Содержание

A Guide to Building Secure Web Applications and Web Services

Authentication

Best Practices

- Authentication is only as strong as your user management processes, and in particular the user issuance and evidence of identity policies. The stronger the requirement for non-repudiation, the more expensive the process.
- Use the most appropriate form of authentication suitable for your asset classification. For example, username and password is suitable for low value systems such as blogs and forums, SMS challenge response is suitable for low value e-commerce systems (in 2005), whilst transaction signing is suitable for high value systems such as high value e-commerce (all e-commerce sites should consider it by 2007), banking and trading systems.
- Re-authenticate the user for high value transactions and access to protected areas (such as changing from user to administrative level access)
- Authenticate the transaction, not the user. Phishers rely on poorly implemented user authentication schemes
- Passwords are trivially broken and are unsuitable for high value systems. Therefore, the controls should reflect this. Any password less than 16 characters in length can be brute forced in less than two weeks, so set your password policy to be reasonable:
    1. Train your users as to suitable password construction
    2. Allow your users to write down their passwords as long as they keep them safe
    3. Encourage users to use pass phrases instead of passwords
    4. Relax password expiry requirements upon the strength of the password chosen – passwords between 8 and 16 that cannot be easily cracked should have an expiry of no less than 30 days, and pass phrases above 16 characters probably do not need a hard expiry limit, but a gentle reminder after (say) 90 days instead.

How to protect yourself

- New applications should have no default accounts.
- Ensure the documentation says to determine that the underlying infrastructure has no default accounts left active (such as Administrator, root, sa, ora, dbsnmp, etc)
- Do not allow the code to contain any default, special, or backdoor credentials
- When creating the installer, ensure the installer does not create any default, special, credentials
- Ensure that all accounts, particularly administrative accounts, are fully specified by the installer / user.
- There should be no examples or images in the documentation with usernames in them
- Where possible, allow for users to create their own usernames. Usernames only have to be unique.
- Usernames should be HTML, SQL and LDAP safe – suggest only allowing A..Z, a..z, and 0-9. If you wish to allow spaces, @ symbols or apostrophes, ensure you properly escape the special characters
- Avoid the use of Firstname.Lastname, e-mail address, credit card numbers or customer number,

or any semi-public data, such as social security number (US only – also known as SSN), employee number, or similar.
- Ensure your application has a change password function.
- The form must include the old password, the new password and a confirmation of the new password
- Use AUTOCOMPLETE=off to prevent browsers from caching the password locally
- If the user gets the old password wrong too many times, lock the account and kill the session
- For higher risk applications or those with compliance issues, you should include the ability to prevent passwords being changed too frequently, which requires a password history. The password history should consist only of previous hashes, not clear text versions of the password. Allow up to 24 old password hashes.
- Ensure your application does not allow blank passwords
- Enforce a minimum password length. For higher risk applications, prevent the user from using (a configurable) too short password length. For low risk apps, a warning to the user is acceptable for passwords less than six characters in length.
- Encourage users to use long pass phrases (like “My milk shake brings all the boys to the yard” or “Let me not to the marriage of true minds Admit impediments”) by not strictly enforcing complexity controls for passwords over 14 characters in length
- Ensure your application allows arbitrarily long pass phrases by using a decent one-way hash algorithm, such as AES-128 in digest mode or SHA-256 bit.
- Allow for languages other than English (possibly allowing more than one language at a time for bi-lingual or multi-lingual locales like Russian)
- The application should have the following controls (but optionally enforce):
    1. Password minimum length (but never maximum length)
    2. Password change frequency
    3. Password minimum password age (to prevent users cycling through the password history)
    4. Password complexity requirements
    5. Password history
    6. Password lockout duration and policy (ie no lockout, lockout for X minutes, lockout permanently)
    7. For higher risk applications, use a weak password dictionary helper to decide if the user’s choice for password is too weak.

Note: Complex frequently changed passwords are counterproductive to security. It is better to have a long-lived strong passphrase than a 10 character jumble changed every 30 days. The 30 days will ensure that PostIt™ notes exist all over the organization with passwords written down.

- Use AES-128 in digest mode or SHA1 in 256 bit mode
- Use a non-static salting mechanism
- Never send the password hash or password back to the user in any form
- Automated password resets. Automated password reset mechanisms are common where organizations believe that they need to avoid high help desk support costs from authentication. From a risk management perspective, password reset functionality seems acceptable in many circumstances. However, password reset functionality equates to a secondary, but much weaker password mechanism. From a forthcoming study (see references), it appears that password reset systems with five responses are the equivalent to two character passwords and require reversible or clear text passwords to be stored in the back end system, which is contrary to security best practices and most information security policies. In general, questions required by password reset systems are easily found from public records (mother’s maiden name, car color, etc). In many instances, the password reset asks for data that is illegal or highly problematic to collect, such as social security numbers. In most privacy regimes, you

may only collect information directly useful to your application's needs, and disclose to the user why you are collecting that information. In general, unless the data being protected by your authentication mechanism is practically worthless, you should not use password reset mechanisms.

How to protect yourself

- High value transaction systems should not use password reset systems. It is discouraged for all other applications.
- Consider cheaper and more secure systems, such as pre-sending the user a password reset token in a sealed envelope which is replenished upon use.
- If the questions and answers are used to identify the user to the help desk, simply generate a random number in the "How to call the help desk" page on your web site and verify this number when the user calls in.
- Be careful when implementing automated password resets. The easiest to get right is "e-mail the user" as it creates an audit trail and contains only one secret – the user's e-mail address. However, this is risky if the user's e-mail account has been compromised.
- Send a message to the user explaining that someone has triggered the password reset functionality. Ask them if they didn't ask for the reset to report the incident. If they did trigger it, provide a short cryptographically unique time limited token ready for cut and paste. Do not provide a hyperlink as this is against phishing best practices and will make scamming users easier over time. This value should then be entered into the application which is waiting for the token. Check that the token has not expired and it is valid for that user account. Ask the user to change their password right there. If they are successful, send a follow up e-mail to the user and to the admin. Log everything.
- If you have to choose the hint based alternative, use free-form hints, with non-public knowledge suggestions, like "What is your favorite color?" "What is your favorite memory," etc. Do not use mother's maiden name, SSN, or similar. The user should enter five hints during registration, and be presented with three when they reset the password.
- Obviously, both password reset mechanisms should be over SSL to provide integrity and privacy.
- Has a delay between the user submitting the credential and a success or failure is reported. A delay of three seconds can make automated brute force attacks almost infeasible. A progressive delay (3 seconds then 15 then 30 then disconnect) can make casual brute force attacks completely ineffective
- warns the user with a suitable error message that does not disclose which part of the application credentials are incorrect by using a common authentication error page:
    - logs failed authentication attempts (in fact, a good application logs all authentication attempts)
    - for applications requiring stronger controls, blocking access from abusive IP addresses (ie accessing more than three accounts from the same IP address, or attempting to lock out more than one account)
    - destroys the session after too many retries. In such a scenario, log analysis might reveal multiple accesses to the same page from the same IP address within a short period of time. Event correlation software such as Simple Event Correlator (SEC) can be used to define rules to parse through the logs and generate alerts based on aggregated events. This could also be done by adding a Snort rule for alerting on HTTP Authorization Failed error messages going out from your web server to the user, and SEC can then be used to aggregate and correlate these alerts.
- If your application deals with high value transactions, it should not have "Remember Me" functionality.

- If the risk is minimal, it is enough to warn users of the dangers before allowing them to tick the box.
- Never use a predictable “pre-authenticated” token. The token should be kept on record to ensure that the authentication mechanism is not bypassable
- Determine a suitable time out period with the business (session)
- Configure the time out in the session handler to abandon or close the session after the time out has expired.
- Implement logout functionality
- Include a log out link or button in every view and not just in the index page
- Ensure that logout abandons or closes out the session, and clears any cookies left on the browser
- (High risk applications) Include text to warn the user to clear their browser’s cache and history if they are on a shared PC
- Users should have the ability to remove their account. This process should require confirmation, but otherwise should not overly make it difficult to the user to remove their records.
- Accounts that have not logged in for a long period of time should be locked out, or preferably removed.
- If you retain records, you are required by most privacy regimes to detail what you keep and why to the user in your privacy statement.
- When partially scrubbing accounts (ie you need to maintain a transaction history or accounting history), ensure all personally identifiable information is not available or reachable from the front end web application, i.e. export to an external database of archived users or CSV format
- Implement self-registration carefully based upon the risk to your business. For example, you may wish to put monetary or transaction limits on new accounts.
- If limits are imposed, they should be validated by business rules, and not just by security through obscurity.
- Ensure the process to maximize the features of an account is simple and transparent.
- When accounts are modified, ensure that a reasonable trace or audit of activity is maintained. Do not use CAPTCHA tags. They are illegal if you are required to be accessible to all users (often the case for government sites, health, banking, and nationally protected infrastructure, particularly if there is no other method of interacting with that organization). If you have to:
    - always provide a method by which a user may sign up or register for your web site offline or via another method
    - deter the use of automated sign ups by using the “no follow” tag (see section TODO) . Search engines will ignore hyperlinks and pages with this tag set, immensely devaluing the use of link spamming
- Limit the privileges of newly signed up accounts or similar until a positive validation has occurred. This can be as simple as including a unique reference ID to a registered credit card, or requiring a certain amount of time before certain features are unlocked, such as public posting rights or unfettered access to all features

Authorization

- Development, test and staging environments must be set up to function with the lowest possible privilege so that production will also work with lowest possible privileges
- Ensure that system level accounts (those that run the environment) should be as low privilege as possible. Never should “Administrator”, “root”, “sa”, “sysman”, “Supervisor”, or any other all privileged account be used to run the application or connect to the web server, database, or middleware.
- User accounts should possess just enough privileges within the application to do their assigned

tasks
- Users should not be administrators and vice versa
- Users should not be able to use any unauthorized or administrative functions. See the authorization section for more details
- Database access should be through parameterized stored procedures (or similar) to allow all table access to be revoked (ie select, drop, update, insert, etc) using a low privilege database account. This account should not hold any SQL roles above “user” (or similar)
- Code access security should be evaluated and asserted. If you only need the ability to look up DNS names, you only ask for code access permissions to permit this. That way if the code tries to read /etc/password, it can’t and will be terminated
- Infrastructure accounts should be low privilege users like LOCAL SERVICE or nobody. However, if all code runs as these accounts, the “keys to the kingdom” problem may re-surface. If you know what you’re doing, with careful replication of the attributes of low privilege accounts like LOCAL SERVICE or nobody is better to create low privilege users and partition than to share LOCAL SERVICE or “nobody”.
- Always start ACL’s using “deny all” and then adding only those roles and privileges necessary
- Network access controls: firewalls and host based filters
- File system access controls: file and directory permissions
- User access controls: user and group platform security. Java / .NET / PHP access controls: always write a Java 2 security policy or in .NET ensure that Code Access security is asserted either programmatically or by assembly permissions. In PHP, consider the use of “safe mode” functionality, including open_basedir directives to limit file system access. Data access controls: try to use stored procedures only, so you can drop most privilege grants to database users – prevents SQL injection. Exploit your platform: Most Unix variants have “trusted computing base” extensions which include access control lists. Windows has them out of the box. Use them!
- Always prefer to write less code in applications, particularly when frameworks provide high quality alternatives.
- If custom code is required, consider positive authentication issues (see section 0) and exception handling – ensure that if an exception is thrown, the user is logged out or at least prevented from accessing the protected resource or function.
- Ensure that coverage approaches 100% by default.
- Code a library of authorization checks
- Standardize on calling one or more of the authorization checks
- Either use the inbuilt authorization checks of the framework, or place the call to a centralized authorization check right at the top of the view or action.
- When your application is satisfied that a user is authenticated, associate the session ID with the authentication tokens, flags or state. For example, once the user is logged in, a flag with their authorization levels is set in the session object.
- Check your application:
    - Does not set any client-side authentication or authorization tokens in headers, cookies, hidden form fields, or in URL arguments.
    - Does not trust any client-side authentication or authorization tokens (often in old code)
- If your application uses an SSO agent, such as IBM’s Tivoli Access Manager, Netegrity’s SiteMinder, or RSA’s ClearTrust, ensure your application validates the agent tokens rather than simply accepting them, and ensure these tokens are not visible to the end user in any form (header, cookie, hidden fields, etc). If the tokens are visible to the end user, ensure that all the properties of a cryptographically secure session handler as per chapter 12 are taken into account.
- Use model code instead of direct access to protected resources
- Ensure that model code checks the logged in user has access to the resource

- Ensure that the code asking for the resource has adequate error checking and does not assume that access will always be granted
- Best - generate the static content on the fly and send directly to the browser rather than saving to the web server's file system
- If protecting static sensitive content, implement authorization checks to prevent anonymous access
- If you have to save to disk (not recommended), use random filenames (such as a GUID) and clean up temporary files regularly

Session Management

Best practices

Best practice is to not re-write the wheel, but to use a robust, well-known session manager. Most popular web application frameworks contain a suitable implementation. However, early versions often had significant weaknesses. Always use the latest version of your chosen technology, as its session handler will likely be more robust and use cryptographically strong tokens. Use your favorite search engine to determine if this is indeed the case.

Consider carefully where you store application state:

- Authorization and role data should be stored on the server side only
- Navigation data is almost certainly acceptable in the URL as long as it can be validated and authorization checks are effective
- Presentation flags (such as theme or user language) can belong in cookies.
- Form data should not contain hidden fields – if it is hidden, it probably needs to be protected and only available on the server side. However, hidden fields can (and should) be used for sequence protection and to prevent brute force pharming attacks
- Data from multi-page forms can be sent back to the user in two cases:
- When there are integrity controls to prevent tampering
- When data is validated after every form submission, or at least by the end of the submission process
- Application secrets (such as server-side credentials and role information) should never be visible to the client. These must be stored in a session or server-side accessible way
- If in doubt, do not take a chance and stash it in a session.

How to protect yourself

- Set the idle timeout to 5 minutes for highly protected applications through to no more than 20 minutes for low risk applications
- For highly protected applications:
    - Do not write idle defeat mechanisms
    - Do not write "remember me" functionality
- To reduce the risk from session hijacking and brute force attacks, the HTTP server can seamlessly expire and regenerate tokens. This shortens the window of opportunity for a replay or brute force attack. Token regeneration should be performed based on number of requests (high value sites) or as a function of time, say every 20 minutes.
- Many websites have prohibitions against unrestrained password guessing (e.g., it can

temporarily lock the account or stop listening to the IP address), however an attacker can often try hundreds or thousands of session tokens embedded in a legitimate URL or cookie without a single complaint from the web site. Many intrusion-detection systems do not actively look for this type of attack; penetration tests also often overlook this weakness in web e-commerce systems. Designers can use «booby trapped» session tokens that never actually get assigned but will detect if an attacker is trying to brute force a range of tokens.

- Resulting actions could be:
  - Go slow or ban the originating IP address (which can be troublesome as more and more ISPs are using transparent caches to reduce their costs. Because of this: always check the «proxy_via» header)
  - Lock out an account if you're aware of it (which may cause a user a potential denial of service).
  - Anomaly/misuse detection hooks can also be built in to detect if an authenticated user tries to manipulate their token to gain elevated privileges.
  - There are Apache web server modules, such as mod_dosevasive and mod_security, that could be used for this kind of protection. Although mod_dosevasive is used to lessen the effect of DoS attacks, it could be rewritten for other purposes as well
- When the user logs out of the application:
  - Destroy the session
  - Overwrite session cookies.
  - Provide a method for users to log out of the application. Logging out should clear all session state and remove or invalidate any residual cookies.
  - Set short expiry times on persistent cookies, no more than a day or preferably use non-persistent cookies.
  - Do not store session tokens in the URL or other trivially modified data entry point.
- Always check that the currently logged on user has the authorization to access, update or delete data or access certain functions.
- Ensure that the frameworks' session ID can only be obtained from the cookie value. This may require changing the default behavior of the application framework, or overriding the session handler.
- Use session fixation controls (see next section) to strongly tie a single browser to a single session
- Don't assume a valid session equals logged in – keep the session authorization state secret and check authorization on every page or entry point.
- Use SSL, especially for sites with privacy or high value transactions. One of the properties of SSL is that it authenticates the server to the clients, with most browsers objecting to certificates that do not have adequate intermediate trust or if the certificate does not match the DNS address of the server. This is not full protection, but it will defeat many naive attacks. Sensitive site operations, such as administration, logon, and private data viewing / updating should be protected by SSL in any case.
  - Use a cryptographically sound token generation algorithm. Do not create your own algorithm, and seed the algorithm in a safe fashion. Or just use your application framework's session management functions.
  - Preferably send the token to the client in a non-persistent cookie or within a hidden form field within the rendered page.
  - Put in «telltale» token values so you can detect brute forcing.
  - Limit the number of unique session tokens you see from the same IP address (ie 20 in the last five minutes).
  - Periodically regenerate tokens to reduce the window of opportunity for brute forcing.
  - If you are able to detect a brute force attempt, completely clear session state to prevent that session from being used again.

    - Tie the session to a particular browser by using a hash of the server-side IP address (REMOTE_ADDR) and if the header exists, PROXY_FORWARDED_FOR. Note that you shouldn't use the client-forgeable headers, but take a hash of them. If the new hash doesn't match the previous hash, then there is a high likelihood of session replay.
    - Use session token timeouts and token regeneration to reduce the window of opportunity to replay tokens.
    - Use a cryptographically well-seeded pseudo-random number generator to generate session tokens.
    - Use non-persistent cookies to store the session token, or at worst, a hidden field on the form.
    - Implement a logout function for the application. When logging off a user or idle expiring the session, ensure that not only is the client-side cookie cleared (if possible), but also the server side session state for that browser is also cleared. This ensures that session replay attacks cannot occur after idle timeout or user logoff.

Interpreter Injection

- Avoiding client side document rewriting, redirection, or other sensitive actions, using client side data. Most of these effects can be achieved by using dynamic pages (server side).
- Analyzing and hardening the client side (Javascript) code. Reference to DOM objects that may be influenced by the user (attacker) should be inspected, including (but not limited to):
    - document.URL
    - document.URLUnencoded
    - document.location (and many of its properties)
    - document.referrer
    - window.location (and many of its properties)
    - Note that a document object property or a window object property may be referenced syntactically in many ways - explicitly (e.g. window.location), implicitly (e.g. location), or via obtaining a handle to a window and using it (e.g. handle_to_some_window.location).
- Special attention should be given to scenarios wherein the DOM is modified, either explicitly or potentially, either via raw access to the HTML or via access to the DOM itself, e.g. (by no means an exhaustive list, there are probably various browser extensions):
- Write raw HTML, e.g.:
    - document.write(…)
    - document.writeln(…)
    - document.body.innerHtml=…
    - Directly modifying the DOM (including DHTML events), e.g.:
    - document.forms[0].action=… (and various other collections)
    - document.attachEvent(…)
    - document.create…(…)
    - document.execCommand(…)
    - document.body. … (accessing the DOM through the body object)
    - window.attachEvent(…)
- Replacing the document URL, e.g.:
    - document.location=… (and assigning to location's href, host and hostname)
    - document.location.hostname=…
    - document.location.replace(…)
    - document.location.assign(…)
    - document.URL=…
    - window.navigate(…)

- Opening/modifying a window, e.g.:
  - document.open(…)
  - window.open(…)
  - window.location.href=… (and assigning to location's href, host and hostname)
- Directly executing script, e.g.:
  - eval(…)
  - window.execScript(…)
  - window.setInterval(…)
  - window.setTimeout(…)
- Input validation should be used to remove suspicious characters, preferably by strong validation strategies; it is always better to ensure that data does not have illegal characters to start with.
- Investigate all uses of HTTP headers, such as
  - setting cookies
  - using location (or redirect() functions)
  - setting mime-types, content-type, file size, etc
  - or setting custom headers
- If these contain unvalidated user input, the application is vulnerable when used with application frameworks that cannot detect this issue.
- If the application has to use user-supplied input in HTTP headers, it should check for double "\n" or "\r\n" values in the input data and eliminate it.
- Many application servers and frameworks have basic protection against HTTP response splitting, but it is not adequate to task, and you should not allow unvalidated user input in HTTP headers.
- This requires the following characters to be removed (ie prohibited) or properly escaped:
  - < > / ' = « to prevent straight parameter injection
  - XPath queries should not contain any meta characters (such as ' = * ? / or similar)
  - XSLT expansions should not contain any user input, or if they do, that you comprehensively test the existence of the file, and ensure that the files are within the bounds set by the Java 2 Security Policy
- A suitable canonical form should be chosen and all user input canonicalized into that form before any authorization decisions are performed. Security checks should be carried out after UTF-8 decoding is completed. Moreover, it is recommended to check that the UTF-8 encoding is a valid canonical encoding for the symbol it represents.
- Determine your application's needs, and set both the asserted language locale and character set appropriately.
- Review and implement these guidelines: http://www.w3.org/International/technique-index At a minimum, select the correct output locale and character set.
- Assert the correct locale and character set for your application
- Use HTML entities, URL encoding and so on to prevent Unicode characters being treated improperly by the many divergent browser, server and application combinations
- Test your code and overall solution extensively
- Minimize the total number of components that may interpret the inbound HTTP request
- Keep your infrastructure up to date with patches

Error Handling, Auditing and Logging

Best practices

- Fail safe – do not fail open

- Dual purpose logs
- Audit logs are legally protected – protect them
- Reports and search logs using a read-only copy or complete replica

How to protect yourself

- Приложение должно иметь «безопасный режим», в котором возвращается информация обо всем, что происходит в системе.
- Продуктивная среда не должна генерировать отладочные сообщения. Режим отладки должен активироваться путем редактирования файла конфигурации на сервере. В частности, режим отладки не должен включаться в самом приложении.
- If the framework or language has a structured exception handler (ie try {} catch {}), it should be used in preference to functional error handling
- If the application uses functional error handling, its use must be comprehensive and thorough
- Подробные сообщения об ошибках, такие как трассировки стека или утечки конфиденциальной информации никогда не должны быть представлены пользователю. Вместо этого следует использовать общее сообщение об ошибке. Это включает в себя коды ответа HTTP статуса (т.е. 404 или 500 Внутренняя ошибка сервера).
- Логи должны записывать только новую информацию (старые записи не должны перезаписываться или удаляться). Для дополнительной безопасности, логи стоит записывать на устройства одноразовой записи и многоразового чтения, таких как CD-R.
- Копии логов должны создаваться регулярно в зависимости от объема и размера (ежедневно, еженедельно, ежемесячно...) A common naming convention should be adopted with regards to logs, making them easier to index. На удивление часто забывают проверять, что логирование по-прежнему работает. Это достигается с помощью cron.
- Убедитесь, что данные не перезаписываются.
- Файлы логирования должны копироваться и перемещаться в постоянные места хранения и включены в общую стратегию резервного копирования организации.
- Log files and media should be deleted and disposed of properly and incorporated into an organization's shredding or secure media disposal plan. Reports should be generated on a regular basis, including error reporting and anomaly detection trending.
- Обязательно хранить журналы безопасно и конфиденциально, даже при резервном копировании.
- Logs can be fed into real time intrusion detection and performance and system monitoring tools. All logging components should be synced with a timeserver so that all logging can be consolidated effectively without latency errors. This time server should be hardened and should not provide any other services to the network.
- Никаких манипуляций и удалений во время анализа.
- Logs are useful in reconstructing events after a problem has occurred, security related or not. Event reconstruction can allow a security administrator to determine the full extent of an intruder's activities and expedite the recovery process.
- Logs may in some cases be needed in legal proceedings to prove wrongdoing. In this case, the actual handling of the log data is crucial.
- Logs are often the only record that suspicious behavior is taking place: Therefore logs can sometimes be fed real-time directly into intrusion detection systems.
- Repetitive polls can be protocol led so that network outages or server shutdowns get protocolled and the behavior can either be analyzed later on or a responsible person can take immediate actions.
- Application developers sometimes write logs to prove to customers that their applications are behaving as expected.

- Logs can provide individual accountability in the web application system universe by tracking a user's actions.
- It can be corporate policy or local law to be required to (as example) save header information of all application transactions. These logs must then be kept safe and confidential for six months before they can be deleted.
- The points from above show all different motivations and result in different requirements and strategies. This means, that before we can implement a logging mechanism into an application or system, we have to know the requirements and their later usage. If we fail in doing so this can lead to unintentional results.
- Failure to enable or design the proper event logging mechanisms in the web application may undermine an organization's ability to detect unauthorized access attempts, and the extent to which these attempts may or may not have succeeded. We will look into the most common attack methods, design and implementation errors as well as the mitigation strategies later on in this chapter.
- There is another reason why the logging mechanism must be planned before implementation. In some countries, laws define what kind of personal information is allowed to be not only logged but also analyzed. For example, in Switzerland, companies are not allowed to log personal information of their employees (like what they do on the internet or what they write in their emails). So if a company wants to log a workers surfing habits, the corporation needs to inform her of their plans in advance.
- This leads to the requirement of having anonymized logs or de-personalized logs with the ability to re-personalized them later on if need be. If an unauthorized person has access to (legally) personalized logs, the corporation is acting unlawful again. So there can be a few (not only) legal traps that must be kept in mind.
- Logs can contain different kinds of data. The selection of the data used is normally affected by the motivation leading to the logging. This section contains information about the different types of logging information and the reasons why we could want to log them.
- In general, the logging features include appropriate debugging information’s such as time of event, initiating process or owner of process, and a detailed description of the event. The following are types of system events that can be logged in an application. It depends on the particular application or system and the needs to decide which of these will be used in the logs:
- Reading of data file access and what kind of data is read. This not only allows to see if data was read but also by whom and when.
- Writing of data logs also where and with what mode (append, replace) data was written. This can be used to see if data was overwritten or if a program is writing at all.
- Modification of any data characteristics, including access control permissions or labels, location in database or file system, or data ownership. Administrators can detect if their configurations were changed.
- Administrative functions and changes in configuration regardless of overlap (account management actions, viewing any user's data, enabling or disabling logging, etc.)
- Miscellaneous debugging information that can be enabled or disabled on the fly.
- All authorization attempts (include time) like success/failure, resource or function being authorized, and the user requesting authorization. We can detect password guessing with these logs. These kinds of logs can be fed into an Intrusion Detection system that will detect anomalies.
- Deletion of any data (object). Sometimes applications are required to have some sort of versioning in which the deletion process can be cancelled.
- Network communications (bind, connect, accept, etc.). With this information an Intrusion Detection system can detect port scanning and brute force attacks.
- All authentication events (logging in, logging out, failed logins, etc.) that allow to detect brute force and guessing attacks too.

- Intentionally invoking security errors to fill an error log with entries (noise) that hide the incriminating evidence of a successful intrusion. When the administrator or log parser application reviews the logs, there is every chance that they will summarize the volume of log entries as a denial of service attempt rather than identifying the 'needle in the haystack'.
- This is difficult since applications usually offer an unimpeded route to functions capable of generating log events. If you can deploy an intelligent device or application component that can shun an attacker after repeated attempts, then that would be beneficial. Failing that, an error log audit tool that can reduce the bulk of the noise, based on repetition of events or originating from the same source for example. It is also useful if the log viewer can display the events in order of severity level, rather than just time based.
- The top prize in logging mechanism attacks goes to the contender who can delete or manipulate log entries at a granular level, «as though the event never even happened!». Intrusion and deployment of rootkits allows an attacker to utilize specialized tools that may assist or automate the manipulation of known log files. In most cases, log files may only be manipulated by users with root / administrator privileges, or via approved log manipulation applications. As a general rule, logging mechanisms should aim to prevent manipulation at a granular level since an attacker can hide their tracks for a considerable length of time without being detected. Simple question; if you were being compromised by an attacker, would the intrusion be more obvious if your log file was abnormally large or small, or if it appeared like every other day's log?
- Assign log files the highest security protection, providing reassurance that you always have an effective 'black box' recorder if things go wrong. This includes:
- Applications should not run with Administrator, or root-level privileges. This is the main cause of log file manipulation success since super users typically * have full file system access. Assume the worst case scenario and suppose your application is exploited. Would there be any other security layers in place to prevent the application's user privileges from manipulating the log file to cover tracks?
- Ensuring that access privileges protecting the log files are restrictive, reducing the majority of operations against the log file to alter and read.
- Ensuring that log files are assigned object names that are not obvious and stored in a safe location of the file system.
- Writing log files using publicly or formally scrutinized techniques in an attempt to reduce the risk associated with reverse engineering or log file manipulation.
- Writing log files to read-only media (where event log integrity is of critical importance).
- Use of hashing technology to create digital fingerprints. The idea being that if an attacker does manipulate the log file, then the digital fingerprint will not match and an alert generated.
- Use of host-based IDS technology where normal behavioral patterns can be 'set in stone'. Attempts by attackers to update the log file through anything but the normal approved flow would generate an exception and the intrusion can be detected and blocked. This is one security control that can safeguard against simplistic administrator attempts at modifications.
- Taking cue from the classic 1966 film «How to Steal a Million», or similarly the fable of Aesop; «The Boy Who Cried Wolf», be wary of repeated false alarms, since this may represent an attacker's actions in trying to fool the security administrator into thinking that the technology is faulty and not to be trusted until it can be fixed.
- Simply be aware of this type of attack, take every security violation seriously, always get to the bottom of the cause event log errors rather, and don't just dismiss errors unless you can be completely sure that you know it to be a technical problem.
- By repeatedly hitting an application with requests that cause log entries, multiply this by ten thousand, and the result is that you have a large log file and a possible headache for the security administrator. Where log files are configured with a fixed allocation size, then once full, all logging will stop and an attacker has effectively denied service to your logging mechanism.

Worse still, if there is no maximum log file size, then an attacker has the ability to completely fill the hard drive partition and potentially deny service to the entire system. This is becoming more of a rarity though with the increasing size of today's hard disks.
- The main defense against this type of attack are to increase the maximum log file size to a value that is unlikely to be reached, place the log file on a separate partition to that of the operating system or other critical applications and best of all, try to deploy some kind of system monitoring application that can set a threshold against your log file size and/or activity and issue an alert if an attack of this nature is underway.
- Following the same scenario as the Denial of Service above, if a log file is configured to cycle round overwriting old entries when full, then an attacker has the potential to do the evil deed and then set a log generation script into action in an attempt to eventually overwrite the incriminating log entries, thus destroying them.
- If all else fails, then an attacker may simply choose to cover their tracks by purging all log file entries, assuming they have the privileges to perform such actions. This attack would most likely involve calling the log file management program and issuing the command to clear the log, or it may be easier to simply delete the object which is receiving log event updates (in most cases, this object will be locked by the application). This type of attack does make an intrusion obvious assuming that log files are being regularly monitored, and does have a tendency to cause panic as system administrators and managers realize they have nothing upon which to base an investigation on.
- Following most of the techniques suggested above will provide good protection against this attack. Keep in mind two things:
- Administrative users of the system should be well trained in log file management and review. 'Ad-hoc' clearing of log files is never advised and an archive should always be taken. Too many times a log file is cleared, perhaps to assist in a technical problem, erasing the history of events for possible future investigative purposes.
- An empty security log does not necessarily mean that you should pick up the phone and fly the forensics team in. In some cases, security logging is not turned on by default and it is up to you to make sure that it is. Also, make sure it is logging at the right level of detail and benchmark the errors against an established baseline in order measure what is considered 'normal' activity.

File System

Best Practices

- Use “chroot” jails on Unix platforms
- Use minimal file system permissions on all platforms
- Consider the use of read-only file systems (such as CD-ROM or locked USB key) if practical

How to protect yourself

- Ensure or recommend that the underlying operating system and web application environment are kept up to date
- Ensure the application files and resources are read-only
- Ensure the application does not take user supplied file names when saving or working on local files
- Ensure the application properly checks all user supplied input to prevent additional commands cannot be run

- Ensure that all behind the scenes programs check user input prior to operating on it
- Run the application with the least privilege – in particular, the batch application should not require write privileges to any front end files, the network, or * similar
- Use inbuilt language or operating system features to curtail the resources and features which the background application may use. For example, batch programs rarely if ever require network access.
- Consider the use of host based intrusion detection systems and anti-virus systems to detect unauthorized file creation.
- Remove or move all files that do not belong in the web root
- Rename include files to be normal extension (such as foo.inc foo.jsp or foo.aspx)
- Map all files that need to remain, such as .xml or .cfg to an error handler or a renderer that will not disclose the file contents. This may need to be done in both the web application framework's configuration area or the web server's configuration.
- Temporary file usage is not always important to protect from unauthorized access. For medium to high-risk usage, particularly if the files expose the inner workings of your application or exposes private user data, the following controls should be considered:
- The temporary file routines could be re-written to generate the content on the fly rather than storing on the file system
- Ensure that all resources are not retrievable by unauthenticated users, and that users are authorized to retrieve only their own files
- Use a "garbage collector" to delete old temporary files, either at the end of a session or within a timeout period, such as 20 minutes
- If deployed under Unix like operating systems, use chroot jails to isolate the application from the primary operating system. On Windows, use the inbuilt ACL support to prevent the IIS users from retrieving or overwriting the files directly
- Move the files to outside the web root to prevent browser-only attacks
- Use random file names to decrease the likelihood of a brute force pharming attack
- Use source code control to prevent the need to keep old copies of files around
- Periodically ensure that all files in the web root are actually required
- Ensure that the application's temporary files are not accessible from the web root

Buffer Overflows

- Deploy on systems capable of using no execute stacks (AMD and Intel x86-64 chips with associated 64 bit operating systems: XP SP2 (both 32 and 64 bit), Windows 2003 SP1 (both 32 and 64 bit), Linux after 2.6.8 on AMD and x86-64 processors in 32 and 64 bit mode, OpenBSD (w^x on Intel, AMD, Sparc, Alpha and PowerPC), Solaris 2.6 and later with noexec_user_stack enabled)
- Use programming languages other than C or C++
- Validate user input to prevent overlong input and check the values to ensure they are within spec (ie A-Z, a-z, 0-9, etc)
- If relying upon operating systems and utilities written in C or C++, ensure that they use the principle of least privilege, use compilers that can protect against stack and heap overflows, and keep the system up to date with patches.
- Use programming languages other than C or C++
- Validate user input to prevent overlong input and check the values to ensure they are within spec (ie A-Z, a-z, 0-9, etc)
- If relying upon operating systems and utilities written in C or C++, ensure that they use the principle of least privilege, use compilers which can protect against heap overflows, and keep the system up to date with patches.

- Use programming languages other than C or C++
- Avoid the use of functions like printf() and friends which allow user input to modify the output format
- Validate user inp * ut to prevent format string meta characters from being used in input
- If relying upon operating systems and utilities written in C or C++, ensure that they use the principle of least privilege, deploy on systems with no execute stacks (not a complete protection), and keep the system up to date with patches.
- Keep up with the latest bug reports for your web and application server products and other products in your Internet infrastructure. Apply the latest patches to these products.
- Periodically scan your website with one or more of the commonly available scanners that look for buffer overflow flaws in your server products and your custom web applications.
- Review your code for Unicode exploits
- For your custom application code, you need to review all code that accepts input from untrusted sources, and ensure that it provides appropriate size checking on all such inputs.
- This should be done even for environments that are not susceptible to such attacks as overly large inputs that are uncaught may still cause denial of service or other operational problems.
- .NET: Use David LeBlanc's SafeInt<> C++ class or a similar construct
- If your compiler supports it, change the default for integers to be unsigned unless otherwise explicitly stated. Use unsigned whenever you mean it
- Use range checking if your language or framework supports it
- Be careful when using arithmetic operations near small values, particularly if underflow or overflow, signed or other errors may creep in

Administrative Interfaces

Best practices

Administrative interfaces is one of the few controls within the Guide which is legally mandated – Sarbanes Oxley requires administrative functions to be segregated from normal functionality as it is a key fraud control. For organizations that have no need to comply with US law, ISO 17799 also strongly suggests that there is segregation of duties. It is obviously up to the designers to take into account the risk of not complying with SOX or ISO 17799.

- When designing applications, map out administrative functionality and ensure that appropriate access controls and auditing are in place.
- Consider processes – sometimes all that is required is to understand how users may be prevented from using a feature by simple lack of access
- Help desk access is always a middle ground – they need access to assist customers, but they are not administrators.
- Carefully design help desk / moderator / customer support functionality around limited administration capability and segregated application or access if possible
- This is not to say that administrators logging on as users to the primary application are not allowed, but when they do, they should be normal users. An example is a system administrator of a major e-commerce site who also buys or sells using the site.

How to protect yourself

- All systems should code separate applications for administrator and user access. High value

systems should separate these systems to separate hosts, which may not be accessible to the wider Internet without access to management networks, such as via the use of a strongly authenticated VPN or from trusted network operations center
- For high value systems:
- Use strong authentication to log on, and re-authenticate major or dangerous transactions to prevent administrative phishing and session riding attacks.
- Use encryption (such as SSL encrypted web pages) to protect the confidentiality and integrity of the session.

Cryptography

How to protect yourself

Assuming you have chosen an open, standard algorithm, the following recommendations should be considered when reviewing algorithms:

Symmetric:

- Key sizes of 128 bits (standard for SSL) are sufficient for most applications
- Consider 168 or 256 bits for secure systems such as large financial transactions

Asymmetric:

- Key sizes of 1028 bits are sufficient for most personal applications
- 2048 bits should now be the minimum for any sensitive functions
- 4096 bits or higher should be used for secure applications.

Hashes:

- Hash sizes of 128 bits (standard for SSL) are sufficient for most applications
- Consider 168 or 256 bits for secure systems, as many hash functions are currently being revised (see above).

NIST and other standards bodies will provide up to date guidance on suggested key sizes.

- Design your application to cope with new hashes and algorithms
- Include an “algorithmName” or “algorithmVer” attribute with your encrypted data. You may not be able to reverse the values, but you can (over time) convert to stronger algorithms without disrupting existing users.
- Cryptographic keys should be protected as much as is possible with file system permissions. They should be read only and only the application or user directly accessing them should have these rights.
- Certificates should be marked as not exportable when generating the certificate signing request
- Once imported into the key store (CryptoAPI, Certificates snap-in, Java Key Store, etc), the private certificate import file obtained from the certificate provider should be safely destroyed from front-end systems. This file should be safely stored in a safe until required (such as installing or replacing a new front end server)
- Host based intrusion systems should be deployed to monitor access of keys. At least, changes in keys should be monitored.
- Applications should log any changes to keys.

- Pass phrases used to protect keys should be stored in physically secure places; in some environments, it may be necessary to split the pass phrase or password into two components such that two people will be required to authorize access to the key. These physical, manual processes should be tigh * tly monitored and controlled.
- Storage of keys within source code or binaries should be avoided. This not only has consequences if developers have access to source code, but key management will be almost impossible.
- In a typical web environment, web servers themselves will need permission to access the key. This has obvious implications that other web processes or malicious code may also have access to the key. In these cases, it is vital to minimize the functionality of the system and application requiring access to the keys.
- For interactive applications, a sufficient safeguard is to use a pass phrase or password to encrypt the key when stored on disk. This requires the user to supply a password on startup, but means the key can safely be stored in cases where other users may have greater file system privileges.
- Storage of keys in hardware crypto devices is beyond the scope of this document. If you require this level of security, you should really be consulting with crypto specialists.
- We have the possibility to encrypt or otherwise protect data at different levels. Choosing the right place for this to occur can involve looking at both security as well as resource requirements.
- Application: at this level, the actual application performs the encryption or other crypto function. This is the most desirable, but can place additional strain on resources and create unmanageable complexity. Encryption would be performed typically through an API such as the OpenSSL toolkit (www.openssl.com) or operating system provided crypto functions.
- An example would be a S/MIME encrypted email, which is transmitted as encoded text within a standard email. No changes to intermediate email hosts are necessary to transmit the message because we do not require a change to the protocol itself.
- Protocol: at this layer, the protocol provides the encryption service. Most commonly, this is seen in HTTPS, using SSL encryption to protect sensitive web traffic. The application no longer needs to worry about securing data – the web server, at the protocol level now handles this.
- The big advantage of this is that it requires very little of the application, thereby reducing the risk of a bad crypto implementation.
- The main disadvantage here is that we loose visibility inside the protocol. In the example of HTTPS, it may be possible for an attacker to hide malicious requests (an SQL injection attack for example) within SSL; a content scanner may not be able to decode this, letting it pass to the vulnerable web server.
- Network: below the protocol layer, we can use technologies such as Virtual Private Networks (VPN) to protect data. This has many incarnations, the most popular being IPsec (Internet Protocol v6 Security), typically implemented as a protected ‘tunnel’ between two gateway routers. Neither the application nor the protocol needs to be crypto aware – all traffic is encrypted regardless.
- Possible issues at this level are computational and bandwidth overheads on network devices.
- The only way to generate secure authentication tokens is to ensure there is no way to predict their sequence. In other words: true random numbers.
- It could be argued that computers can not generate true random numbers, but using new techniques such as reading mouse movements and key strokes to improve entropy has significantly increased the randomness of random number generators.
- Again, it is critical that you do not try to implement this on your own; use of existing, proven implementations is highly desirable.
- Most operating systems include functions to generate random numbers that can be called from almost any programming language.

- Windows & .NET: On Microsoft platforms including .NET, it is recommended to use the inbuilt CryptGenRandom function (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/seccrypto/security/cryptgenrandom.asp).
- Unix: For all Unix based platforms, OpenSSL is one of the recommended options (www.openssl.org). It features tools and API functions to generate random numbers.
- PHP: mt_rand() uses a Mersenne Twister, but is nowhere near as good as CryptoAPI's secure random number generation options, OpenSSL, or /dev/urandom which is available on many Unix variants. mt_rand() has been noted to produce the same number on some platforms – test prior to deployment. Do not use rand() as it is very weak.
- Java: java.security.SecureRandom within the Java Cryptography Extension (JCE) provides secure random numbers. This should be used in preference to other random number generators.
- Do not cut and paste UUIDs and GUIDs from anything other than the UUIDGEN program or from the UuidCreate() API
- Generate fresh UUIDs or GUIDs for each new program

Secure Deployment

How to protect yourself

- Do not ship the product with any configured accounts
- Do not hard code any backdoor accounts or special access mechanisms
- Sometimes, no password is just as good as a clear text password
- On the Win32 platform, use "TrustedConnection=yes", and create the DSN with a stored credential. The credential is stored as a LSA Secret, which is not perfect, but is better than clear text passwords
- Develop a method to obfuscate the password in some form, such as "encrypting" the name using the hostname or similar within code in a non-obvious way.
- Ask the database developer to provide a library which allows remote connections using a password hash instead of a clear text credential.
- Use SSL, SSH and other forms of encryption (such as encrypted database connections) to prevent data from being intercepted or interfered with over the wire.
- Highly protected applications and any application that has a requirement to encrypt data:
- Passwords should only be stored in a non-reversible format, such as SHA-256 or similar
- Sensitive data like credit cards should be carefully considered – do they have to be stored at all? The PCI guidelines are very strict on the storage of credit card data. We strongly recommend against it.
- Encrypted data should not have the key on the database server.
- The last requirement requires the attacker to take control of two machines to bulk decrypt data. The encryption key should be able to be changed on a regular basis, and the algorithm should be sufficient to protect the data in a temporal timeframe. For example, there is no point in using 40 bit DES today; data should be encrypted using AES-128 or better.
- The application should connect to the database using as low privilege user as is possible
- The application should connect to the database with different credentials for every trust distinction (eg, user, read-only user, guest, administrators) and permissions applied to those tables and databases to prevent unauthorized access and modification
- The application should prefer safer constructs, such as stored procedures which do not require direct table access. Once all access is through stored procedures, access to the tables should be revoked

- Highly protected applications:
  - 1. The database should be on another host, which should be locked down with all current patches deployed and latest database software in use.
  - 2. The application should connect to the database using an encrypted link. If not, the application server and database server must reside in a restricted network with minimal other hosts. Do not deploy the database server in the main office network.

Maintenance

Best Practices

There is a strong inertia to resist patching “working” (but vulnerable) systems. It is your responsibility as a developer to ensure that the user is as safe as is possible and encourage patching vulnerable systems rapidly by ensuring that your patches are comprehensive (ie no more fixes of this type are likely), no regression of previous issues (ie fixes stay fixed), and stable (ie you have performed adequate testing). Supported applications should be regularly maintained, looking for new methods to obviate security controls It is normal within the industry to provide support for n-1 to n-2 versions, so some form of source revision control, such as CVS, ClearCase, or SubVersion will be required to manage security bug fixes to avoid regression errors Updates should be provided in a secure fashion, either by digitally signing packages, or using a message digest which is known to be relatively free from collisions Support policy for security fixes should be clearly communicated to users, to ensure users are aware of which versions are supported for security fixes and when products are due to be end of lifed.

How to protect yourself

- Create and maintain an incident management policy
- Monitor abuse@…
- Monitor Bugtraq and similar mail lists. Use the experience of similar products to learn from their mistakes and fix them before they are found in your own products
- Publish a security section on their web site, with the ability to submit a security incident in a secure fashion (such as exchange of PGP keys or via SSL)
- Have a method of getting security fixes turned around quickly, certainly fully tested within 30 days.
- Ensure that root cause analysis is used to identify the underlying reason for the defect
- Use attack surface area reduction and risk methodologies to remove as many vulnerabilities of this type as is possible within the prescribed time frame or budget
- Preferably, the application should have the ability to “phone home” to check for newer versions and alert system administrators when new versions are available. If this is not possible, for example, in highly protected environments where “phone home” features are not allowed, another method should be offered to keep the administrators up to date.
- The application should regularly review the permissions of key files, directories and resources that contain application secrets to ensure that permissions have not been relaxed. If the permissions expose an immediate danger, the application should stop functioning until the issue is fixed, otherwise, notifying or alerting the administrator should be sufficient.

Denial Of Service attacks

How to protect yourself

- One of the most widely used How to protect yourself is the one mentioned in the session ID brute-forcing mitigation section. The account creation or transaction confirmation page should contain a dynamically generated image, which displays a string that the user must enter in order to continue with the transaction.
- To mitigate against malicious account lockouts of the application users, ensure that it is extremely difficult for an attacker to enumerate valid user accounts in the first place. If for some reason, the application is built in a way where it is not possible to prevent users from knowing the user IDs of other users (say a web-based emails service), then ensure that the after say 3 failed logins, the user must type in the authentication credentials along with the dynamically generated string image. This will drastically slow down an attacker. After 3 more failed logins at this stage, the user's account will be locked out. The application could then provide a means for the user to unlock his/her account using the same mechanism used in the 'forgot password' scheme.
- Another option might be to use mod_throttle with Apache web server, or the RLimitCPU, RlimitMem, RlimitNProc directives of Apache.