CHAPTER 11

Attacking Application Logic

All web applications employ logic to deliver their functionality. Writing code in a programming language involves at its root nothing more than breaking a complex process into simple and discrete logical steps. Translating a piece of functionality that is meaningful to human beings into a sequence of small operations that can be executed by a computer involves a great deal of skill and discretion. Doing so in an elegant and secure fashion is harder still. When large numbers of different designers and programmers work in parallel on the same application, there is ample opportunity for mistakes to occur.

In all but the simplest of web applications, a vast amount of logic is performed at every stage. This logic presents an intricate attack surface that is always present but often overlooked. Many code reviews and penetration tests focus exclusively on common "headline" vulnerabilities such as SQL injection and cross-site scripting, because these have an easily recognizable signature and well-researched exploitation vector. By contrast, flaws in an application's logic are harder to characterize: each instance may appear to be a unique one-off occurrence, and they usually are not identified by any automated vulnerability scanners. As a result, they generally are not as well appreciated or understood, and therefore they are of great interest to an attacker.

This chapter describes the kinds of logic flaws that often exist in web applications and the practical steps you can take to probe and attack an application's logic. We will present a series of real-world examples, each of which manifests a different kind of logical defect. Together, they illustrate the variety of assumptions

that designers and developers make that can lead directly to faulty logic and expose an application to security vulnerabilities.

The Nature of Logic Flaws

Logic flaws in web applications are extremely varied. They range from simple bugs manifested in a handful of lines of code, to complex vulnerabilities arising from the interoperation of several core components of the application. In some instances, they may be obvious and easy to detect; in other cases, they may be exceptionally subtle and liable to elude even the most rigorous code review or penetration test.

Unlike other coding flaws such as SQL injection or cross-site scripting, no common "signature" is associated with logic flaws. The defining characteristic, of course, is that the logic implemented within the application is defective in some way. In many cases, the defect can be represented in terms of a specific assumption that the designer or developer made, either explicitly or implicitly, that turns out to be flawed. In general terms, a programmer may have reasoned something like "If A happens, then B must be the case, so I will do C." The programmer did not ask the entirely different question "But what if X occurs?" and therefore failed to consider a scenario that violates the assumption. Depending on the circumstances, this flawed assumption may open a significant security vulnerability.

As awareness of common web application vulnerabilities has increased in recent years, the incidence and severity of some categories of vulnerabilities have declined noticeably. However, because of the nature of logic flaws, it is unlikely that they will ever be eliminated via standards for secure development, use of code-auditing tools, or normal penetration testing. The diverse nature of logic flaws, and the fact that detecting and preventing them often requires a good measure of lateral thinking, suggests that they will be prevalent for a good while to come. Any serious attacker, therefore, needs to pay serious attention to the logic employed in the application being targeted to try to figure out the assumptions that designers and developers probably made. Then he should think imaginatively about how those assumptions may be violated.

Real-World Logic Flaws

The best way to learn about logic flaws is not by theorizing, but by becoming acquainted with some actual examples. Although individual instances of logic flaws differ hugely, they share many common themes, and they demonstrate the kinds of mistakes that human developers will always be prone to making.

Hence, insights gathered from studying a sample of logic flaws should help you uncover new flaws in entirely different situations.

Example 1: Asking the Oracle

The authors have found instances of the "encryption oracle" flaw within many different types of applications. They have used it in numerous attacks, from decrypting domain credentials in printing software to breaking cloud computing. The following is a classic example of the flaw found in a software sales site.

The Functionality

The application implemented a "remember me" function whereby a user could avoid logging in to the application on each visit by allowing the application to set a permanent cookie within the browser. This cookie was protected from tampering or disclosure by an encryption algorithm that was run over a string composed of the name, user ID, and volatile data to ensure that the resultant value was unique and could not be predicted. To ensure that it could not be replayed by an attacker who gained access to it, data specific to the machine was also collected, including the IP address.

This cookie was justifiably considered a robust solution for protecting a potentially vulnerable piece of required business functionality.

As well as a "remember me" function, the application had functionality to store the user's screen name within a cookie named `ScreenName`. That way, the user could receive a personalized greeting in the corner of the site whenever she next visited the site. Deciding that this name was also a piece of security information, it was deemed that this should also be encrypted.

The Assumption

The developers decided that because the `ScreenName` cookie was of considerably less value to an attacker than the `RememberMe` cookie, they may as well use the same encryption algorithm to protect it. What they did not consider was that a user can specify his screen name and view it onscreen. This inadvertently gave users access to the encryption function (and encryption key) used to protect the persistent authentication token `RememberMe`.

The Attack

In a simple attack, a user supplied the encrypted value of his or her `RememberMe` cookie in place of the encrypted `ScreenName` cookie. When displaying the screen name back to the user, the application would decrypt the value, check that

decryption had worked, and then print the result on-screen. This resulted in the following message:

```
Welcome, marcus|734|192.168.4.282750184
```

Although this was interesting, it was not necessarily a high-risk issue. It simply meant that given an encrypted `RememberMe` cookie, an attacker could list the contents, including a username, user ID, and IP address. Because no password was stored in the cookie, there was no immediate way to act on the information obtained.

The real issue arose from the fact that users could specify their screen names. As a result, a user could choose this screen name, for example:

```
admin|1|192.168.4.282750184
```

When the user logged out and logged back in, the application encrypted this value and stored it in the user's browser as the encrypted `ScreenName` cookie. If an attacker submitted this encrypted token as the value of the `RememberMe` cookie, the application decrypted it, read the user ID, and logged in the attacker as the administrator! Even though the encryption was Triple DES, using a strong key and protected against replay attacks, the application could be harnessed as an "encryption oracle" to decrypt and encrypt arbitrary values.

HACK STEPS

Manifestations of this type of vulnerability can be found in diverse locations. Examples include account recovery tokens, token-based access to authenticated resources, and any other value being sent to the client side that needs to be either tamper-proof or unreadable to the user.

1. **Look for locations where encryption (not hashing) is used in the application. Determine any locations where the application encrypts or decrypts values supplied by a user, and attempt to substitute any other encrypted values encountered within the application. Try to cause an error within the application that reveals the decrypted value or where the decrypted value is purposely displayed on-screen.**
2. **Look for an "oracle reveal" vulnerability by determining where an encrypted value can be supplied that results in the corresponding decrypted value's being displayed in the application's response. Determine whether this leads to the disclosure of sensitive information, such as a password or credit card.**
3. **Look for an "oracle encrypt" vulnerability by determining where supplying a cleartext value causes the application to return a corresponding encrypted value. Determine where this can be abused by specifying arbitrary values, or malicious payloads that the application will process.**

Example 2: Fooling a Password Change Function

The authors have encountered this logic flaw in a web application implemented by a financial services company and also in the AOL AIM Enterprise Gateway application.

The Functionality

The application implemented a password change function for end users. It required the user to fill out fields for username, existing password, new password, and confirm new password.

There was also a password change function for use by administrators. This allowed them to change the password of any user without supplying the existing password. The two functions were implemented within the same server-side script.

The Assumption

The client-side interface presented to users and administrators differed in one respect: the administrator's interface did not contain a field for the existing password. When the server-side application processed a password change request, it used the presence or absence of the existing password parameter to indicate whether the request was from an administrator or an ordinary user. In other words, it assumed that ordinary users would always supply an existing password parameter.

The code responsible looked something like this:

```
String existingPassword = request.getParameter("existingPassword");
if (null == existingPassword)
{
    trace("Old password not supplied, must be an administrator");
    return true;
}
else
{
    trace("Verifying user's old password");
    ...
```

The Attack

When the assumption is explicitly stated in this way, the logic flaw becomes obvious. Of course, an ordinary user could issue a request that did not contain an existing password parameter, because users controlled every aspect of the requests they issued.

This logic flaw was devastating for the application. It enabled an attacker to reset the password of any other user and take full control of that person's account.

HACK STEPS

1. **When probing key functionality for logic flaws, try removing in turn each parameter submitted in requests, including cookies, query string fields, and items of `POST` data.**
2. **Be sure to delete the actual name of the parameter as well as its value. Do not just submit an empty string, because typically the server handles this differently.**
3. **Attack only one parameter at a time to ensure that all relevant code paths within the application are reached.**
4. **If the request you are manipulating is part of a multistage process, follow the process through to completion, because some later logic may process data that was supplied in earlier steps and stored within the session.**

Example 3: Proceeding to Checkout

The authors encountered this logic flaw in the web application employed by an online retailer.

The Functionality

The process of placing an order involved the following stages:

1. Browse the product catalog, and add items to the shopping basket.
2. Return to the shopping basket, and finalize the order.
3. Enter payment information.
4. Enter delivery information.

The Assumption

The developers assumed that users would always access the stages in the intended sequence, because this was the order in which the stages are delivered to the user by the navigational links and forms presented to the user's browser. Hence, any user who completed the ordering process must have submitted satisfactory payment details along the way.

The Attack

The developers' assumption was flawed for fairly obvious reasons. Users controlled every request they made to the application and therefore could access

any stage of the ordering process in any sequence. By proceeding directly from stage 2 to stage 4, an attacker could generate an order that was finalized for delivery but that had not actually been paid for.

HACK STEPS

The technique for finding and exploiting flaws of this kind is known as *forced browsing*. It involves circumventing any controls imposed by in-browser navigation on the sequence in which application functions may be accessed:

1. **When a multistage process involves a defined sequence of requests, attempt to submit these requests out of the expected sequence. Try skipping certain stages, accessing a single stage more than once, and accessing earlier stages after later ones.**
2. **The sequence of stages may be accessed via a series of `GET` or `POST` requests for distinct URLs, or they may involve submitting different sets of parameters to the same URL. The stage being requested may be specified by submitting a function name or index within a request parameter. Be sure to understand fully the mechanisms that the application is employing to deliver access to distinct stages.**
3. **From the context of the functionality that is implemented, try to understand what assumptions the developers may have made and where the key attack surface lies. Try to identify ways of violating those assumptions to cause undesirable behavior within the application.**
4. **When multistage functions are accessed out of sequence, it is common to encounter a variety of anomalous conditions within the application, such as variables with null or uninitialized values, a partially defined or inconsistent state, and other unpredictable behavior. In this situation, the application may return an interesting error message and debug output, which you can use to better understand its internal workings and thereby fine-tune the current or a different attack (see Chapter 15). Sometimes, the application may get into a state entirely unanticipated by developers, which may lead to serious security flaws.**

NOTE **Many types of access control vulnerability are similar in nature to this logic flaw. When a privileged function involves multiple stages that normally are accessed in a defined sequence, the application may assume that users will always proceed through the functionality in this sequence. The application may enforce strict access control on the initial stages of the process and assume that any user who reaches the later stages therefore must be authorized. If a low-privileged user proceeds directly to a later stage, she may be able to access it without any restrictions. See Chapter 8 for more details on finding and exploiting vulnerabilities of this kind.**

Example 4: Rolling Your Own Insurance

The authors encountered this logic flaw in a web application deployed by a financial services company.

The Functionality

The application enabled users to obtain quotes for insurance and, if desired, complete and submit an insurance application online. The process was spread across a dozen stages:

- At the first stage, the applicant submitted some basic information and specified either a preferred monthly premium or the value he wanted insurance for. The application offered a quote, computing whichever value the applicant did not specify.
- Across several stages, the applicant supplied various other personal details, including health, occupation, and pastimes.
- Finally, the application was transmitted to an underwriter working for the insurance company. Using the same web application, the underwriter reviewed the details and decided whether to accept the application as is or modify the initial quote to reflect any additional risks.

Through each of the stages described, the application employed a shared component to process each parameter of user data submitted to it. This component parsed all the data in each `POST` request into name/value pairs and updated its state information with each item of data received.

The Assumption

The component that processed user-supplied data assumed that each request would contain only the parameters that had been requested from the user in the relevant HTML form. Developers did not consider what would happen if a user submitted parameters he was not asked to supply.

The Attack

Of course, the assumption was flawed, because users could submit arbitrary parameter names and values with every request. As a result, the application's core functionality was broken in various ways:

- An attacker could exploit the shared component to bypass all server-side input validation. At each stage of the quotation process, the application performed strict validation of the data expected at that stage and rejected any data that failed this validation. But the shared component updated

the application's state with every parameter supplied by the user. Hence, if an attacker submitted data out of sequence by supplying a name/value pair that the application expected at an earlier stage, that data would be accepted and processed, with no validation having been performed. As it happened, this possibility paved the way for a stored cross-site scripting attack targeting the underwriter, which allowed a malicious user to access the personal information of other applicants (see Chapter 12).

- An attacker could buy insurance at an arbitrary price. At the first stage of the quotation process, the applicant specified either her preferred monthly premium or the value she wanted to insure, and the application computed the other item accordingly. However, if a user supplied new values for either or both of these items at a later stage, the application's state was updated with these values. By submitting these parameters out of sequence, an attacker could obtain a quote for insurance at an arbitrary value and arbitrary monthly premium.
- There were no access controls regarding which parameters a given type of user could supply. When an underwriter reviewed a completed application, he updated various items of data, including the acceptance decision. This data was processed by the shared component in the same way as data supplied by an ordinary user. If an attacker knew or guessed the parameter names used when the underwriter reviewed an application, the attacker could simply submit these, thereby accepting his own application without any actual underwriting.

HACK STEPS

The flaws in this application were fundamental to its security, but none of them would have been identified by an attacker who simply intercepted browser requests and modified the parameter values being submitted.

1. **Whenever an application implements a key action across multiple stages, you should take parameters that are submitted at one stage of the process and try submitting these to a different stage. If the relevant items of data are updated within the application's state, you should explore the ramifications of this behavior to determine whether you can leverage it to carry out any malicious action, as in the preceding three examples.**
2. **If the application implements functionality whereby different categories of user can update or perform other actions on a common collection of data, you should walk through the process using each type of user and observe the parameters submitted. Where different parameters are ordinarily submitted by the different users, take each parameter submitted by one user and try to submit it as the other user. If the parameter is accepted and processed as that user, explore the implications of this behavior as previously described.**

Example 5: Breaking the Bank

The authors encountered this logic flaw in the web application deployed by a major financial services company.

The Functionality

The application enabled existing customers who did not already use the online application to register to do so. New users were required to supply some basic personal information to provide a degree of assurance of their identity. This information included name, address, and date of birth, but it did not include anything secret such as an existing password or PIN.

When this information had been entered correctly, the application forwarded the registration request to back-end systems for processing. An information pack was mailed to the user's registered home address. This pack included instructions for activating her online access via a telephone call to the company's call center and also a one-time password to use when first logging in to the application.

The Assumption

The application's designers believed that this mechanism provided a robust defense against unauthorized access to the application. The mechanism implemented three layers of protection:

- A modest amount of personal data was required up front to deter a malicious attacker or mischievous user from attempting to initiate the registration process on other users' behalf.
- The process involved transmitting a key secret out-of-band to the customer's registered home address. An attacker would need to have access to the victim's personal mail.
- The customer was required to telephone the call center and authenticate himself there in the usual way, based on personal information and selected digits from a PIN.

This design was indeed robust. The logic flaw lay in the implementation of the mechanism.

The developers implementing the registration mechanism needed a way to store the personal data submitted by the user and correlate this with a unique customer identity within the company's database. Keen to reuse existing code, they came across the following class, which appeared to serve their purposes:

```
class CCustomer
{
    String firstName;
    String lastName;
```

```
CDoB dob;
CAddress homeAddress;
long custNumber;
...
```

After the user's information was captured, this object was instantiated, populated with the supplied information, and stored in the user's session. The application then verified the user's details and, if they were valid, retrieved that user's unique customer number, which was used in all the company's systems. This number was added to the object, together with some other useful information about the user. The object was then transmitted to the relevant back-end system for the registration request to be processed.

The developers assumed that using this code component was harmless and would not lead to a security problem. However, the assumption was flawed, with serious consequences.

The Attack

The same code component that was incorporated into the registration functionality was also used elsewhere within the application, including within the core functionality. This gave authenticated users access to account details, statements, funds transfers, and other information. When a registered user successfully authenticated herself to the application, this same object was instantiated and saved in her session to store key information about her identity. The majority of the functionality within the application referenced the information within this object to carry out its actions. For example, the account details presented to the user on her main page were generated on the basis of the unique customer number contained within this object.

The way in which the code component was already being employed within the application meant that the developers' assumption was flawed, and the manner in which they reused it did indeed open a significant vulnerability.

Although the vulnerability was serious, it was in fact relatively subtle to detect and exploit. Access to the main application functionality was protected by access controls at several layers, and a user needed to have a fully authenticated session to pass these controls. To exploit the logic flaw, therefore, an attacker needed to follow these steps:

- Log in to the application using his own valid account credentials.
- Using the resulting authenticated session, access the registration functionality and submit a different customer's personal information. This caused the application to overwrite the original `CCustomer` object in the attacker's session with a new object relating to the targeted customer.
- Return to the main application functionality and access the other customer's account.

A vulnerability of this kind is not easy to detect when probing the application from a black-box perspective. However, it is also hard to identify when reviewing or writing the actual source code. Without a clear understanding of the application as a whole and how different components are used in different areas, the flawed assumption made by developers may not be evident. Of course, clearly commented source code and design documentation would reduce the likelihood of such a defect's being introduced or remaining undetected.

HACK STEPS

1. **In a complex application involving either horizontal or vertical privilege segregation, try to locate any instances where an individual user can accumulate an amount of state within his session that relates in some way to his identity.**
2. **Try to step through one area of functionality, and then switch to an unrelated area, to determine whether any accumulated state information has an effect on the application's behavior.**

Example 6: Beating a Business Limit

The authors encountered this logic flaw in a web-based enterprise resource planning application used within a manufacturing company.

The Functionality

Finance personnel could perform funds transfers between various bank accounts owned by the company and its key customers and suppliers. As a precaution against fraud, the application prevented most users from processing transfers with a value greater than $10,000. Any transfer larger than this required a senior manager's approval.

The Assumption

The code responsible for implementing this check within the application was simple:

```
bool CAuthCheck::RequiresApproval(int amount)
{
    if (amount <= m_apprThreshold)
        return false;
    else return true;
}
```

The developers assumed that this transparent check was bulletproof. No transaction for greater than the configured threshold could ever escape the requirement for secondary approval.

The Attack

The developers' assumption was flawed because they overlooked the possibility that a user would attempt to process a transfer for a negative amount. Any negative number would clear the approval test, because it is less than the threshold. However, the banking module of the application accepted negative transfers and simply processed them as positive transfers in the opposite direction. Hence, any user who wanted to transfer $20,000 from account A to account B could simply initiate a transfer of –$20,000 from account B to account A, which had the same effect and required no approval. The antifraud defenses built into the application could be bypassed easily!

NOTE **Many kinds of web applications employ numeric limits within their business logic:**

- **A retailing application may prevent a user from ordering more than the number of units available in stock.**
- **A banking application may prevent a user from making bill payments that exceed her current account balance.**
- **An insurance application may adjust its quotes based on age thresholds.**

Finding a way to beat such limits often does not represent a security compromise of the application itself. However, it may have serious business consequences and represent a breach of the controls that the owner is relying on the application to enforce.

The most obvious vulnerabilities of this kind often are detected during the user-acceptance testing that normally occurs before an application is launched. However, more subtle manifestations of the problem may remain, particularly when hidden parameters are being manipulated.

HACK STEPS

The first step in attempting to beat a business limit is to understand what characters are accepted within the relevant input that you control.

1. **Try entering negative values, and see if the application accepts them and processes them in the way you would expect.**
2. **You may need to perform several steps to engineer a change in the application's state that can be exploited for a useful purpose. For example, several transfers between accounts may be required until a suitable balance has been accrued that can actually be extracted.**

Example 7: Cheating on Bulk Discounts

The authors encountered this logic flaw in the retail application of a software vendor.

The Functionality

The application allowed users to order software products and qualify for bulk discounts if a suitable bundle of items was purchased. For example, users who purchased an antivirus solution, personal firewall, and antispam software were entitled to a 25% discount on the individual prices.

The Assumption

When a user added an item of software to his shopping basket, the application used various rules to determine whether the bundle of purchases he had chosen entitled him to a discount. If so, the prices of the relevant items within the shopping basket were adjusted in line with the discount. The developers assumed that the user would go on to purchase the chosen bundle and therefore would be entitled to the discount.

The Attack

The developers' assumption is rather obviously flawed because it ignores the fact that users may remove items from their shopping baskets after they have been added. A crafty user could add to his basket large quantities of every single product on sale from the vendor to attract the maximum possible bulk discounts. After the discounts were applied to items in his shopping basket, he could remove items he did not want and still receive the discounts applied to the remaining products.

HACK STEPS

1. **In any situation where prices or other sensitive values are adjusted based on criteria that are determined by user-controllable data or actions, first understand the algorithms that the application uses and the point within its logic where adjustments are made. Identify whether these adjustments are made on a one-time basis or whether they are revised in response to further actions performed by the user.**
2. **Think imaginatively. Try to find a way of manipulating the application's behavior to cause it to get into a state where the adjustments it has applied do not correspond to the original criteria intended by its designers. In the most obvious case, as just described, this may simply involve removing items from a shopping cart after a discount has been applied!**

Example 8: Escaping from Escaping

The authors encountered this logic flaw in various web applications, including the web administration interface used by a network intrusion detection product.

The Functionality

The application's designers had decided to implement some functionality that involved passing user-controllable input as an argument to an operating system command. The application's developers understood the inherent risks involved in this kind of operation (see Chapter 9) and decided to defend against these risks by sanitizing any potentially malicious characters within the user input. Any instances of the following would be escaped using the backslash character:

`; | & < > '` space and newline

Escaping data in this way causes the shell command interpreter to treat the relevant characters as part of the argument being passed to the invoked command, rather than as shell metacharacters. Such metacharacters could be used to inject additional commands or arguments, redirect output, and so on.

The Assumption

The developers were certain that they had devised a robust defense against command injection attacks. They had brainstormed every possible character that might assist an attacker and had ensured that they were all properly escaped and therefore made safe.

The Attack

The developers forgot to escape the escape character itself.

The backslash character usually is not of direct use to an attacker when exploiting a simple command injection flaw. Therefore, the developers did not identify it as potentially malicious. However, by failing to escape it, they provided a means for the attacker to defeat their sanitizing mechanism.

Suppose an attacker supplies the following input to the vulnerable function:

```
foo\;ls
```

The application applies the relevant escaping, as described previously, so the attacker's input becomes:

```
foo\\;ls
```

When this data is passed as an argument to the operating system command, the shell interpreter treats the first backslash as the escape character. Therefore, it treats the second backslash as a literal backslash—not as an escape character, but as part of the argument itself. It then encounters a semicolon that is apparently not escaped. It treats this as a command separator and therefore goes on to execute the injected command supplied by the attacker.

HACK STEPS

Whenever you probe an application for command injection and other flaws, having attempted to insert the relevant metacharacters into the data you control, always try placing a backslash immediately before each such character to test for the logic flaw just described.

NOTE **This same flaw can be found in some defenses against cross-site scripting attacks (see Chapter 12). When user-supplied input is copied directly into the value of a string variable in a piece of JavaScript, this value is encapsulated within quotation marks. To defend themselves against cross-site scripting, many applications use backslashes to escape any quotation marks that appear within the user's input. However, if the backslash character itself is not escaped, an attacker can submit `\'` to break out of the string and therefore take control of the script. This exact bug was found in early versions of the Ruby On Rails framework in the `escape_javascript` function.**

Example 9: Invalidating Input Validation

The authors encountered this logic flaw in a web application used in an e-commerce site. Variants can be found in many other applications.

The Functionality

The application contained a suite of input validation routines to protect against various types of attacks. Two of these defense mechanisms were a SQL injection filter and a length limiter.

It is common for applications to try to defend themselves against SQL injection by escaping any single quotation marks that appear within string-based user input (and rejecting any that appear within numeric input). As described in Chapter 9, two single quotation marks together are an escape sequence that represents one literal single quote, which the database interprets as data within a quoted string rather than the closing string terminator. Many developers reason, therefore, that by doubling any single quotation marks within user-supplied input, they will prevent any SQL injection attacks from occurring.

The length limiter was applied to all input, ensuring that no variable supplied by a user was longer than 128 characters. It achieved this by truncating any variables to 128 characters.

The Assumption

It was assumed that both the SQL injection filter and length truncation were desirable defenses from a security standpoint, so both should be applied.

The Attack

The SQL injection defense works by doubling any quotation marks appearing within user input, so that within each pair of quotes, the first quote acts as an escape character to the second. However, the developers did not consider what would happen to the sanitized input if it was then handed to the truncation function.

Recall the SQL injection example in a login function in Chapter 9. Suppose that the application doubles any single quotation marks contained in user input and also then imposes a length limit on the data, truncating it to 128 characters. Supplying this username:

```
admin'--
```

now results in the following query, which fails to bypass the login:

```
SELECT * FROM users WHERE username = 'admin''--' and password = ''
```

However, if you submit a following username (containing 127 a's followed by a single quotation mark):

```
aaaaaaaa[...]aaaaaaaaaaa'
```

the application first doubles up the single quotation mark and then truncates the string to 128 characters, returning your input to its original value. This results in a database error, because you have injected an additional single quotation mark into the query without fixing the surrounding syntax. If you now also supply the password:

```
or 1=1--
```

the application performs the following query, which succeeds in bypassing the login:

```
SELECT * FROM users WHERE username = 'aaaaaaaa[...]aaaaaaaaaaa'' and
 password = 'or 1=1--'
```

The doubled quotation mark at the end of the string of a's is interpreted as an escaped quotation mark and, therefore, as part of the query data. This string effectively continues as far as the next single quotation mark, which in the original query marked the start of the user-supplied password value. Thus, the actual username that the database understands is the literal string data shown here:

```
aaaaaaaa[...]aaaaaaaaaaa'and password =
```

Hence, whatever comes next is interpreted as part of the query itself and can be crafted to interfere with the query logic.

TIP **You can test for this type of vulnerability without knowing exactly what length limit is being imposed by submitting in turn two long strings of the following form:**

```
'''''''''''''''''''''''''''''''''''''''''''''''''' and so on

a''''''''''''''''''''''''''''''''''''''''''''''''' and so on
```

and determining whether an error occurs. Any truncation of escaped input will occur after either an even or odd number of characters. Whichever possibility is the case, one of the preceding strings will result in an odd number of single quotation marks being inserted into the query, resulting in invalid syntax.

HACK STEPS

Make a note of any instances in which the application modifies user input, in particular by truncating it, stripping out data, encoding, or decoding. For any observed instances, determine whether a malicious string can be contrived:

1. **If data is stripped once (nonrecursively), determine whether you can submit a string that compensates for this. For example, if the application filters SQL keywords such as `SELECT`, submit `SELSELECTECT` and see if the resulting filtering removes the inner `SELECT` substring, leaving the word `SELECT`.**
2. **If data validation takes place in a set order and one or more validation processes modifies the data, determine whether this can be used to beat one of the prior validation steps. For example, if the application performs URL decoding and then strips malicious data such as the `<script>` tag, it may be possible to overcome this with strings such as:**

```
%<script>3cscript%<script>3ealert(1)%<script>3c/
script%<script>3e
```

NOTE **Cross-site scripting filters frequently inadvisably strip all data that occurs between HTML tag pairs, such as `<tag1>aaaaa</tag1>`. These are often vulnerable to this type of attack.**

Example 10: Abusing a Search Function

The authors encountered this logic flaw in an application providing subscription-based access to financial news and information. The same vulnerability was later found in two completely unrelated applications, illustrating the subtle and pervasive nature of many logic flaws.

The Functionality

The application provided access to a huge archive of historical and current information, including company reports and accounts, press releases, market analyses, and the like. Most of this information was accessible only to paying subscribers.

The application provided a powerful and fine-grained search function that all users could access. When an anonymous user performed a query, the search function returned links to all documents that matched the query. However, the user was required to subscribe to retrieve any of the actual protected documents his query returned. The application's owners regarded this behavior as a useful marketing tactic.

The Assumption

The application's designer assumed that users could not use the search function to extract any useful information without paying for it. The document titles listed in the search results were typically cryptic, such as "Annual Results 2010," "Press Release 08-03-2011," and so on.

The Attack

Because the search function indicated how many documents matched a given query, a wily user could issue a large number of queries and use inference to extract information from the search function that normally would need to be paid for. For example, the following queries could be used to zero in on the contents of an individual protected document:

```
wahh consulting
>> 276 matches
wahh consulting "Press Release 08-03-2011" merger
>> 0 matches
wahh consulting "Press Release 08-03-2011" share issue
>> 0 matches
wahh consulting "Press Release 08-03-2011" dividend
>> 0 matches
wahh consulting "Press Release 08-03-2011" takeover
>> 1 match
wahh consulting "Press Release 08-03-2011" takeover haxors inc
>> 0 matches
wahh consulting "Press Release 08-03-2011" takeover uberleet ltd
>> 0 matches
wahh consulting "Press Release 08-03-2011" takeover script kiddy corp
>> 0 matches
wahh consulting "Press Release 08-03-2011" takeover ngs
>> 1 match
```

```
wahh consulting "Press Release 08-03-2011" takeover ngs announced
>> 0 matches
wahh consulting "Press Release 08-03-2011" takeover ngs cancelled
>> 0 matches
wahh consulting "Press Release 08-03-2011" takeover ngs completed
>> 1 match
```

Although the user cannot view the document itself, with sufficient imagination and use of scripted requests, he may be able to build a fairly accurate understanding of its contents.

TIP In certain situations, being able to leach information via a search function in this way may be critical to the security of the application itself, effectively disclosing details of administrative functions, passwords, and technologies in use.

TIP This technique has proven to be an effective attack against internal document management software. The authors have used this technique to brute-force a key password from a configuration file that was stored in a wiki. Because the wiki returned a hit if the search string appeared anywhere in the page (instead of matching on whole words), it was possible to brute-force the password letter by letter, searching for the following:

```
Password=A
Password=B
Password=BA
...
```

Example 11: Snarfing Debug Messages

The authors encountered this logic flaw in a web application used by a financial services company.

The Functionality

The application was only recently deployed. Like much new software, it still contained a number of functionality-related bugs. Intermittently, various operations would fail in an unpredictable way, and users would receive an error message.

To facilitate the investigation of errors, developers decided to include detailed, verbose information in these messages, including the following details:

- The user's identity
- The token for the current session
- The URL being accessed
- All the parameters supplied with the request that generated the error

Generating these messages had proven useful when help desk personnel attempted to investigate and recover from system failures. They also were helping iron out the remaining functionality bugs.

The Assumption

Despite the usual warnings from security advisers that verbose debug messages of this kind could potentially be misused by an attacker, the developers reasoned that they were not opening any security vulnerability. The user could readily obtain all the information contained in the debugging message by inspecting the requests and responses processed by her browser. The messages did not include any details about the actual failure, such as stack traces, so conceivably they were not helpful in formulating an attack against the application.

The Attack

Despite their reasoning about the contents of the debug messages, the developers' assumption was flawed because of mistakes they made in implementing the creation of debugging messages.

When an error occurred, a component of the application gathered all the required information and stored it. The user was issued an HTTP redirect to a URL that displayed this stored information. The problem was that the application's storage of debug information, and user access to the error message, was not session-based. Rather, the debugging information was stored in a static container, and the error message URL always displayed the information that was last placed in this container. Developers had assumed that users following the redirect would therefore see only the debug information relating to their error.

In fact, in this situation, ordinary users would occasionally be presented with the debugging information relating to a different user's error, because the two errors had occurred almost simultaneously. But aside from questions about thread safety (see the next example), this was not simply a race condition. An attacker who discovered how the error mechanism functioned could simply poll the message URL repeatedly and log the results each time they changed. Over a period of few hours, this log would contain sensitive data about numerous application users:

- A set of usernames that could be used in a password-guessing attack
- A set of session tokens that could be used to hijack sessions
- A set of user-supplied input, which may contain passwords and other sensitive items

The error mechanism, therefore, presented a critical security threat. Because administrative users sometimes received these detailed error messages, an

attacker monitoring error messages would soon obtain sufficient information to compromise the entire application.

HACK STEPS

1. **To detect a flaw of this kind, first catalog all the anomalous events and conditions that can be generated and that involve interesting user-specific information being returned to the browser in an unusual way, such as a debugging error message.**
2. **Using the application as two users in parallel, systematically engineer each condition using one or both users, and determine whether the other user is affected in each case.**

Example 12: Racing Against the Login

This logic flaw has affected several major applications in the recent past.

The Functionality

The application implemented a robust, multistage login process in which users were required to supply several different credentials to gain access.

The Assumption

The authentication mechanism had been subject to numerous design reviews and penetration tests. The owners were confident that no feasible means existed of attacking the mechanism to gain unauthorized access.

The Attack

In fact, the authentication mechanism contained a subtle flaw. Occasionally, when a customer logged in, he gained access to the account of a completely different user, enabling him to view all that user's financial details, and even make payments from the other user's account. The application's behavior initially appeared to be random: the user had not performed any unusual action to gain unauthorized access, and the anomaly did not recur on subsequent logins.

After some investigation, the bank discovered that the error was occurring when two different users logged in to the application at precisely the same moment. It did not occur on every such occasion—only on a subset of them. The root cause was that the application was briefly storing a key identifier about each newly authenticated user within a static (nonsession) variable. After being written, this variable's value was read back an instant later. If a different thread (processing another login) had written to the variable during this instant, the earlier user would land in an authenticated session belonging to the subsequent user.

The vulnerability arose from the same kind of mistake as in the error message example described previously: the application was using static storage to hold information that should have been stored on a per-thread or per-session basis. However, the present example is far more subtle to detect and is more difficult to exploit because it cannot be reliably reproduced.

Flaws of this kind are known as "race conditions" because they involve a vulnerability that arises for a brief period of time under certain specific circumstances. Because the vulnerability exists only for a short time, an attacker "races" to exploit it before the application closes it again. In cases where the attacker is local to the application, it is often possible to engineer the exact circumstances under which the race condition arises and reliably exploit the vulnerability during the available window. Where the attacker is remote to the application, this is normally much harder to achieve.

A remote attacker who understood the nature of the vulnerability could conceivably have devised an attack to exploit it by using a script to log in continuously and check the details of the account accessed. But the tiny window during which the vulnerability could be exploited meant that a huge number of requests would be required.

It was not surprising that the race condition was not discovered during normal penetration testing. The conditions in which it arose came about only when the application gained a large-enough user base for random anomalies to occur, which were reported by customers. However, a close code review of the authentication and session management logic would have identified the problem.

HACK STEPS

Performing remote black-box testing for subtle thread safety issues of this kind is not straightforward. It should be regarded as a specialized undertaking, probably necessary only in the most security-critical of applications.

1. **Target selected items of key functionality, such as login mechanisms, password change functions, and funds transfer processes.**
2. **For each function tested, identify a single request, or a small number of requests, that a given user can use to perform a single action. Also find the simplest means of confirming the result of the action, such as verifying that a given user's login has resulted in access to that person's account information.**
3. **Using several high-spec machines, accessing the application from different network locations, script an attack to perform the same action repeatedly on behalf of several different users. Confirm whether each action has the expected result.**
4. **Be prepared for a large volume of false positives. Depending on the scale of the application's supporting infrastructure, this activity may well amount to a load test of the installation. Anomalies may be experienced for reasons that have nothing to do with security.**

Avoiding Logic Flaws

Just as there is no unique signature by which logic flaws in web applications can be identified, there is also no silver bullet that will protect you. For example, there is no equivalent to the straightforward advice of using a safe alternative to a dangerous API. Nevertheless, a range of good practices can be applied to significantly reduce the risk of logical flaws appearing within your applications:

- Ensure that every aspect of the application's design is clearly documented in sufficient detail for an outsider to understand every assumption the designer made. All such assumptions should be explicitly recorded within the design documentation.
- Mandate that all source code is clearly commented to include the following information throughout:
  - The purpose and intended uses of each code component.
  - The assumptions made by each component about anything that is outside of its direct control.
  - References to all client code that uses the component. Clear documentation to this effect could have prevented the logic flaw within the online registration functionality. (Note that "client" here refers not to the user end of the client/server relationship but to other code for which the component being considered is an immediate dependency.)
- During security-focused reviews of the application design, reflect on every assumption made within the design, and try to imagine circumstances under which each assumption might be violated. Focus on any assumed conditions that could conceivably be within the control of application users.
- During security-focused code reviews, think laterally about two key areas: the ways in which the application will handle unexpected user behavior, and the potential side effects of any dependencies and interoperation between different code components and different application functions.

In relation to the specific examples of logic flaws we have described, a number of individual lessons can be learned:

- Be constantly aware that users control every aspect of every request (see Chapter 1). They may access multistage functions in any sequence. They may submit parameters that the application did not ask for. They may omit certain parameters, not just interfere with the parameters' values.
- Drive all decisions regarding a user's identity and status from her session (see Chapter 8). Do not make any assumptions about the user's privileges on the basis of any other feature of the request, including the fact that it occurs at all.

- When implementing functions that update session data on the basis of input received from the user, or actions performed by the user, carefully consider any impact that the updated data may have on other functionality within the application. Be aware that unexpected side effects may occur in entirely unrelated functionality written by a different programmer or even a different development team.
- If a search function is liable to index sensitive data that some users are not authorized to access, ensure that the function does not provide any means for those users to infer information based on search results. If appropriate, maintain several search indexes based on different levels of user privilege, or perform dynamic searches of information repositories with the privileges of the requesting user.
- Be extremely wary of implementing any functionality that enables any user to delete items from an audit trail. Also, consider the possible impact of a high-privileged user creating another user of the same privilege level in heavily audited applications and dual-authorization models.
- When carrying out checks based on numeric business limits and thresholds, perform strict canonicalization and data validation on all user input before processing it. If negative numbers are not expected, explicitly reject requests that contain them.
- When implementing discounts based on order volumes, ensure that orders are finalized before actually applying the discount.
- When escaping user-supplied data before passing to a potentially vulnerable application component, always be sure to escape the escape character itself, or the entire validation mechanism may be broken.
- Always use appropriate storage to maintain any data that relates to an individual user—either in the session or in the user's profile.

Summary

Attacking an application's logic involves a mixture of systematic probing and lateral thinking. We have described various key checks that you should always carry out to test the application's behavior in response to unexpected input. These include removing parameters from requests, using forced browsing to access functions out of sequence, and submitting parameters to different locations within the application. Often, how an application responds to these actions points toward some defective assumption that you can violate, to malicious effect.

In addition to these basic tests, the most important challenge when probing for logic flaws is to try to get inside the developers' minds. You need to understand what they were trying to achieve, what assumptions they probably made,