CHAPTER 16

Attacking Native Compiled Applications

Compiled software that runs in a native execution environment has historically been plagued by vulnerabilities such as buffer overflows and format string bugs. Most web applications are written using languages and platforms that run in a managed execution environment in which these classic vulnerabilities do not arise. One of the most significant advantages of languages such as C# and Java is that programmers do not need to worry about the kind of buffer management and pointer arithmetic problems that have affected software developed in native languages such as C and C++ and that have given rise to the majority of critical bugs found in that software.

Nevertheless, you may occasionally encounter web applications that are written in native code. Also, many applications written primarily using managed code contain portions of native code or call external components that run in an unmanaged context. Unless you know for certain that your target application does not contain any native code, it is worth performing some basic tests designed to uncover any classic vulnerabilities that may exist.

Web applications that run on hardware devices such as printers and switches often contain some native code. Other likely targets include any page or script whose name includes possible indicators of native code, such as `dll` or `exe`, and any functionality known to call legacy external components, such as logging mechanisms. If you believe that the application you are attacking contains substantial amounts of native code, it may be desirable to test every piece of

user-supplied data processed by the application, including the names and values of every parameter, cookie, request header, and other data.

This chapter covers three main categories of classic software vulnerability: buffer overflows, integer vulnerabilities, and format string bugs. In each case, we will describe some common vulnerabilities and then outline the practical steps you can take when probing for these bugs within a web application. This topic is huge and extends far beyond the scope of a book about hacking web applications. To learn more about native software vulnerabilities and how to find them, we recommend the following books:

- *The Shellcoder's Handbook*, 2nd Edition, by Chris Anley, John Heasman, Felix Linder, and Gerardo Richarte (Wiley, 2007)
- *The Art of Software Security Assessment* by Mark Dowd, John McDonald, and Justin Schuh (Addison-Wesley, 2006)
- *Gray Hat Hacking*, 2nd Edition, by Shon Harris, Allen Harper, Chris Eagle, and Jonathan Ness (McGraw-Hill Osborne, 2008)

NOTE **Remote probing for the vulnerabilities described in this chapter carries a high risk of denial of service to the application. Unlike vulnerabilities such as weak authentication and path traversal, the mere detection of classic software vulnerabilities is likely to cause unhandled exceptions within the target application, which may cause it to stop functioning. If you intend to probe a live application for these bugs, you must ensure that the application owner accepts the risks associated with the testing before you begin.**

Buffer Overflow Vulnerabilities

Buffer overflow vulnerabilities occur when an application copies user-controllable data into a memory buffer that is not sufficiently large to accommodate it. The destination buffer is overflowed, resulting in adjacent memory being overwritten with the user's data. Depending on the nature of the vulnerability, an attacker may be able to exploit it to execute arbitrary code on the server or perform other unauthorized actions. Buffer overflow vulnerabilities have been hugely prevalent in native software over the years and have been widely regarded as Public Enemy Number One that developers of such software need to avoid.

Stack Overflows

Buffer overflows typically arise when an application uses an unbounded copy operation (such as `strcpy` in C) to copy a variable-size buffer into a fixed-size buffer without verifying that the fixed-sized buffer is large enough. For example,

the following function copies the `username` string into a fixed-size buffer allocated on the stack:

```
bool CheckLogin(char* username, char* password)
{
    char _username[32];
    strcpy(_username, username);
    ...
```

If the `username` string contains more than 32 characters, the `_username` buffer is overflowed, and the attacker overwrites the data in adjacent memory.

In a stack-based buffer overflow, a successful exploit typically involves overwriting the saved return address on the stack. When the `CheckLogin` function is called, the processor pushes onto the stack the address of the instruction following the call. When `CheckLogin` is finished, the processor pops this address back off the stack and returns execution to that instruction. In the meantime, the `CheckLogin` function allocates the `_username` buffer on the stack right next to the saved return address. If an attacker can overflow the `_username` buffer, he can overwrite the saved return address with a value of his choosing, thereby causing the processor to jump to this address and execute arbitrary code.

Heap Overflows

Heap-based buffer overflows essentially involve the same kind of unsafe operation as described previously, except that the overflowed destination buffer is allocated on the heap, not the stack:

```
bool CheckLogin(char* username, char* password)
{
    char* _username = (char*) malloc(32);
    strcpy(_username, username);
    ...
```

In a heap-based buffer overflow, what is typically adjacent to the destination buffer is not any saved return address but other blocks of heap memory, separated by heap control structures. The heap is implemented as a doubly linked list: each block is preceded in memory by a control structure that contains the size of the block, a pointer to the previous block on the heap, and a pointer to the next block on the heap. When a heap buffer is overflowed, the control structure of an adjacent heap block is overwritten with user-controllable data.

This type of vulnerability is less straightforward to exploit than a stack-based overflow, but a common approach is to write crafted values into the overwritten heap control structure to cause an arbitrary overwrite of a critical pointer at some future time. When the heap block whose control structure has been overwritten is freed from memory, the heap manager needs to update the linked list of

heap blocks. To do this, it needs to update the back link pointer of the following heap block and update the forward link pointer of the preceding heap block so that these two items in the linked list point to each other. To do this, the heap manager uses the values in the overwritten control structure. Specifically, to update the following block's back link pointer, the heap manager dereferences the forward link pointer taken from the overwritten control structure and writes into the structure at this address the value of the back link pointer taken from the overwritten control structure. In other words, it writes a user-controllable value to a user-controllable address. If an attacker has crafted his overflow data appropriately, he can overwrite any pointer in memory with a value of his choosing, with the objective of seizing control of the path of execution and therefore executing arbitrary code. Typical targets for the arbitrary pointer overwrite are the value of a function pointer that the application will later call and the address of an exception handler that will be invoked the next time an exception occurs.

NOTE Modern compilers and operating systems have implemented various defenses to protect software against programming errors that lead to buffer overflows. These defenses mean that real-world overflows today are generally more difficult to exploit than the examples described here. For further information about these defenses and ways to circumvent them, see *The Shellcoder's Handbook*.

"Off-by-One" Vulnerabilities

A specific kind of overflow vulnerability arises when a programming error enables an attacker to write a single byte (or a small number of bytes) beyond the end of an allocated buffer.

Consider the following code, which allocates a buffer on the stack, performs a counted buffer copy operation, and then null-terminates the destination string:

```
bool CheckLogin(char* username, char* password)
{
    char _username[32];
    int i;
    for (i = 0; username[i] && i < 32; i++)
        _username[i] = username[i];
    _username[i] = 0;
    ...
```

The code copies up to 32 bytes and then adds the null terminator. Hence, if the username is 32 bytes or longer, the null byte is written beyond the end of the `_username` buffer, corrupting adjacent memory. This condition may be exploitable. If the adjacent item on the stack is the saved frame pointer of the calling frame, setting the lower-order byte to zero may cause it to point to

the `_username` buffer and therefore to data that the attacker controls. When the calling function returns, this may enable an attacker to take control of the flow of execution.

A similar kind of vulnerability arises when developers overlook the need for string buffers to include room for a null terminator. Consider the following "fix" to the original heap overflow:

```
bool CheckLogin(char* username, char* password)
{
    char* _username = (char*) malloc(32);
    strncpy(_username, username, 32);
    ...
```

Here, the programmer creates a fixed-size buffer on the heap and then performs a counted buffer copy operation, designed to ensure that the buffer is not overflowed. However, if the username is longer than the buffer, the buffer is completely filled with characters from the username, leaving no room to append a trailing null byte. The copied version of the string therefore has lost its null terminator.

Languages such as C have no separate record of a string's length. The end of the string is indicated by a null byte (that is, one with the ASCII character code zero). If a string loses its null terminator, it effectively increases in length and continues as far as the next byte in memory, which happens to be zero. This unintended consequence can often cause unusual behavior and vulnerabilities within an application.

The authors encountered a vulnerability of this kind in a web application running on a hardware device. The application contained a page that accepted arbitrary parameters in a `POST` request and returned an HTML form containing the names and values of those parameters as hidden fields. For example:

```
POST /formRelay.cgi HTTP/1.0
Content-Length: 3

a=b

HTTP/1.1 200 OK
Date: THU, 01 SEP 2011 14:53:13 GMT
Content-Type: text/html
Content-Length: 278

<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=iso-8859-1">
</head>
<form name="FORM_RELAY" action="page.cgi" method="POST">
<input type="hidden" name="a" value="b">
```

```
</form>
<body onLoad="document.FORM_RELAY.submit();">
</body>
</html>
```

For some reason, this page was used throughout the application to process all kinds of user input, much of which was sensitive. However, if 4096 or more bytes of data were submitted, the returned form also contained the parameters submitted by the *previous* request to the page, even if these were submitted by a different user. For example:

```
POST /formRelay.cgi HTTP/1.0
Content-Length: 4096

a=bbbbbbbbbbbbb[lots more b's]

HTTP/1.1 200 OK
Date: THU, 01 SEP 2011 14:58:31 GMT
Content-Type: text/html
Content-Length: 4598

<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=iso-8859-1">
</head>
<form name="FORM_RELAY" action="page.cgi" method="POST">
<input type="hidden" name="a" value="bbbbbbbbbbbbb[lots more b's]">
<input type="hidden" name="strUsername" value="agriffiths">
<input type="hidden" name="strPassword" value="aufwiedersehen">
<input type="hidden" name="Log_in" value="Log+In">
</form>
<body onLoad="document.FORM_RELAY.submit();">
</body>
</html>
```

Having identified this vulnerability, it was possible to poll the vulnerable page continuously with overlong data and parse the responses to log every piece of data submitted to the page by other users. This included login credentials and other sensitive information.

The root cause of the vulnerability was that the user-supplied data was being stored as null-terminated strings within 4096-byte blocks of memory. The data was copied in a checked operation, so no straight overflow was possible. However, if overlong input was submitted, the copy operation resulted in the loss of the null terminator, so the string flowed into the next data in memory. Therefore, when the application parsed the request parameters, it continued up until the next null byte and therefore included the parameters supplied by another user.

Detecting Buffer Overflow Vulnerabilities

The basic methodology for detecting buffer overflow vulnerabilities is to send long strings of data to an identified target and monitor for anomalous results. In some cases, subtle vulnerabilities exist that can be detected only by sending an overlong string of a specific length, or within a small range of lengths. However, in most cases vulnerabilities can be detected simply by sending a string that is longer than the application is expecting.

Programmers commonly create fixed-size buffers using round numbers in either decimal or hexadecimal, such as 32, 100, 1024, 4096, and so on. A simple approach to detecting any "low-hanging fruit" within the application is to send long strings as each item of target data is identified and to monitor the server's responses for anomalies.

HACK STEPS

1. **For each item of data being targeted, submit a range of long strings with lengths somewhat longer than common buffer sizes. For example:**

   ```
   1100
   4200
   33000
   ```

2. **Target one item of data at a time to maximize the coverage of code paths within the application.**
3. **You can use the character blocks payload source in Burp Intruder to automatically generate payloads of various sizes.**
4. **Monitor the application's responses to identify any anomalies. An uncontrolled overflow is almost certain to cause an exception in the application. Detecting when this has occurred in a remote process is difficult, but here are some anomalous events to look for:**
   - **An HTTP 500 status code or error message, where other malformed (but not overlong) input does not have the same effect**
   - **An informative message, indicating that a failure occurred in some native code component**
   - **A partial or malformed response is received from the server**
   - **The TCP connection to the server closes abruptly without returning a response**
   - **The entire web application stops responding**
5. **Note that when a heap-based overflow is triggered, this may result in a crash at some future point, rather than immediately. You may need to experiment to identify one or more test cases that are causing heap corruption.**
6. **An off-by-one vulnerability may not cause a crash, but it may result in anomalous behavior such as unexpected data being returned by the application.**

In some instances, your test cases may be blocked by input validation checks implemented either within the application itself or by other components such as the web server. This often occurs when overlong data is submitted within the URL query string and may be indicated by a generic message such as "URL too long" in response to every test case. In this situation, you should experiment to determine the maximum length of URL permitted (which is often about 2,000 characters) and adjust your buffer sizes so that your test cases comply with this requirement. Overflows may still exist behind the generic length filtering, which can be triggered by strings short enough to get past that filtering.

In other instances, filters may restrict the type of data or range of characters that can be submitted within a particular parameter. For example, an application may validate that a submitted username contains only alphanumeric characters before passing it to a function containing an overflow. To maximize the effectiveness of your testing, you should attempt to ensure that each test case contains only characters that are permitted in the relevant parameter. One effective technique for achieving this is to capture a normal request containing data that the application accepts and to extend each targeted parameter in turn, using the same characters it already contains, to create a long string that is likely to pass any content-based filters.

Even if you are confident that a buffer overflow condition exists, exploiting it remotely to achieve arbitrary code execution is extremely difficult. Peter Winter-Smith of NGSSoftware has produced some interesting research regarding the possibilities for blind buffer overflow exploitation. For more information, see the following whitepaper:

```
www.ngssoftware.com/papers/NISR.BlindExploitation.pdf
```

Integer Vulnerabilities

Integer-related vulnerabilities typically arise when an application performs some arithmetic on a length value before performing some buffer operation but fails to take into account certain features of how compilers and processors handle integers. Two types of integer bugs are worthy of note: overflows and signedness errors.

Integer Overflows

These occur when an operation on an integer value causes it to increase above its maximum possible value or decrease below its minimum possible value. When this occurs, the number wraps, so a very large number becomes very small, or vice versa.

Consider the following "fix" to the heap overflow described previously:

```
bool CheckLogin(char* username, char* password)
{
    unsigned short len = strlen(username) + 1;
    char* _username = (char*) malloc(len);
    strcpy(_username, username);
    ...
```

Here, the application measures the length of the user-submitted username, adds 1 to accommodate the trailing null, allocates a buffer of the resulting size, and then copies the username into it. With normal-sized input, this code behaves as intended. However, if the user submits a username of 65,535 characters, an integer overflow occurs. A short-sized integer contains 16 bits, which is enough for its value to range between 0 and 65,535. When a string of length 65,535 is submitted, the program adds 1 to this, and the value wraps to become 0. A zero-length buffer is allocated, and the long username is copied into it, causing a heap overflow. The attacker has effectively subverted the programmer's attempt to ensure that the destination buffer is large enough.

Signedness Errors

These occur when an application uses both signed and unsigned integers to measure the lengths of buffers and confuses them at some point. Either the application makes a direct comparison between a signed and unsigned value, or it passes a signed value as a parameter to a function that takes an unsigned value. In both cases, the signed value is treated as its unsigned equivalent, meaning that a negative number becomes a large positive number.

Consider the following "fix" to the stack overflow described previously:

```
bool CheckLogin(char* username, int len, char* password)
{
    char _username[32] = "";
    if (len < 32)
        strncpy(_username, username, len);
    ...
```

Here, the function takes both the user-supplied username and a signed integer indicating its length. The programmer creates a fixed-size buffer on the stack and checks whether the length is less than the size of the buffer. If it is, the programmer performs a counted buffer copy, designed to ensure that the buffer is not overflowed.

If the `len` parameter is a positive number, this code behaves as intended. However, if an attacker can cause a negative value to be passed to the function, the programmer's protective check is subverted. The comparison with 32 still

succeeds, because the compiler treats both numbers as signed integers. Hence, the negative length is passed to the `strncpy` function as its count parameter. Because `strncpy` takes an unsigned integer as this parameter, the compiler implicitly casts the value of `len` to this type, so the negative value is treated as a large positive number. If the user-supplied username string is longer than 32 bytes, the buffer is overflowed just as in a standard stack-based overflow.

This kind of attack usually is feasible only when the attacker can directly control a length parameter. For example, perhaps it is computed by client-side JavaScript and submitted with a request alongside the string to which it refers. However, if the integer variable is small enough (for example, a short) and the program computes the length on the server side, an attacker may also be able to introduce a negative value via an integer overflow by submitting an overlong string to the application.

Detecting Integer Vulnerabilities

Naturally, the primary locations to probe for integer vulnerabilities are any instances where an integer value is submitted from the client to the server. This behavior usually arises in two different ways:

- The application may pass integer values in the normal way as parameters within the query string, cookies, or message body. These numbers usually are represented in decimal form using standard ASCII characters. The most likely targets for testing are fields that appear to represent the length of a string that is also being submitted.
- The application may pass integer values embedded within a larger blob of binary data. This data may originate from a client-side component such as an ActiveX control, or it may have been transmitted via the client in a hidden form field or cookie (see Chapter 5). Length-related integers may be harder to identify in this context. They typically are represented in hexadecimal form and often directly precede the string or buffer to which they relate. Note that binary data may be encoded using Base64 or similar schemes for transmission over HTTP.

HACK STEPS

1. **Having identified targets for testing, you need to send suitable payloads designed to trigger any vulnerabilities. For each item of data being targeted, send a series of different values in turn, representing boundary cases for the signed and unsigned versions of different sizes of integer. For example:**
   - **0x7f and 0x80 (127 and 128)**
   - **0xff and 0x100 (255 and 256)**

- 0x7ffff and 0x8000 (32767 and 32768)
- 0xffff and 0x10000 (65535 and 65536)
- 0x7fffffff and 0x80000000 (2147483647 and 2147483648)
- 0xffffffff and 0x0 (4294967295 and 0)

2. When the data being modified is represented in hexadecimal form, you should send little-endian as well as big-endian versions of each test case — for example, ff7f as well as 7fff. If hexadecimal numbers are submitted in ASCII form, you should use the same case that the application itself uses for alphabetical characters to ensure that these are decoded correctly.
3. You should monitor the application's responses for anomalous events in the same way as described for buffer overflow vulnerabilities.

Format String Vulnerabilities

Format string vulnerabilities arise when user-controllable input is passed as the format string parameter to a function that takes format specifiers that may be misused, as in the `printf` family of functions in C. These functions take a variable number of parameters, which may consist of different data types such as numbers and strings. The format string passed to the function contains specifiers, which tell it what kind of data is contained in the variable parameters, and in what format it should be rendered.

For example, the following code outputs a message containing the value of the `count` variable, rendered as a decimal:

```
printf("The value of count is %d", count.);
```

The most dangerous format specifier is `%n`. This does not cause any data to be printed. Rather, it causes the number of bytes output so far to be written to the address of the pointer passed in as the associated variable parameter. For example:

```
int count = 43;
int written = 0;
printf("The value of count is %d%n.\n", count, &written.);
printf("%d bytes were printed.\n", written);
```

outputs the following:

```
The value of count is 43.
24 bytes were printed.
```

If the format string contains more specifiers than the number of variable parameters passed, the function has no way of detecting this, so it simply continues processing parameters from the call stack.

If an attacker controls all or part of the format string passed to a `printf`-style function, he can usually exploit this to overwrite critical parts of process memory and ultimately cause arbitrary code execution. Because the attacker controls the format string, he can control both the number of bytes that the function outputs and the pointer on the stack that gets overwritten with the number of bytes output. This enables him to overwrite a saved return address, or a pointer to an exception handler, and take control of execution in much the same way as in a stack overflow.

Detecting Format String Vulnerabilities

The most reliable way to detect format string bugs in a remote application is to submit data containing various format specifiers and monitor for any anomalies in the application's behavior. As with uncontrolled triggering of buffer overflow vulnerabilities, it is likely that probing for format string flaws will result in a crash within a vulnerable application.

HACK STEPS

1. **Targeting each parameter in turn, submit strings containing large numbers of the format specifiers `%n` and `%s`:**

```
%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n
%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s
```

   Note that some format string operations may ignore the `%n` specifier for security reasons. Supplying the `%s` specifier instead causes the function to dereference each parameter on the stack, probably resulting in an access violation if the application is vulnerable.

2. **The Windows `FormatMessage` function uses specifiers in a different way than the `printf` family. To test for vulnerable calls to this function, you should use the following strings:**

```
%1!n!%2!n!%3!n!%4!n!%5!n!%6!n!%7!n!%8!n!%9!n!%10!n! etc...
%1!s!%2!s!%3!s!%4!s!%5!s!%6!s!%7!s!%8!s!%9!s!%10!s! etc...
```

3. **Remember to URL-encode the `%` character as `%25`.**
4. **You should monitor the application's responses for anomalous events in the same way as described for buffer overflow vulnerabilities.**

Summary

Software vulnerabilities in native code represent a relatively niche area in relation to attacks on web applications. Most applications run in a managed execution environment in which the classic software flaws described in this chapter do not arise. However, occasionally these kinds of vulnerabilities are highly relevant and have been found to affect many web applications running on hardware devices and other unmanaged environments. A large proportion of such vulnerabilities can be detected by submitting a specific set of test cases to the server and monitoring its behavior.

Some vulnerabilities in native applications are relatively easy to exploit, such as the off-by-one vulnerability described in this chapter. However, in most cases, they are difficult to exploit given only remote access to the vulnerable application.

In contrast to most other types of web application vulnerabilities, even the act of probing for classic software flaws is quite likely to cause a denial-of-service condition if the application is vulnerable. Before performing any such testing, you should ensure that the application owner accepts the inherent risks involved.

Questions

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

1. Unless any special defenses are in place, why are stack-based buffer overflows generally easier to exploit than heap-based overflows?
2. In the C and C++ languages, how is a string's length determined?
3. Why would a buffer overflow vulnerability in an off-the-shelf network device normally have a much higher likelihood of exploitation than an overflow in a proprietary web application running on the Internet?
4. Why would the following fuzz string fail to identify many instances of format string vulnerabilities?

   ```
   %n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n...
   ```

5. You are probing for buffer overflow vulnerabilities in a web application that makes extensive use of native code components. You find a request that may contain a vulnerability in one of its parameters; however, the anomalous behavior you have observed is difficult to reproduce reliably. Sometimes submitting a long value causes an immediate crash. Sometimes you need to submit it several times in succession to cause a crash. And sometimes a crash occurs after a large number of benign requests.

   What is the most likely cause of the application's behavior?