M_SERVICE JSC	
FORM	SECURE CODE REVIEW GUILINES

UPDATE HISTORY

DATE	VERSION	DESCRIPTION	AUTHOR
10/03/2015	1.0	Create document	Dinh Huy Cuong

APPROVAL HISTORY

CREATOR	NAME: DINH HUY CUONG DATE: 6/2016 POSITION: SECURITY SPECIALIST
REVIEWER	NAME: PHAN CANH NHAT DATE: 6/2016 POSITION:
APPROVER	NAME: DATE: POSITION:

Contents

I. PURPOSE

This guidelines for conducting code security reviewing in M_Service.

II. SCOPE

III. REFERENCES

IV. DEFINITIONS

V. REVIEW GUIDELINES

1. Methodology

Code review is systematic examination of computer source code and reviews are done in various forms and can be accomplished in various stages of each organization S-SDLC. This book does not attempt to tell each organization how to implement code reviews in their organization but this section does go over in generic terms and methodology of doing code reviews from informal walkthroughs, formal inspections, or Tool-assist-ed code reviews.

1.1. Factors to Consider when Developing a Code Review Process

When planning to execute a security code review, there are multiple factors to consider since every code review is unique to its context. In addition to the elements discussed in this section, one must consider any technical or business related factors (business decisions such as deadlines and resources) that impact the analysis as these factors and may ultimately decide the course of the code review and the most effective way to execute it.

Risks

It is impossible to secure everything at 100%, therefore it is essential to prioritize what features and components must be securely reviewed with a risk based approach. While this project highlights some of the vital areas of design security peer programmers should review all code being submitted to a repository, not all code will receive the attention and scrutiny of a secure code review.

Purpose & Context

Computer programs have different purposes and consequently the grade of security will vary depending on the functionality being implemented. A payment web application will have higher security standards than a promotional website. Stay reminded of what the business wants to protect. In the case of a payment application, data such as credit cards will have the highest priority however in the case of a promotional website, one of the most important things to protect would be the connection credentials to the web servers. This is another way to place context into a risk-based approach. Persons conducting the security review should be aware of these priorities.

Lines of Code

An indicator of the amount of work is the number of lines of code that must be reviewed. IDEs (Integrated Development Environments) such as Visual Studio or Eclipse contain features which allows the amount of lines of code to be calculated, or in Unix/Linux there are simple tools like 'wc' that can count the lines. Programs written in object-oriented languages are divided into classes and each class is equivalent to a page of code.

Generally line numbers help pinpoint the exact location of the code that must be corrected and is very useful when reviewing corrections done by a developer (such as the history in a code repository). The more lines of code a program contains, the greater the chances that errors are present in the code.

Programming language

Programs written in typed safe languages (such as C# or Java) are less vulnerable to certain security bugs such as buffer overflows than others like C and C++. When executing code review, the kind of language will determine the types of expected bugs. Typically software houses tend towards a few languages that their programmers are experienced in, however when a decision is made to create new code in a language new to the developer management must be aware of the increased risk of securely reviewing that code due to the lack of in-house experience. Throughout this guide, sections explain the most common issues surrounding the specific programming language code to be reviewed, use this as a reference to spot specific security issues in the code.

Resources, Time & Deadlines

As ever, this is a fundamental factor. A proper code review for a complex program will take longer and it will need higher analysis skills than a simple one. The risks involved if resources are not properly provided are higher. Make sure that this is clearly assessed when executing a review

1.2. Integrating Code Reviews in the S-SDLC

Code reviews exist in every formal Secure Software Development Lifecycle (S-SDLC), but code reviews also vary widely in their level of formality. To confuse the subject more, code reviews vary in purpose and in relation to what the code reviewer is looking for, be it security, compliance, programming style, etc. Throughout the S-SDLC (XP, Agile, RAD, BSIMM, CMMI, Microsoft ALM) there are points where an application security SME should to be involved. The idea of integrating secure code reviews into an S-SLDC may sound daunting as there is another layer of complexity or additional cost and time to an already over budget and time constrained project. However it is proven to be cost effective and provides an additional level of security that static analyzers cannot provide.

In some industries the drive for secure enhancements to a company's S-SDLC may not be driven purely by the desire to produce better code, these industries have regulations and laws that demand a level of due care when writing software (e.g. the governmental and financial industries) and the fines levelled at a company who has not attempted to secure

their S-SDLC will be far greater than the costs of adding security into the development lifecycle.

When integrating secure code reviews into the S-SDLC the organization should create standards and policies that the secure code reviewer should adhere to. This will create the right importance of the task so it is not just looked at as a project task that just needs to be checked off. Project time also needs to be assigned to the task so there is enough time to complete the tasks (and for any remedial tasks that come out of the secure code review). Standards also allow management and security experts (e.g. CISOs, security architects) to direct employees on what secure coding is to be adhered to, and allows the employees to refer to the (standard) when review arbitration is necessary.

Today most organizations have modified their S-SDLC process to add agile into their S-SDLC process. Because of this the organization is going to need to look at their own internal development practices to best determine where and how often secure code reviews need to happen. If the project is late and over budget then this increases the chance that a software fix could cause a secure vulnerability since now the emphasis is on getting the project to deployment quicker. Code reviews for code in production may find software vulnerabilities but understand that there is a race with hackers to find the bug and the vulnerable software will remain in production while the remedial fix is being worked on.

1.3. When to Code Review

Once an organizations decides to include code reviews part of their internal code process. The next big question is when should the code be reviewed. This section talks about three possible ways to include code reviews. There are three stages be in the SDLC when code can be reviewed:

When code is about to be checked in (pre-commit)

The development organization can state in their process that all code has to be reviewed before the code can be submitted to the source code repository. This has the disadvantage of slowing the check-in process down, as the review can take time, however it has many advantages in that below standard code is never placed in the code line, and management can be confident that (if processes are being followed) the submitted code is at the quality that has been stipulated.

For example, processes may state that code to be submitted must include links to requirements and design documentation and necessary unit and automated tests. This way the reviewers will have context on the exact code modification being done (due to the documentation) and they will know how the developer has tested the code (due to the tests). If the peer reviewers do not think the documentation is complete, or the tests are extensive enough, they can reject the review, not because of the code itself, but because the necessary docs or tests are not complete. In an environment using CI with automated tests running nightly, the development team as a whole will know the next day (following check-in) if the submitted code was of enough quality. Also management know that once a bug or

feature is checked in that the developer has finished their task, there's no "I'll finish those tests up next week" scenarios which adds risk to the development task.

When code has just been checked into a code base (post-commit)

Here the developer submits their code change, and then uses the code repository change-lists to send the code diff for review. This has the advantage of being faster for the developer as there's no review gate to pass before they check-in their code. The disadvantage is that, in practice, this method can lead to a lesser quality of code. A developer will be less inclined to fix smaller issues once the code has been checked in, usually with a mantra of "Well the code is in now, it'll do.". There also a risk of timing, as other developers could write other code fixes into the same module before the review is done or changes and tests have been written, meaning

• Link to documents related to task/feature, including requirements, design, testing and threat modeling documents.

• Code Review checklist if used, or link to organization Code Review checklist. (see appendix XXX)

• Testing the developer has carried out on the code. Preferably the unit or automated tests themselves can be part of the review submission.

• If any tools such as FxCop, BinScope Binary Analyzer, etc. were used prior to code review. the developer not only has to implement the code changes from the peer or security review, but they also have to do so in a way that does not break other subsequent changes. Suddenly the developer has to re-test the subsequent fixes to ensure no regressions.

Some development organizations using the Agile methodology add a 'security sprint' into their processes.

During the security sprint the code can be security reviewed, and have security specific test cases (written or automated) added.

When code audits are done

Some organizations have processes to review code at certain intervals (i.e. yearly) or when a vulnerable piece of code is suspected of being repeated throughout the code base. Here static code analyzers, or simple string searches through the code (for specific vulnerability patterns) can speed up the process. This review is not connected to the submission of a feature or bug fix, they are triggered by process considerations and are likely to involve the review of an entire application or code base rather than a review of a single submission.

1.4. Security Code Review for Agile and Waterfall Development

Today agile development is an umbrella term for a lot of practices that include programming, continuous integration, testing, project management, etc. There are many flavors of agile development, perhaps as many flavors as there are practitioners. Agile

development is a heterogeneous reference framework where the development team can pick what practices they want to use.

Agile has some practices that could affect how and when code is reviewed, for example agile tries to keep code review and testing as near as possible to the development phase. It is a common practice to define short development cycles (a.k.a. Iterations or Sprints). At the end of each cycle, all the code should be production quality code. It can be incomplete, but it must add some value. That affects the review process, as reviewing should be continuous.

From the point of view of secure coding review, it shouldn't make a difference if the development organization uses agile or waterfall development practices. Code review is aligned to the code submitted, not the order of feature development vs testing, or the time patterns assigned to the coding task. In many organizations the line between waterfall and agile is becoming blurred, with traditional waterfall departments introducing the continuous integration (CI) aspects from agile, including nightly builds, automated testing, test driven development, etc.

1.5. A Risk Based Approach to Code Review

A development house will have various degrees of code changes being reviewed, from simple one line bug fixes in backend scripts that run once a year, to large feature submissions in critical business logic. Typically the intensity of the code review varies based on the perceived risk that the change presents.

In the end, the scale of the code review comes down to the management of resources (skilled persons, company time, machines, etc.). It would not be scalable to bring in multiple security experts for every code change occurring on a product, the resources of those persons or those teams would not be large enough to handle every change. Therefore companies can make a call on which changes are important and need to be closely scrutinized, and which ones can be allowed through with minimal inspection. This will allow management to better size the development cycle, if a change is going to be done in an area which is high risk, management can know to set aside sufficient time for code review and ensure persons with relevant skills will be available. The process of deciding which changes need which level of code review is based on the risk level of the module the change is within.

If the review intensity of code changes is based on the risk level of the module being changed, who should decide the level of risk? Ultimately management is responsible for the output of a company, and thus they are responsible for the risk associated with products sold by the company. Therefore it is up to management (or persons delegated by management) to create a reproducible measure or framework for deciding the risk associated with a code change.

Decisions on the risk of a module or piece of code should be based on solid cost benefit analysis and it would be irresponsible to decide all modules are high risk. Therefore management should meet with persons who have an understanding of the code base and

security issues faced by the products, and create a measure of risk for various elements of code. Code could be split up into modules, directories, products, etc., each with a risk level associated with it.

Various methods exist in the realm of risk analysis to assign risk to entities, and many books have been dedicated to this type of discussion. The three main techniques for establishing risk are outlined in table 1 below.

OPTIONS FOR ESTABLISHING RISK	
Technique	**Method**
Quantitative	Bring people together and establish a monetary value on the potential loss associated with the code. Gauge the likelihood that the code could be compromised. Use dollar values produced from these calculations to determine the level
Qualitative	Bring people together and discuss opinions on what level of loss is associated with the modules, and opinions on likelihood of compromise. Qualitative does not attempt to nail down monetary associations with the loss, but tends towards the perception or opinion of associated losses.
Delphi	Independently interview or question people on the losses and compromises of the modules, whilst letting them know the feedback will be anonymous. The impression here is that the people will give more honest answers to the questions and will not be swayed by other people's arguments and answers.

Risk is chance of something bad happening and the damage that can be caused if it occurs. The criteria for deciding the risk profile of different code modules will be up to the management team responsible for delivering the changes, examples are provided in table 2.

COMMON CRITERIA FOR ESTABLISHING THE RISK PROFILE OF A CODE MODULE	
Criteria	**Explanation**
Ease of exposure	Is the code change in a piece of code directly exposed to the internet? Does an insider use the interface directly?
Value of loss	How much could be lost if the module has a vulnerability introduced? Does the module contain some critical password

	hashing mechanism, or a simple change to HTML border on some internal test tool?
Regulatory controls	If a piece of code implements business logic associated with a standard that must be complied with, then these modules can be considered high risk as the penalties for non-conformity can be high.

When levels of risk have been associated with products and modules, then the policies can be created determining what level of code review must be conducted. It could be that code changes in a level one risk module must be reviewed by 3 persons including a Security Architect, whereas changes in a level 4 risk module only need a quick one person peer review.

Other options (or criteria) for riskier modules can include demands on automated testing or static analysis, e.g. code changes in high risk code must include 80% code coverage on static analysis tools, and sufficient automated tests to ensure no regressions occur. These criteria can be demanded and checked as part of the code review to ensure they are capable of testing the changed code.

Some companies logically split their code into differing repositories, with more sensitive code appearing in a repository with a limited subset of developers having access. If the code is split in this fashion, then it must be remembered that only developers with access to the riskier code should be able to conduct reviews of that code.

Risk analysis could also be used during the code review to decide how to react to a code change that introduces risk into the product, as in table 3. In a typical risk analysis process, the team needs to decide whether to accept, transfer, avoid or reduce the risks. When it comes to code reviews it is not possible to transfer the risk as transferring risk normally means taking out insurance to cover the cost of exposure.

OPTIONS FOR HANDLING RISKS IDENTIFIED IN A CODE REVIEW	
Risk Resolution	**Explanation**
Reduce	This is the typical resolution path. When a code reviewer finds that the code change introduces risk into an element of business logic (or simply a bug) the code will be changed to fix the bug or code in a way that reduces the risk.
Accept	When the code change introduces a risk in the code but there is no other way to implement the business logic, the code change can pass code review if the risk is considered acceptable. The risk and any

	workarounds or mitigating factors should be documented correctly so that it is not ignored.
Avoid	When the code change introduces a risk that is too great to be accepted, and it is not possible to reduce the risk by implementing a code change, then the team need to consider not performing the change. Ideally this decision should be reached before the code review stage, but it is entirely possible that factors can arise during code implementation that changes the understood risk profile of a code module and prompts management to reconsider if a change should go ahead.

1.6. Code Review Preparation

A security review of the application should uncover common security bugs as well as the issues specific to business logic of the application. In order to effectively review a body of code it is important that the reviewers understand the business purpose of the application and the critical business impacts. The reviewers should understand the attack surface, identify the different threat agents and their motivations, and how they could potentially attack the application.

For the software developer whose code is being reviewed, performing code review can feel like an audit and developers may find it challenging to not take the feedback personally. A way to approach this is to create an atmosphere of collaboration be-tween the reviewer, the development team, the business representatives, and any other vested interests. Por-traying the image of an advisor and not a policeman is important to get co-operation from the development team.

The extent to which information gathering occurs will depend on the size of the organization, the skill set of the reviewers, and the criticality/risk of the code being reviewed. A small change to the CSS file in a 20- person start-up will not result in a full threat model and a separate secure review team. At the same time a new single sign-on authentication module in a multi-billion dollar company will not be secure code reviewed by a person who once read an article on secure coding. Even within the same organization, high-risk modules or applications may get threat modeled, where the lower risk modules can be reviewed with a lesser emphasis on the reviewer understanding the security model of the module.

This section will present the basic items the reviewer (or review team) should attempt to understand about the application subjected to a secure code review. This can be used in smaller companies that don't have the resources to create a full security baseline, or on low risk code within larger companies. A later section goes into detail on threat modeling, which would be used by larger companies on their highest risk code bases.

In an ideal world the reviewer would be involved in the design phase of the application, but this is rarely the case. However regardless of the size of the code change, the engineer initiating the code review should direct reviewers to any relevant architecture or design documents. The easiest way to do this is to include a link to the documents (assuming they're stored in an online document repository) in the initial e-mail, or in the code review tool. The reviewer can then verify that the key risks have been properly addressed by security controls and that those controls are used in the right places.

To effectively conduct the review the reviewer should develop familiarity with the following aspects:

Application features and Business Rules

The reviewer should understand all the features currently provided by the application and capture all the business restrictions/rules related to them. There is also a case for being mindful of potential future functionality that might be on the roadmap for an application, thereby future-proofing the security decisions made during current code reviews. What are the consequences of this system failing? Shall the enterprise be affected in any great way if the application cannot perform its functions as intended?

Context

All security is in context of what we are trying to secure. Recommending military standard security mechanisms on an application that vends apples would be overkill and out of context. What type of data is being manipulated or processed, and what would the damage to the company be if this data was compromised? Context is the "Holy Grail" of secure code inspection and risk assessment.

Sensitive Data

The reviewer should also make a note of the data entities like account numbers and passwords that are sensitive to the application. The categorizing the data entities based on their sensitivity will help the reviewer to determine the impact of any kind of data loss in the application.

User roles and access rights

It is important to understand the type of users allowed to access the application. Is it externally facing or internal to "trusted" users? Generally an application that is accessible only for the internal users of an organization might be exposed to threats that are different than the one that is available for anyone on the Internet. Hence, knowing the users of the application and its deployed environment would allow the reviewer to realize the threat agents correctly.

In addition to this, the different privileges levels present in the application must also be understood. It would help the reviewer to enumerate different security violations/privilege escalation attacks that can be applicable to the application.

Application type

This refers to understanding whether the application is browser based application, a desktop based standalone application, a web-service, a mobile applications or a hybrid application. Different type of application faces different kinds of security threats and understanding the type of the application would help the reviewer to look for specific security flaws, determine correct threats agents and highlight necessary controls suitable to the application.

Code

The language(s) used, the features and issues of that language from a security perspective. The issues a programmer needs to look out for and language best practices from a security and performance perspective.

Design

Generally web applications have a well-defined code layout if they are developed using MVC design principle. Applications can have their own custom design or they may use some well-known design frameworks like Struts/Spring etc. Where are the application properties/configuration parameters stored? How is the business class identified for any feature/URL? What types of classes get executed for any processing any request? (e.g. centralized controller, command classes, view pages etc.) How is the view rendered to the users for any request?

Company Standards and Guidelines

Many companies will have standards and guidelines dictated by management. This is how the management (ultimately responsible for the organizations information security) control what levels of security are applied to various functions, and how they should be applied. For example, if the company has a Secure Coding Guidelines document, reviewers should know and understand the guidelines and apply them during the code review.

1.7. Code Review Discovery and Gathering Information

The reviewers will need certain information about the application in order to be effective. Frequently, this information can be obtained by studying design documents, business requirements, functional specifications, test results, and the like. However, in most real-world projects, the documentation is significantly out of date and almost never has appropriate security information. If the development organization has procedures and templates for architecture and design documents, the reviewer can suggest updates to ensure security is considered (and documented) at these phases.

If the reviewers are initially unfamiliar with the application, one of the most effective ways to get started is to talk with the developers and the lead architect for the application. This does not have to be a long meeting, it could be a whiteboard session for the development team to share some basic information about the key security considerations and controls. A walkthrough of the actual running application is very helpful to give the reviewers a good

idea about how the application is intended to work. Also a brief overview of the structure of the code base and any libraries used can help the reviewers get started.

If the information about the application cannot be gained in any other way, then the reviewers will have to spend some time doing reconnaissance and sharing information about how the application appears to work by examining the code. Preferably this information can then be documented to aid future reviews.

Security code review is not simply about the code structure. It is important to remember the data; the reason that we review code is to ensure that it adequately protects the information and assets it has been entrusted with, such as money, intellectual property, trade secrets, or lives. The context of the data with which the application is intended to process is very important in establishing potential risk.

If the application is developed using an inbuilt/well-known design framework the answers to the most of these questions would be pre-defined. But, in case it is custom then this information will surely aid the review process, mainly in capturing the data flow and internal validations. Knowing the architecture of the application goes a long way in understanding the security threats that can be applicable to the application.

A design is a blueprint of an application; it lays a foundation for its development. It illustrates the layout of the application and identifies different application components needed for it. It is a structure that determines execution flow of the application. Most of the application designs are based on a concept of MVC. In such designs different components interact with each other in an ordered sequence to serve any user request. Design review should be an integral part of secure software development process. Design reviews also help to implementing the security requirements in a better way.

Collecting all the required information of the proposed design including flow charts, sequence diagrams, class diagrams and requirements documents to understand the objective of the proposed design. The design is thoroughly studied mainly with respect to the data flow, different application component interactions and data handling. This is achieved through manual analysis and discussions with the design or technical architect's team. The design and the architecture of the application must be understood thoroughly to analyze vulnerable areas that can lead to security breaches in the application

After analyzing the insecure areas in the design a list of security requirements corresponding to them must be created. Requirements are high level changes or additions to be incorporated in the design, for instance: Validate the inputs fetched from the web service response before processing them. Any protection that is needed for resolving the vulnerable area identified in the design would go as a security requirement for the design. This phase involves listing all the security requirements for the design along with security risk associated with them. This risk-based approach would help the development teams in prioritizing the security requirements.

Every security requirement should be associated with a security control best suited for the design. Here, we would identify exact changes or additions to be incorporated in the design that are needed to meet any requirement or mitigate a threat.

The list of security requirements and proposed controls can be then discussed with the development teams. The queries of the teams should be addressed and feasibility of incorporating the controls must be determined. Exceptions, if any must be taken into account and alternate recommendations should be proposed. In this phase a final agreement on the security controls is achieved. The final design incorporated by the development teams can be reviewed again and finalized for further development process.

1.8. Static Code Analysis

Static Code Analysis is carried out during the implementation phase of S-SDLC. Static code analysis commonly refers to running static code analysis tools that attempt to highlight possible vulnerabilities within the 'static' (non-running) source code.

Ideally static code analysis tools would automatically find security flaws with few false positives. That means it should have a high degree of confidence that the bugs that it finds are real flaws. However, this ideal is beyond the state of the art for many types of application security flaws. Thus, such tools frequently serve as aids for an analyst to help them zero in on security relevant portions of code so they can find flaws more efficiently, rather than a tool that finds all flaws automatically.

Bugs may exist in the application due to insecure code, design or configuration. Automated analysis can be carried on the application code to identify bugs through either of the following two options:

1. Static code scanner scripts based on a pattern search (in-house and open source).

2. Static code analyzers (commercial and open source)

1.9. Application Threat Modeling

Threat modeling is an in-depth approach for analyzing the security of an application. It is a structured approach that enables employees to identify, quantify, and address the security risks associated with an application. Threat modeling is not an approach to reviewing code, but it complements the secure code review process by providing context and risk analysis of the application. The inclusion of threat modeling in the S-SDLC can help to ensure that applications are being developed with security built-in from the very beginning. This, combined with the documentation produced as part of the threat modeling process, can give the reviewer a greater understanding of the system, allows the reviewer to see where the entry points to the application are (i.e. the attack surface) and the associated threats with each entry point (i.e. attack vectors).

The concept of threat modeling is not new but there has been a clear mind-set change in recent years. Modern threat modeling looks at a system from a potential attacker's perspective, as opposed to a defender's viewpoint. Many companies have been strong advocates of the process over the past number of years, including Microsoft who has made

threat modeling a core component of their S-SDLC, which they claim to be one of the reasons for the increased security of their products in recent years.

When source code analysis is performed outside the S-SDLC, such as on existing applications, the results of the threat modeling help in reducing the complexity of the source code analysis by promoting a risk based approach. Instead of reviewing all source code with equal focus, a reviewer can prioritize the security code review of components whose threat modeling has ranked with high risk threats.

1.10. Metrics and Code Review

Metrics measure the size and complexity of a piece of code. There is a long list of quality and security characteristics that can be considered when reviewing code (such as, but not limited to, correctness, efficiency, portability, maintainability, reliability and securability). No two-code review sessions will be the same so some judgment will be needed to decide the best path. Metrics can help decide the scale of a code review.

Metrics can also be recorded relating to the performance of the code reviewers and the accuracy of the review process, the performance of the code review function, and the efficiency and effectiveness of the code review function.

Some of the options for calculating the size of a review task include:

Lines of Code (LOC):

A count of the executable lines of code (commented-out code or blank lines are not counted). This gives a rough estimate but is not particularly scientific.

Function Point:

The estimation of software size by measuring functionality. The combination of a number of statements which perform a specific task, independent of programming language used or development methodology. In an object orientated language a class could be a functional point.

Defect Density:

The average occurrence of programming faults per Lines of Code (LOC). This gives a high level view of the code quality but not much more. Fault density on its own does not give rise to a pragmatic metric. Defect density would cover minor issues as well as major security flaws in the code; all are treated the same way. Security of code cannot be judged accurately using defect density alone.

Risk Density:

Similar to defect density, but discovered issues are rated by risk (high, medium & low). In doing this we can give insight into the quality of the code being developed via a [X Risk / LoC] or [Y Risk / Function Point] value (X&Y being high, medium or low risks) as defined by internal application development policies and standards.

For example:

4 High Risk Defects per 1000 (Lines of Code)

2 Medium Risk Defects per 3 Function Points

Cyclomatic complexity (CC):

A static analysis metric used to assist in the establishment of risk and stability estimations on an item of code, such as a class, method, or even a complete system. It was defined by Thomas McCabe in the 70's and it is easy to calculate and apply, hence its usefulness.

The McCabe cyclomatic complexity metric is designed to indicate a program's testability, understandability and maintainability. This is accomplished by measuring the control flow structure, in order to predict the difficulty of understanding, testing, maintaining, etc. Once the control flow structure is understood one can gain a realization of the extent to which the program is likely to contain defects. The cyclomatic complexity metric is intended to be independent of language and language format that measures the number of linearly independent paths through a program module. It is also the minimum number of paths that should be tested.

By knowing the cyclomatic complexity of the product, one can focus on the module with the highest complexity. This will most likely be one of the paths data will take, thus able to guide one to a potentially high risk location for vulnerabilities. The higher the complexity the greater potential for more bugs. The more bugs the higher the probability for more security flaws.

Does cyclomatic complexity reveal security risk? One will not know until after a review of the security posture of the module. The cyclomatic complexity metric provides a risk-based approach on where to begin to review and analyze the code. Securing an application is a complex task and in many ways complexity an enemy of security as software complexity can make software bugs hard to detect. Complexity of software increases over time as the product is updated or maintained

Cyclomatic complexity can be calculated as:

CC = Number of decisions +1

… where a decision would be considered as commands where execution is branched with if/else, switch, case, catch, while, do, templated class calls, etc.,

As the decision count increases, so do the complexity and the number of paths. Complex code leads to less stability and maintainability.

The more complex the code, the higher risk of defects. A company can establish thresholds for cyclomatic complexity for a module:

0-10: Stable code, acceptable complexity

11-15: Medium Risk, more complex

16-20: High Risk code, too many decisions for a unit of code.

Modules with a very high cyclomatic complexity are extremely complex and could be refactored into smaller methods.

Bad Fix Probability:

This is the probability of an error accidentally inserted into a program while trying to fix a previous error, known in some companies as a regression.

Cyclomatic Complexity: 1 – 10 == Bad Fix Probability: 5%

Cyclomatic Complexity: 20 –30 == Bad Fix Probability: 20%

Cyclomatic Complexity: > 50 == Bad Fix Probability: 40%

Cyclomatic Complexity: Approaching 100 == Bad Fix Probability: 60%

As the complexity of software increase so does the probability to introduce new errors.

Inspection Rate:

This metric can be used to get a rough idea of the required duration to perform a code review. The inspection rate is the rate of coverage a code reviewer can cover per unit of time. For example, a rate of 250 lines per hour could be a baseline. This rate should not be used as part of a measure of review quality, but simply to determine duration of the task.

Defect Detection Rate:

This metric measure the defects found per unit of time. Again, can be used to measure performance of the code review team, but not to be used as a quality measure. Defect detection rate would normally increase as the inspection rate (above) decreases.

Re-inspection Defect Rate:

The rate at which upon re-inspection of the code more defects exist, some defects still exist, or other defects manifest through an attempt to address previously discovered defects (regressions).

1.11. Crawling Code

Crawling code is the practice of scanning a code base of the review target and interface entry points, looking for key code pointers wherein possible security vulnerability might reside. Certain APIs are related to interfacing to the external world or file IO or user management, which are key areas for an attacker to focus on. In crawling code we look for APIs relating to these areas. We also need to look for business logic areas which may cause security issues, but generally these are bespoke methods which have bespoke names and

cannot be detected directly, even though we may touch on certain methods due to their relationship with a certain key API.

We also need to look for common issues relating to a specific language; issues that may not be security related but which may affect the stability/availability of the application in the case of extraordinary circumstances.

Other issues when performing a code review are areas such a simple copyright notice in order to protect one's intellectual property. Generally these issues should be part of a companies Coding Guidelines (or Standard), and should be enforceable during a code review. For example a reviewer can reject a code review because the code violates something in the Coding Guidelines, regardless of whether or not the code would work in its current state.

Crawling code can be done manually or in an automated fashion using automated tools. However working manually is probably not effective, as (as can be seen below) there are plenty of indicators, which can apply to a language. Tools as simple as grep or wingrep can be used. Other tools are available which would search for keywords relating to a specific programming language. If a team is using a particular review tool that allows it to specify strings to be highlighted in a review (e.g. Python based review tools using pygments syntax highlighter, or an in-house tool for which the team can change the source code) then they could add the relevant string indicators from the lists below and have them highlighted to reviewers automatically.

The basis of the code review is to locate and analyze areas of code, which may have application security implications. Assuming the code reviewer has a thorough understanding of the code, what it is intended to do, and the context in which it is to be used, firstly one needs to sweep the code base for areas of interest

2. Vulnerability Categories

2.1. A1 – INJECTION

2.1.1. Overview

What is Injection?

Injection attacks are common, widespread, an easy for a hacker to test if a web site is vulnerable and easy and quick for the attacker to take advantage of. Today they are very common in legacy applications that haven't been updated.

2.1.2. SQL Injection

The most common injection vulnerability is SQL injection. Injection vulnerability is also easy to remediate and protect against. This vulnerability covers SQL, LDAP, Xpath, OS commands, XML parsers.

Injection vulnerability can lead to…

1. Disclose/leaking of sensitive information.

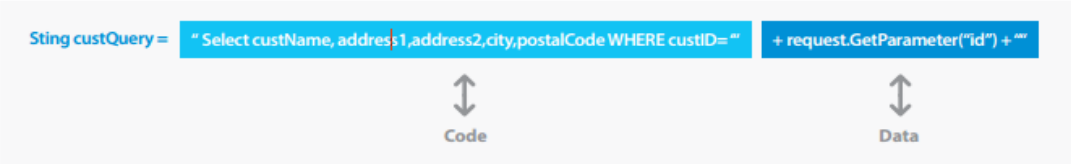
2. Data integrity issues. SQL injection may modify data, add new data, or delete data.

3. Elevated privilege.

4. Gain access to back-end network.

Two main points are the cause of injection vulnerabilities…

1. SQL commands are not protected from the untrusted input. SQL parser is not able to distinguish between code and data.



Using SQL strings is very common in legacy applications where developers were not considering security. The issue is this coding technique does not tell the parser which part of the statement is code and which part is data. This allows the attacker to modify the SQL statement by adding code to the data that was visible from the web site for the attacker to manipulate.

1. Untrusted input is acceptable by the application. There are several ways to mitigate injection vulnerability, whitelisting, regex, etc. The five best ways are. All five should be used together for a defense in depth approach.

2. HtmlEncode all user input.

3. Using static analysis tools. Most static analysis for languages like .NET, Java, python are accurate. However static analysis can become an issue when injection comes from JavaScript and CSS.

4. Using SQL platform API to parameterize the data so the SQL parser can distinguish between code and data.

5. Using Store Procedures. Store procedures again help the SQL parser to known the difference between code and data. However Store Procedures can be used to build dynamic SQL statements allowing the differentiation of code and data to become blended together causing the Store Procedure to become vulnerability to injection.

6. Provide developer training for best practices for secure coding.

2.1.3. Blind SQL Injection

So if the UI being developed does not return search results, an SQL cannot occur? Unfortunately attackers can still glean information based on the error responses from various UI elements. Blind SQL injection is a type of attack that asks the database true or false questions and determines the answer based on the applications response. Effectively the attacker uses SQL queries to determine what error responses are returned for valid SQL, and which responses are returned for invalid SQL. Then the attacker can probe; for example check if a table called "user_password_table" exists. Once they have that information, they could use an attack like the one described above to maliciously delete the table, or attempt to return

information from the table (does the username “john” exist?).

Blind SQL injections can also use timings instead of error messages, e.g. if invalid SQL takes 2 seconds to respond, but valid SQL returns in 0.5 seconds, the attacker can use this information.

2.1.4. Parameterized SQL Queries

Parameterized SQL queries (also called prepared statements) allow the SQL query string to be defined in such a way that the client input can’t be treated as part of the SQL syntax.

Take the following example:

```
1      String query = "SELECT id, firstname, lastname FROM authors WHERE
forename = ? and surname = ?";
2      PreparedStatement pstmt = connection.prepareStatement( query );
3      pstmt.setString( 1, firstname );
4      pstmt.setString( 2, lastname );
```

In this example the string ‘query’ is constructed in a way that it does not rely on any client input, and the ‘PreparedStatement’ is constructed from that string. When the client input is to be entered into the SQl, the ‘setString’ function is used and the first question mark “?” is replaced by the string value of ‘firstname’, the second question mark is replaced by the value of ‘lastname’. When the ‘setString’ function is called, this function automatically checks that no SQL syntax is contained within the string value. Most prepared statement APIs allow you to specify the type that should be entered, e.g. ‘setInt’, or ‘setBinary’, etc.

2.1.5. Safe String Concatenation?

So does this mean you can’t use string concatenation at all in your DB handling code? It is possible to use string concatenation safely, but it does increase the risk of an error, even without an attacker attempting to inject SQL syntax into your application.

You should never use string concatenation in combination with the client input. value Take an example where the existence (not the value) of a client input variable “surname” is used to construct the SQL query of the prepared statement;

```
1 String query = "Select id, firstname, lastname FROM authors WHERE forename
= ?";
2 if (lastname!= NULL && lastname.length != 0) {
3     query += " and surname = ?"; });
4 query += ";";
5
6      PreparedStatement pstmt = connection.prepareStatement( query );
7      pstmt.setString( 1, firstname);
8
9      if (lastname!= NULL || lastname.length != 0) { pstmt.setString( 2,
lastname ); }
```

Here the value of 'lastname' is not being used, but the existance of it is being evaluated. However there is still a risk when the SQL statement is larger and has more complex business logic involved in creating it. Take the following example where the function will search based on firstname or lastname:

```
1 String query = "select id, firstname, lastname FROM authors";
2 if ( firstname != NULL && firstname.length != 0 ) {
3     query += "WHERE forename = ?;" }
4  if ( lastname!= NULL && lastname.length != 0 ) {
5     query += "WHERE surname = ?;" }
6         PreparedStatement pstmt = connection.prepareStatement( query ):
```

This logic will be fine when either firstname, or lastname is given, however if neither were given then the SQL statement would not have any WHERE clause, and the entire table would be returned. This is not an SQL injection (the attacker has done nothing to cause this situation, except not passing two values) however the end result is the same, information has been leaked from the database, despite the fact that a parameterized query was used.

For this reason, the advice is to avoid using string concatenation to create SQL query strings, even when using parameterized queries, especially if the concatenation involves building any items in the where clause.

2.1.6. Using Flexible Parameterized Statements

Functional requirements often need the SQL query being executed to be flexible based on the user input, e.g. if the end user specifies a time span for their transaction search then this should be used, or they might wish to query based on either surname or forename, or both. In this case the safe string concatenation above could be used, however from a maintenance point of view this could invite future programmers to misunderstand the difference between safe concatenation and the unsafe version (using input string values directly).

One option for flexible parameterized Statements is to use 'if' statements to select the correct query based on the input values provided, for example:

```
1       String query;
2       PreparedStatement pstmt;
3
4       if ( (firstname!= NULL && firstname.length != 0) &&
5            lastname!= NULL && lastname.length != 0) ) {
6            query = "Select id, firstname, lastname FROM authors WHERE
forename = ? and surname = ?;"
```

```
7           pstmt = connection.prepareStatement( query );
8           pstmt.setString( 1, firstname );
9           pstmt.setString( 2, lastname );
10     }
11     else if (firstname != NULL && firstname.length != 0) {
12          query = "Select id, firstname, lastname FROM authors WHERE
forename = ?:";
13          pstmt = connection.prepareStatement( query );
14          pstmt.setString( 1, firstname );
15     }
16     else if (lastname != NULL && lastname.length != 0){
17          query = "Select id, firstname, lastname FROM authors WHERE
surname= ?:";
18          pstmt = connection.prepareStatement( query );
19          pstmt.setString( 1, lastname);
20     }
21     else{
22          throw NameNotSpecifiedException(); }
```

2.1.7. PHP SQL Injection

An SQL injection attack consists of injecting SQL query portions in the back-end database system via the client interface in the web application. The consequence of a successful exploitation of an SQL injection varies from just reading data to modifying data or executing system commands. SQL Injection in PHP remains the number one attack vector, and also the number one reason for Data Compromises

Example 1 :

```
1 <?php
1 $pass=$_GET["pass"];
2 $con = mysql_connect('localhost', 'owasp', 'abc123');
3  mysql_select_db("owasp_php", $con);
4  $sql="SELECT card FROM users WHERE password = '".$pass."'";
5 $result = mysql_query($sql);
6  ?>
```

Suspicious Validation

The most common ways to prevent SQL Injection in PHP are using functions such as addslashes() and mysql_real_escape_string() but those function can always cause SQL Injections in some cases.

"'addslashes :'"

you will avoid SQL injection using addslashes() only in the case when you wrap the query string with quotes.The following example would still be vulnerable

```
1      $id = addslashes( $_GET['id'] );
2     $query = 'SELECT title FROM books WHERE id = ' . $id;
```

"'mysql_real_escape_string():'"

mysql_real_escape_string() is a little bit more powerful than addslashes() as it calls MySQL's library function mysql_real_escape_string, which prepends backslashes to the following characters: \x00, \n, \r, \, ', " and \x1a.

As with addslashes(), mysql_real_escape_string() will only work if the query string is wrapped in quotes. A string such as the following would still be vulnerable to an SQL injection:

2.1.8. Java SQL Injection

```
1      $bid = mysql_real_escape_string( $_GET['id'] );
2      $query = 'SELECT title FROM books WHERE id = '. $bid
```

SQL injections occur when input to a web application is not controlled or sanitized before executing to the back-end database.

The attacker tries to exploit this vulnerability by passing SQL commands in her/his input and therefore will create a undesired response from the database such as providing information that bypasses the authorization and authentication programmed in the web application

An example of a vulnerable java code

```
1  HttpServletRequest request = ...;
2 String userName = request.getParameter("name");
3  Connection con  = ...
4   String query = "SELECT * FROM Users " + " WHERE                name = '"
+ userName + "'";
5 con.execute(query);
```

The input parameter "name" is passed to the String query without any proper validation or verification. The query 'SELECT* FROM users where name" is equal to the string 'username' can be easily misused to bypass something different that just the 'name'. For

example, the attacker can attempt to pass instead in this way accessing all user records and not only the one entitled to the specific user

```
"   OR  1=1.
```

2.1.9. .NET SQL Injection

Framework 1.0 & 2.0 might be more vulnerable to SQL injections than the later versions of .NET. Thanks to the proper implementation and use of design patterns already embedded in ASP.NET such as MVC(also depending on the version), it is possible to create applications free from SQL injections, however, there might be times where a developer might prefer to use SQL code directly in the code.

Example.

A developer creates a webpage with 3 fields and submit button, to search for employees on.

Field name, lastname + id

The developer implements a dynamic SQL statement or stored procedure in the code such as this one:

```
1 SqlDataAdapter thisCommand = new SqlDataAdapter(

2       "SELECT name, lastname FROM employees WHERE ei_id = '" +

          idNumber.Text + "'", thisConnection);
```

The first code is equivalent to the following SQL statement, which gets executed:

```
1  SqlDataAdapter thisCommand = new SqlDataAdapter(

2   "SearchEmployeeSP '" + idNumber.Text + "'", thisConnection);
```

SELECT name, lastname FROM employees WHERE idNumber = '567892'

A hacker can then insert the following statement and execute the following code:

SELECT name, lastname FROM authors WHERE au_id = ''; DROP DATABASE pubs --'

The semicolon ";" provides SQL with a signal that it has reached the end of the SQL statement, however, the hacker uses this to continue the statement with the malicious SQL code

; DROP DATABASE pubs

2.1.10.Parameter collections

Parameter collections such as SqlParameterCollection provide type checking and length

validation. If you use a parameters collection, input is treated as a literal value, and SQL Server does not treat it as executable code, and therefore the payload can not be injected. Using a parameters collection lets you enforce type and length checks. Values outside of the range trigger an exception. Make sure you handle the exception correctly. Example of the SqlParameterCollection:

Hibernate Query Language (HQL)

```
1 using (SqlConnection conn = new SqlConnection(connectionString)) {

2 DataSet dataObj = new DataSet();

3 SqlDataAdapter sqlAdapter = new SqlDataAdapter( "StoredProc", conn);
sqlAdapter.SelectCommand.CommandType =

4 CommandType.StoredProcedure;

5 sqlAdapter.SelectCommand.Parameters.Add("@usrId", SqlDbType.VarChar, 15);

6  sqlAdapter.SelectCommand.Parameters["@usrId "].Value = UID.Text;
```

Hibernate facilitates the storage and retrieval of Java domain objects via Object/Relational Mapping (ORM).

It is a very common misconception that ORM solutions, like hibernate, are SQL Injection proof. Hibernate allows the use of "native SQL" and defines a proprietary query language, called HQL (Hibernate Query Language); the former is prone to SQL Injection and the later is prone to HQL (or ORM) injection.

What to Review

- Always validate user input by testing type, length, format, and range.
- Test the size and data type of input and enforce appropriate limits.
- Test the content of string variables and accept only expected values. Reject entries that contain binary data, escape sequences, and comment characters.
- When you are working with XML documents, validate all data against its schema as it is entered.
- Never build SQL statements directly from user input.
- Use stored procedures to validate user input, when not using stored procedures use SQL API provided by platform. i.e. Parameterized Statements.
- Implement multiple layers of validation.
- Never concatenate user input that is not validated. String concatenation is the primary point of entry for script injection.

2.1.11.Client side JavaScript

JavaScript has several known security vulnerabilities, with HTML5 and JavaScript becoming more prevalent in web sites today and with more web sites moving to responsive web design with its dependence on JavaScript the code reviewer needs to understand what

vulnerabilities to look for. JavaScript is fast becoming a significant point of entry of hackers to web application. For that reason we have included in the A1 Injection sub section.

The most significant vulnerabilities in JavaScript are cross-site scripting (XSS) and Document Object Model, DOM-based XSS.

Detection of DOM-based XSS can be challenging. This is cause by the following reasons.

• JavaScript is often obfuscated to protect intellectual property.

• JavaScript is often compressed out of concerned for bandwidth.

In both of these cases it is strongly recommended the code review be able to review the JavaScript before it has been obfuscated and or compressed. This is a huge point of contention with QA software professionals because you are reviewing code that is not in its production state.

Another aspect that makes code review of JavaScript challenging is its reliance of large frameworks such as Microsoft .NET and Java Server Faces and the use of JavaScript frameworks, such as JQuery, Knockout, Angular, Backbone. These frameworks aggravate the problem because the code can only be fully analyzed given the source code of the framework itself. These frameworks are usually several orders of magnitude larger then the code the code reviewer needs to review.

Because of time and money most companies simple accept that these frameworks are secure or the risks are low and acceptable to the organization.

Because of these challenges we recommend a hybrid analysis for JavaScript. Manual source to sink validation when necessary, static analysis with black-box testing and taint testing.

First use a static analysis. Code Reviewer and the organization needs to understand that because of event-driven behaviors, complex dependencies between HTML DOM and JavaScript code, and asynchronous communication with the server side static analysis will always fall short and may show both positive, false, false –positive, and positive-false findings.

Black-box traditional methods detection of reflected or stored XSS needs to be preformed. However this approach will not work for DOM-based XSS vulnerabilities.

Taint analysis needs to be incorporated into static analysis engine. Taint Analysis attempts to identify variables that have been ‘tainted’ with user controllable input and traces them to possible vulnerable functions also known as a ‘sink’. If the tainted variable gets passed to a sink without first being sanitized it is flagged as vulnerability.

Second the code reviewer needs to be certain the code was tested with JavaScript was turned off to make sure all client sided data validation was also validated on the server side.

Code examples of JavaScript vulnerabilities.

```
<html>
<script type="text/javascript">
var pos=document.URL.indexOf("name=")+5;
document.write( document.URL.substring(pos,document.URL.length));
</script>
<html>
```

Explanation: An attacker can send a link such as
"http://hostname/welcome.html#name=<script>alert(1)</script>" to the victim resulting in the victim's browser executing the injected client-side code.

```
var url = document.location.url;
var loginIdx = url.indexOf('login');
var loginSuffix = url.substring(loginIdx);
url = 'http://mySite/html/sso/' + loginSuffix;
document.location.url = url;
```

Line 5 may be a false-positive and prove to be safe code or it may be open to "Open redirect attack" with taint analysis the static analysis should be able to correctly identified if this vulnerability exists. If static analysis relies only on black-box component this code will have flagged as vulnerable requiring the code reviewer to do a complete source to sink review.

Additional examples and potential security risks

Source: document.url

Sink: document.write()

Results: document.write("<script>malicious code</script>

Cybercriminal may controlled the following DOM elements including…
document.url,document.location,document.referrer,window.location

Source: document.location

Sink: windon.location.href

Results: windon.location.href = http://www.BadGuysSite; - Client code open redirect.

Source: document.url

Storage: windows.localstorage.name

Sink: elem.innerHTML

Results: elem.innerHTML = <value> =Stored DOM-based Cross-site Scripting

eval() is prone to security threats, and thus not recommended to be used.

Consider these points:

1. Code passed to the eval is executed with the privileges of the executer. So, if the code passed can be affected by some malicious intentions, it leads to running malicious code in a user's machine with your website's privileges.

2. A malicious code can understand the scope with which the code passed to the eval was called.

3. You also shouldn't use eval() or new Function() to parse JSON data.

The above if used may raise security threats. JavaScript when used to dynamically evaluate code will create a potential security risk.

```
eval('alert("Query String ' + unescape(document.location.search) + '");');
eval(untrusted string); Can lead to code injection or client-side open redirect.
```

JavaScripts "new function" also may create a potential security risk.

Three points of validity are required for JavaScript

1. Have all the logic server-side, JavaScript validation be turned off on the client

2. Check for all sorts of XSS DOM Attacks, never trust user data, know your source and sinks

3. Check for insecure JavaScript libraries and update them frequently.

References:

• http://docstore.mik.ua/orelly/web/jscript/ch20_04.html

• https://www.owasp.org/index.php/Static_Code_Analysis

• http://www.cs.tau.ac.il/~omertrip/fse11/paper.pdf

• http://www.jshint.com

2.1.12.JSON (JavaScript Object Notation)

JSON (JavaScript Object Notation)is an open standard format that uses easy to read text to transmit data between a server and web applications. JSON data can be used by a large number of programming Languages and is becoming the de-facto standard in replacing XML.

JSON main security concern is JSON text dynamically embedded in JavaScript, because of this injection is a very real vulnerability. The vulnerability in the program that may inadvertently to run a malicious script or store the malicious script to a database. This is a very real possibility when dealing with data retrieved from the Internet.

The code reviewer needs to make sure the JSON is not used with Javascript eval. Make sure JSON.parse(…) is used.

Var parsed_object = eval("(" + Jason_text + ")"); // Red flag for the code reviewer.

JSON.parse(text[, reviver]); .. // Much better then using javascript eval function.

Code reviewer should check to make sure the developer is not attempting to reject known bad patterns in text/string data, Using regex or other devices is fraught with error and makes testing for correctness very hard. Allow only whitelisted alphanumeric keywords and carefully validated numbers.

Do not allow JSON data to construct dynamic HTML. Always us safe DOM features like innerText or CreateTextNode(…)

Object/Relational Mapping (ORM)

Object/Relation Mapping (ORM) facilitates the storage and retrieval of domain objects via HQL (Hibernate Query Language) or .NET Entity framework.

It is a very common misconception that ORM solutions, like hibernate are SQL Injection proof. They are not. ORM's allow the use of "native SQL". Thru proprietary query language, called HQL is prone to SQL Injection and the later is prone to HQL (or ORM) injection. Linq is not SQL and because of that is not prone to SQL injection. However using excutequery or excutecommand via linq causes the program not to use linq protection mechanism and is vulnerability to SQL injection.

Bad Java Code Examples

```
List results = session.createQuery("from Items as item where item.id = " +
currentItem.getId()).list();
```

NHibernate is the same as Hibernate except it is an ORM solution for Microsoft .NET platform. NHibernate is also vulnerable to SQL injection if used my dynamic queries.

Bad .NET Code Example

```
string userName = ctx.GetAuthenticatedUserName();
 String query = "SELECT * FROM Items WHERE owner = '"
 + userName + "' AND itemname = '"
 + ItemName.Text + "'";
 List items = sess.CreateSQLQuery(query).List()
```

Code Reviewer Action

Code reviewer needs to make sure any data used in an HQL query uses HQL parameterized queries so that it would be used as data and not as code.

Content Security Policy (CSP).

Is a W3C specification offering the possibility to instruct the client browser from which location and/or which type of resources are allowed to be loaded. To define a loading behavior, the CSP specification use "directive" where a directive defines a loading behavior for a target resource type. CSP helps to detect and mitigate certain types of attacks, including Cross Site Scripting (XSS) and data injection attacks. These attacks are used for everything from data theft to site defacement or distribution of malware

Directives can be specified using HTTP response header (a server may send more than one CSP HTTP header field with a given resource representation and a server may send different CSP header field values with different representations of the same resource or with different resources) or HTML Meta tag, the HTTP headers below are defined by the specs:

- Content-Security-Policy : Defined by W3C Specs as standard header, used by Chrome version 25 and later, Firefox version 23 and later, Opera version 19 and later.
- X-Content-Security-Policy : Used by Firefox until version 23, and Internet Explorer version 10 (which partially implements Content Security Policy).
- X-WebKit-CSP : Used by Chrome until version 25

Risk

- The risk with CSP can have 2 main sources:
- Policies misconfiguration,
- Too permissive policies.

What to Review

Code reviewer needs to understand what content security policies were required by application design and how these policies are tested to ensure they are in use by the application.

Useful security-related HTTP headers

In most architectures these headers can be set in web servers configuration ([http://httpd.apache.org/docs/2.0/mod/mod_headers.html Apache], [http://technet.microsoft.com/pl-pl/library/cc753133(v=ws.10).aspx IIS]), without changing actual application's code. This offers significantly faster and cheaper method for at least partial mitigation of existing issues, and an additional layer of defense for new applications.

Header name	Description	Example
Strict-Transport-Security	HTTP Strict-Transport-Security (HSTS) enforces secure (HTTP over SSL/TLS) connections to the server. This reduces impact of bugs in web	Strict-Transport-Security: max-

https://tools.ietf.org/html/rfc6797	applications leaking session data through cookies and external links and defends against Man-in-the-middle attacks. HSTS also disables the ability for user's to ignore SSL negotiation warnings.	age=16070400; includeSubDomains
X-Frame-Options https://tools.ietf.org/html/draft-ietf-websec-x-frame-options-01 Frame-Options https://tools.ietf.org/html/draft-ietf-websec-frame-options-00	Provides Click jacking protection. Values: deny - no rendering within a frame, sameorigin - no rendering if origin mismatch, allow-from: DOMAIN - allow rendering if framed by frame loaded from DOMAIN	X-Frame-Options: deny
X-XSS-Protection [http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-iv-the-xss-filter.aspx X-XSS-Protection]	This header enables the Cross-site scripting (XSS) filter built into most recent web browsers. It's usually enabled by default anyway, so the role of this header is to re-enable the filter for this particular website if it was disabled by the user. This header is supported in IE 8+, and in Chrome (not sure which versions). The anti-XSS filter was added in Chrome 4. Its unknown if that version honored this header.	X-XSS-Protection: 1; mode=block
X-Content-Type-Options https://blogs.msdn.microsoft.com/ie/2008/09/02/ie	The only defined value, "nosniff", prevents Internet Explorer and Google Chrome from MIME-sniffing a response away from the declared content-type. This also applies to Google Chrome, when downloading extensions. This reduces exposure to drive-by download attacks and sites serving user uploaded content that, by clever	X-Content-Type-Options: nosniff

8-security-part-vi-beta-2-update/	naming, could be treated by MSIE as executable or dynamic HTML files.	
Content-Security-Policy, X-Content-Security-policy,X-WebKit-CSP https://www.w3.org/TR/CSP/	Content Security Policy requires careful tuning and precise definition of the policy. If enabled, CSP has significant impact on the way browser renders pages (e.g., inline JavaScript disabled by default and must be explicitly allowed in policy). CSP prevents a wide range of attacks, including Cross-site scripting and other cross-site injections.	Content-Security-Policy: default-src 'self'
Content-Security-Policy-Report_Only https://www.w3.org/TR/CSP/	Like Content-Security-Policy, but only reports. Useful during implementation, tuning and testing efforts.	Content-Security-Policy-Report-Only: default-src 'self'; report-uri http://loghost.example.com/reports.jsp

References :

Apache: http://httpd.apache.org/docs/2.0/mod/mod_headers.html

IIS : http://technet.microsoft.com/pl-pl/library/cc753133(v=ws.10).aspx

Input Validation

Input validation is one of the most effective technical controls for application security. It can mitigate numerous vulnerabilities including cross-site scripting, various forms of injection, and some buffer overflows. Input validation is more than checking form field values.

All data from users needs to be considered untrusted. Remember one of the top rules of secure coding is “Don’t trust user input”. Always validate user data with the full knowledge of what your application is trying to accomplish.

Regular expressions can be used to validate user input, but the more complicated the regular express are the more chance it is not full proof and has errors for corner cases. Regular expressions are also very hard fro QA to test. Regular expressions may also make it hard for the code reviewer to do a good review of the regular expressions.

Data Validation

All external input to the system should undergo input validation. The validation rules are defined by the business requirements for the application. If possible, an exact match

validator should be implemented. Exact match only permits data that conforms to an expected value. A "Known good" approach (white-list), which is a little weaker, but more flexible, is common. Known good only permits characters/ASCII ranges defined within a white-list. Such a range is defined by the business requirements of the input field. The other approaches to data validation are "known bad," which is a black list of "bad characters". This approach is not future proof and would need maintenance. "Encode bad" would be very weak, as it would simply encode characters considered "bad" to a format, which should not affect the functionality of the application.

Business Validation

Business validation is concerned with business logic. An understanding of the business logic is required prior to reviewing the code, which performs such logic. Business validation could be used to limit the value range or a transaction inputted by a user or reject input, which does not make too much business sense. Reviewing code for business validation can also include rounding errors or floating point issues which may give rise to issues such as integer overflows, which can dramatically damage the bottom line.

Canonicalization

Canonicalization is the process by which various equivalent forms of a name can be resolved to a single standard name, or the "canonical" name.

The most popular encodings are UTF-8, UTF-16, and so on (which are described in detail in RFC 2279). A single character, such as a period/full-stop (.), may be represented in many different ways: ASCII 2E, Unicode C0 AE, and many others.

With the myriad ways of encoding user input, a web application's filters can be easily circumvented if they're not carefully built.

Bad Example

```
public static void main(String[] args) {
     File x = new File("/cmd/" + args[1]);
     String absPath = x.getAbsolutePath();
 }
```

Good Example

```
 public static void main(String[] args) throws IOException {
     File x = new File("/cmd/" + args[1]);
     String canonicalPath = x.getCanonicalPath();
 }
```

.NET Request Validation

One solution is to use .Net "Request Validation". Using request validation is a good start on validating user data and is useful. The downside is too generic and not specific enough to meet all of our requirements to provide full trust of user data.

You can never use request validation for securing your application against cross-site scripting attacks.

The following example shows how to use a static method in the Uri class to determine whether the Uri provided by a user is valid.

```
var isValidUri = Uri.IsWellFormedUriString(passedUri, UriKind.Absolute);
```

However, to sufficiently verify the Uri, you should also check to make sure it specifies http or https. The following example uses instance methods to verify that the Uri is valid.

```
var uriToVerify = new Uri(passedUri);
var isValidUri = uriToVerify.IsWellFormedOriginalString();
var isValidScheme = uriToVerify.Scheme == "http" || uriToVerify.Scheme == "https";
```

Before rendering user input as HTML or including user input in a SQL query, encode the values to ensure malicious code is not included.

You can HTML encode the value in markup with the <%: %> syntax, as shown below.

```
<span><%: userInput %></span>
```

Or, in Razor syntax, you can HTML encode with @, as shown below.

```
<span>@userInput</span>
```

The next example shows how to HTML encode a value in code-behind.

```
var encodedInput = Server.HtmlEncode(userInput);
```

Managed Code and Non-Managed Code

- Both Java and .Net have the concept of managed and non-managed code. To offer some of these protections during the invocation of native code, do not declare a native method public. Instead, declare it private and expose the functionality through a public wrapper method. A wrapper can safely perform any necessary input validation prior to the invocation of the native method:

```
Java Sample code to call a Native Method with Data Validation in place
        public final class NativeMethodWrapper {
            private native void nativeOperation(byte[] data, int offset,
int len);


            public void doOperation(byte[] data, int offset, int len) {
                // copy mutable input
                data = data.clone();

                // validate input
                // Note offset+len would be subject to integer overflow.
                // For instance if offset = 1 and len = Integer.MAX_VALUE,
                //   then offset+len == Integer.MIN_VALUE which is lower
                //   than data.length.
                // Further,
                //   loops of the form
                //       for (int i=offset; i<offset+len; ++i) { ... }
                //   would not throw an exception or cause native code to
                //   crash.

                if (offset < 0 || len < 0 || offset > data.length - len) {
                      throw new IllegalArgumentException();
                }

                nativeOperation(data, offset, len);
            }
        }
```

Data validations checklist for the Code Reviewer.

- Ensure that a Data Validation mechanism is present.
- Ensure all input that can (and will) be modified by a malicious user such as HTTP headers, input fields, hidden fields, drop down lists, and other web components are properly validated.
- Ensure that the proper length checks on all input exist.
- Ensure that all fields, cookies, http headers/bodies, and form fields are validated.
- Ensure that the data is well formed and contains only known good chars if possible.
- Ensure that the data validation occurs on the server side.
- Examine where data validation occurs and if a centralized model or decentralized model is used.
- Ensure there are no backdoors in the data validation model.
- "Golden Rule: All external input, no matter what it is, will be examined and validated."

Resources:

http://msdn.microsoft.com/en-us/library/vstudio/system.uri

2.2. BROKEN AUTHENTICATION AND SESSION MANAGEMENT

2.2.1. Broken Authentication

Overview

Authentication is the ability to determine who has sent a message. Web applications and Web services both use authentication as the primary means of access control from log-ins via user id and passwords. This control is essential to the prevention of confidential files, data, or web pages from being access by hackers or users who do not have the necessary access control level.

Description

Authentication is important, as it is the gateway to the functionality you are wishing to protect. Once a user is authenticated their requests will be authorized to perform some level of interaction with your application that non-authenticated users will be barred from. You cannot control how users manage their authentication information or tokens, but you can ensure there is now way to perform application functions without proper authentication occurring.

There are many forms of authentication with passwords being the most common. Other forms include client certificates, biometrics, one time passwords over SMS or special devices, or authentication frameworks such as Open Authorization (OAUTH) or Single Sign On (SSO).

Typically authentication is done once, when the user logs into a website, and successful authentication results in a web session being setup for the user (see Session Management). Further (and stronger) authentication can be subsequently requested if the user attempts to perform a high risk function, for example a bank user could be asked to confirm an 6 digit number that was sent to their registered phone number before allowing money to be transferred.

Authentication is just as important within a companies firewall as outside it. Attackers should not be able to run free on a companies internal applications simply because they found a way in through a firewall. Also separation of privilege (or duties) means someone working in accounts should not be able to modify code in a repository, or application managers should not be able to edit the payroll spreadsheets.

What to Review

When reviewing code modules, which perform authentication functions, some common issues to look out for include:

• Ensure the login page is only available over TLS. Some sites leave the login page has HTTP, but make the form submission URL HTTPS so that the users username and password are encrypted when sent to the server. However if the login page is not secured, a risk

exists for a man-in-the-middle to modify the form submission URL to an HTTP URL, and when the user enters their username & password they are sent in the clear.

• Make sure your usernames/user-ids are case insensitive. Many sites use email addresses for usernames and email addresses are already case insensitive. Regardless, it would be very strange for user 'smith' and user 'Smith' to be different users. Could result in serious confusion.

• Ensure failure messages for invalid usernames or passwords do not leak information. If the error message indicates the username was valid, but the password was wrong, then attackers will know that username exists. If the password was wrong, do not indicate how it was wrong.

• Make sure that every character the user types in is actually included in the password.

• Do not log invalid passwords. Many times an e-mail address is used as the username, and those users will have a few passwords memorized but may forget which one they used on your web site. The first time they may use a password that in invalid for your site, but valid for many other sites that this user (identified by the username). If you log that username and password combination, and that log leaks out, this low level compromise on your site could negatively affect many other sites.

• Longer passwords provide a greater combination of characters and consequently make it more difficult for an attacker to guess. Minimum length of the passwords should be enforced by the application. Passwords shorter than 10 characters are considered to be weak ([1]). Passphrases should be encouraged. For more on password lengths see the OWASP Authentication Cheat Sheet.

• To prevent brute force attacks, implement temporary account lockouts or rate limit login responses. If a user fails to provide the correct username and password 5 times, then lock the account for X minutes, or implement logic where login responses take an extra 10 seconds. Be careful though, this could leak the fact that the username is valid to attackers continually trying random usernames, so as an extra measure, consider implementing the same logic for invalid usernames.

• For internal systems, consider forcing the users to change passwords after a set period of time, and store a reference (e.g. hash) of the last 5 or more passwords to ensure the user is not simply re-using their old password.

• Password complexity should be enforced by making users choose password strings that include various types of characters (e.g. upper- and lower-case letters, numbers, punctuation, etc.). Ideally, the application would indicate to the user as they type in their new password how much of the complexity policy their new password meets. For more on password complexity see the OWASP Authentication Cheat Sheet.

• It is common for an application to have a mechanism that provides a means for a user to gain access to their account in the event they forget their password. This is an example of web site functionality this is invoked by unauthenticated users (as they have not provided their password). Ensure interfaces like this are protected from misuse, if asking for password reminder results in an e-mail or SMS being sent to the registered user, do not allow the password reset feature to be used to spam the user by attackers constantly entering the

username into this form. Please see Forgot Password Cheat Sheet for details on this feature.

• It is critical for an application to store a password using the right cryptographic technique. Please see Password Storage Cheat Sheet for details on this feature.

When reviewing MVC .NET is is important to make sure the application transmits and received over a secure link. It is recommended to have all web pages not just login pages use SSL.

We need to protect session cookies, which are useful as users credentials.

```
public static void RegisterGlobalFilters(GlobalFilterCollection filters) {
'''filters.Add(new RequireHttpsAttribute());'''
}
```

In the global.asax file we can review the RegisterGlobalFilters method.

The attribute RequireHttpsAttribute() can be used to make sure the application runs over SSL/TLS

It is recommended that this be enabled for SSL/TLS sites.

• For high risk functions, e.g. banking transactions, user profile updates, etc., utilize multi-factor authentication (MFA). This also mitigates against CSRF and session hijacking attacks. MFA is using more than one authentication factor to logon or process a transaction:

- Something you know (account details or passwords)
- Something you have (tokens or mobile phones)
- Something you are (biometrics)

• If the client machine is in a controlled environment utilize SSL Client Authentication, also known as two-way SSL authentication, which consists of both browser and server sending their respective SSL certificates during the TLS handshake process. This provides stronger authentication as the server administrator can create and issue client certificates, allowing the server to only trust login requests from machines that have this client SSL certificate installed. Secure transmission of the client certificate is important.

References

• https://www.owasp.org/index.php/Authentication_Cheat_Sheet

• http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf

• http://www.codeproject.com/Articles/326574/An-Introduction-to-Mutual-SSL-Authentication

• https://cwe.mitre.org/data/definitions/287.html Improper Authentication

• OWASP ASVS requirements areas for Authentication (V2)

2.2.2. Forgot Password

Overview

If your web site needs to have user authentication then most likely it will require user name and password to authenticate user accesses. However as computer system have increased in complexity, so has authenticating users has also increased. As a result the code reviewer needs to be aware of the benefits and drawbacks of user authentication referred to as “Direct Authentication” pattern in this section. This section is going to emphasis design patterns for when users forget user id and or password and what the code reviewer needs to consider when reviewing how user id and passwords can be retrieved when forgotten by the user and how to do this in a secure manner.

General considerations

Notified user by (phone sms, email) an email where the user has to click a link in the email that takes them to your site and ask the user to enter a new password.

Ask user to enter login credentials they already have (Facebook, Twitter, Google, Microsoft Live, OpenID etc) to validate user before allowing user to change password.

Send notification to user to confirm register and or forgot password.

Send notifications that account information has been changed for registered email. Set appropriate time out value. I.e. If user does not respond to email within 48 hours then user will be frozen out of system until user re-affirms password change.

• The identity and shared secret/password must be transferred using encryption to provide data confidentiality. HTTPS should also be used but in itself should not be the only mechanism used for data confidentiality.

• A shared secret can never be stored in clear text format, even if only for a short time in a message queue.

• A shared secret must always be stored in hashed or encrypted format in a database.

• The organization storing the encrypted shared secret does not need the ability to view or decrypt users passwords. User password must never be sent back to a user.

• If the client must cache the username and password for presentation for subsequent calls to a Web service then a secure cache mechanism needs to be in place to protect user name and password.

• When reporting an invalid entry back to a user, the username and or password should no be identified as being invalid. User feed back/error message must consider both user name and password as one item “user credential”. I.e. “The username or password you entered is incorrect.”

2.2.3. CAPTCHA

Overview

CAPTCHA (an acronym for “Completely Automated Public Turing test to tell Computers and Humans Apart”.) is an access control technique.

CAPTCHA is used to prevent automated software from gaining access to webmail services like Gmail, Hotmail and Yahoo to create e-mail spam, automated postings to blogs, forums

and wikis for the purpose of promotion (commercial, and or political) or harassment and vandalism and automated account creation.

CAPTCHA's have proved useful and their use has been upheld in court. Circumventing CAPTCHA has been upheld in US Courts as a violation Digital Millennium Copyright Act anti-circumvention section 1201(a)(3) and European Directive 2001/29/EC.

General considerations

Code review of CAPTCHA's the reviewer needs to pay attention to the following rules to make sure the CAPTCHA is built with strong security principals.

• Do not allow the user to enter multiple guesses after an incorrect attempt.

• The software designer and code review need to understand the statics of guessing. I.e. One CAPTCHA design shows four (3 cats and 1 boat) pictures, User is requested to pick the picture where it is not in the same category of the other pictures. Automated software will have a success rate of 25% by always picking the first picture. Second depending on the fixed pool of CAPTCHA images over time an attacker can create a database of correct answers then gain 100% access.

• Consider using a key being passed to the server that uses a HMAC (Hash-based message authentication code) the answer.

Text base CAPTCHA's should adhere to the following security design principals...

1. Randomize the CAPTCHA length: Don't use a fixed length; it gives too much information to the attacker.

2. Randomize the character size: Make sure the attacker can't make educated guesses by using several font sizes / several fonts.

3. Wave the CAPTCHA: Waving the CAPTCHA increases the difficulty for the attacker.

4. Don't use a complex charset: Using a large charset does not improve significantly the CAPTCHA scheme's security and really hurts human accuracy.

5. Use anti-recognition techniques as a means of strengthening CAPTCHA security: Rotation, scaling and rotating some characters and using various font sizes will reduce the recognition efficiency and increase security by making character width less predictable.

6. Keep the line within the CAPTCHAs: Lines must cross only some of the CAPTCHA letters, so that it is impossible to tell whether it is a line or a character segment.

7. Use large lines: Using lines that are not as wide as the character segments gives an attacker a robust discriminator and makes the line anti-segmentation technique vulnerable to many attack techniques.

8. CAPTCHA does create issues for web sites that must be ADA (Americans with Disabilities Act of 1990) compliant. Code reviewer may need to be aware of web accessibilities and security to review the CAPTCHA implementation where web site is required to be ADA

complaint by law.

References

• UNITED STATES of AMERICA vs KENNETH LOWSON, KRISTOFER KIRSCH, LOEL STEVENSON Federal Indictment. February 23, 2010. Retrieved 2012-01-02.

• http://www.google.com/recaptcha/captcha

• http://www.ada.gov/anprm2010/web%20anprm_2010.htm

• Inaccessibility of CAPTCHA - Alternatives to Visual Turing Tests on the Web http://www.w3.org/TR/turingtest/

2.2.4. Out-of-Band Communication

Overview

The term 'out-of-band' is commonly used when an web application communicates with an end user over a channel separate to the HTTP request/responses conducted through the users' web browser. Common examples include text/SMS, phone calls, e-mail and regular mail.

Description

The main reason an application would wish to communicate with the end user via these separate channels is for security. A username and password combination could be sufficient authentication to allow a user to browse and use non-sensitive parts of a website, however more sensitive (or risky) functions could require a stronger form of authentication. A username and password could have been stolen through an infected computer, through social engineering, database leak or other attacks, meaning the web application cannot put too much in trust a web session providing the valid username and password combination is actually the intended user.

Examples of sensitive operations could include:

• Changing password

• Changing account details, such as e-mail address, home address, etc

• Transferring funds in a banking application

• Submitting, modifying or cancelling orders

In these cases many applications will communicate with you via a channel other than a browsing session. Many large on-line stores will send you confirmation e-mails when you change account details, or purchase items. This protects in the case that an attacker has the username and password, if they buy something, the legitimate user will get an e-mail and have a chance to cancel the order or alert the website that they did not modify the account.

When out-of-band techniques are performed for authentication it is termed two-factor

authentication. There are three ways to authenticate:

1. Something you know (e.g. password, passphrase, memorized PIN)

2. Something you have (e.g. mobile phone, cash card, RSA token)

3. Something you are (e.g. iris scan, fingerprint)

If a banking website allows users to initiate transactions online, it could use two-factor authentication by taking 1) the password used to log in and 2) sending an PIN number over SMS to the users registered phone, and then requiring the user enter the PIN before completing the transaction. This requires something the user knows (password) and has (phone to receive the PIN).

A 'chip-and-pin' banking card will use two-factor authentication, by requiring users to have the card with them (something they have) and also enter a PIN when performing a purchase (something they know). A 'chip-and-pin' card is not use within the PIN number, likewise knowing the PIN number is useless if you do not have the card.

What to Review

When reviewing code modules which perform out-of-band functions, some common issues to look out for include:

1. Recognize the risk of the system being abused. Attackers would like to flood someone with SMS messages from your site, or e-mails to random people. Ensure:

2. When possible, only authenticated users can access links that cause an out-of-band feature to be invoked (forgot password being an exception).

3. Rate limit the interface, thus users with infected machines, or hacked accounts, can't use it to flood out-of-band messages to a user.

4. Do not allow the feature to accept the destination from the user, only use registered phone numbers, e-mails, addresses.

5. For high risk sites (e.g. banking) the users phone number can be registered in person rather than via the web site.

6. Do not send any personal or authentication information in the out-of-band communication.

7. Ensure any PINs or passwords send over out-of-band channels have a short life-span and are random.

8. A consideration can be to prevent SMS messages being sent to the device currently conducting the

browsing session (i.e. user browsing on their iPhone, were the SMS is sent to). However this can be hard to enforce.

a. If possible give users the choice of band they wish to use. For banking sites Zitmo malware on mobile devices (see references) can intercept the SMS messages, however iOS devices have not been affected by this malware yet, so users could choose to have SMS PINs sent to their Apple devices, and when on Android they could use recorded voice messages to receive the PIN.

9. In a typical deployments specialized hardware/software separate from the web application will handle the out-of-band communication, including the creation of temporary PINs and possibly passwords. In this scenario there is no need to expose the PIN/password to your web application (increasing risk of exposure), instead the web application should query the specialized hardware/software with the PIN/password supplied by the end user, and receive a positive or negative response.

Many sectors including the banking sector have regulations requiring the use of two-factor authentication for certain types of transactions. In other cases two-factor authentication can reduce costs due to fraud and re-assure customers of the security of a website.

References

- https://www.owasp.org/index.php/Forgot_Password_Cheat_Sheet
- http://securelist.com/blog/virus-watch/57860/new-zitmo-for-android-and-blackberry/

2.2.5. Session Management

Overview

A web session is a sequence of network HTTP request and response transactions associated to the same user. Session management or state is needed by web applications that require the retaining of information or status about each user for the duration of multiple requests. Therefore, sessions provide the ability to establish variables – such as access rights and localization settings – which will apply to each and every interaction a user has with the web application for the duration of the session.

Description

Code reviewer needs to understand what session techniques the developers used, and how to spot vulnerabilities that may create potential security risks. Web applications can create sessions to keep track of anonymous users after the very first user request. An example would be maintaining the user language preference. Additionally, web applications will make use of sessions once the user has authenticated. This ensures the ability to identify the user on any subsequent requests as well as being able to apply security access controls, authorized access to the user private data, and to increase the usability of the application. Therefore, current web applications can provide session capabilities both pre and post authentication.

The session ID or token binds the user authentication credentials (in the form of a user session) to the user HTTP traffic and the appropriate access controls enforced by the web application. The complexity of these three components (authentication, session management, and access control) in modern web applications, plus the fact that its implementation and binding resides on the web developer's hands (as web development framework do not provide strict relationships between these modules), makes the implementation of a secure session management module very challenging.

The disclosure, capture, prediction, brute force, or fixation of the session ID will lead to session hijacking (or sidejacking) attacks, where an attacker is able to fully impersonate a

victim user in the web application. Attackers can perform two types of session hijacking attacks, targeted or generic. In a targeted attack, the attacker's goal is to impersonate a specific (or privileged) web application victim user. For generic attacks, the attacker's goal is to impersonate (or get access as) any valid or legitimate user in the web application.

With the goal of implementing secure session IDs, the generation of identifiers (IDs or tokens) must meet the following properties:

• The name used by the session ID should not be extremely descriptive nor offer unnecessary details about the purpose and meaning of the ID.

• It is recommended to change the default session ID name of the web development framework to a generic name, such as "id".

• The session ID length must be at least 128 bits (16 bytes) (The session ID value must provide at least 64 bits of entropy).

• The session ID content (or value) must be meaningless to prevent information disclosure attacks, where an attacker is able to decode the contents of the ID and extract details of the user, the session, or the inner workings of the web application.

It is recommended to create cryptographically strong session IDs through the usage of cryptographic hash functions such as SHA1 (160 bits).

What to Review

Require cookies when your application includes authentication. Code reviewer needs to understand what information is stored in the application cookies. Risk management is needed to address if sensitive information is stored in the cookie requiring SSL for the cookie

.NET ASPX web.config

```
<authentication mode="Forms">
 <forms loginUrl="member_login.aspx"
   cookieless="UseCookies"
   requireSSL="true"
   path="/MyApplication" />
</authentication>
```

Java web.xml

```
<session-config>
 <cookie-config>
  <secure>true</secure>
 </cookie-config>
</session-config>
```

php.ini

```
void session_set_cookie_params ( int $lifetime [, string $path [, string
$domain [, bool $secure = true [, bool $httponly = true ]]]] )
```

2.2.6. Session Expiration

In reviewing session handling code the reviewer needs to understand what expiration timeouts are needed by the web application or if default session timeout are being used. Insufficient session expiration by the web application increases the exposure of other session-based attacks, as for the attacker to be able to reuse a valid session ID and hijack the associated session, it must still be active.

Remember for secure coding one of our goals is to reduce the attack surface of our application.

.NET ASPX

ASPX the developer can change the default time out for a session. This code in the web.config file sets the timeout session to 15 minutes. The default timeout for an aspx session is 30 minutes.

```
<system.web>
   <sessionState
     mode=”InProc”
     cookieless=”true”
     timeout=”15” />
 </system.web>
```

Java

```
<session-config>
<session-timeout>1</session-timeout>
</session-config>
```

PHP

Does not have a session timeout mechanism. PHP developers will need to create their own custom session timeout.

2.2.7. Session Logout/Ending.

Web applications should provide mechanisms that allow security aware users to actively close their session once they have finished using the web application.

.NET ASPX Session.Abandon() method destroys all the objects stored in a Session object and releases their resources. If you do not call the Abandon method explicitly, the server destroys these objects when the session times out. You should use it when the user logs out. Session.Clear() Removes all keys and values from the session. Does not change session ID. Use this command if you if you don't want the user to relogin and reset all the session specific data.

2.2.8. Session Attacks

Generally three sorts of session attacks are possible:

• Session Hijacking: stealing someone's session-id, and using it to impersonate that user.

• Session Fixation: setting someone's session-id to a predefined value, and impersonating them using that known value

• Session Elevation: when the importance of a session is changed, but its ID is not.

Session Hijacking

• Mostly done via XSS attacks, mostly can be prevented by HTTP-Only session cookies (unless Javascript code requires access to them).

• It's generally a good idea for Javascript not to need access to session cookies, as preventing all flavors of XSS is usually the toughest part of hardening a system.

• Session-ids should be placed inside cookies, and not in URLs. URL information are stored in browser's history, and HTTP Referrers, and can be accessed by attackers.

• As cookies can be accessed by default from javascript and preventing all flavors of XSS is usually the toughest part of hardening a system, there is an attribute called "HTTPOnly", that forbids this access. The session cookie should has this attribute set. Anyway, as there is no need to access a session cookie from the client, you should get suspicious about client side code that depends on this access.

• Geographical location checking can help detect simple hijacking scenarios. Advanced hijackers use the same IP (or range) of the victim.

• An active session should be warned when it is accessed from another location.

• An active users should be warned when s/he has an active session somewhere else (if the policy allows multiple sessions for a single user).

Session Fixation

1. If the application sees a new session-id that is not present in the pool, it should be rejected and a new session-id should be advertised. This is the sole method to prevent fixation.

2. All the session-ids should be generated by the application, and then stored in a pool to be checked later for. Application is the sole authority for session generation.

Session Elevation

• Whenever a session is elevated (login, logout, certain authorization), it should be rolled.

• Many applications create sessions for visitors as well (and not just authenticated users). They should definitely roll the session on elevation, because the user expects the application to treat them securely after they login.

• When a down-elevation occurs, the session information regarding the higher level should be flushed.

• Sessions should be rolled when they are elevated. Rolling means that the session-id should be changed, and the session information should be transferred to the new id.

2.2.9. Server-Side Defenses for Session Management

.NET ASPX

Generating new session Id's helps prevent, session rolling, fixation, hijacking.

```
public class GuidSessionIDManager : SessionIDManager {
   public override string CreateSessionID(HttpContext context){
return Guid.NewGuid().ToString();
   }
   public override bool Validate(string id) {
try{
       Guid testGuid = new Guid(id);
       if (id == testGuid.ToString())
         return true;
     }catch(Exception e) { throw e }
     return false;
   }
 }
```

Java

```
request.getSession(false).invalidate();
//and then create a new session with
getSession(true) (getSession())
```

PHP.INI

```
session.use_trans_sid = 0
session.use_only_cookies
```

References

• https://wwww.owasp.org/index.php/SecureFlag

2.3. CROSS-SITE SCRIPTING (XSS)

2.3.1. Overview

What is Cross-Site Scripting (XSS)?

Cross-site scripting (XSS) is a type of coding vulnerability. It is usually found in web applications. XSS enables attackers to inject malicious into web pages viewed by other users. XSS may allow attackers to bypass access controls such as the same-origin policy may. This is one of the most common vulnerabilities found accordingly with OWASP Top 10. Symantec in its annual threat report found that XSS was the number two vulnerability found on web servers. The severity/risk of this vulnerability may range from a nuisance to a major security risk, depending on the sensitivity of the data handled by the vulnerable site and the nature of any security mitigation implemented by the site's organization.

Description

There are three types of XSS, Reflected XSS (Non-Persistent), Stored XSS(Persistent), and DOM based XSS. Each of these types has different means to deliver a malicious payload to the server. The important takeaway is the consequences are the same.

2.3.2. What to Review

Cross-site scripting vulnerabilities are difficult to identify and remove from a web application

Cross-site scripting flaws can be difficult to identify and remove from a web application. The best practice to search for flaws is to perform an intense code review and search for all places where user input through a HTTP request could possibly make its way into the HTML output.

Code reviewer needs to closely review.

1. That untrusted data is not transmitted in the same HTTP responses as HTML or JavaScript.

2. When data is transmitted from the server to the client, untrusted data must be properly encoded in JSON format and the HTTP response MUST have a Content-Type of application/json. Do not assume data from the server is safe. Best practice is to always check data.

3. When introduced into the DOM, untrusted data MUST be introduced using one of the following APIs:

a. Node.textContent

b. document.createTextNode

c. Element.setAttribute (second parameter only)

Code reviewer should also be aware of the HTML tags (such as <img src…>, <iframe…>, <bgsound src…> etc.) that can be used to transmit malicious JavaScript.

Web application vulnerability automated tools/scanners can help to find Cross-Site scripting flaws. They cannot find all this is way manual code reviews are important. Manual code reviews wont catch all either but a defense in depth approach is always the best approach based on your level of risk.

One such tool is OWASP Zed Attack Proxy(ZAP) is an easy to use integrated penetration-testing tool for finding vulnerabilities in web applications. ZAP provides automated scanners as well as a set of tools that allow you to find security vulnerabilities manually. It acts as a web proxy that you point your browser to so it can see the traffic going to a site and allows you to spider, scan, fuzz, and attack the application. There are other scanners available both open source and commercial.

Use Microsft's Anti-XSS library

Another level of to help prevent XSS is to use an Anti-XSS library.

Unfortunately, HtmlEncode or validation feature is not enough to deal with XSS, especially if the user input needs to be added to JavaScript code, tag attributes, XML or URL. In this case a good option is the Anti-XSS libray

.NET ASPX

1. On ASPX .NET pages code review should check to make sure web config file does not turn off page validation. <pages validateRequest="false" />

2. .NET framework 4.0 does not allow page validation to be turned off. Hence if the programmer wants to turn of page validation the developer will need to regress back to 2.0 validation mode. <httpRuntime requestValidationMode="2.0" />

3. Code reviewer needs to make sure page validation is never turned off on anywhere and if it is understandwhy and the risks it opens the organization to. <%@ Page Language="C#" ValidationRequest="false"

.NET MVC

When MVC web apps are exposed to malicious XSS code, they will throw an error like the following one:

To avoid this vulnerability, make sure the following code is included:

<%server.HtmlEncode(stringValue)%>

The HTMLEncode method applies HTML encoding to a specified string. This is useful as a quick method of encoding form data and other client request data before using it in your Web application. Encoding data converts potentially unsafe characters to their HTML-encoded equivalent.(MSDN,2013)

JavaScript and JavaScript Frameworks

Both Javascript and Javascript frameworks are now widely use in web applications today. This hinders the code reviewer in knowing what frameworks do a good job on preventing XSS flaws and which ones don't. Code reviewer should check to see what to see if any CVE exists for the framework being used and also check that the javascript framework is the latest stable version.

OWASP References

- OWASP XSS Prevention Cheat Sheet
- OWASP XSS Filter Evasion Cheat Sheet
- OWASP DOM based XSS Prevention Cheat Sheet
- Testing Guide: 1st 3 chapters on Data Validation Testing
- OWASP_Zed_Attack_Proxy_Project

External References

- https://www4.symantec.com/mktginfo/whitepaper/ISTR/21347932_GA-internet-security-threat-report-volume-20-2015-social_v2.pdf
- https://cwe.mitre.org/data/definitions/79.html
- http://webblaze.cs.berkeley.edu/papers/scriptgard.pdf
- http://html5sec.org
- https://cve.mitre.org

2.3.3. HTML Attribute Encoding

HTML attributes may contain untrusted data. It is important to determine if any ot the HTML attributes on a given page contains data from outside the trust boundary.

Some HTML attributes are considered safer than others such as

align, alink, alt, bgcolor, border, cellpadding, cellspacing, class, color, cols, colspan, coords, dir, face, height, hspace, ismap, lang, marginheight, marginwidth, multiple, nohref, noresize,

noshade, nowrap, ref, rel, rev, rows, rowspan, scrolling, shape, span, summary, tabindex, title, usemap, valign, value, vlink, vspace, width

When reviewing code for XSS we need to look for HTML attributes such as the following reviewing code for XSS we need to look for HTML attributes such as the following

<input type="text" name="fname" value="UNTRUSTED DATA">

Attacks may take the following format:

"><script>/* bad stuff */</script>

What is Attribute encoding?

HTML attribute encoding replaces a subset of characters that are important to prevent a string of characters from breaking the attribute of an HTML element.
We replace ", &, and < with ", &, and >.
This is because the nature of attributes, the data they contain, and how they are parsed and interpreted by a browser or HTML parser is different than how an HTML document and its elements are read; OWASP XSS Prevention Cheat Sheet. Except for alphanumeric characters, escape all characters with ASCII values less than 256 with the &#xHH; format (or a named entity if available) to prevent switching out of the attribute. The reason this rule is so broad is that developers frequently leave attributes unquoted. Properly quoted attributes can only be escaped with the corresponding quote. Unquoted attributes can be broken out of with many characters, including [space] % * + , - / ; < = > ^ and |.

Attribute encoding may be performed in a number of ways. Two resources are

1. HttpUtility.HtmlAttributeEncode
   http://msdn.microsoft.com/en-us/library/wdek0zbf.aspx
2. OWASP Java Encoder Project
   https://www.owasp.org/index.php/OWASP_Java_Encoder_Project

HTML Entity

HTML elements which contain user controlled data or data from untrusted sourced should be reviewed for contextual output encoding. In the case of HTML entities we need to help ensure HTML Entity encoding is performed:

Example HTML Entity containing untrusted data:

```
   HTML Body Context        <span>UNTRUSTED DATA</span>    OR
<body>...UNTRUSTED DATA </body>    OR    <div>UNTRUSTED DATA </div>
```

HTML Entity Encoding is required

```
   & --> &amp;    < --> &lt;    > --> &gt;    " --> &quot;    ' --> &#x27;
```

It is recommended to review where/if untrusted data is placed within entity objects. Searching the source code fro the following encoders may help establish if HTML entity encoding is being done in the application and in a consistent manner.

OWASP Java Encoder Project.
https://www.owasp.org/index.php/OWASP_Java_Encoder_Project

```
   <input type="text" name="data"
value="<%=Encode.forHtmlAttribute(dataValue) %>" />
```

OWASP ESAPI
http://code.google.com/p/owasp-esapi-java/source/browse/trunk/src/main/java/org/owasp/esapi/codecs/HTMLEntityCodec.java

```
String safe = ESAPI.encoder().encodeForHTML( request.getParameter( "input" ) );
```

JavaScript Parameters

Untrusted data, if being placed inside a JavaScript function/code requires validation. Invalidated data may break out of the data context and wind up being executed in the code context on a users browser.

Examples of exploitation points (sinks) that are worth reviewing for:

```
    <script>var currentValue='UNTRUSTED DATA';</script>
<script>someFunction('UNTRUSTED DATA');</script>      attack: ');/* BAD STUFF
*/
```

Potential solutions:

OWASP HTML Sanitizer Project
OWASP JSON Sanitizer Project

ESAPI JavaScript escaping can be call in this manner:

```
    String safe = ESAPI.encoder().encodeForJavaScript( request.getParameter(
"input" ) );
```

Please note there are some JavaScript functions that can never safely use untrusted data as input - even if javascript escaped!

For example:

```
 <script>     window.setInterval('...EVEN IF YOU ESCAPE UNTRUSTED DATA YOU
ARE XSSED HERE...');     </script>
```

eval()

```
 var txtField = "A1";     var txtUserInput = "'test@google.ie';alert(1);";
eval(   "document.forms[0]." + txtField + ".value =" + A1);
```

jquery

```
    var txtAlertMsg = "Hello World: ";     var txtUserInput =
"test<script>alert(1)<\/script>";     $("#message").html(   txtAlertMsg +"" +
txtUserInput + "");
```

```
Safe usage (use text, not html)     $("#userInput").text (
"test<script>alert(1)<\/script>");<-- treat user input as text
```

Nested Contexts Best to avoid such nested contexts: an element attribute calling a JavaScript function etc. these contexts can really mess with your mind.

```
<div onclick="showError('<%=request.getParameter("errorxyz")%>')" >An error
occurred </div>          Here we have a HTML attribute(onClick) and within a
nested Javascript function call (showError).
```

When the browser processes this it will first HTML decode the contents of the onclick attribute. It will pass the results to the JavaScript Interpreter. So we have 2 contextx here...HTML and Javascript (2 browser parsers). We need to apply “layered” encoding in the RIGHT order:

1) JavaScript encode
2) HTML Attribute Encode so it "unwinds" properly and is not vulnerable.

<div onclick="showError   ('<%= Encoder.encodeForHtml(Encoder.encodeForJavaScript(request.getParameter("error")%>'))"   >An error occurred ....

2.7. INSECURE DIRECT OBJECT REFERENCE

2.7.1. Overview

Insecure Direct Object Reference is a commonplace vulnerability with web applications that provide varying levels of access or exposes an internal object to the user. Examples of what can be exposed are database records, URLs, files or allowing users to bypass web security controls by manipulating URLs.

The user may be authorized to access the web application, but not a specific object, such as a database record, specific file or even an URL. Potential threats can come from an authorized user of the web application who alters a parameter value that directly points to an object that the user isn’t authorized to access. If the application doesn’t verify the user for that specific object, it can result in an insecure direct object reference flaw.

2.7.2. Description

The source of the problem of this risk is based on the manipulation or updating of the data generated previously at server side.

2.7.3. What to Review

SQL Injection

An example of an attack making use of this vulnerability could be a web application where the user has already been validated. Now the user wants to view open his open invoices via another web page. The application passes the account number using the URL string. The an application uses unverified data in a SQL call that is accessing account information:

```
String query = "SELECT * FROM accts WHERE account = ?"; PreparedStatement
pstmt = connection.prepareStatement(query , ... );
pstmt.setString( 1, request.getParameter("acct"));
ResultSet results = pstmt.executeQuery();
```

The attacker simply modifies the 'acct' parameter in their browser to send whatever account number they want. If the application does not perform user verification, the attacker can access any user's account, instead of only the intended customer's account.

HTTP POST requests

A Cyber Security Analyst (Ibrahim Raafat) found an insecure direct object reference vulnerability with Yahoo! Suggestions by using Live HTTP Headers check the content in the POST request he could see:

```
prop=addressbook&fid=367443&crumb=Q4.PSLBfBe.&cid=1236547890&cmd=
delete_comment
```

Where parameter 'fid' is the topic id and 'cid' is the respective comment ID. While testing, he found changing the fid and cid parameter values allow him to delete other comments from the forum, that are actually posted by another user.

Next, he used the same method to test post deletion mechanism and found a similar vulnerability in that. A normal HTTP Header POST request of deleting a post is:

```
POST cmd=delete_item&crumb=SbWqLz.LDP0
```

He found that, appending the fid (topic id) variable to the URL allows him to delete the respective post of other users:

```
POST cmd=delete_item&crumb=SbWqLz.LDP0&fid=xxxxxxxx
```

After futher analysis he found an attacker could modify the parameters in HTTP POST requests to delete 1.5 million records entered by Yahoo users

Indirect Reference Maps

Moreover, the attackers may find out the internal naming conventions and infer the method names for operation functionality. For instance, if an application has URLs for retrieving detail information of an object like:

```
xyz.com/Customers/View/2148102445  or
xyz.com/Customers/ViewDetails.aspx?ID=2148102445
```

Attackers will try to use the following URLS to perform modification on the object:

```
xyz.com/Customers/Update/2148102445 or
xyz.com/Customers/Modify.aspx?ID=2148102445
```

```
Or xyz.com/Customers/admin
```

Also if web application is returning an object listing part of the directory path or object name an attacker can modify these.

Data Binding Technique

Another popular feature seen in most of the design frameworks (JSP/Structs, Spring) is data binding, where HTTP GET request parameters or HTTP POST variables get directly bound to the variables of the corresponding business/command object. Binding here means that the instance variables of such classes get automatically initialize with the request parameter values based on their names. Consider a sample design given below; observe that the business logic class binds the business object class binds the business object with the request parameters.

The flaw in such design is that the business objects may have variables that are not dependent on the request parameters. Such variables could be key variables like price, max limit, role, etc. having static values or dependent on some server side processing logic. A threat in such scenarios is that an attacker may supply additional parameters in request and try to bind values for unexposed variable of business object class. As illustrated in the figure below, the attacker sends an additional “price” parameter in the request and binds with the unexposed variable “price” in business object, thereby manipulating business logic.

Secure Design Recommendation:

• An important point to be noted here is that the business/form/command objects must have only those instance variables that are dependent on the user inputs.

• If additional variables are present those must not be vital ones like related to the business rule for the feature.

• In any case the application must accept only desired inputs from the user and the rest

must be rejected or left unbound. And initialization of unexposed of variables, if any must take place after the binding logic.

Review Criteria

Review the application design and check if it incorporates a data binding logic. In case it does, check if business objects/beans that get bound to the request parameters have unexposed variables that are meant to have static values. If such variables are initialized before the binding logic this attack will work successfully.

2.7.4. What the Code Reviewer needs to do:

Code reviewer needs to "map out all locations in the code being reviewed where user input is used to reference objects directly". These locations include where user input is used to access a database row, a file, application pages, etc. The reviewer needs to understand if modification of the input used to reference objects can result in the retrieval of objects the user is not authorized to view.

If untrusted input is used to access objects on the server side, then proper authorization checks must be employed to ensure data cannot be leaked. Proper input validation will also be required to ensure the untrusted input is properly understood and used by the server side code. Note that this authorization and input validation must be performed on the server side; client side code can be bypassed by the attacker.

Binding issues in MVC .NET

A.K.A Over-Posting A.K.A Mass assignments

In MVC framework, mass assignments are a mechanism that allows us to update our models with data coming in a request in HTTP form fields. As the data that needs to be updated comes in a collection of form fields, a user could send a request and modify other fields in the model that may not be in the form and the developer didn't intend to be updated.

Depending on the models you create, there might be sensitive data that you would not like to be modified. The vulnerability is exploited when a malicious user modifys a model's fields, which are not exposed to the user via the view, and the malicious user to change hidden model values adds additional model parameters.

```
public class user
   {
       public int ID { get; set; }  <- exposed via view
       public string Name { get; set; } <- exposed via view
       public bool isAdmin{ get; set; } <-hidden from view
   }
```

Corresponding view (HTML)

```
ID: <%= Html.TextBox("ID") %> <br>
   Name:  <%= Html.TextBox("Name")  %> <br>
                  <-- no isAdmin here!
```

The corresponding HTML for this model contain 2 fields: I' and Name.

If an attacker adds the isAdmin parameter to the form and submits they can change the model object above.

So a malicious attacker may change isAdmin=true

Recommendations:

*-1 Use a model which does not have values the user should not edit.

*-2 Use the bind method and whitelist attributes, which can be updated.

*-3 Use the controller. UpdateModel method to exclude certain attributes updates.

References

• OWASP Top 10-2007 on Insecure Dir Object References

• ESAPI Access Reference Map API

• ESAPI Access Control API (See isAuthorizedForData(), isAuthorizedForFile(), isAuthorizedForFunction())

• https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project

• https://cwe.mitre.org/data/definitions/639.html

• https://cwe.mitre.org/data/definitions/22.html

2.8. SECURITY MISCONFIGURATION

Many modern applications are developed on frameworks. These frameworks provide the developer less work to do, as the framework does much of the "housekeeping". The code developed will extend the functionality of the framework. It is here that the knowledge of a given framework, and language in which the application is implemented, is of paramount importance. Much of the transactional functionality may not be visible in the developer's code and handled in "parent" classes. The reviewer should be aware and knowledgeable of the underlying framework.

Web applications do not execute in isolation, they typically are deployed within an application server framework, running within an operating system on a physical host,

within a network.

Secure operating system configuration (also called hardening) is not typically within the scope of code review. For more information, see the Center for Internet Security operating system benchmarks.

Networks today consist of much more than routers and switches providing transport services. Filtering switches, VLANs (virtual LANs), firewalls, WAFs (Web Application Firewall), and various middle boxes (e.g. reverse proxies, intrusion detection and prevention systems) all provide critical security services when configured to do so. This is a big topic, but outside the scope of this web application code review guide. For a good summary, see the SANS (System Administration, Networking, and Security) Institute Critical Control 10: Secure Configurations for Network Devices such as Firewalls, Routers, and Switches.

Application server frameworks have many security related capabilities. These capabilities are enabled and configured in static configuration files, commonly in XML format, but may also be expressed as annotations within the code.

2.8.1. Apache Struts

In struts the struts-config.xml and the web.xml files are the core points to view the transactional functionality of an application. The struts-config.xml file contains the action mappings for each HTTP request while the web.xml file contains the deployment descriptor.

The struts framework has a validator engine, which relies on regular expressions to validate the input data. The beauty of the validator is that no code has to be written for each form bean. (Form bean is the Java object which received the data from the HTTP request). The validator is not enabled by default in struts. To enable the validator, a plug-in must be defined in the <plug-in> section of struts-config.xml. The property defined tells the struts framework where the custom validation rules are defined (validation.xml) and a definition of the actual rules themselves (validation-rules.xml).

Without a proper understanding of the struts framework, and by simply auditing the Java code, one would not see any validation being executed, and one does not see the relationship between the defined rules and the Java functions.

The action mappings define the action taken by the application upon receiving a request. Here, in **figure A5.0**, we can see that when the URL contains "/login" the LoginAction shall be called. From the action mappings we can see the transactions the application performs when external input is received.

Figure A5.0

```
Example of struts-config.xml file

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.0//EN"
```

```
        "http://jakarta.apache.org/struts/dtds/struts-config_1_0.dtd">
<struts-config>
<form-beans>
   <form-bean name="login" type="test.struts.LoginForm" />
</form-beans>
<global-forwards>
</global-forwards>
<action-mappings>
   <action
       path="/login"
       type="test.struts.LoginAction" >
           <forward name="valid" path="/jsp/MainMenu.jsp" />
           <forward name="invalid" path="/jsp/LoginView.jsp" />
   </action>
</action-mappings>
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
     <set-property property="pathnames"
     value="/test/WEB-INF/validator-rules.xml, /WEB-INF/validation.xml"/>
</plug-in>
</struts-config>
```

2.8.2. Java Enterprise Edition Declarative Configuration

Some security capabilities are accessible from within a Java program. Programmatic security is done within the web application using framework specific or standard Java Enterprise Edition (JEE) framework APIs. JEE has the JEE security model which, uses a role-based security model in which, access to application resources is granted based on the security role. The security role is a logical grouping of principals (authenticated entities, usually a user), and access is declared by specifying a security constraint on the role.

The constraints and roles are expressed as deployment descriptors expressed as XML elements. Different types of components use different formats, or schemas, for their deployment descriptors:

• Web components may use a web application deployment descriptor in the file web.xml

• Enterprise JavaBeans components may use an EJB deployment descriptor named META-INF/ejb-jar.xml The deployment descriptor can define resources (e.g. servlets accessible via a specific URL), which roles are authorized to access the resource, and how access is constrained (e.g. via GET but not POST).

The example web component descriptor in figure X (included in the "web.xml" file) defines a Catalog servlet, a "manager" role, a SalesInfo resource within the servlet accessible via GET and POST requests, and specifies that only users with "manager" role, using SSL and successfully using HTTP basic authentication should be granted access.

Figure A5.1

Example of creating login roles in Apache Struts

```
<?xml version=”1.0” encoding=”ISO-8859-1”?>
<web-app xmlns=”http://java.sun.com/xml/ns/j2ee”
  xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
  xsi:schemaLocation=”http://java.sun.com/xml/ns/j2ee
   http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd” version=?2.5?>

    <display-name>A Secure Application</display-name>
    <servlet>
      <servlet-name>catalog</servlet-name>
      <servlet-class>com.mycorp.CatalogServlet</servlet-class>
      <init-param>
        <param-name>catalog</param-name>
        <param-value>Spring</param-value>
      </init-param>

      <!-- Define Security Roles -->
      <security-role-ref>
        <role-name>MGR</role-name>
        <role-link>manager</role-link>
      </security-role-ref>
    </servlet>

    <security-role>
      <role-name>manager</role-name>
    </security-role>
    <servlet-mapping>
      <servlet-name>catalog</servlet-name>
      <url-pattern>/catalog/*</url-pattern>
    </servlet-mapping>

    <!-- Define A Security Constraint -->
    <security-constraint>

    <!-- Specify the Resources to be Protected -->
```

```
    <web-resource-collection>
      <web-resource-name>SalesInfo</web-resource-name>
      <url-pattern>/salesinfo/*</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
     </web-resource-collection>

    <!-- Specify which Users Can Access Protected Resources -->
    <auth-constraint>
      <role-name>manager</role-name>
    </auth-constraint>

   <!-- Specify Secure Transport using SSL (confidential guarantee) -->
    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
    </security-constraint>

    <!-- Specify HTTP Basic Authentication Method -->
    <login-config>
      <auth-method>BASIC</auth-method>
      <realm-name>file</realm-name>
    </login-config>
</web-app>
```

Security roles can also be declared for enterprise Java beans in the “ejb-jar.xml” file as seen in figure **A5.2**.

Figure A5.2

Creating security roles using Java beans

```
<ejb-jar>
    <assembly-descriptor>
        <security-role>
            <description>The single application role</description>
            <role-name>TheApplicationRole</role-name>
        </security-role>
    </assembly-descriptor>
</ejb-jar>
```

For beans, however, rather than specifying access to resources within servlets, access to bean methods is specified. The example in figure **A5.3** illustrates several types of method access constraints for several beans.

Figure A5.3

Controlling access to Java beans

```
<ejb-jar>
    <assembly-descriptor>
        <method-permission>
            <description>The employee and temp-employee roles may access any
                method of the EmployeeService bean </description>
            <role-name>employee</role-name>
            <role-name>temp-employee</role-name>
            <method>
                <ejb-name>EmployeeService</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <method-permission>
            <description>The employee role may access the findByPrimaryKey,
                getEmployeeInfo, and the updateEmployeeInfo(String) method of
                the AardvarkPayroll bean </description>
            <role-name>employee</role-name>
            <method>
                <ejb-name>AardvarkPayroll</ejb-name>
                <method-name>findByPrimaryKey</method-name>
            </method>
            <method>
                <ejb-name>AardvarkPayroll</ejb-name>
                <method-name>getEmployeeInfo</method-name>
            </method>
            <method>
                <ejb-name>AardvarkPayroll</ejb-name>
                <method-name>updateEmployeeInfo</method-name>
                <method-params>
                    <method-param>java.lang.String</method-param>
                </method-params>
```

```
            </method>
        </method-permission>
        <method-permission>
            <description>The admin role may access any method of the
                EmployeeServiceAdmin bean </description>
            <role-name>admin</role-name>
            <method>
                <ejb-name>EmployeeServiceAdmin</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <method-permission>
            <description>Any authenticated user may access any method of the
                EmployeeServiceHelp bean</description>
            <unchecked/>
            <method>
                <ejb-name>EmployeeServiceHelp</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <exclude-list>
            <description>No fireTheCTO methods of the EmployeeFiring bean may
be
                used in this deployment</description>
            <method>
                <ejb-name>EmployeeFiring</ejb-name>
                <method-name>fireTheCTO</method-name>
            </method>
        </exclude-list>
    </assembly-descriptor>
</ejb-jar>
```

See JBoss Security [1]

If XML deployment descriptors are used to secure the application, code review should include the “web.xml” and “ejb-jar.xml” files to ensure that access controls are properly applied to the correct roles, and authentication methods are as expected.

Java Annotations

JEE annotations for security are defined in the Java Docs [2]. The available annotations are:

- **@DeclareRoles**

- **@DenyAll -** no roles may invoke the method.
- **@PermitAll -** all roles may invoke the method.
- **@RolesAllowed -** roles permitted to invoke the method.
- **@RunAs -** dynamically run the method as a particular role.

For example the code in **figure A5.4** allows employees and managers to add movies to the persistent store, anyone to list movies, but only managers may delete movies.

Figure A5.4

Controlling access using Java annotations

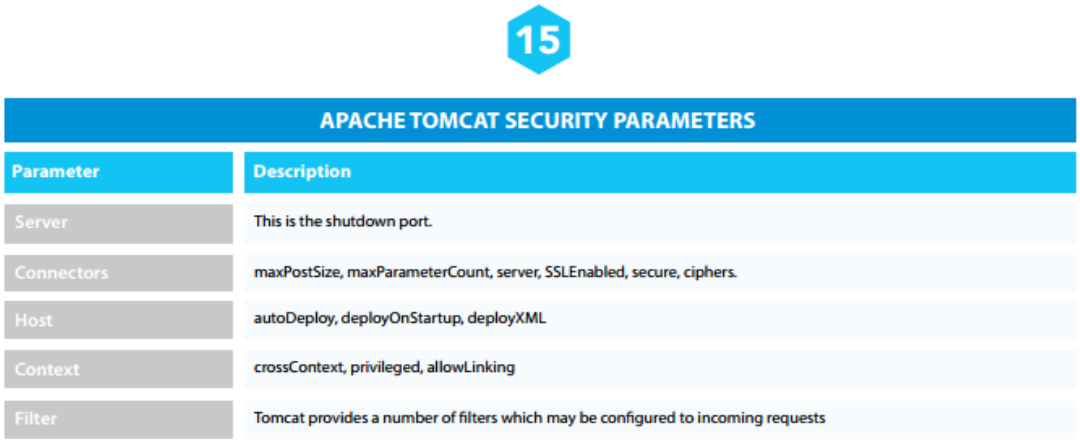
```
public class Movies {
    private EntityManager entityManager;
    @RolesAllowed({"Employee", "Manager"})
    public void addMovie(Movie movie) throws Exception {
        entityManager.persist(movie);
    }
    @RolesAllowed({"Manager"})
    public void deleteMovie(Movie movie) throws Exception {
        entityManager.remove(movie);
    }
    @PermitAll
    public List<Movie> getMovies() throws Exception {
        Query query = entityManager.createQuery("SELECT m from Movie as m");
        return query.getResultList();
    }
}
```

Code review should look for such annotations. If present, ensure they reflect the correct roles and permissions, and are consistent with any declared role permissions in the "ejb-jar.xml" file.

2.8.3. Framework Specific Configuration

Apache Tomcat

The server.xml[3] file should be reviewed to ensure security related parameters are configured as expected. The tomcat server.xml file defines many security related parameters.



Filters are especially powerful, and a code review should validate they are used unless there is a compelling reason not to. See Provided Filters[4] for detailed information. More guidelines on securely deploying Apache Tomcat can be found in the CIS Apache Tomcat [5].

Jetty

Jetty adds several security enhancements:

• Limiting form content

• Obfuscating passwords

The maximum form content size and number of form keys can be configured at server and web application level in the “jetty-web.xml” file.

Figure A5.5

```
Security configuration in Jetty

<Configure class=”org.eclipse.jetty.webapp.WebAppContext”>
  …
  <Set name=”maxFormContentSize”>200000</Set>
  <Set name=”maxFormKeys”>200</Set>
</Configure>

<configure class=”org.eclipse.jetty.server.Server”>
  ...
  <Call name=”setAttribute”>
    <Arg>org.eclipse.jetty.server.Request.maxFormContentSize</Arg>
```

```
    <Arg>100000</Arg>
   </Call>
  <Call name="setAttribute">
    <Arg>org.eclipse.jetty.server.Request.maxFormKeys</Arg>
    <Arg>2000</Arg>
   </Call>
</configure>
```

Jetty also supports the use of obfuscated passwords in jetty XML files where a plain text password is usually needed. **Figure A5.6** shows example code setting the password for a JDBC Datasource with obfuscation (the obfuscated password is generated by Jetty org.eclipse.jetty.util.security.Password utility).

Figure A5.6

Example Jetty code setting JDBC passwords

```
<New id="DSTest" class="org.eclipse.jetty.plus.jndi.Resource">
<Arg></Arg>
<Arg>jdbc/DSTest</Arg>
<Arg>
      <New class="com.jolbox.bonecp.BoneCPDataSource">
            <Set name="driverClass">com.mysql.jdbc.Driver</Set>
            <Set name="jdbcUrl">jdbc:mysql://localhost:3306/foo</Set>
            <Set name="username">dbuser</Set>
            <Set name="password">
            <Call class="org.eclipse.jetty.util.security.Password"
name="deobfuscate">
                  <Arg>OBF:1ri71v1r1v2n1ri71shq1ri71shs1ri71v1r1v2n1ri7</Arg>
            </Call>
            </Set>
            <Set name="minConnectionsPerPartition">5</Set>
            <Set name="maxConnectionsPerPartition">50</Set>
            <Set name="acquireIncrement">5</Set>
            <Set name="idleConnectionTestPeriod">30</Set>
      </New>
</Arg>
</New>
```

JBoss AS

JBoss Application Server, like Jetty, allows password obfuscation (called password masking in JBoss) in its XML configuration files. After using JBoss password utility to create password mask, replace any occurrence of a masked password in XML configuration files with the following annotation.

Figure A5.7

```
Setting passwords in JBoss

<annotation>@org.jboss.security.integration.password.Password
                    (securityDomain=MASK_NAME,methodName=setPROPERTY_NAME)
</annotation>
```

See Masking Passwords in XML Configuration in the JBoss Security Guide[1]

Oracle WebLogic

WebLogic server supports additional deployment descriptors in the "weblogic.xml" file as shown in **table16**.

16

WEBLOGIC SECURITY PARAMETERS	
Parameter	**Description**
externally-defined	Role to principal mappings are externally defined in WebLogic Admin Console
run-as-principal-name	Assign a principal to a role when running as that role
run-as-role-assignment	Contains the run-as-principal-name descriptor
security-permission	Contains security-permission-spec descriptor
Filter	Tomcat provides a number of filters which may be configured to incoming requests
security-permission-spec	Specify application permissions [6]
security-role-assignment	Explicitly assign principals to a role

More information on WebLogic additional deployment descriptors may be found at weblogic.xml Deployment Descriptors. **[7]**

For general guidelines on securing web applications running within WebLogic, see Programming WebLogic Security [8] and the NSA's BEA WebLogic Platform Security Guide. **[8]**

2.8.4. Programmatic Configuration: JEE

The JEE API for programmatic security consists of methods of the EJBContext interface and the HttpServletRequest interface. These methods allow components to make business-logic

decisions based on the security role of the caller or remote user (there are also methods to authenticate users, but that is outside the scope of secure deployment configuration).

The JEE APIs that interact with JEE security configuration include:

• getRemoteUser, which determines the user name with which the client authenticated

• isUserInRole, which determines whether a remote user is in a specific security role.

• getUserPrincipal, which determines the principal name of the current user and returns a java.security. Principal object

Use of these programmatic APIs should be reviewed to ensure consistency with the configuration. Specifically, the security-role-ref element should be declared in the deployment descriptor with a role-name subelement containing the role name to be passed to the isUserInRole method.

The code in **figure A5.8** demonstrates the use of programmatic security for the purposes of programmatic login and establishing identities and roles. This servlet does the following:

• displays information about the current user.

• prompts the user to log in.

• prints out the information again to demonstrate the effect of the login method.

• logs the user out.

• prints out the information again to demonstrate the effect of the logout method.

Figure A5.8

Example programmatic security for Java

```
package enterprise.programmatic_login;

import java.io.*;
import java.net.*;
import javax.annotation.security.DeclareRoles;
import javax.servlet.*;
import javax.servlet.http.*;

@DeclareRoles("javaee6user")
public class LoginServlet extends HttpServlet {

    /**
     * Processes requests for both HTTP GET and POST methods.
     * @param request servlet request
```

```
     * @param response servlet response
     */
    protected void processRequest(HttpServletRequest request,
                HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            String userName = request.getParameter("txtUserName");
            String password = request.getParameter("txtPassword");

            out.println("Before Login" + "<br><br>");
            out.println("IsUserInRole?.."
                        + request.isUserInRole("javaee6user")+"<br>");
            out.println("getRemoteUser?.." + request.getRemoteUser()+"<br>");
            out.println("getUserPrincipal?.."
                        + request.getUserPrincipal()+"<br>");
            out.println("getAuthType?.." + request.getAuthType()+"<br><br>");

            try {
                request.login(userName, password);
            } catch(ServletException ex) {
                out.println("Login Failed with a ServletException.."
                    + ex.getMessage());
                return;
            }
            out.println("After Login..."+"<br><br>");
            out.println("IsUserInRole?.."
                        + request.isUserInRole("javaee6user")+"<br>");
            out.println("getRemoteUser?.." + request.getRemoteUser()+"<br>");
            out.println("getUserPrincipal?.."
                        + request.getUserPrincipal()+"<br>");
            out.println("getAuthType?.." + request.getAuthType()+"<br><br>");

            request.logout();
            out.println("After Logout..."+"<br><br>");
            out.println("IsUserInRole?.."
                        + request.isUserInRole("javaee6user")+"<br>");
```

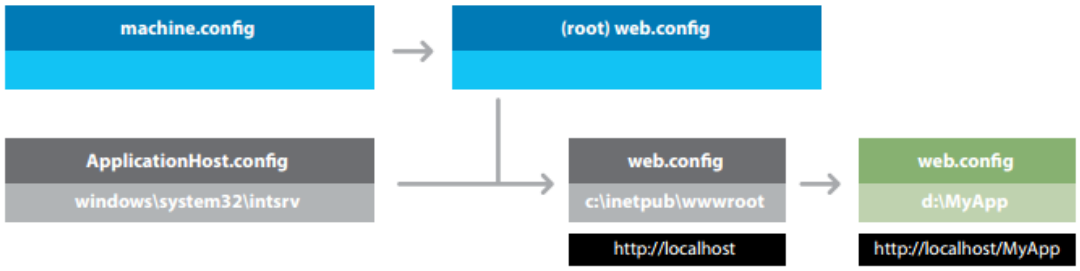
```
            out.println("getRemoteUser?.." + request.getRemoteUser()+"<br>");
            out.println("getUserPrincipal?.."
                        + request.getUserPrincipal()+"<br>");
            out.println("getAuthType?.." + request.getAuthType()+"<br>");
        } finally {
            out.close();
        }
    }
    ...
}
```

More detailed information can be found in the Java EE Tutorial: Using Programmatic Security with Web Applications **[10]**

2.8.5. Microsoft IIS

ASP.NET / IIS applications use an optional XML-based configuration file, named web.config, to maintain application configuration settings. This covers issues such as authentication, authorization, error pages, HTTP settings, debug settings, web service settings, etc. Without knowledge of these files, a transactional analysis would be very difficult and not accurate.

In IIS 7 there is a new configuration system, which affects the hierarchy level and how one file can inherit from another. The following figure resumes how this work and the location of each file (Aguilar, 2006)



It is possible to provide a file web.config at the root of the virtual directory for a web application. If the file is absent, the default configuration settings in machine.config will be used. If the file is present, any settings in web.config will override the default settings.

Figure A5.9

```
Example web.config file

<authentication mode="Forms">
   <forms name="name"
      loginUrl="url"
```

```
        protection="Encryption"
        timeout="30" path="/"
        requireSSL="true|"
        slidingExpiration="false">
       <credentials passwordFormat="Clear">
          <user name="username" password="password"/>
       </credentials>
   </forms>
   <passport redirectUrl="internal"/>
</authentication>
```

Many of the important security settings are not set in the code, but in the framework configuration files. Knowledge of the framework is of paramount importance when reviewing framework-based applications. Some examples of framework specific parameters in the web.config file are shown in **table 17**.

17

PARAMETERS IN THE WEB.CONFIG FILE

Parameter	Description
authentication mode	The default authentication mode is ASP.NET forms-based authentication.
loginUrl	Specifies the URL where the request is redirected for login if no valid authentication cookie is found.
protection	Specifies that the cookie is encrypted using 3DES or DES but DV is not performed on the cookie. Beware of plaintext attacks.
timeout	Cookie expiry time in minutes.

2.8.6. Framework Specific Configuration: Microsoft IIS

Security features can be configured in IIS using the Web.config (application level) or ApplicationHost.config (server level) file, in the <system.webServer><security> section. The types of features that may be configured include:

- Permitted authentication methods
- Authorization rules
- Request filters and limits
- Use of SSL
- Source IP address filtering
- Error handling

The Web.config and ApplicationHost.config files should be included in code review. The

<system.webServer><security> sections should be reviewed to ensure all security configuration is as expected.

For guidelines on securing the overall configuration of Microsoft IIS [11], see the IIS supports basic, client certificate, digest, IIS client certificate, and Windows authentication methods. They are configured in the <system.webServer><security><authentication> section.

The example in **figure A5.10** disables anonymous authentication for a site named MySite, then enables both basic authentication and windows authentication for the site.

Figure A5.10

IIS authentication configuration

```
<location path="MySite">
   <system.webServer>
      <security>
         <authentication>
            <anonymousAuthentication enabled="false" />
            <basicAuthentication enabled="true" defaultLogonDomain="MySite"
/>
            <windowsAuthentication enabled="true" />
          </authentication>
      </security>
   </system.webServer>
</location>
```

IIS authorization configuration allows specification of users access to the site or server and is configured in the **<system.webServer><security><authorization>** section.

The configuration in **figure A5.11** removes the default IIS authorization settings, which allows all users access to Web site or application content, and then configures an authorization rule that allows only users with administrator privileges to access the content.

Figure A5.11

Authorization configuration in IIS

```
<configuration>
   <system.webServer>
      <security>
         <authorization>
            <remove users="*" roles="" verbs="" />
```

```
            <add accessType="Allow" users="" roles="Administrators" />
         </authorization>
      </security>
   </system.webServer>
</configuration>
```

IIS supports filtering, including enforcing limits, on incoming HTTP requests. **Table 18** shows many of the IIS security parameters that can be set.

IIS SECURITY PARAMETERS	
Parameter	**Function**
denyUrlSequences	A list of prohibited URL patterns
fileExtensions	Allowed or prohibited file extensions
hiddenSegments	URLs that cannot be browsed
requestLimits	URL, content, query string, and HTTP header length limits
verbs	Allowed or prohibited verbs
alwaysAllowedUrls	URLs always permitted
alwaysAllowedQueryStrings	Query strings always allowed
denyQueryStringSequences	Prohibited query strings
filteringRules	Custom filtering rules

These parameters are configured in the <system.webServer><security><requestFiltering> section. The example in **figure A5.12**:

• Denies access to two URL sequences. The first sequence prevents directory transversal and the second sequence prevents access to alternate data streams.

• Sets the maximum length for a URL to 2KB and the maximum length for a query string to 1KB.

• Denies access to unlisted file name extensions and unlisted HTTP verbs.

Figure A5.12

```
Examples IIS security configuration

<configuration>
   <system.webServer>
      <security>
         <requestFiltering>
```

```
            <denyUrlSequences>
               <add sequence=”..” />
               <add sequence=”:” />
            </denyUrlSequences>
            <fileExtensions allowUnlisted=”false” />
            <requestLimits maxUrl=”2048” maxQueryString=”1024” />
            <verbs allowUnlisted=”false” />
         </requestFiltering>
      </security>
   </system.webServer>
</configuration>
```

IIS allows specifying whether SSL is supported, is required, whether client authentication is supported or required, and cipher strength. It is configured in the <system.webServer><security><access> section. The example in **figure A5.13** specifies SSL as required for all connections to the site MySite.

Figure A5.13

IIS SSL configuration

```
<location path=”MySite”>
   <system.webServer>
      <security>
         <access sslFlags=”ssl”>
      </security>
   </system.webServer>
</location>
```

IIS allows restrictions on source IP addresses or DNS names. It is configured in the **<system.webServer><security><ipSecurity>** section as shown in figure X where the example configuration denies access to the IP address **192.168.100.1** and to the entire **169.254.0.0** network:

Figure A5.14

IIS network security configuration

```
<location path=”Default Web Site”>
   <system.webServer>
      <security>
```

```
        <ipSecurity>
           <add ipAddress=”192.168.100.1” />
           <add ipAddress=”169.254.0.0” subnetMask=”255.255.0.0” />
        </ipSecurity>
      </security>
   </system.webServer>
</location
```

Detailed information on IIS security configuration can be found at IIS Security Configuration. Specific security feature configuration information can be found at Authentication, Authorization, SSL, Source IP, Request Filtering, and Custom Request Filtering[12].

2.8.7. Programmatic Configuration: Microsoft IIS

Microsoft IIS security configuration can also be programmatically set from various languages:

- appcmd.exe set config
- C#
- Visual Basic
- JavaScript

For example, disabling anonymous authentication for a site named MySite, then enabling both basic authentication and windows authentication for the site (as done via configuration in the section above) can be accomplished from the command line using the commands in figure **A5.15**.

Figure A5.15

```
Controlling Windows authentication from the command line

appcmd.exe set config “MySite” -
section:system.webServer/security/authentication
      /anonymousAuthentication /enabled:”False” /commit:apphost
appcmd.exe set config “MySite” -
section:system.webServer/security/authentication
     /basicAuthentication /enabled:”True” /commit:apphost
appcmd.exe set config “MySite” -
section:system.webServer/security/authentication
    /windowsAuthentication /enabled:”True” /commit:apphost
```

Alternatively the same authentication setup can be coded programmatically as in **figure A5.16**.

Figure A5.16

Controlling Windows authentication through code

```
using System;
using System.Text;
using Microsoft.Web.Administration;
internal static class Sample {
   private static void Main() {

      using(ServerManager serverManager = new ServerManager()) {
         Configuration config =
serverManager.GetApplicationHostConfiguration();

         ConfigurationSection anonymousAuthenticationSection =
           config.GetSection("system.webServer/security/authentication
                             /anonymousAuthentication", "MySite");
         anonymousAuthenticationSection["enabled"] = false;

         ConfigurationSection basicAuthenticationSection =
           config.GetSection("system.webServer/security/authentication
                              /basicAuthentication", "MySite");
         basicAuthenticationSection["enabled"] = true;
         ConfigurationSection windowsAuthenticationSection =
           config.GetSection("system.webServer/security/authentication
                            /windowsAuthentication", "MySite");
         windowsAuthenticationSection["enabled"] = true;

         serverManager.CommitChanges();
      }
   }
}
```

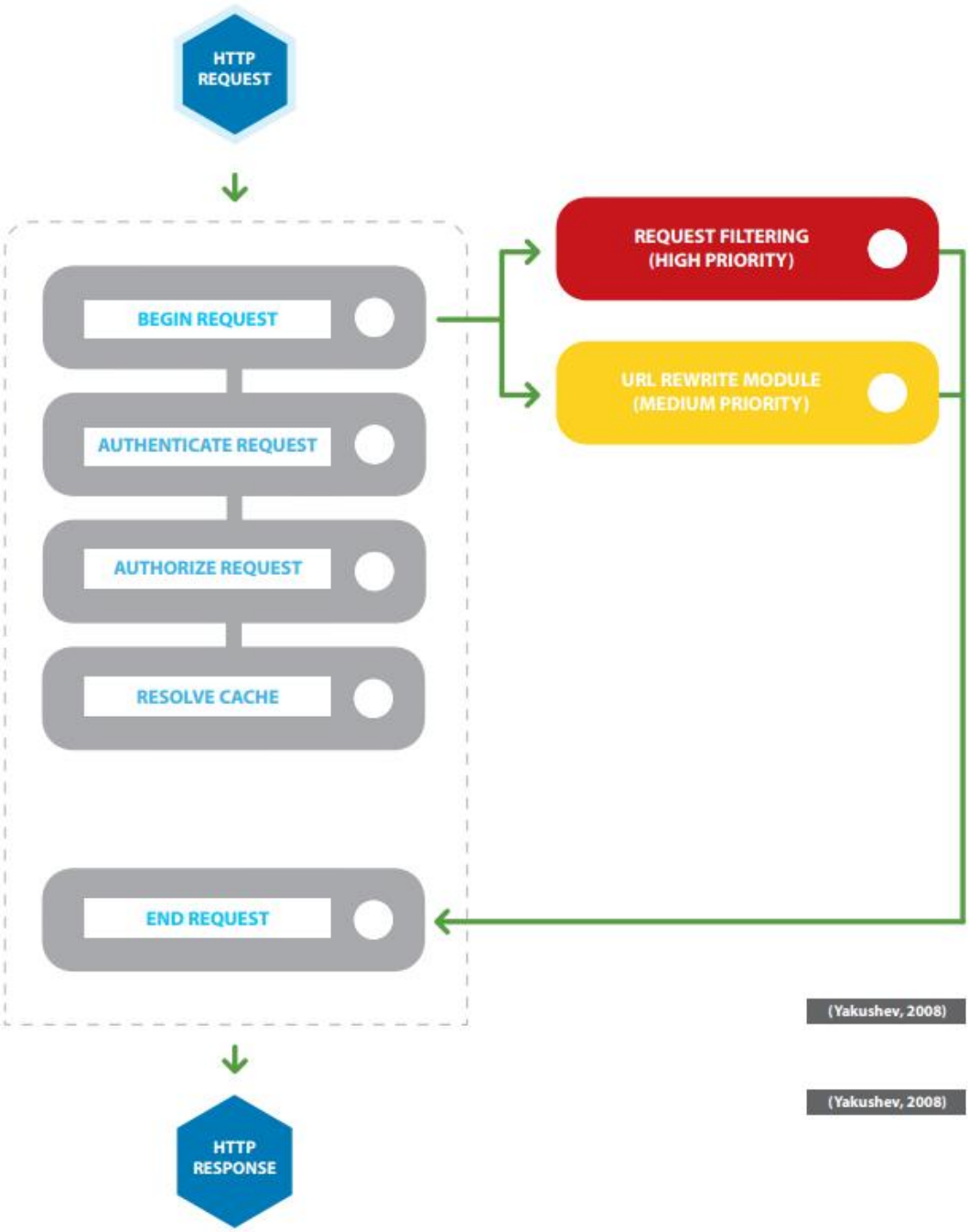
When reviewing source code, special attention should be paid to configuration updates in security sections.

2.8.8. Further IIS Configurations

Filtering Requests and URL Rewriting

Request Filtering was introduced in IIS7 and it has replaced the functionality UrlScan add-on for IIS 6.0. This built-in security feature allows to filter undesired URL request but it is

also possible to configure different kinds of filtering. To begin with, it is important to understand how the IIS pipeline works when a request is done. The following diagram shows the order in these modules



Request filtering can be setup through the IIS interface or on the web.config file. Example:

```
<configuration>
   <system.webServer>
      <security>
         <requestFiltering>
            <denyUrlSequences>
               <add sequence=”..” />
               <add sequence=”:” />
            </denyUrlSequences>
            <fileExtensions allowUnlisted=”false” />
            <requestLimits maxUrl=”2048” maxQueryString=”1024” />
            <verbs allowUnlisted=”false” />
         </requestFiltering>
      </security>
   </system.webServer>
</configuration>
```

(Yakushev, 2008)

This can also be done through the application code, for example:

```
using System;
 using System.Text;
 using Microsoft.Web.Administration;
 internal static class Sample
 {
    private static void Main()
    {
       using (ServerManager serverManager = new ServerManager())
       {
          Configuration config = serverManager.GetWebConfiguration(“Default
Web Site”);
          ConfigurationSection requestFilteringSection =
config.GetSection(“system.webServer/security
          /requestFiltering”);
         ConfigurationElementCollection denyUrlSequencesCollection =
         requestFilteringSection.GetCollection(“denyUrlSequences”);
         ConfigurationElement addElement =
denyUrlSequencesCollection.CreateElement(“add”);
         addElement[“sequence”] = @”..”;
```

```
            denyUrlSequencesCollection.Add(addElement);
            ConfigurationElement addElement1 =
denyUrlSequencesCollection.CreateElement("add");
            addElement1["sequence"] = @":";
            denyUrlSequencesCollection.Add(addElement1);
            ConfigurationElement addElement2 =
denyUrlSequencesCollection.CreateElement("add");
            addElement2["sequence"] = @"\";
            denyUrlSequencesCollection.Add(addElement2);
            serverManager.CommitChanges();
        }
    }
}
```

(Yakushev, 2008)

Filtering Double - Encoded Requests

This attack technique consists of encoding user request parameters twice in hexadecimal format in order to bypass security controls or cause unexpected behavior from the application. It's possible because the webserver accepts and processes client requests in many encoded forms.

By using double encoding it's possible to bypass security filters that only decode user input once. The second decoding process is executed by the backend platform or modules that properly handle encoded data, but don't have the corresponding security checks in place.

Attackers can inject double encoding in pathnames or query strings to bypass the authentication schema and security filters in use by the web application.

There are some common character that are used in Web applications attacks. For example, Path Traversal attacks use "../" (dot-dot-slash) , while XSS attacks use "<" and ">" characters. These characters give a hexadecimal representation that differs from normal data.

For example, "../" (dot-dot-slash) characters represent %2E%2E%2f in hexadecimal representation. When the % symbol is encoded again, its representation in hexadecimal code is %25. The result from the double encoding process "../"(dot-dot-slash) would be %252E%252E%252F:

- The hexadecimal encoding of "../" represents "%2E%2E%2f"
- Then encoding the "%" represents "%25"
- Double encoding of "../" represents "%252E%252E%252F"

If you do not want IIS to allow doubled-encoded requests to be served, use the following

(IIS Team,2007):

```
<configuration>
   <system.webServer>
    <security>
      <requestFiltering
       allowDoubleEscaping=”false”>
      </requestFiltering>
    </security>
   </system.webServer>
 </configuration>
```

Filter High Bit Characters

This allows or rejects all requests to IIS that contain non-ASCII characters . When this occurs error code 404.12. is displayed to the user . The UrlScan (IIS6 add-on) equivalent is AllowHighBitCharacters.

```
<configuration>
  <system.webServer>
   <security>
     <requestFiltering
     allowHighBitCharacters=”true”>
     </requestFiltering>
   </security>
  </system.webServer>
 </configuration>
```

Filter Based on File Extensions

Using this filter you can allow IIS to a request based on file extensions, the error code logged is 404.7. The AllowExtensions and DenyExtensions options are the UrlScan equivalents.

```
  <configuration>
   <system.webServer>
    <security>
     <requestFiltering>
      <fileExtensions allowUnlisted=”true” >
      <add fileExtension=”.asp” allowed=”false”/>
```

```
      </fileExtensions>
     </requestFiltering>
    </security>
   </system.webServer>
  </configuration>
```

Filter Based on Request Limits

When IIS rejects a request based on request limits, the error code logged is:

- 404.13 if the content is too long.
- 404.14 if the URL is too large.
- 404.15 if the query string is too long.

This can be used to limit a long query string or too much content sent to an application which you cannot change the source code to fix the issue.

```
    <configuration>
    <system.webServer>
      <security>
       <requestFiltering>
        <requestLimits
         maxAllowedContentLength=”30000000”
         maxUrl=”260”
         maxQueryString=”25”
         />
       </requestFiltering>
      </security>
     </system.webServer>
   </configuration>
```

Filter by Verbs

When IIS reject a request based on this feature, the error code logged is 404.6. This corresponds to the UseAllowVerbs, AllowVerbs, and DenyVerbs options in UrlScan.

In case you want the application to use only certain type of verb, it is necessary to firt set the allowUnlisted to ‘false’ and then set the verbs that you would like to allow(see example)

```
<configuration>
    <system.webServer>
     <security>
```

```
      <requestFiltering>
       <verbs allowUnlisted="false">
         <add verb="GET" allowed="true" />
       </verbs>
      </requestFiltering>
     </security>
    </system.webServer>
 </configuration>
```

Filter Based on URL Sequences

This feature defines a list of sequences that IIS can reject when it is part of a request. When IIS reject a request for this feature, the error code logged is 404.5.This corresponds to the DenyUrlSequences feature in UrlScan. This is a very powerful feature. This avoids a given character sequence from ever being attended by IIS:

```
   <configuration>
    <system.webServer>
      <security>
       <requestFiltering>
         <denyUrlSequences>
           <add sequence=".."/>
         </denyUrlSequences>
       </requestFiltering>
      </security>
     </system.webServer>
   </configuration>
```

Filter Out Hidden Segments

In case you want IIS to serve content in binary directory but not the binary, you can apply this configuration.

```
<configuration>
   <system.webServer>
      <security>
        <requestFiltering>
          <hiddenSegments>
            <add segment="BIN"/>
          </hiddenSegments>
```

```
        </requestFiltering>
      </security>
    </system.webServer>
  </configuration>
```

Password protection and sensitive information

The web.config files might include sensitive information in the connection strings such as database passwords, mail server user names among others.

Sections that are required to be encrypted are:

• <appSettings>. This section contains custom application settings.

• <connectionStrings>. This section contains connection strings.

• <identity>. This section can contain impersonation credentials.

• <sessionState>. This section contains the connection string for the out-of-process session state provider.

Passwords and user names contained in a <connectionstring> section should be encrypted. ASP.NET allows you to encrypt this information by using the functionality aspnet_regiis .This utility is found in the installed .NET framework under the folder

%windows%\Microsoft.NET\Framework\v2.0.50727

You can specify the section you need to encrypt by using the command:

aspnet_regiis -pef sectiontobeencryoted

Encrypting sections in Web.Config file

Even though encrypting sections is possible, not all sections can be encrypted, specifically sections that are read before user code is run. The following sections cannot be encrypted:

* <processModel>
* <runtime>
* <mscorlib>
* <startup>
* <system.runtime.remoting>
* <configProtectedData>
* <satelliteassemblies>
* <cryptographySettings>
* <cryptoNameMapping>
* <cryptoClasses>

Machine-Level RSA key container or User-Level Key Containers

Encrypting a single file using machine-level RSA key has its disadvantages when this file is moved to other servers. In this case, user-level RSA key container is strongly advised. The RSAProtectedConfigurationProvider supports machine-level and user-level key containers for key storage.

RSA machine key containers are stored in the following folder:

\Documents and Settings\All Users\Application Data\Microsoft\Crypto\RSA\MachineKeys

User Key Container

When the application that needs to be protected is in a shared hosting environment and protection of sensitive data cannot be accessible to other applications, the user key container is strongly recommended.

In this case each application should have a separate identity.

RSA user-level key containers are stored in the following folder:

\Documents and Settings\{UserName}\Application Data\Microsoft\Crypto\RSA

IIS configurations

Depending on the version of IIS that must be configured, it is important to revise some of its settings which can comprise security in the server.

Trust level

The trust level is a set of Code Access Security permissions granted to an application within a hosting environment. These are defined using policy files. Depending on the trust level that must be configured, it is possible to grant FULL, HIGH, MEDIUM, LOW or MINIMAL level. The ASP.NET host does not apply any additional policy to applications that are running at the full-trust level.

```
Example:
<system.web>
   <securityPolicy>
     <trustLevel name=”Full” policyFile=”internal”/>
   </securityPolicy>
 </system.web>
```

Lock Trust Levels

In the .NET framework web.config file is possible to lock applications from changing their

trust level

This file is found at:

C:\Windows\Microsoft.NET\Framework\{version}\CONFIG

The following example shows how to lock 2 different application configuration trust levels (MSDN, 2013)

```
<configuration>
   <location path="application1" allowOverride="false">
     <system.web>
       <trust level="High" />
     </system.web>
   </location>
   <location path="application2" allowOverride="false">
     <system.web>
       <trust level="Medium" />
     </system.web>
   </location>
 </configuration>
```

References

• Yakushev Ruslan , 2008 "IIS 7.0 Request Filtering and URL Rewriting " available at http://www.iis.net/learn/extensions/url-rewrite-module/iis-request-filtering-and-url-rewriting (Last accessed on 14 July, 2013)

• OWASP, 2009 "Double Encoding" available at https://www.owasp.org/index.php/Double_Encoding (Last accessed on 14 July, 2013)

• IIS Team, 2007 "Use Request Filtering " available at http://www.iis.net/learn/manage/configuring-security/ use-request-filtering (Last accessed on 14 July, 2013)

• Aguilar Carlos ,2006 "The new Configuration System in IIS 7" available at http://blogs.msdn.com/b/carlosag/archive/2006/04/25/iis7configurationsystem.aspx (Last accessed on 14 July, 2013)

• MSDN, 2013 . How to: Lock ASP.NET Configuration Settings available at http://msdn.microsoft.com/en-us/library/ms178693.aspx (Last accessed on 14 July, 2013)

2.8.9. Strongly Named Assemblies

During the build proccess either QA or Developers are going to publish the code into

executable formats. Usually this consists of an exe or and one or several DLL's. During the build/publish process a decision needs to be made to sign or not sign the code. Signing your code is called creating "strong names" by Microsoft. If you create a project using Visual Studio and use Microsofts "Run code analysis" most likely your will encounter a Microsoft design error if the code is not strong named; "Warning 1 CA2210 : Microsoft.Design : Sign 'xxx.exe' with a strong name key."

Code review needs to be aware if strong naming is being used, benefits and what threat vectors strong naming helps prevent or understand the reasons for not using strong naming.

A strong name is a method to sign an assembly's identity using its text name, version number, culture information, a public key and a digital signature. (Solis, 2012)

* Strong naming guarantees a unique name for that assembly.

* Strong names protect the version lineage of an assembly. A strong name can ensure that no one can produce a subsequent version of your assembly. Users can be sure that a version of the assembly they are loading comes from the same publisher that created the version the application was built with.

The above two points are very important if you are going to use Global Assembly Cache (GAC).

* Strong names provide a strong integrity check and prevent spoofing. Passing the .NET

Framework security checks guarantees that the contents of the assembly have not been changed since it was built. Note, however, that strong names in and of themselves do not imply a level of trust like that provided, for example, by a digital signature and supporting certificate. If you use the GAC assemblies remember the assemblies are not verified each time they load since the GAC by design is a locked-down, admin-only store.

What strong names can't prevent is a malicious user from stripping the strong name signature entirely, modifying the assembly, or re-signing it with the malicious user's key.

The code reviewer needs to understand how the strong name private key will be kept secure and managed. This is crucible if you decide strong name signatures are a good fit for your organization.

If principal of least privilege is used so code is not or less susceptible to be access by the hacker and the GAC is not being used strong names provides less benefits or no benefits at all.

How to use Strong Naming

Signing tools

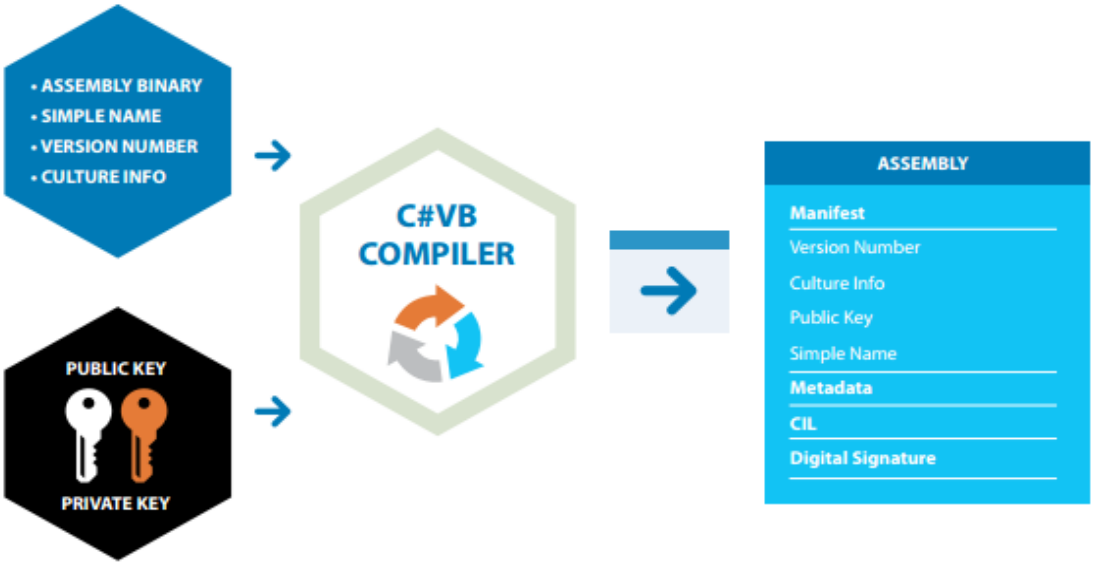
In order to create a strong name assembly there are a set of tools and steps that you need to follow

Using Visual Studio

In order to use Visual Studio to create a Strongly Named Assembly, it is necessary to have a copy of the public/private key pair file. It is also possible to create this pair key in Visual Studio

In Visual Studio 2005, the C#, Visual Basic, and Visual J# integrated development environments (IDEs) allow you to generate key pairs and sign assemblies without the need to create a key pair using Sn.exe (Strong Name Tool). These IDEs have a Signing tab in the Project Designer. . The use of the AssemblyKeyFileAttribute to identify key file pairs has been made obsolete in Visual Studio 2005.

The following figure illustrates the process done by the compiler



Using Strong Name tool

The Sign Tool is a command-line tool that digitally signs files, verifies signatures in files, or time stamps files.

The Sign Tool is not supported on Microsoft Windows NT, Windows Me, Windows 98, or Windows 95.

In case you aren't using the "Visual Studio Command Prompt" (Start >> Microsoft Visual Studio 2010 >> Visual Studio Tools >> Visual Studio Command Prompt (2010)) you can locate sn.exe at %ProgramFiles%\Microsoft SDKs\Windows\v7.0A\bin\sn.exe

The following command creates a new, random key pair and stores it in keyPair.snk.

sn -k keyPair.snk

The following command stores the key in keyPair.snk in the container MyContainer in the strong name CSP.

sn -i keyPair.snk MyContainer

The following command extracts the public key from keyPair.snk and stores it in publicKey.snk.

sn -p keyPair.snk publicKey.snk

The following command displays the public key and the token for the public key contained in publicKey.snk.

sn -tp publicKey.snk

The following command verifies the assembly MyAsm.dll.

sn -v MyAsm.dll

The following command deletes MyContainer from the default CSP.

sn -d MyContainer

Using the Assembly Linker(AI.exe)

This tool is automatically installed with Visual Studio and with the Windows SDK. To run the tool, we recommend that you use the Visual Studio Command Prompt or the Windows SDK Command Prompt (CMD Shell). These utilities enable you to run the tool easily, without navigating to the installation folder. For more information, see Visual Studio and Windows SDK Command Prompts.

If you have Visual Studio installed on your computer:

On the taskbar, click Start, click All Programs, click Visual Studio, click Visual Studio Tools, and then click Visual Studio Command Prompt.

-or-

If you have the Windows SDK installed on your computer:

On the taskbar, click Start, click All Programs, click the folder for the Windows SDK, and then click Command Prompt (or CMD Shell).

At the command prompt, type the following:

al sources options

Remarks

All Visual Studio compilers produce assemblies. However if you have one or more modules (metadata without a manifest) you can use Al.exe to create an assembly with the manifest in a separate file.

To install assemblies in the cache, remove assemblies from the cache, or list the contents of the cache, use the Global Assembly Cache Tool (Gacutil.exe).

The following command creates an executable file t2a.exe with an assembly from the t2.netmodule module. The entry point is the Main method in MyClass.

```
al t2.netmodule /target:exe /out:t2a.exe /main:MyClass.Main
```

Use Assembly attributes

You can insert the strong name information in the code directly. For this, depending on where the key file is located you can use AssemblyKeyFileAttribute or AssemblyKeyNameAttribute

```
Use Compiler options :use /keyfile or /delaysign
```

Safeguarding the key pair from developers is necessary to maintain and guarantee the integrity of the assemblies. The public key should be accessible, but access to the private key is restricted to only a few individuals. When developing assemblies with strong names, each assembly that references the strong-named target assembly contains the token of the public key used to give the target assembly a strong name. This requires that the public key be available during the development process.

You can use delayed or partial signing at build time to reserve space in the portable executable (PE) file for the strong name signature, but defer the actual signing until some later stage (typically just before shipping the assembly).

You can use /keyfile or /delaysign in C# and VB.NET (MSDN)

References

- http://msdn.microsoft.com/en-us/library/wd40t7ad(v=vs.80).aspx
- http://msdn.microsoft.com/en-us/library/c405shex(v=vs.110).aspx
- http://msdn.microsoft.com/en-us/library/k5b5tt23(v=vs.80).aspx
- http://msdn.microsoft.com/en-us/library/t07a3dye(v=vs.80).aspx
- http://msdn.microsoft.com/en-us/library/t07a3dye(v=vs.110).aspx

2.8.10.Round Tripping

Round Tripping is a reverse engineering technique that allows and attacker to decompile an assembly from a certain application. Ildasm.exe can be used for this purpose, and ILAsm is used to recompiled the assembly.

The MSIL Disassembler(Ildasm.exe) is a companion tool to the MSIL Assembler (Ilasm.exe). Ildasm.exe takes a portable executable (PE) file that contains Microsoft intermediate language (MSIL) code and creates a text file suitable as input to Ilasm.exe. This tool is automatically installed with Visual Studio and with the Windows SDK. s

The importance of Obfuscation

As mentioned before, Round Tripping is a technique used to reverse engineer assemblies. Therefore, if you want to avoid your assemblies being reversed engineered or even worse, that the code is victim of malicious manipulation using the Ildasm and Ilasm tools, then it is advisable to apply it. There are different kinds of products that can be used for this purpose such as DeepSea, Crypto or Dotfuscator.

Using Obfuscation

The most effective technique used to avoid reverse engineering and tampering of assemblies is the use of Obfuscation. Visual Studio contains a version of Dotfuscator. This program is accessible by choosing on the VS menu, Tools → Dotfuscator(Community Edition menu command). Note: This tools is not available in Express versions

To obfuscate your assemblies:

* Build the project in VS Studio

* Tools--> Dotfuscator Community Edition

* A screen prompts asking for which project type, choose ‘Creat New Project’ and click OK

* On the Input tab of the Dotfuscator interface, click ‘Browse and Add assembly to list’

Browse for the compiled application

ASPX .NETWeb Configs

Introduction

Securing resources in ASP.NET applications is a combination of configuration settings in the Web.config file but also, it’s important to remember that the IIS configurations play also a big part on this. It’s an integrated approach which provides a total framework of security.

The following highlights the most important aspects of ASP.NET configuration settings within the web.config file. For a total overview see chapter ASP.NET security (https://www.owasp.org/index.php/CRV2_FrameworkSpecIssuesASPNet)

Secure Configuration Values

Sensitive Information saved in config files should be encrypted. Encryption keys stored in

the machineKey element for example or connectionstrings with username and passwords to login to database.

Lock ASP.NET Configuration settings

You can lock configuration settings in ASP.NET configuration files (Web.config files) by adding an allowOverride attribute to a location element

Configure directories using Location Settings

Through the <location> element you can establish settings for specific folders and files. The Path attribute is used to specify the file or subdirectory. This is done in the Web.config file example:

```
<location path=”.” >
    <section1 .../>
    <section2 ... />
  </location>
  <location path=”Default Web Site” >
    <section1 … />
    <section2 … />
  </location
  <location path=”Default Web Site/MyApplication/Admin/xyz.html” >
    <section1 ... />
    <section2 ... />
  </location>
```

Configure exceptions for Error Code handling

Showing and handling the correct error code when a user sends a bad request or invalid parameters is an important configuration subject. Logging these errors are also an excellent help when analyzing potential attacks to the application.

It is possible to configure these errors in the code or in the Web.Config file

The HttpException method describes an exception that occurred during the processing of HTTP requests.For example:

```
if (string.IsNullOrEmpty(Request[“id”]))
     throw new HttpException(400, “Bad request”);
```

or in the Web.config file:

```
<configuration>
  <system.web>
    <customErrors mode=”On” defaultRedirect=”ErrorPage.html”
```

```
                redirectMode="ResponseRewrite">
      <error statusCode="400" redirect="BadRequest.html" />
      <error statusCode="404" redirect="FileNotFound.html" />
    </customErrors>
  </system.web>
</configuration>
```

Input validation

Anything coming from external sources can be consider as input in a web application. Not only the user inserting data through a web form, but also data retrieved from a web service or database, also headers sent from the browsers fall under this concept. A way of defining when input is safe can be done through outlining a trust boundary.

Defining what is known as trust boundary can help us to visualize all possible untrusted inputs. One of those are user input.ASP.NET has different types of validations depending on the level of control to be applied. By default, web pages code is validated against malicious users. The following is a list types of validations used (MSDN, 2013):

Type of validation	Control to use	Description
Required entry	RequiredFieldValidator	Ensures that the user does not skip an entry.
Comparison to a value	CompareValidator	Compares a user's entry against a constant value, against the value of another control (using a comparison operator such as less than, equal, or greater than), or for a specific data type.
Range checking	RangeValidator	Checks that a user's entry is between specified lower and upper boundaries. You can check ranges within pairs of numbers, alphabetic characters, and dates.
Pattern matching	RegularExpressionValidator	Checks that the entry matches a pattern defined by a regular expression. This type of validation enables you to check for predictable sequences of characters, such as those in e-mail addresses, telephone numbers, postal codes, and so on.
User-defined	CustomValidator	Checks the user's entry using validation logic that you write yourself. This type of validation enables you to check for values derived at run time.

References

MSDN, 2013 "Securing ASP.NET Configurations" available at http://msdn.microsoft.com/en-us/library/ms178699%28v=vs.100%29.aspx (Last Viewed, 25th July 2013)

2.8.11.*.NET Authentication Controls*

In the .NET, there are Authentication tags in the configuration file.

The <authentication> element configures the authentication mode that your applications use.

<authentication>

The appropriate authentication mode depends on how your application or Web service has

been designed. The default Machine.config setting applies a secure Windows authentication default as shown below.

authentication Attributes:mode="[Windows|Forms|Passport|None]"

<authentication mode="Windows" />

Forms Authentication Guidelines

To use Forms authentication, set mode="Forms" on the <authentication> element. Next, configure Forms authentication using the child <forms> element. The following fragment shows a secure <forms> authentication element configuration:

```
<authentication mode="Forms">
  <forms loginUrl="Restricted\login.aspx" Login page in an SSL protected
folder
   protection="All" Privacy and integrity
   requireSSL="true" Prevents cookie being sent over http
   timeout="10" Limited session lifetime
   name="AppNameCookie" Unique per-application name
   path="/FormsAuth" and path
   slidingExpiration="true" > Sliding session lifetime
  </forms>
 </authentication>
```

Use the following recommendations to improve Forms authentication security:

*Partition your Web site.

*Set protection="All".

*Use small cookie time-out values.

*Consider using a fixed expiration period.

*Use SSL with Forms authentication.

*If you do not use SSL, set slidingExpiration = "false".

*Do not use the <credentials> element on production servers.

*Configure the <machineKey> element.

*Use unique cookie names and paths.

Classic ASP

For classic ASP pages, authentication is usually performed manually by including the user information in session variablesafter validation against a DB, so you can look for something like:

Session ("UserId") = UserName

Session ("Roles") = UserRoles

Code Review .NET Manage Code

.NET Managed code is less vulnerable to common vulnerabilities found in unmanaged code such as Buffer Overflows and memory corruption however there could be issues in the code that can affect performance and security. The following is a summary of the recommended practices to look for during the code review. Also, it is worth mentioning some tools that can make the work easier on this part and they can help you understand and pin point flaws in your code

Code Access Security

This supports the execution of semi-trusted code, preventing several forms of security threats. The following is a summary of possible vulnerabilities due to improper use of Code Access security:

Vulnerability	Implications
Improper use of link demands or asserts	The code is susceptible to luring attacks
Code allows untrusted callers	Malicious code can use the code to perform sensitive operations and access resources

Declarative security

Use declarative security instead of imperative whenever possible. Example of declarative syntax(MSDN[2], 2013):

```
[MyPermission(SecurityAction.Demand, Unrestricted = true)]
  public class MyClass
  {
   public MyClass()
   {
      //The constructor is protected by the security call.
   }
   public void MyMethod()
   {
      //This method is protected by the security call.
   }
   public void YourMethod()
   {
```

```
        //This method is protected by the security call.
    }
}
```

Unmanaged code

Even though C# is a strong type language, it is possible to use unmanaged code calls by using the 'unsafe' code. Check that any class that uses an unmanaged resource, such as a database connection across method calls, implements the IDisposable interface. If the semantics of the object are such that a Close method is more logical than a Dispose method, provide a Close method in addition to Dispose.

Exception handling

Manage code should use exception handling for security purposes among other reasons. Make sure that you follow these recommendations:

- Avoid exception handling in loops, use try/catch block if it is necessary.
- Identify code that swallows exceptions
- Use exceptions handling for unexpected conditions and not just to control the flow in the application

Tools

FxCop

FxCop is an analysis tool that analyses binary assemblies, not source code. The tool has a predefined set of rules and it is possible to configure and extend them.

Some of the available rules regarding security are (CodePlex, 2010):

Rule	Description
EnableEventValidationShouldBeTrue	Verifies if the EnableEventValidation directive is disabled on a certain page.
ValidateRequestShouldBeEnabled	Verifies if the ValidateRequest directive is disabled on a certain page.
ViewStateEncryptionModeShouldBeAlways	Verifies if the ViewStateEncryptionMode directive is not set to Never on a certain page.
EnableViewStateMacShouldBeTrue	Verifies if the EnableViewStateMac directive is not set to false on a certain page.
EnableViewStateShouldBeTrue	Verifies if the EnableViewState directive is not set to false on a certain page.
ViewStateUserKeyShouldBeUsed	Verifies if the Page.ViewStateUserKey is being used in the application to prevent CSRF.
DebugCompilationMustBeDisabled	Verifies that debug compilation is turned off. This eliminates potential performance and security issues related to debug code enabled and additional extensive error messages being returned.
CustomErrorPageShouldBeSpecified	Verifies that the CustomErrors section is configured to have a default URL for redirecting uses in case of error.
FormAuthenticationShouldNotContainFormAuthenticationCredentials	Verifies that no credentials are specified under the form authentication configuration.
EnableCrossAppRedirectsShouldBeTrue	Verifies that system.web.authentication.forms enableCrossAppRedirects is set to true. The settings indicate if the user should be redirected to another application url after the authentication process. If the setting is false, the authentication process will not allow redirection to another application or host. This helps prevent an attacker to force the user to be redirected to another site during the authentication process. This attack is commonly called Open redirect and is used mostly during phishing attacks.
FormAuthenticationProtectionShouldBeAll	Verifies that the protection attribute on the system.web.authentication.forms protection is set to All which specifies that the application use both data validation and encryption to help protect the authentication cookie.
FormAuthenticationRequireSSLShouldBeTrue	Verifies that the requireSSL attribute on the system.web.authentication.forms configuration element is set to True which forces the authentication cookie to specify the secure attribute. This directs the browser to only provide the cookie over SSL.
FormAuthenticationSlidingExpirationShouldBeFalse	Verifies that system.web.authentication.forms slidingExpiration is set to false when the site is being served over HTTP. This will force the authentication cookie to have a fixed timeout value instead of being refreshed by each request. Since the cookie will traverse over clear text network and could potentially be intercepted, having a fixed timeout value on the cookie will limit the amount of time the cookie can be replayed. If the cookie is being sent only over HTTPS, it is less likely to be intercepted and having the slidingExpiration setting to True will cause the timeout to be refreshed after each request which gives a better user experience.
HttpCookiesHttpOnlyCookiesShouldBeTrue	Verifies that the system.web.httpCookies httpOnlyCookies configuration setting is set to True which forces all cookies to be sent with the HttpOnly attribute.

Rule	Description
HttpCookiesRequireSSLShouldBeTrue	Verifies that the system.web.httpCookies requireSSL configuration is set to True which forces all cookies to be sent with the secure attribute. This indicates the browser to only provide the cookie over SSL.
TraceShouldBeDisabled	Verifies that the system.web.trace enabled setting is set to false which disables tracing. It is recommended to disable tracing on production servers to make sure that an attacker cannot gain information from the trace about your application. Trace information can help an attacker probe and compromise your application.
AnonymousAccessIsEnabled	Looks in the web.config file to see if the authorization section allows anonymous access.
RoleManagerCookieProtectionShouldBeAll	Verifies that the system.web.rolemanager cookieProtection is set to All which enforces the cookie to be both encrypted and validated by the server.
RoleManagerCookieRequireSSLShouldBeTrue	Verifies that the system.web.rolemanager cookieRequireSSL attribute is set to True which forces the role manager cookie to specify the secure attribute. This directs the browser to only provide the cookie over SSL.
RoleManagerCookieSlidingExpirationShouldBeTrue	Verifies that the system.web.rolemanager cookieSlidingExpiration is set to false when the site is being served over HTTP. This will force the authentication cookie to have a fixed timeout value instead of being refreshed by each request. Since the cookie will traverse over clear text network and could potentially be intercepted, having a fixed timeout value on the cookie will limit the amount of time the cookie can be replayed. If the cookie is being sent only over HTTPS, it is less likely to be intercepted and having the slidingExpiration setting to True will cause the timeout to be refreshed after each request which gives a better user experience.
PagesEnableViewStateMacShouldBeTrue	Verifies that the viewstate mac is enabled.
PagesEnableEventValidationMustBeTrue	Verifies that event validation is enabled.
HttpRuntimeEnableHeaderCheckingShouldBeTrue	Verifies that the system.web.httpRuntime enableHeaderChecking attribute is set to true. The setting indicates whether ASP.NET should check the request header for potential injection attacks. If an attack is detected, ASP.NET responds with an error. This forces ASP.NET to apply the ValidateRequest protection to headers sent by the client. If an attack is detected the application throws HttpRequestValidationException.
PagesValidateRequestShouldBeEnabled	Verify that validateRequest is enabled.
PagesViewStateEncryptionModeShouldBeAlways	Verifies that the viewstate encryption mode is not configured to never encrypt.
CustomErrorsModeShouldBeOn	Verifies that the system.web.customErrors mode is set to On or RemoteOnly. This disable detailed error message returned by ASP.NET to remote users.
MarkVerbHandlersWithValidateAntiforgeryToken	ValidateAntiforgeryTokenAttribute is used to protect against potential CSRF attacks against ASP.NET MVC applications.

2.9. SENSITIVE DATA EXPOSURE

Many web applications do not properly protect sensitive data, such as credit cards, tax IDs, and authentication credentials. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.

2.9.1. Cryptographic Controls

Software developers, architects and designers are at the forefront of deciding which category a particular application resides in. Cryptography provides for security of data at rest (via encryption), enforcement of data integrity (via hashing/digesting), and non-repudiation of data (via signing). To ensure this cryptographic code adequately protections the data, all source code must use a standard (secure) algorithms with strong key sizes.

Common flaws when implementing cryptographic code includes the use of non-standard

cryptographic algorithms, custom implementation of cryptography (standard & non-standard) algorithms, use of standard algorithms which are cryptographically insecure (e.g. DES), and the implementation of insecure keys can weaken the overall security posture of any application. Implementation of the aforementioned flaws enable attackers to use cryptanalytic tools and techniques to decrypt sensitive data.

Description

Many companies handle sensitive information for their customers, for instance medical details or credit card numbers, and industry regulations dictate this sensitive information must be encrypted to protect the customers' information. In the medical industry the HIPAA regulations advise businesses what protections must be applied to medical data, in the financial industry many regulations cover PII (personally identifiable information) controls.

Regardless of the financial impact of regulatory penalties, there are many business reasons to protect (though encryption or hashing) the information processed by an application, including privacy and fraud detection/protection.

All sensitive data that the application handles should be identified and encryption should be enforced. Similarly a decision should be made as to whether sensitive data must be encrypted in transit (i.e. being sent from one computer to another) and/or at rest (i.e. stored in a DB, file, keychain, etc.):

1) Protection in transit; this typically means using the SSL/TLS layer to encrypt data travelling on the HTTP protocol, although it can also include FTPS, or even SSL on TCP. Frameworks such as IIS and Apache Struts come with SSL/TLS functionality included, and thus the developer will not be coding the actual TLS encryption, but instead will be configuring a framework to provide TLS security.

However the decisions made here, even at an architectural level, need to be well informed, and a discussion on TLS design decisions is covered in section 1.3.

2) Protection at rest; this can include encryption of credit cards in the database, hashing of passwords, producing message authentication codes (MACs) to ensure a message has not been modified between computers. Where TLS code will come with a framework, code to encrypt or hash data to be stored will typically need to use APIs provided by cryptographic libraries.

The developer will not be writing code to implement the AES algorithm (OpenSSL or CryptoAPI will do that), the developer will be writing modules to use an AES implantation in the correct way. Again the correct decisions need to be made regarding up-to-date algorithms, key storage, and other design decisions, which are covered in section 1.4.

Cryptography Definitions

Before diving into discussions on encrypting traffic and data, some terminology used in the realm of cryptography is defined in **table 19**.

Term	Description
Encoding	Transforming data from one form into another, typically with the aim of making the data easier to work with. For example encoding binary data (which could not be printed to a screen into printable ASCII format which can be copy/pasted. Note that encoding does not aim to hide the data, the method to return the encoded data back to its original form will be publically known.
Entropy	Essentially this is randomness. Cryptographic functions will need to work with some form of randomness to allow the source data to be encrypted in such a way that an attacker cannot reverse the encryption without the necessary key. Having a good source of entropy is essential to any cryptographic algorithm.
Hashing	Non-reversible transformation of data into what is called a 'fingerprint' or 'hashvalue'. Input of any size can be taken and always results in the same size of output (for the algorithm). The aim is not to convert the fingerprint back into the source data at a later time, but to run the hash algorithm over two sets of data to determine if they produce the same fingerprint. This would show that data has not been tampered with.
Salt	A non-secret value that can be added to a hashing algorithm to modify the fingerprint result. One attack against hashing algorithms is a 'rainbow table attack' where all source values are pre-computed and a table produced. The attacker can then take a fingerprint, look it up in the table, and correspond it to the original data. Using a unique salt value for each data to be hashed protects against rainbow tables, as a rainbow table for each salt value would need to be created, which would greatly extend the time taken by an attacker. The salt is not a secret and can be stored or sent with the fingerprint.
Encryption	Transformation of source data into an encrypted form that can be reversed back to the original source. Typically the algorithms used to encrypt are publically known, but rely on a secret 'key' to guide the transformation. An attacker without the key should not be able to transform the data back into the original source.
Symmetric Encryption	A form of encryption where the same key is known to both the sender and the receiver. This is a fast form of encryption, however requires a secure, out-of-band method to pass the symmetric key between the sender and receiver.
Public-Key Encryption (PKI)	A form of encryption using two keys, one to encrypt the data, and one to decrypt the data back to its original form. This is a slower method of encryption however one of the keys can be publically known (referred to as a 'public key'). The other key is called a 'private key' and is kept secret. Any data encrypted with the public key can be decrypted back into its original form using the private key. Similarly any data encrypted with the private key can be decrypted back to its original form using the public key.
Certificate	An association between an entity (e.g. person, company) and a public key. Typically this forms part of a public-key infrastructure where certain trusted entities (e.g. Certificate Authorities in internet TLS) perform validation of an entitles credentials and assert (using their own certificate) that a declared public key belongs to the entity.

2.9.2. What to Review: Protection in Transit

The terms, Secure Socket Layer (SSL) and Transport Layer Security (TLS) are often used interchangeably. In fact, SSL v3.1 is equivalent to TLS v1.0. However, different versions of SSL and TLS are supported by modern web browsers and by most modern web frameworks and platforms. Note that since developments in attacks against the SSL protocol have shown it to be weaker against attacks, this guide will use the term TLS to refer to transport layer security over the HTTP or TCP protocols.

The primary benefit of transport layer security is the protection of web application data from unauthorized disclosure and modification when it is transmitted between clients (web browsers) and the web application server, and between the web application server and back end and other non-browser based enterprise components.

In theory, the decision to use TLS to protect computer to computer communication should be based on the nature of the traffic or functionality available over the interface. If sensitive information is passing over the interface, TLS will prevent eavesdroppers from being able to view or modify the data. Likewise if the interface allows money to be transferred, or sensitive functions to be initiated, then TLS will protect the associated login or session information authorizing the user to perform those functions. However with the

price of certificates dropping, and TLS configuration within frameworks becoming easier, TLS protection of an interface is not a large endeavor and many web sites are using TLS protections for their entire site (i.e. there are only HTTPS pages, no HTTP pages are available).

The server validation component of TLS provides authentication of the server to the client. If configured to require client side certificates, TLS can also play a role in client authentication to the server. However, in practice client side certificates are not often used in lieu of username and password based authentication models for clients.

Using Validated Implementations

The US government provides a list of software that has been validated to provide a strong and secure implementation of various cryptographic functions, including those used in TLS. This list is referred to as the FIPS 140-2 validated cryptomodules [insert reference].

A cryptomodule, whether it is a software library or a hardware device, implements cryptographic algorithms (symmetric and asymmetric algorithms, hash algorithms, random number generator algorithms, and message authentication code algorithms). The security of a cryptomodule and its services (and the web applications that call the cryptomodule) depend on the correct implementation and integration of each of these three parts. In addition, the cryptomodule must be used and accessed securely. In order to leverage the benefits of TLS it is important to use a TLS service (e.g. library, web framework, web application server) which has been FIPS 140-2 validated. In addition, the cryptomodule must be installed, configured and operated in either an approved or an allowed mode to provide a high degree of certainty that the FIPS 140-2 validated cryptomodule is providing the expected security services in the expected manner.

When reviewing designs or code that is handling TLS encryption, items to look out for include:

• Use TLS for the login pages and any authenticated pages. Failure to utilize TLS for the login landing page allows an attacker to modify the login form action, causing the user's credentials to be posted to an arbitrary location. Failure to utilize TLS for authenticated pages after the login enables an attacker to view the unencrypted session ID and compromise the user's authenticated session.

• Use TLS internally when transmitting sensitive data or exposing authenticated functionality. All networks, both external and internal, which transmit sensitive data must utilize TLS or an equivalent transport layer security mechanism. It is not sufficient to claim that access to the internal network is "restricted to employees". Numerous recent data compromises have shown that the internal network can be breached by attackers. In these attacks, sniffers have been installed to access unencrypted sensitive data sent on the internal network.

• Prefer all interfaces (or pages) being accessible only over HTTPS. All pages which are available over TLS must not be available over a non-TLS connection. A user may inadvertently bookmark or manually type a URL to a HTTP page (e.g. http://example.com/myaccount) within the authenticated portion of the application.

• Use the “secure” and “http-only” cookie flags for authentication cookies. Failure to use the “secure” flag enables an attacker to access the session cookie by tricking the user’s browser into submitting a request to an unencrypted page on the site. The “http-only” flag denies JavaScript functions access to the cookies contents.

• Do not put sensitive data in the URL. TLS will protect the contents of the traffic on the wire, including the URL when transported, however remember that URL’s are visible in browser history settings, and typically written to server logs.

• Prevent the caching of sensitive data. The TLS protocol provides confidentiality only for data in transit but it does not help with potential data leakage issues at the client or intermediary proxies.

• Use HTTP Strict Transport Security (HSTS) for high risk interfaces. HSTS will prevent any web clients from attempting to connect to your web site over a non-TLS protocol. From a server-side point of view this may seem irrelevant if no non-TLS pages are provided, however a web site setting up HSTS does protect clients from other attacks (e.g. DNS cache poisioning).

• Use 2048 key lengths (and above) and SHA-256 (and above). The private key used to generate the cipher key must be sufficiently strong for the anticipated lifetime of the private key and corresponding certificate. The current best practice is to select a key size of at least 2048 bits. Note that attacks against SHA-1 have shown weakness and the current best practice is to use at least SHA-256 or equivalent.

• Only use specified, fully qualified domain names in your certificates. Do not use wildcard certificates, or RFC 1918 addresses (e.g. 10.* or 192.168.*). If you need to support multiple domain names use Subject Alternate Names (SANs) which provide a specific listing of multiple names where the certificate is valid. For example the certificate could list the subject’s CN as example.com, and list two SANs: abc.example.com and xyz.example.com. These certificates are sometimes referred to as “multiple domain certificates”.

• Always provide all certificates in the chain. When a user receives a server or host’s certificate, the certificate must be validated back to a trusted root certification authority. This is known as path validation. There can be one or more intermediate certificates in between the end-entity (server or host) certificate and root certificate. In addition to validating both endpoints, the client software will also have to validate all intermediate certificates, which can cause failures if the client does not have the certificates. This occurs in many mobile platforms.

2.9.3. *What to Review: Protection at Rest*

As a general recommendation, companies should not create their own custom cryptographic libraries and algorithms. There is a huge distinction between groups, organizations, and individuals developing cryptographic algorithms and those that implement cryptography either in software or in hardware. Using an established cryptographic library that has been developed by experts and tested by the industry is the safest way to implement cryptographic functions in a company’s code. Some common

examples of libraries used in various languages and environments are covered in the below table.

Popular cryptographic implementations according to environment

Language	Libraries	Discussion
C#.NET	Class libraries within 'System.Security.Cryptography'	For applications coded in C#.NET there are class libraries and implementations within the 'System.Security.Cryptography' that should be used. This namespace within .NET aims to provide a number of warppers that do not require proficient knowledge of cryptography in order to use it.
C/C++ (Win32)	CryptoAPI and DPAPI	For C/C++ code running on Win32 platforms, the CreyptoAPI and DPAPI are recommended.
C/C++ (Linux)	OpenSSL, NSS, boringssl	For C/C++ on Linux/Unix operating systems, us OpenSSL, NSS, or one of the many forks of these libraries.
ASP	CryptoAPI and DPAPI	Classis ASP pages do not have direct access to cryptographic functions, so the only way is to create COM wrappers in Visual C++ or Visual Basic, implementing calls to CryptoAPI or DPAPI. Then call them from ASP pages using the Server.CreateObject method.
Java	Java Cryptography Extension, BouncyCastle, Spring Security	JCE is a standard API that any cryptographic library can implement to provide cryptographic functions to the developer. Oracle provide a list of companies that act as Cryptographic Service Providers and/or offer clean room implementations of the JCE. BouncyCastle is on of the more popular implementations. Spring Secuirty is also popular in application where Spring is

already being utilized.

A secure way to implement robust encryption mechanisms within source code is by using FIPS [7] compliant algorithms with the use of the Microsoft Data Protection API (DPAPI) [4] or the Java Cryptography Extension (JCE) [5].

A company should identify minimum standards for the following when establishing your cryptographic code strategy:

• Which standard algorithms are to be used by applications

• Minimum key sizes to be supported

• What types of data must be encrypted

When reviewing code handling cryptography look out for:

• Is strong enough encryption algorithms being used, and is the implementation of those algorithms FIPS-140 complaint.

• Is the right type of cryptographic algorithm being used, is data being hashed that should be encrypted with a symmetric key? If there is no way to safely transfer the symmetric key to the other party, is public key cryptographic algorithms being employed?

• In any cryptographic system the protection of the key is the most important aspect. Exposure of the symmetric or private key means the encrypted data is no longer private. Tightly control who has access to enter or view the keys, and how the keys are used within applications.

• Any code implementing cryptographic processes and algorithms should be reviewed and audited against a set of company or regulatory specifications. High level decisions need to be made (and continually revisited) as to what an organization considers 'strong encryption' to be, and all implementation instances should adhere to this standard.

• Cryptographic modules must be tested under high load with multithreaded implementations, and each piece of encrypted data should be checked to ensure it was encrypted and decrypted correctly.

• In .NET check for examples of cryptography in the MSDN Library Security Practices: .NET Framework 2.0 Security Practices at a Glance

o Check that the Data Protection API (DPAPI) is being used.

o Verify no proprietary algorithms are being used.

o Check that RNGCryptoServiceProvider is used for PRNG.

o Verify key length is at least 128 bits.

• In ASP perform all of these checks on the COM wrapper as ASP does not have direct access to cryptographic functions

o Check that the Data Protection API (DPAPI) or CryptoAPI is being used into COM object

o Verify no proprietary algorithms are being used

o Check that RNGCryptoServiceProvider is used for PRNG

o Verify key length is at least 128 bits

• For Java check that the Java Cryptography Extension (JCE) is being used

o Verify no proprietary algorithms are being used

o Check that SecureRandom (or similar) is used for PRNG

o Verify key length is at least 128 bits

Bad Practice: Use of Insecure Cryptographic Algorithms

The DES and SHA-0 algorithms are cryptographically insecure. The example in figure A6.0 outlines a cryptographic module using DES (available per using the Java Cryptographic Extensions) which should not be used. Additionally, SHA-1 and MD5 should be avoided in new applications moving forward.

FIGURE A6.0

```
Example code using insecure cryptographic algorithms

package org.badexample.crypto;


<snip>
try {
            /**  Step 1. Generate a DES key using KeyGenerator */
            KeyGenerator keyGen = KeyGenerator.getInstance("DES");
            SecretKey secretKey = keyGen.generateKey();

            /**  Step2. Create a Cipher by specifying the following
parameters
             *                a. Algorithm name - here it is DES
             *                b. Mode - here it is CBC
             *                c. Padding - PKCS5Padding */
            Cipher desCipher = Cipher.getInstance("DES/CBC/PKCS5Padding");
<snip>
```

Good Practice: Use Strong Entropy

The source code in figure A6.1 outlines secure key generation per use of strong entropy:

FIGURE A6.1

Example code using strong entropy

```
package org.owasp.java.crypto;
import java.security.SecureRandom;
import java.security.NoSuchAlgorithmException;
import sun.misc.BASE64Encoder;
/**
 * This program provides the functionality for Generating a Secure Random
Number.
 * There are 2 ways to generate a Random number through SecureRandom.
 * 1. By calling nextBytes method to generate Random Bytes
 * 2. Using setSeed(byte[]) to reseed a Random object
*/
public class SecureRandomGen {
      public static void main(String[] args) {
            try {
              // Initialize a secure random number generator
              SecureRandom secureRandom = SecureRandom.getInstance("SHA512");

              // Method 1 - Calling nextBytes method to generate Random Bytes
              byte[] bytes = new byte[512];
              secureRandom.nextBytes(bytes);

              // Printing the SecureRandom number by calling
secureRandom.nextDouble()
              System.out.println(" Secure Random # generated by calling
nextBytes() is " + secureRandom.nextDouble());

              // Method 2 - Using setSeed(byte[]) to reseed a Random object
              int seedByteCount = 10;
              byte[] seed = secureRandom.generateSeed(seedByteCount);

              secureRandom.setSeed(seed);
              System.out.println(" Secure Random # generated using
setSeed(byte[]) is  " + secureRandom.nextDouble());

          } catch (NoSuchAlgorithmException noSuchAlgo)
```

```
            {
                  System.out.println(" No Such Algorithm exists " +
noSuchAlgo);
            }
      }
}
```

Good Practice: Use Strong Algorithms

Below illustrates the implementation of AES (available per Using the Java Cryptographic Extensions):

FIGURE A6.2

Example code using strong entropy

```
package org.owasp.java.crypto;

import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.Cipher;

import java.security.NoSuchAlgorithmException;
import java.security.InvalidKeyException;
import java.security.InvalidAlgorithmParameterException;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.BadPaddingException;
import javax.crypto.IllegalBlockSizeException;

import sun.misc.BASE64Encoder;

/**
 * This program provides the following cryptographic functionalities
 * 1. Encryption using AES
 * 2. Decryption using AES
 *
 * High Level Algorithm :
 * 1. Generate a DES key (specify the Key size during this phase)
 * 2. Create the Cipher
```

```
 * 3. To Encrypt : Initialize the Cipher for Encryption
 * 4. To Decrypt : Initialize the Cipher for Decryption
 */

public class AES {
      public static void main(String[] args) {

            String strDataToEncrypt = new String();
            String strCipherText = new String();
            String strDecryptedText = new String();
try{
            /**
             *  Step 1. Generate an AES key using KeyGenerator
             *          Initialize the keysize to 128
             */
            KeyGenerator keyGen = KeyGenerator.getInstance("AES");
            keyGen.init(128);
            SecretKey secretKey = keyGen.generateKey();

            /**
             *  Step2. Create a Cipher by specifying the following parameters
             *                a. Algorithm name - here it is AES
             */
            Cipher aesCipher = Cipher.getInstance("AES");

            /**
             *  Step 3. Initialize the Cipher for Encryption
             */
            aesCipher.init(Cipher.ENCRYPT_MODE,secretKey);

            /**
             *  Step 4. Encrypt the Data
             *          1. Declare / Initialize the Data. Here the data is of
type String
             *          2. Convert the Input Text to Bytes
             *          3. Encrypt the bytes using doFinal method
             */
            strDataToEncrypt = "Hello World of Encryption using AES ";
            byte[] byteDataToEncrypt = strDataToEncrypt.getBytes();
```

```
            byte[] byteCipherText = aesCipher.doFinal(byteDataToEncrypt);
            strCipherText = new BASE64Encoder().encode(byteCipherText);
            System.out.println("Cipher Text generated using AES is "
+strCipherText);

            /**
             *  Step 5. Decrypt the Data
             *          1. Initialize the Cipher for Decryption
             *          2. Decrypt the cipher bytes using doFinal method
*/

      aesCipher.init(Cipher.DECRYPT_MODE,secretKey,aesCipher.getParameters())
;
            byte[] byteDecryptedText = aesCipher.doFinal(byteCipherText);
            strDecryptedText = new String(byteDecryptedText);
            System.out.println(" Decrypted Text message is "
+strDecryptedText);
            }

            catch (NoSuchAlgorithmException noSuchAlgo)
            {
                  System.out.println(" No Such Algorithm exists " +
noSuchAlgo);
            }
                  catch (NoSuchPaddingException noSuchPad)
                  {
                        System.out.println(" No Such Padding exists " +
noSuchPad);
                  }
                        catch (InvalidKeyException invalidKey)
                        {
                              System.out.println(" Invalid Key " +
invalidKey);
                        }
                        catch (BadPaddingException badPadding)
                        {
                              System.out.println(" Bad Padding " +
badPadding);
                        }
                        catch (IllegalBlockSizeException illegalBlockSize)
                        {
                              System.out.println(" Illegal Block Size " +
```

```
illegalBlockSize);
                        }
                        catch (InvalidAlgorithmParameterException
invalidParam)
                        {
                              System.out.println(" Invalid Parameter " +
invalidParam);
                        }
      }
}
```

References

[1] Bruce Schneier, Applied Cryptography, John Wiley & Sons, 2nd edition, 1996.

[2] Michael Howard, Steve Lipner, The Security Development Lifecycle, 2006, pp. 251 - 258

[3] .NET Framework Developer's Guide, Cryptographic Services, http://msdn2.microsoft.com/en-us/library/93bskf9z.aspx

[4] Microsoft Developer Network, Windows Data Protection, http://msdn2.microsoft.com/en-us/library/ms995355.aspx

[5] Sun Developer Network, Java Cryptography Extension, http://java.sun.com/products/jce/

[6] Sun Developer Network, Cryptographic Service Providers and Clean Room Implementations, http://java.sun.com/products/jce/jce122_providers.html

[7] Federal Information Processing Standards, http://csrc.nist.gov/publications/fips/

2.9.4. Encryption, Hashing & Salting

A cryptographic hash algorithm; also called a hash "function" is a computer algorithm designed to provide a random mapping from an arbitrary block of data (string of binary data) and return a fixed-size bit string known as a "message digest" and achieve certain security.

Cryptographic hashing functions are used to create digital signatures, message authentication codes (MACs), other forms of authentication and many other security applications in the information infrastructure. They are also used to store user passwords in databases instead of storing the password in clear text and help prevent data leakage in session management for web applications. The actual algorithm used to create a cryptology function varies per implementation (SHA-256, SHA-512, etc.)

Never accept in a code review an algorithm created by the programmer for hashing. Always use cryptographic functions that are provided by the language, framework, or common (trusted) cryptographic libraries. These functions are well vetted and well tested by experience cryptographers.

In the United States in 2000, the department of Commerce Bureau of Export revised encryption export regulations. The results of the new export regulations it that the regulations have been

greatly relaxed. However if the code is to be exported outside of the source country current export laws for the export and import counties should be reviewed for compliance.

Case in point is if the entire message is hashed instead of a digital signature of the message the National Security Agency (NSA) considers this a quasi-encryption and State controls would apply.

It is always a valid choice to seek legal advice within the organization if the code review is being done to ensure legal compliance.

With security nothing is secure forever. This is especially true with cryptographic hashing functions. Some hashing algorithms such as Windows LanMan hashes are considered completely broken. Others like MD5, which in the past were considered safe for password hash usage, have known issues like collision attacks (note that collision attacks do not affect password hashes). The code reviewer needs to understand the weaknesses of obsolete hashing functions as well as the current best practices for the choice of cryptographic algorithms.

Working with Salts

The most common programmatic issue with hashing is:

- Not using a salt value
- Using a salt the salt value is too short
- Same salt value is used in multiple hashes.

The purpose of a salt is to make it harder for an attacker to perform pre-computed hashing attack (e.g., using rainbow tables). Take for example that the SHA512 has of 'password' is as shown in row 1 of table X, and any attacker with a rainbow table will spot the hash value corresponding to 'password'. Taking into consideration it takes days or weeks to compute a rainbow table to values up to around 8 or 10 characters, the effort to produce this table is worth it when an application is not using any salts.

Now take a scenario where an application adds a salt of 'WindowCleaner' to all passwords entered. Now the hash of 'password' becomes the hash of 'passwordWindowCleaner', which is shown in row 2 of table X. This is unlikely to be in the attackers rainbow table, however the attacker can now spend the next 10 days (for example) computing a new rainbow table with 'WindowCleaner' on the end of every 8 to 10 character string and they again can now decode our hashed database of passwords.

At a last step, an application can create a random salt for each entry, and store that salt in the DB with the hashed password. Now for user1, the random salt is 'a0w8hsdfas8ls587uas87', meaning the password to be hashed is 'passworda0w8hsdfas8ls587uas87', shown in row 3 of table X, and for user2, the random salt is '8ash87123klnf9d8dq3w', meaning the password to be hashed is 'password8ash87123klnf9d8dq3w', shown in row 4 of table X, and repeat for all

users. Now an attacker would need a rainbow table for each users' password they mean to decrypt – whereas before it took 10 days to decrypt all of the DB passwords using the same salt, now it takes 10 days to create a rainbow table for user1's password, and another 10 days for user2's password, etc. If there were 100,000 users, that's now 100,000 x 10 days = 1,000,000 days or 2738 years, to create rainbow tables for all the users.

As can be seen, the salt does not need to be secret, as it is the fact that unique salts are used for each user that slows down an attacker.

Salt usages and associated fingerprints	
Method to hash 'password'	**Fingerprint**
No salt	B109F3BBBC244EB82441917ED06D618B9008DD09B3 BEFD1B5E07394C706A8BB980B1D7785E5976EC049B 46DF5F1326AF5A2EA6D103FD07C95385FFAB0CACBC86
Salt = 'WindowCleaner'	E6F9DCB1D07E5412135120C0257BAA1A27659D41DC7 7FE2DE4C345E23CB973415F8DFDFFF6AA7F0AE0BDD 61560FB028EFEDF2B5422B40E5EE040A0223D16F06F
Salt = 'a0w8hsdfas8ls587uas87'	5AA762E7C83CFF223B5A00ADA939FBD186C4A2CD01 1B0A7FE7AF86B8CA5420C7A47B52AFD2FA6B98B172 22ACF32B3E13F8C436447C36364A5E2BE998416A103A
Salt = '8ash87123klnf9d8dq3w'	8058D43195B1CF2794D012A86AC809BFE73254A82C8C E6C10256D1C46B9F45700D040A6AC6290746058A63E5 0AAF8C87ABCD5C3AA00CDBDB31C10BA6D12A1A7

One way to generate a salt value is using a pseudo-random number generator, as shown in **Figure A6.3** below.

TABLE A6.3

Adding Salt in .NET

```
   private int minSaltSize = 8;
    private int maxSaltSize = 24;
    private int saltSize;
```

```
    private byte[] GetSalt(string input) {
            byte[] data;
            byte[] saltBytes;
            RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
            saltBytes = new byte[saltSize];
            rng.GetNonZeroBytes(saltBytes);
            data = Encoding.UTF8.GetBytes(input);
            byte[] dataWithSaltBytes =
                new byte[data.Length + saltBytes.Length];
            for (int i = 0; i < data.Length; i++)
                dataWithSaltBytes[i] = data[i];
            for (int i = 0; i < saltBytes.Length; i++)
                dataWithSaltBytes[data.Length + i] = saltBytes[i];
            return dataWithSaltBytes;
}
```

Note that a salt value does not need to possess the quality of a cryptographically secure randomness.

Best practices is to use a cryptographically function to create the salt, create a salt value for each hash value, and a minimum value of 128 bits (16 characters). The bits are not costly so don't save a few bits thinking you gain something back in performance instead use a value of 256-bit salt value. It is highly recommended.

Best Practices

Industry leading Cryptographer's are advising that MD5 and SHA-1 should not be used for any applications. The United State FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION (FIPS) specifies seven cryptographic hash algorithms — SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256 are approved for federal use. The code reviewer should consider this standard because the FIPS is also widely adopted by the information technology industry.

The code reviewer should raise a red flag if MD5 and SHA-1 are used and a risk assessment be done to understand why these functions would be used instead of other better-suited hash functions. FIPS does allow that MD5 can be used only when used as part of an approved key transport scheme where no security is provided by the algorithm.

Figure **Figure A6.4** below shows an example function which could implement a generic hash feature for an application.

FIGURE A6.4

```
.NET Agile Code example for hashing
App Code File:
<add key="HashMethod" value="SHA512"/>


C# Code:
   1:  preferredHash =
HashAlgorithm.Create((string)ConfigurationManager.AppSettings["HashMethod"]);
   2:
   3:  hash = computeHash(preferredHash, testString);
   4:
   5:  private string computeHash(HashAlgorithm myHash, string input) {
   6:       byte[] data;
   7:       data = myHash.ComputeHash(Encoding.UTF8.GetBytes(input));
   8:       sb = new StringBuilder();
   9:       for (int i = 0; i < data.Length; i++) {
  10:           sb.Append(data[i].ToString("x2"));
  11:       }
  12:      return sb.ToString();
  13:  }
```

Line 1 lets us get our hashing algorithm we are going to use from the config file. If we use the machine config file our implementation would be server wide instead of application specific.

Line 3 allows us to use the config value and set it according as our choice of hashing function. ComputHash could be SHA-256 or SHA-512.

References

http://valerieaurora.org/hash.html (Lifetimes of cryptographic hash functions)

http://docs.oracle.com/javase/6/docs/api/java/security/SecureRandom.html

http://msdn.microsoft.com/en-us/library/system.security.cryptography.rngcryptoserviceprovider.aspx

http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

Ferguson and Schneier (2003) Practical Cryptography (see Chapter 6; section 6.2 Real Hash Functions)

2.9.5. Reducing the attack surface

The Attack Surface of an application is a description of the entry/exit points, the roles/entitlements of the users, and the sensitivity of the data held within the application. For

example, entry points such as login screens, HTML forms, file upload screens, all introduce a level of risk to the application. Note that the code structure also forms part of the Attack Surface, in that the code checking authentication, or crypto, etc., is exercised by critical functions on the application.

Description

The attack surface of a software environment is a description of the entry points where an attacker can try to manipulate an application, it typically takes the form of a systems diagram where all entry points (interfaces) are pointed out. Michael Howard (at Microsoft) and other researchers have developed a method for measuring the Attack Surface of an application, and to track changes to the Attack Surface over time, called the Relative Attack Surface Quotient (RSQ).

It is assumed that the application Attack Surface is already known, probably through some previous threat modeling exercise, or Architectural Risk Analysis. Therefor the entry and exit points are known, the sensitivity of the data within the application is understood, and the various users of the system, and their entitlements, have been mapped in relation to the functions and data.

From a code review point of view, the aim would be to ensure the change being reviewed is not unnecessarily increasing the Attack Surface. For example, is the code change suddenly using HTTP where only HTTPS was used before? Is the coder deciding to write their own hash function instead of using the pre-existing (and well exercised/tested) central repository of crypto functions? In some development environments the Attack Surface changes can be checked during the design phase if such detail is captured, however at code review the actual implementation is reflected in the code and such Attack Surface exposures can be identified.

You can also build up a picture of the Attack Surface by scanning the application. For web apps you can use a tool like the OWASP Zed Attack Proxy Project (ZAP), Arachni, Skipfish, w3af or one of the many commercial dynamic testing and vulnerability scanning tools or services to crawl your app and map the parts of the application that are accessible over the web. Once you have a map of the Attack Surface, identify the high risk areas, then understand what compensating controls you have in place.

Note that backups of code and data (online, and on offline media) are an important but often ignored part of a system's Attack Surface. Protecting your data and IP by writing secure software and hardening the infrastructure will all be wasted if you hand everything over to bad guys by not protecting your backups.

What to Review

When reviewing code modules from an Attack Surface point of view, some common issues to look out for include:

• Does the code change modify the attack surface? By applying the change to the current Attack Surface of the application does it open new ports or accept new inputs? If it does could the change be done in a way that does not increase the attack surface? If a better implementation exists then that should be recommended, however if there is no way to implement the code without increasing the Attack Surface, make sure the business knows of the increased risk.

• Is the feature unnecessarily using HTTP instead of HTTPS?

• Is the function going to be available to non-authenticated users? If no authentication is necessary for the function to be invoked, then the risk of attackers using the interface is increased. Does the function invoke a backend task that could be used to deny services to other legitimate users?

E.g. if the function writes to a file, or sends an SMS, or causes a CPU intensive calculation, could an attacker write a script to call the function many times per second and prevent legitimate users access to that task?

• Are searches controlled? Search is a risky operation as it typically queries the database for some criteria and returns the results, if attacker can inject SQL into query then they could access more data than intended.

• Is important data stored separately from trivial data (in DB, file storage, etc). Is the change going to allow unauthenticated users to search for publicly available store locations in a database table in the same partition as the username/password table? Should this store location data be put into a different database, or different partition, to reduce the risk to the database information?

• If file uploads are allowed, are they be authenticated? Is there rate limiting? Is there a maximum file size for each upload or aggregate for each user? Does the application restrict the file uploads to certain types of file (by checking MIME data or file suffix). Is the application is going to run virus checking?

• If you have administration users with high privilege, are their actions logged/tracked in such a way that they a) can't erase/modify the log and b) can't deny their actions?

Are there any alarms or monitoring to spot if they are accessing sensitive data that they shouldn't be? This could apply to all types of users, not only administrators.

• Will changes be compatible with existing countermeasures, or security code, or will new code/countermeasures need to be developed?

• Is the change attempting to introduce some non-centralized security code module, instead of re-using or extending an existing security module?

• Is the change adding unnecessary user levels or entitlements that will complicate the attack surface.

• If the change is storing PII or confidential data, is all of the new information absolutely necessary? There is little value in increasing the risk to an application by storing the social

security numbers of millions of people, if the data is never used.

• Does application configuration cause the attack surface to vary greatly depending on configuration settings, and is that configuration simple to use and alert the administrator when the attack surface is being expanded?

• Could the change be done in a different way that would reduce the attack surface, i.e instead of making help items searchable and storing help item text in a database table beside the main username/password store, providing static help text on HTML pages reduces the risk through the 'help' interface.

• Is information stored on the client that should be stored on the server?

References

https://www.owasp.org/index.php/Attack_Surface_Analysis_Cheat_Sheet

https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

http://www.cs.cmu.edu/~wing/publications/Howard-Wing03.pdf

2.10. MISSING FUNCTION LEVEL ACCESS CONTROL

Most web applications verify function level access rights before making that functionality visible in the UI. However, applications need to perform the same access control checks on the server when each function is accessed. If requests are not verified, attackers will be able to forge requests in order to access functionality without proper authorization.

2.10.1.Authorization

Authorization is as important as authentication. Access to application functionality and access to all data should be authorized. For data access authorization, application logic should check if the data belongs to the authenticated user, or if the user should be able to access that data.

Placement of security checks is a vital area of review in an application design. Incorrect placement can render the applied security controls useless, thus it is important to review the application design and determine the correctness of such checks. Many web application designs are based on the concept of Model-View-Controller (MVC) that have a central controller which listens to all incoming request and delegates control to appropriate form/business processing logic. Ultimately the user is rendered with a view. In such a layered design, when there are many entities involved in processing a request, developers can go wrong in placing the security controls in the incorrect place, for example some application developers feel "view" is the right place to have the authorization checks.

Authorization issues cover a wide array of layers in a web application; from the functional authorization of a user to gain access to a particular function of the application at the application layer, to the Database access authorization and least privilege issues at the persistence layer.

In most of the applications the request parameters or the URL's serve as sole factors to determine the processing logic. In such a scenario the elements in the request, which are used

for such identifications, may be subject to manipulation attacks to obtain access to restricted resources or pages in the application.

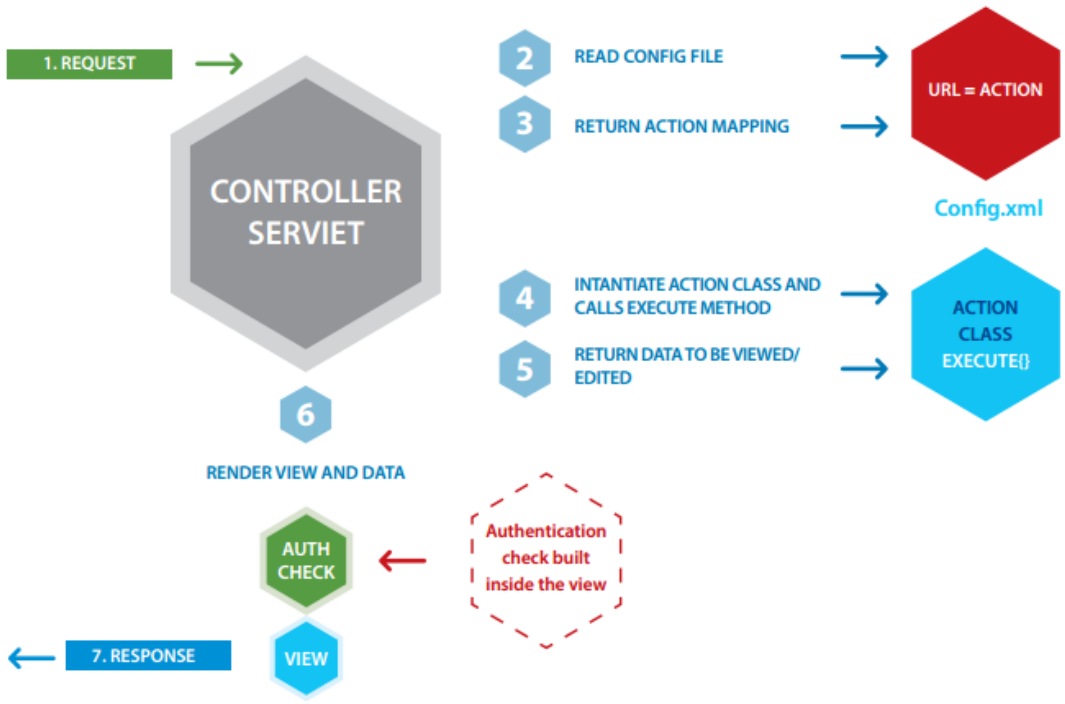
There are two main design methods to implements authorization: Role Base Access Control (RBAC) and Access Control Lists (ACLs). RBAC is used when assigning users to roles, and then roles to permissions. This is a more logical modeling of actual system authorization. It also allows administrators to fine-grain and re-check role-permission assignments, while making sure that every role has the permissions it is supposed to have (and nothing more or less).

Thus assigning users to roles should reduce the chance of human-error. Many web frameworks allow roles to be assigned to logged in users, and custom code can check the session information to authorize functionality based on the current users role.

2.10.2.Description

It seems logical to restrict the users at the page/view level they won't be able to perform any operation in the application. But what if instead of requesting for a page/view an unauthorized user tries to request for an internal action such as to add/modify any data in the application? It will be processed but the resultant view will be denied to the user; because the flaw lies in just having a view based access control in the applications. Much of the business logic processing (for a request) is done before the "view" is executed. So the request to process any action would get processed successfully without authorization.

In an MVC based system given this issue is shown in image X below, where the authentication check is present in the view action.



In this example neither the controller servlet (central processing entity) nor the action classes have any access control checks. If the user requests an internal action such as add user details, without authentication, it will get processed, but the only difference is that the user will be shown an error page as resultant view will be disallowed to the user.

A similar flaw is observed in ASP.NET applications where the developers tend to mix the code for handling POSTBACK's and authentication checks. Usually it is observed that the authentication check in the ASP.NET pages are not applied for POSTBACKs, as indicated below. Here, if an attacker tries to access the page without authentication an error page will be rendered. Instead, if the attacker tries to send an internal POSTBACK request directly without authentication it would succeed.

Unused or undeclared actions or functionality can present in the configuration files. Such configurations that are not exposed (or tested) as valid features in the application expand the attack surface and raise the risk to the application. An unused configuration present in a configuration file is shown in **Figure A7.0**, where the 'TestAction' at the end of the file has been left over from the testing cycle and will be exposed to external users. It is likely this action would not be checked on every release and could expose a vulnerability.

FIGURE A7.0

Example of unused action in an application configuration

```
<mapping>
  <url>/InsecureDesign/action/AddUserDetails</url>
  <action>Action.UserAction</action>
  <success>JSP_WithDesign/Success.jsp</success>
</mapping>

<mapping>
  <url>/InsecureDesign/action/ChangePassword</url>
  <action>Action.ChangePasswordAction</action>
  <success>JSP_WithDesign/Success.jsp</success>
</mapping>

<mapping>
  <url>/InsecureDesign/action/test</url>
  <action>Action.TestAction</action>
  <success>JSP_WithDesign/Success.jsp</success>
</mapping>
```

Another popular feature seen in most of the design frameworks today is data binding, where the request parameters get directly bound to the variables of the corresponding

business/command object. Binding here means that the instance variables of such classes get automatically initialized with the request parameter values based on their names. The issue with this design is that the business objects may have variables that are not dependent on the request parameters. Such variables could be key variables like price, max limit, role etc. having static values or dependent on some server side processing logic. A threat in such scenarios is that an attacker may supply additional parameters in request and try to bind values for unexposed variable of business object class. In this case the attacker can send an additional "price" parameter in the request which binds with the unexposed variable "price" in business object, thereby manipulating business logic.

What to Review

It is imperative to place all validation checks before processing any business logic and in case of ASP.NET applications independent of the POSTBACKs. The security controls like authentication check must be place before processing any request.

The use of filters is recommended when authorization is being implemented in MVC 3 and above as .NET MVC 3 introduced a method in global.asax called RegisterGlobalFilters which can be used to default deny access to URL's in the application.

FIGURE A7.1

```
Registering filters in .NET

public static void RegisterGlobalFilters(GlobalFilterCollection filters)
{
   filters.Add(new HandleErrorAttribute());
   filters.Add(new System.Web.Mvc.AuthorizeAttribute());
}
```

It is recommended when reviewing MVC3/4 .NET to take a look at how authorization is being implemented. The line above, "filters.Add(new System.Web.Mvc.AuthorizeAttribute());" default denies access to any request without a valid session. If this is implemented we may need to provide unauthorized access to certain pages such as a registration page, public welcome page or a login page.

The directive "AllowAnonymous" is used to provide access to public pages with no valid session required. The code may look like this:

FIGURE A7.2

```
Allowing anonymous access to a function in IS

[AllowAnonymous]
    public ActionResult LogMeIn(string returnUrl)
```

When reviewing code for authorization, the following considerations can be checked for:

• Every entry point should be authorized. Every function should be authorized.

• Authorization checks should be efficient, and implemented in a central code base such that it can be applied consistently.

• In cases where authorization fails, a HTTP 403 not authorized page should be returned.

• When using RBAC, there must be some way for the application to report on the currently provisioned users of the system and their associated roles. This allows the business to periodically audit the user access to the system and ensure it is accurate. For example, if a user is provisioned as an admin on the system, then that user changes job to another department, it could be the case that the admin role is no longer appropriate.

• There should be an easy method to change or remove a user's role (in RBAC systems). Adding, modifying or removing a user from a role should result in audit logs.

• For roles that are higher risk, addition, modification and deletion of those roles should involve multiple levels of authorization (e.g. maker/checker), this may be tracked within the application itself, or through some centralized role application. Both the functionality and code of the system controlling roles should be part of the review scope.

• At a design level attempt to keep the range of roles simple. Applications with multiple permission levels/roles often increases the possibility of conflicting permission sets resulting in unanticipated privileges.

• In application architectures with thick clients (i.e. mobile apps or binaries running on a PC) do not attempt to perform any authorization in the client code, as this could be bypassed by an attacker. In browser based applications do not perform any authorization decisions in JavaScript.

• Never base authorization decisions on untrusted data. For example do not use a header, or hidden field, from the client request to determine the level of authorization a user will have, again this can be manipulated by an attacker.

• Follow the principle of 'complete mediation', where authorization is checked at every stage of a function. For example, if an application has four pages to browse through to purchase an item (browse.html, basket.html, inputPayment.html, makePayment.html) then check user authorization at every page, and stage within pages, instead of only performing a check in the first page.

• By default deny access to any page, and then use authorization logic to explicitly allow access based on roles/ACL rules.

• Remove all redundant/test/unexposed business logic configurations from the file

• The business/form/command objects must have only those instance variables that are dependent on the user inputs.

2.11. CROSS-SITE REQUEST FORGERY (CSRF)

A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

2.11.1.Description

CSRF is an attack, which forces an end user to execute unwanted actions on a web application in which they are currently authenticated. With a little help of social engineering (like sending a link via email/chat), an attacker may force the users of a web application to execute actions of the attacker's choosing. A successful CSRF exploit can compromise end user data, and protected functionality, in the case of a normal privileged user. If the targeted end user is the administrator account, this can compromise the entire web application.

The impact of a successful cross-site request forgery attack is limited to the capabilities exposed by the vulnerable application. For example, this attack could result in a transfer of funds, changing a password, or purchasing an item in the user's context. In effect, CSRF attacks are used by an attacker to make a target system perform a function (funds Transfer, form submission etc.) via the target's browser without knowledge of the target user, at least until the unauthorized function has been committed.

CSRF is not the same as XSS (Cross Site Scripting), which forces malicious content to be served by a trusted website to an unsuspecting victim. Cross-Site Request Forgery (CSRF, a.k.a C-SURF or Confused-Deputy) attacks are considered useful if the attacker knows the target is authenticated to a web based system. They only work if the target is logged into the system, and therefore have a small attack footprint. Other logical weaknesses also need to be present such as no transaction authorization required by the user. In effect CSRF attacks are used by an attacker to make a target system perform a function (Funds Transfer, Form submission etc..) via the target's browser without the knowledge of the target user, at least until the unauthorized function has been committed. A primary target is the exploitation of "ease of use" features on web applications (One-click purchase).

Impacts of successful CSRF exploits vary greatly based on the role of the victim. When targeting a normal user, a successful CSRF attack can compromise end-user data and their associated functions. If the targeted end user is an administrator account, a CSRF attack can compromise the entire Web application. The sites that are more likely to be attacked are community Websites (social networking, email) or sites that have high dollar value accounts associated with them (banks, stock brokerages, bill pay services). This attack can happen even if the user is logged into a Web site using strong encryption (HTTPS). Utilizing social engineering, an attacker will embed malicious HTML or JavaScript code into an email or Website to request a specific 'task url'. The task then executes with or without the user's knowledge, either directly or by utilizing a Cross-site Scripting flaw (ex: Samy MySpace Worm).

How They Work

CSRF attacks work by sending a rogue HTTP request from an authenticated user's browser to the application, which then commits a transaction without authorization given by the target user. As long as the user is authenticated and a meaningful HTTP request is sent by the user's browser to

a target application, the application does not know if the origin of the request is a valid transaction or a link clicked by the user (that was, say, in an email) while the user is authenticated to the application. The request will be authenticated as the request from the users browser will automatically include the 'Cookie' header, which is the basis for authentication. So an attacker makes the victim perform actions that they didn't intend to, such as purchase an item. Figure X shows an example an HTTP POST to a ticket vendor to purchase a number of tickets.

FIGURE A8.0

```
HTTP request/response to purchase tickets

POST http://TicketMeister.com/Buy_ticket.htm HTTP/1.1

Host: ticketmeister
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O;) Firefox/1.4.1
Cookie: JSPSESSIONID=34JHURHD894LOP04957HR49I3JE383940123K
ticketId=ATHX1138&to=PO BOX 1198 DUBLIN 2&amount=10&date=11042008
The response of the vendor is to acknowledge the purchase of the tickets:


HTTP/1.0 200 OK

Date: Fri, 02 May 2008 10:01:20 GMT
Server: IBM_HTTP_Server
Content-Type: text/xml;charset=ISO-8859-1
Content-Language: en-US
X-Cache: MISS from app-proxy-2.proxy.ie
Connection: close

<?xml version="1.0" encoding="ISO-8859-1"?>
<pge_data> Ticket Purchased, Thank you for your custom.
</pge_data>
```

What to Review

This issue is simple to detect, but there may be compensating controls around the functionality of the application which may alert the user to a CSRF attempt. As long as the application accepts a well formed HTTP request and the request adheres to some business logic of the application CSRF shall work.

By checking the page rendering we need to see if any unique identifiers are appended to the links rendered by the application in the user's browser. If there is no unique identifier relating to each HTTP request to tie a HTTP request to the user, we are vulnerable. Session ID is not enough, as the session ID shall be sent automatically if a user clicks on a rogue link, as the user is

already authenticated.

Prevention Measures That Do NOT Work

E
2

UNSUCCESSFUL COUNTERMEASURES FOR CSRF ATTACKS

Measure	Description
Using a Secret Cookie	Remember that all cookies, even the secret ones, will be submitted with every request. All authentication tokens will be submitted regardless of whether or not the end-user was tricked into submitting the request. Furthermore, session identifiers are simply used by the application container to associate the request with a specific session object. The session identifier does not verify that the end-user intended to submit the request.
Only Accepting POST Requests	Applications can be developed to only accept POST requests for the execution of business logic. The misconception is that since the attacker cannot construct a malicious link, a CSRF attack cannot be executed. Unfortunately, this logic is incorrect. There are numerous methods in which an attacker can trick a victim into submitting a forged POST request, such as a simple form hosted in an attacker's website with hidden values. This form can be triggered automatically by JavaScript or can be triggered by the victim who thinks the form will do something else.
Salt = 'a0w8hsdfas8ls587uas87'	Multi-Step transactions are not an adequate prevention of CSRF. As long as an attacker can predict or deduce each step of the completed transaction, then CSRF is possible.
URL Rewriting	This might be seen as a useful CSRF prevention technique as the attacker cannot guess the victim's session ID. However, the user's credential is exposed over the URL.

Preventing CSRF

Checking if the request has a valid session cookie is not enough, the application needs to have a unique identifier assocaited with every HTTP request sent to the application. CSRF requests (from an attackers e-mail) will not have this valid unique identifier. The reason CSRF requests won't have this unique request identifier is the unique ID is rendered as a hidden field, or within the URL, and is appended to the HTTP request once a link/button press is selected. The attacker will have no knowledge of this unique ID, as it is random and rendered dynamically per link, per page.

Application logic to prevent CSRF then include:

1. A list of unique IDs is compiled prior to delivering the page to the user. The list contains all valid unique IDs generated for all links on a given page. The unique ID could be derived from a secure random generator such as SecureRandom for J2EE and could be stored in the session or another centralized cache.

2. A unique ID is appended to each link/form on the requested page prior to being displayed to the user.

3. The application checks if the unique ID passed with the HTTP request is valid for a given request. If the unique ID passed with the HTTP request is valid for a given request.

4. If the unique ID is not present, terminate the user session and display an error to the user.

General Recommendation: Synchronizer Token Pattern

In order to facilitate a "transparent but visible" CSRF solution, developers are encouraged to adopt the Synchronizer Token Pattern http://www.corej2eepatterns.com/Design/PresoDesign.htm) [add reference]. The synchronizer token pattern requires the generating of random "challenge" tokens that are associated with the user's current session. These challenge tokens are then inserted within the HTML forms and links associated with sensitive server-side operations. When the user wishes to invoke these sensitive operations, the HTTP request should include this challenge token. It is then the responsibility of the server application to verify the existence and correctness of this token. By including a challenge token with each request, the developer has a strong control to verify that the user actually intended to submit the desired requests. Inclusion of a required security token in HTTP requests associated with sensitive business functions helps mitigate CSRF attacks as successful exploitation assumes the attacker knows the randomly generated token for the target victim's session. This is analogous to the attacker being able to guess the target victim's session identifier. The following synopsis describes a general approach to incorporate challenge tokens within the request.

When a Web application formulates a request (by generating a link or form that causes a request when submitted or clicked by the user), the application should include a hidden input parameter with a common name such as "CSRFToken". The value of this token must be randomly generated such that it cannot be guessed by an attacker. Consider leveraging the java.security.SecureRandom class for Java applications to generate a sufficiently long random token. Alternative generation algorithms include the use of 256-bit BASE64 encoded hashes. Developers that choose this generation algorithm must make sure that there is randomness and uniqueness utilized in the data that is hashed to generate the random token.

FIGURE A8.1

```
Example showing CSRF token in hidden field


<form action="/transfer.do" method="post">
<input type="hidden" name="CSRFToken"
value="OWY4NmQwODE4ODRjN2Q2NTlhMmZlYWE...
wYzU1YWQwMTVhM2JmNGYxYjJiMGI4MjJjZDE1ZDZ...
MGYwMGEwOA==">
…
</form>
```

Depending on the risk level of the product, it may only generate this token once for the current session. After initial generation of this token, the value is stored in the session and is utilized for each subsequent request until the session expires. When a request is issued by the end-user, the server-side component must verify the existence and validity of the token in the request as compared to the token found in the session. If the token was not found within the request or the value provided does not match the value within the session, then the request should be aborted, token should be reset and the event logged as a potential CSRF attack in progress.

To further enhance the security of this proposed design, consider randomizing the CSRF token parameter name and or value for each request. Implementing this approach results in the generation of per-request tokens as opposed to per-session tokens. Note, however, that this may result in usability concerns. For example, the "Back" button browser capability is often hindered as the previous page may contain a token that is no longer valid. Interaction with this previous page will result in a CSRF false positive security event at the server. Regardless of the approach taken, developers are encouraged to protect the CSRF token the same way they protect authenticated session identifiers, such as the use of SSLv3/TLS.

Disclosure of Token in URL

Many implementations of this control include the challenge token in GET (URL) requests as well as POST requests. This is often implemented as a result of sensitive server-side operations being invoked as a result of embedded links in the page or other general design patterns. These patterns are often implemented without knowledge of CSRF and an understanding of CSRF prevention design strategies. While this control does help mitigate the risk of CSRF attacks, the unique per-session token is being exposed for GET requests. CSRF tokens in GET requests are potentially leaked at several locations: browser history, HTTP log files, network appliances that make a point to log the first line of an HTTP request, and Referer headers if the protected site links to an external site.

In the latter case (leaked CSRF token due to the Referer header being parsed by a linked site), it is trivially easy for the linked site to launch a CSRF attack on the protected site, and they will be able to target this attack very effectively, since the Referer header tells them the site as well as the CSRF token. The attack could be run entirely from javascript, so that a simple addition of a script tag to the HTML of a site can launch an attack (whether on an originally malicious site or on a hacked site). This attack scenario is easy to prevent, the referer will be omitted if the origin of the request is HTTPS. Therefore this attack does not affect web applications that are HTTPS only.

The ideal solution is to only include the CSRF token in POST requests and modify server-side actions that have state changing affect to only respond to POST requests. This is in fact what the RFC 2616 requires for GET requests. If sensitive server-side actions are guaranteed to only ever respond to POST requests, then there is no need to include the token in GET requests.

Viewstate (ASP.NET)

ASP.NET has an option to maintain your ViewState. The ViewState indicates the status of a page when submitted to the server. The status is defined through a hidden field placed on each page with a <form runat="server"> control. Viewstate can be used as a CSRF defense, as it is difficult for an attacker to forge a valid Viewstate. It is not impossible to forge a valid Viewstate since it is feasible that parameter values could be obtained or guessed by the attacker. However, if the current session ID is added to the ViewState, it then makes each Viewstate unique, and thus immune to CSRF.

To use the ViewStateUserKey property within the Viewstate to protect against spoofed post backs add the following in the OnInit virtual method of the page-derived class (This property must be set in the Page.Init event)

FIGURE A8.2

Example setting ViewStateUserKey in ASP.NET

```
protected override OnInit(EventArgs e) {
     base.OnInit(e);
     if (User.Identity.IsAuthenticated)
     ViewStateUserKey = Session.SessionID; }
```

To key the Viewstate to an individual using a unique value of your choice use "(Page.ViewStateUserKey)". This must be applied in Page_Init because the key has to be provided to ASP.NET before Viewstate is loaded. This option has been available since ASP.NET 1.1. However, there are limitations on this mechanism. Such as, ViewState MACs are only checked on POSTback, so any other application requests not using postbacks will happily allow CSRF.

Double Submit Cookies

Double submitting cookies is defined as sending a random value in both a cookie and as a request parameter, with the server verifying if the cookie value and request value are equal.

When a user authenticates to a site, the site should generate a (cryptographically strong) pseudorandom value and set it as a cookie on the user's machine separate from the session id. The site does not have to save this value in any way. The site should then require every sensitive submission to include this random value as a hidden form value (or other request parameter) and also as a cookie value. An attacker cannot read any data sent from the server or modify cookie values, per the same-origin policy. This means that while an attacker can send any value he wants with a malicious CSRF request, the attacker will be unable to modify or read the value stored in the cookie. Since the cookie value and the request parameter or form value must be the same, the attacker will be unable to successfully submit a form unless he is able to guess the

random CSRF value.

Direct Web Remoting (DWR) Java library version 2.0 has CSRF protection built in as it implements the double cookie submission transparently. [add reference]

The above CSRF prevents rely on the use of a unique token and the Same-Origin Policy to prevent CSRF by maintaining a secret token to authenticate requests. The following methods can prevent CSRF by relying upon similar rules that CSRF exploits can never break.

Checking The Referer Header

Although it is trivial to spoof the referer header on your own browser, it is impossible to do so in a CSRF attack. Checking the referer is a commonly used method of preventing CSRF on embedded network devices because it does not require a per-user state. This makes a referer a useful method of CSRF prevention when memory is scarce. This method of CSRF mitigation is also commonly used with unauthenticated requests, such as requests made prior to establishing a session state which is required to keep track of a synchronization token.

However, checking the referer is considered to be a weaker from of CSRF protection. For example, open redirect vulnerabilities can be used to exploit GET-based requests that are protected with a referer check and some organizations or browser tools remove referrer headers as a form of data protection. There are also common implementation mistakes with referer checks. For example if the CSRF attack originates from an HTTPS domain then the referer will be omitted. In this case the lack of a referer should be considered to be an attack when the request is performing a state change. Also note that the attacker has limited influence over the referer. For example, if the victim's domain is "site.com" then an attacker have the CSRF exploit originate from "site.com.attacker.com" which may fool a broken referer check implementation. XSS can be used to bypass a referer check.

In short, referer checking is a reasonable form of CSRF intrusion detection and prevention even though it is not a complete protection. Referer checking can detect some attacks but not stop all attacks. For example, if the HTTP referrer is from a different domain and you are expecting requests from your domain only, you can safely block that request.

Checking The Origin Header

The Origin HTTP Header standard [add reference] was introduced as a method of defending against CSRF and other Cross-Domain attacks. Unlike the referer, the origin will be present in HTTP request that originates from an HTTPS URL. If the origin header is present, then it should be checked for consistency.

Challenge-Response

Challenge-Response is another defense option for CSRF. As mentioned before it is typically used when the functionality being invoked is high risk. While challenge-response is a very strong defense to CSRF (assuming proper implementation), it does impact user experience. For applications in need of high security, tokens (transparent) and challenge-response should be used on high risk functions.

The following are some examples of challenge-response options:

- CAPTCHA
- Re-Authentication (password)
- One-time Token

No Cross-Site Scripting (XSS) Vulnerabilities

Cross-Site Scripting is not necessary for CSRF to work. However, any cross-site scripting vulnerability can be used to defeat token, Double-Submit cookie, referer and origin based CSRF defenses. This is because an XSS payload can simply read any page on the site using a XMLHttpRequest and obtain the generated token from the response, and include that token with a forged request. This technique is exactly how the MySpace (Samy) worm defeated MySpace's anti CSRF defenses in 2005, which enabled the worm to propagate. XSS cannot defeat challenge-response defenses such as Captcha, re-authentication or one-time passwords. It is imperative that no XSS vulnerabilities are present to ensure that CSRF defenses can't be circumvented.

References

TBD

2.12. USING COMPONENTS WITH KNOWN VULNERABILITIES

Components, such as libraries, frameworks, and other software modules, almost always run with full privileges. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications using components with known vulnerabilities may undermine application defenses and enable a range of possible attacks and impacts.

2.12.1.Description

Today it would be rare for an application or software component to be developed without the re-use of some open source or paid-for library or framework. This makes a lot of sense as these frameworks and libraries are already developed and working, and have had a good degree of testing applied. However these third party components can also be a source of security vulnerabilities when an attacker finds a flaw in the components code, in fact this flaw has added attraction since the attacker knows the exploit will work on everyone who is using the component.

This issue has matured to such a state that flaws/exploits for popular frameworks, libraries and operating systems are sold on underground markets for large sums of money.

2.12.2.What to Review

There is really no code to review for this topic, unless your organization has taken it upon itself to review the code of the component (assuming its open source and not a closed source third party library), in which case the code review would be similar to any other audit review. However code review can be used within the larger company-wide tracking or audit mechanisms that lets the organization know what third party code it is using.

Regardless of the size of company, the use of third party components, and their versions, should be tracked to ensure the organization can be alerted when any security vulnerabilities are

flagged. For smaller companies with 1 or 2 products this tracking could be as easy as a spreadsheet or wiki page, however for larger companies with 100s of applications or products, the task of tracking developer use of third party frameworks and libraries is equally as large as the risk posed by those libraries.

If a company has 20 products and each of those products use 5 or 6 third party components (e.g. Apache web servers, OpenSSL crypto libraries, Java libraries for regex and DB interactions, etc.) that leaves the company with over 100 external sources where security vulnerabilities can come from. If the company suddenly hears of a heartbleed type vulnerability, it has to be able to react and upgrade those affected applications, or take other countermeasures, to protect itself and its customers.

Controlling the Ingredients

One method used by larger companies to limit their exposure to third party vulnerabilities is to control which libraries can be used by their developers. For example they could specify that developers should use a certain version of OpenSSL as the crypto library instead of other options.

This allows management and risk controllers to know their risk profile to vulnerabilities on the market, if a bug appears in bouncycastle, they know they are not exposed (i.e. some developer didn't use bouncycastle on one of the products, because it's not on the list of crypto libraries to use). On the other hand, if there is a bug in OpenSSL, all their eggs are in that basket and they need to upgrade immediately.

There will obviously be technical challenges to limiting the choices of third party components, and such a policy could be unpopular with developers who'll want to use the latest and greatest framework, but the first step to securing a product is knowing what ingredients you've made it with.

How can such a policy be tracked or enforced? At some point the library or framework, in the form of .dll/.so or as source code, will be integrated into the codeline.

Such integrations should be subject to code review, and as a task of this code review the reviewer can check:

1. The library is one that can be used in the product suite (or maybe is already used and the developer is simply unaware, in which case the review should be rejected and the original integration used)

2. Any tracking or auditing software (even a basic spread sheet) is updated to reflect that the product is using the third party library. This allows for rapid remediation if a vulnerability appears, meaning the product will be patched.

SledgeHammer to Crack a Nut

One last responsibility of the reviewer will be to ensure the correct third party library is being used for the functionality needed. Many libraries come with vast amounts of functionality which may not be used. For example, is the developer including a library to perform their regex's, but which also will include other functionality not used/needed by the application?

This increases the attack surface of the application and can cause unexpected behavior when that extra code opens a port and communicates to the internet.

If the reviewer thinks too much functionality/code is being introduced they can advise to turn off non-used functionality, or better still find a way to not include that functionality in the product (e.g. by stripping out code, or hardcoding branches so unused functions are never used).

References

- TBD

2.13. UNVALIDATED REDIRECTS AND FORWARDS

Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.

2.13.1.Description

Unvalidated redirects and forwards are possible when a web application accepts untrusted input that could cause the web application to redirect the request to a URL contained within untrusted input. By modifying untrusted URL input to a site, an attacker may successfully launch a phishing scam and steal user credentials.

As the server name in the modified link is identical to the original site, phishing attempts may have a more trustworthy appearance. Invalidated redirect and forward attacks can also be used to maliciously craft a URL that would pass the application's access control check and then forward the attacker to privileged functions that they would normally not be able to access.

Redirects

Redirect functionality on a web site allows a user's browser to be told to go to a different page on the site. This can be done to improve user interface or track how users are navigating the site.

To provide the redirect functionality a site may have a specific URL to perform the redirect:

- http://www.example.com/utility/redirect.cgi

This page will take a parameter (from URL or POST body) of 'URL' and will send back a message to the user's browser to go to that page, for example:

- http://www.example.com/utility/redirect.cgi?URL=http://www.example.com/viewtxn.html

However this can be abused as an attacker can attempt to make a valid user click on a link that appears to be for www.example.com but which will invoke the redirect functionality on example.com to cause the users browser to go to a malicious site (one that could look like example.com and trick the user into entering sensitive or authentication information:

• http://www.example.com/utiltiy/redirect cgi?URL=http://attacker.com/fakelogin.html

Forwards

Forwards are similar to redirects however the new page is not retrieved by the users browser (as occurred with the redirect) but instead the server framework will obtain the forwarded page and return it to the users browser. This is achieved by 'forward' commands within Java frameworks (e.g. Struts) or 'Server.Transfer' in .NET

As the forward is performed by the server framework itself, it limits the range of URLs the attacker can exploit to the current web site (i.e. attacker cannot 'forward' to attacker.com), however this attack can be used to bypass access controls. For example, where a site sends the forwarded page in the response:

• If purchasing, forward to 'purchase.do'

• If cancelling, forward to 'cancelled.do'

This will then be passed as a parameter to the web site:

• http://www.example.com/txn/acceptpayment.html?FWD=purchase

If instead an attacker used the forward to attempt to access to a different page within the web site, e.g. admin.do, then they may access pages that they are not authorized to view, because authorization is being applied on the 'acceptpayment' page, instead of the forwarded page.

2.13.2. What to Review

If any part of the URL being forwarded, or redirected, to is based on user input, then the site could be at risk.

Redirects

The following examples demonstrate unsafe redirect and forward code. The following Java code receives the URL from the 'url' parameter and redirects to that URL.

FIGURE A10.0

```
Unsafe redirect code in Java

response.sendRedirect(request.getParameter("url"));
```

The following PHP code obtains a URL from the query string and then redirects the user to that URL.

FIGURE A10.1

```
Unsafe redirect code in PHP

$redirect_url = $_GET['url'];
 header("Location: " . $redirect_url);
```

A similar example of C# .NET Vulnerable Code:

FIGURE A10.2

```
Unsafe redirect code in C#.NET

string url = request.QueryString["url"];
 Response.Redirect(url);
```

The above code is vulnerable to an attack if no validation or extra method controls are applied to verify the certainty of the URL. This vulnerability could be used as part of a phishing scam by redirecting users to a malicious site. If user input has to be used as part of the URL to be used, then apply strict validation to the input, ensuring it cannot be used for purposes other than intended.

Note that vulnerable code does not need to explicitly call a 'redirect' function, but instead could directly modify the response to cause the client browser to go to the redirected page. Code to look for is shown in **table 23**.

23

REDIRECT RISKS	
Method of Redirection	**Description**
Redirect Response (note 301 and 307 responses will also cause a redirect)	HTTP/1.1 302 Found Location: http://www.attacker.com/page.html
Meta Tag	<html><head> <meta http-equiv="Refresh" content="0;url=http://attacker.com/page.html">
JavaScript	<script type="text/javascript"> <!— window.location="http://attacker.com/page.html" //-->
Refresh Header	HTTP/1.1 200 OK Refresh=0; url=http://attacker.com/page.html

Where an attacker has posted a redirecting URL on a forum, or sends in an e-mail, the web site can check the referer header to ensure the user is coming from a page within the site,

although this countermeasure will not apply if the malicious URL is contained within the site itself.

Consider creating a whitelist of URLs or options that a redirect is allowed to go to, or deny the ability for the user input to determine the scheme or hostname of the redirect. A site could also encode (or encrypt) the URL value to be redirected to such that an attacker cannot easily create a malicious URL parameter that, when unencoded (or unencypted), will be considered valid.

Forwards

The countermeasure for forwards is to either whitelist the range of pages that can be forwarded to (similar to redirects) and to enforce authentication on the forwarded page as well as the forwarding page. This means that even if an attacker manages to force a forward to a page they should not have access to, the authentication check on the forwarded page will deny them access.

Note on J2EE

There is a noted flaw related to the "sendRedirect" method in J2EE applications. For example:

- **response**.sendRedirect("home.html");

This method is used to send a redirection response to the user who then gets redirected to the desired web component whose URL is passed an argument to the method. One such misconception is that execution flow in the Servlet/JSP page that is redirecting the user stops after a call to this method. Note that if there is code present after the 'If' condition it will be executed.

The fact that execution of a servlet or JSP continues even after sendRedirect() method, also applies to Forward method of the RequestDispatcher Class. However, <jsp:forward> tag is an exception, it is observed that the execution flow stops after the use of <jsp:forward> tag.

After issue a redirect or forward, terminate code flow using a "return" statement.

References

• OWASP Article on Open Redirects https://www.owasp.org/index.php/Open_redirect

• CWE Entry 601 on Open Redirects http://cwe.mitre.org/data/definitions/601.html

• WASC Article on URL Redirector Abuse
http://projects.webappsec.org/w/page/13246981/URL%20Redirector%20Abuse

• Google blog article on the dangers of open redirects
http://googlewebmastercentral.blogspot.com/2009/01/open-redirect-urls-is-your-site-being.html

• Preventing Open Redirection Attacks (C#)
http://www.asp.net/mvc/tutorials/security/preventing-open-redirection-attacks

AX - General

Not all vulnerabilities and associated advice fits into the OWASP Top 10, hence this section covers technologies that are not specific to the Top 10 categories.

2.14. HTML5

HTML5 was created to replace HTLML4, XHTML and the HTML DOM Level 2. The main purpose of this new standard is to provide dynamic content without the use of extra proprietary client side plugins. This allows designers and developers to create exceptional sites providing a great user experience without having to install any additional plug-ins into the browser.

2.14.1.Description

Ideally users should have the latest web browser installed but this does not happens as regularly as security experts advice, therefore the website should implement 2 layer controls, one layer independent from browser type, second, as an additional control.

2.14.2.What to Review

Web Messaging

Web Messaging (also known as Cross Domain Messaging) provides a means of messaging between documents from different origins in a way that is generally safer than the multiple hacks used in the past to accomplish this task. The communication API is as follows:

FIGURE A10.3

HTML5 Web Messaging API

```
otherWindow.postMessage(message, targetOrigin, [transfer]);
```

However, there are still some recommendations to keep in mind:

• When posting a message, explicitly state the expected origin as the second argument to 'postMessage' rather than '*' in order to prevent sending the message to an unknown origin after a redirect or some other means of the target window's origin changing.

• The receiving page should always:

o Check the 'origin' attribute of the sender to verify the data is originating from the expected location.

o Perform input validation on the 'data' attribute of the event to ensure that it's in the desired format.

• Don't assume you have control over the 'data' attribute. A single Cross Site Scripting flaw in the sending page allows an attacker to send messages of any given format.

• Both pages should only interpret the exchanged messages as 'data'. Never evaluate passed messages as code (e.g. via 'eval()') or insert it to a page DOM (e.g. via 'innerHTML'), as that would create a DOM-based XSS vulnerability.

• To assign the data value to an element, instead of using an insecure method like 'element.innerHTML = data', use the safer option: 'element.textContent = data;'

• Check the origin properly exactly to match the FQDN(s) you expect. Note that the following code: ' if(message.orgin.indexOf(".owasp.org")!=-1) { /* ... */ }' is very insecure and will not have the desired behavior as 'www.owasp.org.attacker.com' will match.

• If you need to embed external content/untrusted gadgets and allow user-controlled scripts (which is highly discouraged), consider using a JavaScript rewriting framework such as Google's Caja or check the information on sandboxed frames.

Cross Origin Resource Sharing

Cross Origin Resource Sharing or CORS is a mechanism that enables a web browser to perform "cross-domain" requests using the XMLHttpRequest L2 API in a controlled manner. In the past, the XMLHttpRequest L1 API only allowed requests to be sent within the same origin as it was restricted by the same origin policy.

Cross-Origin requests have an Origin header that identifies the domain initiating the request and is automatically included by the browser in the request sent to the server. CORS defines the protocol between a web browser and a server that will determine whether a cross-origin request is allowed. In order to accomplish this goal, there are HTTP headers that provide information on the messaging context including: Origin, Access-Control-Request-Method, Access-Control-Request-Headers, Access-Control-Allow-Origin, Access-Control-Allow-Credentials, Access-Control-Allow-Methods, Access-Control-Allow-Headers.

The CORS specification mandates that for non simple requests, such as requests other than GET or POST or requests that uses credentials, a pre-flight OPTIONS request must be sent in advance to check if the type of request will have a negative impact on the data. The pre-flight request checks the methods, headers allowed by the server, and if credentials are permitted, based on the result of the OPTIONS request, the browser decides whether the request is allowed or not. More information on CORS requests can be found at reference [x] and [y].

Items to note when reviewing code related to CORS includes:

• Ensure that URLs responding with 'Access-Control-Allow-Origin: *' do not include any sensitive content or information that might aid attacker in further attacks. Use the 'Access-Control-Allow-Origin' header only on chosen URLs that need to be accessed cross-domain. Don't use the header for the whole domain.

• Allow only selected, trusted domains in the 'Access-Control-Allow-Origin' header. Prefer whitelisting domains over blacklisting or allowing any domain (do not use '*' wildcard nor blindly return the 'Origin' header content without any checks).

• Keep in mind that CORS does not prevent the requested data from going to an unauthenticated

location. It's still important for the server to perform usual Cross-Site Request Forgery prevention.

• While the RFC recommends a pre-flight request with the 'OPTIONS' verb, current implementations might not perform this request, so it's important that "ordinary" ('GET' and 'POST') requests perform any access control necessary.

• Discard requests received over plain HTTP with HTTPS origins to prevent mixed content bugs.

• Don't rely only on the Origin header for Access Control checks. Browser always sends this header in CORS requests, but may be spoofed outside the browser. Application-level protocols should be used to protect sensitive data.

WebSockets

Traditionally the HTTP protocol only allows one request/response per TCP connection. Asynchronous JavaScript and XML (AJAX) allows clients to send and receive data asynchronously (in the background without a page refresh) to the server, however, AJAX requires the client to initiate the requests and wait for the server responses (half-duplex). HTML5 WebSockets allow the client/server to create a 'full-duplex' (two-way) communication channels, allowing the client and server to truly communicate asynchronously. WebSockets conduct their initial 'upgrade' handshake over HTTP and from then on all communication is carried out over TCP channels.

The **following** is sample code of an application using Web Sockets:

FIGURE A10.4

```
HTML5 Web Sockets implementation

[Constructor(in DOMString url, optional in DOMString protocol)]
 interface WebSocket
 { readonly attribute DOMString URL;
 // ready state const unsigned short CONNECTING = 0;
 const unsigned short OPEN = 1;
 const unsigned short CLOSED = 2;
 readonly attribute unsigned short readyState;
 readonly attribute unsigned long bufferedAmount;
 // networking
 attribute Function onopen;
 attribute Function onmessage;
 attribute Function onclose;
 boolean send(in DOMString data);
 void close();
 };
 WebSocket implements EventTarget;
```

```
var myWebSocket = new WebSocket("ws://www.websockets.org");

myWebSocket.onopen = function(evt) { alert("Connection open ..."); };
myWebSocket.onmessage = function(evt) { alert( "Received Message: " +
evt.data); };
myWebSocket.onclose = function(evt) { alert("Connection closed."); };
```

When reviewing code implementing websockets, the following items should be taken into consideration:

• Drop backward compatibility in implemented client/servers and use only protocol versions above hybi-00. Popular Hixie-76 version (hiby-00) and older are outdated and insecure.

• The recommended version supported in latest versions of all current browsers is rfc6455 RFC 6455 (supported by Firefox 11+, Chrome 16+, Safari 6, Opera 12.50, and IE10).

• While it's relatively easy to tunnel TCP services through WebSockets (e.g. VNC, FTP), doing so enables access to these tunneled services for the in-browser attacker in case of a Cross Site Scripting attack. These services might also be called directly from a malicious page or program.

• The protocol doesn't handle authorization and/or authentication. Application-level protocols should handle that separately in case sensitive data is being transferred.

• Process the messages received by the websocket as data. Don't try to assign it directly to the DOM nor evaluate as code. If the response is JSON, never use the insecure eval() function; use the safe option JSON.parse() instead.

• Endpoints exposed through the 'ws://' protocol are easily reversible to plain text. Only 'wss://' (WebSockets over SSL/TLS) should be used for protection against Man-In-The-Middle attacks.

• Spoofing the client is possible outside a browser, so the WebSockets server should be able to handle incorrect/malicious input. Always validate input coming from the remote site, as it might have been altered.

• When implementing servers, check the 'Origin:' header in the Websockets handshake. Though it might be spoofed outside a browser, browsers always add the Origin of the page that initiated the Websockets connection.

• As a WebSockets client in a browser is accessible through JavaScript calls, all Websockets communication can be spoofed or hijacked through [[Cross Site Scripting Flaw|Cross Site Scripting]]. Always validate data coming through a WebSockets connection.

Server-Sent Events

Server sent events seem similar to WebSockets, however they do not use a special protocol (they re-used HTTP) and they allow the client browser to solely listen for updates (messages)

from the server, thereby removing the need for the client to send any polling or other messages up to the server. More information on server sent events can be found at [x].

When reviewing code that is handling server sent events, items to keep in mind are:

- Validate URLs passed to the 'EventSource' constructor, even though only same-origin URLs are allowed.
- As mentioned before, process the messages ('event.data') as data and never evaluate the content as HTML or script code.
- Always check the origin attribute of the message ('event.origin') to ensure the message is coming from a trusted domain. Use a whitelist.

2.15. SAME ORIGIN POLICY

Same Origin Policy (SOP), also called Single Origin Policy is a part of web application security model. Same Origin Policy has vulnerabilities that the code reviewer needs to take into consideration. SOP covers three main areas of web development, Trust, Authority, and Policy. Same Origin Policy is made of the combination of three components (Scheme, Hostname and Port).

2.15.1.Description

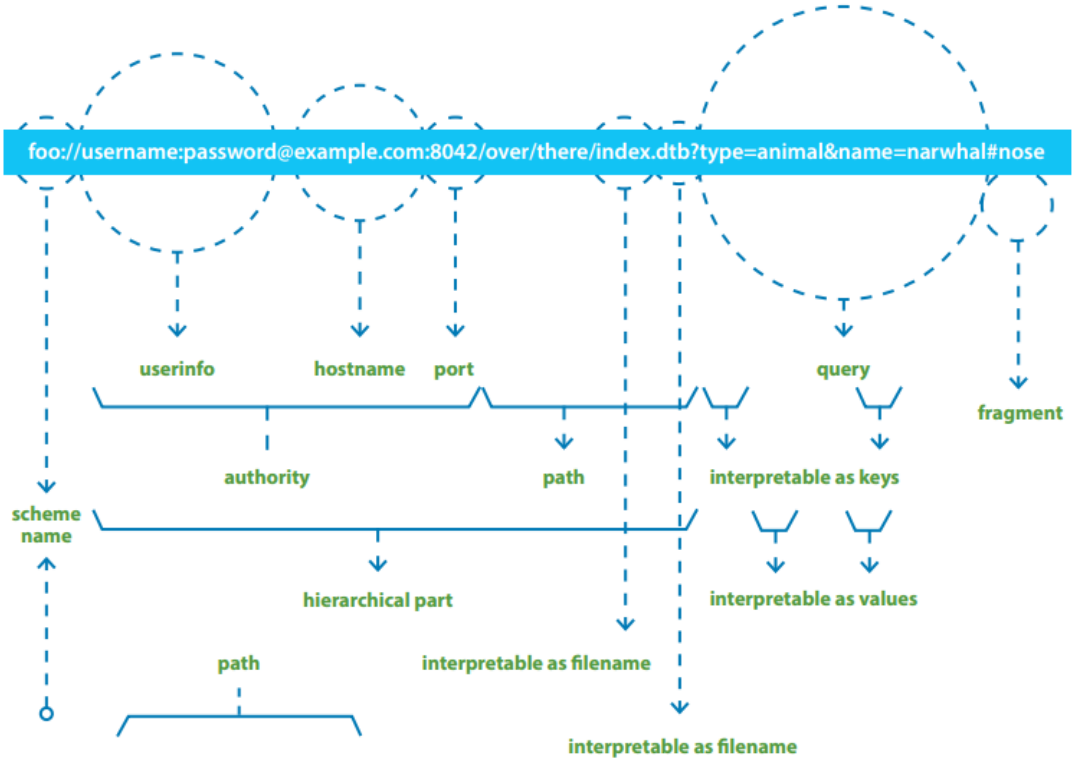
Internet Explorer has two major exceptions when it comes to same origin policy:

1. Trust Zones: if both domains are in highly trusted zone e.g, corporate domains, then the same origin limitations are not applied.

2. Port: IE doesn't include port into Same Origin components, therefore http://yourcompany.com:81/index.html and http://yourcompany.com/index.html are considered from same origin and no restrictions are applied.

These exceptions are non-standard and not supported in any of other browser but would be helpful if developing an app for Windows RT (or) IE based web application.

The following figure displays the various parts of the URL:



• foo://username:password@example.com:8042/over/there/index.dtb?type=animal&name =narwhal#nose

2.15.2.What to Review

• If application allows user-supplied data in the URL then the code reviewer needs to make sure the path, query or Fragment Id Code data is validated.

• Make sure user-supplied scheme name or authority section has good input validation. This is a major code injection and phishing risk. Only permit prefixes needed by the application. Do not use blacklisting. Code reviewer should make sure only whitelisting is used for validation.

• Make sure authority section should only contain alphanumerics, "-", and "." And be followed by "/", "?","#". The risk here an IDN homograph attack.

• Code reviewer needs to make sure the programmer is not assuming default behavior because the programmers browser properly escapes a particular character or browser standard says the character will be escaped properly before allowing any URL-derived values are put inside a database query or the URL is echoed back to the user.

• Resources with a MIME type of image/png are treated as images and resources with MIME type of text/html are treated as HTML documents. Web applications can limit that content's authority by restricting its MIME type. For example, serving user-generated content as image/png is less risky than serving user-generated content as text/html.s

• Privileges on document and resources should grant or withhold privileges from origins as a

whole (rather than discriminating between individual documents within an origin). Withholding privileges is ineffective because the document without the privilege can usually obtain the privilege anyway because SOP does not isolate documents within an origin.

2.16. REVIEWING LOGGING CODE

Applications log messages of varying intensity and to varying sinks. Many logging APIs allow you to set the granularity of log message from a state of logging nearly all messages at level 'trace' or 'debug' to only logging the most important messages at level 'critical'. Where the log message is written to is also a consideration, sometimes it can be written to a local file, other times to a database log table, or it could be written over a network link to a central logging server.

The volume of logging has to be controlled since the act of writing messages to the log uses CPU cycles, thus writing every small detail to a log will use up more resources (CPU, network bandwidth, disk space). Couple that with the fact that the logs have to be parsed or interpreted by a tool or human in order for them to be useful, the consumer of the log could have to parse through thousands of lines to find a message of consequence.

2.16.1.Description

Logs can vary by type; for example a log may simply contain application state or process data, allowing support or development track what the system is doing when a bug has occurred. Other logs may be specific to security, only logging important information that a central security system will have interest in. Further logs could be used for business purposes, such as billing.

Application logs can be powerful as the application business logic has the most information about the user (e.g. identity, roles, permissions) and the context of the event (target, action, outcomes), and often this data is not available to either infrastructure devices, or even closely-related applications. Application logging is an important feature of a production system, especially to support personnel and auditors, however it is often forgotten and is rarely described in sufficient detail in design/requirement documentation. The level and content of security monitoring, alerting and reporting needs to be set during the requirements and design stage of projects, and should be proportionate to the information security risks. Application logging should also be consistent within the application, consistent across an organization's application portfolio and use industry standards where relevant, so the logged event data can be consumed, correlated, analyzed and managed by a wide variety of systems.

All types of applications may send event data to remote systems, either directly over a network connection, or asynchronously though a daily/weekly/monthly secure copy of the log to some centralized log collection and management system (e.g. SIEM or SEM) or another application elsewhere.

If the information in the log is important, and could possibly be used for legal matters, consider how the source (log) can be verified, and how integrity and non-repudiation can be enforced. Log data, temporary debug logs, and backups/copies/extractions, must not be destroyed before the duration of the required data retention period, and must not be kept beyond this time. Legal,

regulatory and contractual obligations may impact on these periods.

Server applications commonly write event log data to the file system or a database (SQL or NoSQL), however logging could be required on client devices such as applications installed on desktops and on mobile devices may use local storage and local databases. Consider how this client logging data is transferred to the server.

2.16.2. What to Review

When reviewing code modules from a logging point of view, some common issues to look out for include:

• When using the file system, it is preferable to use a separate partition than those used by the operating system, other application files and user generated content

• For file-based logs, apply strict permissions concerning which users can access the directories, and the permissions of files within the directories

• In web applications, the logs should not be exposed in web-accessible locations, and if done so, should have restricted access and be configured with a plain text MIME type (not HTML)

• When using a database, it is preferable to utilize a separate database account that is only used for writing log data and which has very restrictive database, table, function and command permissions

• Consider what types of messages should be logged:

- **o** Input validation failures e.g. protocol violations, unacceptable encodings, invalid parameter names and values
- **o** Output validation failures e.g. database record set mismatch, invalid data encoding
- **o** Authentication successes and failures
- **o** Authorization (access control) failures
- **o** Session management failures e.g. cookie session identification value modification
- **o** Connection timings

• Consider what each log message should contain:

- **o** Date and time, in some common format (also makes sense to ensure all nodes of an application are synced through something like NTP
- **o** User performing the action
- **o** Action being performed/attempted
- **o** Information on the client, e.g. IP address, source port, user-agent
- **o** External classifications e.g. NIST Security Content Automation Protocol (SCAP), Mitre Common Attack Pattern Enumeration and Classification (CAPEC)
- **o** Perform sanitization on all event data to prevent log injection attacks e.g. carriage return (CR), line feed (LF) and delimiter characters (and optionally to remove sensitive data)

• If writing to databases, read, understand and apply the SQL injection cheat sheet

• Ensure logging is implemented and enabled during application security, fuzz, penetration and performance testing

• Ensure logging cannot be used to deplete system resources, for example by filling up disk space or exceeding database transaction log space, leading to denial of service

• The logging mechanisms and collected event data must be protected from mis-use such as tampering in transit, and unauthorized access, modification and deletion once stored

• Store or copy log data to read-only media as soon as possible

• Consider what should not be logged:

  o Session identification values (consider replacing with a hashed value if needed to track session specific events)

  o Sensitive personal data and some forms of personally identifiable information (PII)

  o Authentication passwords (successful or unsuccessful)

  o Database connection strings

  o Keys

  o Data of a higher security classification than the logging system is allowed to store

References

• See NIST SP 800-92 Guide to Computer Security Log Management for more guidance.

• Mitre Common Event Expression (CEE)

• PCISSC PCI DSS v2.0 Requirement 10 and PA-DSS v2.0 Requirement 4

• Other Common Log File System (CLFS), Microsoft

2.17. ERROR HANDLING

Proper error handling is important in two ways:

1. It may affect the state of the application. The initial failure to prevent the error may cause the application to traverse into an insecure state. This covers the key premise of 'failing securely', errors induced should not leave the application in an insecure state. Resources should be locked down and released, sessions terminated (if required), and calculations or business logic should be halted (depending on the type of error, of course).

2. It may leak system information to a user. An important aspect of secure application development is to prevent information leakage. Error messages give an attacker great insight into the inner workings of an application. Weak error handling also aids the attacker, as the errors returned may assist them in constructing correct attack vectors.

A generic error page for most errors is recommended, this approach makes it more difficult for

attackers to identify signatures of potentially successful attacks such as blind SQL injection using booleanization ''<should include reference>'' or analysis of response time characteristics.

2.17.1.Description

The purpose of reviewing the Error Handling code is to assure that the application fails safely under all possible error conditions, expected and unexpected. No sensitive information is presented to the user when an error occurs. A company coding guidelines should include sections on Error Handling and how it should be controlled by an application suite, this will allow developers to code against this guidelines as well as review against them.A company coding guidelines should include sections on Error Handling and how it should be controlled by an application suite, this will allow developers to code against this guidelines as well as review against them.

For example, SQL injection is much tougher to successfully execute without some healthy error messages. It lessens the attack footprint, and an attacker would have to resort to using "blind SQL injection" which is more difficult and time consuming.

A well-planned error/exception handling guideline is important in a company for three reasons:

1. Good error handling does not give an attacker any information which is a means to an end, attacking the application

2. A proper centralised error strategy is easier to maintain and reduces the chance of any uncaught errors "bubbling up" to the front end of an application.

3. Information leakage could lead to social engineering exploits, for example if the hosting companies name is returned, or some employees name can be seen.

Regardless of whether the development language provide checked exceptions or not, reviewers should remember:

• Not all errors are exceptions. Don't rely on exception handling to be your only way of handling errors, handle all case statement 'default' sections, ensure all 'if' statements have their 'else' clauses covered, ensure that all exits from a function (e.g. return statements, exceptions, etc.) are covered. RAII concepts (e.g. auto pointers and the like) are an advantage here. In languages like Java and C#, remember that errors are different from exceptions (different hierarchy) and should be handled.

• Catching an exception is not automatically handling it. You've caught your exception, so how do you handle it? For many cases this should be obvious enough, based on your business logic, but for some (e.g. out of memory, array index out of bounds, etc.) the handling many not be so simple.

• Don't catch more that you can handle. Catch all clauses (e.g. 'catch(Exception e)' in Java & C# or 'catch(...) in C++) should be avoided as you will not know what type of exception you are handling, and if you don't know the exception type, how do you accurately handle it? It could be that the downstream server is not responding, or a user may have exceeded their quota, or you may be out of memory, these issues should be handled in different ways and thus should be

caught in exception clauses that are specific.

When an exception or error is thrown, we also need to log this occurrence. Sometimes this is due to bad development, but it can be the result of an attack or some other service your application relies on failing. This has to be imagined in the production scenario, if your application handles 'failing securely' by returning an error response to the client, and since we don't want to leak information that error will be generic, we need to have some way of identifying why the failure occurred. If your customer reports 1000's of errors occurred last night, you know that customer is going to want to know why. If you don't have proper logging and traceability coded into your application then you will not be able to establish if those errors were due to some attempted hack, or an error in your business logic when handling a particular type of error.

All code paths that can cause an exception to be thrown should check for success in order for the exception not to be thrown. This could be hard to impossible for a manual code review to cover, especially for large bodies of code. However if there is a debug version of the code, then modules/functions could throw relevant exceptions/errors and an automated tool can ensure the state and error responses from the module is as expected. This then means the code reviewer has the job of ensuring all relevant exceptions/errors are tested in the debug code.

2.17.2. What to Review

When reviewing code it is recommended that you assess the commonality within the application from an error/exception handling perspective. Frameworks have error handling resources which can be exploited to assist in secure programming, and such resources within the framework should be reviewed to assess if the error handling is "wired-up" correctly. A generic error page should be used for all exceptions if possible as this prevents the attacker from identifying internal responses to error states. This also makes it more difficult for automated tools to identify successful attacks.

For JSP struts this could be controlled in the struts-config.xml file, a key file when reviewing the wired-up struts environment:

FIGURE A10.5

Generic exception handling in struts

```
<exception key="bank.error.nowonga"
                    path="/NoWonga.jsp"
                    type="mybank.account.NoCashException"/>
```

Specification can be done for JSP in web.xml in order to handle unhandled exceptions.

When unhandled exceptions occur, but are not caught in code, the user is forwarded to a generic error page:

FIGURE A10.6

Unhandled exception in struts

```
<error-page>
       <exception-type>UnhandledException</exception-type>
       <location>GenericError.jsp</location>
 </error-page>
```

Also in the case of HTTP 404 or HTTP 500 errors during the review you may find:

FIGURE A10.7

Specific exception handling in struts

```
<error-page>
  <error-code>500</error-code>
  <location>GenericError.jsp</location>
 </error-page>
```

For IIS development the 'Application_Error()' handler will allow the application to catch all uncaught exceptions and handle them in a consistent way. Note this is important or else there is a chance your exception informaiton could be sent back to the client in the response.

For Apache development, returning failures from handlers or modules can prevent an further processing by the Apache engine and result in an error response from the server. Response headers, body, etc can be set by by the handler/module or can be configured using the "ErrorDocument" configuration. We should use a localized description string in every exception, a friendly error reason such as "System Error – Please try again later". When the user sees an error message, it will be derived from this description string of the exception that was thrown, and never from the exception class which may contain a stack trace, line number where the error occurred, class name, or method name.

Do not expose sensitive information like exception messages. Information such as paths on the local file system is considered privileged information; any internal system information should be hidden from the user. As mentioned before, an attacker could use this information to gather private user information from the application or components that make up the app.

Don't put people's names or any internal contact information in error messages. Don't put any

"human" information, which would lead to a level of familiarity and a social engineering exploit.

Failing Securely

There can be many different reasons why an application may fail, for example:

- The result of business logic conditions not being met.
- The result of the environment wherein the business logic resides fails.
- The result of upstream or downstream systems upon which the application depends fail.
- Technical hardware / physical failure.

Failures are like the Spanish Inquisition; popularly nobody expected the Spanish Inquisition (see Monty Python) but in real life the Spanish knew when an inquisition was going to occur and were prepared for it, similarly in an application, though you don't expect errors to occur your code should be prepared for them to happen. In the event of a failure, it is important not to leave the "doors" of the application open and the keys to other "rooms" within the application sitting on the table. In the course of a logical workflow, which is designed based upon requirements, errors may occur which can be programmatically handled, such as a connection pool not being available, or a downstream server returning a failure.

Such areas of failure should be examined during the course of the code review. It should be examined if resources should be released such as memory, connection pools, file handles etc.

The review of code should also include pinpointing areas where the user session should be terminated or invalidated. Sometimes errors may occur which do not make any logical sense from a business logic perspective or a technical standpoint, for example a logged in user looking to access an account which is not registered to that user. Such conditions reflect possible malicious activity. Here we should review if the code is in any way defensive and kills the user's session object and forwards the user to the login page. (Keep in mind that the session object should be examined upon every HTTP request).

Potentially Vulnerable Code

Java

In Java we have the concept of an error object; the Exception object. This lives in the Java package java.lang and is derived from the Throwable object. Exceptions are thrown when an abnormal occurrence has occurred. Another object derived from Throwable is the Error object, which is thrown when something more serious occurs. The Error object can be caught in a catch clause, but cannot be handled, the best you can do is log some information about the Error and then re-throw it.

Information leakage can occur when developers use some exception methods, which 'bubble' to the user UI due to a poor error handling strategy. The methods are as follows:

- printStackTrace()
- getStackTrace()

Also important to know is that the output of these methods is printed in System console, the same as System.out.println(e) where there is an Exception. Be sure to not redirect the outputStream to PrintWriter object of JSP, by convention called "out", for example:

- printStackTrace(out);

Note it is possible to change where system.err and system.out write to (like modifying fd 1 & 2 in bash or C/C++), using the java.lang.system package:

- setErr() for the System.err field, and
- setOut() for the System.out field.

This could be used on a process wide basis to ensure no output gets written to standard error or standard out (which can be reflected back to the client) but instead write to a configured log file.

C#.NET

In .NET a System.Exception object exists and has commonly used child objects such as ApplicationException and SystemException are used. It is not recommended that you throw or catch a SystemException this is thrown by runtime.

When an error occurs, either the system or the currently executing application reports it by throwing an exception containing information about the error, similar to Java. Once thrown, an exception is handled by the application or by the default exception handler. This Exception object contains similar methods to the Java implementation such as:

- StackTrace
- Source
- Message
- HelpLink

In .NET we need to look at the error handling strategy from the point of view of global error handling and the handling of unexpected errors. This can be done in many ways and this article is not an exhaustive list. Firstly, an Error Event is thrown when an unhandled exception is thrown.

This is part of the TemplateControl class, see reference:

•http://msdn.microsoft.com/library/default.asp?url=/library/enus/cpref/html/frlrfSystemWebUITemplateControlClassErrorTopic.asp

Error handling can be done in three ways in .NET, executed in the following order:

- On the aspx or associated codebehind page in the Page_Error.
- In the global.asax file's Application_Error (as mentioned before).
- In the web.config file's customErrors section.

It is recommended to look in these areas to understand the error strategy of the application.

Classic ASP

Unlike Java and .NET, classic ASP pages do not have structured error handling in try-catch blocks. Instead they have a specific object called "err". This makes error handling in a classic ASP pages hard to do and prone to design errors on error handlers, causing race conditions and information leakage. Also, as ASP uses VBScript (a subtract of Visual Basic), sentences like "On Error GoTo label" are not available. In classic ASP there are two ways to do error handling, the first is using the err object with a "On Error Resume Next" and "On Error GoTo 0".

FIGURE A10.8

Error handling in ASP.NET

```
Public Function IsInteger (ByVal Number)
   Dim Res, tNumber
   Number = Trim(Number)
   tNumber=Number
   On Error Resume Next                    'If an error occurs continue
execution
   Number = CInt(Number)                   'if Number is a alphanumeric
string a Type Mismatch error will occur
   Res = (err.number = 0)                  'If there are no errors then
return true
   On Error GoTo 0                        'If an error occurs stop execution
and display error
   re.Pattern = "^[\+\-]? *\d+$"          'only one +/- and digits are allowed
   IsInteger = re.Test(tNumber) And Res
 End Function
The second is using an error handler on an error page
(http://support.microsoft.com/kb/299981).
```

FIGURE A10.9

Error handling in ASP.NET

```
Dim ErrObj
```

```
set ErrObj = Server.GetLastError()
'Now use ErrObj as the regular err object
```

C++

In the C++ language, any object or built in type can be thrown. However there is a STL type std::exception which is supposed to be used as the parent of any user defined exception, indeed this type is used in the STL and many libraries as the parent type of all exceptions. Having std::exception encourages developers to create a hierarchy of exceptions which match the exception usage and means all exceptions can be caught by catching a std::exception object (instead of a 'catch (...)' block). Unlike Java, even errors that you can't recover from (e.g. std::bad_alloc which means your out of memory) derive from std::exception, so a 'catch(std::exception& e)' is similar to 'catch (...)' except that it allows you access to the exception so you can know what occurred and possibly print some error information using e.what().

There are many logging libraries for C++, so if your codebase uses a particular logging class look for usages of that logger for anywhere sensitive information can be written to the logs.

Error Handling in IIS

Page_Error is page level handling which is run on the server side in .NET. Below is an example but the error information is a little too informative and hence bad practice.

FIGURE A10.10

Page_Error handling in IIS

```
<script language="C#" runat="server">
 Sub Page_Error(Source As Object, E As EventArgs)
 Dim message As String = Request.Url.ToString()&
Server.GetLastError().ToString()
 Response.Write(message) // display message
 End Sub
 </script>
```

The text in the example above has a number of issues. Firstly, it displays the HTTP request to the user in the form of Request.Url.ToString(). Assuming there has been no data validation prior to this point, we are vulnerable to cross site scripting attacks. Secondly, the error message and stack trace is displayed to the user using Server.GetLastError().ToString() which divulges internal information regarding the application.

After the Page_Error is called, the Application_Error sub is called.

When an error occurs, the Application_Error function is called. In this method we can log

the error and redirect to another page. In fact catching errors in Application_Error instead of Page_Error would be an example of centralizing errors as described earlier.

FIGURE A10.11

Application_Error handling in IIS

```
<%@ Import Namespace="System.Diagnostics" %>
 <script language="C#" runat="server">
 void Application_Error(Object sender, EventArgs e) {
   String Message = "\n\nURL: http://localhost/" + Request.Path
                          + "\n\nMESSAGE:\n " +
Server.GetLastError().Message
                          + "\n\nSTACK TRACE:\n" +
Server.GetLastError().StackTrace;
   // Insert into Event Log
   EventLog Log = new EventLog();
   Log.Source = LogName;
   Log.WriteEntry(Message, EventLogEntryType.Error);
   Server.Redirect(Error.htm) // this shall also clear the error
 }
 </script>
```

Above is an example of code in Global.asax and the Application_Error method. The error is logged and then the user is redirected. Non-validated parameters are being logged here in the form of Request.Path. Care must be taken not to log or display non-validated input from any external source. ''<link to XSS>"

Web.config has custom error tags which can be used to handle errors. This is called last and if Page_error or Application_error is called and has functionality, that functionality shall be executed first. If the previous two handling mechanisms do not redirect or clear (Response.Redirect or a Server.ClearError), this will be called and you shall be forwarded to the page defined in web.config in the customErrors section, which is configured as follows:

FIGURE A10.12

IIS customError syntax

```
<customErrors mode="<On|Off|RemoteOnly>" defaultRedirect="<default redirect page>">
 <error statusCode="<HTTP status code>" redirect="<specific redirect page for listed status
code>"/>
 </customErrors>
```

The "mode" attribute value of "On" means that custom errors are enabled whilst the "Off" value means that custom errors are disabled. The "mode" attribute can also be set to "RemoteOnly"

which specifies that custom errors are shown only to remote clients and ASP.NET errors are shown to requests coming from the the local host. If the "mode" attribute is not set then it defaults to "RemoteOnly".

When an error occurs, if the status code of the response matches one of the error elements, then the relevent 'redirect' value is returned as the error page. If the status code does not match then the error page from the 'defaultRedirect' attribute will be displayed. If no value is set for 'defaultRedirect' then a generic IIS error page is returned.

An example of the customErrors section completed for an application is as follows:

FIGURE A10.13

IIS customError example

```
<customErrors mode="On" defaultRedirect="error.html">
     <error statusCode="500" redirect="err500.aspx"/>
     <error statusCode="404" redirect="notHere.aspx"/>
     <error statusCode="403" redirect="notAuthz.aspx"/>
 </customErrors>
```

Error Handling in Apache

In Apache you have two choices in how to return error messages to the client:

1. You can write the error status code into the req object and write the response to appear the way you want, then have you handler return 'DONE' (which means the Apache framework will not allow any further handlers/filters to process the request and will send the response to the client.

2. Your handler or filter code can return pre-defined values which will tell the Apache framework the result of your codes processsing (essentially the HTTP status code). You can then configure what error pages should be returned for each error code.

In the interest of centralizing all error code handling, option 2 can make more sense. To return a specific pre-defined value from your handler, refer to the Apache documentation for the list of values to use, and then return from the handler function as shown in the following example:

FIGURE A10.14

Apache handler error reporting

```
static int my_handler(request_rec *r)
 {
     if ( problem_processing() )
     {
       return HTTP_INTERNAL_SERVER_ERROR;
```

```
    }
     ... continue processing request ...
}
```

In the httpd.conf file you can then specify which page should be returned for each error code using the 'ErrorDocument' directive. The format of this directive is as follows:

- ErrorDocument <3-digit-code> <action>

... where the 3 digit code is the HTTP response code set by the handler, and the action is a local or external URL to be returned, or specific text to display. The following examples are taken from the Apache ErrorDocument documentation (https://httpd.apache.org/docs/2.4/custom-error.html) which contains more information and options on ErrorDocument directives:

FIGURE A10.15

Apache ErrorDocument configuration

```
ErrorDocument 500 "Sorry, our script crashed. Oh dear"
 ErrorDocument 500 /cgi-bin/crash-recover
 ErrorDocument 500 http://error.example.com/server_error.html
 ErrorDocument 404 /errors/not_found.html
 ErrorDocument 401 /subscription/how_to_subscribe.html
```

Leading Practice for Error Handling

Code that might throw exceptions should be in a try block and code that handles exceptions in a catch block. The catch block is a series of statements beginning with the keyword catch, followed by an exception type and an action to be taken.

"'Example: Java Try-Catch:'"

FIGURE A10.16

Java try-catch block

```
public class DoStuff {
    public static void Main() {
        try {
            StreamReader sr = File.OpenText("stuff.txt");
            Console.WriteLine("Reading line {0}", sr.ReadLine());
        }
        catch(MyClassExtendedFromException e) {
```

```
            Console.WriteLine("An error occurred. Please leave to room");
          logerror("Error: ", e);
        }
    }
 }
```

"'.NET Try–Catch"'

FIGURE A10.17

C#.NET try-catch block

```
public void run() {
            while (!stop) {
                try {
                    // Perform work here
                } catch (Throwable t) {
                    // Log the exception and continue
                WriteToUser("An Error has occurred, put the kettle on");
                    logger.log(Level.SEVERE, "Unexception exception", t);
                }
            }
        }
```

"'C++ Try–Catch"'

FIGURE A10.18

C++ try-catch block

```
void perform_fn() {
   try {
      // Perform work here

   } catch ( const MyClassExtendedFromStdException& e) {
      // Log the exception and continue
      WriteToUser("An Error has occurred, put the kettle on");
      logger.log(Level.SEVERE, "Unexception exception", e);
   }
 }
```

In general, it is best practice to catch a specific type of exception rather than use the basic

catch(Exception) or catch(Throwable) statement in the case of Java.

The Order of Catching Exceptions

Keep in mind that many languages will attempt to match the thrown exception to the catch clause even if it means matching the thrown exception to a parent class. Also remember that catch clauses are checked in the order they are coded on the page. This could leave you in the situation where a certain type of exception might never be handled correctly, take the following example where 'non_even_argument' is a subclass of 'std::invalid_argument':

FIGURE A10.19

C++ exception example

```
class non_even_argument : public std::invalid_argument {
 public:
  explicit non_even_argument (const string& what_arg);
 };


 void do_fn()
 {
    try
    {
       // Perform work that could throw
    }
    catch ( const std::invalid_argument& e )
    {
       // Perform generic invalid argument processing and return failure
    }
    catch ( const non_even_argument& e )
    {
       // Perform specific processing to make argument even and continue
processing
    }
 }
```

The problem with this code is that when a 'non_even_argument is thrown, the catch branch handling 'std::invalid_argument' will always be executed as it is a parent of 'non_even_argument' and thus the runtime system will consider it a match (this could also lead to slicing). Thus you need to be aware of the hierarchy of your exception objects and ensure that you list the catch for the more specific exceptions first in your code.

If the language in question has a finally method, use it. The finally method is guaranteed to always be called. The finally method can be used to release resources referenced by the method that threw the exception. This is very important. An example would be if a method

gained a database connection from a pool of connections, and an exception occurred without finally, the connection object shall not be returned to the pool for some time (until the timeout). This can lead to pool exhaustion. finally() is called even if no exception is thrown.

FIGURE A10.20

Java example of releasing resources

```
void perform_fn() {
   try {
      // Perform work here

   } catch ( const MyClassExtendedFromStdException& e) {
      // Log the exception and continue
      WriteToUser("An Error has occurred, put the kettle on");
      logger.log(Level.SEVERE, "Unexception exception", e);
   }
 }
```

A Java example showing finally() being used to release system resources.

Releasing resources and good housekeeping

RAII is Resource Acquisition Is Initialization, which is a way of saying that when you first create an instance of a type, it should be fully setup (or as much as possible) so that it's in a good state. Another advantage of RAII is how objects are disposed of, effectively when an object instance is no longer needed then it resources are automatically returned when the object goes out of scope (C++) or when it's 'using' block is finished (C# 'using' directive which calls the Dispose method, or Java 7's try-with-resources feature)

RAII has the advantage that programmers (and users to libraries) don't need to explicitly delete objects, the objects will be removed themselves, and in the process of removing themselves (destructor or Dispose)

For Classic ASP pages it is recommended to enclose all cleaning in a function and call it into an error handling statement after an "On Error Resume Next".

Building an infrastructure for consistent error reporting proves more difficult than error handling. Struts provides the ActionMessages and ActionErrors classes for maintaining a stack of error messages to be reported, which can be used with JSP tags like <html: error> to display these error messages to the user.

To report a different severity of a message in a different manner (like error, warning, or information) the following tasks are required:

1. Register, instantiate the errors under the appropriate severity

2. Identify these messages and show them in a consistent manner.

Struts ActionErrors class makes error handling quite easy:

FIGURE A10.21

ActionErrors example in Struts

```
ActionErrors errors = new ActionErrors()
 errors.add("fatal", new ActionError("...."));
 errors.add("error", new ActionError("...."));
 errors.add("warning", new ActionError("...."));
 errors.add("information", new ActionError("...."));
 saveErrors(request,errors); // Important to do this
```

Now that **we** have added the errors, we display them by using tags in the HTML page.

FIGURE A10.22

ActionErrors configuration within JSP page

```
<logic:messagePresent property="error">
 <html:messages property="error" id="errMsg" >
     <bean:write name="errMsg"/>
 </html:messages>
 </logic:messagePresent >
```

References

• For classic ASP pages you need to do some IIS configuration, follow http://support.microsoft.com/kb/299981 for more information.

2.18. REVIEWING SECURITY ALERTS

How will your code and applications react when something has gone wrong? Many companies that follow secure design and coding principals do so to prevent attackers from getting into their network, however many companies do not consider designing and coding for the scenario where an attacker may have found a vulnerability, or has already exploited it to run code within a companies firewalls (i.e. within the Intranet).

Many companies employ SIEM logging technologies to monitor network and OS logs for

patterns that detect suspicions activity, this section aims to further encourage application layers and interfaces to do the same.

2.18.1.Description

This section concentrates on:

1. Design and code that allows the user to react when a system is being attacked.

2. Concepts allowing applications to flag when they have been breached.

When a company implements secure design and coding, it will have the aim of preventing attackers from misusing the software and accessing information they should not have access to. Input validation checks for SQL injections, XSS, CSRF, etc. should prevent attackers from being able to exploit these types of vulnerabilities against the software. However how should software react when an attacker is attempting to breach the defenses, or the protections have been breached?

For an application to alert to security issues, it needs context on what is 'normal' and what constitutes a security issue. This will differ based on the application and the context within which it is running. In general applications should not attempt to log every item that occurs as the excessive logging will slow down the system, fill up disk or DB space, and make it very hard to filter through all the information to find the security issue.

At the same time, if not enough information is monitored or logged, then security alerting will be very hard to do based on the available information. To achieve this balance an application could use its own risk scoring system, monitoring at a system level what risk triggers have been spotted (i.e. invalid inputs, failed passwords, etc.) and use different modes of logging. Take an example of normal usage, in this scenario only critical items are logged. However if the security risk is perceived to have increased, then major or security level items can be logged and acted upon. This higher security risk could also invoke further security functionality as described later in this section.

Take an example where an online form (post authentication) allows a user to enter a month of the year. Here the UI is designed to give the user a drop down list of the months (January through to December). In this case the logged in user should only ever enter one of 12 values, since they typically should not be entering any text, instead they are simply selecting one of the pre-defined drop down values.

If the server receiving this form has followed secure coding practices, it will typically check that the form field matches one of the 12 allowed values, and then considers it valid. If the form field does not match, it returns an error, and may log a message in the server. This prevents the attacker from exploiting this particular field, however this is unlikely to deter an attacker and they would move onto other form fields.

In this scenario we have more information available to us than we have recorded. We have returned an error back to the user, and maybe logged an error on the server. In fact we know a lot more; an authenticated user has entered an invalid value which they should never have been able to do (as it's a drop down list) in normal usage.

This could be due to a few reasons:

• There's a bug in the software and the user is not malicious.

• An attacker has stolen the users login credentials and is attempting to attack the system.

• A user has logged in but has a virus/trojan which is attempting to attack the system.

• A user has logged in but is experiencing a man-in-the-middle attack.

• A user is not intending to be malicious but has somehow changed the value with some browser plugin, etc.

If it's the first case above, then the company should know about it to fix their system. If it's case 2, 3 or 3 then the application should take some action to protect itself and the user, such as reducing the functionality available to the user (i.e. no PII viewable, can't change passwords, can't perform financial transactions) or forcing further authentication such as security questions or out-of-band authentication. The system could also alert the user to the fact that the unexpected input was spotted and advise them to run antivirus, etc., thus stopping an attack when it is underway.

Obviously care must be taken in limiting user functionality or alerting users encase it's an honest mistake, so using a risk score or noting session alerts should be used. For example, if everything has been normal in the browsing session and 1 character is out of place, then showing a red pop-up box stating the user has been hacked is not reasonable, however if this is not the usual IP address for the user, they have logged in at an unusual time, and this is the 5th malformed entry with what looks like an SQL injection string, then it would be reasonable for the application to react. This possible reaction would need to be stated in legal documentation.

In another scenario, if an attacker has got through the application defenses extracted part of the applications customer database, would the company know? Splitting information in the database into separate tables makes sense from an efficiency point of view, but also from a security view, even putting confidential information into a separate partition can make it harder for the attacker. However if the attacker has the information it can be hard to detect and applications should make steps to aid alerting software (e.g. SIEM systems). Many financial institutions use risk scoring systems to look at elements of the user's session to give a risk score, if Johnny always logs in at 6pm on a Thursday from the same IP, then we have a trusted pattern. If suddenly Johnny logs in at 2:15am from an IP address on the other side of the world, after getting the password wrong 7 times, then maybe he's jetlagged after a long trip, or perhaps his account has been hacked. Either way, asking him for out-of-band authentication would be reasonable to allow Johnny to log in, or to block an attacker from using Johnny's account.

If the application takes this to a larger view, it can determine that on a normal day 3% of the users log on in what would be considered a riskier way, i.e. different IP address, different time,

etc. If on Thursday it sees this number rise to 23% then has something strange happened to the user base, or has the database been hacked? This type of information can be used to enforce a blanket out-of-band authentication (and internal investigation of the logs) for the 23% of 'riskier' users, thereby combining the risk score for the user with the overall risk score for the application.

Another good option is 'honey accounts' which are usernames and passwords that are never given out to users. These accounts are added just like any other user, and stored in the DB, however they are also recorded in a special cache and checked on login. Since they are never given to any user, no user should ever logon with them, however if one of those accounts are used, then the only way that username password combination could be known is if an attacker got the database, and this information allows the application to move to a more secure state and alert the company that the DB has been hacked.

2.18.2. What to Review

When reviewing code modules from a security alerting point of view, some common issues to look out for include:

• Will the application know if it's being attacked? Does it ignore invalid inputs, logins, etc. or does it log them and monitor this state to capture a cumulative perception of the current risk to the system?

• Can the application automatically change its logging level to react to security threats? Is changing security levels dynamic or does it require a restart?

• Does the SDLC requirements or design documentation capture what would constitute a security alert? Has this determination been peer reviewed? Does the testing cycle run through these scenarios?

• Does the system employ 'honey accounts' such that the application will know if the DB has been compromised?

• Is there a risk based scoring system that records the normal usage of users and allows for determination or reaction if the risk increases?

• If a SIEM system is being used, have appropriate triggers been identified? Has automated tests been created to ensure those trigger log messages are not accidentally modified by future enhancements or bug fixes?

• Does the system track how many failed login attempts a user has experienced? Does the system react to this?

• Does certain functionality (i.e. transaction initiation, changing password, etc) have different modes of operation based on the current risk score the application is currently operating within?

• Can the application revert back to 'normal' operation when the security risk score drops to normal levels?

• How are administrators alerted when security risk score rises? Or when a breach has been assumed? At an operational level, is this tested regularly? How are changes of

personnel handled?

References

TBD

2.19. REVIEW FOR ACTIVE DEFENSE

Attack detection undertaken at the application layer has access to the complete context of an interaction and enhanced information about the user. If logic is applied within the code to detect suspicious activity (similar to an application level IPS) then the application will know what is a high-value issue and what is noise. Input data are already decrypted and canonicalized within the application and therefore application-specific intrusion detection is less susceptible to advanced evasion techniques. This leads to a very low level of attack identification false positives, providing appropriate detection points are selected.

The fundamental requirements are the ability to perform four tasks:

1. Detection of a selection of suspicious and malicious events.

2. Use of this knowledge centrally to identify attacks.

3. Selection of a predefined response.

4. Execution of the response.

2.19.1.Description

Applications can undertake a range of responses that may include high risk functionality such as changes to a user's account or other changes to the application's defensive posture. It can be difficult to detect active defense in dynamic analysis since the responses may be invisible to the tester. Code review is the best method to determine the existence of this defense.

Other application functionality like authentication failure counts and lock-out, or limits on rate of file uploads are 'localized' protection mechanisms. This sort of standalone logic is 'not' active defense equivalents in the context of this review, unless they are rigged together into an application-wide sensory network and centralized analytical engine.

It is not a bolt-on tool or code library, but instead offers insight to an approach for organizations to specify or develop their own implementations – specific to their own business, applications, environments and risk profile – building upon existing standard security controls.

2.19.2.What to Review

In the case where a code review is being used to detect the presence of a defense, its absence should be noted as a weakness. Note that active defense cannot defend an application that has known vulnerabilities, and therefore the other parts of this guide are extremely important. The code reviewer should note the absence of active defense as a vulnerability.

The purpose of code review is not necessarily to determine the efficacy of the active

defense, but could simply be to determine if such capability exists.

Detection points can be integrated into presentation, business and data layers of the application. Application-specific intrusion detection does not need to identify all invalid usage, to be able to determine an attack. There is no need for "infinite data" or "big data" and therefore the location of "detection points" may be very sparse within source code.

A useful approach for identifying such code is to find the name of a dedicated module for detecting suspicious activity (such as OWASP AppSensor). Additionally a company can implement a policy of tagging active defense detection points based on [http://capec.mitre.org/ Mitre's Common Attack Pattern Enumeration and Classification] [reference] (CAPEC), strings such as CAPEC-212, CAPEC-213, etc.

The OWASP AppSensor detection point type identifiers and CAPEC codes will often have been used in configuration values [reference] (e.g. [https://code.google.com/p/appsensor/source/browse/trunk/AppSensor/src/test/resources/.esapi/ESAPI.properties?r=53 in ESAPI for Java]), parameter names and security event classification. Also, examine error logging and security event logging mechanisms as these may be being used to collect data that can then be used for attack detection. Identify the code or services called that perform this logging and examine the event properties recorded/sent. Then identify all places where these are called from.

An examination of error handling code relating to input and output validation is very likely to reveal the presence of detection points. For example, in a whitelist type of detection point, additional code may have been added adjacent, or within error handling code flow:

```
if ( var !Match this ) {
    Error handling
    Record event for attack detection
}
```

In some situations attack detection points are looking for blacklisted input, and the test may not exist otherwise, so brand new code is needed. Identification of detection points should also have found the locations where events are recorded (the "event store"). If detection points cannot be found, continue to review the code for execution of response, as this may provide insight into the existence of active defense.

The event store has to be analysed in real time or very frequently, in order to identify attacks based on predefined criteria. The criteria should be defined in configuration settings (e.g. in configuration files, or read from another source such as a database). A process will examine the event store to determine if an attack is in progress, typically this will be attempting to identify an authenticated user, but it may also consider a single IP address, range of IP addresses, or groups of users such as one or more roles, users with a particular privilege or even all users.

Once an attack has been identified, the response will be selected based on predefined criteria. Again an examination of configuration data should reveal the thresholds related to each detection point, groups of detection points or overall thresholds.

The most common response actions are user warning messages, log out, account lockout and administrator notification. However, as this approach is connected into the application, the possibilities of response actions are limited only by the coded capabilities of the application.

Search code for any global includes that poll attack identification/response identified above. Response actions (again a user, IP address, group of users, etc) will usually be initiated by the application, but in some cases other applications (e.g. alter a fraud setting) or infrastructure components (e.g. block an IP address range) may also be involved.

Examine configuration files and any external communication the application performs.

The following types of responses may have been coded:

- Logging increased
- Administrator notification
- Other notification (e.g. other system)
- Proxy
- User status change
- User notification
- Timing change
- Process terminated (same as traditional defenses)
- Function disabled
- Account log out
- Account lock out
- Collect data from user.

Other capabilities of the application and related system components can be repurposed or extended, to provide the selected response actions. Therefore review the code associated with any localised security measures such as account lock out.

References

- The guidance for adding active response to applications given in theOWASP_AppSensor_Project
- Category:OWASP Enterprise Security API
- https://code.google.com/p/appsensor/ AppSensor demonstration code

2.20. RACE CONDITIONS

Race Conditions occur when a piece of code does not work as it is supposed to (like many security issues). They are the result of an unexpected ordering of events, which can result

in the finite state machine of the code to transition to a undefined state, and also give rise to contention of more than one thread of execution over the same resource. Multiple threads of execution acting or manipulating the same area in memory or persisted data which gives rise to integrity issues.

2.20.1.Description

With competing tasks manipulating the same resource, we can easily get a race condition as the resource is not in step-lock or utilises a token based system such as semaphores. For example if there are two processes (Thread 1, T1) and (Thread 2, T2). The code in question adds 10 to an integer X. The initial value of X is 5.

X = X + 10

With no controls surrounding this code in a multithreaded environment, the code could experience the following problem:

T1 **places** X into a register in thread 1

T2 places X into a register in thread 2

T1 adds 10 to the value in T1's register resulting in 15

T2 adds 10 to the value in T2's register resulting in 15

T1 saves the register value (15) into X.

T1 saves the register value (15) into X.

The value should actually be 25, as each thread added 10 to the initial value of 5. But the actual value is 15 due to T2 not letting T1 save into X before it takes a value of X for its addition.

This leads to undefined behavior, where the application is in an unsure state and therefore security cannot be accurately enforced.

2.20.2.What to Review

- In C#.NET look for code which used multithreaded environments:
  - **o** Thread
  - **o** System.Threading
  - **o** ThreadPool
  - **o** System.Threading.Interlocked
- In Java code look for
  - **o** java.lang.Thread
  - **o** start()
  - **o** stop()
  - **o** destroy()

- **o** init()
- **o** synchronized
- **o** wait()
- **o** notify()
- **o** notifyAll()

• For classic ASP multithreading is not a directly supported feature, so this kind of race condition could be present only when using COM objects.

• Static methods and variables (one per class, not one per object) are an issue particularly if there is a shared state among multiple threads. For example, in Apache, struts static members should not be used to store information relating to a particular request. The same instance of a class can be used by multiple threads, and the value of the static member cannot be guaranteed.

• Instances of classes do not need to be thread safe as one is made per operation/request. Static states must be thread safe.

o References to static variables, these must be thread locked.

o Releasing a lock in places other then finally{} may cause issues.

o Static methods that alter static state.

References

• http://msdn2.microsoft.com/en-us/library/f857xew0(vs.71).aspx

2.21. BUFFER OVERRUNS

A buffer is an amount of contiguous memory set aside for storing information. For example if a program has to remember certain things, such as what your shopping cart contains or what data was inputted prior to the current operation. This information is stored in memory in a buffer. Languages like C, C++ (which many operating systems are written in), and Objective-C are extremely efficient, however they allow code to access process memory directly (through memory allocation and pointers) and intermingle data and control information (e.g. in the process stack). If a programmer makes a mistake with a buffer and allows user input to run past the allocated memory, the user input can overwrite program control information and allow the user to modify the execution of the code.

Note that Java, C#.NET, Python and Ruby are not vulnerable to buffer overflows due to the way they store their strings in char arrays, of which the bounds are automatically checked by the frameworks, and the fact that they do not allow the programmer direct access to the memory (the virtual machine layer handles memory instead). Therefore this section does not apply to those languages. Note however that native code called within those languages (e.g. assembly, C, C++) through interfaces such as JNI or 'unsafe' C# sections can be susceptible to buffer overflows.

2.21.1.Description

To allocate a buffer the code declares a variable of a particular size:

- char myBuffer[100]; // large enough to hold 100 char variables
- int myIntBuf[5]; // large enough to hold 5 integers
- Widget myWidgetArray[17]; // large enough to hold 17 Widget objects

As there is no automatic bounds checking code can attempt to add a Widget at array location 23 (which does not exist). When the code does this, the complier will calculate where the 23rd Widget should be placed in memory (by multiplying 23 x sizeof(Widget) and adding this to the location of the 'myWidgetArray' pointer). Any other object, or program control variable/register, that existed at this location will be overwritten.

Arrays, vectors, etc. are indexed starting from 0, meaning the first element in the container is at 'myBuffer[0]', this means the last element in the container is not at array index 100, but at array index 99. This can often lead to mistakes and the 'off by one' error, when loops or programming logic assume objects can be written to the last index without corrupting memory.

In C, and before the C++ STL became popular, strings were held as arrays of characters:

- char nameString[10];

This means that the 'nameString' array of characters is vulnerable to array indexing problems described above, and when many of the string manipulation functions (such as strcpy, strcat, described later) are used, the possibility of writing beyond the 10th element allows a buffer overrun and thus memory corruption.

As an example, a program might want to keep track of the days of the week. The programmer tells the computer to store a space for 7 numbers. This is an example of a buffer. But what happens if an attempt to add 8 numbers is performed? Languages such as C and C++ do not perform bounds checking, and therefore if the program is written in such a language, the 8th piece of data would overwrite the program space of the next program in memory, and would result in data corruption. This can cause the program to crash at a minimum or a carefully crafted overflow can cause malicious code to be executed, as the overflow payload is actual code.

FIGURE A10.23

Buffer overflow example

```
void copyData(char *userId) {
    char  smallBuffer[10]; // size of 10
    strcpy (smallBuffer, userId);
 }
```

```
  int main(int argc, char *argv[]) {
    char *userId = "01234567890"; // Payload of 12 when you include the '\n'
string termination
                  // automatically added by the "01234567890" literal
    copyData (userId); // this shall cause a buffer overload
  }
```

What to Review: Buffer Overruns

C library functions such as strcpy (), strcat (), sprintf () and vsprintf () operate on null terminated strings and perform no bounds checking. gets() is another function that reads input (into a buffer) from stdin until a terminating newline or EOF (End of File) is found. The scanf () family of functions also may result in buffer overflows.

Using strncpy(), strncat() and snprintf() functions allows a third 'length' parameter to be passed which determines the maximum length of data that will be copied/etc. into the destination buffer. If this is correctly set to the size of the buffer being written to, it will prevent the target buffer being overflowed. Also note fgets() is a replacement for gets(). Always check the bounds of an array before writing it to a buffer. The Microsoft C runtime also provides additional versions of many functions with an '_s' suffix (strcpy_s, strcat_s, sprintf_s). These functions perform additional checks for error conditions and call an error handler on failure.

The C code below is not vulnerable to buffer overflow as the copy functionality is performed by 'strncpy' which specifies the third argument of the length of the character array to be copied, 10.

FIGURE A10.24

Buffer overflow fix using strncpy

```
void copyData(char *userId) {
    char  smallBuffer[10]; // size of 10
    strncpy(smallBuffer, userId, sizeof(smallBuffer)); // only copy first 10
elements
    smallBuffer[10] = 0; // Make sure it is terminated.
 }
 int main(int argc, char *argv[]) {
    char *userId = "01234567890"; // Payload of 11
    copyData (userId);
 }
```

Modern day C++ (C++11) programs have access to many STL objects and templates that help prevent security vulnerabilities. The std::string object does not require the calling code have

any access to underlying pointers, and automatically grows the underlying string representation (character buffer on the heap) to accommodate the operations being performed. Therefore code is unable to cause a buffer overflow on a std::string object.

Regarding pointers (which can be used in other ways to cause overflows), C++11 has smart pointers which again take away any necessity for the calling code to user the underlying pointer, these types of pointers are automatically allocated and destroyed when the variable goes out of scope. This helps to prevent memory leaks and double delete errors.

Also the STL containers such as std::vector, std::list, etc., all allocate their memory dynamically meaning normal usage will not result in buffer overflows. Note that it is still possible to access these containers underlying raw pointers, or reinterpret_cast the objects, thus buffer overflows are possible, however they are more difficult to cause.

Compliers also help with memory issues, in modern compilers there are 'stack canaries' which are subtle elements placed in the complied code which check for out-of-bound memory accesses. These can be enabled when compiling the code, or they could be enabled automatically. There are many examples of these stack canaries, and for some system many choices of stack canaries depending on an organizations appetite for security versus performance. Apple also have stack canaries for iOS code as Objective-C is also susceptible to buffer overflows.

In general, there are obvious examples of code where a manual code reviewer can spot the potential for overflows and off-by-one errors, however other memory leaks (or issues) can be harder to spot. Therefore manual code review should be backed up by memory checking programs available on the market.

2.21.2. What to Review:

Format Function Overruns

A format function is a function within the ANSI C specification that can be used to tailor primitive C data types to human readable form. They are used in nearly all C programs to output information, print error messages, or process strings.

TABLE 24

Some format parameters:

The %s in this case ensures that value pointed to by the parameter 'abc' is printed as an array of characters.

```
For example:

char* myString = "abc";
printf ("Hello: %s\n", abc);
```

Through supplying the format string to the format function we are able to control the

behaviour of it. So supplying input as a format string makes our application do things it's not meant to. What exactly are we able to make the application do?

If we supply %x (hex unsigned int) as the input, the 'printf' function shall expect to find an integer relating to that format string, but no argument exists. This cannot be detected at compile time. At runtime this issue shall surface.

For every % in the argument the printf function finds it assumes that there is an associated value on the stack. In this way the function walks the stack downwards reading the corresponding values from the stack and printing them to the user.

Using format strings we can execute some invalid pointer access by using a format string such as:

• printf ("%s%s%s%s%s%s%s%s%s%s%s%s");

Worse again is using the '%n' directive in 'printf()'. This directive takes an 'int*' and 'writes' the number of bytes so far to that location.

Where to look for this potential vulnerability. This issue is prevalent with the 'printf()' family of functions, ''printf(),fprintf(), sprintf(), snprintf().' Also 'syslog()' (writes system log information) and setproctitle(const char *fmt, ...); (which sets the string used to display process identifier information).

Integer Overflows

Data representation for integers will have a finite amount of space, for example a short in many languages is 16 bits twos complement number, which means it can hold a maximum number of 32,767 and a minimum number of -32,768. Twos complement means that the very first bit (of the 16) is a representation of whether the number of positive or negative. If the first bit is '1', then it is a negative number.

The representation of some boundary numbers are given in table 25.

Table 25

If you add 1 to 32,766, it adds 1 to the representation giving the representation for 32,767 shown above. However if you add one more again, it sets the first bit (a.k.a. most significant bit), which is then interpreted by the system as -32,768.

If you have a loop (or other logic) which is adding or counting values in a short, then the application could experience this overflow. Note also that subtracting values below -32,768 also means the number will wrap around to a high positive, which is called underflow.

FIGURE A10.25

Integer overflow

```
#include <stdio.h>
int main(void){
            int val;
            val = 0x7fffffff;       /* 2147483647*/
            printf("val = %d (0x%x)\n", val, val);
            printf("val + 1 = %d (0x%x)\n", val + 1 , val + 1); /*Overflow
the int*/
            return 0;
}
```

The binary representation of 0x7fffffff is 11111111111111111111111111111111; this integer is initialized with the highest positive value a signed long integer can hold.

Here when we add 1 to the hex value of 0x7fffffff the value of the integer overflows and goes to a negative number (0x7fffffff + 1 = 80000000) In decimal this is (-2147483648). Think of the problems this may cause. Compilers will not detect this and the application will not notice this issue.

We get these issues when we use signed integers in comparisons or in arithmetic and also when comparing signed integers with unsigned integers.

FIGURE A10.26

Integer overflow flaw comparing integers

```
int myArray[100];

    int fillArray(int v1, int v2){
        if(v2 > sizeof(myArray) / sizeof(int) -1 ){
            return -1; /* Too Big */
        }
        myArray [v2] = v1;
        return 0;
    }
```

Here if v2 is a massive negative number the "if" condition shall pass. This condition checks to see if v2 is bigger than the array size.

If the bounds check was not performed the line "myArray[v2] = v1" could have assigned the value v1 to a location out of the bounds of the array causing unexpected results.