Payload Encryption - RC4

Introduction

RC4 is a fast and efficient stream cipher that is also a bidirectional encryption algorithm that allows the same function to be used for both encryption and decryption. There are several C implementations of RC4 publicly available but this module will demonstrate three ways of performing RC4 encryption.

Note that diving into how the RC4 algorithm works is not the goal of this module and it's not required to fully understand it in depth. Rather the goal is encrypting the payload to evade detection.

RC4 Encryption - Method 1

This method uses the RC4 implementation found here due to its stability and well-written code. There are two functions `rc4Init` and `rc4Cipher` which are used to initialize a `rc4context` structure and perform the RC4 encryption, respectively.

```
typedef struct
{
    unsigned int i;
    unsigned int j;
    unsigned char s[256];

} Rc4Context;


void rc4Init(Rc4Context* context, const unsigned char* key, size_t
length)
{
    unsigned int i;
    unsigned int j;
    unsigned char temp;

    // Check parameters
    if (context == NULL || key == NULL)
        return ERROR_INVALID_PARAMETER;
```

```
    // Clear context
    context->i = 0;
    context->j = 0;

    // Initialize the S array with identity permutation
    for (i = 0; i < 256; i++)
    {
        context->s[i] = i;
    }

    // S is then processed for 256 iterations
    for (i = 0, j = 0; i < 256; i++)
    {
        //Randomize the permutations using the supplied key
        j = (j + context->s[i] + key[i % length]) % 256;

        //Swap the values of S[i] and S[j]
        temp = context->s[i];
        context->s[i] = context->s[j];
        context->s[j] = temp;
    }

}


void rc4Cipher(Rc4Context* context, const unsigned char* input,
unsigned char* output, size_t length){
    unsigned char temp;

    // Restore context
    unsigned int i = context->i;
    unsigned int j = context->j;
    unsigned char* s = context->s;

    // Encryption loop
    while (length > 0)
    {
        // Adjust indices
        i = (i + 1) % 256;
        j = (j + s[i]) % 256;

        // Swap the values of S[i] and S[j]
        temp = s[i];
```

```
        s[i] = s[j];
        s[j] = temp;

        // Valid input and output?
        if (input != NULL && output != NULL)
        {
            //XOR the input data with the RC4 stream
            *output = *input ^ s[(s[i] + s[j]) % 256];

            //Increment data pointers
            input++;
            output++;
        }

        // Remaining bytes to process
        length--;
    }

    // Save context
    context->i = i;
    context->j = j;
}
```

RC4 Encryption

The code below shows how the `rc4Init` and `rc4Cipher` functions are used to encrypt a payload.

```
    // Initialization
    Rc4Context ctx = { 0 };

    // Key used for encryption
    unsigned char* key = "maldev123";
    rc4Init(&ctx, key, sizeof(key));

    // Encryption //
    // plaintext - The payload to be encrypted
    // ciphertext - A buffer that is used to store the outputted
encrypted data
    rc4Cipher(&ctx, plaintext, ciphertext, sizeof(plaintext));
```

RC4 Decryption

The code below shows how the `rc4Init` and `rc4Cipher` functions are used to decrypt a payload.

```
    // Initialization
    Rc4Context ctx = { 0 };

    // Key used to decrypt
    unsigned char* key = "maldev123";
    rc4Init(&ctx, key, sizeof(key));

    // Decryption //
    // ciphertext - Encrypted payload to be decrypted
    // plaintext - A buffer that is used to store the outputted
plaintext data
    rc4Cipher(&ctx, ciphertext, plaintext, sizeof(ciphertext));
```

RC4 Encryption - Method 2

The undocumented Windows NTAPI `SystemFunction032` offers a faster and smaller implementation of the RC4 algorithm. Additional information about this API can be found on this Wine API page.

SystemFunction032

The documentation page states that the function `SystemFunction032` accepts two parameters of type `USTRING`.

```
NTSTATUS SystemFunction032
(
 struct ustring*       data,
 const struct ustring* key
)
```

USTRING Structure

Unfortunately, since this is an undocumented API the structure of `USTRING` is unknown. But through additional research, it's possible to locate the `USTRING` structure definition in wine/crypt.h. The structure is shown below.

```
typedef struct
{
    DWORD   Length;         // Size of the data to encrypt/decrypt
    DWORD   MaximumLength;  // Max size of the data to
encrypt/decrypt, although often its the same as Length
(USTRING.Length = USTRING.MaximumLength = X)
    PVOID   Buffer;         // The base address of the data to
encrypt/decrypt

} USTRING;
```

Now that the `USTRING` struct is known, the `SystemFunction032` function can be used.

Retrieving SystemFunction032's Address

To use `SystemFunction032`, its address must first be retrieved. Since `SystemFunction032` is exported from `advapi32.dll`, the DLL must be loaded into the process using `LoadLibrary`. The return value of the function call can be used directly in `GetProcAddress`.

Once the address of `SystemFunction032` has been successfully retrieved, it should be type-casted to a function pointer matching the definition found on the previously referenced Wine API page. However, the returned address can be casted directly from `GetProcAddress`. This is all demonstrated in the snippet below.

```
fnSystemFunction032 SystemFunction032 = (fnSystemFunction032)
GetProcAddress(LoadLibraryA("Advapi32"), "SystemFunction032");
```

The function pointer of `SystemFunction032` is defined as the `fnSystemFunction032` data type which is shown below.

```
typedef NTSTATUS(NTAPI* fnSystemFunction032)(
    struct USTRING* Data,   // Structure of type USTRING that holds
information about the buffer to encrypt / decrypt
    struct USTRING* Key     // Structure of type USTRING that holds
information about the key used while encryption / decryption
);
```

SystemFunction032 Usage

The snippet below provides a working code sample that utilizes the `SystemFunction032` function to perform RC4 encryption and decryption.

```
typedef struct
{
    DWORD   Length;
    DWORD   MaximumLength;
    PVOID   Buffer;

} USTRING;

typedef NTSTATUS(NTAPI* fnSystemFunction032)(
    struct USTRING* Data,
    struct USTRING* Key
);

/*
Helper function that calls SystemFunction032
* pRc4Key - The RC4 key use to encrypt/decrypt
* pPayloadData - The base address of the buffer to encrypt/decrypt
* dwRc4KeySize - Size of pRc4key (Param 1)
* sPayloadSize - Size of pPayloadData (Param 2)
*/
BOOL Rc4EncryptionViaSystemFunc032(IN PBYTE pRc4Key, IN PBYTE
pPayloadData, IN DWORD dwRc4KeySize, IN DWORD sPayloadSize) {

    NTSTATUS STATUS  = NULL;

    USTRING Data = {
        .Buffer         = pPayloadData,
        .Length         = sPayloadSize,
        .MaximumLength  = sPayloadSize
    };

    USTRING Key = {
        .Buffer         = pRc4Key,
        .Length         = dwRc4KeySize,
        .MaximumLength  = dwRc4KeySize
    },

    fnSystemFunction032 SystemFunction032 =
(fnSystemFunction032)GetProcAddress(LoadLibraryA("Advapi32"),
"SystemFunction032");
```

```
    if ((STATUS = SystemFunction032(&Data, &Key)) != 0x0) {
        printf("[!] SystemFunction032 FAILED With Error: 0x%0.8X
\n", STATUS);
        return FALSE;
    }

    return TRUE;
}
```

RC4 Encryption - Method 3

Another way to implement the RC4 algorithm is using the `SystemFunction033` which takes the same parameters as the previously shown `SystemFunction032` function.

```
typedef struct
{
    DWORD   Length;
    DWORD   MaximumLength;
    PVOID   Buffer;

} USTRING;


typedef NTSTATUS(NTAPI* fnSystemFunction033)(
    struct USTRING* Data,
    struct USTRING* Key
    );


/*
Helper function that calls SystemFunction033
* pRc4Key - The RC4 key use to encrypt/decrypt
* pPayloadData - The base address of the buffer to encrypt/decrypt
* dwRc4KeySize - Size of pRc4key (Param 1)
* sPayloadSize - Size of pPayloadData (Param 2)
*/
BOOL Rc4EncryptionViSystemFunc033(IN PBYTE pRc4Key, IN PBYTE
pPayloadData, IN DWORD dwRc4KeySize, IN DWORD sPayloadSize) {

    NTSTATUS    STATUS = NULL;
```

```
    USTRING     Key = {
            .Buffer        = pRc4Key,
            .Length        = dwRc4KeySize,
            .MaximumLength = dwRc4KeySize
    };

    USTRING     Data = {
            .Buffer         = pPayloadData,
            .Length         = sPayloadSize,
            .MaximumLength  = sPayloadSize
    };

    fnSystemFunction033 SystemFunction033 =
(fnSystemFunction033)GetProcAddress(LoadLibraryA("Advapi32"),
"SystemFunction033");

    if ((STATUS = SystemFunction033(&Data, &Key)) != 0x0) {
        printf("[!] SystemFunction033 FAILED With Error: 0x%0.8X
\n", STATUS);
        return FALSE;
    }

    return TRUE;
}
```

Encryption/Decryption Key Format

The code snippets in this module and other encryption modules use one valid way of representing the encryption/decryption key. However, it's important to be aware that the key can be represented using several different ways.

Be aware that hardcoding the plaintext key into the binary is considered bad practice and can be easily pulled when the malware is analyzed. Future modules will provide solutions to ensure the key cannot be easily retrieved.

```
// Method 1
unsigned char* key = "maldev123";

// Method 2
// This is 'maldev123' represented as an array of hexadecimal bytes
unsigned char key[] = {
```

```
    0x6D, 0x61, 0x6C, 0x64, 0x65, 0x76, 0x31, 0x32, 0x33
};

// Method 3
// This is 'maldev123' represented in a hex/string form (hexadecimal
escape sequence)
unsigned char* key = "\x6D\x61\x64\x65\x76\x31\x32\x33";

// Method 4 - better approach (via stack strings)
// This is 'maldev123' represented in an array of chars
unsigned char key[] = {
    'm', 'a', 'l', 'd', 'e', 'v', '1', '2', '3'
```