

Core Defense Mechanisms

The fundamental security problem with web applications — that all user input is untrusted — gives rise to a number of security mechanisms that applications use to defend themselves against attack. Virtually all applications employ mechanisms that are conceptually similar, although the details of the design and the effectiveness of the implementation vary greatly.

The defense mechanisms employed by web applications **comprise** the following core elements:

- Handling user access to the application's data and functionality to prevent users from gaining unauthorized access
- Handling user input to the application's functions to prevent **malformed** input from causing **undesirable** behavior
- Handling attackers to ensure that the application behaves **appropriately** when being directly targeted, taking suitable defensive and offensive measures to **frustrate** the attacker
- Managing the application itself by enabling administrators to monitor its activities and configure its functionality

Because of their central role in addressing the core security problem, these mechanisms also make up the vast majority of a typical application's attack surface. If knowing your enemy is the first rule of **warfare**, then understanding these mechanisms **thoroughly** is the main **prerequisite** for being able to attack

applications effectively. If you are new to hacking web applications (and even if you are not), you should be sure to take time to understand how these core mechanisms work in each of the applications you encounter, and identify the weak points that leave them vulnerable to attack.

Handling User Access

A central security requirement that virtually any application needs to meet is controlling users' access to its data and functionality. A typical situation has several different categories of user, such as anonymous users, ordinary authenticated users, and administrative users. Furthermore, in many situations different users are permitted to access a different set of data. For example, users of a web mail application should be able to read their own e-mail but not other people's.

Most web applications handle access using a trio of interrelated security mechanisms:

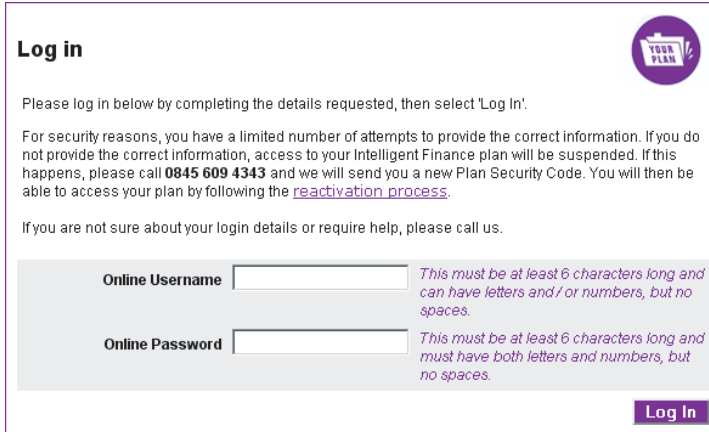
- Authentication
- Session management
- Access control

Each of these mechanisms represents a significant area of an application's attack surface, and each is fundamental to an application's overall security posture. Because of their interdependencies, the overall security provided by the mechanisms is only as strong as the weakest link in the chain. A defect in any single component may enable an attacker to gain unrestricted access to the application's functionality and data.

Authentication

The authentication mechanism is logically the most basic dependency in an application's handling of user access. Authenticating a user involves establishing that the user is in fact who he claims to be. Without this facility, the application would need to treat all users as anonymous — the lowest possible level of trust.

The majority of today's web applications employ the conventional authentication model, in which the user submits a username and password, which the application checks for validity. Figure 2-1 shows a typical login function. In security-critical applications such as those used by online banks, this basic model is usually supplemented by additional credentials and a multistage login process. When security requirements are higher still, other authentication models may be used, based on client certificates, smartcards, or challenge-response tokens. In addition to the core login process, authentication mechanisms often employ a range of other supporting functionality, such as self-registration, account recovery, and a password change facility.



Log in

Please log in below by completing the details requested, then select 'Log In'.

For security reasons, you have a limited number of attempts to provide the correct information. If you do not provide the correct information, access to your Intelligent Finance plan will be suspended. If this happens, please call **0845 609 4343** and we will send you a new Plan Security Code. You will then be able to access your plan by following the [reactivation process](#).

If you are not sure about your login details or require help, please call us.

Online Username	<input type="text"/>	<i>This must be at least 6 characters long and can have letters and / or numbers, but no spaces.</i>
Online Password	<input type="password"/>	<i>This must be at least 6 characters long and must have both letters and numbers, but no spaces.</i>

Log In

Figure 2-1: A typical login function

Despite their **superficial** simplicity, authentication mechanisms suffer from a wide range of defects in both design and implementation. Common problems may enable an attacker to identify other users' usernames, guess their passwords, or bypass the login function by exploiting defects in its logic. When you are attacking a web application, you should invest a significant amount of attention to the various authentication-related functions it contains. Surprisingly frequently, defects in this functionality enable you to gain unauthorized access to sensitive data and functionality.

Session Management

The next logical task in the process of handling user access is to manage the authenticated user's session. After successfully logging in to the application, the user accesses various pages and functions, making a series of HTTP requests from his browser. At the same time, the application receives countless other requests from different users, some of whom are authenticated and some of whom are anonymous. To enforce effective access control, the application needs a way to identify and process the series of requests that originate from each unique user.

Virtually all web applications meet this requirement by creating a session for each user and **issuing** the user a token that identifies the session. The session itself is a set of data structures held on the server that track the state of the user's interaction with the application. The token is a unique string that the application maps to the session. When a user receives a token, the browser automatically submits it back to the server in each subsequent HTTP request, enabling the application to associate the request with that user. HTTP cookies are the standard method for transmitting session tokens, although many applications use hidden form fields or the URL query string for this purpose. If a user does not make a request for a certain amount of time, the session is ideally expired, as shown in Figure 2-2.

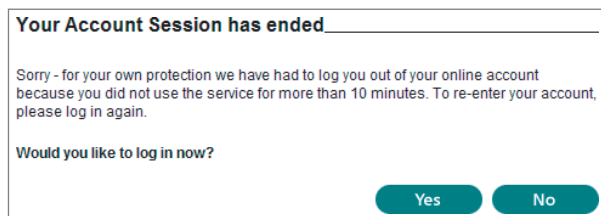


Figure 2-2: An application enforcing session timeout

In terms of attack surface, the session management mechanism is highly dependent on the security of its tokens. The majority of attacks against it seek to compromise the tokens issued to other users. If this is possible, an attacker can **masquerade** as the victim user and use the application just as if he had actually authenticated as that user. The principal areas of vulnerability arise from defects in how tokens are generated, enabling an attacker to guess the tokens issued to other users, and defects in how tokens are subsequently handled, enabling an attacker to capture other users' tokens.

A small number of applications **dispense** with the need for session tokens by using other means of reidentifying users across multiple requests. If HTTP's built-in authentication mechanism is used, the browser automatically resubmits the user's credentials with each request, enabling the application to identify the user directly from these. In other cases, the application stores the state information on the client side rather than the server, usually in encrypted form to prevent **tampering**.

Access Control

The final logical step in the process of handling user access is to make and enforce correct decisions about whether each individual request should be permitted or denied. If the mechanisms just described are functioning correctly, the application knows the identity of the user from whom each request is received. On this basis, it needs to decide whether that user is authorized to perform the action, or access the data, that he is requesting, as shown in Figure 2-3.

The access control mechanism usually needs to implement some fine-grained logic, with different considerations being relevant to different areas of the application and different types of functionality. An application might support numerous user roles, each involving different combinations of specific privileges. Individual users may be permitted to access a subset of the total data held within the application. Specific functions may implement transaction limits and other checks, all of which need to be properly enforced based on the user's identity.

Because of the complex nature of typical access control requirements, this mechanism is a frequent source of security vulnerabilities that enable an attacker

to gain unauthorized access to data and functionality. Developers often make flawed assumptions about how users will interact with the application and frequently make oversights by omitting access control checks from some application functions. Probing for these vulnerabilities is often **laborious**, because essentially the same checks need to be repeated for each item of functionality. Because of the **prevalence** of access control flaws, however, this effort is always a worthwhile investment when you are attacking a web application. Chapter 8 describes how you can automate some of the effort involved in performing rigorous access control testing.

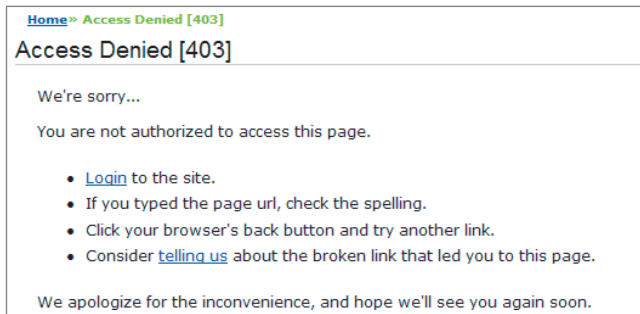


Figure 2-3: An application enforcing access control

Handling User Input

Recall the fundamental security problem described in Chapter 1: All user input is untrusted. A huge variety of attacks against web applications involve submitting unexpected input, crafted to cause behavior that was not intended by the application's designers. Correspondingly, a key requirement for an application's security defenses is that the application must handle user input in a safe manner.

Input-based vulnerabilities can arise anywhere within an application's functionality, and in relation to practically every type of technology in common use. "Input validation" is often cited as the necessary defense against these attacks. However, no single protective mechanism can be employed everywhere, and defending against malicious input is often not as straightforward as it sounds.

Varieties of Input

A typical web application processes user-supplied data in many different forms. Some kinds of input validation may not be feasible or desirable for all these forms of input. Figure 2-4 shows the kind of input validation often performed by a user registration function.

First Name	a	Must contain at least 4 characters
Last Name	a	Must contain at least 4 characters
Email	a	Please provide a valid email address
Phone number	a	Must contain only numbers

Figure 2-4: An application performing input validation

In many cases, an application may be able to impose very stringent validation checks on a specific item of input. For example, a username submitted to a login function may be required to have a maximum length of eight characters and contain only alphabetical characters.

In other cases, the application must **tolerate** a wider range of possible input. For example, an address field submitted to a personal details page might legitimately contain letters, numbers, spaces, hyphens, apostrophes, and other characters. However, for this item, restrictions still can be feasibly imposed. The data should not exceed a reasonable length limit (such as 50 characters) and should not contain any HTML markup.

In some situations, an application may need to accept arbitrary input from users. For example, a user of a blogging application may create a blog whose subject is web application hacking. Posts and comments made to the blog may quite legitimately contain explicit attack strings that are being discussed. The application may need to store this input in a database, write it to disk, and display it back to users in a safe way. It cannot simply reject the input just because it looks potentially malicious without substantially diminishing the application's value to some of its user base.

In addition to the various kinds of input that users enter using the browser interface, a typical application receives numerous items of data that began their life on the server and that are sent to the client so that the client can transmit them back to the server on subsequent requests. This includes items such as cookies and hidden form fields, which are not seen by ordinary users of the application but which an attacker can of course view and modify. In these cases, applications can often perform very specific validation of the data received. For example, a parameter might be required to have one of a specific set of known values, such as a cookie indicating the user's preferred language, or to be in a specific format, such as a customer ID number. Furthermore, when an application detects that server-generated data has been modified in a way that is not possible for an ordinary user with a standard browser, this often indicates that the user is attempting to probe the application for vulnerabilities. In these

cases, the application should reject the request and log the incident for potential investigation (see the “Handling Attackers” section later in this chapter).

Approaches to Input Handling

Various broad approaches are commonly taken to the problem of handling user input. Different approaches are often preferable for different situations and different types of input, and a combination of approaches may sometimes be desirable.

“Reject Known Bad”

This approach typically employs a blacklist containing a set of literal strings or patterns that are known to be used in attacks. The validation mechanism blocks any data that matches the blacklist and allows everything else.

In general, this is regarded as the least effective approach to validating user input, for two main reasons. First, a typical vulnerability in a web application can be exploited using a wide variety of input, which may be encoded or represented in various ways. Except in the simplest of cases, it is likely that a blacklist will omit some patterns of input that can be used to attack the application. Second, techniques for exploitation are constantly evolving. Novel methods for exploiting existing categories of vulnerabilities are unlikely to be blocked by current blacklists.

Many blacklist-based filters can be bypassed with almost embarrassing ease by making trivial adjustments to the input that is being blocked. For example:

- If `SELECT` is blocked, try `SeLeCt`
- If `or 1=1--` is blocked, try `or 2=2--`
- If `alert('xss')` is blocked, try `prompt('xss')`

In other cases, filters designed to block specific keywords can be bypassed by using nonstandard characters between expressions to disrupt the tokenizing performed by the application. For example:

```
SELECT/*foo*/username,password/*foo*/FROM/*foo*/users
<img%09onerror=alert(1) src=a>
```

Finally, numerous blacklist-based filters, particularly those implemented in web application firewalls, have been vulnerable to NULL byte attacks. Because of the different ways in which strings are handled in managed and unmanaged execution contexts, inserting a NULL byte anywhere before a blocked expression can cause some filters to stop processing the input and therefore not identify the expression. For example:

```
%00<script>alert(1)</script>
```

Various other techniques for attacking web application firewalls are described in Chapter 18.

NOTE Attacks that exploit the handling of NULL bytes arise in many areas of web application security. In contexts where a NULL byte acts as a string delimiter, it can be used to terminate a filename or a query to some back-end component. In contexts where NULL bytes are tolerated and ignored (for example, within HTML in some browsers), arbitrary NULL bytes can be inserted within blocked expressions to defeat some blacklist-based filters. Attacks of this kind are discussed in detail in later chapters.

“Accept Known Good”

This approach employs a whitelist containing a set of literal strings or patterns, or a set of criteria, that is known to match only benign input. The validation mechanism allows data that matches the whitelist and blocks everything else. For example, before looking up a requested product code in the database, an application might validate that it contains only alphanumeric characters and is exactly six characters long. Given the subsequent processing that will be done on the product code, the developers know that input passing this test cannot possibly cause any problems.

In cases where this approach is feasible, it is regarded as the most effective way to handle potentially malicious input. Provided that due care is taken in constructing the whitelist, an attacker will be unable to use crafted input to interfere with the application’s behavior. However, in numerous situations an application must accept data for processing that does not meet any reasonable criteria for what is known to be “good.” For example, some people’s names contain an **apostrophe** or **hyphen**. These can be used in attacks against databases, but it may be a requirement that the application should permit anyone to register under his or her real name. Hence, although it is often extremely effective, the whitelist-based approach does not represent an all-purpose solution to the problem of handling user input.

Sanitization

This approach recognizes the need to sometimes accept data that cannot be guaranteed as safe. Instead of rejecting this input, the application sanitizes it in various ways to prevent it from having any adverse effects. Potentially malicious characters may be removed from the data, leaving only what is known to be safe, or they may be suitably encoded or “escaped” before further processing is performed.

Approaches based on data sanitization are often highly effective, and in many situations they can be relied on as a general solution to the problem of malicious

input. For example, the usual defense against cross-site scripting attacks is to HTML-encode dangerous characters before these are embedded into pages of the application (see Chapter 12). However, effective sanitization may be difficult to achieve if several kinds of potentially malicious data need to be accommodated within one item of input. In this situation, a boundary validation approach is desirable, as described later.

Safe Data Handling

Many web application vulnerabilities arise because user-supplied data is processed in unsafe ways. Vulnerabilities often can be avoided not by validating the input itself but by ensuring that the processing that is performed on it is inherently safe. In some situations, safe programming methods are available that avoid common problems. For example, SQL injection attacks can be prevented through the correct use of parameterized queries for database access (see Chapter 9). In other situations, application functionality can be designed in such a way that inherently unsafe practices, such as passing user input to an operating system command interpreter, are avoided.

This approach cannot be applied to every kind of task that web applications need to perform. But where it is available, it is an effective general approach to handling potentially malicious input.

Semantic Checks

The defenses described so far all address the need to defend the application against various kinds of malformed data whose content has been crafted to interfere with the application's processing. However, with some vulnerabilities the input supplied by the attacker is identical to the input that an ordinary, nonmalicious user may submit. What makes it malicious is the different circumstances under which it is submitted. For example, an attacker might seek to gain access to another user's bank account by changing an account number transmitted in a hidden form field. No amount of syntactic validation will distinguish between the user's data and the attacker's. To prevent unauthorized access, the application needs to validate that the account number submitted belongs to the user who has submitted it.

Boundary Validation

The idea of validating data across trust boundaries is a familiar one. The core security problem with web applications arises because data received from users is untrusted. Although input validation checks implemented on the client side may improve performance and the user's experience, they do not provide any assurance about the data that actually reaches the server. The point at which

user data is first received by the server-side application represents a huge trust boundary. At this point the application needs to take measures to defend itself against malicious input.

Given the nature of the core problem, it is tempting to think of the input validation problem in terms of a frontier between the Internet, which is “bad” and untrusted, and the server-side application, which is “good” and trusted. In this picture, the role of input validation is to clean potentially malicious data on arrival and then pass the clean data to the trusted application. From this point onward, the data may be trusted and processed without any further checks or concern about possible attacks.

As will become evident when we begin to examine some actual vulnerabilities, this simple picture of input validation is inadequate for several reasons:

- Given the wide range of functionality that applications implement, and the different technologies in use, a typical application needs to defend itself against a huge variety of input-based attacks, each of which may employ a diverse set of crafted data. It would be very difficult to devise a single mechanism at the external boundary to defend against all these attacks.
- Many application functions involve chaining together a series of different types of processing. A single piece of user-supplied input might result in a number of operations in different components, with the output of each being used as the input for the next. As the data is transformed, it might come to bear no resemblance to the original input. A skilled attacker may be able to manipulate the application to cause malicious input to be generated at a key stage of the processing, attacking the component that receives this data. It would be extremely difficult to implement a validation mechanism at the external boundary to foresee all the possible results of processing each piece of user input.
- Defending against different categories of input-based attack may entail performing different validation checks on user input that are incompatible with one another. For example, preventing cross-site scripting attacks may require the application to HTML-encode the `>` character as `>`, and preventing command injection attacks may require the application to block input containing the `&` and `;` characters. Attempting to prevent all categories of attack simultaneously at the application's external boundary may sometimes be impossible.

A more effective model uses the concept of *boundary validation*. Here, each individual component or functional unit of the server-side application treats its inputs as coming from a potentially malicious source. Data validation is performed at each of these trust boundaries, in addition to the external frontier between the client and server. This model provides a solution to the problems just described. Each component can defend itself against the specific types of crafted input to which it may be vulnerable. As data passes through different

components, validation checks can be performed against whatever value the data has as a result of previous transformations. And because the various validation checks are implemented at different stages of processing, they are unlikely to come into conflict with one another.

Figure 2-5 illustrates a typical situation where boundary validation is the most effective approach to defending against malicious input. The user login results in several steps of processing being performed on user-supplied input, and suitable validation is performed at each step:

1. The application receives the user's login details. The form handler validates that each item of input contains only permitted characters, is within a specific length limit, and does not contain any known attack signatures.
2. The application performs a SQL query to verify the user's credentials. To prevent SQL injection attacks, any characters within the user input that may be used to attack the database are escaped before the query is constructed.
3. If the login succeeds, the application passes certain data from the user's profile to a SOAP service to retrieve further information about her account. To prevent SOAP injection attacks, any XML metacharacters within the user's profile data are suitably encoded.
4. The application displays the user's account information back to the user's browser. To prevent cross-site scripting attacks, the application HTML-encodes any user-supplied data that is embedded into the returned page.

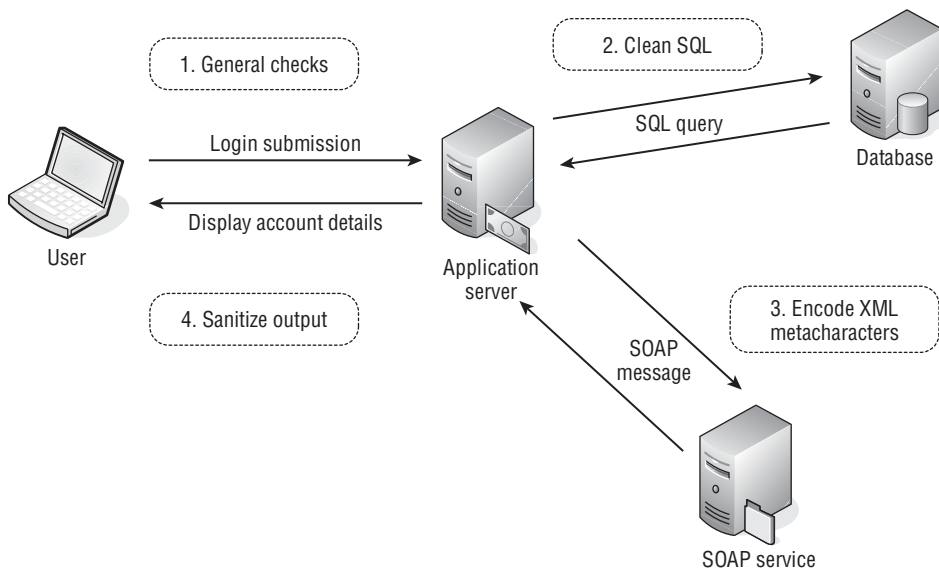


Figure 2-5: An application function using boundary validation at multiple stages of processing

The specific vulnerabilities and defenses involved in this scenario will be examined in detail in later chapters. If variations on this functionality involved passing data to further application components, similar defenses would need to be implemented at the relevant trust boundaries. For example, if a failed login caused the application to send a warning e-mail to the user, any user data incorporated into the e-mail may need to be checked for SMTP injection attacks.

Multistep Validation and Canonicalization

A common problem encountered by input-handling mechanisms arises when user-supplied input is manipulated across several steps as part of the validation logic. If this process is not handled carefully, an attacker may be able to construct crafted input that succeeds in smuggling malicious data through the validation mechanism. One version of this problem occurs when an application attempts to sanitize user input by removing or encoding certain characters or expressions. For example, an application may attempt to defend against some cross-site scripting attacks by stripping the expression:

```
<script>
```

from any user-supplied data. However, an attacker may be able to bypass the filter by supplying the following input:

```
<scr<script>ipt>
```

When the blocked expression is removed, the surrounding data contracts to restore the malicious payload, because the filter is not being applied recursively.

Similarly, if more than one validation step is performed on user input, an attacker may be able to exploit the ordering of these steps to bypass the filter. For example, if the application first removes `. . /` recursively and then removes `. . \` recursively, the following input can be used to defeat the validation:

```
....\ /
```

A related problem arises in relation to data canonicalization. When input is sent from the user's browser, it may be encoded in various ways. These encoding schemes exist so that unusual characters and binary data may be transmitted safely over HTTP (see Chapter 3 for more details). Canonicalization is the process of converting or decoding data into a common character set. If any canonicalization is carried out after input filters have been applied, an attacker may be able to use a suitable encoding scheme to bypass the validation mechanism.

For example, an application may attempt to defend against some SQL injection attacks by blocking input containing the apostrophe character. However, if

the input is subsequently canonicalized, an attacker may be able to use double URL encoding to defeat the filter. For example:

```
%2527
```

When this input is received, the application server performs its normal URL decode, so the input becomes:

```
%27
```

This does not contain an apostrophe, so it is permitted by the application's filters. But when the application performs a further URL decode, the input is converted into an apostrophe, thereby bypassing the filter.

If the application strips the apostrophe instead of blocking it, and then performs further canonicalization, the following bypass may be effective:

```
%%2727
```

It is worth noting that the multiple validation and canonicalization steps in these cases need not all take place on the server side of the application. For example, in the following input several characters have been HTML-encoded:

```
<iframe src=j&#x61;vasc&#x72ipt&#x3a;alert&#x28;1&#x29; >
```

If the server-side application uses an input filter to block certain JavaScript expressions and characters, the encoded input may succeed in bypassing the filter. However, if the input is then copied into the application's response, some browsers perform an HTML decode of the `src` parameter value, and the embedded JavaScript executes.

In addition to the standard encoding schemes that are intended for use in web applications, canonicalization issues can arise in other situations where a component employed by the application converts data from one character set to another. For example, some technologies perform a "best fit" mapping of characters based on similarities in their printed glyphs. Here, the characters « and » may be converted into < and >, respectively, and Š and Â are converted into ˇ and ˆ. This behavior can often be leveraged to smuggle blocked characters or keywords past an application's input filters.

Throughout this book, we will describe numerous attacks of this kind, which are effective in defeating many applications' defenses against common input-based vulnerabilities.

Avoiding problems with multistep validation and canonicalization can sometimes be difficult, and there is no single solution to the problem. One approach is to perform sanitization steps recursively, continuing until no further modifications have been made on an item of input. However, where the desired sanitization involves escaping a problematic character, this may result in an infinite loop. Often, the problem can be addressed only on a case-by-case basis, based on the types of validation being performed. Where feasible, it may be preferable to avoid attempting to clean some kinds of bad input, and simply reject it altogether.

Handling Attackers

Anyone designing an application for which security is remotely important must assume that it will be directly targeted by dedicated and skilled attackers. A key function of the application's security mechanisms is being able to handle and react to these attacks in a controlled way. These mechanisms often incorporate a mix of defensive and offensive measures designed to frustrate an attacker as much as possible and give the application's owners appropriate notification and evidence of what has taken place. Measures implemented to handle attackers typically include the following tasks:

- Handling errors
- Maintaining audit logs
- Alerting administrators
- Reacting to attacks

Handling Errors

However careful an application's developers are when validating user input, it is virtually inevitable that some unanticipated errors will occur. Errors resulting from the actions of ordinary users are likely to be identified during functionality and user acceptance testing. Therefore, they are taken into account before the application is deployed in a production context. However, it is difficult to anticipate every possible way in which a malicious user may interact with the application, so further errors should be expected when the application comes under attack.

A key defense mechanism is for the application to handle unexpected errors gracefully, and either recover from them or present a suitable error message to the user. In a production context, the application should never return any system-generated messages or other debug information in its responses. As you will see throughout this book, overly verbose error messages can greatly assist malicious users in furthering their attacks against the application. In some situations, an attacker can leverage defective error handling to retrieve sensitive information within the error messages themselves, providing a valuable channel for stealing data from the application. Figure 2-6 shows an example of an unhandled error resulting in a verbose error message.

Most web development languages provide good error-handling support through try-catch blocks and checked exceptions. Application code should make extensive use of these constructs to catch specific and general errors and handle them appropriately. Furthermore, most application servers can be configured to deal with unhandled application errors in customized ways, such as

by presenting an uninformative error message. See Chapter 15 for more details on these measures.

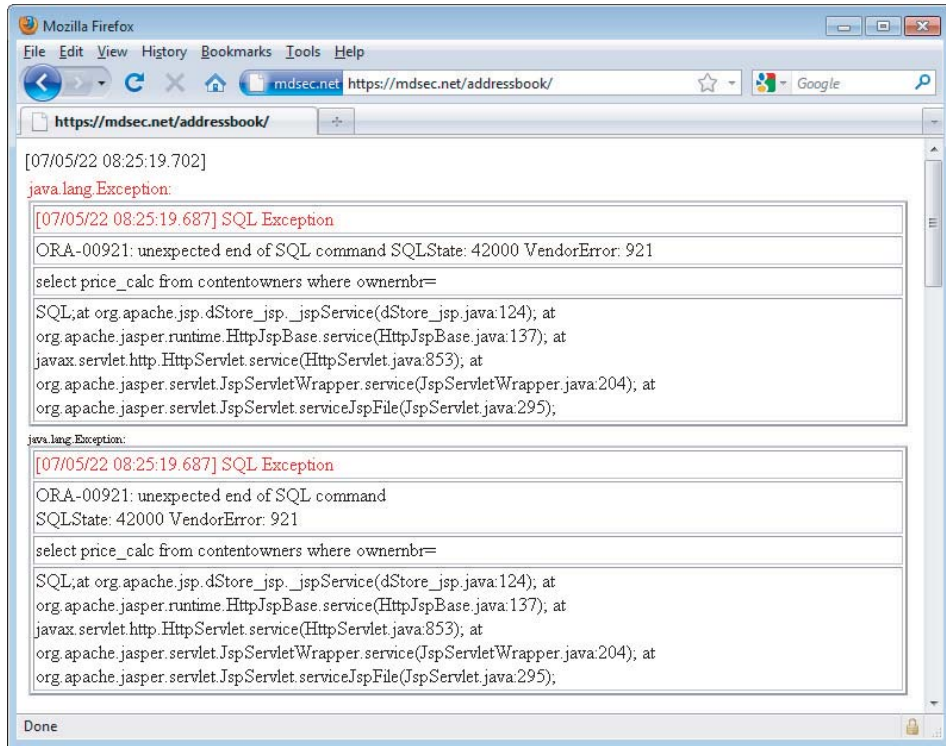


Figure 2-6: An unhandled error

Effective error handling is often integrated with the application's logging mechanisms, which record as much debug information as possible about unanticipated errors. Unexpected errors often point to defects within the application's defenses that can be addressed at the source if the application's owner has the required information.

Maintaining Audit Logs

Audit logs are valuable primarily when investigating intrusion attempts against an application. Following such an incident, effective audit logs should enable the application's owners to understand exactly what has taken place, which vulnerabilities (if any) were exploited, whether the attacker gained unauthorized access to data or performed any unauthorized actions, and, as far as possible, provide evidence of the intruder's identity.

In any application for which security is important, key events should be logged as a matter of course. At a minimum, these typically include the following:

- All events relating to the authentication functionality, such as successful and failed login, and change of password
- Key transactions, such as credit card payments and funds transfers
- Access attempts that are blocked by the access control mechanisms
- Any requests containing known attack strings that indicate overtly malicious intentions

In many security-critical applications, such as those used by online banks, every client request is logged in full, providing a complete forensic record that can be used to investigate any incidents.

Effective audit logs typically record the time of each event, the IP address from which the request was received, and the user's account (if authenticated). Such logs need to be strongly protected against unauthorized read or write access. An effective approach is to store audit logs on an autonomous system that accepts only update messages from the main application. In some situations, logs may be flushed to write-once media to ensure their integrity in the event of a successful attack.

In terms of attack surface, poorly protected audit logs can provide a gold mine of information to an attacker, disclosing a host of sensitive information such as session tokens and request parameters. This information may enable the attacker to immediately compromise the entire application, as shown in Figure 2-7.

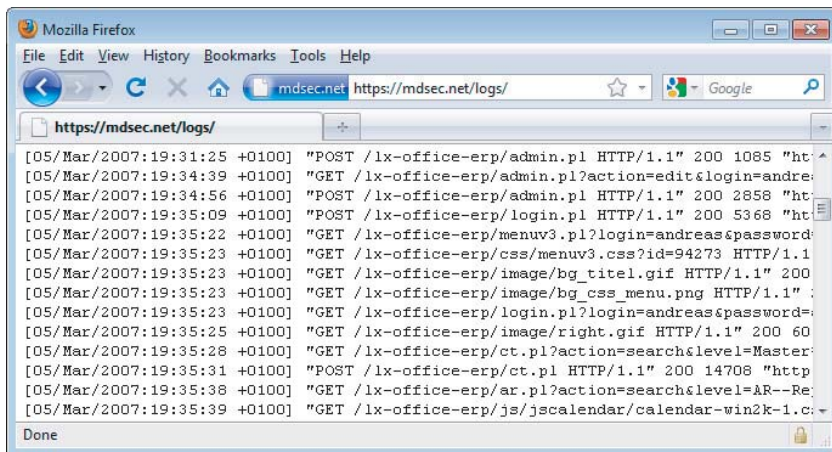


Figure 2-7: Poorly protected application logs containing sensitive information submitted by other users

Alerting Administrators

Audit logs enable an application's owners to retrospectively investigate intrusion attempts and, if possible, take legal action against the perpetrator. However, in many situations it is desirable to take much more immediate action, in real time, in response to attempted attacks. For example, administrators may block the IP address or user account an attacker is using. In extreme cases, they may even take the application offline while investigating the attack and taking remedial action. Even if a successful intrusion has already occurred, its practical effects may be mitigated if defensive action is taken at an early stage.

In most situations, alerting mechanisms must balance the conflicting objectives of reporting each genuine attack reliably and of not generating so many alerts that these come to be ignored. A well-designed alerting mechanism can use a combination of factors to diagnose that a determined attack is under way and can aggregate related events into a single alert where possible. Anomalous events monitored by alerting mechanisms often include the following:

- Usage anomalies, such as large numbers of requests being received from a single IP address or user, indicating a scripted attack
- Business anomalies, such as an unusual number of funds transfers being made to or from a single bank account
- Requests containing known attack strings
- Requests where data that is hidden from ordinary users has been modified

Some of these functions can be provided reasonably well by off-the-shelf application firewalls and intrusion detection products. These typically use a mixture of signature- and anomaly-based rules to identify malicious use of the application and may reactively block malicious requests as well as issue alerts to administrators. These products can form a valuable layer of defense protecting a web application, particularly in the case of existing applications known to contain problems but where resources to fix these are not immediately available. However, their effectiveness usually is limited by the fact that each web application is different, so the rules employed are inevitably generic to some extent. Web application firewalls usually are good at identifying the most obvious attacks, where an attacker submits standard attack strings in each request parameter. However, many attacks are more subtle than this. For example, perhaps they modify the account number in a hidden field to access another user's data, or submit requests out of sequence to exploit defects in the application's logic. In these cases, a request submitted by an attacker may be

identical to that submitted by a benign user. What makes it malicious are the circumstances under which it is made.

In any security-critical application, the most effective way to implement real-time alerting is to integrate this tightly with the application's input validation mechanisms and other controls. For example, if a cookie is expected to have one of a specific set of values, any violation of this indicates that its value has been modified in a way that is not possible for ordinary users of the application. Similarly, if a user changes an account number in a hidden field to identify a different user's account, this strongly indicates malicious intent. The application should already be checking for these attacks as part of its primary defenses, and these protective mechanisms can easily hook into the application's alerting mechanism to provide fully customized indicators of malicious activity. Because these checks have been tailored to the application's actual logic, with a fine-grained knowledge of how ordinary users should be behaving, they are much less prone to false positives than any off-the-shelf solution, however configurable or easy-to-learn that solution may be.

Reacting to Attacks

In addition to alerting administrators, many security-critical applications contain built-in mechanisms to react defensively to users who are identified as potentially malicious.

Because each application is different, most real-world attacks require an attacker to probe systematically for vulnerabilities, submitting numerous requests containing crafted input designed to indicate the presence of various common vulnerabilities. Effective input validation mechanisms will identify many of these requests as potentially malicious and block the input from having any undesirable effect on the application. However, it is sensible to assume that some bypasses to these filters exist and that the application does contain some actual vulnerabilities waiting to be discovered and exploited. At some point, an attacker working systematically is likely to discover these defects.

For this reason, some applications take automatic reactive measures to frustrate the activities of an attacker who is working in this way. For example, they might respond increasingly slowly to the attacker's requests or terminate the attacker's session, requiring him to log in or perform other steps before continuing the attack. Although these measures will not defeat the most patient and determined attacker, they will deter many more casual attackers and will buy additional time for administrators to monitor the situation and take more drastic action if desired.

Reacting to apparent attackers is not, of course, a substitute for fixing any vulnerabilities that exist within the application. However, in the real world, even the most diligent efforts to purge an application of security flaws may leave some exploitable defects. Placing further obstacles in the way of an attacker is an effective defense-in-depth measure that reduces the likelihood that any residual vulnerabilities will be found and exploited.

Managing the Application

Any useful application needs to be managed and administered. This facility often forms a key part of the application's security mechanisms, providing a way for administrators to manage user accounts and roles, access monitoring and audit functions, perform diagnostic tasks, and configure aspects of the application's functionality.

In many applications, administrative functions are implemented within the application itself, accessible through the same web interface as its core nonsecurity functionality, as shown in Figure 2-8. Where this is the case, the administrative mechanism represents a critical part of the application's attack surface. Its primary attraction for an attacker is as a vehicle for privilege escalation. For example:

- Weaknesses in the authentication mechanism may enable an attacker to gain administrative access, effectively compromising the entire application.
- Many applications do not implement effective access control of some of their administrative functions. An attacker may find a means of creating a new user account with powerful privileges.
- Administrative functionality often involves displaying data that originated from ordinary users. Any cross-site scripting flaws within the administrative interface can lead to compromise of a user session that is guaranteed to have powerful privileges.
- Administrative functionality is often subjected to less rigorous security testing, because its users are deemed to be trusted, or because penetration testers are given access to only low-privileged accounts. Furthermore, the functionality often needs to perform inherently dangerous operations, involving access to files on disk or operating system commands. If an attacker can compromise the administrative function, he can often leverage it to take control of the entire server.

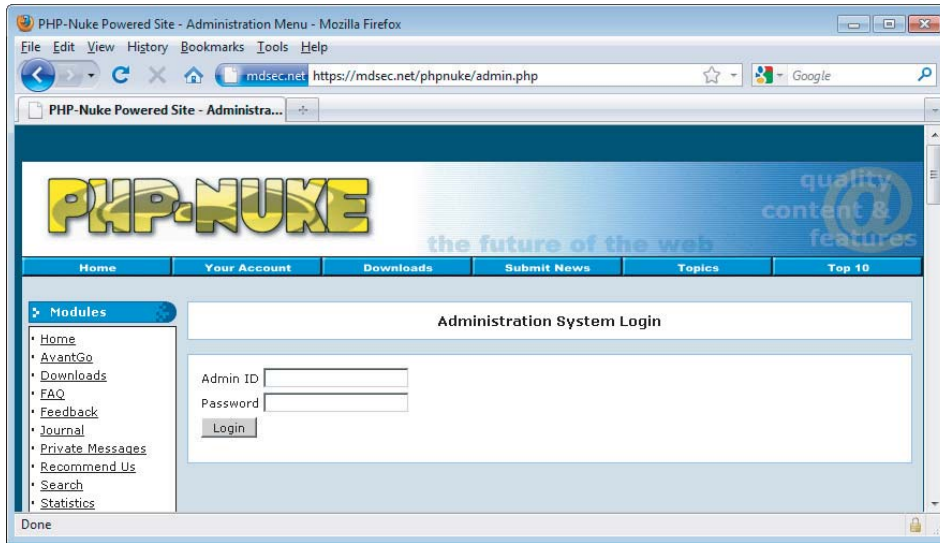


Figure 2-8: An administrative interface within a web application

Summary

Despite their extensive differences, virtually all web applications employ the same core security mechanisms in some shape or form. These mechanisms represent an application's primary defenses against malicious users and therefore also comprise the bulk of the application's attack surface. The vulnerabilities we will examine later in this book mainly arise from defects within these core mechanisms.

Of these components, the mechanisms for handling user access and user input are the most important and should receive most of your attention when you are targeting an application. Defects in these mechanisms often lead to complete compromise of the application, enabling you to access data belonging to other users, perform unauthorized actions, and inject arbitrary code and commands.

Questions

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

1. Why are an application's mechanisms for handling user access only as strong as the weakest of these components?
2. What is the difference between a session and a session token?
3. Why is it not always possible to use a whitelist-based approach to input validation?

4. You are attacking an application that implements an administrative function. You do not have any valid credentials to use the function. Why should you nevertheless pay close attention to it?
5. An input validation mechanism designed to block cross-site scripting attacks performs the following sequence of steps on an item of input:
 1. Strip any `<script>` expressions that appear.
 2. Truncate the input to 50 characters.
 3. Remove any quotation marks within the input.
 4. URL-decode the input.
 5. If any items were deleted, return to step 1.

Can you bypass this validation mechanism to smuggle the following data past it?

```
"><script>alert("foo")</script>
```