OWASP
The Open Web Application Security Project
6th OWASP AppSec Conference - Italy 2007

Software Security - The Bigger Picture

Rudolph Araujo – Principal Software Security Consultant

Foundstone Professional Services, A Division of McAfee

1 Abstract

Developers are often blamed for software security mishaps and punished through losses in wages or embarrassed on walls of shame[1]! At Foundstone, we believe developers, for the most part, don't write insecure code intentionally or because they are negligent, they do so because they haven't been taught any better and don't receive adequate help and guidance from other stakeholders. Essentially, when dealing with software security, it is often a common fallacy to focus all together too much on the development phase of the software development lifecycle and not enough on the others. This paper therefore focuses on three key support activities that could help tremendously in improving the security of projects churned out by your development teams:

- Security Requirements Engineering - As with software quality in general, the lack of security requirements leads to insecure software.
- Security Acceptance Testing - Quality assurance teams specialize in testing software yet rarely test for security. And no, we don't mean penetration testing!
- Security Knowledge Management - When a security incident occurs can we ensure lessons are learned across the organization?

Through our experience having worked with a number of organizations to augment their software development lifecycles we have discovered how such activities can help produce higher quality and secure applications that make everyone happy especially developers who keep their reputation, jobs and hard-earned pay checks!

2 Introduction

The security community and industry has evolved tremendously since the late 80s when the first "security attack" was perpetrated in the form of the Morris Worm. This led to the creation of the Computer Emergency Response Team or CERT as it is popularly known. For the next decade or so the focus on the industry was on securing the network and to a lesser extent on securing the host. As a result of this, the major security technologies of that era were the devices and software we almost take for granted today - the firewalls, intrusion detection systems and virus scanners. However, as the Internet exploded and the World Wide Web went from being an academic network of computers to a platform upon which business was done, the threats also evolved. Now the attackers began to attack not just the network and the host but the applications that sat on top of these. In many ways these applications represented the crown jewels - the confidential data, the precious intellectual property and business intelligence that organizations and indeed consumers did not want to lose. Increasingly, organizations succeeded in getting their “ducks in a row" on the network and host side as tried and tested solutions became available. However, development teams were struggling with dealing with securing the application. Enter vulnerabilities such as the buffer overflow, SQL injection, and cross site scripting; the list could go on.

So how have we dealt with this problem over the last few years? As one would expect, the first attempt at dealing with problems was to not deal with them at all. The approach was to release software, hope for the best and then fix issues as they were publicly reported. Next came the phase of penetration testing a few weeks or days before going into production. This again provided little time to effectively fix the issues discovered. As an industry, we continued to evolve and the next phase was to go hunting through code for the common classes of vulnerabilities that were in the news – whether this was buffer overflows in the 90s or common web application vulnerabilities more recently. The more strategic of the organizations at this point invested in software security training and building policies such as language specific coding standards to aid their developers to deal with the problem and to prevent the introduction of vulnerabilities in the future. The focus from the beginning has been on developers and the development phase for the most part and only rarely touching on some of the secure design elements. As a consequence of this focus, it almost became instinctive to blame developers and hold them responsible for vulnerabilities in the application. If something went wrong, it must have been the developer's fault - especially now that we have this great firewall solution and this secure coding standard!

3 Holistic Software Security

Unfortunately it appears that as a community we the software security folks have not learned as much as we should have from the decades of research into software engineering. If you treat a security vulnerability as a bug first and a security issue second, you can quickly adapt many of the lessons that have been learned with regards to improving the security of software applications.

Software security must be viewed holistically. It is achieved through the combination of effective people, process and technology with none of these three on their own capable of fully replacing the other two. This also means that just like software quality in general, software security requires that we focus on security throughout the application's life cycle - or from cradle to grave as some like to say. Unfortunately thus far, most of the effort has focused on activities such as application penetration testing, security code reviews and to a lesser extent on threat modeling.

While all of the aforementioned activities are critical to improving the security of your applications, they are by no means the only ones. Unfortunately, both as a community at large and as individuals looking to tackle the software security problem in our development teams we have tended to ignore the non-developer focused activities. In this paper we present three of these activities. We share the experiences we have gained in effectively implementing these activities for large development teams as well as the value they bring to improving the security of the applications developed by these teams.

3.1 The Foundstone Security Frame[2]

Before we dig into the specifics of each of the three activities that are the focus of this paper, it helps to define a common frame of reference to view software security problems and solutions. Defining such a frame has helped to both be better prepared going into a software development project as well as to perform better root cause analysis when faced with vulnerabilities. In the context of this paper, we will use this security frame in each of the three activities to help us be more efficient, effective and thorough within each domain.

- Configuration Management: As part of this category we consider all issues surrounding the security of configuration information and deployment. For instance, any authentication and / or authorization rules embedded in configuration files or how the framework and application deal with error messages.

- Data Protection in Storage & Transit: The nature of issues included in this category cover the handling of sensitive information such as social security numbers, user credentials or credit card information. It is also covers the quality of cryptographic primitives being used, required / minimum key lengths, entropy and usage vis-à-vis industry standards and best practices.

- Authentication: We consider here the usage of strong protocols to validate the identity of a user or component. Further, issues such as the possibility or potential for authentication attacks such as brute-force or dictionary based guessing attacks also fall within the realm of this category.

- Authorization: The types of issues that are considered under this category include those dealing with appropriate mechanisms to enforce access control on protected resources in the system. Authorization flaws could result in either horizontal or vertical privilege escalation.

- User & Session Management: This category is concerned with how a user's account and session is managed within the application. The quality of session identifiers and the mechanism for maintaining sessions are some of the considerations here. Similarly, user management issues such as user provisioning and de-provisioning, password management and policies are also covered as part of this category.

- Data Validation: This is the category responsible for the most well known bugs and flaws including buffer overflows, SQL injection and cross site scripting. Length, range, format and type checking for inputs and outputs are considerations here.

- Error Handling & Exception Management: This category is responsible for ensuring that all failure conditions such as errors and exceptions are dealt with in a secure manner. The nature of issues covered in this category range from detailed error messages, which lead to information disclosure, to how user friendly security error messages are.

- Auditing and Logging: This category of issues is concerned with how information is logged for debugging and auditing purposes. The security of the logging mechanism itself, the need and presence of an audit trail and information disclosure through log files are all important aspects.

3.2 Security Requirements Engineering

One of the most ignored parts of a security enhanced software development life cycle is the security requirements engineering process. One of the prime reasons for this oversight is that security is assumed to be a technical issue and therefore best handled during architecture and design or better still during implementation. Since software requirements are often written by business analysts who are non-technical, this is a common conclusion.

The problem with this approach, as any experienced software professional would tell you, is that software which does not have its requirements elicited, enumerated and well documented will most likely be lacking in quality. This is because developers do not have a specific target with regards to embodying security into the design and implementation. Further, quality assurance folks have no benchmark to validate the software against, and traceability - a key software engineering attribute - is unachievable. In fact it is hard to even build a good threat model without a clear idea of the security requirements.

This is a well understood concept in the general field of software engineering. A lot of research[3] has been performed on how to effectively elicit, validate and document software requirements. Further, most modern SDLC support tools already provide some mechanism for documenting requirements[4]. Hence, it should not be too difficult to extend these systems and the process itself to include security requirements. The challenge however as mentioned above is that most organizations we work with are used to thinking solely about functional requirements – requirements that the system and business analysts writing them can put their arms around. What different widgets should the application have? How should it respond to the click of a button in the top right corner and so on? The non-functional requirements on the other hand are often marked as "N/A". Our findings have been that this is not necessarily because they are considered unimportant, but because they are assumed to be *de facto* requirements – "the developers should know better than to build a slow or insecure or unreliable system". The assumption always seems to be that these requirements would be obvious and hence don't need to be documented.

On examining this problem a little bit further, we discovered that the problem to a large extent was a lack of awareness and knowledge of the people writing the requirements. The non-functional requirements can be very technical –consider specification of the encryption algorithm, cipher mode, key lengths and rotation parameters. Defining requirements around all of those would typically require a detailed understanding of the mechanisms around cryptography - not something that is typically found in the job description of a business analyst.

As a solution to this issue we present a template driven approach which is designed specifically to help the non-technical stakeholder to define very technical security requirements. While this approach does involve some amount of prior effort to create the templates, we have seen it to be tremendously effective in both ensuring that security requirements are documented (and not just with "N/A"!) and then implemented and tested.

The first step in this approach is for an organization or team (depending on the size and variety of applications involved) to identify all the relevant drivers for security requirements that would, could and should influence development. In our experience, most often you will see a lot of commonality among the various applications developed within the organization or team and hence we attempt to leverage that commonality and thus gain efficiencies across multiple projects

In our experience it is best to think about these drivers along the following categories. As mentioned above, most of these drivers will influence many, if not all, of the applications churned out within an organization.

- Regulatory Compliance[5] – This involves specific requirements that would be mandated by various governmental agencies. Depending on the legal environment within which the organization operates and the application's scope, a number of regulations could be relevant. Some of these include:
    - Sarbanes-Oxley, Section 404
    - Health Insurance Portability and Accountability Act
    - Payment Card Industry Data Security Standard
    - Gramm-Leachy Bliley Act
    - SB 1386 and other State Notification Laws
    - BASEL II
    - Federal Information Security Management Act
    - EU Data Protection Directive
    - Children's Online Privacy Protection Act
    - Local Key Escrow Laws
- Industry Regulations and Standards: These include typically standards that are specific to an industry such as financial services. This category in our classification is also setup to include standards bodies such as ISO and the norms they define. Examples include:
    - ISO 17799
    - FFIEC Information Technology Examination Handbook[6]
    - SCADA Security[7]
    - OWASP Standards[8]
    - OASIS[9]
- Company Policies: Most organizations that we work with have a slew of internal policies that should and could affect the development of an application. Among the most common ones here are:
    - Privacy Policies
    - Coding Standards
    - Patching Policies
    - Data Classification Policies
    - Information Security Policies
    - Acceptable Use Policies
    - Export Control
    - Open Source Usage
- Security Features: Finally, most applications will have some form of security feature. For instance, authentication and authorization models that replicate real world role based access control. Similarly, administrative interfaces that will be used for user management including provisioning and de-provisioning.

For some of the above axes it is best to work with the legal department and internal audit to arrive at the list of relevant regulations. Once that superset has been defined, the next step is to examine each of these regulations through the eyes of both someone who speaks legalese and a software development expert. The aim here is to convert the list of legal requirements which would guarantee compliance to a set of core technical requirements for software that is impacted by these regulations. The Foundstone Security Frame can come in extremely handy here. For each of the relevant drivers from above consider the various categories in the security frame and how they might be impacted. For instance, if your organization is regulated by Gramm-Leach-Bliley Act (GLBA), privacy of personally identifiable information (PII) is absolutely critical. This in turn can have implications across multiple Security Frame categories not the least of which being Data Protection in Storage & Transit. The outcome of this step should essentially be a set of specific requirements along the various security frame categories that would satisfy each of the drivers defined above. It is vital at this stage to also rationalize the various requirements obtained above getting rid of overlapping or redundant requirements.

A parallel step in this requirements process is to classify the applications as being impacted by the drivers. In our experience this is best done by creating a large matrix with the various drivers above forming the columns and the application set forming the rows. Classification then is the task of checking the appropriate boxes depending on whether, based on legal and other opinions an application is impacted by a specific driver.

As a result of the two parallel steps mentioned above, the team should now have a specific set of technical requirements for each application based on its requirement driver environment. All of the above effort is intended to be performed once and then revisited periodically. In our experience, it is very rare that these change very often or with each application release. This is primarily because applications tend to evolve very slowly with regards to the drivers mentioned above. Further, as mentioned above there is much opportunity to leverage commonality across applications as well since it is not atypical for many of the applications to be operating within a similar driver environment.

Having now defined this universal set of requirements *a priori*, as each application release is defined; the specific set of requirements for that release can be drawn out of this set. As part of this process, the data classification and privacy policy can help to identify which data elements handled by the application are impacted by the drivers. Additionally, it is also important to consider which features being added in this release would be impacted as well. Based on these pieces of input and the universal set of requirements a subset of those requirements will be obtained that are relevant for this specific release of this specific application. The person formulating these requirements now need not be an expert in security or any of the security frame categories but can simply check the appropriate boxes to obtain a set of requirements. In fact this last per application step can be easily automated through a template or lightweight application which references all the relevant policies, the universal set of requirements, considers the data elements in use and provides a set of technical requirements that may leverage encryption, access control and other security mechanisms. These can then literally be copy-pasted into the master requirements list.

To wrap-up this section let us consider an illustrative example. Take for instance, an online loan processing application. Such an application will obviously make extensive use of personally identifiable information and is determined to be impacted by the Gramm-Leachy Bliley Act driver. This in turn defines specific requirements around the confidentiality, integrity, availability and access to data as well as audit trails that monitor and report on such access. Now consider that a new feature is being added that emails the result of the loan decision to the customer. When a business analyst is defining the requirements around this new feature, he / she would need to consider all of the different data elements that would be part of this email, the transport mechanism used by the email and authentication around it. Based on business need and security, it can then be decided to avoid certain data elements or perhaps use a secure email solution.

3.3 Security Acceptance Testing

As touched upon in the introduction to this paper, traditional security tests have focused on penetration testing. However, in most software development life cycles we have a specific quality assurance (QA) phase. Moreover, very commonly now we see unit tests and build verification tests in addition to the QA phase. Unfortunately, penetration testing is often performed too late in the life cycle - after the entire system has been built and deployed. It would be remiss for us to not leverage these earlier testing opportunities to catch as many problems as early as possible.

However, incorporating security into these testing methodologies is non-trivial. Part of the reason is that the typical security tester's mindset is different from a quality assurance tester or developer. Both traditional software testers and security testers attempt to break down software, however, the former typically approach this problem by attempting to look for failures to meet a specific set of requirements and features. For instance, a classical software tester is concerned with whether the login feature works i.e. when they type in a valid user name and password that the specific user is logged on. Further, they will also test to make sure that if an incorrect username and password are entered the user is not logged in. A security tester on the other is not so much concerned with the login feature but with getting access to the application in an unauthorized manner. Hence, he / she are going to attempt brute forcing or SQL injection to access the account. Similarly, he / she will be paying special attention to the error messages displayed on a failed login. You can see a clear difference in the two approaches and similar examples can be drawn from all of the various features of the application. Hence, the first step in embodying security testing within the usual testing phases in a software development lifecycle is to work on developing the right mindset. This is primarily an issue of training and exposure. Using training tools such as the Hacme Series[10] from Foundstone, software testers can learn about the various types of vulnerabilities as well as the simple mechanisms used to test for the presence of such vulnerabilities.

The next step is to determine the level of effort of security testing that should be embodied within the development cycle. Most QA folks are used to estimating testing effort based on the number of features and software requirements. They do this by being intimately involved in the functional and design reviews.

Along similar lines it is important to involve these stakeholders in the threat modeling process. This helps to not only identify security risks but can also help prioritize those risks based on business impact and thus help identify the list of test areas. Once this has been done, it is useful to simply integrate the relevant test cases into test plans that exercise those areas during a regular testing schedule.

Testing can then be split into a number of phases. In our experience, different parts of the security frame lend themselves better to some phases rather than others.

Unit testing is typically performed by the developers themselves and is often used as the exit criteria from the development phase to a formal testing phase. Testers here are looking for both coverage and pass percentages. A number of frameworks have been developed to help with this process and to essentially provide the infrastructure to make developers more efficient and effective. One such framework is provided within Visual Studio 2005. The framework can auto-generate unit test cases based on the APIs and functions exposed by your code. This is therefore an effective place to test data validation code. By resorting to fuzzing -style techniques which provide random data ("fuzz") to the inputs of an application, exposed interfaces can be tested to observe how they react to different data elements both valid and invalid. It is possible to test here for issues such as buffer overflows, SQL injection and cross site scripting as well as other types of injection attacks. Unit testing can also be useful in testing method level authorization rules wherein the called method checks the authentication state and permissions of the caller.

In our experience the most effective manner to build such security unit test cases is to create an attack library[11] with commonly used attack patterns and then running through this library enumerating attacks against the API being tested. The results in turn can be compared against a known good value to determine if the attack was successful. It is often best if the unit test cases are defined within a peer development group. Thus, one developer defines the unit test cases and attack libraries for code written by a different developer.

Build verification testing is another area that could provide opportunities to test the security of an application. A number of security testing tools such as source code analyzers and web application penetration testing tools provide both the ability to integrate into build scripts as well as scripting interfaces that allow their core engines to be adapted to fit into existing testing processes. Further, these tools also often have rules engines that can be modified and extended to create custom rule sets based on risk and relevance. Organizations that have successfully deployed such tools will often define criteria for build acceptance that specify along with other more traditional QA parameters, the number and type of security bugs that maybe present in a build that is provided to the QA teams for rigorous testing.

One of the key lessons we have learned over the years having worked with QA teams is that they are very accepting of new testing techniques and methodologies. However, it is important to not cause any upheaval of the actual QA process. It is therefore vital that all security testing ultimately ties into the existing and often times tried and testing quality practices. One critical area this becomes relevant is in the bug reporting area. We have seen most success when security bugs prior to release are treated just like other software quality issues. This implies that firstly, all security issues are entered into the same bug tracking system and follow the same lifecycle. For instance, a number of organizations are used to thinking about bugs based on their severity and priority. It is therefore vital to convert the risk rating that is usually associated with security bugs into a severity and priority. These can vary based on the business risk that these bugs pose but as a general rule of thumb all high risk issues can be considered to have the highest severity and priority, medium risk to have medium severity but high priority and all low risk issues to have low severity and medium priority. It is however not atypical to see business rules that mandate all security issues be treated as highest priority. Once the security issues have been entered into the bug tracking system they should follow the same path other bugs do in being assigned to a developer, fixed, assigned back to the tester verification and being closed and potentially added to regression testing cycles. It is however recommended to tag security bugs under a separate SECURITY category in order to facilitate obtaining an aggregate view. Secondly this can also help to ensure that a security trained developer is assigned to work on these bugs rather than any developer. Doing this will improve the chances of a correct solution free from recurrent bugs. Further, given that most bug tracking systems allow for their schema to be extended it is also helpful to further classify security bugs based on the security frame. As mentioned above, performing statistical analysis across bugs or after a release can thus help identify root causes as well as techniques and tools to prevent such bugs from manifesting themselves again.

Finally, it is also helpful to classify security issues into four categories while entering them into the bug tracking system or while commenting on the fix:

- Security Flaws: A security flaw is an issue that has come about due to an inappropriate design decision or implementation choice. They are generally architectural or design problems and have global implications. An example of a flaw would be a user authentication system that does not adequately protect passwords in the data store. Moreover this class of problems often stem from the fact that the design specification was itself incomplete or didn’t address the issue specifically. Thus, it is quite possible that a flaw might exist in an application even though the developers have implemented the design specification completely and correctly.

- Security Bugs: A security bug is a code level problem. Often semantic in nature, they are usually the result of a coding error or bad implementation of a design decision. An example of a bug is a buffer overflow. These issues typically tend to be language specific.

- Commendations: A commendation is a note of good security. Our experience is that this is as important to note the positive security attributes as it is the negative. Positive enforcement encourages repetition of good practices and can also help to highlight best practices that may have been implemented unbeknownst of their security value.

- Recommendations: There are many ways of designing software. These findings list some considerations for future revisions that would facilitate enhanced or improved security. This category of issues is also used to convey any insight into better software development practices that the application developers can adopt. For instance, specific software or architecture design patterns might be recommended.

This classification like the security frame classification can provide valuable results that help measure the effectiveness and merit of performing security testing earlier rather than later in the lifecycle. They can also help reemphasize the need and value of activities such as threat modeling and peer code reviews.

3.4 Security Knowledge Management

Of all the activities in the software development life cycle, this is probably the one that may not seem very important. Unfortunately as members of the security community we have all too often seen organizations fall victim to issues that were fixed in other parts of the organization - sometimes even within the same team! While this is embarrassing, it is easily a symptom of the lack of effective knowledge management. Valuable lessons can be learned from the experiences and mistakes of others both within the same organization and externally. However, most development teams have no way of drawing these lessons and it is therefore vital to share this information through an appropriate channel. While training is certainly an important aspect in improving overall security consciousness among developers, it is also important that developers have access to a central repository and portal for providing them with guidance on a day to day basis. This is especially important in large development organizations.

In our experience, the medium that lends itself best to this form of information sharing is a software security portal. Such a portal would provide a number of key functions such as:

- Document repository to house all of the policies, methodologies, process documents, guidelines and best practices developed as part of the security enhanced SDLC process.

- Threat modeling artifact storage so that incremental threat modeling can be easily performed after the initial effort of building the first threat model. Moreover, this is especially significant since it is not uncommon that a number of the applications within an organization or group are architecturally similar in nature and technology and hence share a similar attack surface. Thus the threat model for one such application can serve as an excellent building block for the others.

- Metrics reporting to provide the stakeholders with measurements to justify the return on security investment. These can include statistics from actual measurements within the organization on effectiveness of the S-SDLC process, improvements in productivity, decrease in security vulnerabilities, and other data points of interest.

In addition, another tremendously effective tool is an organization wide knowledge sharing Wiki. Through its support for effective content management and quick refactoring, Wiki[12] technology is an excellent model for collaboration between large numbers of individuals. One has to only look at the success of wikipedia.org at building a collaborative online encyclopedia to understand the potential contained within this technology. Having been in the field working with development teams it is quite common to observe that a number of groups and development teams are already using Wikis within their teams to document design and architecture as well as lessons learned from prior bugs and testing efforts. When this is the case extending such a Wiki to encompass the software security drive is not hard. Further, there are a number of free and useful Wiki solutions available such as the very popular MediaWiki[13].

A software security Wiki would be a central repository providing readers with a single and common location where they can find information covering aspects such as:

- security architectures that are in use within other groups
- thoroughly reviewed and tested code snippets for commonly used tasks
- links to additional information about software security on the Internet as well as,
- lessons learned from previous security issues identified in applications during internal testing or third party reviews.

We strongly believe that by encouraging such sharing, development teams and organizations can recognize tremendous gains by the wide distribution of best practices, prevention of the repetition of similar mistakes, improved productivity through code and knowledge sharing, and overall better software quality.

Especially when considering the disclosure of information about vulnerabilities onto such a knowledge sharing platform, some considerations are vital. Firstly, such sharing must only be considered after the issues in question have been fixed and tested in production applications. This will mitigate any risk of the issue being exploited. Special care must be taken when the vulnerability affects other applications or if it exists in a product or library that maybe shared. The bottom line is the risk of disclosing the vulnerability too soon and before affected applications have been thoroughly patched must be weighed against the benefits to be gained by the knowledge sharing. Secondly, for each issue it is important to sufficiently anonymize specific data. The aim behind doing this is to avoid any kind of finger pointing that could have a negative impact on morale. On the other hand all such sharing must be performed with a positive outlook of learning from past mistakes rather than focusing on the person or team that made the mistake. Finally, it is also vital that not just the issue be shared but also if possible the mechanism used to discover that issue, the design and architectural changes and thought process that went into fixing the issue, the fix itself as well as root cause analysis covering why the issue was introduced, why it was not caught earlier in the lifecycle and if and how the software development lifecycle process and mechanisms will be tweaked to prevent such issues from making their way undetected into applications. It is off course vital to make sure that any code shared thus be thoroughly reviewed and be free from security bugs and flaws itself.

Another aspect of knowledge sharing deals with the handling of third party components. A number of software development projects these days leverage third party components (both open source and otherwise) and ship these with their applications. Perhaps the biggest case in point is libraries such as OpenSSL [14]and zlib[15]. While as much as possible it is recommended to use existing tried and tested solutions, it is also important to track updates and changes to those solutions. Because components such as these are used so often they are extensively tested both by the teams that produce them as well as by the security researcher community[16][17]. This in turn implies that security updates and patches are being constantly released. If development teams are not tracking these updates it is quite likely that they would be distributing an old and possibly vulnerable version of these shared, third party libraries. On the other hand, the typical developer may not have the wherewithal to be constantly scouring the Internet and security mailing lists to keep track of the latest security vulnerabilities and patches. It is therefore vital that as a team or an organization a process be created to deal with this issue.

An effective way of doing this is to create a listing of all the open source and shared third party components in use across the organization along with a matrix that tracks which applications use which components. Alongside this listing, it is also important to maintain a link to the mailing list maintained by the vendor through which it notifies about security updates and patches. All of this information can in fact be maintained on the software security portal and wiki mentioned above. Beyond this however, it is important to assign a point person to each component whose responsibility it would be to track such mailing lists for their specific component. As soon as they learn of an update they can then notify the other teams using that component based on the matrix described above. Further it is also important that the point person also track aspects such as reliability issues with the patches that may affect the decision about whether to deploy the updated components.

In our experience without such a mechanism it is all too common to see applications that are free from vulnerabilities themselves but are exposed to highly critical problems in third party components. A knowledge management scheme as mentioned above not only can help prevent this by defining the appropriate roles and responsibilities but can also help share that information across the development team and organization to prevent others from falling victim to it as well.

4 Conclusion

This paper has covered three of the activities that we believe should exist as part of a security enhanced software development life cycle for it to be successful in improving the security of your applications. These activities are non-traditional in the sense that they do not merely go after the development phase of the lifecycle. However in our experience, having helped a number of large organizations implement a secure development lifecycle, we believe that without getting these parts of the puzzle correct your team will not achieve the best possible results from any investment into software security. It is also important to note that we cover only three such activities in this paper in order to provide them the justice they deserve. However, they are by no means the only activities. A truly effective security enhanced software development lifecycle has many parts to it and while they can be built over time, true Zen is achieved only when all the parts are in place. In getting there though, each additional part will result in a marked improvement in the security quality of your applications.

5 About the Author

Rudolph serves as a Principal Software Security Consultant and Trainer at Foundstone Professional Services, A Division of McAfee. Rudolph is responsible for creating and delivering the threat modeling, security code review and secure software engineering service lines. He is also responsible for content creation and training delivery for Foundstone's Building Secure Software and Writing Secure Code - ASP.NET and C++ classes. At Foundstone, Rudolph has had the opportunity to work with some of the biggest financial institutions, technology and telecom companies as well as with the open source community.

5.1 Experience

Rudolph has a solid background in computer science fundamentals and many years of software development experience on UNIX and Windows at all levels of the application stack. Prior to joining Foundstone, Rudolph led the checks development team for BindView's (now Symantec) bv-Control for Internet Security, a vulnerability assessment product. In his role as lead developer, Rudolph collaborated with a global team in creating updates to the product, which scanned for the presence of vulnerabilities as they were released.

Rudolph has also worked as a software developer at Morgan Stanley, where he was responsible for creating Microsoft Office based solutions for the Equity Research group. Most recently, Rudolph was a researcher at Carnegie Mellon University's CYLAB, investigating virus and worm threats, especially over peer-to-peer networks. His research interests also span the domain of web service security, survivability, and reliability.

Rudolph has diverse experience in a number of areas of software development and security. He has worked with both independent software vendors as well as large corporate IT organizations. Because of this, he has a unique perspective on the challenges of building real-world secure applications.

5.2 Notable Accomplishments

Rudolph is an experienced C / C++ and C#/.NET developer and the author of a number of Foundstone's software security auditing tools including The .NET Security Toolkit, SSLDigger, and Hacme Bank tools. Rudolph is also a regular contributor to MSDN's webcast series. Rudolph has been honored with the Microsoft Visual Developer Security MVP Award in recognition of his thought leadership and contributions to the application security and developer communities.

Rudolph is also a contributor to multiple online and print journals such as MSDN, IEEE Security & Privacy and Software Magazine, where he writes a column on writing secure code. He has also written the foreword for the Microsoft Patterns and Practices Group's Web Services Security Guide.

5.3 Professional Education

Rudolph earned an MS from Carnegie Mellon University specializing in Information Security and a BS in Computer Engineering from Goa University in India.

[1] http://news.zdnet.co.uk/software/developer/0,39020387,39228663,00.htm
[2] http://www.codesecurely.org/wiki/
[3] http://en.wikipedia.org/wiki/Requirements_analysis
[4] http://msdn2.microsoft.com/en-us/teamsystem/default.aspx
[5] http://msdn2.microsoft.com/en-us/library/aa480484.aspx
[6] http://www.ffiec.gov/ffiecinfobase/index.html
[7] http://www.sandia.gov/scada/standards_and_outreach.htm
[8] http://www.owasp.org/index.php/Category:OWASP_Guide_Project
[9] http://www.oasis-open.org/
[10] http://www.foundstone.com/index.htm?subnav=resources/navigation.htm&subcontent=/resources/s3i_tools.htm
[11] http://www.cs.umd.edu/ugrad/current/Research/Reilly&Marshall.pdf
[12] http://en.wikipedia.org/wiki/Wiki
[13] http://www.mediawiki.org/
[14] http://www.openssl.org/
[15] http://www.zlib.net/
[16] http://osvdb.org/searchdb.php?action=search_title&vuln_title=openssl
[17] http://osvdb.org/searchdb.php?action=search_title&vuln_title=zlib