Chapter 12
Attack and Defense

THE FOLLOWING CEH EXAM TOPICS ARE COVERED IN THIS CHAPTER:

- ✓ **Firewalls**
- ✓ **Network security**
- ✓ **Boundary protection appliances**
- ✓ **Log analysis tools**
- ✓ **Security models**
- ✓ **Information security incidents**
- ✓ **Technical assessment methods**

Attacking is an important element of an ethical hacking engagement. Once you've gotten through all of your evaluation of the environment, you can move into trying to gain access to it. While social engineering can be an important element in an attack, it's not the only way in. Once you get inside, you will also want to work on moving laterally. This can require some technical attacks. It may involve engaging with web applications and the different vulnerabilities that may be exposed there. You may also need to know about denial-of-service attacks. It may be necessary to take out a server while you are performing spoofing attacks. There are also application-level compromises of listening services.

Another element to consider while you are attacking a network is what defenses are in place. Since the primary objective of ethical hacking is to improve the defensive posture of an organization, it's helpful to understand defensive strategies. On the attack side, knowing what may be in place will help you circumvent the defenses. When you are providing remediations to

the organization you are working with, you should have a better understanding of what might be done.

Web applications are a common way for users to get access to functions and data from businesses. There are several ways to attack web applications. Attacking a web application successfully may have multiple potential outcomes, including getting system-level access. Websites are also targets of denial-of-service attacks sometimes. While bandwidth consumption attacks may come to mind when you think of denial of service, it's not the only type of denial-of-service attack. There are some ways to perform denial-of-service attacks that remain effective and have nothing to do with bandwidth consumption.

Attackers will usually try to move from system to system—a process we call *lateral movement*. Because of this, and because an easy avenue is often through already open holes, it's important to have a good defensive strategy. While defense-in-depth and defense-in-breadth have long been thought to be good approaches to protecting the network, another way to think about the network design is implementing a defensible network architecture.

Web Application Attacks

Web applications are more than just a set of web pages. It's possible to have a static site where users have no programmatic interaction with the pages. Pages are presented to the user and there may be links and other content, but there is no place to enter anything into the page or otherwise send something to the server that may be acted on. A web application may have code within the page, using a scripting language, or code that runs on the server.

One way to think about a web application is from the architecture standpoint. One common way to develop an application has been through the use of the model/view/controller pattern. Figure 12.1 shows a diagram of how the model/view/controller pattern would be implemented through a web application. The model is the structure of the data. While it is often stored in a database, there is often an application server that manages the model by creating the structure and acting on it. The application server uses the database server for persistent storage, but all handling of the data (the

model) is done in the application server. Traditionally, this database server has been a relational database. This means all data manipulation is initiated in the application server, though execution of the logic may sometimes happen on the database server. Keep in mind that interaction with a database is done through the use of Structured Query Language (SQL) when using a relational database server. While the business and application logic happens in the application server, sometimes the manipulation is actually done on the database server.

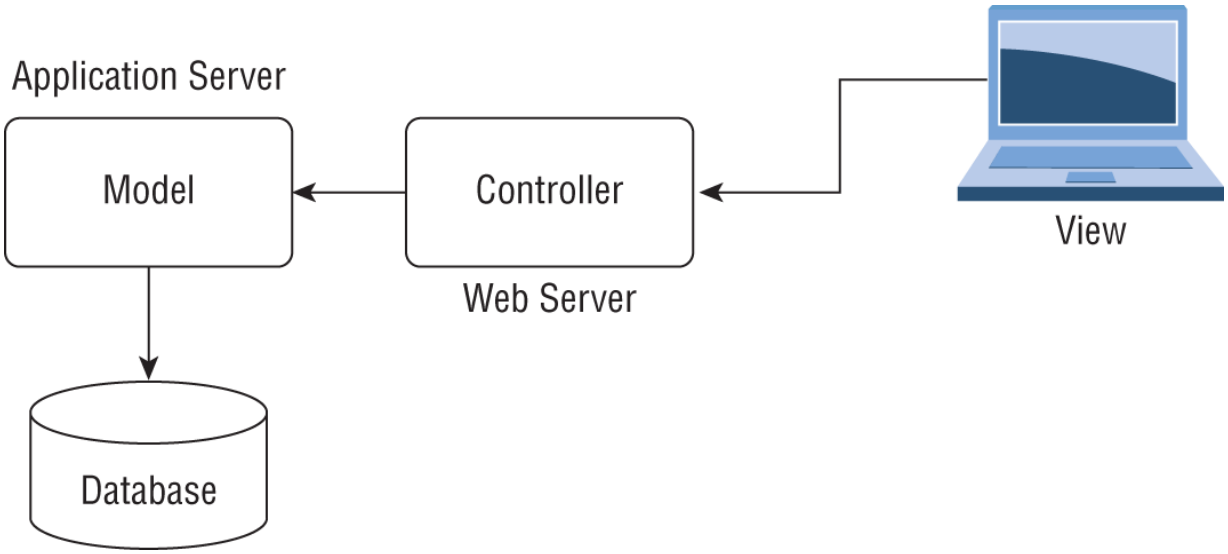


FIGURE 12.1 Model/view/controller design

The controller is the part of the application that receives requests and handles them, vectoring the request to the correct part of the application running in the application server. Finally, the view is the user interaction component, generally a web browser. The model is presented in a way that is consumable by a user in the browser. There is code in the form of Hypertext Markup Language (HTML) used to describe and render the pages. There may also be scripts that help with the function of the page in the browser.

The application server handles the business logic of the application and may use different programming languages, depending on the type of application server. On Windows, someone may be using Microsoft's Internet Information Services, which would have an application server supporting .NET languages like C# and Visual Basic. Java applications are available from many vendors. Tomcat, WildFly (formerly JBoss), WebLogic, and

many others are types of Java application servers. The application server itself may introduce different vulnerabilities and methods of attack, but there are also many generic attacks common across web applications.

One important consideration when you are looking at web application attacks, however, is which component is being attacked. Some attacks may be aimed at the client device (the view), while others may be targeting the web server (the controller). Others may be after the data storage element (the database). The point of attack, though, is not always where you would remediate the attack. This means that even if, for example, the client was the target of an attack, the client may not be the best place to protect against the attack happening.

OWASP Top 10 Vulnerabilities

The Open Web Application Security Project (OWASP) is really a collection of projects. One of these is the Top 10 Vulnerabilities project. This is a list of vulnerabilities that are common across web applications. While they have been specifically identified for web applications, many of the issues underlying these identified vulnerabilities exist in native applications as well. The list is updated periodically, though most of the vulnerabilities in the current list are variations on vulnerabilities that have existed in some form for many versions. They undergo name changes and consolidation over time, but many of these vulnerabilities have existed in some form for many years. The 2021 version of the list, updated from the 2017 version, includes the following vulnerabilities:

- **A01:2021 Broken Access Control**—This includes a number of different situations, including giving too many permissions to users, bypassing access control checks, elevating privileges, and forced browsing.
- **A02:2021 Cryptographic Failures**—This was previously called Sensitive Data Exposure, which isn't a vulnerability as much as it is an outcome from a vulnerability. This class of vulnerabilities includes not using encryption, using weak keys, using poor encryption mechanisms, using outdated encryption, and using improperly validated certificate elements.

- **A03:2021 Injection**—This is a category that used to be broken out into multiple vulnerabilities. Later in the chapter, there will be some specific examples of injection attacks, including SQL injection and cross-site scripting.
- **A04:2021 Insecure Design**—Poor design ends up introducing problems in applications because security was not considered. A better understanding of threats to an application would help introduce improved designs.
- **A05:2021 Security Misconfiguration**—This includes a lack of appropriate hardening of either the operating system or the application. It may also include allowing default usernames and passwords to be left in an application or system.
- **A06:2021 Vulnerable or Outdated Components**—A lack of patching leads to these vulnerabilities. Additionally, poor vetting of any libraries or frameworks can lead to their selection as part of an application. The selected library or framework may have vulnerabilities.
- **A07:2021 Identification and Authentication Failures**—This includes being vulnerable to attacks such as credential stuffing or brute-force attacks. Passwords stored in cleartext also fall into this class of vulnerabilities. Poor handling of session identification values is also a problem here.
- **A08:2021 Software and Data Integrity Failures**—This may be more commonly thought of as software supply chain failures, since it includes failures in code verification, as well as issues related to build pipelines that could result in the build tools creating malicious output. Any code or infrastructure that doesn't protect against integrity failures would fall into this category.
- **A09:2021 Security Logging and Monitoring Failures**—Security logging is not always considered when applications are developed. Worse, operations teams may not be monitoring any logs that do exist. Logs can be used to detect exploitations of some of the previously identified vulnerabilities.

- **A10:2021 Server-Side Request Forgery**—This was added to the list as a result of a request by the community and is a problem that occurs when the web application does not validate a user-provided URL. This may cause the application to send requests to unexpected destinations, which would allow the attacker to bypass other protections like firewalls or access control lists.

While there are a lot of remediations that cover all of these vulnerabilities, appropriate design and input validation result in a good start. The following are some examples of the different vulnerabilities identified by OWASP.

XML External Entity Processing

The XML external entity attack now falls under the category of security misconfiguration, though it was previously its own vulnerability. This is an attack that allows an attacker to manipulate data being sent to the server to get results not expected or otherwise allowed by the application.

A data model may be a complex set of data where multiple pieces of data are related and need to be organized together. This is very different than just passing a string from a web page to a web server. It may be passing multiple strings, a few integers, and also some organizing information. One way of organizing the data is to use the Extensible Markup Language (XML). XML allows you to put all related data together into a single data structure, complete with the organizing information indicating what each piece is. A common way to pass data back and forth between a client and a server is to use XML to structure the data and transmit that XML.

As you will see with other web application attack types, an XML entity injection attack comes from code on the web server accepting data that comes from the client without doing any data validation. This means programmers have not written any sort of check on the data when it enters the application. Even if it’s script code in the page contents that is rendering the data, it’s still possible for a user to tamper with it or even generate a data transmission to send to the server. As a result, any data coming from the user should never be trusted. As a general rule, data passed into a function from the outside should not be trusted because there is no guarantee the normal control flow of the program hasn’t been tampered with. When

programmers don’t do any checking of the data, bad data can slip into the program and alter the expected behavior.

An example of an XML package that could allow the attacker to gain access to system files is shown in the following code listing. The XML refers to an external entity from the system. In this case, it’s a file. The attacker is requesting that the contents of the file `/etc/passwd` be delivered back to the view.

XML External Entity Processing Example

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE wubble [
   <!ELEMENT wubble ANY>
   <!ENTITY xxe SYSTEM "file:///etc/passwd"  >]><wubble>&xxe;
</wubble>
```

This may not necessarily seem like a big deal. However, in some cases, you can get the application server to do some of your work for you. Let’s say you want to know whether an internal web server (e.g., one you, as the attacker, can’t access because of firewall rules) is running. One way of doing this is to have the application server initiate a request by passing a URL into the system tag. This would have the application server make a call to the system with a URL, which the system would interpret as a request to pull the page indicated. Here you can see an example of XML that would allow this to happen.

XML Request for Internal Page

```
<?xml version="1.0"?>
<!DOCTYPE GVI [<!ENTITY xxe SYSTEM
"https://172.30.42.25:8443">]>
```

Of course, doing this sort of resource checking, or port scanning if you will, is not the only thing you can do with this approach. The best way to prevent against this sort of attack is to do input validation. Input validation should be done on the server side. You can do input validation on the client side,

but expect that it will be bypassed. It's easy enough to use an intercepting proxy or even an in-browser tool to capture requests and modify them before sending them off to the server.

In examples such as this, simply not allowing any XML to use the external entity SYSTEM is a start toward protecting against this attack being successful. Any action the application has to perform through the system should be done directly by the application after it has parsed the data. Going a step further, evaluating if there is a way around the use of any external entity would be useful. Again, any external need should probably be handled directly by the application code after it has properly evaluated any data. This means all the external entities could be ignored when parsing the XML.

An external entity XML attack targets the server. This could be the web server or the application server, depending on who handles parsing the XML.

Cross-Site Scripting

A cross-site scripting (XSS) attack is one that uses the web server to attack the client side. This is an injection attack, according to the OWASP Top 10. This injects a code fragment from a scripting language into an input field to have that code executed within the browser of a user visiting a site. There are three ways to run a cross-site scripting attack. They all target the user. The difference is whether the script is stored somewhere or not. The first type, stored on the server and displayed for any user visiting a page, is called *persistent cross-site scripting*. The second is called *reflected cross-site scripting*. The last makes use of the Document Object Model (DOM).

You'll note that cross-site scripting is abbreviated XSS, where the *X* is a cross, rather than CSS, where the *C* is the first letter in *cross*. There is another web technology, Cascading Style Sheets, that is abbreviated CSS, which could easily become confusing if they were both CSS.

All three types make use of a scripting language in the same way. The attacker would use a short script inside a `<script></script>` block. For example, the attacker may inject `<script>alert (‘wubble’);</script>` into an input field. With persistent cross-site scripting, this would be a field where the value is stored on the server. A classic example of this sort of attack is a web forum where a user may be asked for a name, an email address, and then a comment. All of the values from these fields in the web form would be stored. When someone came back to the forum, the information would be displayed, including the script, which would be run by the browser. In the case of the previous example, an alert dialog box would be displayed with the word *wubble* in it.

When it comes to a reflected attack, the script isn’t stored. Instead, it is included in a URL you would send to a victim. This sort of URL would look like this, and you can see the script passed in as a parameter:

```
http://www.badsite.com/foo.php?param=<script>alert(5000);
</script>
```

You would send this URL to a victim and do something to get them to click it, such as something like saying Apple is giving away free iPhones to the first 1,000 people to click this link. Then you’d make the link refer to the preceding URL. You can do this in an HTML email using the following HTML tag:

```
<a href=" http://www.badsite.com/foo.php?param=
<script>alert(5000);
</script>"</a>
```

The user, getting this email, would only see the clickable link. To see the URL the link goes to, they would have to hover their cursor over the link or perhaps look at the source code of the message. Even this assumes the user is capable of understanding HTML enough to know what is being done here. Obviously, a real script would do something more malicious, like stealing session information stored in cookies. Once the information was stolen, it could be transmitted to an attacker by making a request with the stolen information in it to a server managed by the attacker.

The DOM is what enables the DOM-based XSS attack. DOM is a way of referring to elements in the page as though they were objects. This means you can call methods on the objects—specifically, you can get and set parameters for those objects. The DOM-based XSS would look just like the reflected XSS. The difference is that the variable referred to in the URL would be a portion of the page. Inside the page is a script that pulls that variable and does something based on what the value is. Sending a URL with a value of a script inside the variable will insert the script into the page, meaning the script will get executed.

Since scripts can become complex and may use characters that are illegal in a URL, such as a space or other special characters, the characters need to be encoded. You encode these characters using something called URL encoding. To URL-encode something, you convert the ASCII characters into a numeric value using the ASCII table. The numeric value is rendered in hexadecimal, and that value has a % in front of it to indicate that the following value is a hexadecimal representation of an ASCII character. Back to the space character, the decimal value of the space character in the ASCII character set is 32. In hexadecimal, that's 20 (2 × 16). To render a space character into a URL, you would use %20. Anytime you see %20 in a URL, you know it's replacing a space character, which wouldn't be handled correctly by a server since white space is sometimes considered a delimiter, meaning the character before the space would be interpreted as the last character in the URL to the web server.

Because encoding is often required in normal operations to pass information from the server to the client, this opens the door to encoding attacks. Attackers can similarly encode information, which makes it harder to detect encoded attacks. This is, in part, because of the number of types of encoding that can be used, which means defense has to be aware of the different types of encoding so decoding can be performed before looking for the attack. This is one reason why website attacks are so common—the defense against these attacks is difficult.

SQL Injection

Of course, cross-site scripting is not the only place where URL encoding is used. Anywhere you want to obscure text, you can URL-encode it. It makes it harder for regular users to know what is being done because they can't

easily convert the hexadecimal values back to their ASCII equivalents. Many of the other attacks can be obscured or made successful through the use of URL encoding, and SQL injection is one of them. SQL is the Structured Query Language, used to make programmatic requests of a relational database server. The server manages the storage, which will vary from one server type to another, but to get to the storage, either by inserting data or retrieving it, you would use the SQL programming language.

An SQL attack is an attack against the database server, ultimately, though it takes advantage of poor programming practices in the application code. As an injection attack, it happens when a malicious user sends in (injects) unexpected data through a web request. Sometimes, form data is passed directly into an SQL query from the application server to the database server to execute. If a fragment of legitimate SQL can be sent into a server where it is executed, the database could be altered or damaged. Also, data could be extracted or even have authentication bypassed.

Applications you are trying to attack with SQL injection must already have SQL in place to run a query necessary for the application to succeed. For example, perhaps you are browsing an electronic storefront and you are trying to find all *Doctor Who* action figures because you really want one of Jodie Whittaker, since you have all the others. You issue a search on the site for *jodie whittaker doll* in order to get the right results back. Behind the scenes, the application may have an SQL statement that reads as follows:

```
SELECT * FROM inventory_table WHERE description ==
'$searchstr';
```

What you have submitted (*jodie whittaker doll*) goes into the variable `$searchstr`. This means it's not as simple as just issuing your own query and expecting that to run. Instead, you need to find a way to get your query to work in the context of the existing one. Figure 12.2 shows the results from a successful SQL injection attack using one of the potential techniques.

FIGURE 12.2 SQL injection attack outcome

This particular attack was run against a web application named Damn Vulnerable Web App, which is a deliberately vulnerable web application that you can use to learn more about attacks, trying against different levels of security in the application. The string entered was `' or 'a' = 'a`. This uses the single quote to close out the existing query string and then introduces the Boolean logic term `or` along with a test that will return a true. This means the overall expression will return true. Every row in the database will evaluate to true. Since the query here likely starts with `SELECT *`, we get every row in the database back. You'll notice that the query inserted leaves the last quote out. The reason for this is that the query in the application already has a single quote at the end. I introduced a single quote to terminate the evaluation in the program, but there is still a single quote hanging out there, so if I leave out the single quote, the one in place will replace it, leaving a valid SQL statement.

This may not always work, depending on the SQL query in place in the application. In some cases, you may need to replace the rest of the query in the application. This can be done using comment characters. If the SQL server that underpins the application is based on MySQL, including both

MySQL and MariaDB, you can use the double dash (- -) to indicate a comment that goes to the end of the line. Everything after your double dash is commented out. This can allow you to inject your own complete SQL statement and then just comment out everything else in the application. MySQL syntax also allows the use of the # character to indicate a comment. It functions just like the double dash. On an Oracle server or Microsoft SQL Server, the double dash also works.

Since SQL doesn't always work the same from one server type to another, the question becomes how you can determine what the underlying database server is. You may be able to know based on the reconnaissance you did earlier. For example, a poorly configured Apache server may tell you what modules are loaded. You may be able to tell based on the results of an application version scan that there is a MySQL server. If you are testing against a Microsoft IIS installation, you could guess it may be Microsoft SQL Server. A more definitive way would be to introduce invalid SQL into the application. If the application is well programmed, you shouldn't get anything back, but often you will get an error from the database server. Either you will get an error message that includes what the database server is or you will get an error message that allows you to determine the server type because of the wording of the error.

In the case in Figure 12.2, we got output from our query. Not all queries will generate output. This means you are flying blind because you can't see the results of your query. In that case, you are issuing blind SQL injection attacks. Because you may be flying blind, you need to change your approach. You need to structure your query so you get either success or failure that you can see because of how the application works. The purpose of a blind injection is to see if the page behaves differently based on the input provided. This may allow you to determine whether a page is vulnerable to SQL injection before you spend a lot of time trying to run these attacks.

Let's say you have a search field, as we discussed before. Instead of a full string, you could do something simple like *jodie* instead of the complete *jodie whittaker doll* search used earlier. Along with that, though, you're going to try to get a false by appending *and 1 = 2*. Since 1 never equals 2, the entire search should return false. Once you see the results from that page, you can run another query that should generate a true. Instead of 1 =

2, you can specify 1 = 1. If you get a different result from the query 1 = 1 than you did from 1 = 2, the page may be vulnerable to SQL injection. You can make this assumption because the response from the application was different when different SQL was submitted.

There are a couple of ways to remediate against these attacks. The first is to screen any input from the user, just as you were doing for the other attacks. You can look for special characters and strip them out. You can look for anything that looks like an SQL statement and remove those elements. This may include the use of a feature like allow list, which is a list of text that would be allowed.

Another way is to use parameterized queries or stored procedures. These have similar effects because you can't manipulate the string in the program. Values sent from the user side become parameters passed into the queries. Because of that, comment characters won't eliminate the rest of the query string. Additionally, anything that tries to insert SQL statements by manipulating the quoting won't work because the behavior as a parameter is different from just inserting text into a string value.

Command Injection

The outcome from a command injection attack may be similar to an XML external entity injection attack. This is another injection attack, according to the OWASP Top 10. The target of this sort of attack is the operating system. The application takes a value from the user and passes it to a system function or an evaluate function. These sorts of functions pass the parameters to the operating system to handle. This means if there is no input validation, you can pass an operating system command into the input field and that operating system command will be executed. You can see an example of the potential results of a command injection attack in Figure 12.3. The input box expects an IP address or hostname. The application runs `ping` with the IP address as the parameter to the program. This demonstration uses the Damn Vulnerable Web Application (DVWA) for a target platform.

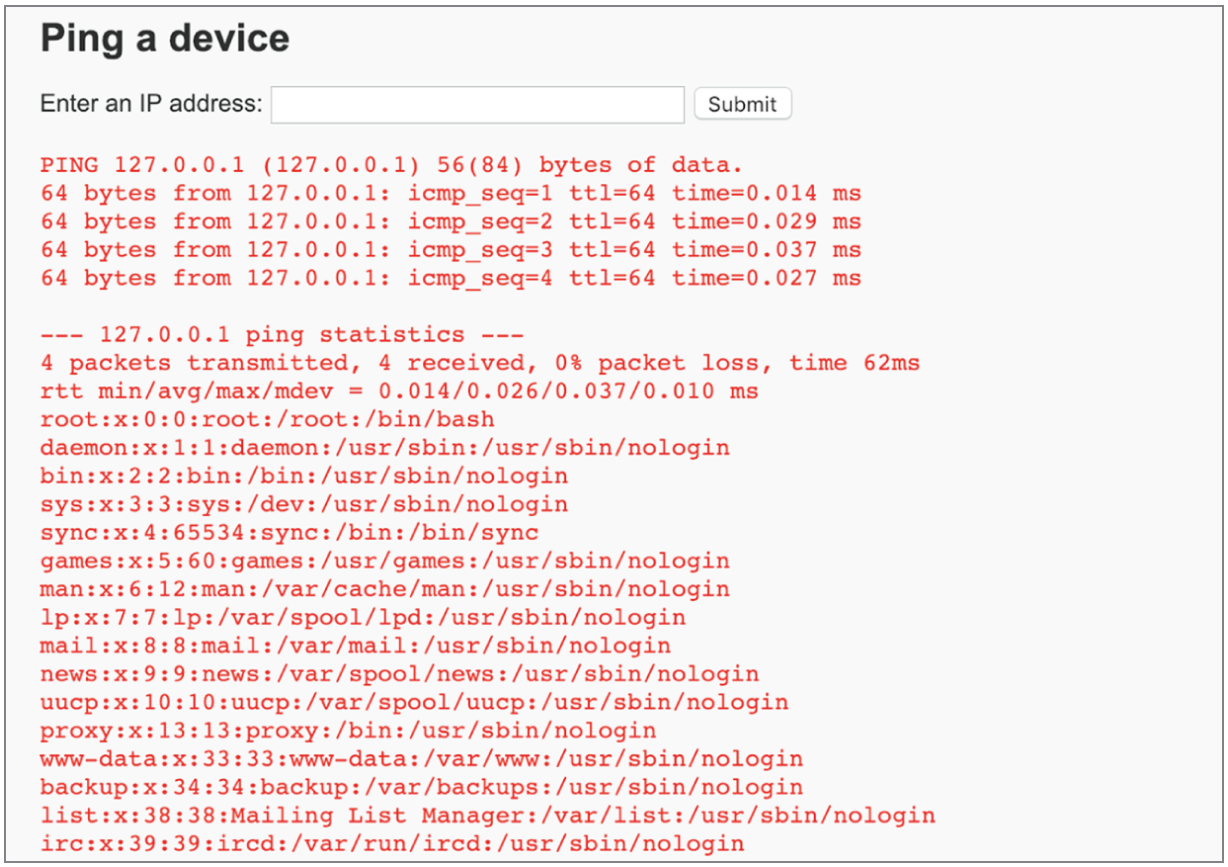


FIGURE 12.3 Command-line injection attack output

You can take advantage of this by using a command delimiter. This means knowing the operating system that underpins the web application. This is because the command delimiter on Linux is different from that on Windows. On Linux, which is the operating system under the web page in Figure 12.3, the string entered was `127.0.0.1; cat /etc/passwd`. The semicolon (`;`) was used to terminate the command `ping 127.0.0.1` before running the second command, `cat /etc/passwd`. On a Windows system, the semicolon is used to separate parameters. Instead, you would use the ampersand (`&`) between commands through the Windows command processor.

With both Windows and Linux, you can use Boolean operators between commands. This conditions the execution of the second command on the success of the first. As an example, `ping -c 5 127.0.0.1 && cat /etc/passwd` would mean the `cat` command would run only if the `ping` command succeeded. If you wanted to condition in the other direction, meaning run the second command only if the first one failed, you could use

something like `dir \Windows | echo wubble`. If the `dir` command fails, for some reason, the `echo` command is run.

As with the other web application attacks, there are a couple ways of remediating. As always, input validation is the right answer. Know what the input should look like. If anything doesn't look like you expect it to look, fail the input and demand something cleaner from the user. In the case in Figure 12.3, an IP address has a clearly defined structure. You could check input against that structure. If the input doesn't look like an IP address, you don't accept any of the input. You could also structure the application such that no input from the user is ever directly passed to the system, even if it's part of a command.

Directory or File Traversal

Directory or file traversal has been a common attack, as identified previously by EC-Council. This is an attack used to try to break out of a jail constructed by a web server. A web server uses a subset of the entire file system to serve files from. If a request comes in to `www.webserver.com`, the request goes to the index page in the directory the web server believes to be the top of the directory tree. On a Linux system, for instance, that may be `/var/www/html`. The request to the URL `www.webserver.com` would have the receiving web server respond with the contents of the page `index.html`, assuming that was the configured index page according to the web server.

File Traversal

This type of attack, file traversal, is also called *directory traversal* or *file path traversal*. If you see any of these terms, they all mean the same thing: trying to break out of the jailed root of a web server in order to get to the files within an operating system.

This configuration means you couldn't make a request to `http://www.webserver.com/etc/passwd`, for instance, and get the passwd file in the `/etc` directory. As far as the web server is concerned, there is no

such thing as the `/etc` directory because life begins and ends with the `/var/www/html` directory. Every request is relative to that top-level directory. However, as you may know if you are familiar with command-line utilities, you can refer to paths relative upward as well as downward.

Let's say you were using a command-line interface, commonly called a *shell*. If you start in `/var/www/html`, you can still get to the root of the file system by referring to the parent directory. In `/var/www/html`, we can use `../../../etc` to get to the `/etc` directory because the expression `../` pops us up one level. You can see gaining access to the `/etc/passwd` file using this technique in the following example:

```
kilroy@yaz:/usr/share/modsecurity-crs/rules$ cd /var/www/html
kilroy@yaz:/var/www/html$ ls
index.nginx-debian.html
kilroy@yaz:/var/www/html$ sudo cat ../../../etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
```

You can use the same technique in some web servers, if they are vulnerable to this approach. Of course, this is a technique that has been used for years, so most web servers are on to it and don't allow it. However, web applications can be exploited to bypass this protection on the part of the web server. If a web application contains code that reads files from disk and especially if those files are then presented to the user through the browser interface, you can use a file traversal attack. A common approach with this type of attack is to stack a lot of the `../../` combinations together. If you don't provide enough, you won't be at the top of the file system. If you provide too many, they will just get ignored because at some point you are at the top of the tree and simply can't go any higher.

The file traversal attack is used to obtain sensitive information like the passwd file (though on a Unix-like operating system, the shadow file contains the password hashes so is probably more useful). You may also want other types of files, depending on what may be housed on the server. If there happens to be a database of usernames and passwords for the application stored on this server, you might be able to retrieve that file using

a file traversal attack. While you may not be able to use this sort of attack in the address bar of a web browser, you may be able to in fields within a web application.

URL Manipulation

Applications often have parameters or other information passed through the uniform resource locator (URL). Tampering with the information provided to the application in the URL may result in bypassing the application's control flow. When using the common gateway interface (CGI) model, passing parameters into the application through the URL includes the ? character. One example of this would be a URL like `www.bank.com/account.php?acctno=123456`. The parameter `acctno` is the account number and has a value of 123456. If there were more complex activities being performed by `account.php`, for example, a bank transfer, manipulating the URL may result in causing a transfer to the attacker's account.

Modern web applications use endpoints as part of how the application communicates between the client and the server. An endpoint is a structured URL. It looks like a regular URL, except that the URL contains messages that are passed to the application, causing specific functions to be called. You might see `www.bank.com/transfer`, for instance. The last part of the URL, `transfer`, is the endpoint. It tells the receiving application to call the transfer function. The function then has to take care of input and output, as well as any additional communication, such as passing data in an asynchronous fashion using either XML or JSON. Adjusting the endpoint in the URL can perhaps call an unpublished function or get enough information out of the server to know what else may need to be sent.

URL manipulation may also be used for forced browsing. Forced browsing is using a dictionary of pages or resources to send requests to determine whether the page or resource exists. This can be a good way to identify pages or resources that were meant to be hidden. You may be able to bypass access controls in some cases, as the expectation on the part of the developer is that the only way to get access to the resource is through a page that does require authentication.

Web Application Protections

When it comes to protecting web applications, there are some common approaches. One of the easiest is to use basic input validation. This would mean the requirements of the application be specified to indicate what the input should look like up front. Unless you know what the input should look like, it's almost impossible to know what it shouldn't look like so you could rule out anything that looked like that. It's not always easy to just strip or block special characters. Once you start stripping special characters, you end up with a lot of names that can't be rendered correctly, for instance. There are also a lot of other things you can't do, like having a broader palette of symbols for complex passwords. Suddenly, you have a smaller character set, which can make for passwords that are easier to brute-force.

Even if you know what you are looking for in malformed data, you can still end up shooting yourself in the foot. A common approach to identifying malicious patterns is to use regular expressions. Regular expressions are effectively a pattern matching language that is commonly used in Unix shell scripts but has since been ported to multiple other uses. Programming languages use regular expressions commonly. In fact, a variation on traditional regular expressions is called *Perl-compatible regular expressions* (PCREs), which originated in the programming language Perl but have been implemented in other places.

This is not the best place to attempt to teach regular expressions, as it's a complex and compact language. An example, just to demonstrate some of the complexity, may be something like `^[0-9a-fA-f]+$`. This example says to start at the beginning of a line (^) and then look for any number of characters that are numeric digits or the letters *a* through *f* in either uppercase or lowercase. Finally, the `$` indicates the end of a line. The short version of this explanation is it looks for hexadecimal values that are on a line by themselves. The regular expression (regex) would not match on anything other than that.

With a language so potentially complex, it's not unheard of to make mistakes or end up with something more complex than it needs to be. What happens in cases where you have regex used to protect web applications in code is you may cause the program to loop around endlessly trying to resolve the regex given the input provided. This is something called *regex denial of service* (ReDoS). Web applications can become their own problem

if programming is not handled well. One way to address a problem like this is to use libraries that have already been well-tested for input validation.

Even using good programming practices, though, it's possible for someone to find a vulnerability in it at some point. This is where having either protection or some detective capabilities in place is useful. A web application firewall (WAF) is an application layer firewall that reads the HTTP requests and matches them against patterns of attack. As with other firewalls, you can write the rules to alert on the traffic or block the request. While there are a lot of commercial WAFs available, you can also use `mod_security`, which is a module that gets loaded into the Apache web server. Many WAFs sit out in front of a web server as a separate application or appliance, but this is built into the web server itself. The following is an example of a `mod_security` rule:

```
SecRule ARGS_NAMES|ARGS|XML:/* "@rx ((?:[~!@#\$%\^&\*\(\)\-
\+=\{\}\[\]\|:;\"'^''`<>][^~!@#\$%\^&\*\(\)\-\+=\{\}\
[\]\|:;\"'^''`<>]*?){2})" \
    "id:942432,\
    phase:2,\
    block,\
    capture,\
    t:none,t:urlDecodeUni,\
    msg:'Restricted SQL Character Anomaly Detection (args): #
of special characters exceeded (2)',\
    logdata:'Matched Data: %{TX.1} found within %
{MATCHED_VAR_NAME}: %{MATCHED_VAR}',\
    tag:'application-multi',\
    tag:'language-multi',\
    tag:'platform-multi',\
    tag:'attack-sqli',\
    tag:'OWASP_CRS',\
    tag:'OWASP_CRS/WEB_ATTACK/SQL_INJECTION',\
    tag:'WASCTC/WASC-19',\
    tag:'OWASP_TOP_10/A1',\
    tag:'OWASP_AppSensor/CIE1',\
    tag:'PCI/6.5.2',\
    tag:'paranoia-level/4',\
    ver:'OWASP_CRS/3.2.0',\
    severity:'WARNING',\
    setvar:'tx.anomaly_score_pl4=+%
{tx.warning_anomaly_score}',\
    setvar:'tx.sql_injection_score=+%
{tx.warning_anomaly_score}'"
```

Most of this rule is about providing metadata that can be provided in the alert. For example, this vulnerability is one that is in the OWASP Top Vulnerabilities list. You'll find that fact in one of the tags in the rule. The important part of the rule is the first line indicating the pattern to be matched. This is a regular expression indicating what a SQL injection attack may look like. `mod_security` uses a phased approach to assessing rules. The first phase looks at request headers. The second phase, which this rule falls into according to the third line, looks at the request body. The other phases look at response headers, response body, and logging.

As with any rule-based security technology, any protection afforded by that technology relies on the rules. If they are written poorly or if rules don't exist for a particular attack, then attacks can make it through the WAF. This is not to say the WAF is a bad idea. It means that it requires well-tested rules in place and a keen eye on modern attacks to introduce new rules. Additionally, it means applications need to be written in a defensive manner since the technologies out in front may not be sufficient to keep attacks out of the application.

Testing Web Application Attacks

The best way to learn how these attacks work is to practice them. Fortunately, there are easy ways to do this without impacting any legitimate or production application. You can download the Damn Vulnerable Web Application (DVWA) from `dvwa.co.uk`. This will give you a safe space with an application that is specifically vulnerable to the most common web application attacks. You can also test and understand evasion techniques by turning on a web application firewall built into DVWA. You can indicate the level of protection used by the WAF to understand protections a WAF and application may use against attacks.

Denial-of-Service Attacks

Remember the triad of confidentiality, integrity, and availability. So far, we've been talking about integrity when it comes to the web application

attacks. The attacks impact the integrity of the overall application. If the attacker manages to gain access to the system through one of the attacks, we start talking about confidentiality. In the following sections, we're going to talk about availability. The purpose of a denial of service is to take an application out of service so legitimate users can't use it. As users often interact with businesses through web applications, they are often the target of denial-of-service attacks. This is also because attackers are often outside the enterprise and web servers are generally exposed to the Internet.

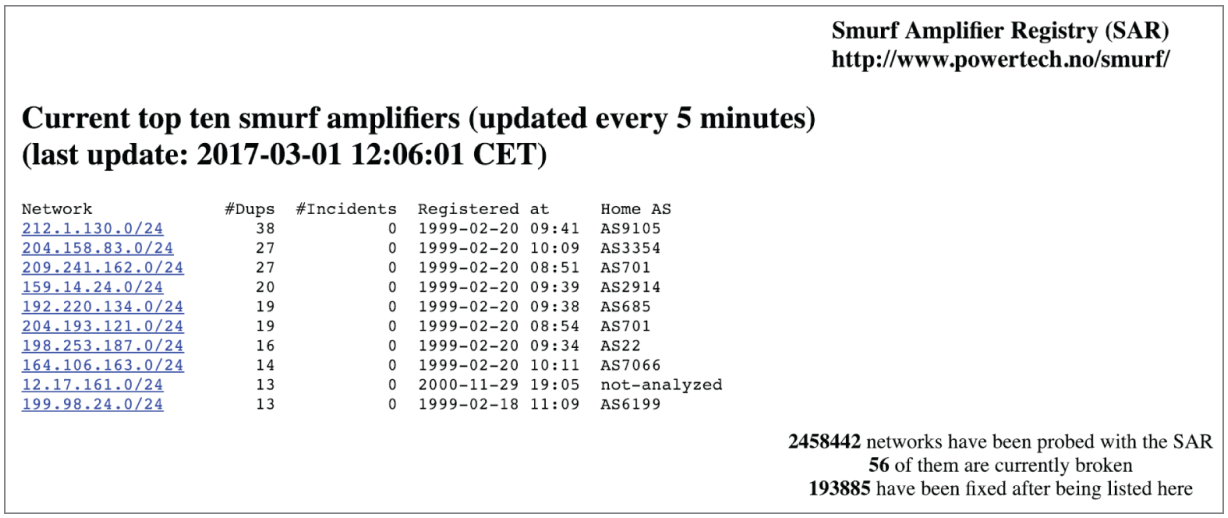
Bandwidth Attacks

A bandwidth attack is used to generate a lot of traffic that overwhelms the network connection a service is using. A problem with bandwidth attacks, especially today, is that your target will likely have a lot more bandwidth capacity than you do. This means alone you can't possibly generate enough traffic to take a site offline. You can take a couple of approaches to this. The first is to just gang up on your target. This may mean you get all of your friends together and send as much data to your target as you absolutely can. If you have enough friends, you may be able to generate enough traffic to take a service offline.

Another approach that's similar to this but more practical, especially if you don't have a lot of friends, is to use a large number of compromised systems. This could be done using a botnet that has the capability of issuing requests to a target. The larger the botnet, the more traffic you can generate. This does assume you either have access to a botnet or know where you can rent one. Not everyone spends time acquiring botnet clients, however. This means you need to take another approach.

Fortunately, there is another approach you can take to get a lot back for putting very little in. Amplification attacks have long been a possibility. One of the first protocols used for amplification attacks was ICMP. A Smurf attack used an ICMP echo request to a broadcast address. If you selected a large enough network and made your requests large enough, you could get a huge volume of large responses. Since the request would have a spoofed source address, the responses would all go to your target. At its peak in the late 1990s and early 2000s, pinging a broadcast address could get tens of thousands of responses, called *duplicates* because after the first response to a request, every successive response is considered a duplicate. This sort of

attack doesn't have nearly the same impact today as it used to. Figure 12.4 shows the largest Smurf amplifiers that still existed in 2017 before the Smurf amplifier registry stopped checking. This page no longer exists except in archives.



Smurf Amplifier Registry (SAR)
http://www.powertech.no/smurf/

Current top ten smurf amplifiers (updated every 5 minutes)
(last update: 2017-03-01 12:06:01 CET)

Network	#Dups	#Incidents	Registered at	Home AS
212.1.130.0/24	38	0	1999-02-20 09:41	AS9105
204.158.83.0/24	27	0	1999-02-20 10:09	AS3354
209.241.162.0/24	27	0	1999-02-20 08:51	AS701
159.14.24.0/24	20	0	1999-02-20 09:39	AS2914
192.220.134.0/24	19	0	1999-02-20 09:38	AS685
204.193.121.0/24	19	0	1999-02-20 08:54	AS701
198.253.187.0/24	16	0	1999-02-20 09:34	AS22
164.106.163.0/24	14	0	1999-02-20 10:11	AS7066
12.17.161.0/24	13	0	2000-11-29 19:05	not-analyzed
199.98.24.0/24	13	0	1999-02-18 11:09	AS6199

2458442 networks have been probed with the SAR
56 of them are currently broken
193885 have been fixed after being listed here

FIGURE 12.4 Smurf amplifier registry

Fortunately, we don't have to rely on Smurf attacks any longer for amplification attacks. Another protocol that can be used is DNS. These attacks have gained some notoriety in recent years. One attack that was very well-known was the attack on Brian Krebs's website Krebs on Security, which provides information security stories written by Krebs, an investigative journalist. The volume of traffic reached 620 gigabits per second at its peak.

A DNS request uses UDP as the transport protocol, and just as in ICMP, there is no verification of the source address performed, which makes spoofing the source address trivial. The attack works because a DNS request is small, whereas the right DNS response can be considerably larger. A common DNS amplification attack has a ratio of 70:1, meaning for every byte sent in the request, the response has 70 bytes. The attack does require a reflector, meaning a DNS server that would perform recursive queries (a resolver rather than an authoritative server) that is open to the Internet. Ideally, you would use multiple resolvers and multiple requesters to get the most volume.

There are a number of tools that can be used to run these denial-of-service attacks, of course. One of them is the Low Orbit Ion Cannon (LOIC). This is a .NET program that can run on Windows systems as well as Linux systems that have the Mono package installed. You can see the interface for LOIC in Figure 12.5. This program can be used to blast requests to both URLs and IP addresses. You could use LOIC to send TCP, UDP, or HTTP requests to the target.

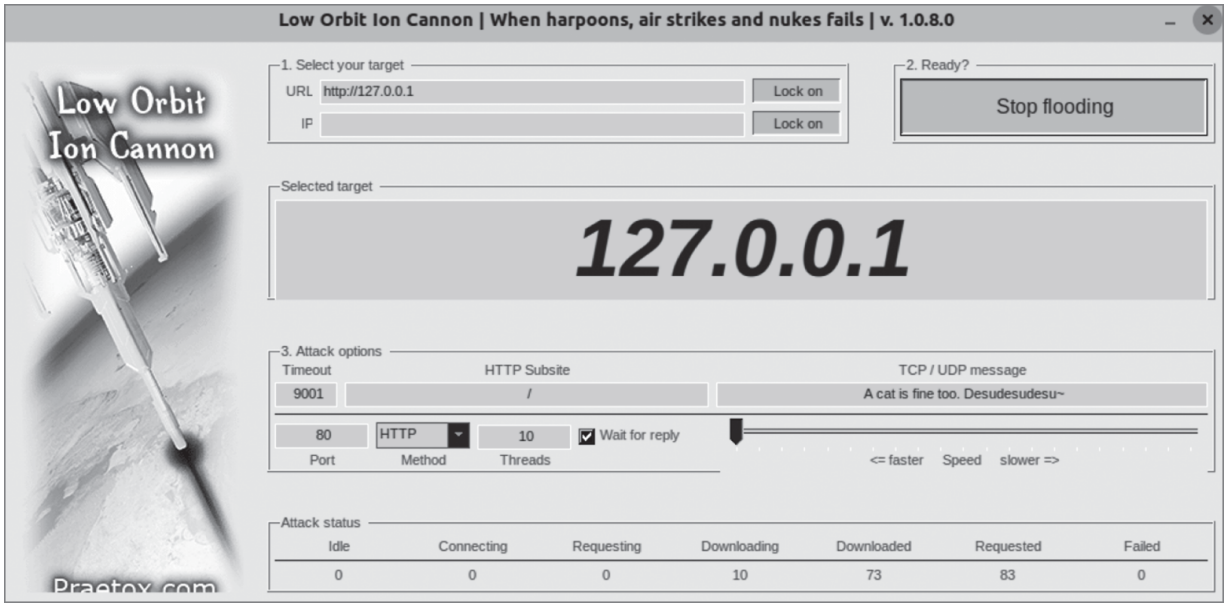


FIGURE 12.5 Low Orbit Ion Cannon

If you have a lot of users participating, a common term for these sorts of attacks is *distributed denial of service*, because the systems involved are often scattered all over the Internet. It's the scattered nature that makes it distributed. As noted earlier, as businesses get more bandwidth, it makes it considerably harder for a single system or site to be able to generate enough traffic to take down sites. Effectively, every modern denial-of-service attack that is bandwidth-oriented is a distributed denial-of-service attack. This is not to say that all denial-of-service attacks are distributed or even bandwidth-oriented.

When it comes to remediation against this sort of attack, there aren't a lot of options. If you are hosting your own services on your own network and you are the target of a denial-of-service attack, the best you can do once you're under attack is contact your Internet service provider (ISP) and hope they will do something to divert the attack before it gets to you. By the time it

gets to your network connection, it's too late. Your connection to your ISP is flooded, which is the source of the denial of service. There's nothing at all your firewall or other device on your premises can do.

Another approach is to use a load balancing service, like that provided by Akamai and other companies. The requests go to their network before being distributed to the target systems, wherever they are. Since Akamai has more than adequate network bandwidth available, it can generally withstand the volume of requests that come in. It has the capability to manage the volume, discarding those that seem to be problematic. Of course, any service that is hosted by a cloud provider like Amazon Web Services, Microsoft's Azure, or Google's Compute can also withstand denial-of-service attacks.

It is worth noting that Krebs on Security was hosted by Akamai at no cost. When the attack on the website happened, Akamai was forced to drop the site to protect its other customers.

Slow Attacks

Volume attacks are not the only way to create a denial of service. You don't need to consume bandwidth or even send a lot of requests quickly. There are some attacks that are slow in nature and essentially look normal. Those slow HTTP attacks are often very effective without needing a lot of effort. An early denial-of-service attack was a SYN flood that filled up the connection buffers on a web server. It didn't take a lot of requests, and it took very little bandwidth to send the requests. This attack is no longer effective, as a general rule. However, the concept remains possible. With the slowloris attack (named after the animal slow loris, a primate found in Southeast Asia), the objective is to send incomplete requests to a web server. When requests are sent, the three-way handshake is complete so the SYN buffer doesn't factor in. Instead, the number of concurrent requests the web server can handle is a factor.

Web servers may be configured with the number of threads that can be run concurrently. This keeps the web server from consuming too many

resources on the system. However, once all of the threads have been used up, no further connections can be accepted. The slowloris attack can be successful against web servers. In fact, in the following code listing, you can see a run of the program `slowhttptest`, which can be used to run the slowloris attack. Within 26 seconds, the program had caused a default installation of Apache to stop responding to requests. This isn't all that impressive, of course, since a default installation of Apache on a single server isn't configured to accept a lot of requests.

Run of `slowhttptest` Against Apache Server

```
slowhttptest version 1.8.2
 - https://code.google.com/p/slowhttptest/ -
test type:                        SLOW HEADERS
number of connections:            50
URL:                              http://192.168.86.44/
verb:                             GET
Content-Length header value:      4096
follow up data max size:          68
interval between follow up data:  10 seconds
connections per seconds:          50
probe connection timeout:         5 seconds
test duration:                    240 seconds
using proxy:                      no proxy

Sat Jan  5 21:28:31 2019:
slow HTTP test status on 25th second:

initializing:        0
pending:             0
connected:           15
error:               0
closed:              35
service available:   YES
Sat Jan  5 21:28:32 2019:
Test ended on 26th second
Exit status: No open connections left
```

The program `slowhttptest` can also be used to run other HTTP attacks, including R-U-Dead-Yet, which is another slow request attack that uses the HTTP body rather than the HTTP headers, as `slowloris` does. The Apache Killer attack can be run by using `slowhttptest`. The program

`slowhttptest` sends requests asking for overlapping ranges of bytes. This causes memory consumption on the server because of a bug in the Apache server program. This is an attack that targets specific versions of Apache, whereas the other attacks are related to the configuration of the server rather than a bug in the software. Finally, `slowhttptest` supports a slow read attack. This is where the program makes a request of a large file from the web server and then reads the file in from the server in small segments. The attacking program can then wait long periods of time between reads. This keeps the connection open for a long time, holding up a connection that might otherwise be used by a legitimate user.

Legacy

There was a time when denial-of-service attacks came from exploiting weak implementations of protocols, such as, for example, the SYN flood mentioned earlier. A SYN flood is where a large number of SYN messages could fill up the connection queue on a system so no legitimate connection requests could get in. This was a weak implementation of TCP in an operating system because half-open connections were utilizing system resources for too long before the system gave up on waiting for the rest of the connection.

A local area network denial (LAND) attack can crash a system. The LAND attack sets the source and destination information of a TCP segment to be the same. This sends the segment into a loop in the operating system, as it is processed as an outbound, then an inbound, and so forth. This loop would lock up the system.

A Fraggle attack is similar to the Smurf attack mentioned earlier. In a Smurf attack, spoofed ICMP messages are sent to a broadcast address. A Fraggle attack uses the same approach, but instead of ICMP, UDP messages are sent. UDP, like ICMP, doesn’t validate the source address and is connectionless, which means it can be spoofed. The attacker sends a UDP request to the broadcast address of a network with the target address set as the source. You may hear a Fraggle attack referred to as UDP flooding.

A teardrop attack is one in which an attacker sends fragmented IP packets to a target, with the fragments set to overlap. When it comes to reassembly

of the packets, the operating system can't handle the overlap. This results in a denial of service at the operating system level.

These attacks, for the most part, don't work any longer because the implementations of the network stack in most operating systems have been fixed to protect against these anomalies. Network stacks today are far more robust than they were 20 years ago when these legacy attacks were popular. This is not to say there are no devices that are still vulnerable to these attacks. This may be especially true of embedded systems with limited computing resources.

Application Exploitation

Application exploits have been common for decades. An application exploit is where an attacker gets control of the execution path of a program. This is commonly done with invalid input being sent into the application and the application doesn't validate the input. This is the same problem discussed in the section on web application attacks. When programmers don't ensure that they are getting what they expect to get, bad things happen. These attacks result not only because of poor input validation within the program, but also because of how programs are placed in memory by the operating system. A couple of common ways to change the flow of execution of a program, called *arbitrary code execution*, are through buffer overflows and also heap-based attacks.

Buffer Overflow

A buffer overflow attack takes advantage of a memory structure called a *stack*. The stack is a section of memory where data is stored while program functions are executing. Every time a function is called, a new stack frame is created on the stack containing the contents of variables that are local to the function. Additionally, and most important for the buffer overflow, the address to return execution to when the function ends is stored on the stack. In Figure 12.6, you can see an example of a single stack frame. The thing about a stack is that once you place something on a stack, you can't get to anything underneath until you take the top off the stack. You're always either adding one to the top of the stack or taking the top item off.

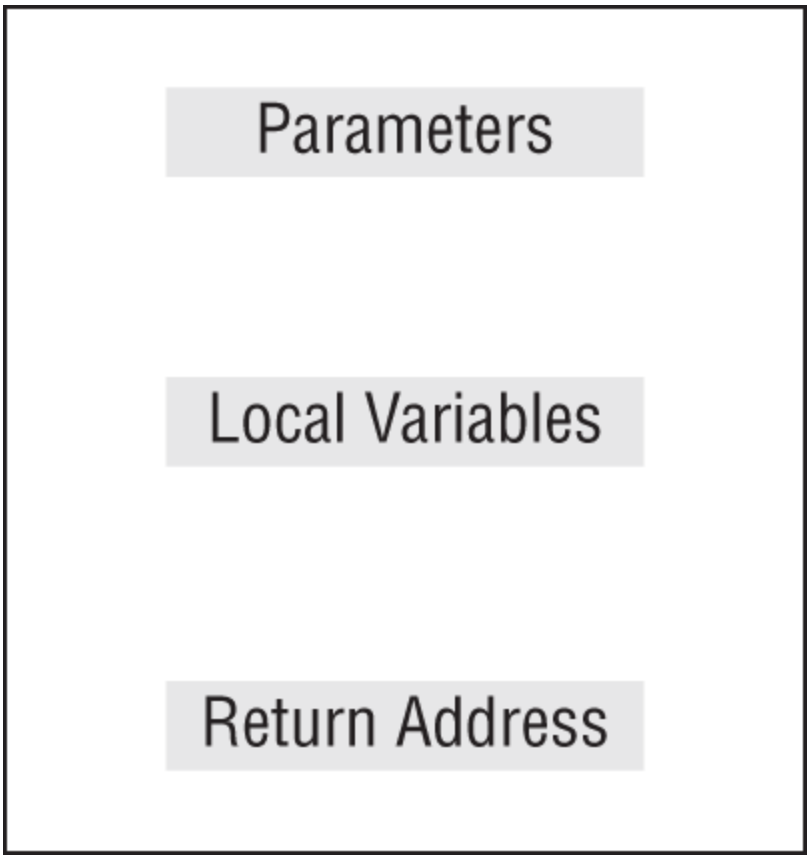


FIGURE 12.6 Stack frame

A buffer overflow works because of the local variables. Each variable is allocated space on the stack. In some cases, data will be copied into the space allocated for it, but if the data being copied into that space, called a *buffer*, is larger than the buffer, the data being copied in will overwrite anything else in the stack. This includes the return address. Figure 12.7 shows what happens if too much data is sent into a buffer. Let's say there is a buffer named `strName` that has been allocated 10 bytes (characters) but the function is copying 16 bytes (characters) into that variable. The 10 bytes (10 *A*s in this case) fill up the 10 bytes allocated for that variable. The remaining 6 bytes write into the next addresses in memory. In this case, the next memory address space is the return address. This gets populated when the function is called and retrieved when the function completes.

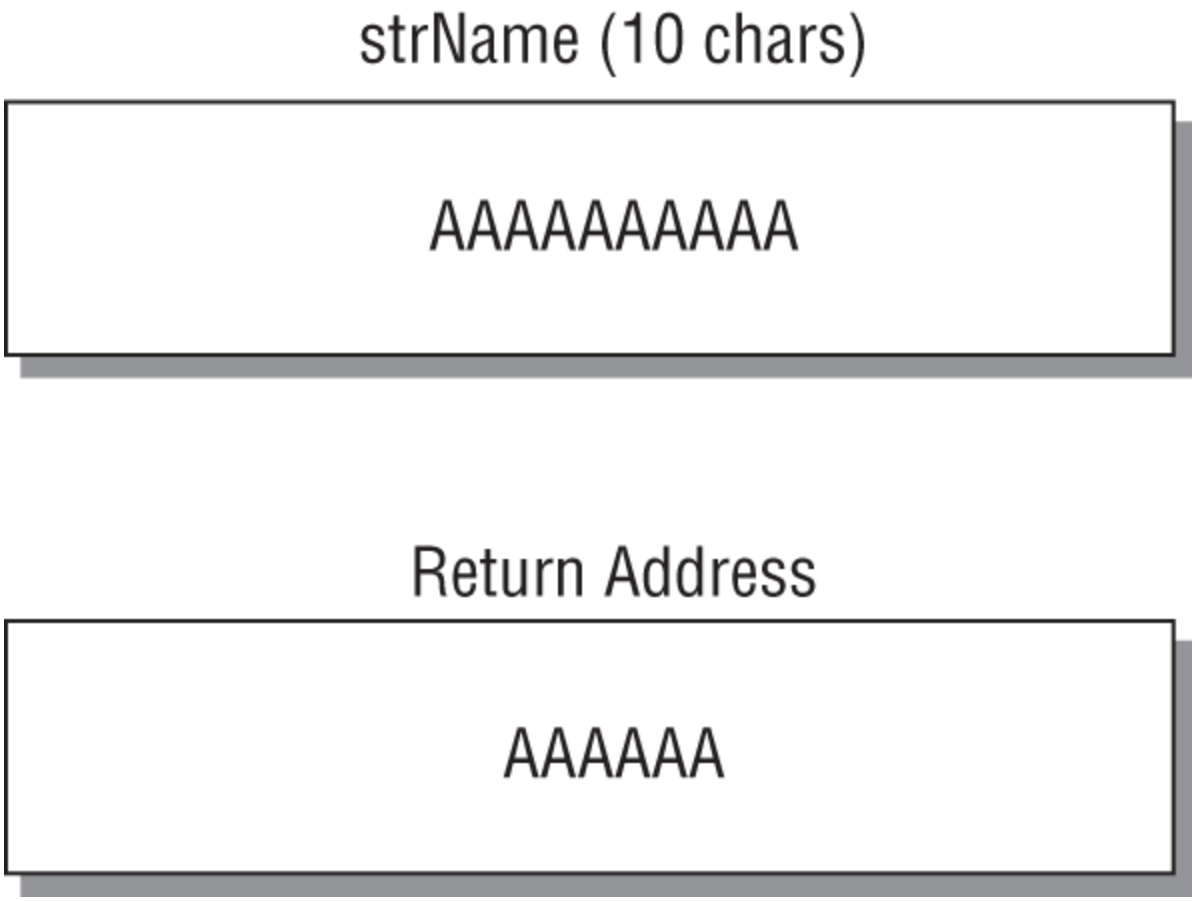


FIGURE 12.7 Buffer overflow

Here, the return address is loaded with *A*s. In reality, the byte would be 0x41, which is the hexadecimal for the ASCII value of A. The address the program would try to jump to would include 0x414141414141. An address like this likely falls outside the address space allocated for the program's execution, which means the program would fail with a segmentation fault. A segmentation fault happens when the program tries to jump to an address outside its memory segment.

This is a simplistic example. In a real computer, everything has to align to bit boundaries. If the operating system is 32 bits, everything would have to align on a 4-byte boundary. This means that even though the variable has been allocated 10 bytes, it would take up 12. In the case of a 64-bit system, everything would have to align on an 8-byte boundary, so while there are 10 bytes allocated, it would take up 16 bytes.

The goal of a buffer overflow is to inject a section of code, called *shellcode*, that the attacker wants to be executed. The place in the stack where the return address is kept needs to point to the space in memory where the shellcode now resides. This means whatever code is there, in assembly language because that's the language the processor understands, will get executed. The attacker may have created a call to a system shell to hand back to them.

It's called shellcode because when these tiny programs were initially created, it was code used to create a shell to get the attacker access to the system.

A way of protecting against this attack is to keep anything on the stack from executing. As the program's code lives in a different memory segment, there's no reason the stack should contain any code to be executed. There are ways around this restriction, though. One is called return to `libc`. The standard C library, which contains all the standard functions that are included in the C language, is often stored in memory to be shared across all programs that were written in C and have been compiled to use a shared library. This library may be stored in a known location in memory, which means the return address can be overwritten with the location of functions from the standard C library that could be used by the attacker. Even if the

location of the library isn't known, it can be determined with some probing, since it's possible to have C functions print out memory locations.

One way to protect against some of this is address space layout randomization (ASLR). Attackers can create exploits because every time a program runs, it appears to get the same memory location. This means the stack is always in the same place. These addresses are compiled into the program, and the operating system allows the program to believe it's in the same place in memory each time it runs. It gets a different physical address each time, but because of virtual memory, the address a program believes it has can be very different from where it really is because the operating system takes care of the translation.

An operating system that supports ASLR will relocate programs, which means the program has to support relative rather than absolute addresses. Every time the program runs, it gets placed into a different address space, meaning the program is told it's in a different part of memory. This makes the job of the attacker harder, but not impossible.

One other way to protect against buffer overflows is through the use of a piece of data called the *stack canary*. This term comes from the practice of miners taking a caged canary into the mine with them. As mines have the potential to have deadly gas in them, the canary was used as an early warning for the miners. Because the canary is more sensitive than humans, if the canary died, the miners knew to get out of the mine because there was gas they perhaps couldn't smell that would kill them if they stayed around. The stack canary performs the same sort of function here. It is a value placed before the return address in the stack. Before the return address is used, the value of the stack canary is checked. If it has been altered in any way, the assumption is the return address has been altered, so it is not used, and the program can fail rather than be compromised.

Heap Spraying

A stack is used for data that is known at compile time. It is data that has been declared in the program's source code. While the program executes, all of this data can be constructed on the stack. Not all data is known ahead of time like this. Some data is known only at runtime. Space needs to be allocated to store this data. Since the stack space in memory is allocated for

data the program knows before it runs, there needs to be memory space for data created and used at runtime. This is a data structure in memory called the *heap*. Memory is allocated during program execution off the heap, and data is placed in those memory allocations. When the data is no longer needed, the memory is freed to be used again if needed.

Because the memory is allocated during runtime, there is no guarantee what memory will be allocated and when, since data needs may change from one running of the program to another, depending on what the program is doing. Also, there are no return addresses stored in the heap to manipulate. This is not to say that there is no way to make use of the heap during an exploit. Heap spraying is one technique that can be used, though it can't be used alone.

One problem with buffer overflows is how to put shellcode into memory and get the return address to point to it reliably. The stack may not have a lot of space to be able to insert code. At a minimum, it really limits what can be done in the shellcode because it has to be small. This is an area the heap can help with. The heap spraying technique makes use of the heap to store the shellcode. The heap is generally in a known location in memory, and it may be possible to predict where the shellcode would be placed if it were in allocated space on the heap.

An attacker would store shellcode into the heap in as many places as possible and then make use of a buffer overflow to put the address of the shellcode into the return address. The program will then jump to the heap and execute the shellcode stored there. A common target for these sorts of attacks is browsers, since browsers regularly have to allocate memory dynamically because the resource utilization varies a lot depending on pages that have been requested and what those pages do. This opens the door to a lot of heap space that can be used to store shellcode because of the amount of dynamic memory being allocated.

Application Protections and Evasions

As noted earlier, one way for the operating system to protect itself and the applications running is to use address space layout randomization. ASLR is a technique that makes it considerably harder for attackers to use buffer overflow attacks. A buffer overflow attack can be successful if the attacker

can control the instruction pointer to get the operating system to execute code provided by the attacker. This means the attacker needs to know where in memory the code they have supplied is. If the address keeps changing every time, it's not predictable. One run of the program may yield one address while a subsequent run will yield an entirely different address.

A way around this, though, is to use an attack technique called return to libc. The C standard library, since a lot of programs continue to be written in C and derivative languages like C++, is a collection of functions that are very commonly used in programs. For example, the following program is written entirely in functions out of the C standard library. One, `printf`, is used to output information, and the other, `scanf`, is used to take information in from the user.

Potentially Vulnerable Application Code

```
#include<stdio.h>

int main(int argc, char **argv) {
         char name[25];
         printf("What is your name? ");
         scanf("%s", name);
         printf("Hello, %s\n", name);

         return 0;
}
```

It would be very memory inefficient to load the C standard library in for every program that was going to use it. Instead, why not use a single copy of the library that is already loaded in memory and just let every program reference it where it lives. This means the C standard library may well be in a fixed place in memory. The correct address for functions is provided to the program at runtime so the addresses instead of function names are used when the program is running. If this is true, the addresses of these function calls can be predicted. This means there is no need for the attacker to provide code any longer. Existing functions from the standard library, such as `execv`, can be used. The function `execv` and its relatives call on the operating system to execute a program that may exist on disk somewhere. Once this is possible, the attacker may be able to simply call a shell outright rather than injecting code to call that shell.

Shellcode

The term *shellcode* is decades old and refers to the fact that most injected attack code is just compiled calls to a shell, which is the command-line interface provided to a user. When you open a terminal window on a Unix-like operating system such as Linux or macOS, you are spawning a shell. When you type a command, it is the shell that receives and interprets it. The entire Windows desktop, managed at least in part by the `Explorer.exe` process, is sometimes referred to as the *Windows shell*.

Another approach to protecting applications is to make the stack nonexecutable. This means any chunk of memory that belongs to the stack segment is flagged by the operating system so any jump to an address in that segment will fail because no executable code can be run from the stack segment. This, again, is a protection against traditional buffer overflow attacks, because the location where the attacker inserts shellcode is in the stack.

While this approach, and the layout randomization, can be used to protect against traditional buffer overflow attacks, it's still possible to use the return to `libc` attack because `libc` has to be executable, just as the remainder of the `.text` segment of a program has to be executable, because that's where the executable code is meant to go.

Lateral Movement

Getting into a system is not the end of the road in the real world. As more organizations get a handle on the modern attack space, many talk about an attack lifecycle that helps to understand what phase an attacker is in. The lifecycle is primarily about knowing how to respond once the attack has been detected. It also helps to plan and prepare. One approach to the lifecycle uses the following phases:

1. **Initial Reconnaissance.** The attacker is performing recon on the target to determine the best methods of attack and what may be available at the target worth gaining access to.
2. **Initial Compromise.** The attacker has gained access in the organization, whether through a phishing attack or an application compromise. They are in the network.
3. **Establish Foothold.** The attacker will have gained access previously, but here they strengthen their position. This may come from installing a means to get back in anytime they want without having to rely on the initial compromise vector. This may also involve establishing a command-and-control mechanism.
4. **Escalate Privileges.** The attacker will start harvesting credentials at this point. They will also try to gain higher-level privileges where they can.
5. **Internal Reconnaissance.** The attacker will get the lay of the land internally and identify other systems that they may be able to compromise.
6. **Move Laterally.** The attacker will compromise other systems in the environment to acquire more systems, credentials, and data.
7. **Maintain Presence.** The attacker will continue to establish the means to gain access to systems in the environment.
8. **Complete Mission.** The attacker will take data out of the environment.

Steps 4 through 7 are a cycle that continues as needed. For every system the attacker compromises, they start the internal reconnaissance all over again to identify other systems that could be compromised. Along the way, for every system compromised, it may be necessary to escalate privileges again.

In the case of a penetration test or ethical hacking engagement, the point of lateral movement is to ensure that the enterprise understands where they may be vulnerable internally. Shoring up defenses on the outside is insufficient, since you may end up with a hard shell but a soft inside. Compromises don’t only come through externally facing services after all. Sometimes they happen much deeper in the business, such as on the

desktop. If the inside of the network is easy to move around in, important resources are still vulnerable.

Lateral movement takes the same sorts of strategies used up to this point: more reconnaissance to identify targets of opportunity, gathering credentials, and making use of various means available to gain remote access to other systems. Moving from Linux system to Linux system may be as simple as just using Secure Shell (SSH) to gain remote access using harvested credentials. On the Windows side, there are other mechanisms that could be used to move from one system to another.

A common approach to moving laterally is to harvest credentials from a victim system. These credentials may be gathered straight out of memory either from the operating system memory space or even from application memory. Tools like Mimikatz are helpful to collect passwords. Once you have credentials, you can use those credentials on other systems within the environment you find yourself in. You can also use techniques like Kerberoasting, discussed in Chapter 7, “System Hacking,” to gain access to other systems that will get you moving laterally within the environment.

Credential Stuffing

As attackers collect usernames and passwords (credentials), they can use a technique called *credential stuffing*. This is effectively an attack where the attacker attempts to make use of known credentials in new systems they are going after. They stuff the known credentials into authentication attempts without knowing if the credentials will work. As usernames and passwords are often reused, credential stuffing attacks can be successful. A couple of techniques that can be used to protect against these attacks is to limit the number of unsuccessful attempts by an attacker. This is often done by username where multiple failed login attempts will result in locking the account. This doesn't address the case where the attacker isn't using the same username and varying passwords but instead varying both username and password. Blocking the IP address in the case of multiple failed logins coming from a single location can be effective in slowing the attacker down. Also, using multifactor authentication everywhere possible can have a lot of success in slowing the attacker down.

In some cases, Windows systems may allow Windows Remote Management (WinRM) to perform administrative functions on other systems. This assumes you have credentials on the remote system, which may be as simple as using domain credentials if you are on a domain controller. If you are further out, of course, you may still be able to use domain credentials if you have managed to gain access to them. If the WinRM service is running, you may also be able to run PowerShell commands to perform administrative tasks. PowerShell has become a powerful language, useful for a lot of tasks.

There is a concept of "living off the land" that attackers use. Years ago, attackers may have had toolkits they would have needed to download. Often, it was compiled software that was used. Today, Linux systems generally have a Python interpreter, and it can be used for several purposes. On the Windows side, PowerShell is available, and each successive release

of PowerShell has introduced a lot of new capabilities. Attackers have realized this and are starting to use PowerShell to run attacks against the local network.

Defense in Depth/Defense in Breadth

For a long time, defense in depth was considered the way to best protect a network and its resources. It's a concept that comes from the military world and is sometimes called the *castle defense*. The point of a defense-in-depth strategy is primarily to delay an attacker to give time for the defenders to rise and boot the attacker out. The network would use a tiered approach with several layers of protections. Figure 12.8 shows a simple network design with the multiple tiers of controls to protect from attack. There are multiple firewalls in place that provide the means to have stricter control of access between network segments. This means that even if an attacker were to get through one line of defense, the next line of defense would be slightly different, and the attacker would have to start over again.

In the case of a defense-in-depth strategy, it's not only the network firewall. On top of the firewall, there are access control lists that could be implemented on routers that separate network segments where there aren't firewalls. This is especially true on the router to the outside world. Fine-grained access controls would not belong on a border router like that. However, there are a number of functions access controls can perform on border routers. This includes dropping all network traffic from so-called Martian packets. This is a packet that has a source address from a space that a packet should not be originating from. This includes all of the private addresses defined in RFC 1918 as well as any other reserved address space.

There are several problems with defense in depth. The first is that it doesn't factor in how the enterprise would know an attack was happening in order to rise to the defense. If an attacker manages to get around the defenses in place and shows up in the midst of the network, how does the operations team know so they can respond? This raises the second point that is problematic in a defense-in-depth approach. It entirely misunderstands modern methods of attack. Today's attackers are more likely to use social engineering attacks like phishing. This means they are coming into the network from the inside as often as not.

Another potential problem with defense in depth is the number of devices in the network, where defensive functions are potentially siloed and where they may not work very well together. A defense-in-depth approach may be such that hosts have their own firewalls, managed by the server teams, while there may be multiple network teams managing their own protections. Each fiefdom may not be in full coordination with others. You end up with the possibility of conflicting controls and certainly not a unified approach to securing the enterprise.

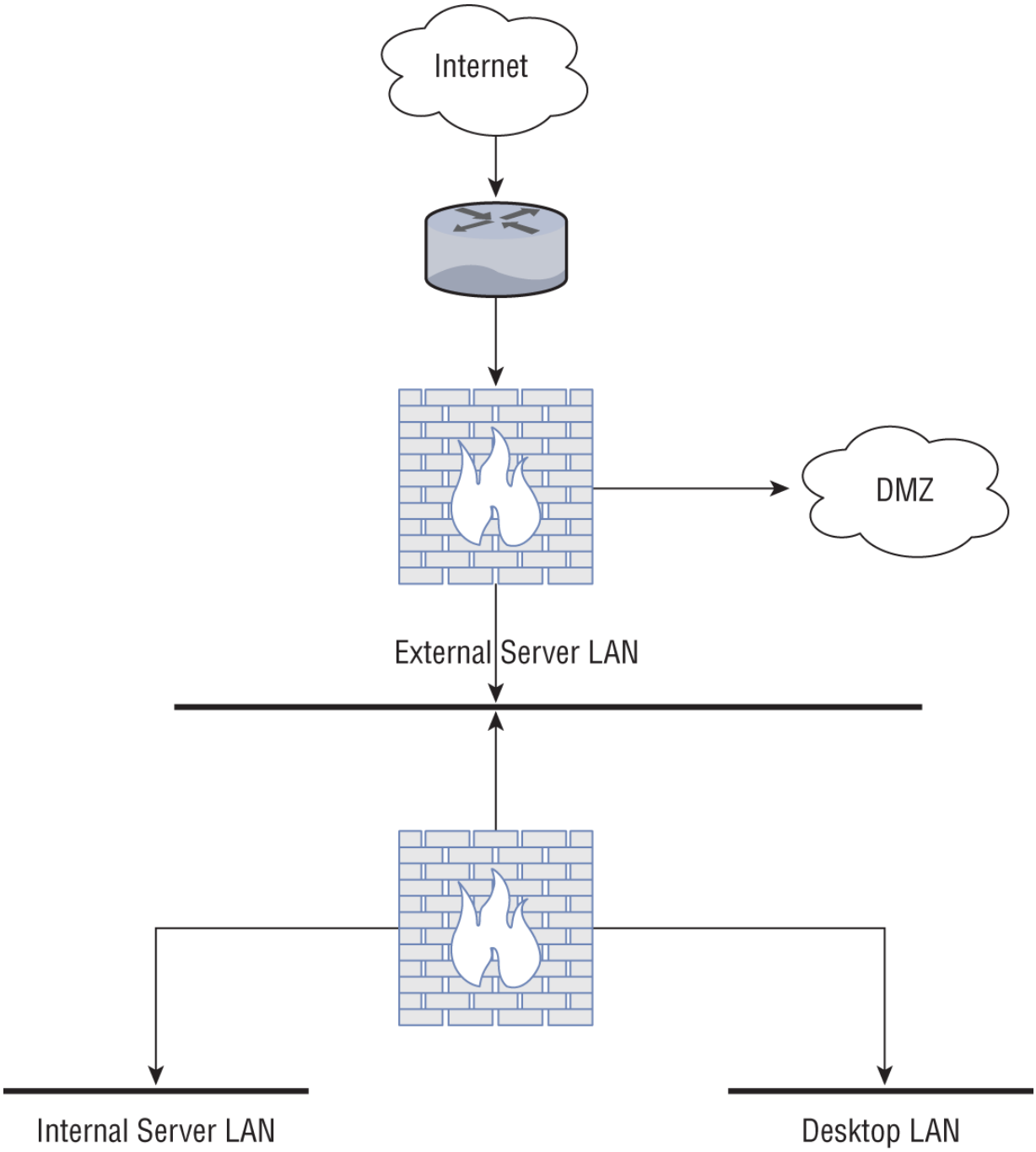


FIGURE 12.8 Defense-in-depth network design

One way to handle this is to use a defense-in-breadth approach. This approach factors in a broader range of attack types, with the understanding that it's not just Network or Transport layer attacks. Attacks are far more likely to happen at the Application layer than they are lower in the stack, where traditional firewalls are more apt to be successful in protecting.

There are next-generation firewalls that cover more of the network stack than the more traditional stateful firewalls. This may include protecting against Application layer attacks by understanding Application layer protocols in order to drop messages that violate those protocols.

A defense-in-breadth strategy should factor in the overall needs of the enterprise without thinking that adding more layers of firewalls increases the hurdles, which may deter the attackers. Modern attackers are organized, funded, and determined. It's best to approach defense of the enterprise with that understanding—a recognition that attackers will not be deterred by a few more hurdles. If there is something in the organization they want, they will keep at it until they get there.

One way of addressing defense in depth and defense in breadth is to introduce a unified threat management (UTM) device. This is a device that may be used on the edge of a network and includes a firewall, intrusion detection, and anti-malware capabilities. This is a next-generation device that recognizes the broad range of attacks that are happening and the vectors that are used.

A common network architecture approach in a defense-in-depth network design is a demilitarized zone (DMZ). A DMZ is used to isolate systems that may be untrustworthy or allow direct access from outside the network. The DMZ can be isolated using firewalls, so even if an attacker managed to get access to systems in the DMZ, they wouldn't be able to get into the enterprise network. As more Internet-facing services are being removed from on-premise and being outsourced to cloud providers, this design has become less useful, though you will continue to find networks that use this approach with systems they don't consider trustworthy.

One system you may find in a DMZ is called a *honeypot*. This is a system that is left out as bait for attackers. The idea with a honeypot is that you can use one to keep attackers occupied, feeding bogus but apparently sensitive information. The feeling is the attacker may get stuck there because they have what they want. This is why these systems are sometimes called *tar pits*. Once you land in one, you are stuck. Another potential benefit to a honeypot is that you get to observe what an attacker does. This can give you insights into how to better tailor your defenses since you know what they are doing. Honeypots can require a lot of care and feeding to be used most

effectively, however. It’s not the kind of system you can drop into place and immediately get a lot of benefit from.

Defensible Network Architecture

Remember when we talked about the attack life cycle? A defensible network architecture factors in the different stages of the attack life cycle and not only creates controls that will protect against the attacker at each stage of the life cycle but, more important, introduces controls that can allow for monitoring of the environment. The monitoring will provide for the ability to alert on anomalous behaviors. When the expectation is that all businesses will be breached at some point, it’s best to create an environment where these breaches can be detected so they can be responded to. This is what is meant by a defensible network architecture—putting in place controls that will allow operations teams to have visibility into the environment, giving them the ability to respond.

The attack life cycle or cyber kill chain can provide guidance for the development of this architecture. At every layer of the environment, there should be visibility and tools in place to respond to any anomaly. This includes developing run books or playbooks for operations teams to follow so that every response to an event is repeatable and based on an understanding of essential assets and risks to the business.

An event is something that happens that is detectable. This could be anything, such as a user logging in or a network connection failing. An incident is usually defined by an organization based on its own requirements, but generally it’s an event that violates a policy. This could be unauthorized use of a system, for instance.

Logging is an essential aspect in developing a defensible network architecture. Network devices can generate NetFlow data, which can be used to store summary information about connections into and out of the network. After an event has happened, having the data necessary to

determine whether the event rises to the point of an incident and requires investigation will be very helpful. Without data like NetFlow to know what systems are talking and who they have been talking to, it will be harder to scope the event and know what systems beyond the initial indicator may be impacted. Without this additional information, there may be an adversary in the network, gathering data.

Of course, logging can also be an expensive activity. At a minimum, beyond just the logs (e.g., web servers, proxy servers, Active Directory servers, firewalls, intrusion detection systems, etc.), you'll need a place to store them and query them. This might be something like Elastic Stack or Splunk, or it may be a full-blown security information and event management (SIEM) solution. A SIEM can help organize and correlate data as well as being able to manage events coming in. This includes generating alerts.

It's not only about logging and visibility, of course. Once an attacker has been identified in the environment, there needs to be the ability to respond. This includes the ability to isolate systems and also provide choke points in the network. Isolating a system isn't just about pulling it off the network. You may need to continue to allow the attacker access to your systems while you try to understand the extent of their infiltration. You will need to keep the attacker from being able to continue to access other systems in the network. Isolation means being able to separate it and its traffic from other parts of the network.

A defensible network architecture takes into account the fact that networks will be compromised. Systems will be compromised. At the moment, we can't ensure that adversaries won't get into the environment, especially as the predominant attack strategy is social engineering. This usually puts the attacker on the inside of the network, and that means the attacker will have access to everything a desktop user has access to. Network designs should take into consideration where the adversary is. The castle defense, or defense in depth, assumes the adversary is on the outside. A defensible network architecture should be able to allow the operations team to protect against anywhere adversaries are found to be.

One important aspect to a defensible network architecture is network segmentation. This doesn't just mean placing systems into virtual local area

network (VLAN) segments, however. This does very little from a security perspective because attacks don't happen at layer 2 if passwords can be easily compromised. All an attacker needs to do is know where the target system is and have some credentials and they will be able to pass from one network segment to another easily. Of course, this is also generally true in the case where networks are actually segmented, meaning every connection from one layer 3 network to another is a firewall or some other device capable of blocking traffic.

Even in cases where the traffic is going to be allowed anyway because it's common application traffic, having these firewalls in place can be beneficial once an attacker has landed in the environment. You can isolate the attacker to a single network segment and prevent them from moving around in the environment if you have some controls in place that allow you to block traffic.

Another way to enable this isolation is to use endpoint detection and response software, which typically has the ability to isolate or contain systems that have been compromised to prevent the spread further into the network. The idea of defensible network architectures is not about keeping the bad guys out but about being able to also monitor and control them once they are in.

Summary

Old-style attacks would have used vulnerabilities in listening services. They may even have used vulnerabilities in the implementation of the network stack. This is not where modern attacks are taking place. As often as not, they happen through social engineering. Attacks often happen at the Application layer, and since web applications are such a common avenue for users to interact with businesses, web applications are a good target for attackers. There are several common attacks against web applications. These attacks can allow the attackers to gain access to data or even to the underlying system. XML External Entity Processing can allow the use of XML to gain access to underlying system functions and files, for instance. SQL injection attacks not only can allow attackers access to the data stored in the database, they can also be used to gain access to the underlying operating system in some cases.

Not all attacks are about the server infrastructure, though. A cross-site scripting attack is about gaining access to something the user has. This may include not only data on the user's system but also data stored on sites the user has access to. Session identification information, stored in cookies, may be used to gain access to other systems where the user has privileges. This may be online retailers or even banking sites. These session tokens could be used to steal from the user.

Web applications can be protected through a number of means when they are developed. First, all input should be validated by each function, even if the expectation is that it comes from a trusted source. Additionally, nothing should be passed directly from the user to any subsystem. Such actions could lead to attacks like command injection.

When it comes to Application layer exploitation, attackers are looking to inject their own code into the memory space belonging to the application. The point is to control the flow of the application by manipulating the instruction pointer, telling the processor where to get its instructions to execute. Buffer overflows can be used to push instructions and return addresses onto the stack, where data for the application is stored. If an attacker can push instructions onto the stack, the program can be manipulated into executing those instructions, sometimes called *arbitrary code execution*. Another attack involves the memory structure called the heap, where dynamic data is stored. Heap spraying involves injecting code from the attacker into the heap. Once it's there, the attacker could cause the program to execute instructions on the heap.

Once attackers are in the environment, they will look to move laterally, to gain access to other systems. This may involve privilege escalation and more reconnaissance so they can gain access to systems where there may be more data.

It has long been a well-respected strategy to use a defense-in-depth approach to network protection. This is sometimes called a castle defense, focused around building a lot of walls to make it more difficult or cumbersome for attackers to gain access to networks. This ignores modern adversaries, who are organized and well-funded and will take as much time as necessary to gain access to a prized target. A few additional hurdles won't be a deterrent, and if there are not detection strategies and controls in

place, slowing the attacker down won't provide much other than a short reprieve from the breach. Defense in breadth is a way to alleviate some of the concerns related to defense in depth by broadening the scope of understanding how modern attackers function. A defense-in-breadth approach would look at the entire network stack, providing controls where possible to protect against attack rather than assuming more firewalls will keep attackers out.

A newer approach to network architecture and design is becoming common. It's called defensible network architecture, and it is based on the understanding that social engineering attacks are common. It also takes into account the importance of visibility and response, since breaches are common and may not be possible to avoid. Logging is essential to detection, and when a lot of logs are collected, a system to manage them is essential, such as a SIEM system. Defensible network architectures recognize that attackers are going to get in through social engineering tactics or other attacks that look like legitimate usage. A defensible network architecture adds on the capability to isolate, contain, and monitor an attacker once the attacker is inside the network.

Review Questions

You can find the answers in the appendix.

What protocol is used for a Smurf attack?

A. DNS

B. ICMP

C. TCP

D. SMTP

If you were to see `' or 1=1;` in a packet capture, what would you expect was happening?

A. Cross-site scripting

B. Command injection

C. SQL injection

D. XML external entity injection

Which protocol is commonly used for amplification attacks?

A. TCP

B. SMTP

C. DNS

D. XML

What is the purpose of a SYN flood?

A. Fill up connection buffers in the operating system

B. Fill up connection buffers in the web server

C. Fill up connection buffers at the Application layer

D. Fill up connection buffers for UDP

How does a slowloris attack work?

A. Holds open connection buffers at the operating system

B. Holds open connection buffers at the web server

C. Holds open connection buffers at the Application layer

D. Holds open connection buffers for UDP

What would be the result of sending the string AAAAAAAAAAAAAAAA into a variable that has been allocated space for 8 bytes?

A. Heap spraying

B. SQL injection

C. Buffer overflow

D. Slowloris attack

What is the target of a cross-site scripting attack?

A. Web server

B. Database server

C. Third-party server

D. User

If you were to see the following in a packet capture, what would you think was happening?

```
<!ENTITY xxe SYSTEM "file:///etc/passwd">]>
```

A. Cross-site scripting

B. SQL injection

C. Command injection

D. XML external entity injection

What protection could be used to prevent an SQL injection attack?

A. Buffer overflows

B. Input validation

C. XML filtering

D. Lateral movement

What security element would be a crucial part of a defense-in-depth network design?

A. Firewall

B. SIEM

C. Web application firewall

D. Log management system

What does a defense-in-breadth approach add?

A. Consideration for a broader range of attacks

B. Protection against SQL injection

C. Buffer overflow protection

D. Heap spraying protection

What attack injects code into dynamically allocated memory?

A. Buffer overflow

B. Cross-site scripting

C. Heap spraying

D. Slowloris

If you were to see the following in a packet capture, what attack would you expect is happening?

```
%3Cscript%3Ealert('wubble');%3C/script%3E
```

A. SQL injection

B. Command injection

C. Cross-site scripting

D. Buffer overflow

What has been done to the following string?

```
%3Cscript%3Ealert('wubble');%3C/script%3E
```

A. Base64 encoding

B. URL encoding

C. Encryption

D. Cryptographic hashing

What technique does a slow read attack use?

A. Small HTTP header requests

B. Small HTTP body requests

C. Small HTTP POST requests

D. Small file retrieval requests

What element could be used to facilitate log collection, aggregation, and correlation?

A. Log manager

B. Firewall

C. IDS

D. SIEM

What is the target of a command injection attack?

A. Operating system

B. Web server

C. Database server

D. User

What would the Low Orbit Ion Cannon be used for?

A. SQL injection attacks

B. Log management

C. Denial-of-service attacks

D. Buffer overflows

What could you use to inform a defensive strategy?

A. SIEM output

B. Attack life cycle

C. Logs

D. Intrusion detection system

What information does a buffer overflow intend to control?

A. Stack pointer

B. Frame pointer

C. Instruction pointer

D. Buffer pointer

Which of these prevention techniques would be best used against a SQL injection attack?

A. Return to `libc`

B. Web application firewall

C. Address space layout randomization

D. Stack canary

If you wanted to get access to a file in the file system on a web server, which of these attack techniques might you use?

A. Cross-site scripting

B. Command injection

C. SQL injection

D. Directory traversal

What are two important characteristics that differentiate defensible network architectures from defense in depth?

A. Firewalls and DMZs

B. Honeypots and DMZs

C. Isolation and malware protection

D. Containment and monitoring

What type of system could you use to trap and monitor an attacker?

A. Web application firewall

B. Next-generation firewall

C. Honeypot

D. DMZ

What attack technique can be used to bypass address space layout randomization?

A. Return to `libc`

B. Stack canary

C. Buffer overflow

D. Return to JavaScript