CHAPTER 6

Attacking Authentication

On the face of it, authentication is conceptually among the simplest of all the security mechanisms employed within web applications. In the typical case, a user supplies her username and password, and the application must verify that these items are correct. If so, it lets the user in. If not, it does not.

Authentication also lies at the heart of an application's protection against malicious attack. It is the front line of defense against unauthorized access. If an attacker can defeat those defenses, he will often gain full control of the application's functionality and unrestricted access to the data held within it. Without robust authentication to rely on, none of the other core security mechanisms (such as session management and access control) can be effective.

In fact, despite its apparent simplicity, devising a secure authentication function is a subtle business. In real-world web applications authentication often is the weakest link, which enables an attacker to gain unauthorized access. The authors have lost count of the number of applications we have fundamentally compromised as a result of various defects in authentication logic.

This chapter looks in detail at the wide variety of design and implementation flaws that commonly afflict web applications. These typically arise because application designers and developers fail to ask a simple question: What could an attacker achieve if he targeted our authentication mechanism? In the majority of cases, as soon as this question is asked in earnest of a particular application, a number of potential vulnerabilities materialize, any one of which may be sufficient to break the application.

Many of the most common authentication vulnerabilities are no-brainers. Anyone can type dictionary words into a login form in an attempt to guess valid passwords. In other cases, subtle defects may lurk deep within the application's processing that can be uncovered and exploited only after painstaking analysis of a complex multistage login mechanism. We will describe the full spectrum of these attacks, including techniques that have succeeded in breaking the authentication of some of the most security-critical and robustly defended web applications on the planet.

Authentication Technologies

A wide range of technologies are available to web application developers when implementing authentication mechanisms:

- HTML forms-based authentication
- Multifactor mechanisms, such as those combining passwords and physical tokens
- Client SSL certificates and/or smartcards
- HTTP basic and digest authentication
- Windows-integrated authentication using NTLM or Kerberos
- Authentication services

By far the most common authentication mechanism employed by web applications uses HTML forms to capture a username and password and submit these to the application. This mechanism accounts for well over 90% of applications you are likely to encounter on the Internet.

In more security-critical Internet applications, such as online banking, this basic mechanism is often expanded into multiple stages, requiring the user to submit additional credentials, such as a PIN or selected characters from a secret word. HTML forms are still typically used to capture relevant data.

In the most security-critical applications, such as private banking for high-worth individuals, it is common to encounter multifactor mechanisms using physical tokens. These tokens typically produce a stream of one-time passcodes or perform a challenge-response function based on input specified by the application. As the cost of this technology falls over time, it is likely that more applications will employ this kind of mechanism. However, many of these solutions do not actually address the threats for which they were devised — primarily phishing attacks and those employing client-side Trojans.

Some web applications employ client-side SSL certificates or cryptographic mechanisms implemented within smartcards. Because of the overhead of administering and distributing these items, they are typically used only in security-critical

contexts where an application's user base is small, such as web-based VPNs for remote office workers.

The HTTP-based authentication mechanisms (basic, digest, and Windows-integrated) are rarely used on the Internet. They are much more commonly encountered in intranet environments where an organization's internal users gain access to corporate applications by supplying their normal network or domain credentials. The application then processes these credentials using one of these technologies.

Third-party authentication services such as Microsoft Passport are occasionally encountered, but at the present time they have not been adopted on any significant scale.

Most of the vulnerabilities and attacks that arise in relation to authentication can be applied to any of the technologies mentioned. Because of the overwhelming dominance of HTML forms-based authentication, we will describe each specific vulnerability and attack in that context. Where relevant, we will point out any specific differences and attack methodologies that are relevant to the other available technologies.

Design Flaws in Authentication Mechanisms

Authentication functionality is subject to more design weaknesses than any other security mechanism commonly employed in web applications. Even in the apparently simple, standard model where an application authenticates users based on their username and password, shortcomings in the design of this model can leave the application highly vulnerable to unauthorized access.

Bad Passwords

Many web applications employ no or minimal controls over the quality of users' passwords. It is common to encounter applications that allow passwords that are:

- Very short or blank
- Common dictionary words or names
- The same as the username
- Still set to a default value

Figure 6-1 shows an example of weak password quality rules. End users typically display little awareness of security issues. Hence, it is highly likely that an application that does not enforce strong password standards will contain a large number of user accounts with weak passwords set. An attacker can easily guess these account passwords, granting him or her unauthorized access to the application.

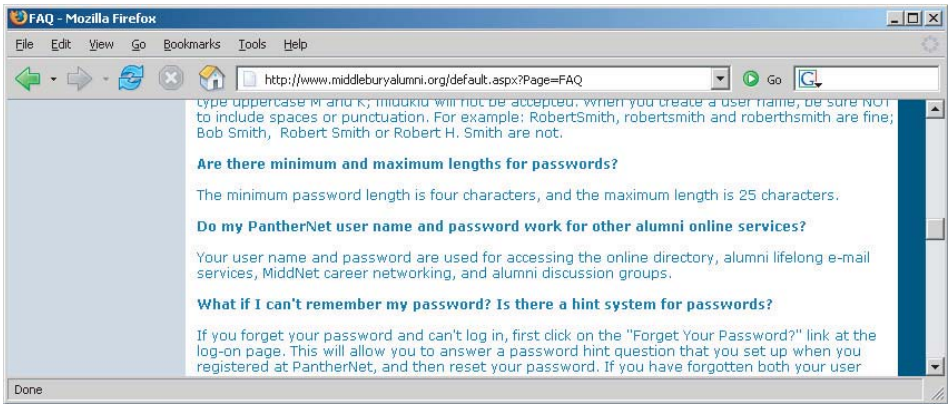

Figure 6-1: An application that enforces weak password quality rules

HACK STEPS

Attempt to discover any rules regarding password quality:

1. **Review the website for any description of the rules.**
2. **If self-registration is possible, attempt to register several accounts with different kinds of weak passwords to discover what rules are in place.**
3. **If you control a single account and password change is possible, attempt to change your password to various weak values.**

NOTE **If password quality rules are enforced only through client-side controls, this is not itself a security issue, because ordinary users will still be protected. It is not normally a threat to an application's security that a crafty attacker can assign himself a weak password.**

TRY IT!

```
http://mdsec.net/auth/217/
```

Brute-Forcible Login

Login functionality presents an open invitation for an attacker to try to guess usernames and passwords and therefore gain unauthorized access to the application. If the application allows an attacker to make repeated login attempts

with different passwords until he guesses the correct one, it is highly vulnerable even to an amateur attacker who manually enters some common usernames and passwords into his browser.

Recent compromises of high-profile sites have provided access to hundreds of thousands of real-world passwords that were stored either in cleartext or using brute-forcible hashes. Here are the most popular real-world passwords:

- `password`
- *`website name`*
- `12345678`
- `qwerty`
- `abc123`
- `111111`
- `monkey`
- `12345`
- `letmein`

NOTE Administrative passwords may in fact be weaker than the password policy allows. They may have been set before the policy was in force, or they may have been set up through a different application or interface.

In this situation, any serious attacker will use automated techniques to attempt to guess passwords, based on lengthy lists of common values. Given today's bandwidth and processing capabilities, it is possible to make thousands of login attempts per minute from a standard PC and DSL connection. Even the most robust passwords will eventually be broken in this scenario.

Various techniques and tools for using automation in this way are described in detail in Chapter 14. Figure 6-2 shows a successful password-guessing attack against a single account using Burp Intruder. The successful login attempt can be clearly distinguished by the difference in the HTTP response code, the response length, and the absence of the "login incorrect" message.

In some applications, client-side controls are employed in an attempt to prevent password-guessing attacks. For example, an application may set a cookie such as `failedlogins=1` and increment it following each unsuccessful attempt. When a certain threshold is reached, the server detects this in the submitted cookie and refuses to process the login attempt. This kind of client-side defense may prevent a manual attack from being launched using only a browser, but it can, of course, be bypassed easily, as described in Chapter 5.

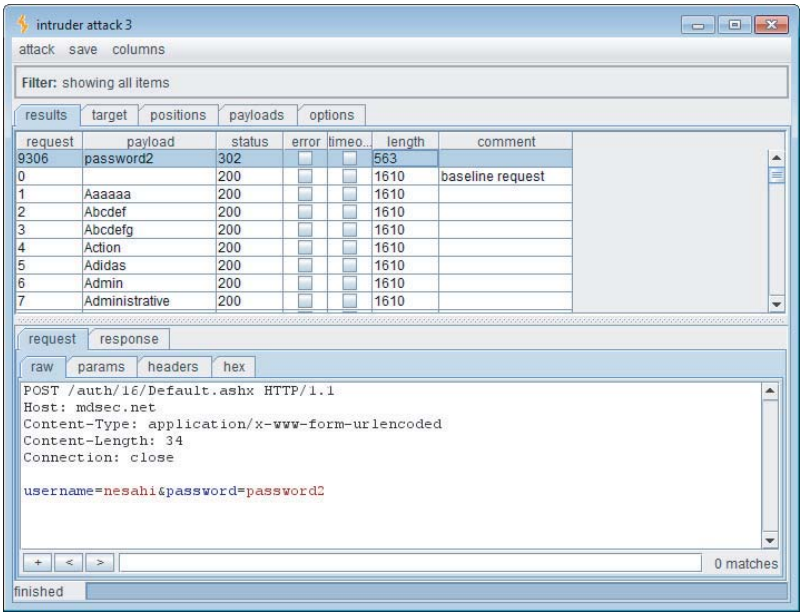

Figure 6-2: A successful password-guessing attack

A variation on the preceding vulnerability occurs when the failed login counter is held within the current session. Although there may be no indication of this on the client side, all the attacker needs to do is obtain a fresh session (for example, by withholding his session cookie), and he can continue his password-guessing attack.

Finally, in some cases, the application locks out a targeted account after a suitable number of failed logins. However, it responds to additional login attempts with messages that indicate (or allow an attacker to infer) whether the supplied password was correct. This means that an attacker can complete his password-guessing attack even though the targeted account is locked out. If the application automatically unlocks accounts after a certain delay, the attacker simply needs to wait for this to occur and then log in as usual with the discovered password.

HACK STEPS

1. **Manually submit several bad login attempts for an account you control, monitoring the error messages you receive.**
2. **After about 10 failed logins, if the application has not returned a message about account lockout, attempt to log in correctly. If this succeeds, there is probably no account lockout policy.**

3. If the account is locked out, try repeating the exercise using a different account. This time, if the application issues any cookies, use each cookie for only a single login attempt, and obtain a new cookie for each subsequent login attempt.
4. Also, if the account is locked out, see whether submitting the valid password causes any difference in the application's behavior compared to an invalid password. If so, you can continue a password-guessing attack even if the account is locked out.
5. If you do not control any accounts, attempt to enumerate a valid username (see the next section) and make several bad logins using this. Monitor for any error messages about account lockout.
6. To mount a brute-force attack, first identify a difference in the application's behavior in response to successful and failed logins. You can use this fact to discriminate between success and failure during the course of the automated attack.
7. Obtain a list of enumerated or common usernames and a list of common passwords. Use any information obtained about password quality rules to tailor the password list so as to avoid superfluous test cases.
8. Use a suitable tool or a custom script to quickly generate login requests using all permutations of these usernames and passwords. Monitor the server's responses to identify successful login attempts. Chapter 14 describes in detail various techniques and tools for performing customized attacks using automation.
9. If you are targeting several usernames at once, it is usually preferable to perform this kind of brute-force attack in a breadth-first rather than depth-first manner. This involves iterating through a list of passwords (starting with the most common) and attempting each password in turn on every username. This approach has two benefits. First, you discover accounts with common passwords more quickly. Second, you are less likely to trigger any account lockout defenses, because there is a time delay between successive attempts using each individual account.

TRY IT!

```
http://mdsec.net/auth/16/
http://mdsec.net/auth/32/
http://mdsec.net/auth/46/
http://mdsec.net/auth/49/
```

Verbose Failure Messages

A typical login form requires the user to enter two pieces of information — a username and password. Some applications require several more, such as date of birth, a memorable place, or a PIN.

When a login attempt fails, you can of course infer that at least one piece of information was incorrect. However, if the application tells you which piece of information was invalid, you can exploit this behavior to considerably diminish the effectiveness of the login mechanism.

In the simplest case, where a login requires a username and password, an application might respond to a failed login attempt by indicating whether the reason for the failure was an unrecognized username or the wrong password, as illustrated in Figure 6-3.

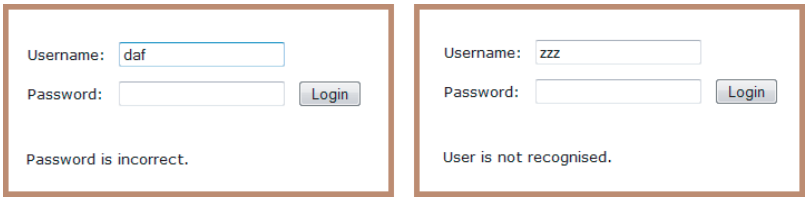

Figure 6-3: Verbose login failure messages indicating when a valid username has been guessed

In this instance, you can use an automated attack to iterate through a large list of common usernames to enumerate which ones are valid. Of course, usernames normally are not considered a secret (they are not masked during login, for instance). However, providing an easy means for an attacker to identify valid usernames increases the likelihood that he will compromise the application given enough time, skill, and effort. A list of enumerated usernames can be used as the basis for various subsequent attacks, including password guessing, attacks on user data or sessions, or social engineering.

In addition to the primary login function, username enumeration can arise in other components of the authentication mechanism. In principle, any function where an actual or potential username is submitted can be leveraged for this purpose. One location where username enumeration is commonly found is the user registration function. If the application allows new users to register and specify their own usernames, username enumeration is virtually impossible to prevent if the application is to prevent duplicate usernames from being registered. Other locations where username enumeration are sometimes found

are the password change and forgotten password functions, as described later in this chapter.

NOTE **Many authentication mechanisms disclose usernames either implicitly or explicitly. In a web mail account, the username is often the e-mail address, which is common knowledge by design. Many other sites expose usernames within the application without considering the advantage this grants to an attacker, or generate usernames in a way that can be predicted (for example, user1842, user1843, and so on).**

In more complex login mechanisms, where an application requires the user to submit several pieces of information, or proceed through several stages, verbose failure messages or other discriminators can enable an attacker to target each stage of the login process in turn, increasing the likelihood that he will gain unauthorized access.

NOTE **This vulnerability may arise in more subtle ways than illustrated here. Even if the error messages returned in response to a valid and invalid username are superficially similar, there may be small differences between them that can be used to enumerate valid usernames. For example, if multiple code paths within the application return the "same" failure message, there may be minor typographical differences between each instance of the message. In some cases, the application's responses may be identical on-screen but contain subtle differences hidden within the HTML source, such as comments or layout differences. If no obvious means of enumerating usernames presents itself, you should perform a close comparison of the application's responses to valid and invalid usernames.**

You can use the Comparer tool within Burp Suite to automatically analyze and highlight the differences between two application responses, as shown in Figure 6-4. This helps you quickly identify whether the username's validity results in any systematic difference in the application's responses.

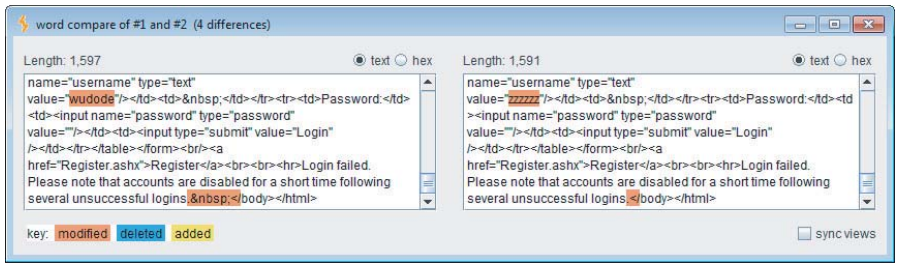

Figure 6-4: Identifying subtle differences in application responses using Burp Comparer

HACK STEPS

1. **If you already know one valid username (for example, an account you control), submit one login using this username and an incorrect password, and another login using a random username.**
2. **Record every detail of the server's responses to each login attempt, including the status code, any redirects, information displayed on-screen, and any differences hidden in the HTML page source. Use your intercepting proxy to maintain a full history of all traffic to and from the server.**
3. **Attempt to discover any obvious or subtle differences in the server's responses to the two login attempts.**
4. **If this fails, repeat the exercise everywhere within the application where a username can be submitted (for example, self-registration, password change, and forgotten password).**
5. **If a difference is detected in the server's responses to valid and invalid usernames, obtain a list of common usernames. Use a custom script or automated tool to quickly submit each username, and filter the responses that signify that the username is valid (see Chapter 14).**
6. **Before commencing your enumeration exercise, verify whether the application performs any account lockout after a certain number of failed login attempts (see the preceding section). If so, it is desirable to design your enumeration attack with this fact in mind. For example, if the application will grant you only three failed login attempts with any given account, you run the risk of "wasting" one of these for every username you discover through automated enumeration. Therefore, when performing your enumeration attack, do not submit a far-fetched password with each login attempt. Instead, submit either a single common password such as *password1* or the username itself as the password. If password quality rules are weak, it is highly likely that some of the attempted logins you perform as part of your enumeration exercise will succeed and will disclose both the username and password in a single hit. To set the password field to be the same as the username, you can use the "battering ram" attack mode in Burp Intruder to insert the same payload at multiple positions in your login request.**

Even if an application's responses to login attempts containing valid and invalid usernames are identical in every intrinsic respect, it may still be possible to enumerate usernames based on the time taken for the application to respond to the login request. Applications often perform very different back-end processing on a login request, depending on whether it contains a valid username. For example, when a valid username is submitted, the application may retrieve user details from a back-end database, perform various processing on these

details (for example, checking whether the account is expired), and then validate the password (which may involve a resource-intensive hash algorithm) before returning a generic message if the password is incorrect. The timing difference between the two responses may be too subtle to detect when working with only a browser, but an automated tool may be able to discriminate between them. Even if the results of such an exercise contain a large ratio of false positives, it is still better to have a list of 100 usernames, approximately 50% of which are valid, than a list of 10,000 usernames, approximately 0.5% of which are valid. See Chapter 15 for a detailed explanation of how to detect and exploit this type of timing difference to extract information from the application.

TIP **In addition to the login functionality itself, there may be other sources of information where you can obtain valid usernames. Review all the source code comments discovered during application mapping (see Chapter 4) to identify any apparent usernames. Any e-mail addresses of developers or other personnel within the organization may be valid usernames, either in full or just the user-specific prefix. Any accessible logging functionality may disclose usernames.**

TRY IT!

```
http://mdsec.net/auth/53/
http://mdsec.net/auth/59/
http://mdsec.net/auth/70/
http://mdsec.net/auth/81/
http://mdsec.net/auth/167/
```

Vulnerable Transmission of Credentials

If an application uses an unencrypted HTTP connection to transmit login credentials, an eavesdropper who is suitably positioned on the network can, of course, intercept them. Depending on the user's location, potential eavesdroppers may reside:

- On the user's local network
- Within the user's IT department
- Within the user's ISP
- On the Internet backbone
- Within the ISP hosting the application
- Within the IT department managing the application

NOTE **Any of these locations may be occupied by authorized personnel but also potentially by an external attacker who has compromised the relevant infrastructure through some other means. Even if the intermediaries on a particular network are believed to be trusted, it is safer to use secure transport mechanisms when passing sensitive data over it.**

Even if login occurs over HTTPS, credentials may still be disclosed to unauthorized parties if the application handles them in an unsafe manner:

- If credentials are transmitted as query string parameters, as opposed to in the body of a `POST` request, these are liable to be logged in various places, such as within the user's browser history, within the web server logs, and within the logs of any reverse proxies employed within the hosting infrastructure. If an attacker succeeds in compromising any of these resources, he may be able to escalate privileges by capturing the user credentials stored there.
- Although most web applications do use the body of a `POST` request to submit the HTML login form itself, it is surprisingly common to see the login request being handled via a redirect to a different URL with the same credentials passed as query string parameters. Why application developers consider it necessary to perform these bounces is unclear, but having elected to do so, it is easier to implement them as 302 redirects to a URL than as `POST` requests using a second HTML form submitted via JavaScript.
- Web applications sometimes store user credentials in cookies, usually to implement poorly designed mechanisms for login, password change, "remember me," and so on. These credentials are vulnerable to capture via attacks that compromise user cookies and, in the case of persistent cookies, by anyone who gains access to the client's local filesystem. Even if the credentials are encrypted, an attacker still can simply replay the cookie and therefore log in as a user without actually knowing her credentials. Chapters 12 and 13 describe various ways in which an attacker can target other users to capture their cookies.

Many applications use HTTP for unauthenticated areas of the application and switch to HTTPS at the point of login. If this is the case, then the correct place to switch to HTTPS is when the login page is loaded in the browser, enabling a user to verify that the page is authentic before entering credentials. However, it is common to encounter applications that load the login page itself using HTTP and then switch to HTTPS at the point where credentials are submitted. This is unsafe, because a user cannot verify the authenticity of the login page itself and therefore has no assurance that the credentials will be submitted securely. A suitably positioned attacker can intercept and modify the login page, changing the target URL of the login form to use HTTP. By the time an astute user realizes that the credentials have been submitted using HTTP, they will have been compromised.

HACK STEPS

1. **Carry out a successful login while monitoring all traffic in both directions between the client and server.**
2. **Identify every case in which the credentials are transmitted in either direction. You can set interception rules in your intercepting proxy to flag messages containing specific strings (see Chapter 20).**
3. **If any instances are found in which credentials are submitted in a URL query string or as a cookie, or are transmitted back from the server to the client, understand what is happening, and try to ascertain what purpose the application developers were attempting to achieve. Try to find every means by which an attacker might interfere with the application's logic to compromise other users' credentials.**
4. **If any sensitive information is transmitted over an unencrypted channel, this is, of course, vulnerable to interception.**
5. **If no cases of actual credentials being transmitted insecurely are identified, pay close attention to any data that appears to be encoded or obfuscated. If this includes sensitive data, it may be possible to reverse-engineer the obfuscation algorithm.**
6. **If credentials are submitted using HTTPS but the login form is loaded using HTTP, the application is vulnerable to a man-in-the-middle attack, which may be used to capture credentials.**

TRY IT!

```
http://mdsec.net/auth/88/
http://mdsec.net/auth/90/
http://mdsec.net/auth/97/
```

Password Change Functionality

Surprisingly, many web applications do not provide any way for users to change their password. However, this functionality is necessary for a well-designed authentication mechanism for two reasons:

- Periodic enforced password change mitigates the threat of password compromise. It reduces the window in which a given password can be targeted in a guessing attack. It also reduces the window in which a compromised password can be used without detection by the attacker.
- Users who suspect that their passwords may have been compromised need to be able to quickly change their password to reduce the threat of unauthorized use.

Although it is a necessary part of an effective authentication mechanism, password change functionality is often vulnerable by design. Vulnerabilities that are deliberately avoided in the main login function often reappear in the password change function. Many web applications' password change functions are accessible without authentication and do the following:

- Provide a verbose error message indicating whether the requested username is valid.
- Allow unrestricted guesses of the "existing password" field.
- Check whether the "new password" and "confirm new password" fields have the same value only after validating the existing password, thereby allowing an attack to succeed in discovering the existing password noninvasively.

A typical password change function includes a relatively large logical decision tree. The application needs to identify the user, validate the supplied existing password, integrate with any account lockout defenses, compare the supplied new passwords with each other and against password quality rules, and feed back any error conditions to the user in a suitable way. Because of this, password change functions often contain subtle logic flaws that can be exploited to subvert the entire mechanism.

HACK STEPS

1. **Identify any password change functionality within the application. If this is not explicitly linked from published content, it may still be implemented. Chapter 4 describes various techniques for discovering hidden content within an application.**
2. **Make various requests to the password change function using invalid usernames, invalid existing passwords, and mismatched "new password" and "confirm new password" values.**
3. **Try to identify any behavior that can be used for username enumeration or brute-force attacks (as described in the "Brute-Forcible Login" and "Verbose Failure Messages" sections).**

TIP **If the password change form is accessible only by authenticated users and does not contain a username field, it may still be possible to supply an arbitrary username. The form may store the username in a hidden field, which can easily be modified. If not, try supplying an additional parameter containing the username, using the same parameter name as is used in the main login form. This trick sometimes succeeds in overriding the username of the current user, enabling you to brute-force the credentials of other users even when this is not possible at the main login.**

TRY IT!

```
http://mdsec.net/auth/104/
http://mdsec.net/auth/117/
http://mdsec.net/auth/120/
http://mdsec.net/auth/125/
http://mdsec.net/auth/129/
http://mdsec.net/auth/135/
```

Forgotten Password Functionality

Like password change functionality, mechanisms for recovering from a forgotten password situation often introduce problems that may have been avoided in the main login function, such as username enumeration.

In addition to this range of defects, design weaknesses in forgotten password functions frequently make this the weakest link at which to attack the application's overall authentication logic. Several kinds of design weaknesses can often be found:

- Forgotten password functionality often involves presenting the user with a secondary challenge in place of the main login, as shown in Figure 6-5. This challenge is often much easier for an attacker to respond to than attempting to guess the user's password. Questions about mothers' maiden names, memorable dates, favorite colors, and the like generally will have a much smaller set of potential answers than the set of possible passwords. Furthermore, they often concern information that is publicly known or that a determined attacker can discover with a modest degree of effort.

Figure 6-5: A secondary challenge used in an account recovery function

In many cases, the application allows users to set their own password recovery challenge and response during registration. Users are inclined

to set extremely insecure challenges, presumably on the false assumption that only they will ever be presented with them. An example is "Do I own a boat?" In this situation, an attacker who wants to gain access can use an automated attack to iterate through a list of enumerated or common usernames, log all the password recovery challenges, and select those that appear most easily guessable. (See Chapter 14 for techniques regarding how to grab this kind of data in a scripted attack.)

- As with password change functionality, application developers commonly overlook the possibility of brute-forcing the response to a password recovery challenge, even when they block this attack on the main login page. If an application allows unrestricted attempts to answer password recovery challenges, it is highly likely to be compromised by a determined attacker.
- In some applications, the recovery challenge is replaced with a simple password "hint" that is configured by users during registration. Users commonly set extremely obvious hints, perhaps even one that is identical to the password itself, on the false assumption that only they will ever see them. Again, an attacker with a list of common or enumerated usernames can easily capture a large number of password hints and then start guessing.
- The mechanism by which an application enables users to regain control of their account after correctly responding to a challenge is often vulnerable. One reasonably secure means of implementing this is to send a unique, unguessable, time-limited recovery URL to the e-mail address that the user provided during registration. Visiting this URL within a few minutes enables the user to set a new password. However, other mechanisms for account recovery are often encountered that are insecure by design:
  - Some applications disclose the existing, forgotten password to the user after successful completion of a challenge, enabling an attacker to use the account indefinitely without any risk of detection by the owner. Even if the account owner subsequently changes the blown password, the attacker can simply repeat the same challenge to obtain the new password.
  - Some applications immediately drop the user into an authenticated session after successful completion of a challenge, again enabling an attacker to use the account indefinitely without detection, and without ever needing to know the user's password.
  - Some applications employ the mechanism of sending a unique recovery URL but send this to an e-mail address specified by the user at the time the challenge is completed. This provides absolutely no enhanced security for the recovery process beyond possibly logging the e-mail address used by an attacker.

TIP **Even if the application does not provide an on-screen field for you to provide an e-mail address to receive the recovery URL, the application may transmit the address via a hidden form field or cookie. This presents a double opportunity: you can discover the e-mail address of the user you have compromised, and you can modify its value to receive the recovery URL at an address of your choosing.**

- Some applications allow users to reset their password's value directly after successful completion of a challenge and do not send any e-mail notification to the user. This means that the compromising of an account by an attacker will not be noticed until the owner attempts to log in again. It may even remain unnoticed if the owner assumes that she must have forgotten her password and therefore resets it in the same way. An attacker who simply desires *some* access to the application can then compromise a different user's account for a period of time and therefore can continue using the application indefinitely.

HACK STEPS

1. **Identify any forgotten password functionality within the application. If this is not explicitly linked from published content, it may still be implemented (see Chapter 4).**
2. **Understand how the forgotten password function works by doing a complete walk-through using an account you control.**
3. **If the mechanism uses a challenge, determine whether users can set or select their own challenge and response. If so, use a list of enumerated or common usernames to harvest a list of challenges, and review this for any that appear easily guessable.**
4. **If the mechanism uses a password "hint," do the same exercise to harvest a list of password hints, and target any that are easily guessable.**
5. **Try to identify any behavior in the forgotten password mechanism that can be exploited as the basis for username enumeration or brute-force attacks (see the previous details).**
6. **If the application generates an e-mail containing a recovery URL in response to a forgotten password request, obtain a number of these URLs, and attempt to identify any patterns that may enable you to predict the URLs issued to other users. Employ the same techniques as are relevant to analyzing session tokens for predictability (see Chapter 7).**

TRY IT!

```
http://mdsec.net/auth/142/
http://mdsec.net/auth/145/
http://mdsec.net/auth/151/
```

"Remember Me" Functionality

Applications often implement "remember me" functions as a convenience to users. This way, users don't need to reenter their username and password each time they use the application from a specific computer. These functions are often insecure by design and leave the user exposed to attack both locally and by users on *other* computers:

- Some "remember me" functions are implemented using a simple persistent cookie, such as `RememberUser=daf` (see Figure 6-6). When this cookie is submitted to the initial application page, the application trusts the cookie to authenticate the user, and it creates an application session for that person, bypassing the login. An attacker can use a list of common or enumerated usernames to gain full access to the application without any authentication.

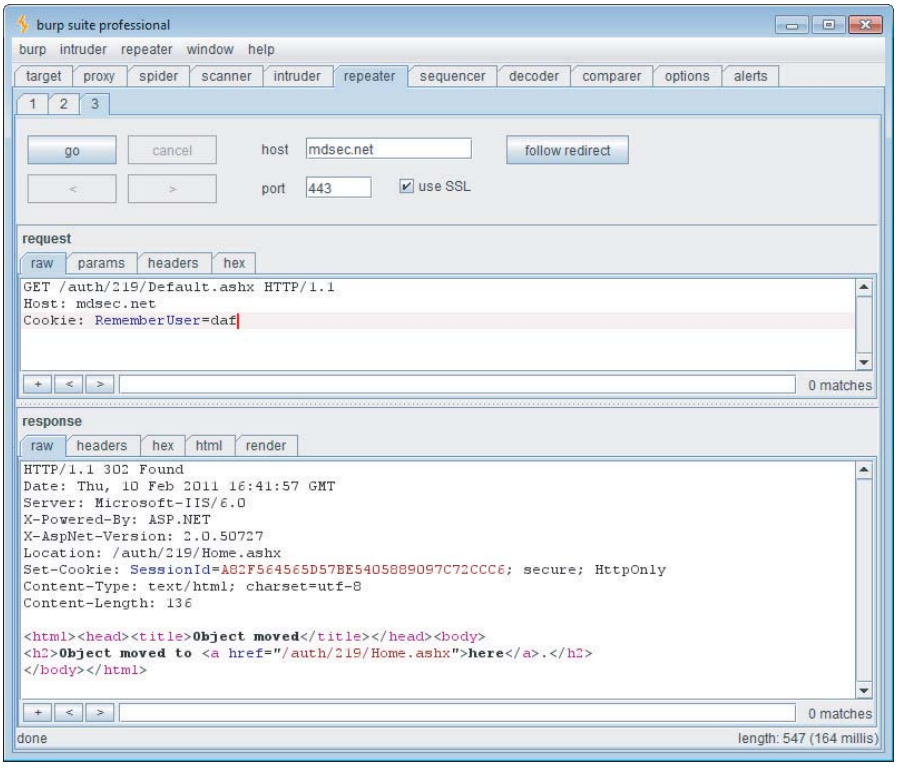

Figure 6-6: A vulnerable "remember me" function, which automatically logs in a user based solely on a username stored in a cookie

- Some "remember me" functions set a cookie that contains not the username but a kind of persistent session identifier, such as `RememberUser=1328`. When the identifier is submitted to the login page, the application looks up the user associated with it and creates an application session for that user. As with ordinary session tokens, if the session identifiers of other users can be predicted or extrapolated, an attacker can iterate through a large number of potential identifiers to find those associated with application users, and therefore gain access to their accounts without authentication. See Chapter 7 for techniques for performing this attack.
- Even if the information stored for reidentifying users is suitably protected (encrypted) to prevent other users from determining or guessing it, the information may still be vulnerable to capture through a bug such as cross-site scripting (see Chapter 12), or by an attacker who has local access to the user's computer.

HACK STEPS

1. **Activate any "remember me" functionality, and determine whether the functionality indeed does fully "remember" the user or whether it remembers only his username and still requires him to enter a password on subsequent visits. If the latter is the case, the functionality is much less likely to expose any security flaw.**
2. **Closely inspect all persistent cookies that are set, and also any data that is persisted in other local storage mechanisms, such as Internet Explorer's userData, Silverlight isolated storage, or Flash local shared objects. Look for any saved data that identifies the user explicitly or appears to contain some predictable identifier of the user.**
3. **Even where stored data appears to be heavily encoded or obfuscated, review this closely. Compare the results of "remembering" several very similar usernames and/or passwords to identify any opportunities to reverse-engineer the original data. Here, use the same techniques that are described in Chapter 7 to detect meaning and patterns in session tokens.**
4. **Attempt to modify the contents of the persistent cookie to try to convince the application that another user has saved his details on your computer.**

TRY IT!

```
http://mdsec.net/auth/219/
http://mdsec.net/auth/224/
http://mdsec.net/auth/227/
http://mdsec.net/auth/229/
http://mdsec.net/auth/232/
http://mdsec.net/auth/236/
http://mdsec.net/auth/239/
http://mdsec.net/auth/245/
```

User Impersonation Functionality

Some applications implement the facility for a privileged user of the application to impersonate other users in order to access data and carry out actions within their user context. For example, some banking applications allow helpdesk operators to verbally authenticate a telephone user and then switch their application session into that user's context to assist him or her.

Various design flaws commonly exist within impersonation functionality:

- It may be implemented as a "hidden" function, which is not subject to proper access controls. For example, anyone who knows or guesses the URL `/admin/ImpersonateUser.jsp` may be able to make use of the function and impersonate any other user (see Chapter 8).
- The application may trust user-controllable data when determining whether the user is performing impersonation. For example, in addition to a valid session token, a user may submit a cookie specifying which account his session is currently using. An attacker may be able to modify this value and gain access to other user accounts without authentication, as shown in Figure 6-7.
- If an application allows administrative users to be impersonated, any weakness in the impersonation logic may result in a vertical privilege escalation vulnerability. Rather than simply gaining access to other ordinary users' data, an attacker may gain full control of the application.
- Some impersonation functionality is implemented as a simple "backdoor" password that can be submitted to the standard login page along with any username to authenticate as that user. This design is highly insecure for many reasons, but the biggest opportunity for attackers is that they are likely to discover this password when performing standard attacks such as brute-forcing of the login. If the backdoor password is matched before the user's actual password, the attacker is likely to discover the function of

the backdoor password and therefore gain access to every user's account. Similarly, a brute-force attack might result in two different "hits," thereby revealing the backdoor password, as shown in Figure 6-8.

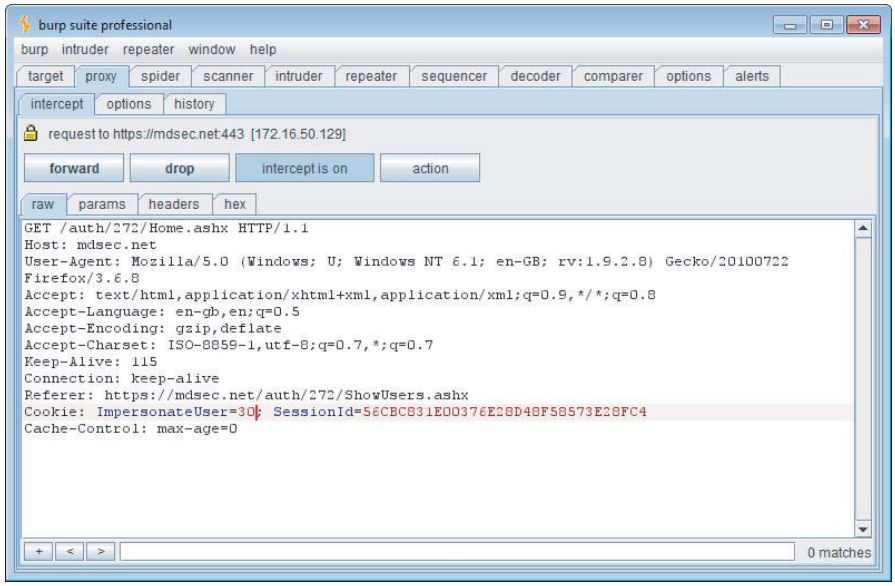

Figure 6-7: A vulnerable user impersonation function

HACK STEPS

1. **Identify any impersonation functionality within the application. If this is not explicitly linked from published content, it may still be implemented (see Chapter 4).**
2. **Attempt to use the impersonation functionality directly to impersonate other users.**
3. **Attempt to manipulate any user-supplied data that is processed by the impersonation function in an attempt to impersonate other users. Pay particular attention to any cases where your username is being submitted other than during normal login.**
4. **If you succeed in making use of the functionality, attempt to impersonate any known or guessed administrative users to elevate privileges.**
5. **When carrying out password-guessing attacks (see the "Brute-Forcible Login" section), review whether any users appear to have more than one valid password, or whether a specific password has been matched against several usernames. Also, log in as many different users with the credentials captured in a brute-force attack, and review whether everything appears normal. Pay close attention to any "logged in as *X*" status message.**

TRY IT!

```
http://mdsec.net/auth/272/
http://mdsec.net/auth/290/
```

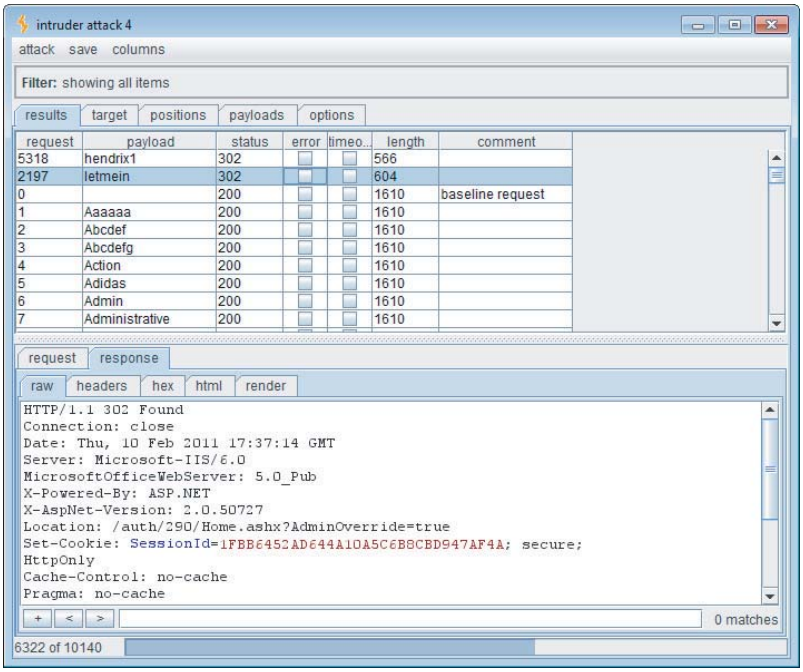

Figure 6-8: A password-guessing attack with two "hits," indicating the presence of a backdoor password

Incomplete Validation of Credentials

Well-designed authentication mechanisms enforce various requirements on passwords, such as a minimum length or the presence of both uppercase and lowercase characters. Correspondingly, some poorly designed authentication mechanisms not only do not enforce these good practices but also do not take into account users' own attempts to comply with them.

For example, some applications truncate passwords and therefore validate only the first *n* characters. Some applications perform a case-insensitive check of passwords. Some applications strip unusual characters (sometimes on the pretext of performing input validation) before checking passwords. In recent times, behavior of this kind has been identified in some surprisingly high-profile web applications, usually as a result of trial and error by curious users.

Each of these limitations on password validation reduces by an order of magnitude the number of variations available in the set of possible passwords. Through experimentation, you can determine whether a password is being fully validated or whether any limitations are in effect. You can then fine-tune your automated attacks against the login to remove unnecessary test cases, thereby massively reducing the number of requests necessary to compromise user accounts.

HACK STEPS

1. **Using an account you control, attempt to log in with variations on your own password: removing the last character, changing the case of a character, and removing any special typographical characters. If any of these attempts is successful, continue experimenting to try to understand what validation is actually occurring.**
2. **Feed any results back into your automated password-guessing attacks to remove superfluous test cases and improve the chances of success.**

TRY IT!

```
http://mdsec.net/auth/293/
```

Nonunique Usernames

Some applications that support self-registration allow users to specify their own username and do not enforce a requirement that usernames be unique. Although this is rare, the authors have encountered more than one application with this behavior.

This represents a design flaw for two reasons:

- One user who shares a username with another user may also happen to select the same password as that user, either during registration or in a subsequent password change. In this eventuality, the application either rejects the second user's chosen password or allows two accounts to have identical credentials. In the first instance, the application's behavior effectively discloses to one user the credentials of the other user. In the second instance, subsequent logins by one of the users result in access to the other user's account.
- An attacker may exploit this behavior to carry out a successful brute-force attack, even though this may not be possible elsewhere due to restrictions on failed login attempts. An attacker can register a specific username

multiple times with different passwords while monitoring for the differential response that indicates that an account with that username and password already exists. The attacker will have ascertained a target user's password without making a single attempt to log in as that user.

Badly designed self-registration functionality can also provide a means for username enumeration. If an application disallows duplicate usernames, an attacker may attempt to register large numbers of common usernames to identify the existing usernames that are rejected.

HACK STEPS

1. **If self-registration is possible, attempt to register the same username twice with different passwords.**
2. **If the application blocks the second registration attempt, you can exploit this behavior to enumerate existing usernames even if this is not possible on the main login page or elsewhere. Make multiple registration attempts with a list of common usernames to identify the already registered names that the application blocks.**
3. **If the registration of duplicate usernames succeeds, attempt to register the same username twice with the same password, and determine the application's behavior:**
   a. **If an error message results, you can exploit this behavior to carry out a brute-force attack, even if this is not possible on the main login page. Target an enumerated or guessed username, and attempt to register this username multiple times with a list of common passwords. When the application rejects a specific password, you have probably found the existing password for the targeted account.**
   b. **If no error message results, log in using the credentials you specified, and see what happens. You may need to register several users, and modify different data held within each account, to understand whether this behavior can be used to gain unauthorized access to other users' accounts.**

Predictable Usernames

Some applications automatically generate account usernames according to a predictable sequence (cust5331, cust5332, and so on). When an application behaves like this, an attacker who can discern the sequence can quickly arrive at a potentially exhaustive list of all valid usernames, which can be used as the basis for further attacks. Unlike enumeration methods that rely on making repeated requests driven by wordlists, this means of determining usernames can be carried out nonintrusively with minimal interaction with the application.

HACK STEPS

1. **If the application generates usernames, try to obtain several in quick succession, and determine whether any sequence or pattern can be discerned.**
2. **If it can, extrapolate backwards to obtain a list of possible valid usernames. This can be used as the basis for a brute-force attack against the login and other attacks where valid usernames are required, such as the exploitation of access control flaws (see Chapter 8).**

TRY IT!

```
http://mdsec.net/auth/169/
```

Predictable Initial Passwords

In some applications, users are created all at once or in sizeable batches and are automatically assigned initial passwords, which are then distributed to them through some means. The means of generating passwords may enable an attacker to predict the passwords of other application users. This kind of vulnerability is more common on intranet-based corporate applications — for example, where every employee has an account created on her behalf and receives a printed notification of her password.

In the most vulnerable cases, all users receive the same password, or one closely derived from their username or job function. In other cases, generated passwords may contain sequences that could be identified or guessed with access to a very small sample of initial passwords.

HACK STEPS

1. **If the application generates passwords, try to obtain several in quick succession, and determine whether any sequence or pattern can be discerned.**
2. **If it can, extrapolate the pattern to obtain a list of passwords for other application users.**
3. **If passwords demonstrate a pattern that can be correlated with usernames, you can try to log in using known or guessed usernames and the corresponding inferred passwords.**
4. **Otherwise, you can use the list of inferred passwords as the basis for a brute-force attack with a list of enumerated or common usernames.**

TRY IT!

```
http://mdsec.net/auth/172/
```

Insecure Distribution of Credentials

Many applications employ a process in which credentials for newly created accounts are distributed to users out-of-band of their normal interaction with the application (for example, via post, e-mail, or SMS text message). Sometimes, this is done for reasons motivated by security concerns, such as to provide assurance that the postal or e-mail address supplied by the user actually belongs to that person.

In some cases, this process can present a security risk. For example, suppose that the message distributed contains both username and password, there is no time limit on their use, and there is no requirement for the user to change the password on first login. It is highly likely that a large number, even the majority, of application users will not modify their initial credentials and that the distribution messages will remain in existence for a lengthy period, during which they may be accessed by an unauthorized party.

Sometimes, what is distributed is not the credentials themselves, but rather an "account activation" URL, which enables users to set their own initial password. If the series of these URLs sent to successive users manifests any kind of sequence, an attacker can identify this by registering multiple users in close succession and then infer the activation URLs sent to recent and forthcoming users.

A related behavior by some web applications is to allow new users to register accounts in a seemingly secure manner and then to send a welcome e-mail to each new user containing his full login credentials. In the worst case, a security-conscious user who decides to immediately change his possibly compromised password then receives another e-mail containing the new password "for future reference." This behavior is so bizarre and unnecessary that users would be well advised to stop using web applications that indulge in it.

HACK STEPS

1. **Obtain a new account. If you are not required to set all credentials during registration, determine the means by which the application distributes credentials to new users.**
2. **If an account activation URL is used, try to register several new accounts in close succession, and identify any sequence in the URLs you receive. If a pattern can be determined, try to predict the activation URLs sent to recent and forthcoming users, and attempt to use these URLs to take ownership of their accounts.**
3. **Try to reuse a single activation URL multiple times, and see if the application allows this. If not, try locking out the target account before reusing the URL, and see if it now works.**

Implementation Flaws in Authentication

Even a well-designed authentication mechanism may be highly insecure due to mistakes made in its implementation. These mistakes may lead to information leakage, complete login bypassing, or a weakening of the overall security of the mechanism as designed. Implementation flaws tend to be more subtle and harder to detect than design defects such as poor-quality passwords and brute-forcibility. For this reason, they are often a fruitful target for attacks against the most security-critical applications, where numerous threat models and penetration tests are likely to have claimed any low-hanging fruit. The authors have identified each of the implementation flaws described here within the web applications deployed by large banks.

Fail-Open Login Mechanisms

Fail-open logic is a species of logic flaw (described in detail in Chapter 11) that has particularly serious consequences in the context of authentication mechanisms.

The following is a fairly contrived example of a login mechanism that fails open. If the call to `db.getUser()` throws an exception for some reason (for example, a null pointer exception arising because the user's request did not contain a username or password parameter), the login succeeds. Although the resulting session may not be bound to a particular user identity and therefore may not be fully functional, this may still enable an attacker to access some sensitive data or functionality.

```
public Response checkLogin(Session session) {
    try {
        String uname = session.getParameter("username");
        String passwd = session.getParameter("password");
        User user = db.getUser(uname, passwd);
        if (user == null) {
            // invalid credentials
            session.setMessage("Login failed. ");
            return doLogin(session);
        }
    }
    catch (Exception e) {}

    // valid user
    session.setMessage("Login successful. ");
    return doMainMenu(session);
}
```

In the field, you would not expect code like this to pass even the most cursory security review. However, the same conceptual flaw is much more likely to exist in more complex mechanisms in which numerous layered method invocations

are made, in which many potential errors may arise and be handled in different places, and where the more complicated validation logic may involve maintaining significant state about the login's progress.

HACK STEPS

1. **Perform a complete, valid login using an account you control. Record every piece of data submitted to the application, and every response received, using your intercepting proxy.**
2. **Repeat the login process numerous times, modifying pieces of the data submitted in unexpected ways. For example, for each request parameter or cookie sent by the client, do the following:**
   a. **Submit an empty string as the value.**
   b. **Remove the name/value pair altogether.**
   c. **Submit very long and very short values.**
   d. **Submit strings instead of numbers and vice versa.**
   e. **Submit the same item multiple times, with the same and different values.**
3. **For each malformed request submitted, review closely the application's response to identify any divergences from the base case.**
4. **Feed these observations back into framing your test cases. When one modification causes a change in behavior, try to combine this with other changes to push the application's logic to its limits.**

TRY IT!

```
http://mdsec.net/auth/300/
```

Defects in Multistage Login Mechanisms

Some applications use elaborate login mechanisms involving multiple stages, such as the following:

- Entry of a username and password
- A challenge for specific digits from a PIN or a memorable word
- The submission of a value displayed on a changing physical token

Multistage login mechanisms are designed to provide enhanced security over the simple model based on username and password. Typically, the first stage requires the users to identify themselves with a username or similar item, and subsequent stages perform various authentication checks. Such mechanisms

frequently contain security vulnerabilities — in particular, various logic flaws (see Chapter 11).

COMMON MYTH

It is often assumed that multistage login mechanisms are less prone to security bypasses than standard username/password authentication. This belief is mistaken. Performing several authentication checks may add considerable security to the mechanism. But counterbalancing this, the process is more prone to flaws in implementation. In several cases where a combination of flaws is present, it can even result in a solution that is *less* secure than a normal login based on username and password.

Some implementations of multistage login mechanisms make potentially unsafe assumptions at each stage about the user's interaction with earlier stages:

- An application may assume that a user who accesses stage three must have cleared stages one and two. Therefore, it may authenticate an attacker who proceeds directly from stage one to stage three and correctly completes it, enabling an attacker to log in with only one part of the various credentials normally required.
- An application may trust some of the data being processed at stage two because this was validated at stage one. However, an attacker may be able to manipulate this data at stage two, giving it a different value than was validated at stage one. For example, at stage one the application might determine whether the user's account has expired, is locked out, or is in the administrative group, or whether it needs to complete further stages of the login beyond stage two. If an attacker can interfere with these flags as the login transitions between different stages, he may be able to modify the application's behavior and cause it to authenticate him with only partial credentials or otherwise elevate privileges.
- An application may assume that the same user identity is used to complete each stage; however, it might not explicitly check this. For example, stage one might involve submitting a valid username and password, and stage two might involve resubmitting the username (now in a hidden form field) and a value from a changing physical token. If an attacker submits valid data pairs at each stage, but for different users, the application might authenticate the user as either one of the identities used in the two stages. This would enable an attacker who possesses his own physical token and discovers another user's password to log in as that user (or vice versa). Although the login mechanism cannot be completely compromised without any prior information, its overall security posture is substantially weakened, and the substantial expense and effort of implementing the two-factor mechanism do not deliver the benefits expected.

HACK STEPS

1. Perform a complete, valid login using an account you control. Record every piece of data submitted to the application using your intercepting proxy.
2. Identify each distinct stage of the login and the data that is collected at each stage. Determine whether any single piece of information is collected more than once or is ever transmitted back to the client and resubmitted via a hidden form field, cookie, or preset URL parameter (see Chapter 5).
3. Repeat the login process numerous times with various malformed requests:
   a. Try performing the login steps in a different sequence.
   b. Try proceeding directly to any given stage and continuing from there.
   c. Try skipping each stage and continuing with the next.
   d. Use your imagination to think of other ways to access the different stages that the developers may not have anticipated.
4. If any data is submitted more than once, try submitting a different value at different stages, and see whether the login is still successful. It may be that some of the submissions are superfluous and are not actually processed by the application. It might be that the data is validated at one stage and then trusted subsequently. In this instance, try to provide the credentials of one user at one stage, and then switch at the next to actually authenticate as a different user. It might be that the same piece of data is validated at more than one stage, but against different checks. In this instance, try to provide (for example) the username and password of one user at the first stage, and the username and PIN of a different user at the second stage.
5. Pay close attention to any data being transmitted via the client that was not directly entered by the user. The application may use this data to store information about the state of the login progress, and the application may trust it when it is submitted back to the server. For example, if the request for stage three includes the parameter `stage2complete=true`, it may be possible to advance straight to stage three by setting this value. Try to modify the values being submitted, and determine whether this enables you to advance or skip stages.

TRY IT!

```
http://mdsec.net/auth/195/
http://mdsec.net/auth/199/
http://mdsec.net/auth/203/
http://mdsec.net/auth/206/
http://mdsec.net/auth/211/
```

Some login mechanisms employ a randomly varying question at one of the stages of the login process. For example, after submitting a username and password, users might be asked one of various "secret" questions (regarding their mother's maiden name, place of birth, name of first school) or to submit two random letters from a secret phrase. The rationale for this behavior is that even if an attacker captures everything that a user enters on a single occasion, this will not enable him to log in as that user on a different occasion, because different questions will be asked.

In some implementations, this functionality is broken and does not achieve its objectives:

- The application may present a randomly chosen question and store the details within a hidden HTML form field or cookie, rather than on the server. The user subsequently submits both the answer and the question itself. This effectively allows an attacker to choose which question to answer, enabling the attacker to repeat a login after capturing a user's input on a single occasion.
- The application may present a randomly chosen question on each login attempt but not remember which question a given user was asked if he or she fails to submit an answer. If the same user initiates a fresh login attempt a moment later, a different random question is generated. This effectively allows an attacker to cycle through questions until he receives one to which he knows the answer, enabling him to repeat a login having captured a user's input on a single occasion.

NOTE **The second of these conditions is really quite subtle, and as a result, many real-world applications are vulnerable. An application that challenges a user for two random letters of a memorable word may appear at first glance to be functioning properly and providing enhanced security. However, if the letters are randomly chosen each time the previous authentication stage is passed, an attacker who has captured a user's login on a single occasion can simply reauthenticate up to this point until the two letters that he knows are requested, without the risk of account lockout.**

HACK STEPS

1. **If one of the login stages uses a randomly varying question, verify whether the details of the question are being submitted together with the answer. If so, change the question, submit the correct answer associated with that question, and verify whether the login is still successful.**
2. **If the application does not enable an attacker to submit an arbitrary question and answer, perform a partial login several times with a single account, proceeding each time as far as the varying question. If the question changes on each occasion, an attacker can still effectively choose which question to answer.**

TRY IT!

```
http://mdsec.net/auth/178/
http://mdsec.net/auth/182/
```

NOTE **In some applications where one component of the login varies randomly, the application collects all of a user's credentials at a single stage. For example, the main login page may present a form containing fields for username, password, and one of various secret questions. Each time the login page is loaded, the secret question changes. In this situation, the randomness of the secret question does nothing to prevent an attacker from replaying a valid login request having captured a user's input on one occasion. The login process cannot be modified to do so in its present form, because an attacker can simply reload the page until he receives the varying question to which he knows the answer. In a variation on this scenario, the application may set a persistent cookie to "ensure" that the same varying question is presented to any given user until that person answers it correctly. Of course, this measure can be circumvented easily by modifying or deleting the cookie.**

Insecure Storage of Credentials

If an application stores login credentials insecurely, the security of the login mechanism is undermined, even though there may be no inherent flaw in the authentication process itself.

It is common to encounter web applications in which user credentials are stored insecurely within the database. This may involve passwords being stored in cleartext. But if passwords are being hashed using a standard algorithm such as MD5 or SHA-1, this still allows an attacker to simply look up observed hashes against a precomputed database of hash values. Because the database account used by the application must have full read/write access to those credentials, many other kinds of vulnerabilities within the application may be exploitable to enable you to access these credentials, such as command or SQL injection flaws (see Chapter 9) and access control weaknesses (see Chapter 8).

TIP **Some online databases of common hashing functions are available here:**

```
http://passcracking.com/index.php
http://authsecu.com/decrypter-dechiffrer-cracker-hash-md5/
script-hash-md5.php
```

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

made and to help you spot the subtle logic flaw that can be exploited to walk right through the door.

The most important lesson when attacking authentication functionality is to look everywhere. In addition to the main login form, there may be functions to register new accounts, change passwords, remember passwords, recover forgotten passwords, and impersonate other users. Each of these presents a rich target of potential defects, and problems that have been consciously eliminated within one function often reemerge within others. Invest the time to scrutinize and probe every inch of attack surface you can find, and your rewards may be great.

Questions

Answers can be found at `http://mdsec.net/wahh`.

1. While testing a web application, you log in using your credentials of `joe` and `pass`. During the login process, you see a request for the following URL appear in your intercepting proxy:

   ```
   http://www.wahh-app.com/app?action=login&uname=joe&password=pass
   ```

   What three vulnerabilities can you diagnose without probing any further?

2. How can self-registration functions introduce username enumeration vulnerabilities? How can these vulnerabilities be prevented?

3. A login mechanism involves the following steps:

   (a) The application requests the user's username and passcode.

   (b) The application requests two randomly chosen letters from the user's memorable word.

   Why is the required information requested in two separate steps? What defect would the mechanism contain if this were not the case?

4. A multistage login mechanism first requests the user's username and then various other items across successive stages. If any supplied item is invalid, the user is immediately returned to the first stage.

   What is wrong with this mechanism, and how can the vulnerability be corrected?

5. An application incorporates an antiphishing mechanism into its login functionality. During registration, each user selects a specific image from a large bank of memorable images that the application presents to her. The login function involves the following steps:

   (a) The user enters her username and date of birth.

(b) If these details are correct, the application shows the user her chosen image; otherwise, a random image is displayed.

(c) The user verifies whether the correct image is displayed. If it is, she enters her password.

The idea behind this antiphishing mechanism is that it enables the user to confirm that she is dealing with the authentic application, not a clone, because only the real application knows the correct image to display to the user.

What vulnerability does this antiphishing mechanism introduce into the login function? Is the mechanism effective at preventing phishing?