HACK STEPS

1. **Review all of the application's authentication-related functionality, as well as any functions relating to user maintenance. If you find any instances in which a user's password is transmitted back to the client, this indicates that passwords are being stored insecurely, either in cleartext or using reversible encryption.**
2. **If any kind of arbitrary command or query execution vulnerability is identified within the application, attempt to find the location within the application's database or filesystem where user credentials are stored:**
   a. **Query these to determine whether passwords are being stored in unencrypted form.**
   b. **If passwords are stored in hashed form, check for nonunique values, indicating that an account has a common or default password assigned, and that the hashes are not being salted.**
   c. **If the password is hashed with a standard algorithm in unsalted form, query online hash databases to determine the corresponding cleartext password value.**

Securing Authentication

Implementing a secure authentication solution involves attempting to simultaneously meet several key security objectives, and in many cases trade off against other objectives such as functionality, usability, and total cost. In some cases "more" security can actually be counterproductive. For example, forcing users to set very long passwords and change them frequently often causes users to write down their passwords.

Because of the enormous variety of possible authentication vulnerabilities, and the potentially complex defenses that an application may need to deploy to mitigate against all of them, many application designers and developers choose to accept certain threats as a given and concentrate on preventing the most serious attacks. Here are some factors to consider in striking an appropriate balance:

- The criticality of security given the functionality that the application offers
- The degree to which users will tolerate and work with different types of authentication controls
- The cost of supporting a less user-friendly system
- The financial cost of competing alternatives in relation to the revenue likely to be generated by the application or the value of the assets it protects

This section describes the most effective ways to defeat the various attacks against authentication mechanisms. We'll leave it to you to decide which kinds of defenses are most appropriate in each case.

Use Strong Credentials

- Suitable minimum password quality requirements should be enforced. These may include rules regarding minimum length; the appearance of alphabetic, numeric, and typographic characters; the appearance of both uppercase and lowercase characters; the avoidance of dictionary words, names, and other common passwords; preventing a password from being set to the username; and preventing a similarity or match with previously set passwords. As with most security measures, different password quality requirements may be appropriate for different categories of user.
- Usernames should be unique.
- Any system-generated usernames and passwords should be created with sufficient entropy that they cannot feasibly be sequenced or predicted — even by an attacker who gains access to a large sample of successively generated instances.
- Users should be permitted to set sufficiently strong passwords. For example, long passwords and a wide range of characters should be allowed.

Handle Credentials Secretively

- All credentials should be created, stored, and transmitted in a manner that does not lead to unauthorized disclosure.
- All client-server communications should be protected using a well-established cryptographic technology, such as SSL. Custom solutions for protecting data in transit are neither necessary nor desirable.
- If it is considered preferable to use HTTP for the unauthenticated areas of the application, ensure that the login form itself is loaded using HTTPS, rather than switching to HTTPS at the point of the login submission.
- Only `POST` requests should be used to transmit credentials to the server. Credentials should never be placed in URL parameters or cookies (even ephemeral ones). Credentials should never be transmitted back to the client, even in parameters to a redirect.
- All server-side application components should store credentials in a manner that does not allow their original values to be easily recovered, even by an attacker who gains full access to all the relevant data within the

application's database. The usual means of achieving this objective is to use a strong hash function (such as SHA-256 at the time of this writing), appropriately salted to reduce the effectiveness of precomputed offline attacks. The salt should be specific to the account that owns the password, such that an attacker cannot replay or substitute hash values.

- Client-side "remember me" functionality should in general remember only nonsecret items such as usernames. In less security-critical applications, it may be considered appropriate to allow users to opt in to a facility to remember passwords. In this situation, no cleartext credentials should be stored on the client (the password should be stored reversibly encrypted using a key known only to the server). Also, users should be warned about risks from an attacker who has physical access to their computer or who compromises their computer remotely. Particular attention should be paid to eliminating cross-site scripting vulnerabilities within the application that may be used to steal stored credentials (see Chapter 12).
- A password change facility should be implemented (see the "Prevent Misuse of the Password Change Function" section), and users should be required to change their password periodically.
- Where credentials for new accounts are distributed to users out-of-band, these should be sent as securely as possible and should be time-limited. The user should be required to change them on first login and should be told to destroy the communication after first use.
- Where applicable, consider capturing some of the user's login information (for example, single letters from a memorable word) using drop-down menus rather than text fields. This will prevent any keyloggers installed on the user's computer from capturing all the data the user submits. (Note, however, that a simple keylogger is only one means by which an attacker can capture user input. If he or she has already compromised a user's computer, in principle an attacker can log every type of event, including mouse movements, form submissions over HTTPS, and screen captures.)

Validate Credentials Properly

- Passwords should be validated in full — that is, in a case-sensitive way, without filtering or modifying any characters, and without truncating the password.
- The application should be aggressive in defending itself against unexpected events occurring during login processing. For example, depending on the development language in use, the application should use catch-all exception handlers around all API calls. These should explicitly delete all

session and method-local data being used to control the state of the login processing and should explicitly invalidate the current session, thereby causing a forced logout by the server even if authentication is somehow bypassed.

- All authentication logic should be closely code-reviewed, both as pseudo-code and as actual application source code, to identify logic errors such as fail-open conditions.
- If functionality to support user impersonation is implemented, this should be strictly controlled to ensure that it cannot be misused to gain unauthorized access. Because of the criticality of the functionality, it is often worthwhile to remove this functionality from the public-facing application and implement it only for internal administrative users, whose use of impersonation should be tightly controlled and audited.
- Multistage logins should be strictly controlled to prevent an attacker from interfering with the transitions and relationships between the stages:
  - All data about progress through the stages and the results of previous validation tasks should be held in the server-side session object and should never be transmitted to or read from the client.
  - No items of information should be submitted more than once by the user, and there should be no means for the user to modify data that has already been collected and/or validated. Where an item of data such as a username is used at multiple stages, this should be stored in a session variable when first collected and referenced from there subsequently.
  - The first task carried out at every stage should be to verify that all prior stages have been correctly completed. If this is not the case, the authentication attempt should immediately be marked as bad.
  - To prevent information leakage about which stage of the login failed (which would enable an attacker to target each stage in turn), the application should always proceed through all stages of the login, even if the user failed to complete earlier stages correctly, and even if the original username was invalid. After proceeding through all the stages, the application should present a generic "login failed" message at the conclusion of the final stage, without providing any information about where the failure occurred.
- Where a login process includes a randomly varying question, ensure that an attacker cannot effectively choose his own question:
  - Always employ a multistage process in which users identify themselves at an initial stage and the randomly varying question is presented to them at a later stage.

- When a given user has been presented with a given varying question, store that question within her persistent user profile, and ensure that the same user is presented with the same question on each attempted login until she successfully answers it.
- When a randomly varying challenge is presented to the user, store the question that has been asked in a server-side session variable, rather than a hidden field in an HTML form, and validate the subsequent answer against that saved question.

NOTE **The subtleties of devising a secure authentication mechanism run deep here. If care is not taken in the asking of a randomly varying question, this can lead to new opportunities for username enumeration. For example, to prevent an attacker from choosing his own question, an application may store within each user's profile the last question that user was asked, and continue presenting that question until the user answers it correctly. An attacker who initiates several logins using any given user's username will be met with the same question. However, if the attacker carries out the same process using an invalid username, the application may behave differently: because no user profile is associated with an invalid username, there will be no stored question, so a varying question will be presented. The attacker can use this difference in behavior, manifested across several login attempts, to infer the validity of a given username. In a scripted attack, he will be able to harvest numerous usernames quickly.**

If an application wants to defend itself against this possibility, it must go to some lengths. When a login attempt is initiated with an invalid username, the application must record somewhere the random question that it presented for that invalid username and ensure that subsequent login attempts using the same username are met with the same question. Going even further, the application could switch to a different question periodically to simulate the nonexistent user's having logged in as normal, resulting in a change in the next question! At some point, however, the application designer must draw a line and concede that a total victory against such a determined attacker probably is not possible.

Prevent Information Leakage

- The various authentication mechanisms used by the application should not disclose any information about authentication parameters, through either overt messages or inference from other aspects of the application's behavior. An attacker should have no means of determining which piece of the various items submitted has caused a problem.
- A single code component should be responsible for responding to all failed login attempts with a generic message. This avoids a subtle vulnerability

that can occur when a supposedly uninformative message returned from different code paths can actually be spotted by an attacker due to typographical differences in the message, different HTTP status codes, other information hidden in HTML, and the like.

- If the application enforces some kind of account lockout to prevent brute-force attacks (as discussed in the next section), be careful not to let this lead to any information leakage. For example, if an application discloses that a specific account has been suspended for *X* minutes due to *Y* failed logins, this behavior can easily be used to enumerate valid usernames. In addition, disclosing the precise metrics of the lockout policy enables an attacker to optimize any attempt to continue guessing passwords in spite of the policy. To avoid enumeration of usernames, the application should respond to *any* series of failed login attempts from the same browser with a generic message advising that accounts are suspended if multiple failures occur and that the user should try again later. This can be achieved using a cookie or hidden field to track repeated failures originating from the same browser. (Of course, this mechanism should not be used to enforce any actual security control — only to provide a helpful message to ordinary users who are struggling to remember their credentials.)
- If the application supports self-registration, it can prevent this function from being used to enumerate existing usernames in two ways:
  - Instead of permitting self-selection of usernames, the application can create a unique (and unpredictable) username for each new user, thereby obviating the need to disclose that a selected username already exists.
  - The application can use e-mail addresses as usernames. Here, the first stage of the registration process requires the user to enter her e-mail address, whereupon she is told simply to wait for an e-mail and follow the instructions contained within it. If the e-mail address is already registered, the user can be informed of this in the e-mail. If the address is not already registered, the user can be provided with a unique, unguessable URL to visit to continue the registration process. This prevents the attacker from enumerating valid usernames (unless he happens to have already compromised a large number of e-mail accounts).

Prevent Brute-Force Attacks

- Measures need to be enforced within all the various challenges implemented by the authentication functionality to prevent attacks that attempt to meet those challenges using automation. This includes the login itself,

as well as functions to change the password, to recover from a forgotten password situation, and the like.

- Using unpredictable usernames and preventing their enumeration presents a significant obstacle to completely blind brute-force attacks and requires an attacker to have somehow discovered one or more specific usernames before mounting an attack.
- Some security-critical applications (such as online banks) simply disable an account after a small number of failed logins (such as three). They also require that the account owner take various out-of-band steps to reactivate the account, such as telephoning customer support and answering a series of security questions. Disadvantages of this policy are that it allows an attacker to deny service to legitimate users by repeatedly disabling their accounts, and the cost of providing the account recovery service. A more balanced policy, suitable for most security-aware applications, is to suspend accounts for a short period (such as 30 minutes) following a small number of failed login attempts (such as three). This serves to massively slow down any password-guessing attack, while mitigating the risk of denial-of-service attacks and also reducing call center work.
- If a policy of temporary account suspension is implemented, care should be taken to ensure its effectiveness:
  - To prevent information leakage leading to username enumeration, the application should never indicate that any specific account has been suspended. Rather, it should respond to any series of failed logins, even those using an invalid username, with a message advising that accounts are suspended if multiple failures occur and that the user should try again later (as just discussed).
  - The policy's metrics should not be disclosed to users. Simply telling legitimate users to "try again later" does not seriously diminish their quality of service. But informing an attacker exactly how many failed attempts are tolerated, and how long the suspension period is, enables him to optimize any attempt to continue guessing passwords in spite of the policy.
  - If an account is suspended, login attempts should be rejected without even checking the credentials. Some applications that have implemented a suspension policy remain vulnerable to brute-forcing because they continue to fully process login attempts during the suspension period, and they return a subtly (or not so subtly) different message when valid credentials are submitted. This behavior enables an effective brute-force attack to proceed at full speed regardless of the suspension policy.

- Per-account countermeasures such as account lockout do not help protect against one kind of brute-force attack that is often highly effective — iterating through a long list of enumerated usernames, checking a single weak password, such as `password`. For example, if five failed attempts trigger an account suspension, this means an attacker can attempt four different passwords on every account without causing any disruption to users. In a typical application containing many weak passwords, such an attacker is likely to compromise many accounts.

  The effectiveness of this kind of attack will, of course, be massively reduced if other areas of the authentication mechanism are designed securely. If usernames cannot be enumerated or reliably predicted, an attacker will be slowed down by the need to perform a brute-force exercise in guessing usernames. And if strong requirements are in place for password quality, it is far less likely that the attacker will choose a password for testing that even a single user of the application has chosen.

  In addition to these controls, an application can specifically protect itself against this kind of attack through the use of CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) challenges on every page that may be a target for brute-force attacks (see Figure 6-9). If effective, this measure can prevent any automated submission of data to any application page, thereby keeping all kinds of password-guessing attacks from being executed manually. Note that much research has been done on CAPTCHA technologies, and automated attacks against them have in some cases been reliable. Furthermore, some attackers have been known to devise CAPTCHA-solving competitions, in which unwitting members of the public are leveraged as drones to assist the attacker. However, even if a particular kind of challenge is not entirely effective, it will still lead most casual attackers to desist and find an application that does not employ the technique.

Figure 6-9: A CAPTCHA control designed to hinder automated attacks

TIP If you are attacking an application that uses CAPTCHA controls to hinder automation, always closely review the HTML source for the page where the image appears. The authors have encountered cases where the solution

to the puzzle appears in literal form within the `ALT` attribute of the image tag, or within a hidden form field, enabling a scripted attack to defeat the protection without actually solving the puzzle itself.

Prevent Misuse of the Password Change Function

- A password change function should always be implemented, to allow periodic password expiration (if required) and to allow users to change passwords if they want to for any reason. As a key security mechanism, this needs to be well defended against misuse.
- The function should be accessible only from within an authenticated session.
- There should be no facility to provide a username, either explicitly or via a hidden form field or cookie. Users have no legitimate need to attempt to change other people's passwords.
- As a defense-in-depth measure, the function should be protected from unauthorized access gained via some other security defect in the application — such as a session-hijacking vulnerability, cross-site scripting, or even an unattended terminal. To this end, users should be required to reenter their existing password.
- The new password should be entered twice to prevent mistakes. The application should compare the "new password" and "confirm new password" fields as its first step and return an informative error if they do not match.
- The function should prevent the various attacks that can be made against the main login mechanism. A single generic error message should be used to notify users of any error in existing credentials, and the function should be temporarily suspended following a small number of failed attempts to change the password.
- Users should be notified out-of-band (such as via e-mail) that their password has been changed, but the message should not contain either their old or new credentials.

Prevent Misuse of the Account Recovery Function

- In the most security-critical applications, such as online banking, account recovery in the event of a forgotten password is handled out-of-band. A user must make a telephone call and answer a series of security questions, and new credentials or a reactivation code are also sent out-of-band (via conventional mail) to the user's registered home address. The majority of applications do not want or need this level of security, so an automated recovery function may be appropriate.

- A well-designed password recovery mechanism needs to prevent accounts from being compromised by an unauthorized party and minimize any disruption to legitimate users.
- Features such as password "hints" should never be used, because they mainly help an attacker trawl for accounts that have obvious hints set.
- The best automated solution for enabling users to regain control of accounts is to e-mail the user a unique, time-limited, unguessable, single-use recovery URL. This e-mail should be sent to the address that the user provided during registration. Visiting the URL allows the user to set a new password. After this has been done, a second e-mail should be sent, indicating that a password change was made. To prevent an attacker from denying service to users by continually requesting password reactivation e-mails, the user's existing credentials should remain valid until they are changed.
- To further protect against unauthorized access, applications may present users with a secondary challenge that they must complete before gaining access to the password reset function. Be sure that the design of this challenge does not introduce new vulnerabilities:
  - The challenge should implement the same question or set of questions for everyone, mandated by the application during registration. If users provide their own challenge, it is likely that some of these will be weak, and this also enables an attacker to enumerate valid accounts by identifying those that have a challenge set.
  - Responses to the challenge should contain sufficient entropy that they cannot be easily guessed. For example, asking the user for the name of his first school is preferable to asking for his favorite color.
  - Accounts should be temporarily suspended following a number of failed attempts to complete the challenge, to prevent brute-force attacks.
  - The application should not leak any information in the event of failed responses to the challenge — regarding the validity of the username, any suspension of the account, and so on.
  - Successful completion of the challenge should be followed by the process described previously, in which a message is sent to the user's registered e-mail address containing a reactivation URL. Under no circumstances should the application disclose the user's forgotten password or simply drop the user into an authenticated session. Even proceeding directly to the password reset function is undesirable. The response to the account recovery challenge will in general be easier for an attacker to guess than the original password, so it should not be relied upon on its own to authenticate the user.

Log, Monitor, and Notify

- The application should log all authentication-related events, including login, logout, password change, password reset, account suspension, and account recovery. Where applicable, both failed and successful attempts should be logged. The logs should contain all relevant details (such as username and IP address) but no security secrets (such as passwords). Logs should be strongly protected from unauthorized access, because they are a critical source of information leakage.
- Anomalies in authentication events should be processed by the application's real-time alerting and intrusion prevention functionality. For example, application administrators should be made aware of patterns indicating brute-force attacks so that appropriate defensive and offensive measures can be considered.
- Users should be notified out-of-band of any critical security events. For example, the application should send a message to a user's registered e-mail address whenever he changes his password.
- Users should be notified in-band of frequently occurring security events. For example, after a successful login, the application should inform users of the time and source IP/domain of the last login and the number of invalid login attempts made since then. If a user is made aware that her account is being subjected to a password-guessing attack, she is more likely to change her password frequently and set it to a strong value.

Summary

Authentication functions are perhaps the most prominent target in a typical application's attack surface. By definition, they can be reached by unprivileged, anonymous users. If broken, they grant access to protected functionality and sensitive data. They lie at the core of the security mechanisms that an application employs to defend itself and are the front line of defense against unauthorized access.

Real-world authentication mechanisms contain a myriad of design and implementation flaws. An effective assault against them needs to proceed systematically, using a structured methodology to work through every possible avenue of attack. In many cases, open goals present themselves — bad passwords, ways to find out usernames, vulnerability to brute-force attacks. At the other end of the spectrum, defects may be very hard to uncover. They may require meticulous examination of a convoluted login process to establish the assumptions being