. The strength of this method lies in its separation from the business functionality; hence it can be deployed separately from the business functionality and be expected not to impact the design functionality of the system in anyway. This makes augmentation an ideal methodology to add security functionality after the fact, as in maintenance patches or hotfix. The main weakness of this method also comes from the same source – the separation from business process functionality. This weakness is best described as a twist on the strength; the system will function with the security functionality removed or impaired, hence to get around it all one has to do is shut it off, or go around it.

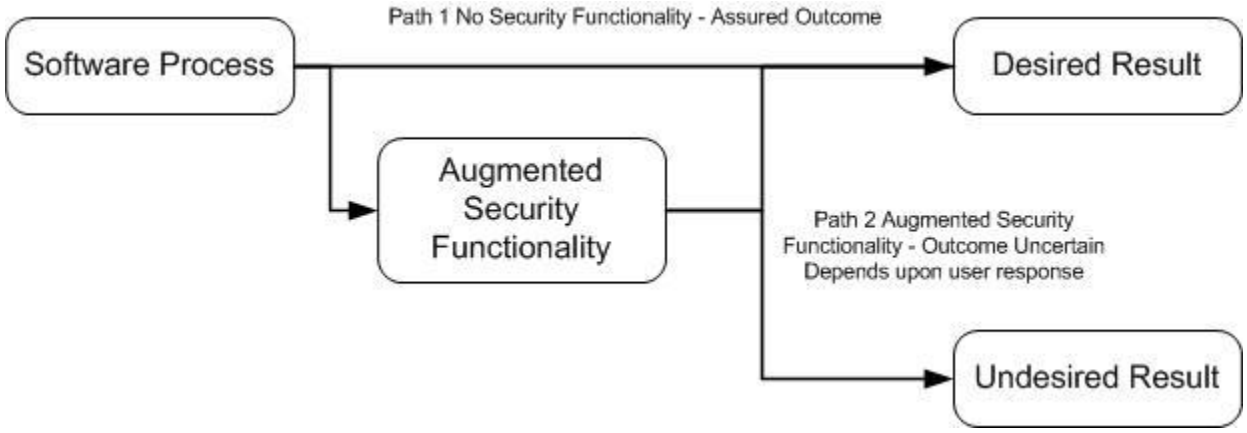


**Figure 1 Augmented Security Functionality**

Figure 1 illustrates the basic diagram of augmented security functionality. Two paths exist through the system, the first being the path through the software process that produces the desired result from a software process point of view. This is the normal operational flow of the software and is by definition the delivery of the desired outcome based on inputs. This is the default outcome if the user cancels or ignores security functionality input. The result is that the user attains the desired result from the specific application, although this result may also include a security issue, but from the original software process point of view, security issues were not part of the equation of success. The second path occurs when the user responds directly to the security functionality issue. Here, the outcome of the process is dependent upon the user's response and they may or may not achieve the desired system output as originally defined.

A good example of augmentation can be seen in the download warning issued before certain executables are installed on your machine. Suppose a user navigates to a web page that needs a specific plug-in. The browser responds with a dialog box asking whether or not the user wants the particular piece of software installed. The intent is to give the user a chance to prevent unwanted software from being installed on their machine. The reality is that most of the time the user doesn't understand the nature of the plug-in, the specific threat or risk involved and simply clicks through the security warning, allowing the installation. The basic mismatch between desired behavior and actual behavior comes from not providing sufficient information to the user so that they can make an informed decision. The typical user views the pop-up warning box as an annoying step that they must go through to get what they asked for, because they have already made the decision to go to the page they are downloading. They can get what they want – the page they are asking for, by simply going around the security mechanism.

### Interruption

Interruption based addition of security functionality can be viewed as an enforced following of the augmented path. The interruption is still placed after the application functionality, but before the final result is obtained. As shown in figure 2, the software process is followed and then passed through the security functionality as a filter before the final output is achieved. Whether or not the user achieves the desired final output is dependent upon how they respond to the invoked security functionality. This filter acts as an interruption to the 'normal' business process and is typically so viewed by the user.

Depending upon the nature of the added security functionality, several outcomes can take place. If the user is allowed to 'opt out' of the security decision, then just saying no will put them to their original desired solution. This effectively removes the security functionality from the loop, but does achieve the desired end user goal. If the user does not have the opportunity to 'opt out' and instead must deal with options, then the issue of making those options coincide with the interests of the user becomes a concern. If the user does not understand the options, or know what to do, then the perception is that security has broken the program and the user will attempt to find another way to get the task done. If the user understands and successfully completes any security functionality related tasking then the question of what outcome is arrived at becomes a central issue. If the security functionality changes the original desired outcome state to some other state, then again security functionality is perceived as breaking the process and alternate methods will be attempted by the user. The only truly good solution is for the security functionality to be such that it is easy to comply with and it does not prevent the achievement of the desired business functionality end state.

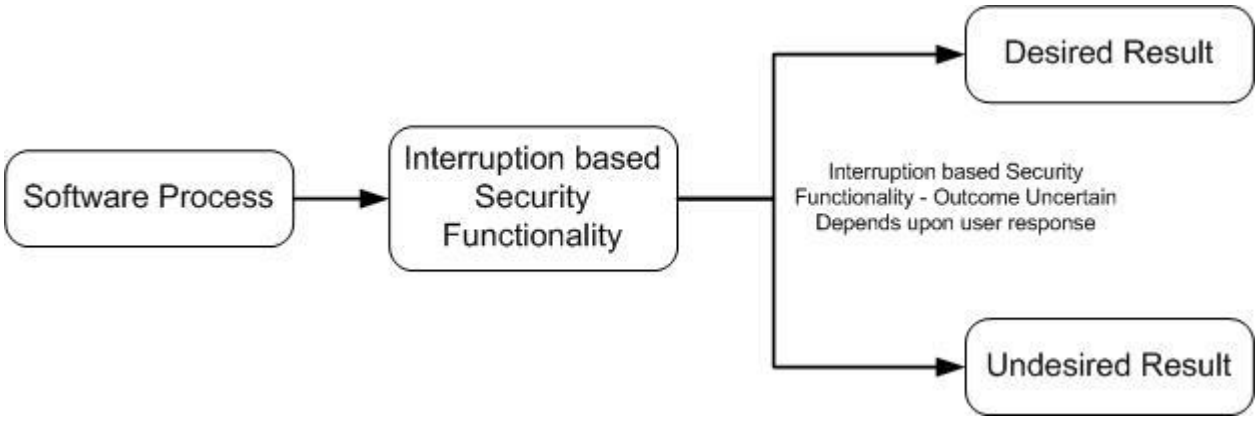


**Figure 2 Interruption Based Security Functionality**

An example of interruption can be seen in a simple email virus example. To prevent a user from inadvertently opening an infected file directly from the email system, certain types of attachments are not permitted to be directly opened from an email message. Instead they must be saved to disk before opening. The assumption is that virus scanning software can scan the file as it is being saved on the disk and hence before the attachment is opened. This can also be seen as shifting the security burden from the email application to another application, a risky assumption if the anti-virus is not installed, kept up-to-date or disabled. The user will typically consider this an inconvenience and as they are forced to open the file outside the email application will do so, but not with consideration of whether the file should be opened or not, for that decision point was already passed when the user was forced to save the file before opening.

### Incorporation

Incorporating security functionality into a software process so that the user is not in direct contact with the added functionality improves the odds of the functionality not being abrogated by user action. Incorporating security functionality increases the complexity of the software development process. Security functionality becomes another set of design requirements that must be addressed during the software engineering process although they can be done separately to a degree from the business logic functions. Properly designed, this level of security functionality can not be avoided by the user and provides a mechanism to log and trace user authorized activities. Users are still aware of the security related activity that is occurring, but it has become an accepted part of the business process being performed by the software.

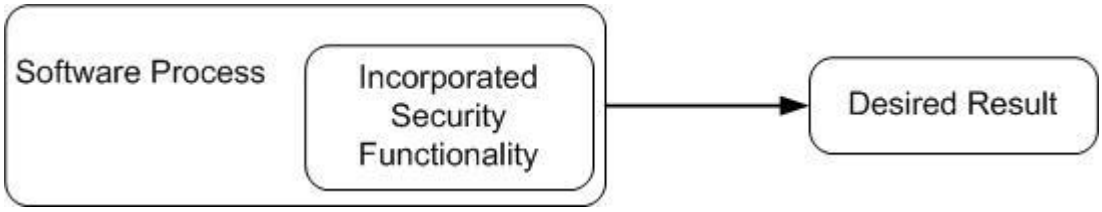


**Figure 3 Incorporation based Security Functionality**

An example of incorporation of security functionality is in the application of a license key when installing software on a computer. This process is hardly seamless, yet with the proper information an end user can enter the correct data and achieve the desired result – registered software.

### Integration

Integration of security functionality within the business process of the software is the ultimate objective in enforcing secure computing. Integration begins with the inclusion of security functionality requirements in the requirements process and involves designing them into the software in such a fashion as that they exclude or minimize the need for specific user input to address security requirements. The idea is to design security into the process so that the software process has only a secure path when implemented by a user. When integrated, security functionality is on even par with other system requirements and becomes transparent to the user in terms of desired result outputs. A good example of this type of system is Kerberos, an authentication method in wide use across a variety of platforms. (Neuman et al. 1994) Other examples are SSL and VPN connections. In these methods, the user has no real input to the system other than their own credentials. The software is designed to take care of the details, seamlessly and transparently to the user. Considering the complexity of these security protocols compared to the level of understanding of a typical user, this level of abstraction is essential for these protocols to function in a business environment.

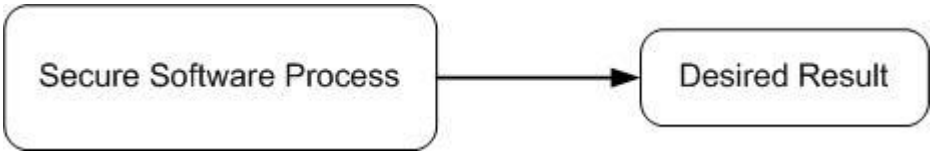


**Figure 4 Integrated Security Functionality**

## SYSTEM THEORY IMPLICATIONS

In all of the cases listed, there is a key element of the system in the form of the user. Frequently neglected, the user is not a passive component in the process and in fact is typically the active component that the security functionality is designed to guard against. When software is designed, users are either left out of the process or taken for granted. This results in a system that many times is hard for an end user to comprehend what the proper requested action is in a given situation; e.g. a warning that a certificate chain is broken. Software engineers, designers, architects and the managers of software engineering processes have a different level of understanding when it comes to technical issues surrounding security functionality. What is obvious in many cases to this group of people leaves ordinary users confused. As mentioned previously, computer systems are deployed to perform specific tasks or business processes. Software exists for the end user, not the developer. When the desired business functionality is compromised for any reason, it is the end user that has to find a way to complete the tasking, for the majority of business managers hold people accountable, not the software. Continued operation of the business process will be attempted by the system as a whole, including user interaction and abrogation of functionality not deemed necessary. Security functionality that counters this objective will be seen as an obstacle to overcome, and a highly skilled user may well indeed succeed in this effort. Use of training and awareness to change the user behavior is a typical response, yet these sessions are infrequent and typically not backed by daily operational management. Users are left with a dilemma of following the training or getting done what the boss wants and the simple reward system in most offices will drive behavior towards what the boss wants over other imperatives.

The implication of the user being part of the overall system is not universally accepted, nor is it evident in many application program designs. Users are an important element of the system as are the objectives that drive their behavior. Failure to include this into the design of security functionality merely places security functionality against the user and more often than not the user will prevail. Considering the user as a part of the system and evaluating the user correctly in terms of their skill level and ability are essential elements not just for usability but to prevent users from abrogating specific elements of the design. Designing a system that allows blank passwords by default is clearly not best practice. But neither is requiring a user to understand the implications of a broken certificate chain when validating software. The objective of an organization is to have efficient, effective computing and this includes security functionality. But it is not enough to preach to users about the importance of security and then have systems that place users in the quandary of "do I get my work done, or do I follow policy"? Crippling software functionality in the name of security and then forcing the user to go through extra steps to achieve desired functionality is counter productive, both from a user perspective and a total security perspective.

Considering the end user to be a part of the environment of the system, and hence an input to the system is also a possible means of looking at the problem. In this case, one could consider this a case of equifinality, one where multiple solutions lead to equivalent acceptable conclusions.(Jennings et al. 2003) The argument of equifinality applies to whether the user is in the system or in the environment, but not to the differing levels of control; augmentation, interruption, incorporation or integration. Equivalent solutions can be achieved, modeling the user as either part of the system, or part of the environment. The issue surrounding whether or not to include the user as part of the system is in reality not an issue. If they are part of the system, they are accounted for in the design. If they are not part of the system, the system interface to them as part of the environment is defined and again they are accounted for in the system design. There are two paths, but the outcome is the same, hence equifinality applies.

Examining the solutions from differing control methodologies shows that the results are substantially different. These differing methodologies have differing levels of variety in their control mechanism and thus have differing levels of ability in controlling environmental variety in a satisfactory fashion. Examining the issue of security functionality in terms of systems theory paradigm proposes several salient observations. This examination permits the identification of opportunities for better system level understanding based on system theory paradigms. (Ashmos et al. 1987) Augmentation, by design has the least ability for general control, limiting its control actions by the specific instance of its adaptation to the unregulated system model. Interruption has more variety in its control in that it is more closely connected to the system under regulation, but it too can be circumvented by an adapting environment (user). At the level of incorporation, the variety of control is less likely to be circumvented, for it is now part of the system, but it is still specifically reactive in its design and limited by the bounds of its ability to detect undesired variety in the input. Only when security functionality is included from basic design of the system, integrated as an integral part of the function of the system can the ability to detect and respond to variety be dynamic and comprehensively in step with the desired function of the system. Each of these solutions thus has an increasing ability in functionality to handle undesired variety in the input of a system. They are not equivalent in this sense; hence they do not represent a case of equifinality. Additionally the technical superiority of the integrated design is also shown through its increased ability to manage variety over the other models.

## SYSTEM DESIGN IMPLICATIONS

When developing any product there are more factors than just technical factors when considering viability. Cost and quality issues are also important. From a technical view point, there is a difference in end result depending upon the deployment methodology chosen. Integration of security functionality clearly provides a more robust solution, but this comes at the price of increased development costs and time to market. Augmentation, on the other end of the spectrum with respect to robustness of security functionality is typically cheaper to deploy and much easier to use to address security issues discovered after deployment through patching. Time to market for augmentation type implementations can be very quick, in some cases in a matter of hours. Interruption and incorporation both offer intermediate levels of robustness and are also in the middle with respect to cost and time. The best solution is always the one that works and this means that there are times and places for each of these deployment methodologies.(Davis et al. 2004) Augmentation may be fine for emergency patches where there isn't time to test the effects of code changes on business functionality. Hot fixes that are issued in augmentation mode could be refined into interruption or better yet incorporation methodology when testing time has permitted a stable software change. Integration is limited by cost and timing issues to new development and significant code overhauls.

Which is best, which is desired? Clearly the case of integration is the strongest from a technical point of view, yet it is bounded by the aspect that it is designed into a system and thus can not be applied to existing systems without significant redesign. The varying levels of system integration mean less and less redesign until you reach augmentation, or bolt on security functionality. Clearly augmentation is the least capable solution, but in many cases, may be the only workable solution in a real world situation of a newly discovered vulnerability in an existing application. The selection of the correct

methodology rests with the development team. Early in a project lifecycle, clearly integration is a better approach, for it offers more ability. As the lifecycle of a product advances, interruption and incorporation may be options during upgrades and subsystem re-writes. And as last effort stop gap defense for deployed systems, augmentation clearly has a temporary role until better measures can be built into the system. The true answer lies in the correct application of the correct model depending upon the circumstances and resources. This presupposes knowledge of the types and their differences in capabilities and requirements. The steps to solving a problem are first admit that there is a problem, second determine the alternatives and third implement a suitable corrective solution. This research is designed to assist developers to understand the alternatives and why they are different so that they can make an informed decision when implementing secure designs.

## REFERENCES

1. Ashby, W.R (1963). *An Introduction to Cybernetics* John Wiley, Sons, New York, p. 295.
2. Ashmos, D.P., and Huber, G.P. (1987), The Systems Paradigm in Organization Theory: Correcting the Record and Suggesting the Future, *Academy of Management Review,* 12:4, pp 607-621.
3. Churchman, C.W. (1968), *The Systems Approach*, Dell Publishing Company, Inc., New York, NY.
4. Davis, N., Humphrey, W., Jr., S.T.R., Zibulski, G., and McGraw, G. (2004), Processes for producing secure software, *IEEE Security & Privacy*, 2:3, pp 18-25.
5. Jennings, D.F., Rajaratnam, D., and Lawrence, F.B. (2003), Strategy-Performance Relationships in Service Firms:A test for equifinality, *Journal of Managerial Issues*, XV:2, pp 208-220.
6. Neuman, B.C., and Ts'o, T. (1994), Kerberos: an authentication service for computer networks, *IEEE Communications*, 32:9, pp 33-38.
7. Smith, B. (2004), Security For Everyone: How Product Design Must Change, ITVision.04, Microsoft, Redmond, WA.