710.2

Encryption in Malware

All reference links are operational in the browser-based delivery of the electronic workbook.

FOR710.2 | Reverse-Engineering Malware: Advanced Code Analysis

Encryption in Malware

Section FOR710.2, also known as Section 2 of the FOR710. This section discusses the use of encryption in malware.

FOR710.2 materials are created and maintained by Anuj Soni. To learn about Anuj's background and expertise, please see https://www.sans.org/instructors/anuj-soni. You can visit his blog at https://malwology.com/ and follow him on Twitter at https://twitter.com/asoni.

Malware Commonly Uses Encryption to Protect Code and Data

- Malware uses encryption to:
  - Encrypt files.
  - Protect keys.
  - Obfuscate configuration data.
  - Hide sensitive strings.
  - Protect C2 communications.
- Proficient reverse engineers must be prepared to identify the use of encryption, its purpose, and specific algorithms.

Malware commonly uses encryption to obfuscate code and data. For example, ransomware uses encryption to prevent access to file content and protect the keys required to decrypt files. Malware also uses encryption to obfuscate strings, configuration data, and content exchanged via command-and-control.

Proficient reverse engineers must be prepared to identify the use of encryption and its purpose. This requires an understanding of basic cryptography principals and familiarity with specific crypto algorithms, which will be discussed in this section.

Course Roadmap

- FOR710.1: Code Deobfuscation and Execution
- FOR710.2: Encryption in Malware
- FOR710.3: Automating Malware Analysis
- FOR710.4: Correlating Malware and Building Rules

SECTION 2

- ***Encryption Essentials***
  - Lab 2.1: Encryption Essentials: Quiz
- File Encryption and Key Protection
  - Lab 2.2: Identifying File Encryption and Key Protection in Ransomware
- Data Encryption in Malware
  - Lab 2.3: Analyzing Data Encryption in Malware

This page intentionally left blank.

Encryption Essentials

This page intentionally left blank.

Key Terms for Our Discussion

- *Plaintext* is unencrypted data and *ciphertext* is encrypted data.
- An *algorithm* takes plaintext as input and performs *rounds (*i.e., iterations) of operations to produce ciphertext.
- A *key* is combined with an algorithm to scramble plaintext into ciphertext.
- A *nonce*:
  - A “number used once”, and it is used by algorithms to add variation so that a message encrypted with the same key does not produce the same ciphertext.
  - It is not necessarily random, but it should only be used once.
- An *Initialization Vector* is like a nonce, but it must be random.

Before we discuss specific approaches to encryption, we must understand some key terms.

Plaintext refers to unencrypted data. *Ciphertext* refers to the encrypted plaintext.

The *algorithm* performs mathematical operations and, with the help of a key, scrambles the original content into ciphertext. The algorithm should be robust such that an attacker cannot reverse the process and obtain the plaintext without the key. The algorithm is assumed to be public.

One way to think of rounds is to consider the metaphor of repainting a car. Multiple rounds of operations are like applying multiple coats of paint to hide the original color. The first coat might not be enough to hide the original, and with each round it becomes harder and harder to see the original layer.

A *key* is data (usually random) that is combined with an algorithm to convert plaintext into encrypted content. The key should be kept secret.

A nonce is a “number used once”. As the name implies, it should not be reused (i.e., it should be unique). While it does not need to be random, it often is. Its purpose is to add some variation to the encryption process so that an identical message encrypted with the same algorithm and same key does not produce the same ciphertext. This prevents a category of attacks known as “replay attacks”, where an attacker could reuse encryption communications or extract some meaning from it. The nonce does not need to be a secret, but as mentioned earlier, it cannot be reused.

An Initialization Vector (IV) is a random value that provided as input to an encryption algorithm with the key. It is used to provide additional randomness to the encryption process, like the goal of a nonce. The IV must be random, but it does not need to be a secret.

Symmetric-Key Encryption Algorithms Use the Same Key to Encrypt and Decrypt Data

- The key is referred to as a "shared secret".
- Examples you may counter during malware analysis include:
  - AES (Rijndael).
  - Salsa20.
  - ChaCha20.
  - RC4.
- Symmetric encryption algorithms are relatively fast and suitable for encrypting large volumes of data.
- In ransomware, symmetric encryption algorithms are used to encrypt files.

The main disadvantage of symmetric-key algorithms is the safe distribution of a shared key.

Symmetric encryption is relatively straightforward, but it requires that two communicating parties have the same key. Sharing this key in a secure manner can be inconvenient and difficult to accomplish. If an attacker were to intercept the exchange of keys, they would be able to decipher messages between the two parties.

Asymmetric solutions provide one solution to this problem—we will discuss this topic soon.

AES Specifies a Symmetric-Key Algorithm and Block Cipher Commonly Used in Ransomware to Encrypt Files

- AES is a 128-bit symmetric block cipher, so 128 bits in = 128 bits out.
- It accommodates key lengths of 128, 192, and 256.
- AES is an SP-network, which means it uses **S**ubstitution and **P**ermutation.
  - Substitution: Replaces bytes with other bytes using a lookup table (i.e., S-box).
  - Permutation: Rearranges bytes.
- AES uses a *key scheduling algorithm,* which expands one key into sub-keys ("round keys") that are applied to individual rounds of operations.

The AES substitution box (S-box) is public, so we can use it to identify AES code embedded in malware.

The Advanced Encryption Standard (AES) is an encryption specification established by the U.S. National Institute of Standards and Technology (NIST). It specifies the use of the Rijndael block cipher.

AES uses a symmetric-key algorithm to encrypt blocks 128 bits and it can use three key lengths: 128, 192, and 256 bits. Note that the 128-bit key length is sufficient for most applications, but the other key lengths are provided for added security. Note that regardless of the key length, the block size is 128 bits—this means 128 bits of plaintext input results in 128 bits of ciphertext output.

AES is a SP-Network, which means the block cipher involves both substitution and permutation. Substitution refers to replacing bytes of plaintext with other bytes (based on a lookup table) as one approach to obfuscate the original bytes. Permutation refers to the process of rearranging, shuffling, or mixing up bytes as an additional scrambling technique.

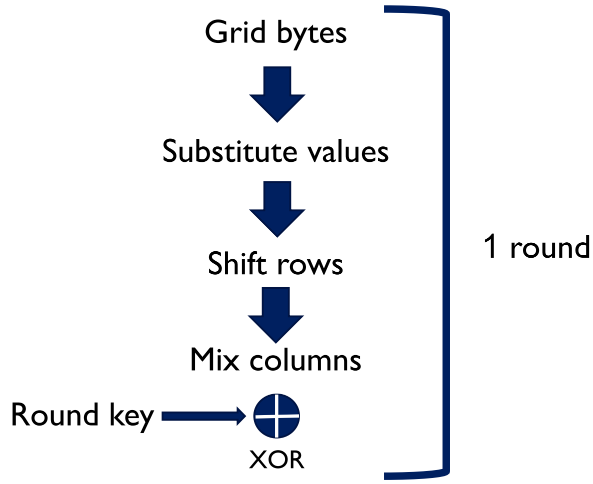
When considering substitution, an S-box (substitution-box) takes some bits as inputs and performs substitutions to output some other bits. The S-box determines the rules for replacing bytes, and it essentially consults a lookup table. Note that the number of bits input may not match the number of bits output. You can see the public S-box page on Wikipedia at https://for710.com/aessbox.

AES uses a key scheduling algorithm. A key scheduling algorithm takes a single input key and produces sub-keys (called "round keys") that are applied to the individual rounds of operations. The process of generating multiple sub-keys from a single initial key is called "key expansion".

AES Maps 128 Bits of Input Data to a 4-Byte by 4-Byte Grid and Performs Rounds of Operations

b_0	b_4	b_8	b_{12}
b_1	b_5	b_9	b_{13}
b_2	b_6	b_{10}	b_{14}
b_3	b_7	b_{11}	b_{15}

b = byte



When operating on 128 bits of data, AES maps the bytes to a 4-byte by 4-byte grid and then performs various operations against the values in the grid.

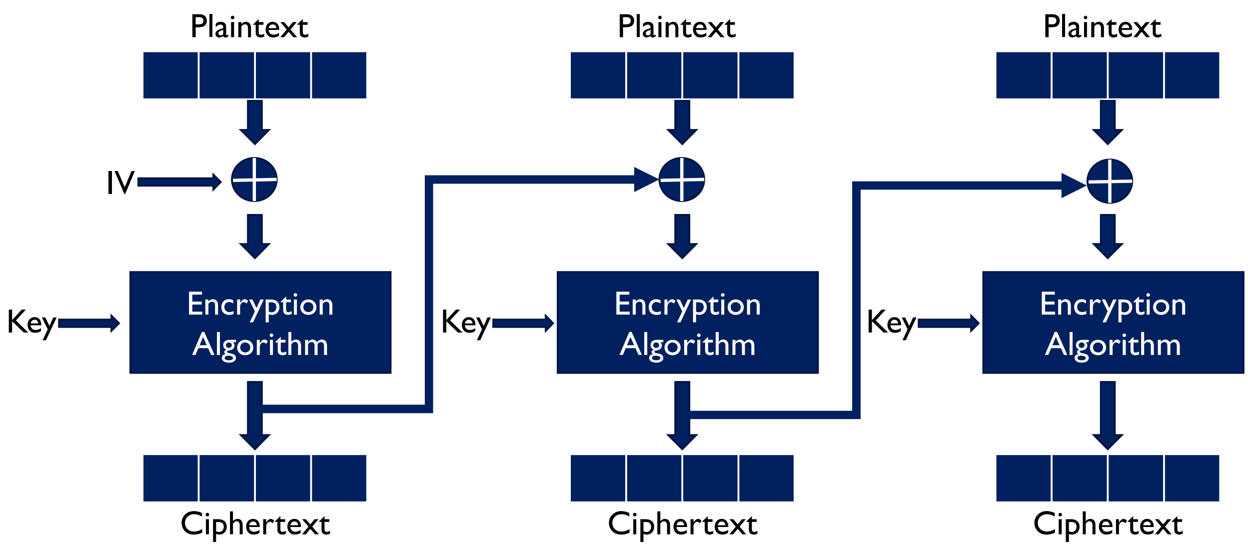
AES takes a 128-bit block of data and feeds it through multiple rounds of operations, where each round consists of substituting bytes (using an S-box) and permutations including shifting rows and mixing columns. Importantly, each round includes XORing each byte of data with a byte of the round key (as a derived from the encryption key).

Depending upon the size of the key, different rounds of operations may be applied. For a 128-bit key, AES uses 10 rounds. For a 192-bit key, AES uses 12 rounds. Finally, for a 256-bit key, AES uses 14 rounds.

For an excellent YouTube video explaining AES, see https://for710.com/ytaes.

Reference:
https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development#nist-reports-aes

Cipher Block Chaining (CBC) Associates One Block and the Next and Uses an Initialization Vector (IV)



Cipher Block Chaining (CBC) interconnects each encrypted block with the next. As shown on the slide, the ciphertext from block one is XORed with the plaintext for block two before the second block is encrypted. Since the first block does not have a block before it, a random Initialization Vector (IV) is used.

The relationship between each block with other blocks and the initialization vector adds additional security when compared to ECB. For example, two plaintext blocks encrypted with the same key algorithm will produce different ciphertext blocks. However, there are drawbacks to this mode of operation. The interconnectedness between blocks means that if an attacker modifies one block, future blocks will be corrupted, and decryption will not be successful. From a performance perspective there are drawbacks as well. For example, decrypting a block of ciphertext requires decrypting all prior blocks of data.

CBC mode is an improvement on ECB, but its weaknesses are considered significant.

Salsa20 and Chacha20 Are Symmetric Stream Ciphers That Perform 20 Rounds of Operations to Encrypt Data

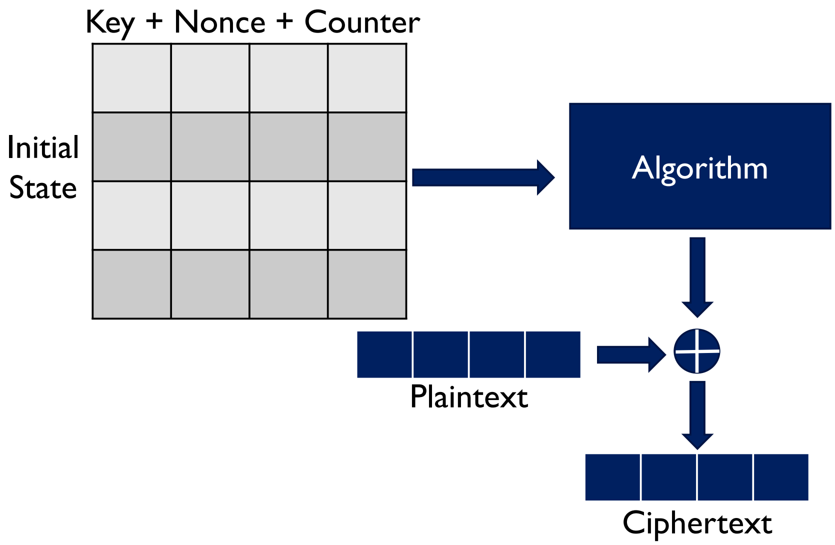
- ChaCha was published only a year after Salsa as an improvement.
- Both algorithms consist of ARX operations (ADD, Rotate, and XOR).
- Inputs to each algorithm include a 256-bit key, a 64-bit nonce, and a 64-bit counter or block number.
- Like AES CTR mode, the inputs are provided to the algorithm, and the output is XORed with the plaintext.
- Malware may use a different number of rounds. For example, if only 12 rounds are used within Salsa, the algorithm is referred to as Salsa12.

ChaCha20 is a symmetric stream cipher created by Daniel J. Bernstein and published in 2008. It is based on the Salsa20 algorithm that he published in 2007. ChaCha20 is considered an improvement upon Salsa20 that adds "diffusion", which means it makes it harder for the attacker to correlate ciphertext with the corresponding plaintext. ChaCha20 is generally faster than AES.

ChaCha20 and Salsa20 algorithms both consist of ARX (ADD, Rotate, and XOR instructions) operations.

As inputs, the algorithms take a 256-bit key, a 64-bit nonce, and a 64-bit counter (this is similar to the key inputs to AES in CTR mode). This produces a 512-bit key stream that is XORed with the plaintext.

ChaCha and Salsa Provide a 4 X 4 Matrix of 32-Bit Values to the Algorithm; This Is Known as the "Initial State"



The 4 x 4 matrix consists of a Key + Nonce + Counter. Each square within the 4 x 4 matrix is 32 bits in size.

The algorithm performs various operations to jumble the initial state and provide a suitable stream of bits to XOR with the plaintext.

ChaCha and Salsa Differ in Their Initial States

ChaCha Initial State

"expa"	"nd 3"	"2-by"	"te k"
Key	Key	Key	Key
Key	Key	Key	Key
Counter	Counter /Nonce	Nonce	Nonce

Salsa Initial State

"expa"	Key	Key	Key
Key	"nd 3"	Nonce	Nonce
Counter	Counter	"2-by"	Key
Key	Key	Key	"te k"

The difference in initial states provides one approach to identifying the use of ChaCha vs. Salsa when performing code analysis.

In addition to the key, nonce, and counter, notice that both matrixes include the ASCII text “expand 32-byte k”. This is a constant, also referred to as a “nothing-up-my-sleeve” number. It is used to provide additional variation to the output of the algorithm. From a malware analyst’s perspective, we can use this value and its location in the matrix to identify which algorithm is used for encryption. When analyzing ransomware that uses ChaCha or Salsa for encryption, the primary encryption function will require an argument that points to one of these initial states.

A 32-byte key is most common, but if a 16-byte key is used the constant is “expand 16-byte k”.

Note that the original ChaCha algorithm had a 64-bit counter and a 64-bit nonce in the initial state, but RFC 7539 modified the counter size to 32 bits (https://for710.com/rfc7539). As a result, initial state counters you observe may be 32 bits or 64 bits in size.

For additional detail, see the official publication introducing ChaCha20: https://for710.com/chacha. The official spec for Salsa20 is at https://for710.com/salsaspec. Also see this YouTube video on ChaCha20: https://for710.com/ytchacha.

Asymmetric Algorithms Use Public/Private Key Pairs

- Also referred to as public-key cryptography, this approach includes two keys: a *public* key that is shared, and a *private* key that is kept secret.
- Data encrypted with the public key can only be decrypted with the corresponding private key.
- Example applications: RSA, Diffie-Hellman, and Elliptic-curve crypto.
- Relative to symmetric encryption, asymmetric encryption is slow and best suited for encrypting small amount of data.
- In ransomware, asymmetric encryption algorithms are often used to protect symmetric keys on the target machine.

With asymmetric encryption, a public/private key pair is created where both keys are mathematically related to one another. The result is that data encrypted with one key can only be decrypted with the other, different key (and vice versa). As the name implies, the "public" key can be made public. For example, you can include it in your email signature or place links to it on social media. The "private" key should not be shared with anyone.

Asymmetric encryption is commonly used to protect messages between two parties. It is also used to support digital signatures, where the creator of a message *signs* the message so other parties can verify the message came from the sender and has not been tampered with.

One popular (and old) example of asymmetric encryption is the RSA (Rivest–Shamir–Adleman) algorithm. This algorithm was first published in the 1970s, and it is commonly used in ransomware today. In the simplest approach, the attacker embeds their public RSA key in the malware, and it is used to encrypt the symmetric keys for each encrypted file. If the target organization pays up, the attacker would provide a decryptor that includes their private RSA key, allowing each per-file symmetric key to be decrypted.

Another popular example of asymmetric encryption is the Diffie-Hellman key exchange protocol. This protocol uses public-key cryptography to help two parties generate a shared key for symmetric encryption. In doing so, it takes advantage of the security of public-key cryptography (i.e., there is no need to exchange a shared key) and the speed of symmetric cryptography.

Finally, elliptic curve cryptography (ECC) is an alternative to RSA, and it uses the math of elliptic curves to public/private key pairs.

Compared to symmetric encryption, asymmetric encryption is very slow. As a result, it is not a good choice for file encryption, but it is an excellent choice for encrypting keys (as discussed in the RSA example earlier in the notes for this slide).

RSA (Rivest–Shamir–Adleman) Is an Asymmetric Algorithm Used to Generate a Public/Private Key Pair

- RSA's security is based on the difficulty of factoring the multiplication of two large prime numbers.
- Since RSA is typically used via the Microsoft CryptoAPI, a detailed understanding of the mathematics is generally not required.
- Ransomware typically uses RSA to secure the per-file symmetric keys used for file encryption.
- Currently, RSA key lengths of 2048 bits or greater are considered secure.

RSA was published in 1977, so it is one of the oldest asymmetric algorithms used today. RSA is used to create a public/private key pair, where data encrypted using one key can be decrypted with the other key. Its security is based on the idea that it is trivial to multiple two large prime numbers but very difficult to factor the product. For example, we can easily multiple 13 x 47 to equal 611 but determining the factors of 611 is more challenging.

RSA involves mathematics that is beyond the scope of this course. Also, since malware typically uses RSA via the Microsoft CryptoAPI, a deeper understanding of the underlying math is not usually necessary. However, for an example of malware that contains RSA code without external dependencies, see this Mandiant blog post on BLACKMATTER ransomware: https://for710.com/blackmatter.

We will often encounter RSA during ransomware analysis because, as mentioned earlier, asymmetric algorithms are often used to protect symmetric keys.

Secure key lengths for RSA are currently 2048 or 4096 bits.

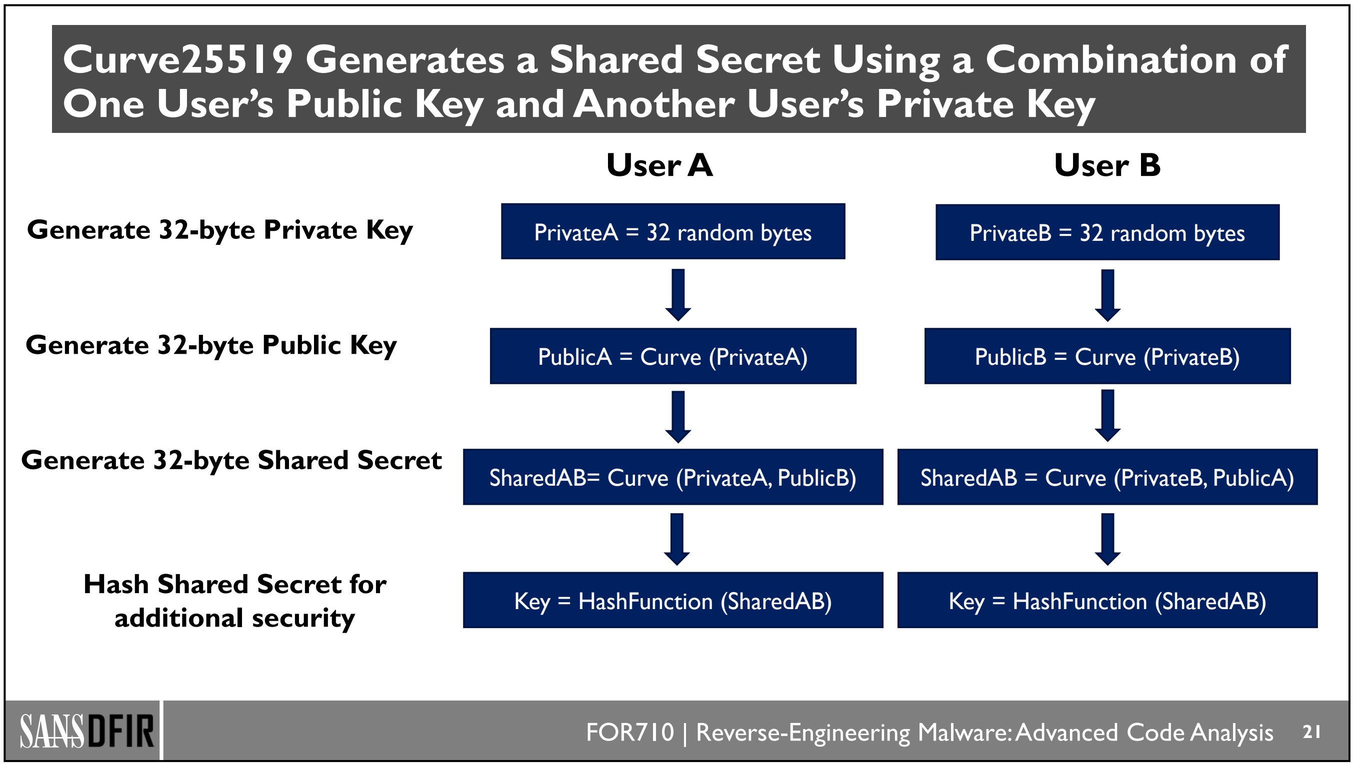
Elliptic Curve Cryptography (ECC) Is an Alternative to RSA for Public/Private Key Generation

- ECC uses the mathematics of elliptic curves to generate a key pair. And achieves the same level of security as RSA with smaller key sizes.
- Elliptic-curve Diffie-Hellman (ECDH) uses elliptic curve cryptography to establish a shared secret key.
- Ransomware uses ECDH with the Curve25519 curve to generate a shared secret as one step in the key and file encryption process.
- Recent examples include DeathRansom, Sodinokibi, and Suncrypt.

Elliptic curve cryptography (ECC) uses the mathematics of elliptic curves to generate a public/private key pair. It is an alternative to using RSA, and ECC can achieve the same level of security with a smaller key size.

Elliptic-curve Diffie-Hellman (ECDH) is a key agreement protocol that allows two parties to establish a shared secret key without exchanging the secret key. Each party has a public/private key pair such that combining one party's public key with the other party's private generates the shared secret key. Curve25519 is a specific curve created by D. J. Bernstein, and it is designed for use with Diffie-Hellman. See https://for710.com/ecdh for more information.

In ransomware, Curve25519 and ECC in general provides one approach for the attacker to generate the symmetric key used to encrypt files without exchanging sensitive information across the network. For example, the DeathRansom ransomware uses the Curve25519 elliptic curve to generate a shared key, and the SHA-256 hash of this key is used to Salsa20 encrypt an RSA public/private key pair. For more information on this malware, see https://for710.com/deathransom. The REvil ransomware also uses Curve25519 to generate a shared secret which is hashed and used as a Salsa20 key for file encryption (see https://for710.com/sodinokibi). The Suncrypt ransomware also uses ECC (see https://for710.com/suncrypt).



As described on this slide, users should begin by generating 32 random bytes to serve as the secret key. To create the corresponding public key, the private key is provided as an input. Finally, one party's public key and the other party's private key is provided as input to a function that produces a 32-byte shared secret. This shared secret is typically hashed for additional security and perhaps to produce a longer key length, depending upon the encryption algorithm.

As discussed on the previous slide, the final key can be used as a symmetric key known by both parties (e.g., victim and attacker) without the need to publicly exchange the key.

As stated on the previous slide, see https://for710.com/ecdh for more detail on the key generation process.

The Microsoft CryptoAPI Includes Functions for Encryption

- The CryptoAPI is often used to perform encryption using the AES and/or RSA algorithms.
- The CryptoAPI may be used to encrypt file contents, but this task is often performed by additional code in the executable.
- Malware authors often resolve these APIs at runtime to hinder detection and avoid including these functions in the IAT (e.g., API hashing).
- The CryptoAPI is deprecated but is still used in ransomware.

Microsoft's CryptoAPI provides well-documented and effective function for performing encryption. Within ransomware, the MS CryptoAPI is often used to perform AES and RSA encryption. To avoid detection, malware authors often obfuscate references to these APIs, so they are not easily detected using basic static file analysis.

When consulting documentation for any CryptoAPI function, you will see a notice that the API is deprecated. Nonetheless, ransomware developers continue to use and take advantage of this powerful API.

In the upcoming slides, we will discuss specific functions within the CryptoAPI that are commonly used in ransomware.

Malware That Performs Encryption via the MS CryptoAPI Typically Calls CryptAcquireContext First

- CryptAcquireContext connects an application with a Cryptographic Service Provider (CSP), which is a module that contains the desired encryption and/or decryption functionality for an algorithm
- CryptAcquireContext has five arguments:
  1. `*phProv`: Address where the returned CSP handle will be located.
  2. `szContainer`: Pointer to a key container name (typically NULL) and only used if the relevant keys are persistent on the system.
  3. `szProvider`: Pointer to the name of the CSP, including the algorithm(s).
  4. `dwProvType`: Symbolic constant that specifies the provider type.
  5. `dwFlags`: Specifies an optional value.

`CryptAcquireContext` is the first API called when the Microsoft CryptoAPI is used for encryption. It returns a handle to a Cryptographic Service Provider (CSP). A CSP is a software module (i.e., a DLL) that provides the desired encryption functionality for a supported algorithm. The handle returned by `CryptAcquireContext` is used in future calls to CryptoAPI functions to perform encryption and/or decryption.

Arguments of `CryptAcquireContext` include:

1. `*phProv`: This is a pointer to the CSP handle, which is returned by this function call.
2. `szContainer`: This is a pointer to the key container name, which is only relevant for keys that must persist on the system. In ransomware, keys do not generally persist, so this value is usually NULL. You can read more about this parameter at https://for710.com/cryptcontext.
3. `szProvider`: This is a pointer to the name of cryptographic service provider (CSP). This string will provide insight into the encryption algorithm(s) used. It typically references one of the strings listed here: https://for710.com/cspnames. If this argument is NULL, the default CSP is used. On recent versions of Windows, the default provider uses RSA.
4. `dwProvType`: This parameter is a symbolic constant, so during code analysis we will see this specified as a numerical value. Predefined provider types are listed in Microsoft documentation (https://for710.com/provtype), and the mapping of some numerical values to constants is listed here: https://for710.com/provtypevalues.
5. `dwFlags`: This is a symbolic constant that specifies optional information. In ransomware, this value is often `CRYPT_VERIFYCONTEXT` (0xF0000000), which indicates private keys will not persist on the system. As described in MS documentation, when this argument is `CRYPT_VERIFYCONTEXT`, `szContainer` must be null.

For MS documentation on `CryptAcquireContext`, see https://for710.com/cryptcontext.

For more information on CSP, Wikipedia provides a good writeup: https://for710.com/csp.

CryptImportKey Is Often Used to Import an Embedded Key

CryptImportKey has six arguments:

1. `hProv:` Handle to the key returned by CryptAcquireContext.
2. `*pbData:` Pointer to a BLOBHEADER structure for a key.
3. `dwDataLen:` Length of the key BLOB, in bytes.
4. `hPubKey:` Handle to a key that decrypts key BLOB (NULL if not encrypted).
5. `dwFlags:` If the key to import is a public key, this is NULL.
6. `*phKey:` Pointer to a handle to the imported key (upon return).

SANS DFIR FOR710 | Reverse-Engineering Malware: Advanced Code Analysis 24

`CryptImportKey` imports a key from a key BLOB. In ransomware, this BLOB is typically embedded in the executable and is often a public key.

Relevant arguments of `CryptImportKey` include:

1. `hProv`: A handle to the key returned by an earlier call to `CryptAcquireContext`.
2. `*pbData`: Pointer to the `PUBLICKEYSTRUC BLOB` that contains a header and the desired key. In case you have not heard this term before, a BLOB is a generic structure that includes a header followed by the relevant data.
3. `dwDataLen`: Length of the key BLOB, in bytes. Note that this includes zero-byte padding, and this does *not* represent the size of the key.
4. `hPubKey`: A handle to the key that decrypts the BLOB referenced by `pbData`. If the BLOB is for a public key, this argument is NULL since it is not encrypted.
5. `dwFlags`: This flag is only relevant if importing a private key BLOB. When importing a public key BLOB, this argument is NULL.
6. `*phKey`: A pointer to handle to the imported key (after the function returns).

For more information on this API, see https://for710.com/cryptmport.

The BLOBHEADER (A.K.A. PUBLICKEYSTRUC) Structure Specifies the Key Type and Algorithm

- `bType` specifies the BLOB type and includes:
  - PLAINTEXTKEYBLOB: 0x8 (symmetric key)
  - PRIVATEKEYBLOB: 0x7
  - PUBLICKEYBLOB: 0x6
- `ALG_ID` indicates the algorithm associated with the key BLOB, such as:
  - CALG_RSA_KEYX (0x0000a400)
  - CALG_AES_128 (0x0000660e)

```
typedef struct _PUBLICKEYSTRUC {
  BYTE   bType;
  BYTE   bVersion;
  WORD   reserved;
  ALG_ID aiKeyAlg;
} BLOBHEADER, PUBLICKEYSTRUC;
```

```
00032C20  06 02 00 00 00 A4 00 00  .....¤..
00032C28  52 53 41 31 00 10 00 00  RSA1....
00032C30  01 00 01 00 F1 19 97 63  ....ñ.—c
```

SANS DFIR FOR710 | Reverse-Engineering Malware: Advanced Code Analysis 25

On the previous slide, we learned that `CryptImportKey's` second argument points to a BLOBHEADER structure (also known as a `PUBLICKEYSTRUC` structure). This structure has four members, and two members are important to our analysis. The first member, `bType,` specifies the type of key BLOB that follows the header. In the example on this slide, observe the highlighted bytes beginning with a 0x6, which indicates a public key blob.

The fourth member of the `BLOBHEADER` structure specifies the algorithm associated with the key. This occurs at offset 0x4 from the beginning of the `BLOBHEADER` structure, and in the example on the slide we see the value 0x0000A400 (little endian). This value correlates with the constant `CALG_RSA_KEYX,` indicating the key is associated with the RSA public key exchange algorithm. If we view MS documentation on the `RSAPUBKEY` structure, we can gather more information about the RSA public key, including its size. See https://for710.com/rsapubkey.

For more information on the `BLOBHEADER` structure, see https://for710.com/blobheader.

For a comprehensive list of algorithm IDs, see https://for710.com/algid.

CryptGenRandom Is Often Used to Generate Random Bytes for a Key, Nonce, or Initialization Vector (IV)

- This API takes three arguments:
  1. `hProv:` Handle to a CSP returned by an earlier call to CryptAcquireContext.
  2. `dwLen:` The number of bytes of random data to generate.
  3. `pbBuffer:` Address of the buffer where random bytes will be placed.
- The number of bytes requested can provide insight into the encryption algorithm, mode of encryption, and purpose of the data.
- Other APIs used for byte generation include GetTickCount, QueryPerformanceCounter, and RtlGenRandom (SystemFunction036), rand, and srand.

```
BOOL CryptGenRandom(
  HCRYPTPROV hProv,
  DWORD      dwLen,
  BYTE       *pbBuffer
```


The `CryptGenRandom` API generates a specified number of random bytes. Ransomware often uses this API to generate a random key, nonce, or initialization vector (IV).

Arguments passed to `CryptGenRandom` include:
1. `hProv`: Handle to a CSP returned by an earlier call to CryptAcquireContext.
2. `dwLen`: How many bytes of random data to return.
3. `pbBuffer`: Address of buffer where random bytes will be returned.

Observe the specified length (`dwLen`) for the size of the key, nonce, or IV generated. This number may provide valuable insight into the algorithm used and purpose of the bytes generated. Ransomware may call API several times in close proximity to one another to generate the key and nonce/IV, if required by the encryption mode. Ransomware may also call this API numerous times to generate per-file symmetric keys.

For more detail on this API, see https://for710.com/cryptgenrandom.

Other APIs may be used as inputs to generate random data. For example, `TeslaCrypt` used `QueryPerformanceCounter` as one input to generate random content (https://for710.com/teslacrypt). In another example, the MountLocker ransomware used `GetTickCount` to generate a session key (https://for710.com/mountlocker). Also, the SunCrypt ransomware uses `RtlGenRandom` (exported as SystemFunction036 in advapi32.dll). Additional APIs used by ransomware include `rand`, `srand`, and `rand_s` (which calls `RtlGenRandom`). Note that not all of these APIs are cryptographically secure. For example, APIs like `GetTickCount` are not recommended for random number generation. You can review the documentation for these and APIs using the links below:

https://for710.com/gettickcount
https://for710.com/queryperformancecounter
https://for710.com/rtlgenrandom
https://for710.com/rand
https://for710.com/srand
https://for710.com/rands

CryptGenKey Takes Four Arguments and Generates a Symmetric Key or Public/Private Key Pair

1. `hProv:` Handle to a CSP returned by an earlier call to CryptAcquireContext.
2. `Algid:` Identifies the specific symmetric key algorithm or requests a key pair.
   - CALG_AES_256: 0x00006610 (generate AES 256-bit key).
   - AT_KEYEXCHANGE: 0x00000001 (generate RSA public/private key pair).
   - CALG_RSA_KEYX: 0x0000a400 (generate RSA public/private key pair).
3. `dwFlags:` Specifies the key type.
4. `*phKey:` Location of handle to the generated key.

```
BOOL CryptGenKey(
  HCRYPTPROV hProv,
  ALG_ID     Algid,
  DWORD      dwFlags,
  HCRYPTKEY  *phKey
```

SANS DFIR

`CryptGenKey` may be used in ransomware to generate a symmetric key or a public/private key pair. This API takes four arguments:

1. `hProv`: Handle to a CSP returned by an earlier call to `CryptAcquireContext`.
2. `Algid`: If `CryptGenKey` is used to generate a symmetric key, then this argument specifies the encryption algorithm. See https://for710.com/algid for possible symbolic constants and their numerical representations. Alternatively, if `CryptGenKey` is used to create a public/private key pair, this argument will specify `AT_KEYEXCHANGE` (0x00000001), which defaults to `CALG_RSA_KEYX` (0x0000a400) for Microsoft Cryptographic Providers. In this latter case, an RSA key pair is generated.
3. `dwFlags`: This 32-bit value specifies the type of key generated. The upper 16 bits specifies the size of the key. For example, if the 32-bit value is 0x08000001, the upper 16 bits are 0x0800, which indicates a key size of 2048. The lower 16 bits typically refers to a sub-id that depends upon the type of algorithm. In the case of RSA, if the lower 16 bits are 0x0001, this indicates `ALG_SID_RSA_PKCS` and refers to generating a public/private key pair.
4. *phKey: Address where the function will place the handle to the generated key.

A comprehensive list of symbolic constants related to the `CryptoAPI` is located in Microsoft's `wincrypt.h` header file. This file is provided in your Windows VM and can be downloaded as part of Visual Studio.

For more information about `CryptGenKey`, see the Microsoft documentation at https://for710.com/cryptgenkey.

CryptSetKeyParam May Be Called to Alter Aspects of a Key's Operation, Including the Encryption Mode

- When CryptGenKey is used to generate a symmetric key, the default encryption mode is cipher block chaining (CBC).
- CryptSetKeyParam accepts four arguments, but we will focus on the first three:
  1. `hKey:` A handle to a symmetric or asymmetric key.
  2. `dwParam:` Specifies the value to update (e.g., IV, cipher operation).
  3. `*pbData:` Pointer to a buffer that contains the value to set.

```
BOOL CryptSetKeyParam(
  HCRYPTKEY  hKey,
  DWORD      dwParam,
  const BYTE *pbData,
  DWORD      dwFlags
);
```


When `CryptGenKey` is used to generate a symmetric key, the default encryption mode is cipher block chaining (CBC) with an initialization vector (IV) of zero. However, this mode can be updated with a subsequent call to `CryptSetKeyParam` (https://for710.com/cryptsetkeyparam). This API can modify various aspects of the key's operation, and the options vary depending upon the key (i.e., symmetric vs. asymmetric key).

CryptExportKey Exports a Key Generated by CryptGenKey

CryptExportKey takes six arguments:

1. `hKey:` Handle to the key for export.
2. `hExpKey:` Specifies a key to encrypt a private key; unused for a public key.
3. `dwBlobType:` Type of key to export: PUBLICKEYBLOB (0x6) is common.
4. `dwFlags:` Typically, NULL.
5. `*pbData:` Pointer to the buffer that receives exported key BLOB data.
6. `*pdwDataLen:` Size of the buffer pointed to byte *pbData.

```
#define SIMPLEBLOB              0x1
#define PUBLICKEYBLOB           0x6
#define PRIVATEKEYBLOB          0x7
#define PLAINTEXTKEYBLOB        0x8
```

```
BOOL CryptExportKey(
  HCRYPTKEY hKey,
  HCRYPTKEY hExpKey,
  DWORD     dwBlobType,
  DWORD     dwFlags,
  BYTE      *pbData,
  DWORD     *pdwDataLen
```


If `CryptGenKey` is used to generate a key, `CryptExportKey` is often called soon thereafter to access the key. This is because `CryptGenKey` provides a handle to the key, but not the key itself.

For example, WannaCry uses `CryptGenKey` to create an RSA public/private key pair and then calls `CryptExportKey` to export the public key and write it to disk. In another example, Ryuk ransomware uses `CryptGenKey` to create a per-file symmetric AES key and calls `CryptExportKey` to append it to each file after an additional layer of encryption.

`CryptExportKey` takes six arguments:

1. `hKey`: A handle to the key. This handle is located at the address specified by the `*phKey` argument passed to `CryptGenKey`.
2. `hExpKey`: If a private key is exported, this argument is a handle to a key to encrypt the private key. If a public key is exported (most common), this argument is unused because the exported key is not encrypted.
3. `dwBlobType`: Indicates the type of key exported. Options for this argument are shown on the right-hand side of this slide in the top-most image. This screenshot is an excerpt of Microsoft's wincrypt.h header file, which contains a comprehensive list of symbolic constants related to the CryptoAPI. This file is provided in your Windows VM and can be downloaded as part of Visual Studio. `PUBLICKEYBLOB` (0x6) is most common.
4. `dwFlags`: This is typically NULL, though some options are listed in the Microsoft documentation (see link below).
5. `*pbData`: The pointer to the buffer that receives the exported key BLOB data.
6. `*pdwDataLen`: The size of the buffer pointed to byte `*pbData`.

For additional detail on this API, see https://for710.com/cryptexportkey.

CryptEncrypt Encrypts with the Key from CryptImportKey and the Algorithm Specified by CryptAcquireContextA

CryptEncrypt takes seven arguments, and we will focus on four:

- `hKey (1):` Handle to a key typically imported by CryptImportKey
- `*pbData (5):` Pointer to the data to encrypt (overwritten with ciphertext)
- `*pdwDataLen (6):` Pointer to size of data to encrypt
- `dwBufLen (7):` Size of buffer for encrypted data

The encrypted content may be larger than the plaintext data.

```
BOOL CryptEncrypt(
  HCRYPTKEY  hKey,
  HCRYPTHASH hHash,
  BOOL       Final,
  DWORD      dwFlags,
  BYTE       *pbData,
  DWORD      *pdwDataLen,
  DWORD      dwBufLen
);
```


`CryptEncrypt`, as the name indicates, actually encrypts data. All arguments are shown on this slide, but only four are relevant to our discussion:

- `hKey` (1st argument): The handle to a key typically imported via `CryptImportKey`.
- `*pbData` (5th argument): The pointer to data to encrypt. This will be overwritten as a result of the encryption process. If this argument is NULL, the function places the size of the encrypted content at the address pointed to by `pdwDataLen` (the next argument). This allows the program to determine the necessary size for the buffer where encrypted data will reside. In this case, you will see `CryptEncrypt` called again with the appropriate buffer length specified.
- `*pdwDataLen` (6th argument): The pointer to size of data to encrypt.
- `dwBufLen` (7th argument): The size of buffer where encrypted data will reside.

The encrypted data may be larger in size than the unencrypted data, so it is not unusual for `dwBufLen` to be larger than the size `*pdwDataLen` points to.

To review the Microsoft documentation on `CryptEncrypt`, see https://for710.com/cryptencrypt.

Course Roadmap

- FOR710.1: Code Deobfuscation and Execution
- FOR710.2: Encryption in Malware
- FOR710.3: Automating Malware Analysis
- FOR710.4: Correlating Malware and Building Rules

SECTION 2

- Encryption Essentials
  - ***Lab 2.1: Encryption Essentials: Quiz***
- File Encryption and Key Protection
  - Lab 2.2: Identifying File Encryption and Key Protection in Ransomware
- Data Encryption in Malware
  - Lab 2.3: Analyzing Data Encryption in Malware

SANS DFIR FOR710 | Reverse-Engineering Malware: Advanced Code Analysis 32

This page intentionally left blank.

SANS DFIR
DIGITAL FORENSICS & INCIDENT RESPONSE

Lab 2.1

Encryption Essentials: Quiz

Please begin Lab 2.1 now.

Encryption Essentials: Module Objectives, Revisited

- ✓Understand basic cryptography terminology.
- ✓Learn about cryptography modes of operation.
- ✓Differentiate symmetric and asymmetric cryptography.
- ✓Identify key differences between common crypto algorithms.
- ✓Recognize Microsoft APIs commonly used for encryption and decryption.

In this module, we covered some cryptography basics. This slide reinforces the key topics we covered.

File Encryption and Key Protection

SANS DFIR

This page intentionally left blank.

File Encryption and Key Protection: Module Objectives

- Understand common combinations of symmetric and asymmetric key algorithms used in ransomware.
- Gain a deeper understanding of common encryption algorithms.
- Learn how to identify ciphers (i.e., encryption/decryption algorithms) used in ransomware samples.
- Determine the purpose of an identified cipher (e.g., file encryption or key protection).
- Recognize Windows APIs that facilitate key generation and encryption.
- Note: This is not a cryptography course, and we only need to understand cryptography algorithms enough to identify their use.

It is worth emphasizing that this is not a cryptography class, so key concepts may seem overly simplified if you have a background in encryption algorithms. As a malware analyst, our goal is to identify the use and purpose of cryptography algorithms so we can include that detail in technical reports. Therefore, we only need to understand cryptography algorithms enough to identify their use.

Ransomware Families Vary by Symmetric/Asymmetric Key Generation and Encryption Algorithms (1)

- **WannaCry:**
  - Symmetric: Encrypts each file with AES.
  - Asymmetric:
    - Contains embedded RSA public key (attacker holds private key).
    - Generates RSA key pair and writes private key to disk (encrypted with embedded RSA public key).
    - Encrypts per-file AES key with generated RSA public key.
- **Ryuk**
  - Symmetric: Encrypts each file with AES.
  - Asymmetric:
    - Contains embedded RSA public key (attacker holds private key).
    - Contains 2nd victim-specific RSA public key and private key encrypted with 1st RSA public key.
    - Encrypts per-file AES key with the second embedded RSA public key.

This slide and the next one describe differences in how various ransomware families generate symmetric and asymmetric keys and how those keys are used for encryption.

For example, WannaCry generates an infections-specific RSA public/private key pair when run on a target machine. In addition, the malware contained an embedded public RSA key that was used to encrypt the infection-specific private key. This private key could only be decrypted with the attacker's private key upon payment. For additional detail on WannaCry analysis, see https://for710.com/wannacry.

For additional detail on how the Ryuk ransomware performs file encryption, see https://for710.com/ryuk1 and https://for710.com/ryuk2.

The main difference between WannaCry and Ryuk relates to key generation—WannaCry creates a public/private key pair upon execution, while Ryuk includes a victim-specific public/private key pair embedded in the executable.

When Identifying the Primary File Encryption Function, Look In between Other APIs Responsible for File Interactions

- Accessing a file: CreateFile
- Reading contents of the original file: ReadFile
- Writing encrypted contents and key information: WriteFile
- Reading/writing to the beginning or end of a file: SetFilePointer, SetFilePointerEx, WriteFile
- Close access to a file: CloseFile
- Update file extension: MoveFile

This page intentionally left blank.

When Analyzing Encryption in Ransomware, Consider APIs Like CreateFile

- CreateFile provides file access to read plaintext data and write ciphertext
- If successful, it returns a handle to the target file
- Consider the first two of its seven arguments:
  1. `lpFileName`: Pointer to a string that specifies the file name
  2. `dwDesiredAccess`:
     - GENERIC_READ (0x80000000)
     - GENERIC_WRITE (0x40000000)
     - GENERIC_READ | GENERIC_WRITE (0xC0000000)

```
HANDLE CreateFileA(
  LPCSTR                lpFileName,
  DWORD                 dwDesiredAccess,
  DWORD                 dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD                 dwCreationDisposition,
  DWORD                 dwFlagsAndAttributes,
  HANDLE                hTemplateFile
);
```

SANS DFIR FOR710 | Reverse-Engineering Malware: Advanced Code Analysis 41

We covered the most important CryptoAPI functions, but there are other APIs to consider when performing code analysis of ransomware.

Since our goal is to understand how a ransomware sample performs file encryption, we want to observe file-related API calls. Among the first file-related API calls we will encounter during ransomware analysis is `CreateFile`. It provides read/write access to a target file, which is necessary to encrypt its data for file encryption.

Although this API has seven arguments, we will focus only on the first two: `lpFileName` is a pointer to the target file, and `dwDesiredAccess` specifies the desired access. Ransomware may require read *and* write access to a file if it overwrites the original file with the encrypted content, and this is specified with the hex value 0xC0000000.

For more information on `CreateFile`, see https://for710.com/createfile.

SetFilePointerEx Updates the File Pointer to Read or Write Content at a Specific Location within a File

- When a file is opened/read from/written to, the *file pointer* is updated.
- SetFilePointerEx updates the file pointer via:
  - `liDistanceToMove`: A positive value moves the file pointer forward, and a negative value moves the pointer backward.
  - `dwMoveMethod`: FILE_BEGIN (0x0), FILE_CURRENT (0x1), FILE_END: (0x2).
- Ransomware commonly calls this API to read the original contents of a file (to be encrypted) or to write data to the end of the file (e.g., key info).
- SetFilePointerEx is best suited for larger files (vs. SetFilePointer).

```
BOOL SetFilePointerEx(
  HANDLE         hFile,
  LARGE_INTEGER  liDistanceToMove,
  PLARGE_INTEGER lpNewFilePointer,
  DWORD          dwMoveMethod
```


`SetFilePointerEx` and `SetFilePointer` are Windows APIs that update the location of a file pointer.

When a file is first opened, the file pointer points to the beginning of the file (zero). Whenever the file is read from or written to, the file pointer increments by the number of bytes specified. `SetFilePointer` and `SetFilePointerEx` can be used to update this file pointer with the `liDistanceToMove` and `dwMoveMethod` arguments. If the `liDistanceToMove` argument is positive, this moves the file pointer forward. If `liDistanceToMove` is negative, this moves the file pointer backward. If `liDistanceToMove` is zero, look at the fourth argument (`dwMoveMethod`) for more information on what this API accomplishes. The `dwMoveMethod` argument is often used to move the file pointer to the beginning (0x0) or end (0x2) of the file. A value of 0x1 usually indicates you must review the second argument (`liDistanceToMove`) for information on how the pointer is updated. As suggested by the above explanation, you must evaluate both the `liDistanceToMove` and `dwMoveMethod` arguments to determine the impact of executing this API. For more information on the file pointer, see https://for710.com/filepointer.

In ransomware, `SetFilePointerEx` is often used to jump to the beginning (to read the unencrypted data) or end of the file (to write data including the encrypted session key).

When comparing `SetFilePointerEx` with `SetFilePointer`, the former is the best choice for handling larger files. To read more about the `SetFilePointerEx` API, see https://for710.com/setfilepointerex.

For a calculator to convert signed hex values to negative decimal values, see https://for710.com/hextodec.

ReadFile Reads Data from the Location of the File Pointer

- Ransomware must first read in a file's contents to encrypt its data
- Consider the first three of its seven arguments:
  - `hFile:` Handle to a file returned by CreateFile
  - `lpBuffer:` Pointer to the buffer that stores the file contents
  - `nNumberOfBytesToRead:` Number of bytes to be read

```
BOOL ReadFile(
  HANDLE       hFile,
  LPVOID       lpBuffer,
  DWORD        nNumberOfBytesToRead,
  LPDWORD      lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped
);
```


In ransomware, the Microsoft ReadFile API is typically used to read file contents for encryption. As shown on this slide, the first three arguments are most important to consider:

- `hFile`: Specifies the handle to a target file. This handle can be tracked to monitor interactions with a particular file.
- `lpBuffer`: This is the address where content read in will be stored. During debugging, we can dump this address to the dump window for further review and confirmation that file contents were read in.
- `nNumberOfBytesToRead`: As is clear from the argument's name, this value specifies the number of bytes to read.

A Series of APIs We May Encounter in Malware

- CryptAcquireContext: Acquire a handle to a Cryptographic Service Provider (CSP)
- CryptImportKey: Imports embedded public key
- CryptGenRandom: Generates per-file key
- CryptEncrypt: Encrypts per-file key with embedded public key
- CreateFile (read/write): Accesses file
- SetFilePointerEx: Update file pointer to the end of the file
- WriteFile: Write the encrypted key to the end of the file
- SetFilePointerEx: Update file pointer to the beginning of the file
- ReadFile: Read contents of unencrypted file
- *<Encrypt contents of the file>*
- WriteFile: Write the encrypted contents to the file

This page intentionally left blank.

Approaches to Identify Encryption Functions and the Algorithm

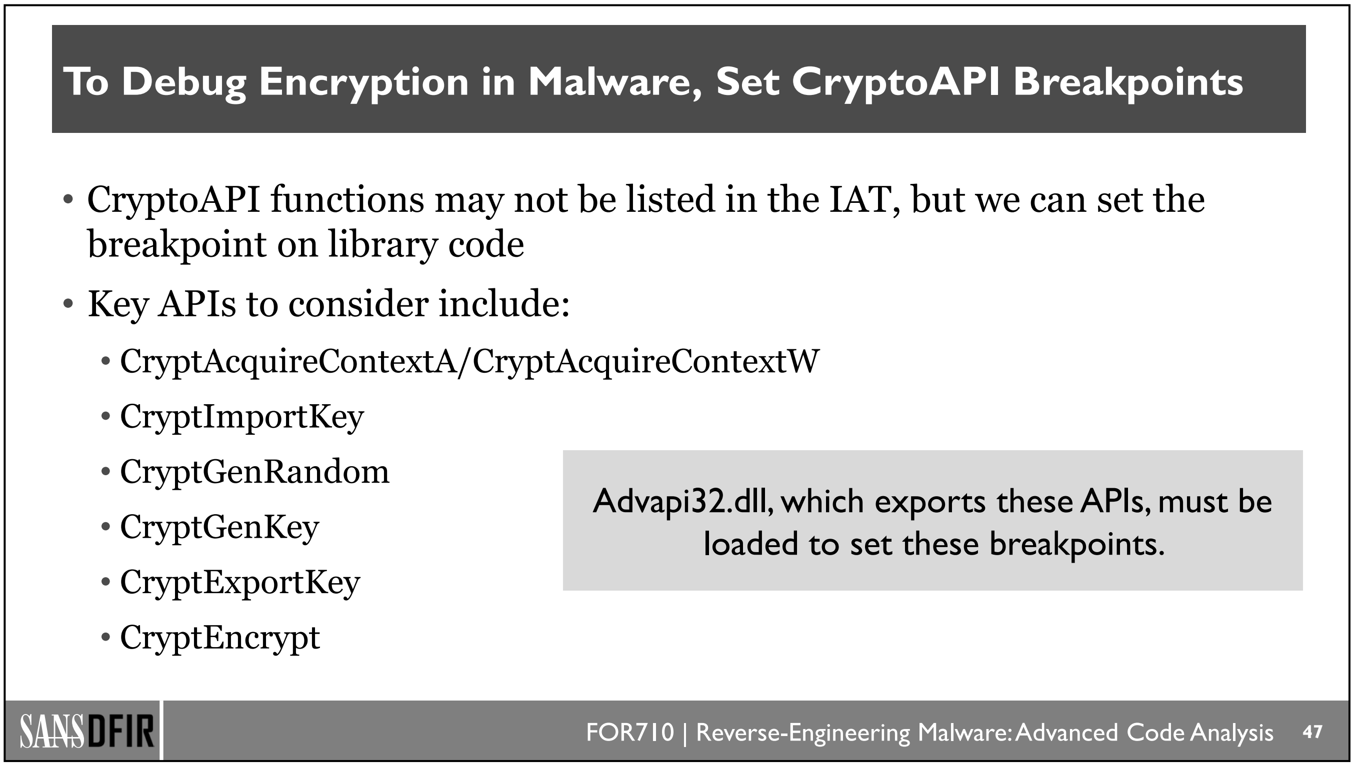
- Locate Microsoft APIs:
  - CryptoAPI functions responsible for importing, generating, and using keys
  - APIs responsible for interacting with files for reading unencrypted data and writing encrypted content
- Identify encryption constants and numerical values associated with encryption algorithms:
  - FindCrypt plugin for Ghidra
  - FindCrypt extension for Ghidra
- Mathematical operations associated with an algorithm (e.g., ROL/ROR)

When performing code analysis of ransomware that performs encryption, we want to identify the encryption algorithms used and the main function that actually performs file encryption. To accomplish this task, we will employ several strategies:

- Locate references to many of the `CryptoAPIs` we just discussed (if they are present).
- Identify API references that read/write content from/to files. As part of the file encryption process, unencrypted data will be read, encrypted, and then written to a new file or over existing data. Between function calls to reading and writing data we will often find the primary function that performs encryption. Investigating this function will give us further insight into the encryption approach used.
- Identify constants associated with crypto algorithms. For example, we can look for the "`expand 32-byte k`" constant associated with Salsa or ChaCha. We can also look for numerical value associated with the algorithm—for example, the AES S-box is public, and we can search for values within that matrix. We will rely upon the `FindCrypt` plugin and `FindCrypt` extension (created by two separate authors) to expedite this search. The extension is more recently updated, but there are cases where the plugin finds constants that the extension does not. As a result, we will use both the plugin and extension to search for crypto related information.
- Finally, we can use our basic understanding of the mathematical operations associated with an algorithm to identify its presence in malware with fairly high confidence.

For the GitHub page on the FindCrypt plugin, see https://for710.com/fcplugin.

For the GitHub page on the FindCrypt extension, see https://for710.com/fcext.



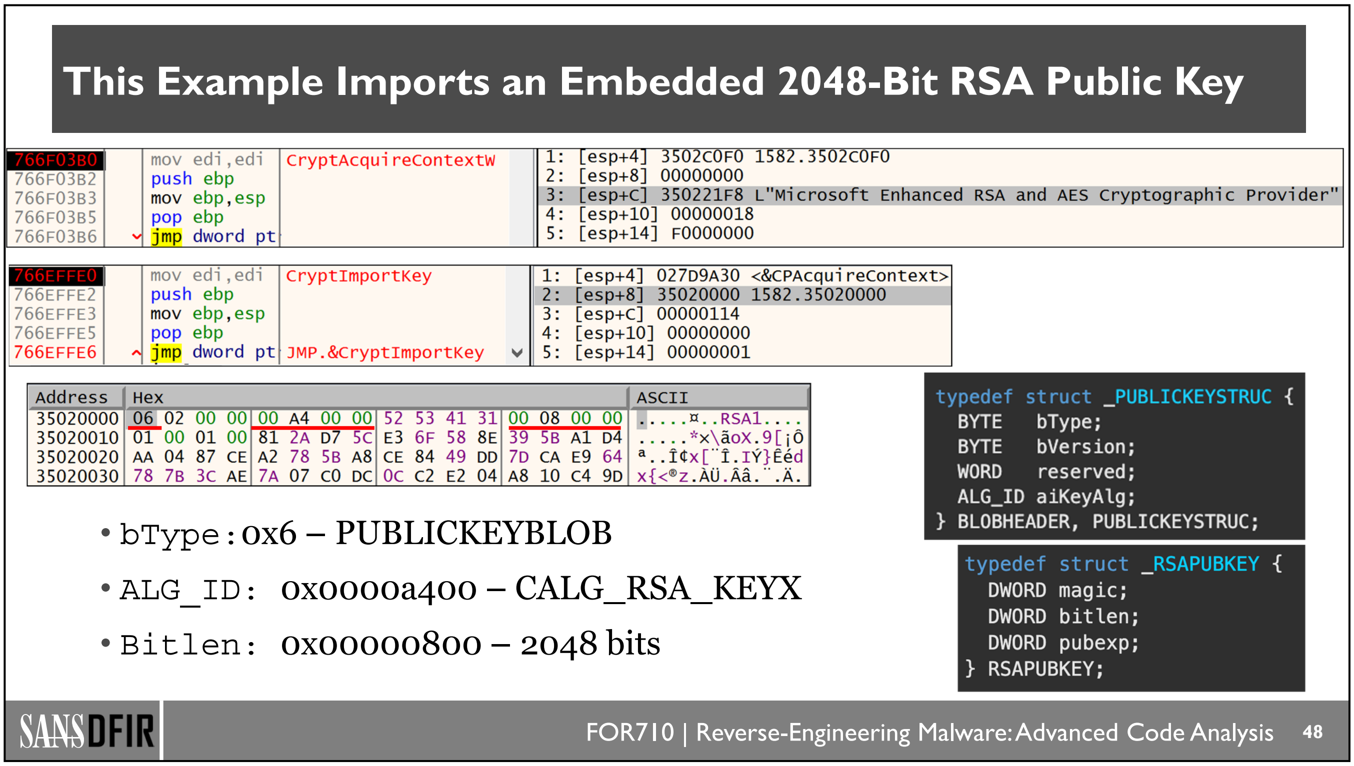
If we suspect a malware sample uses encryption, one approach is to load the target program within a debugger and set breakpoints for `CryptoAPI` functions. If the APIs are encountered during execution, this serves as a starting point for our code analysis.

Consider setting breakpoints on the following APIs:

- `CryptAcquireContextA/CryptAcquireContextW`
- `CryptImportKey`
- `CryptGenRandom`
- `CryptGenKey`
- `CryptExportKey`
- `CryptEncrypt`

We discussed these APIs in detail in earlier slides.

Note that if you attempt to set these breakpoints in x64dbg when you first load the program in a debugger, you may receive a message indicating the breakpoints are invalid. This is because the advapi.dll, which exports these APIs, may not have been loaded yet. In this case, you may need to wait until after the DLL is loaded (assuming it gets loaded at some point). One approach is to set breakpoints on `LoadLibraryA`, `LoadLibraryW`, `LoadLibraryExA`, and `LoadLibraryExW` and observe when advapi32.dll is loaded. Once this DLL is loaded, attempt to set the breakpoints discussed above.



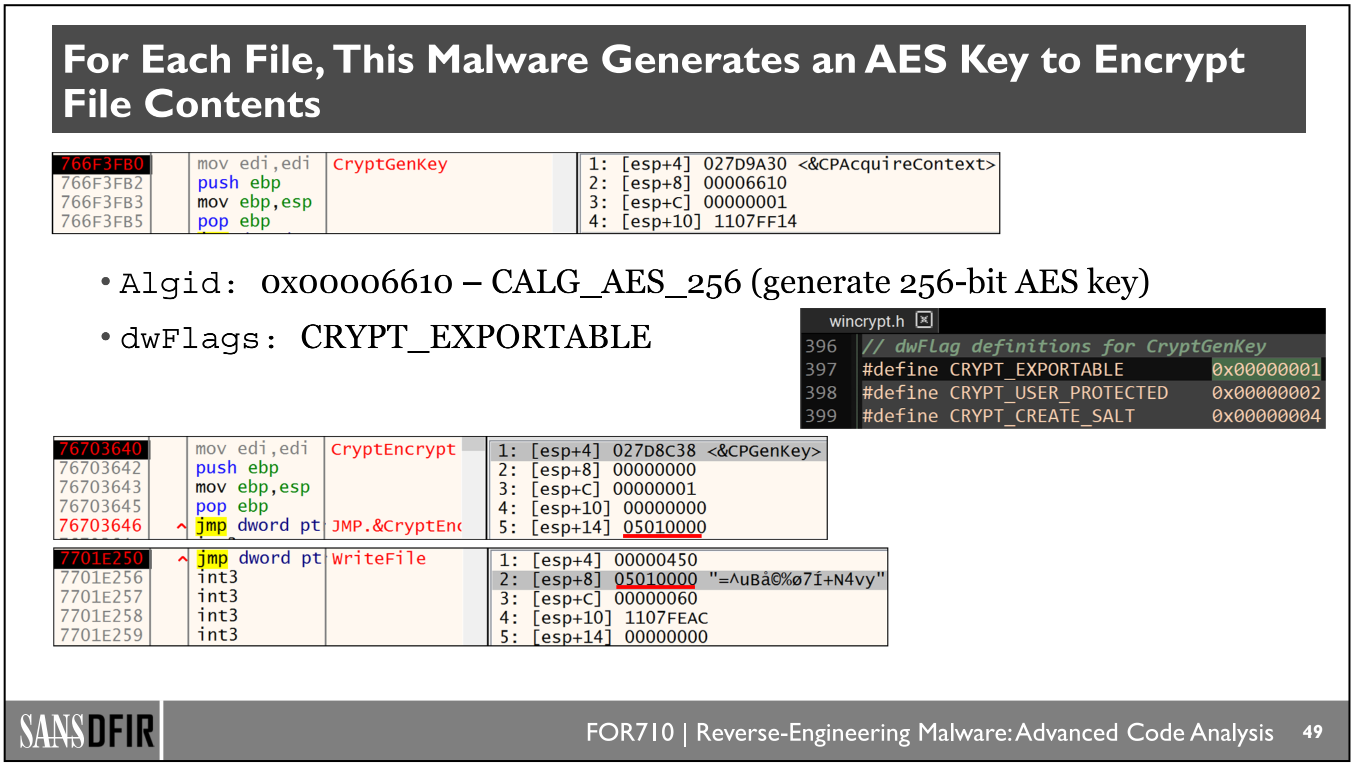
The screenshots on this slide show analysis of a ransomware sample named 1582.exe with SHA-256 hash c97c050b4fe6446a4d92f2fd36f5214c459e539fc5e5339f6a38044fc111861b. While not included in this course as an official lab, you can find it in the Malware directory within the Section 2 folder. If we found this suspicious binary on a machine that suffered a ransomware attack, we may want to investigate its encryption capabilities. As discussed on the previous slide, one approach is to configure breakpoints for common CryptoAPI functions. If we added breakpoints for all APIs on the previous slide and began debugging the program, we would arrive at the locations shown on this slide.

Note that this ransomware launches an individual thread to encrypt each file. If you decide to debug this program in your own environment outside of class, be sure to modify the default breakpoint configuration in the following way: browse to Options > Preferences and check “Thread Entry”. This will increase the likelihood that you will arrive at the beginning of a thread responsible for encryption. Without making this change, CryptoAPI breakpoints may not be triggered.

The first breakpoint we encounter is `CryptAcquireContextW`. The third argument passed to this function tells us the CSP (Microsoft Enhanced RSA and AES Cryptographic Provider), and the fourth argument (0x18) indicates the provider type is PROV_RSA_AES (https://for710.com/provrsaaes). This indicates the target program supports RSA for public key operations and AES for data encryption. However, this provider type is viewed as general purpose and does support additional encryption algorithms like RC2 and RC4.

Next, `CryptImportKey` is called to import a key. The second argument points to the `BLOBHEADER` structure, and if we dump this address to the dump window (as shown on the slide), we see an “RSA” string that is likely associated with an RSA key. As discussed earlier in this section, the `BLOBHEADER` specifies the type and algorithm. In this case, we observe values that correspond with a public RSA key.

If we view MS documentation on the `RSAPUBKEY` structure, we can also conclude this is a 2048 bit key. For more information on the `RSAPUBKEY` structure, see https://for710.com/rsapubkey.



If we continue debugging this program, we will encounter the `CryptGenKey`. As a reminder, this API can generate a symmetric key or public/private key pair. The second argument specifies the algorithm, and in this case the hexadecimal value corresponds with a 256-bit AES key. The third argument (`dwFlags`) specifies additional information about the type of key generated, and if we consult `wincrypt.h`, we can conclude it indicates that the key can be exported. This is common for session keys.

To understand what AES encryption mode is used, we could set a breakpoint on `CryptSetKeyParam`. However, we would find that this API is not called. This means the default encryption mode (CBC) is used.

If the program continues executing, we will soon encounter a call to `CryptEncrypt`. This uses the key generated by `CryptGenKey` to encrypt data located at the address specified by the fifth argument (`pbData`). This is where a file's previously read in content resides.

Next, a `WriteFile` writes the recently encrypted contents to disk. In the screenshots, observe that the fifth argument passed to `CryptEncrypt` matches the second argument passed to `WriteFile`. This makes sense, because the program encrypts the file contents in place (i.e., it overwrites the plaintext with ciphertext) and then writes the new content to the file.

The Malware Exports the Per-File AES Key, Encrypts It with the Public RSA Key, and Appends This Content to the Encrypted File

```
766F0040     mov edi,edi        CryptExportKey       1: [esp+4] 027D8C38 <&CPGenKey>
766F0042     push ebp                                2: [esp+8] 027D8AF8 <&CPGenKey>
766F0043     mov ebp,esp                             3: [esp+C] 00000001
766F0045     pop ebp                                 4: [esp+10] 00000000
766F0046   ^ jmp dword pt       JMP.&CryptExpor      5: [esp+14] 1107FC60
```

- `hKey:` The key to export (matches handle referenced by CryptEncrypt)
- `hExpKey:` The key to encrypt the exported key (RSA public key)
- `dwBlobType:` 0x1 - SIMPLEBLOB
- `dwFlags:` Typically, NULL
- `*pbData:` Address of exported key

```
wincrypt.h
471 // exported key blob definitions
472 // certenrolld_begin -- *BLOB
473 #define SIMPLEBLOB           0x1
474 #define PUBLICKEYBLOB        0x6
475 #define PRIVATEKEYBLOB       0x7
476 #define PLAINTEXTKEYBLOB     0x8
```

```
7701E250   ^ jmp dword pt   WriteFile      1: [esp+4] 00000450
7701E256     int3                          2: [esp+8] 1107FC60
7701E257     int3                          3: [esp+C] 0000010C
7701E258     int3                          4: [esp+10] 1107FE90
7701E259     int3                          5: [esp+14] 00000000
```


Next, `CryptExportKey` exports the per-file AES key and encrypts it with the embedded RSA public key. Observe that the hKey argument passed to `CryptExportKey` matches the first argument passed to `CryptEncrypt` on the previous slide—this supports the conclusion that the AES key is exported.

The second argument passed to `CryptExportKey` specifies a key to use to encrypt the exported key. While not shown on this slide, this handle to a key matches the `hKey` returned from the earlier call to `CryptImportKey`.

The third argument passed to `CryptExportKey` specifies the type of key exported. The value 0x1 corresponds with `SIMPLEBLOB`, which refers to a session key.

The fifth argument passed to `CryptExportKey` specifies the address of the buffer that receives the exported content (we will come back to this in a moment).

Finally, `WriteFile` writes the exported and encrypted per-file AES key to the encrypted file. The second argument passed to `WriteFile` specifies the address of the buffer that contains data to be written. Observe that this address matches the fifth argument passed to `CryptExportKey`, which specifies the address of the exported key.

Note that the `WriteFile` calls listed on this slide and the previous one are only an excerpt of data written to encrypted files. Additional writes are not included because our focus is on the use of CryptoAPI for file encryption and key protection, and not all details of this ransomware sample. For example, the Ryuk ransomware will also write a file marker (in this case "`RYUKTM`") to each encrypted file as an indication that it has been encrypted.

The `FindCrypt` extension and plugin, which we will discuss shortly, does not report any findings for this sample because this target program solely relies upon the CryptoAPI.

Ransomware May Include a Statically Linked Implementation of AES as an Alternative to Using the MS CryptoAPI

- Recall that AES encrypts 128-bit blocks using a series of substitutions and permutations.
- A common optimization for AES uses "T-tables" to combine the substitution, shifting, and mixing operations.
- T-table values are public, and we can search for them to identify a statically linked AES implementation.
  - The four tables used for encryption are referred to as Te0, Te1, Te2, and Te3.
  - The four tables used for decryption are referred to as Td0, Td1, Td2, and Td3.

github.com/openssl/openssl/blob/master/crypto/aes/aes_core.c

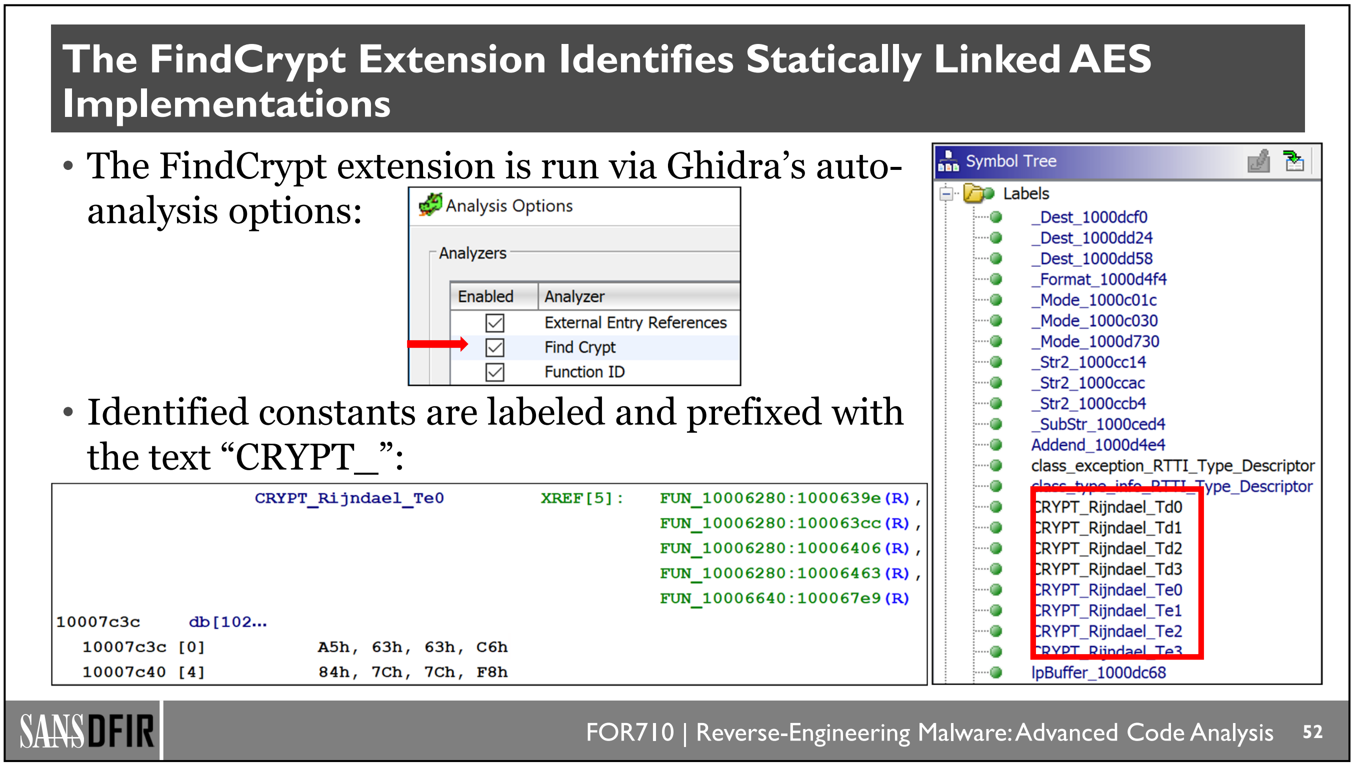
```
706  static const u32 Te0[256] = {
707      0xc66363a5U, 0xf87c7c84U, 0xee777799U, 0xf67b7b8dU,
708      0xfff2f20dU, 0xd66b6bbdU, 0xde6f6fb1U, 0x91c5c554U,
```

SANS DFIR   FOR710 | Reverse-Engineering Malware: Advanced Code Analysis   51

In the last example, we discussed a ransomware sample that uses the Microsoft CryptoAPI to generate AES keys. As an alternative, some ransomware embeds (i.e., statically links) code to generate AES keys and perform encryption. In this case, setting breakpoints on CryptoAPI calls will not be sufficient to identify AES encryption.

As a reminder, AES encrypts 128-bit blocks using a series of substitution, shifting and mixing operations. An optimization described in the initial Rijndael proposal uses tables (referred to as "T-tables") to combine the substitution, shifting and mixing operations. For more detail, see Section 5.2 in the Rijndael proposal (https://for710.com/aesprop).

The T-tables are public and included in most implementations of AES. The four tables used for encryption are referred to as Te0, Te1, Te2, and Te3. The four tables used for decryption are referred to as Td0, Td1, Td2, and Td3. The screenshot on this slide shows an example of one T-table Te0 included in openssl's implementation of AES (https://for710.com/opensslaes).

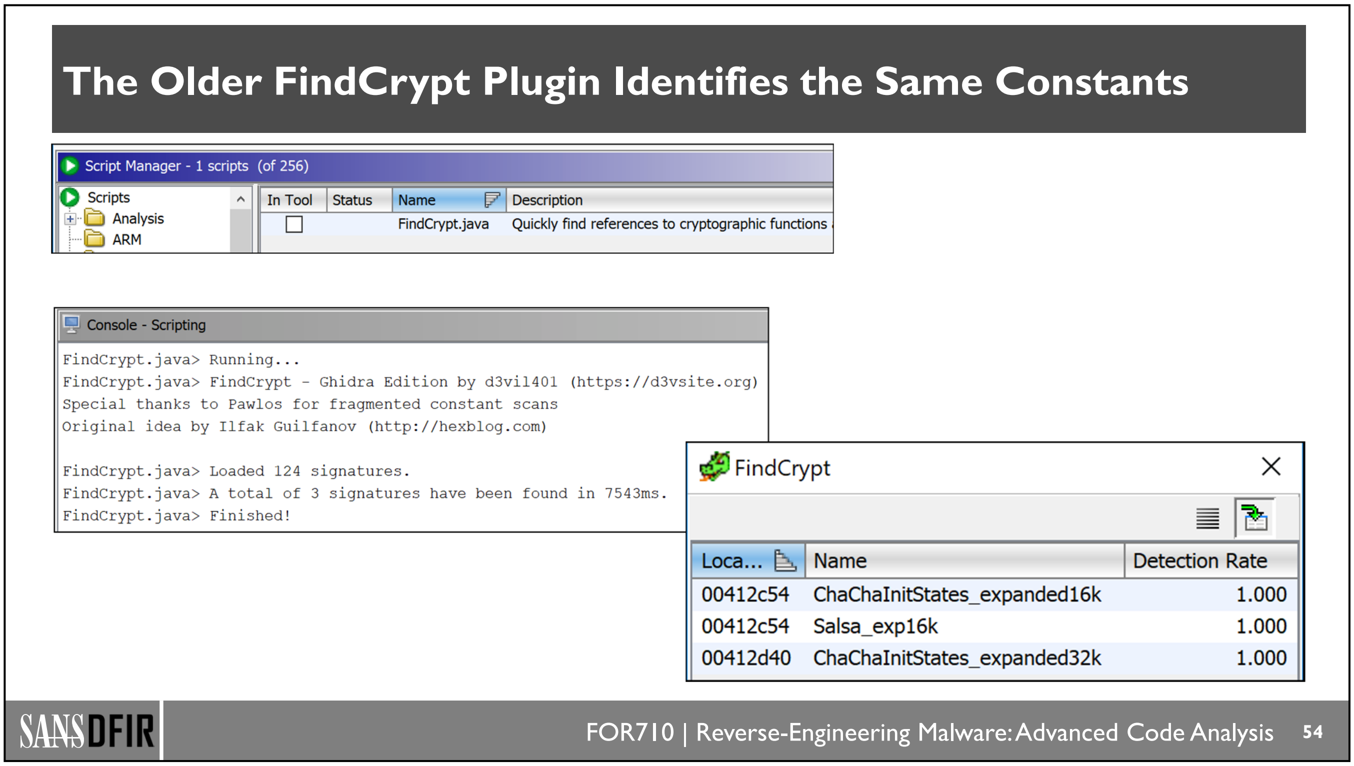


This slide shows an example of a ransomware sample that includes an AES implementation for key generation and file encryption. The file is called kbdlv.dll and it has SHA-256 hash 1be0b96d502c268cb40da97a16952d89674a9329cb60bac81a96e01cf7356830. It is located in Malware\Section2\kbdlv.zip. This ransomware is a WannaCry sample.

First, we will run the `FindCrypt` extension to identify any crypto constants. This extension is already installed in your VM, and it is available as an analysis option in Ghidra. After the extension runs, you can view its findings in the Symbol Tree window under Labels. Any identified crypto constants begin with the prefix "`CRYPT_`". In this case, we observe multiple references to labels beginning with `CRYPT_Rijndael`, indicating values associated with that cipher were located. If we jump to the label `CRYPT_Rijndael_Te0`, we observe the same values shown in the openssl source code on the previous slide.

If you jump to each `Te*` (i.e., encryption) table, you will observe multiple references to each array of values. However, there are no references to `Td*` (i.e., decryption) tables in the code. This is likely because the program only includes encryption and not decryption capabilities.

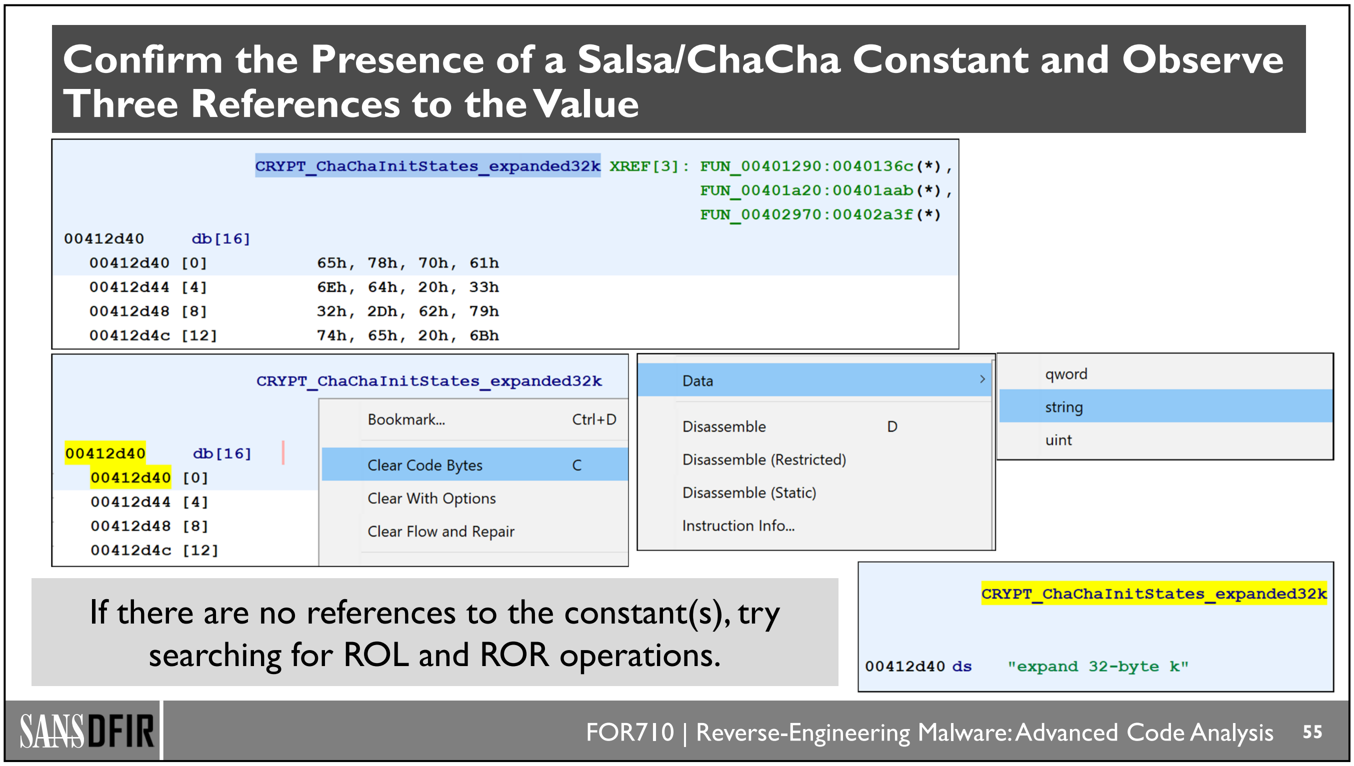
Notice that four of the five references to CRYPT_Rijndael_Te0 are located in the same function, FUN_10006280. This is likely the primary function that performs AES encryption. While we will not explore this sample further in class, you could debug this DLL and set a breakpoint on the function located at 10006280 to investigate additional details about its file encryption capabilities.



In addition to the `FindCrypt` extension, the older `FindCrypt` plugin is also installed in your VM. While the extension is based on this older plugin and *should* include all its detection capabilities, there are cases where the older plugin finds additional constants. For this reason, we will always run *both* the extension and plugin.

To run the plugin, open the Script Manager, find FindCrypt.java, and double-click on it to run the script.

When running the `FindCrypt` plugin against the example ransomware, we find identical results to those shown on the previous slide.



Next, we want to confirm if the Salsa or ChaCha algorithm is actually used for file encryption. If we click on the label `CRYPT_ChaChaInitStates_expanded16k` and attempt to identify references, we find there are none. However, if we click on the label `CRYPT_ChaChaInitStates_expanded32k,` we identify three references to that label. This indicates 32-byte (and not 16-byte) encryption keys are likely used in this sample. If Ghidra does not identify any references to the constants, try searching for ROL and ROR operations to identify potential encryption functions. To perform this search, browse to Search > Program Text from the menu bar and ensure the field "Instructions Mnemonics" is checked.

It's worth noting that for greater clarity on the identified constants, you may need to reinterpret the way bytes are presented. For example, if we jump to 00412d40 where `FindCrypt` applied the label `CRYPT_ChaChaInitStates_expanded32k`, we see an array of bytes instead of the "`expand 32-byte k`" we may expect to find. To view the appropriate string, right-click and choose Clear Code Bytes (alternatively, type C on the keyboard). Then, select and drag the mouse over the relevant bytes (so they are highlighted in green), right-click again, and browse to Data > string within the context menu. After these steps are completed, you should see the string "`expand 32-byte k`" at address 00412d40.

While this may seem obvious, if the crypto constant appears as a string in the malware, you may encounter it earlier in your analysis when reviewing strings output.

FUN_004034f0 Contains Many ARX (ADD, Rotate, XOR) Instructions and Its Shift Operations Match the Salsa Algorithm

```
004035c0   ADD   EAX, EDI
004035c2   MOV   EDI, dword ptr [EBP + local_38]
004035c5   ROL   EAX, 0x7
004035c8   XOR   dword ptr [EBP + local_10], EAX
004035cb   MOV   EAX, dword ptr [EBP + local_10]
004035ce   ADD   EAX, dword ptr [EBP + local_c]
004035d1   ROL   EAX, 0x9
004035d4   XOR   dword ptr [EBP + local_14], EAX
004035d7   MOV   EAX, dword ptr [EBP + local_14]
004035da   ADD   EAX, dword ptr [EBP + local_10]
004035dd   ROL   EAX, 0xd
004035e0   XOR   EDI, EAX
004035e2   MOV   EAX, dword ptr [EBP + local_14]
004035e5   ADD   EAX, EDI
004035e7   MOV   dword ptr [EBP + local_38], EDI
004035ea   ROR   EAX, 0xe
```

Salsa20 Quarter-Round

```
b ^= (a+d) <<< 7;
c ^= (b+a) <<< 9;
d ^= (c+b) <<< 13;
a ^= (d+c) <<< 18;
```

ChaCha20 Quarter-Round

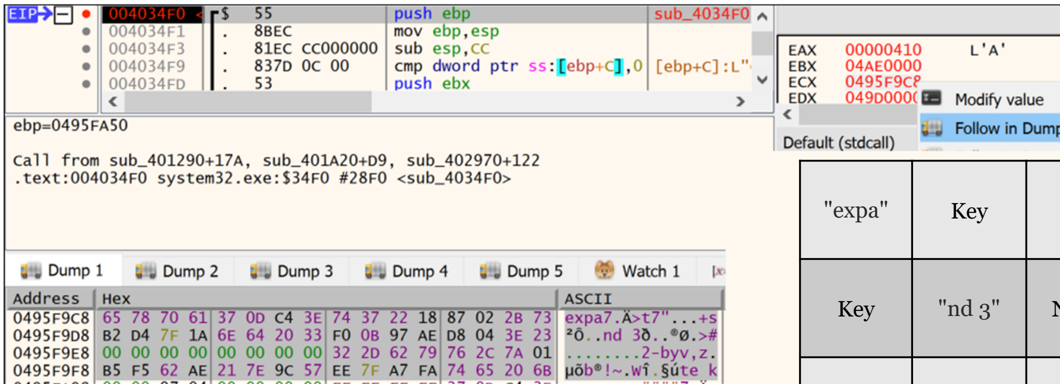
```
a += b; d ^= a; d <<<= 16;
c += d; b ^= c; b <<<= 12;
a += b; d ^= a; d <<<= 8;
c += d; b ^= c; b <<<= 7;
```

When considering a 32-bit value, ROR 0xe (decimal 14) = ROL 0x12 (decimal 18).

Looking at function FUN_004034f0, it consists of many ADD, ROL/ROR, and XOR operations. You may recall in our introduction to algorithms that Salsa/ChaCha contained many ARX operations. In addition, if we review the shift operations (i.e., ROR and ROL), we see they match the Salsa quarter-round mathematics. This is strong evidence that this function performs Salsa encryption. However, is it Salsa20? Recall that the numerical value references the number of “rounds” performed.

Note that when the most significant bit is rotated left, it will be shifted to the least significant bit. Similarly, when the least significant bit is rotated right, it will be shifted to the most significant bit. For example, the final shift left in the Salsa `quarter-round` rotates bits in a 32-bit value 18 positions to the left. This produces the same result as shifting all bits 14 positions to the right.

To Confirm the Salsa Algorithm, Debug the Program and View the Initial State Passed to FUN_004034f0



Salsa Initial State

"expa"	Key	Key	Key
Key	"nd 3"	Nonce	Nonce
Counter	Counter	"2-by"	Key
Key	Key	Key	"te k"

In addition to identify Salsa20 based on the shift operations and the number of rounds, we can view the initial state. Since FUN_004034f0 is the function that performs Salsa20 encryption, it is likely passed the initial state as an argument. Note on the previous slide that FUN_004034f0 uses the `fastcall` calling convention. This means the first two arguments are passed via ECX and EDX, with the remaining arguments pushed onto the stack.

Within your `DynamicVM`, load system32.exe into x32dbg and set a breakpoint on 4034f0. Then, run the program and you should hit the breakpoint shortly. Continue running the program for now—we'll come back to this first breakpoint hit later. When you encounter the second breakpoint hit, begin examining the arguments. Right-click on the value within ECX and dump it to the dump window. Immediately, you will notice some familiar characters. Comparing this to the generic Salsa Initial State (see bottom-right of the slide), we can identify components of the 4 x 4 matrix, including the constant. These similarities further confirm that FUN_004034f0 performs Salsa20 encryption.

Recall that the instructions above asked you to skip the first breakpoint hit. If you restart the program and return to that hit, you will notice the "key" is a readable string. This ransomware actually uses Salsa20 for multiple purposes, and one is to decrypt an embedded public key. The first breakpoint hit is used to decrypt this embedded public key, and the decryption key pays respect to D. J. Bernstein and his twitter handle (@hashbreaker). Students are encouraged to investigate that first breakpoint hit outside of class.

Static Code Analysis Confirms Salsa is Used for File Encryption

```
BVar4 = ReadFile(hFile,local_10,0x100000,&local_8,0x0);
nNumberOfBytesToWrite = local_8;
if ((BVar4 == 0) || (local_8 == 0)) break;
if (local_8 < 0x100000) {
  local_c = 1;
}
puVar1 = lpBuffer + 0x80;
uVar2 = *puVar1;
*puVar1 = *puVar1 + local_8;
lpBuffer[0x81] = lpBuffer[0x81] + CARRY4(uVar2,local_8);
FUN_004034f0(&local_8c,local_10,lpBuffer_00,local_8);
BVar4 = SetFilePointerEx(hFile,-nNumberOfBytesToWrite,0x0,1);
if (BVar4 == 0) {
  GetLastError();
}
iVar3 = WriteFile(hFile,lpBuffer_00,nNumberOfBytesToWrite,&local_14,0x0);
```

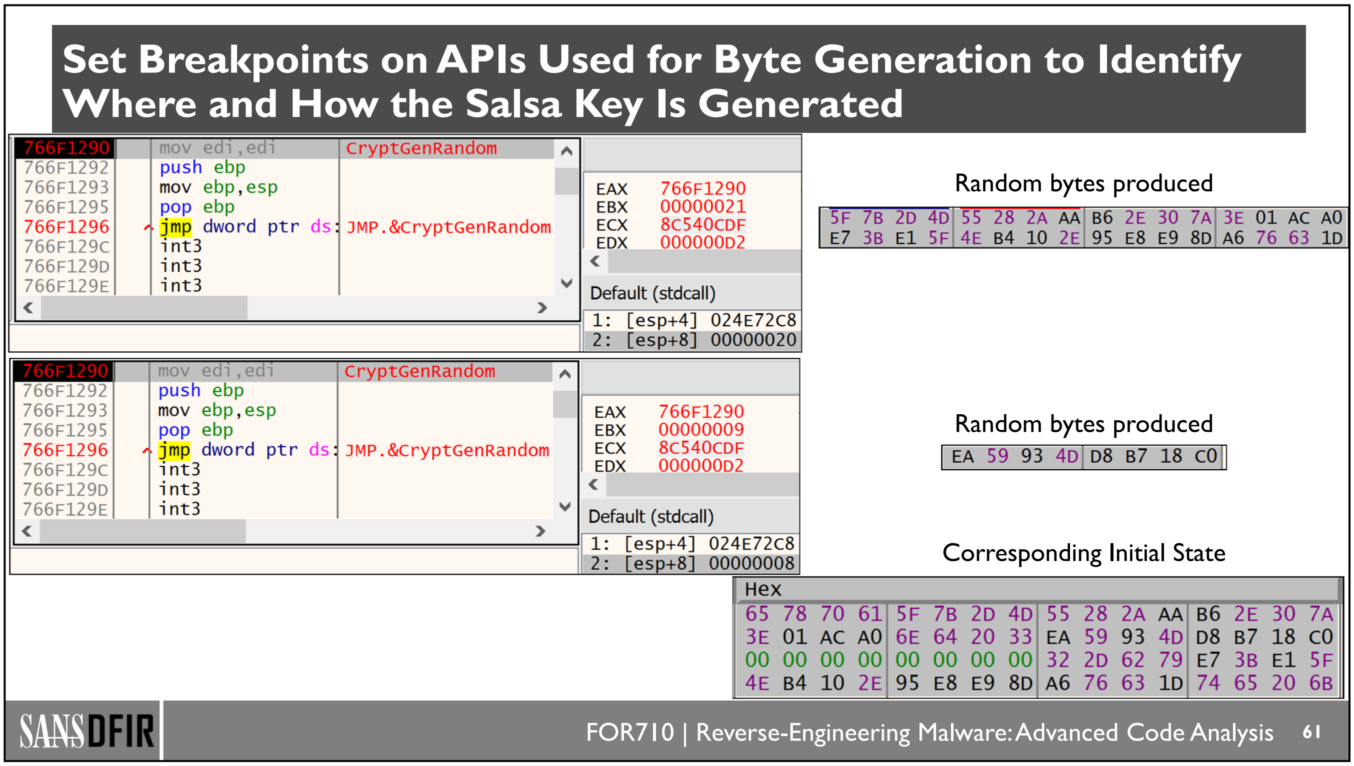

To assess if Salsa20 is used for file encryption, we will review the file interaction APIs we identified earlier and use our knowledge of Windows APIs to better understand the code surrounding the call to FUN_004034f0.

First notice the call to `ReadFile` reads 0x10000 bytes (1 MB) of a file into a buffer specified by the address in the variable local_10. This variable is passed as the second argument to FUN_004034f0, which we know performs Salsa20 encryption. In other words, the pointer to 1 MB of file contents is passed to the Salsa20 encryption function. This makes sense if this function indeed encrypts file contents. Also note that the fourth argument passed to `ReadFile` is a pointer to a variable that receives how many bytes were read in. We'll return to this observation shortly.

Next, observe the reference to `SetFilePointerEx`. It modifies the file pointer of the same file read from via `ReadFile`. Recall that the second argument specifies the distance to move the pointer, and in this case it is negative. Specifically, it shifts the file pointer backward the same number of bytes read in by `ReadFile`.

Finally, observe the `WriteFile` reference on this slide. Recall that the first argument passed to `WriteFile` is a handle to a file, and that handle matches the files referenced by the earlier calls to `ReadFile` and `SetFilePointerEx`. Also recall that the second argument passed to `WriteFile` specifies the address of the buffer with contents to write. This argument is highlighted in purple, and you can see this buffer is also referenced as the third argument passed to the Salsa20 encryption function. This supports our theory that FUN_004034f0 performs file encryption because the second argument points to the content read from a file and the third argument points to a buffer that will be written to the same file. Also, the number of bytes written to the file is the same number of bytes read from the file.

While not shown on this slide, we can also set breakpoints on these function calls and observe the inputs and outputs described in these notes. Debugging this code confirms it performs file encryption.



Now that we have confirmed this malware indeed uses Salsa20, we may wonder where and how it generates encryption keys. We could perform static code analysis and begin reviewing nearby calls to determine if any generate the necessary key and nonce values. However, this can be time consuming. As an alternative, we can set breakpoints on the following APIs commonly used by ransomware to attempt to generate random numbers (note the term *attempt* as some of these APIs are not cryptographically secure).

- CryptGenRandom
- GetTickCount
- QueryPerformanceCounter
- RtlGenRandom/SystemFunction036
- rand
- Srand

Each time we hit a breakpoint for one of the above APIs, we can review the arguments to determine if a Salsa20 key or nonce is generated based on the length of bytes specified. While debugging this program, we eventually encounter two calls to `CryptGenRandom`. Based on the arguments passed, one call requests random bytes of length 0x20 (32 decimal) and the next call to `CryptGentRandom` requests random bytes of length 8. These are the lengths of a Salsa20 key and nonce, respectively. If we continue debugging and arrive at the function at address 004034f0, we will see the generated values appear in the initial state matrix (see bytes on bottom-right of this slide). This confirms `CryptGenRandom` is used to generate the Salsa20 keys and nonces, and our review of this API's documentation indicates this is a cryptographically secure function (i.e., it generates sufficiently random bytes).

You can review Microsoft documentation on `CryptGenRandom` here: https://for710.com/cryptgenrandom.

While Debugging the Encryption Function, Observe a Different Initial State Compared to the Earlier Example

Address	Hex																ASCII
046BFE60	65	78	70	61	6E	64	20	33	32	2D	62	79	74	65	20	6B	expand 32-byte k
046BFE70	5C	15	E7	FC	FD	A9	E4	E5	07	60	96	D0	71	31	02	5D	\.çüý©äå.`.Ðq1.]
046BFE80	5C	42	8D	42	59	BA	BF	60	1F	3F	D8	1D	57	D5	77	02	\B.BYº¿`.?Ø.WÕw.
046BFE90	04	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Observe that the nonce is zero, which may negatively impact the security of the algorithm.

ChaCha Initial State

"expa"	"nd 3"	"2-by"	"te k"
Key	Key	Key	Key
Key	Key	Key	Key
Counter	Nonce	Nonce	Nonce

Continuing with our brief comparison of Salsa vs. ChaCha, if we debug the encryption function of the sample from the previous slide, we see a different initial state. This initial state matches the matrix associated with ChaCha and confirms this samples uses that algorithm for its encryption.

On a side note, observe that the nonce in the displayed initiate state is zero. Recall that the nonce (number used once) adds additional variation to the algorithm so that the same plaintext with the same key does not result in the same ciphertext. This *may* indicate a potential weakness in the encryption, but the details of investigating this possibility are out of scope of this class.

Identifying ECC (Specifically, Curve25519) Requires a More Manual Approach

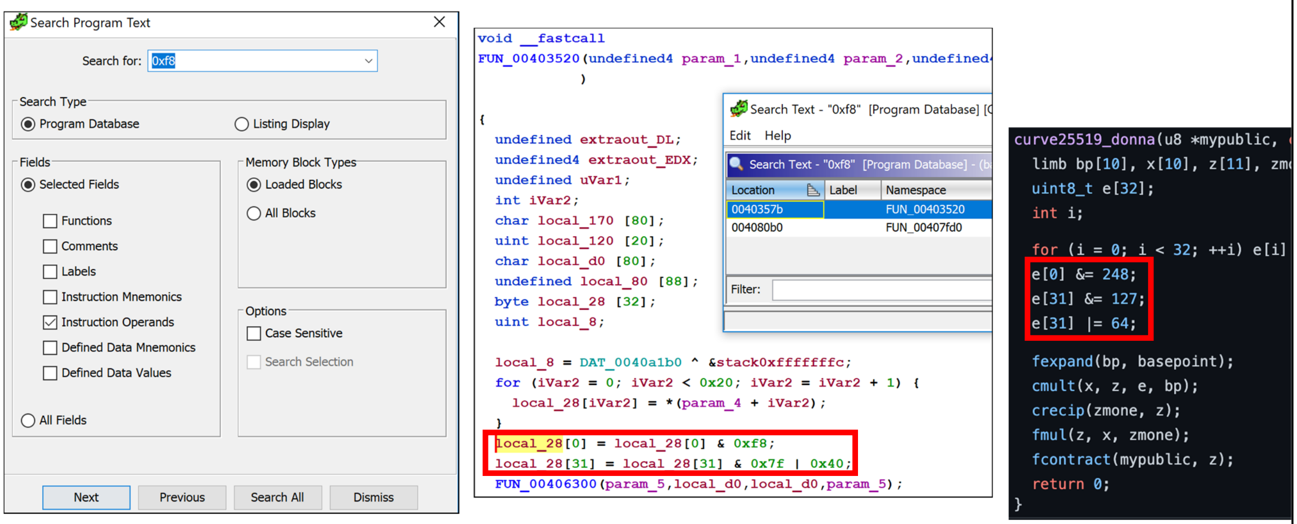
- A commonly used implementation of Curve25519 is available at https://for710.com/curve25519
- We can search for numerical values in the main function to identify potential use of Curve25519
  - 248 (0xF8)
  - 127 (0x7F)
  - 64 (0x40)

```
curve25519_donna(u8 *mypublic, const u8 *secret, const u8 *basepoint) {
  limb bp[10], x[10], z[11], zmone[10];
  uint8_t e[32];
  int i;

  for (i = 0; i < 32; ++i) e[i] = secret[i];
  e[0] &= 248;
  e[31] &= 127;
  e[31] |= 64;

  fexpand(bp, basepoint);
  cmult(x, z, e, bp);
  crecip(zmone, z);
  fmul(z, x, zmone);
  fcontract(mypublic, z);
  return 0;
}
```


Now let's explore how to identify the use of ECC (specifically Curve25519) in malware. As a reminder, recall that ECC allows two parties to compute a shared secret key using a combination of one party's public key and the other party's private key. Unfortunately, the `FindCrypt` extension and script are not helpful in identifying the use of ECC.

When ransomware uses Curve25519 for encryption, the associated code is typically embedded in the program. A common implementation of Curve25519 is located at https://for710.com/curve25519. If we review this code, we will observe that the primary function contains the values 248, 127, and 64 (see slide). We might be able to use these numerical values to locate Curve25519 in a program.

Searching for Numerical Values in Curve25519 Code Yields a Function of Interest



The ransomware sample discussed on this slide has the file name def.exe and SHA-256 hash 5a6a259071facb796bf3e46c3df71f368907397635e6e299a8df3984fae1f6ba. You can find it in your VM desktop at Malware\Section2\def.zip. This ransomware belongs to the Babuk v3 family.

In this case, running the FindCrypt extension yields no information about crypto constants. However, we can manually search for 248 (0xF8), 127 (0x7F), and 64 (0x40).

To search for a value, use the file menu to browse to Search > Program Text…

Since we are searching for values that would appear as operands, only check “Instruction Operands” under “Selected Fields”. First, let’s search for 0xf8 by clicking “Search All”. We observe two results and clicking on the first one takes us to FUN_00403520. There, we find all three numerical values in the context of operations that match the source code we discussed earlier.

There May Be Weaknesses in the Approach and Implementation of Encryption in Ransomware

- Attackers may make poor decisions when deciding what keys to generate in their environment, hardcode in the malware, and generate on target.
- Custom encryption routines are more likely to contain vulnerabilities.
- Using cryptographically insecure functions to generate random numbers *could* make the key and encrypted data more likely to be compromised.
- Incorrect implementation of ciphers (e.g., Babuk v3) may compromise the attacker's ability to decrypt files upon payment.

SANS DFIR

Although rare, some ransomware do contain weaknesses that could be exploited by defenders.

Attackers sometimes employ poor key management when deciding what keys to generate before deployment, hardcode in malware, and generate on the target machine. For example, if attackers generate a public/private key pair in their own environment and embed the public key in ransomware across multiple targets, a compromise of their private key means multiple target organizations can decrypt files. This is not advantageous from the attacker's perspective.

Also related to the proper use of keys, when using the Crypto API, Microsoft recommends using the `CryptDestroyKey` API to release the key, so it is no longer valid. If a key is generated on the target machine via the Microsoft Crypto API and this API is not called, there may be a chance for recovery in memory, but this is rare and unlikely.

The choice in cipher certainly plays an important role in determining the strength of the resulting encryption. If an attacker uses a custom algorithm instead of a standard cipher (e.g., AES), the implementation is more likely to contain vulnerabilities. Keep in mind that discovering these vulnerabilities may require in-depth and tedious code and mathematical analysis.

Another category of weaknesses may arise from the attacker's choice in random number generation. For example, we understand that generating random bytes if often necessary for a key or IV. If these values are not generated randomly, there may be opportunities to predict or brute-force the resulting value and undermine the encryption. Microsoft lists recommendations for cryptographically secure random number generators (https://for710.com/cryptorecs) and cautions developers that APIs including `rand`, `GetTickCount`, and `GetTickCount64` are *not secure* as they relate to random number generation. In other words, it may be possible to predict the numbers these functions generate, and this means the key and encrypted data could be compromised. For example, the MountLocker ransomware used `GetTickCount` to generate key data (https://for710.com/mountlocker). Note that using an insecure function for random byte generation is less secure, but it does not mean identifying the key or breaking the encryption is trivial.

One more category of potential weaknesses in ransomware relates to the implementation of a cipher. For example, in the version 3 of the Babuk ransomware's implementation of Curve25519, the program uses an unusual value for the basepoint constant when generating a public key. This *may* result in issues when decrypting files. For more information, see https://for710.com/babukv3.

For a discussion of weaknesses in ransomware in older variants, see the following resources:
"Encryption glitches in various ransomware families" section in the article located at https://for710.com/encglitches, and also https://for710.com/encglitches2 and https://for710.com/cryptofails.

Challenges When Assessing Ciphers in Malware

- The encryption algorithms we discussed may be customized.
- Malware authors may successfully implement custom ciphers, which means we need additional strategies:
  - Search for code that includes mathematical operations (e.g., XOR, shift, rotate, add).
  - Evaluate the context of any function used to generate random bytes.
  - FireEye's FLOSS tool can help identify potential encryption/decryption functions by evaluating multiple criteria.
- Malware may use non-Microsoft crypto libraries, such as Crypto++.

While this is probably clear from the material in this module, it is worth emphasizing that analyzing the use of encryption in malware can be complicated, and there are many ways a skilled attacker could make our task more challenging. For example, while the algorithms we discuss are common in ransomware, attackers could customize their implementation. As discussed earlier, customizing encryption may result in weaknesses in the algorithm, but it can also hinder our analysis if done appropriately. The result may be that we are unable to easily identify constants or other characteristics of an algorithm. In this case, we may need to rely on manual approaches.

For example, searching the assembly for mathematical operations like XOR, shift, rotate, and add may help identify code responsible for encoding and/or encryption. We also discussed looking for APIs commonly used for generating random bytes—using a reference to one of these APIs as a starting point may help locate custom functions that use the generated key or IV.

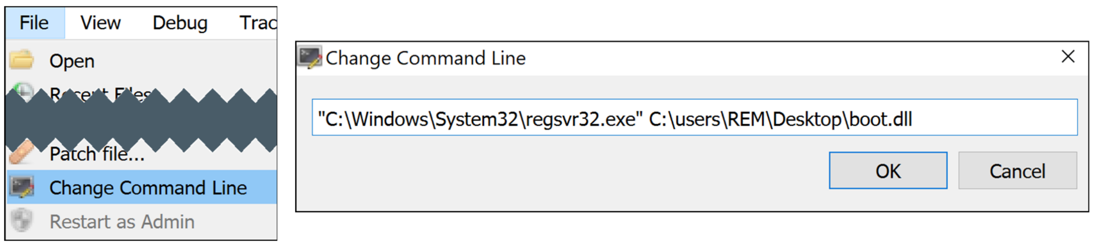
FireEye's FLOSS tool can help identify functions that perform encryption and encryption. It automatically evaluates functions and weighs various criteria including the presence of mathematical operations, the number of function references, and the number of arguments passed to the function and its size. You can learn more about FLOSS at https://for710.com/floss.

The Microsoft CryptoAPI is well documented, and it is relatively easy to understand how individual APIs work together to perform key generation, encryption, and decryption. As an alternative, a malware developer may choose to use another, non-Microsoft crypto library. For example, some ransomware uses the Crypto++ C++ library. The main website for this library is at https://for710.com/cryptopp and you can find documentation here: https://for710.com/cryptoppdocs. When used in ransomware, this library is usually statically linked and complicates analysis. For examples of malware that uses Crypto++, see the following:

https://for710.com/cryptopp1
https://for710.com/cryptopp2
https://for710.com/cryptopp3

Lab 2.2: Background Topics: Debugging DLLs That Export DllRegisterServer (1)

- The 64-bit DLL for this lab exports DllRegisterServer, which is an indication that the DLL must be run via regsvr32.exe.
- Regsvr32.exe (both 32-bit and 64-bit versions have the same name) is a command line program used to register certain types of DLLs.
- To debug the DLL using x64dbg, choose regsvr32.exe as the target and update the command line via File > Change Command Line.



In the upcoming lab, you will analyze a DLL that exports `DllRegisterServer`. This is generally an indication that the DLL must be launched via the built-in Windows program named regsvr32.exe.

Regsvr32.exe is a command line program used to register certain types of DLLs and ActiveX controls. For more information on regsvr32.exe, see https://for710.com/regsvr1 and https://for710.com/regsvr2.

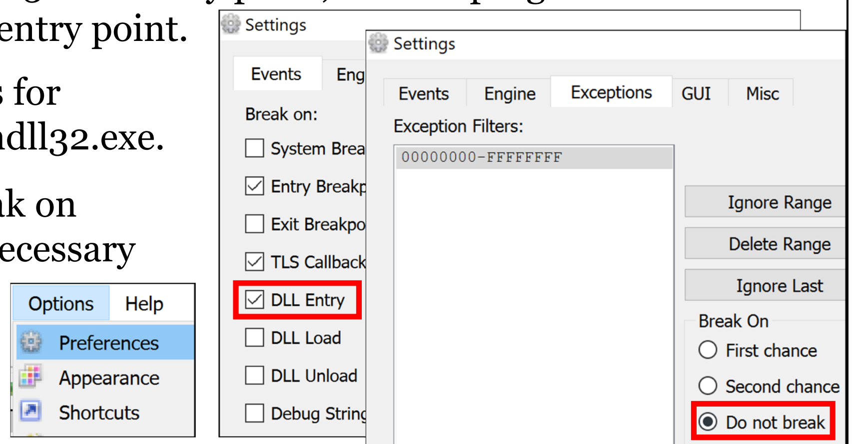
As is the case with many built-in programs on Windows, there are 32-bit and 64-bit versions of regsvr32.exe. Both have the same filename.

To analyze a DLL that exports `DllRegisterServer` in a debugger, you will need to load regsvr32.exe into the debugger first. Of course, you will need to choose the appropriate version of this executable depending upon the architecture of the suspect DLL. For a 64-bit DLL, choose C:\Windows\System32\regsvr32.exe. For a 32-bit DLL, load C:\Windows\SysWOW64\regsvr32.exe.

Then, within x64dbg, browse to File > Change Command Line and update the command line as shown on this slide.

Lab 2.2: Background Topics: Debugging DLLs That Export DllRegisterServer (2)

- Update the debugger’s settings to break on DLL entry.
- After arriving at the regsvr32.exe entry point, run the program and it should pause at the DLL entry point.
- This approach also works for debugging DLLs with rundll32.exe.
- Lastly, choose not to break on exceptions to reduce unnecessary pauses.



To arrive at the entry point of the DLL, one additional change must be made to the default configuration of x64dbg. Browse to Options > Preferences and view the Events tab. There, check the option for DLL Entry. This will ensure the debugger pauses at the DLL’s entry point once the DLL is loaded. This means you will first pause at the entry point for regsvr32.exe, and then you will need to continue running the program to pause at the DLL entry point.

Note that this approach also works for debugging DLLs with rundll32.exe, though in that case you also need to specify a DLL entry point. In fact, you could run the DLL in the upcoming lab with the command line `rundll32.exe boot.dll,DllRegisterServer`.

One final configuration update is not specific to DLLs, but it is helpful for the upcoming exercise. Jump to the Exceptions tab, single-click on the only Exception Filter, and click the radio button Do not break. This will ensure exceptions are processed as Windows would normally process them without halting the debugger. Then, click Save.

Lab 2.2: Background Topics: Multi-Threaded Malware

- Legitimate applications use multiple threads to execute tasks in parallel.
- Ransomware often uses multi-threading to encrypt files simultaneously across the local file system and network.
- Reducing the time to complete file encryption means defenders have less time to interrupt the process.
- Ransomware families that use multi-threading include Sodinokibi, LockerGoga, and LockBit.
- When debugging malware, tracking activity across multiple threads can be challenging.

The upcoming lab involves multi-threaded malware. Legitimate applications use multiple threads (i.e., they are multi-threaded) to execute tasks in parallel.

Some ransomware families use this feature to encrypt multiple files simultaneously. This speeds up the file encryption process and reduces the time defenders have to stop a successful attack. In some cases, one or more threads are dedicated to local file encryption while one or more additional threads focus on encrypting network shares.

Ransomware families including Sodinokibi, LockerGoga, and LockBit all use multiple threads to accomplish their malicious goals quickly.

When debugging a program with multiple threads, be aware that a single breakpoint might be hit by multiple threads. For example, when multiple threads in ransomware perform file encryption using the `Microsoft CryptoAPI`, each thread calls `CryptAcquireContext` separately. Using a debugger to track activity across multiple threads can be confusing for an analyst performing dynamic code analysis.

Lab 2.2

Identifying File Encryption and Key Protection in Ransomware

Please begin Lab 2.2 now.

File Encryption and Key Protection: Module Objectives, Revisited

- ✓Understand common combinations of symmetric and asymmetric key algorithms used in ransomware.
- ✓Gain a deeper understanding of common encryption algorithms.
- ✓Learn how to identify ciphers (i.e., encryption/decryption algorithms) used in ransomware samples.
- ✓Determine the purpose of an identified cipher (e.g., file encryption or key protection).
- ✓Recognize Windows APIs that facilitate encryption.

This page intentionally left blank.

Course Roadmap

- FOR710.1: Code Deobfuscation and Execution
- FOR710.2: Encryption in Malware
- FOR710.3: Automating Malware Analysis
- FOR710.4: Correlating Malware and Building Rules

SECTION 2

- Encryption Essentials
  - Lab 2.1: Encryption Essentials: Quiz
- File Encryption and Key Protection
  - Lab 2.2: Identifying File Encryption and Key Protection in Ransomware
- ***Data Encryption in Malware***
  - Lab 2.3: Analyzing Data Encryption in Malware

This page intentionally left blank.

Data Encryption in Malware

SANS DFIR FOR710 | Reverse-Engineering Malware: Advanced Code Analysis 78

This page intentionally left blank.

Malware Often Encrypts Embedded Data to Hinder Analysis

- Malware commonly encrypts data including:
  - Strings (IOCs, API names).
  - Configuration data.
  - Network data.
  - Next-stage files.
- Since the program must decrypt the content at runtime, the key is discoverable using code analysis techniques.
- We will focus primarily on RC4, a popular algorithm for encrypting data in malware.

Malware often encrypts data including configuration information, network data, and additional executable content.

RC4 Is a Symmetric Stream Cipher Often Used to Encrypt Multi-Stage Binaries, Configuration Data, C2 Traffic, and Strings

- RC4 (a.k.a. ARC4 or ARCFOUR) is a fast algorithm that is relatively easy to implement.
- Weaknesses in RC4 mean most legitimate programs should no longer use this algorithm, but it is still popular in malware.
- Malware may use the Crypto API or implement RC4 code directly in the program; we will focus on the latter, more challenging, scenario.
- The algorithm has a variable key length, usually between 40-2048 bits.

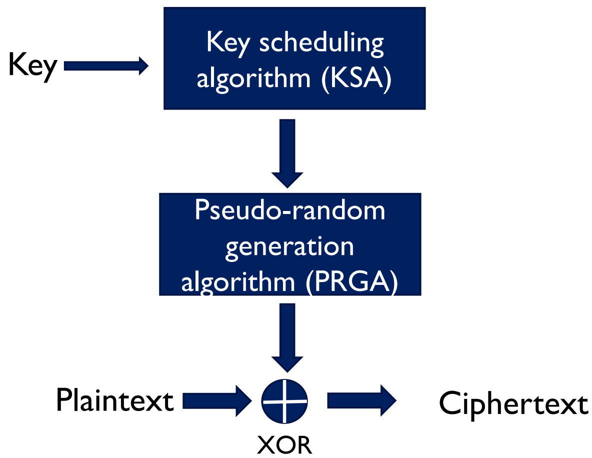
RC4 is a symmetric cipher often used in malware to encrypt data including strings. Key sizes typically range from 40-2048 bits. While a key size of less than 40 bits can be used, the maximum key size is 2048 bits.

Multiple vulnerabilities have been discovered in RC4. As a result, few legitimate applications use this algorithm. For some discussion on the weaknesses in RC4, see https://for710.com/rc4vuln1 and https://for710.com/rc4vuln2. However, malware still commonly uses RC4 because of its speed and because it is relatively easy to implement in as little as a few lines of code. From the perspective of a malware author, the algorithms benefits generally outweigh its weaknesses.

Developers may use the Microsoft CryptoAPI for RC4 encryption or implement the code in their program. If the CryptoAPI is used, the analyst's task is more straightforward, and much can be achieved by simply consulting MS documentation for the relevant APIs. However, since malware often implements RC4 without relying upon library code, it is important to understand the principles of RC4. We will focus on the second, more challenging, scenario.

It's worth noting that in addition to RC4, the RC2, RC5, and RC6 algorithms exist as well. However, RC4 is by far the most popular variation in malware. For a good writeup on the differences between these algorithms on Stack Exchange, see: https://for710.com/rcdiffs.

RC4 Uses a KSA and PRGA to Generate a Keystream



The key scheduling algorithm (KSA) initializes a substitution box and uses the key as an input for randomization. The S-box is then provided as input to the pseudo-random generation algorithm (PRGA), which performs additional math to output the keystream. Individual bytes from the keystream are then XORed with the plaintext to produce the ciphertext.

The Key Scheduling Algorithm (KSA) Initializes a 256-Byte Substitution Box (S-Box) and Mixes in Key Values

```
for i from 0 to 255
        S[i] := i
Endfor
```

Initialize S-box (identity permutation)

```
j := 0
for i from 0 to 255
        j := (j + S[i] + key[i mod keylength]) mod 256
        swap values of S[i] and S[j]
Endfor
```

Mix up S-box values based on key

SANS DFIR FOR710 | Reverse-Engineering Malware: Advanced Code Analysis 83

The code on this slide is from Wikipedia (https://for710.com/wikirc4) and it describes the RC4 key algorithm.

The first for loop initializes a 256-byte array, where each element is one byte. The array is generally referred to as “S”, and each item is assigned the value that corresponds to its index. For example, S[0] is assigned the value zero, S[1] is assigned the value one, S[2] is assigned the value two, and so on. This is known as the *identity permutation.* The “:=“ operator assigns a value to a variable.

The second for loop mixes up the values in the initialized array based on the key. An identical key will produce the same mixed-up array of values.

In the second for loop, observe the *mod* operator. The *modulo* operator (i.e., mod) produces the remainder when dividing one operand by another. For example, in the expression 10 mod 2, 10 /2 = 5 with no remainder. In this expression, 2 is the *modulus* and the result is zero. In the expression 10 mod 3, 10/3 = 3 with a remainder of 1. In this second expression, 3 is the modulus and 1 is the resulting remainder. The modulo operator allows a value to increase and then reset once it hits a certain limit defined by the modulus.

In the context of RC4, each run of the second for loop in the KSA is influenced by a byte in the key, and the position of this byte increments with each iteration. The first modulo operator ensures that once the loop reaches the last byte in the key, it resets to the first byte in the key. The second mod operator ensures that the index value assigned to “j” does not exceed decimal 255, which is the max index value for the S-box.

The PRGA Performs S-Box Lookups and Adds/Exchanges Values to Output 1 Keystream Byte

```
i := 0
j := 0

while GeneratingOutput:
    i := (i + 1) mod 256                Calculate first index
    j := (j + S[i]) mod 256             Look up value and perform sum to calculate second index
    swap values of S[i] and S[j]        Swap two values in the S-box
    K := S[(S[i] + S[j]) mod 256]       Add two S-box values and lookup a third for a keystream byte
    K ^ data byte                       One byte of keystream is XORed with a byte of data for encryption/decryption
endwhile
```


Each iteration of the pseudo-random generation algorithm (PRGA) produces one byte of keystream data. The PRGA swaps two values within the S-box and then uses the sum of those values to look up a third value within the S-box. This third value serves as one byte of the keystream data that is then XORed with a byte of data for encryption or decryption.

Because RC4 Does Not Include Constants, Use Other Approaches to Identify This Algorithm

KSA

```
for i from 0 to 255
        S[i] := i
Endfor
j := 0
for i from 0 to 255
   j := (j + S[i] + key[i mod keylength]) mod 256
   swap values of S[i] and S[j]
Endfor
```

PRGA

```
i := 0
j := 0
while GeneratingOutput:
   i := (i + 1) mod 256
   j := (j + S[i]) mod 256
   swap values of S[i] and S[j]
   K := S[(S[i] + S[j]) mod 256]
   K ^ data byte
endwhile
```

Look for:

- S-box initialization.
- 0x100 (decimal 256).
- 0xff (decimal 255).
- XOR.
- S-box passed between two loops.

(KSA and PRGA code may reside in one function or in two separate functions.)

Although RC4 does not use crypto constants, there are other approaches to identify RC4 code with high confidence. Because this algorithm can be implemented in relatively few lines of code, the code generally resides in the target program and not a separate library.

One approach to identifying RC4 is to look for the S-box initialization. During code analysis, the S-box will be represented as an array, and we can look for a loop where array values are assigned values 0 through 255.

Next, the KSA and PRGA code on the previous slide contains multiple references to the values 255 and 256 across the loop conditions and modulus values. To locate relevant loops, identify conditions that evaluate if a counter is less than 256 (0x100) (this is more common than checking if the counter is less than or equal to 255). We may, however, see a reference to decimal 255 (0xff) when the modulus operator is used. This is because ANDing a value X with 0xff is equivalent to X mod 256.

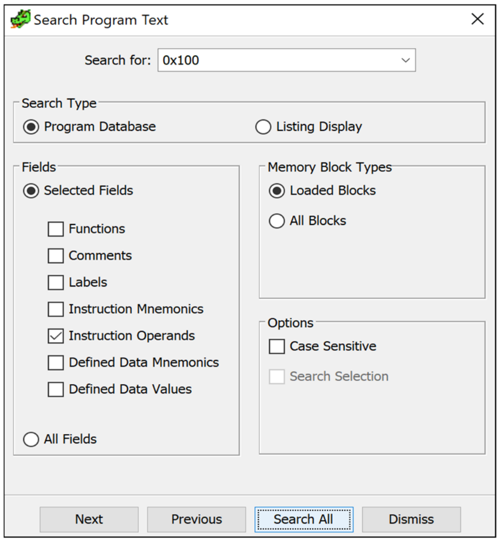
To help locate the PRGA loop specifically, we can look for an XOR at the end of the loop.

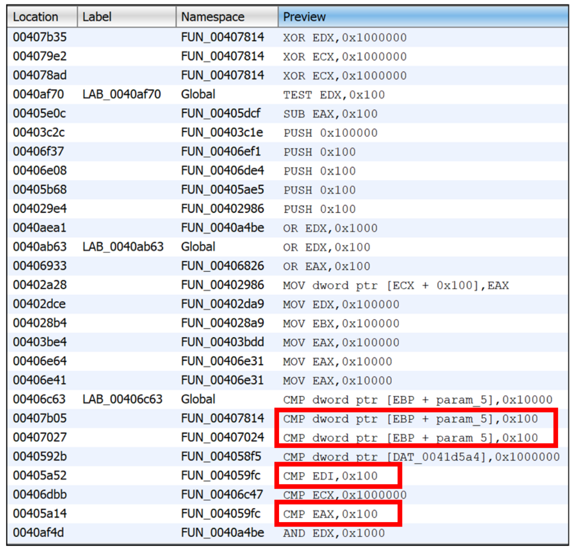
Lastly, we can confirm that the S-box is passed into the PRGA code.

Note that the KSA and PRGA code may be in two separate functions or together in one function.

For an excellent YouTube video on RC4 that inspired some of the strategies on this slide, see https://for710.com/ytrc4.

Search for 0x100 across Instruction Operands and Search for Code That Resembles S-Box Initialization





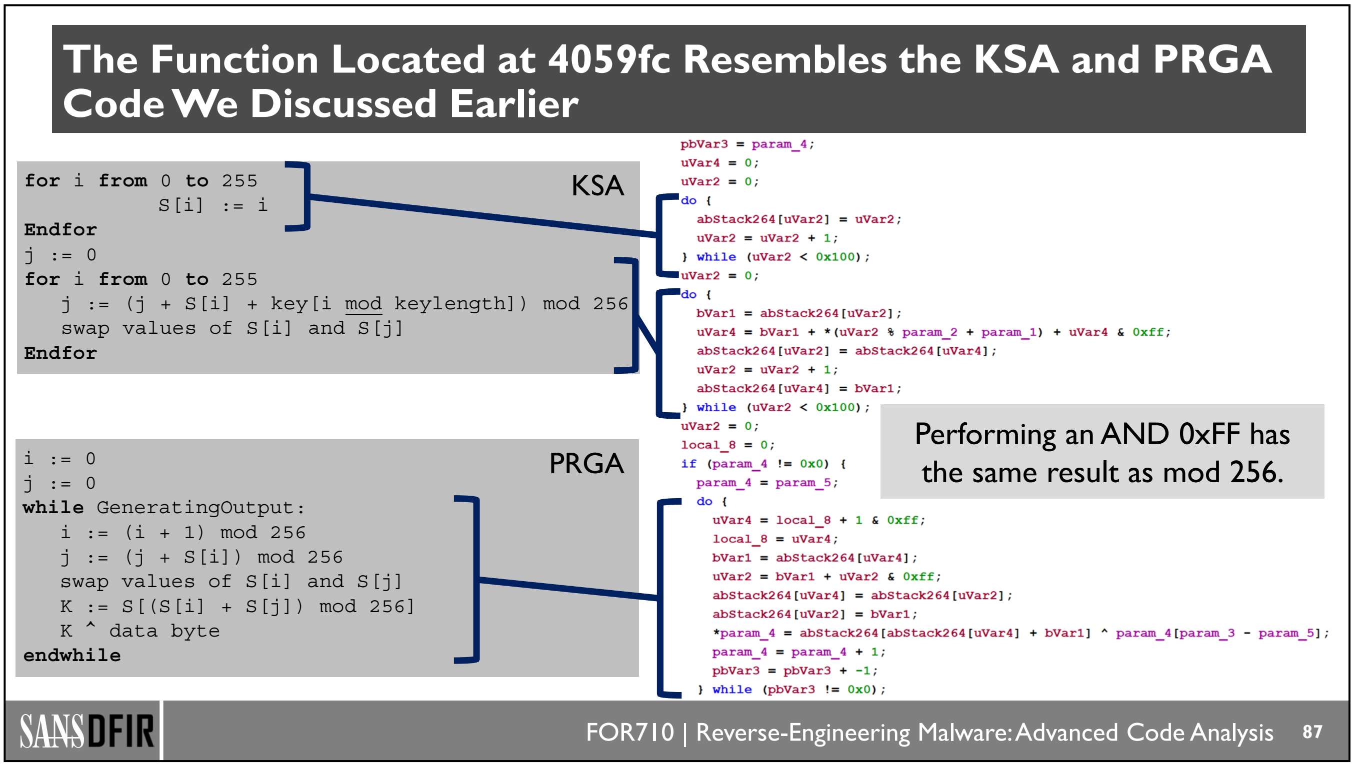
Location	Label	Namespace	Preview
00407b35		FUN_00407814	XOR EDX,0x1000000
004079e2		FUN_00407814	XOR ECX,0x1000000
004078ad		FUN_00407814	XOR ECX,0x1000000
0040af70	LAB_0040af70	Global	TEST EDX,0x100
00405e0c		FUN_00405dcf	SUB EAX,0x100
00403c2c		FUN_00403c1e	PUSH 0x100000
00406f37		FUN_00406ef1	PUSH 0x100
00406e08		FUN_00406de4	PUSH 0x100
00405b68		FUN_00405ae5	PUSH 0x100
004029e4		FUN_00402986	PUSH 0x100
0040aea1		FUN_0040a4be	OR EDX,0x1000
0040ab63	LAB_0040ab63	Global	OR EDX,0x100
00406933		FUN_00406826	OR EAX,0x100
00402a28		FUN_00402986	MOV dword ptr [ECX + 0x100],EAX
00402dce		FUN_00402da9	MOV EDX,0x100000
004028b4		FUN_004028a9	MOV EBX,0x100000
00403be4		FUN_00403bdd	MOV EAX,0x100000
00406e64		FUN_00406e31	MOV EAX,0x10000
00406e41		FUN_00406e31	MOV EAX,0x10000
00406c63	LAB_00406c63	Global	CMP dword ptr [EBP + param_5],0x10000
00407b05		FUN_00407814	CMP dword ptr [EBP + param_5],0x100
00407027		FUN_00407024	CMP dword ptr [EBP + param_5],0x100
0040592b		FUN_004058f5	CMP dword ptr [DAT_0041d5a4],0x1000000
00405a52		FUN_004059fc	CMP EDI,0x100
00406dbb		FUN_00406c47	CMP ECX,0x1000000
00405a14		FUN_004059fc	CMP EAX,0x100
0040af4d		FUN_0040a4be	AND EDX,0x1000

SANS DFIR FOR710 | Reverse-Engineering Malware: Advanced Code Analysis 86

An as example, we will search a malware sample for values associated with RC4 and assess the results. The malware sample has SHA-256 hash 6b212864731c131bd095c2537ca14e10338d3ebf997dda59465c5f1ce73d418b. You can find this file named file.exe in the Malware\Section2 folder within your Windows VMs. For more information on this family of ransomware, see https://for710.com/revil1.

After loading the malware sample into Ghidra, we can search for values from the Menu bar by browsing to Search > Program Text…. There, check “Instruction Operands” only, search for 0x100, and click Search All.

In the results window, sort by the Preview column and look for CMP instructions that include 0x100 as an operand.



As shown on this slide, the code we reviewed when introducing RC4 closely matches the decompiled code within the function starting at address 004059fc.

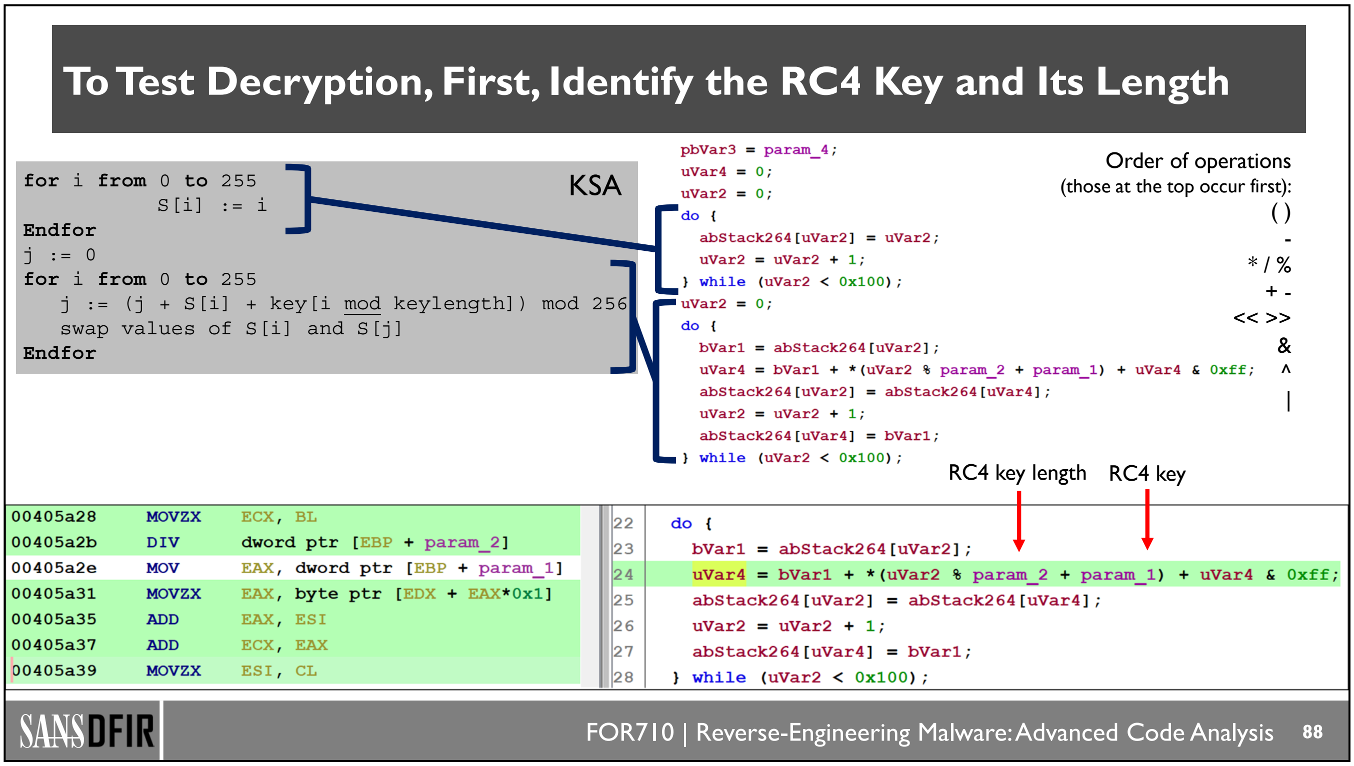
Let's take a closer look at the first do-while loop. uVar2 is initialized to zero before the loop begins, and each value in the array abStack264 is initialized to a value that corresponds to its index until the counter reaches 0x100 (256). This is exactly what we see in the initialization code on the left side of this slide.

Moving on to the second loop, the second line of code is most challenging because it includes both a modulus operator as well as the AND operation. The presence of both these operations means this line likely refers to the encryption key, since that is the only line of code within the KSA that includes two modulo operators (as discussed on the previous slide, performing an AND with 0xff achieves the same result as mod 256). Identifying the key is important, because we eventually want to confirm the presence of RC4 by testing the algorithm with both the key and sample data.

We can see the third loop is very similar to the PRGA code based on the multiple mod 256 operations and XOR operation that occurs near the end of the loop. You may wonder why we only see two values ANDed with 0xff instead of three. In this case, the decompiled output does not reflect the third mod 256 operation. If you look at the corresponding assembly, you will find the instruction MOVZX, ECX, CL at 00405aab. This has the same impact as performing an AND with 0xff because only the lower 8 bits of CL are placed into ECX and the value is zero extended (i.e., the remaining bits are set to zero).

In this case, both the KSA and PRGA code are in one function. In other programs, the KSA and PRGA code may appear in separate functions.

It is also worth noting that the code on this slide shows do-while loops, but other RC4 implementations may use for loops instead.



As suggested on the previous slide, the second line of decompiled code in the second loop of the KSA likely references the key. We need to understand what this corresponds to in the assembly code so we can debug the program and extract the key.

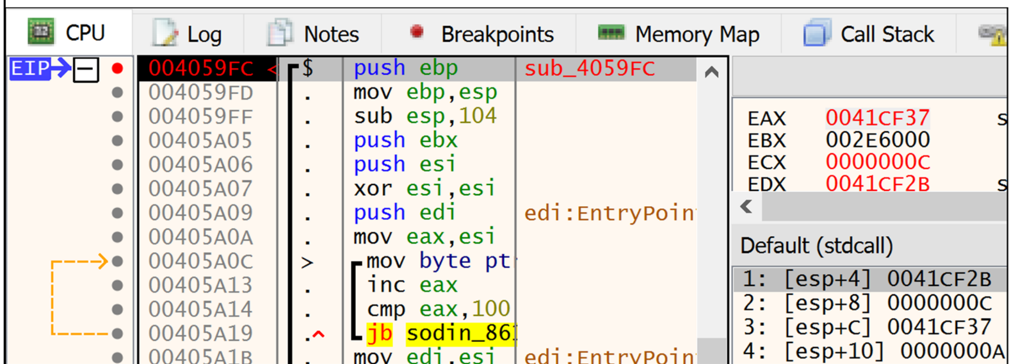
In the decompiled code that is highlighted at the bottom-right of this slide, we see a modulo operator and an `AND` operation. By now we recognize the `AND` operation as mod 256. The first mod operation correlates with the code `key[i mod keylength]` shown on the top-left of this slide. Specifically, let's review the decompiled code `uVar2 % param_2.` Between these two operands, which is the `keylength`, and which is `i`? If we highlight `param_2` in the decompiled code with a single click, we see no other references in the body of this function. If we highlight `uVar2`, we see multiple references, and this variable is clearly incremented and checked with each iteration of the loop. Therefore, `uVar2` correlates with `i` in the KSA code on the top-left of this slide, and `param_2` is the modulus or keylength. What then, does `param_1` refer to? Since the entire expression (`uVar2 % param_2 + param_1`) is dereferenced, `param_1` must contain the starting address for the RC4 key.

When looking at the assembly code, note that `param_2` is the operand for the instruction at 405a2b. The DIV instruction divides EDX:EAX by the specified operand. The result is stored in EAX, and the remainder is stored in EDX. At 405a31, EDX is added to EAX, so EAX correlates with `param_1` in the decompiled code. As discussed above, `param_1` probably contains the address of the RC4 key.

Reference:
https://for710.com/orderops

Confirm RC4 by Identifying the Arguments That Point to the Key and Its Length, and the Ciphertext and Its Length

Use x32dbg to set a breakpoint at 4059fc and identify the arguments.



Argument #1: Pointer to RC4 key
Argument #2: RC4 key length
Argument #3: Pointer to ciphertext
Argument #4: Ciphertext length

We can confirm that this function performs RC4 decryption by using a RC4 key, key length, ciphertext, and ciphertext length.

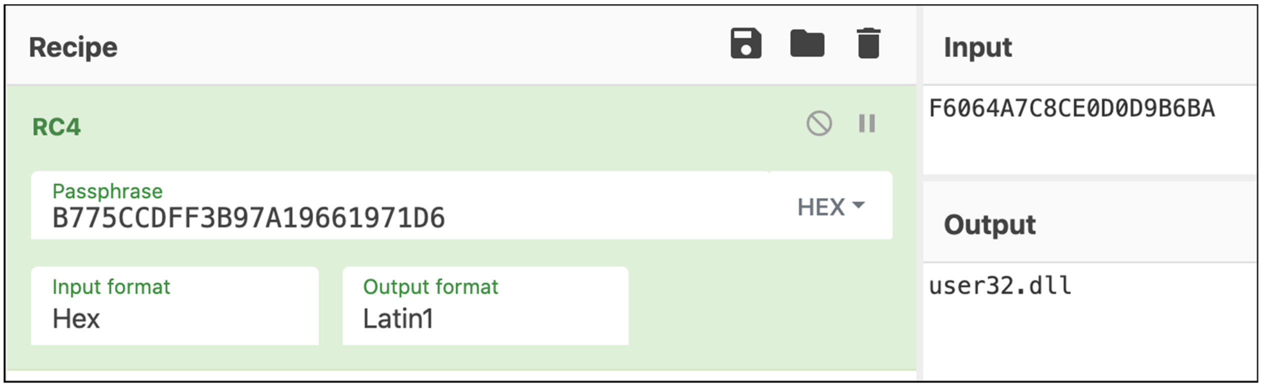
To obtain some test values, debug the program using x32dbg. Load file.exe into the debugger, jump to 4059fc (Ctrl +G), and set a breakpoint. Next, run the program until the breakpoint is hit.

Since the function at 004059fc follows the `fastcall` calling convention, the four arguments we are interested in (the third, fourth, fifth, and sixth arguments) are all on the stack and labelled on this slide.

We can dump the third and fifth arguments to a dump window to view the data. Copy the data from each dump window by highlighting the appropriate size of data (specified by the fourth and sixth arguments), right-clicking, and browsing to Binary > Copy.

Use CyberChef to Test RC4 Decryption

- CyberChef confirms the target function performs RC4 decryption.
- The first call to the function at 004059fc decrypts a DLL name.



Open CyberChef from the desktop and search for the RC4 operation. Double-click on it to add it to the recipe.

Within the RC4 operation, update the Passphrase pulldown to HEX and change the Input format to Hex.

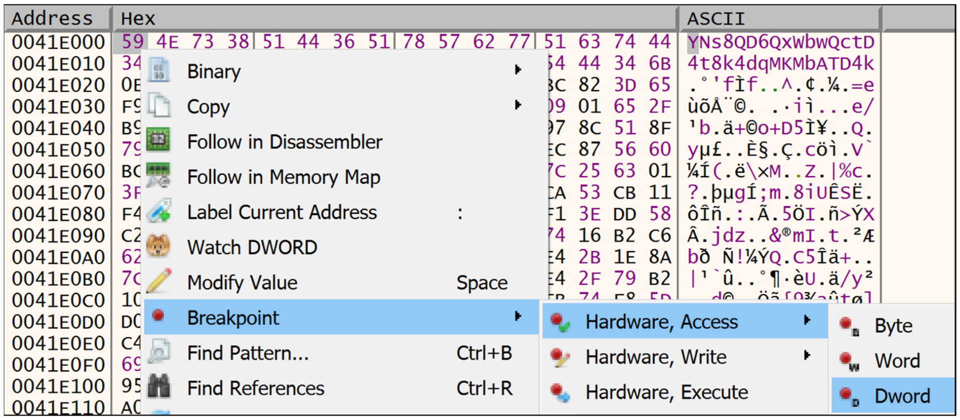
Then, use the arguments passed to 004059fc to test decryption. First, copy and paste the RC4 key (of the appropriate length specified by the fourth argument) from x32dbg to the `Passphrase` field.

Finally, copy and paste the sample ciphertext (of the appropriate length specified by the sixth argument) into the Input field on the top-right.

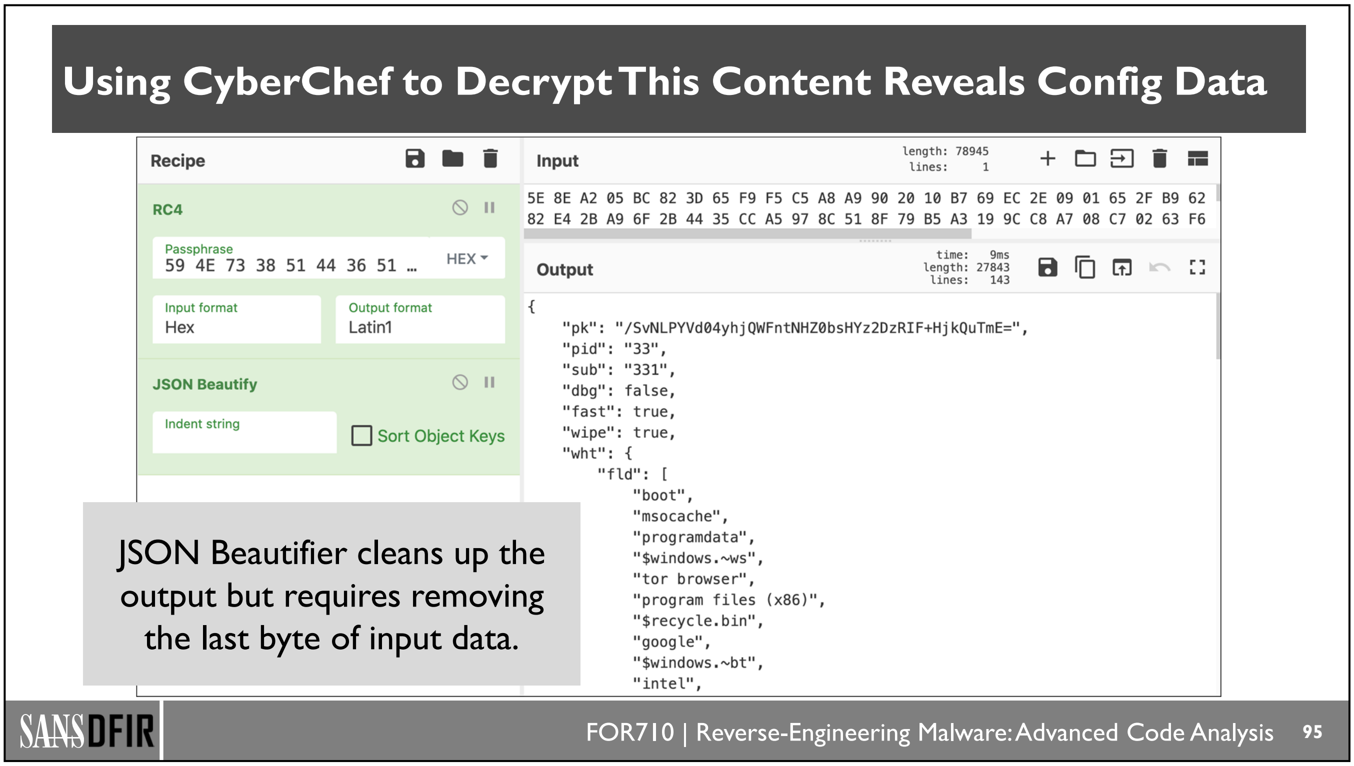
The output should be a decrypted string: user32.dll. This confirms the function of interest does in fact perform RC4 decryption. If we continue debugging the program, we will encounter additional decrypted strings.

Set a HW Access Breakpoint at the Beginning of the Section and Observe When That Content Is Accessed

Set a hardware breakpoint on the section's virtual address: 41E000.



Within x32dbg, we can set a hardware breakpoint at 41E000. Then, run the program.



If we copy and paste the data discussed in the previous slide into CyberChef, we observe what appears to be configuration data for this malware sample. We can add the JSON Beautifier operation to the recipe for easier reading. However, we need to delete the last byte of input data (0x06) for the JSON Beautifier to work properly.

Variations in the RC4 Implementation

- The S-box initialization may include additional code which expands the key to the size of the S-box (i.e., key expansion).
- RC4 code may use do-while loops or for loops.

```
for (local_10 = 0; local_10 < 0x100; local_10 = local_10 + 1) {
  *(iVar2 + local_10) = local_10;      Identity permutation
  *(iVar3 + local_10) = *(param_4 + local_10 % param_5);      Key expansion
}
local_18 = 0;
for (local_14 = 0; local_14 < 0x100; local_14 = local_14 + 1) {
  local_18 = *(iVar2 + local_14) + local_18 + *(iVar3 + local_14) & 0x800000ff;      Only one mod operation
```

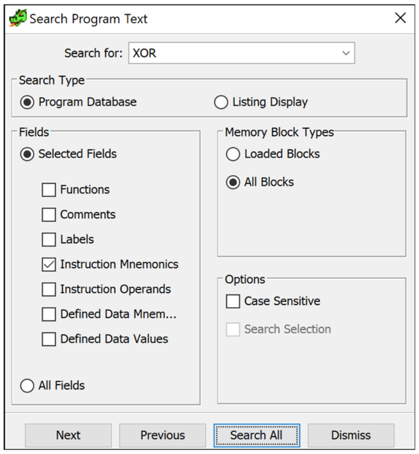
- The KSA and PRGA code may not be within the same function.
- The S-box may be a byte array or integer array, which means four values are assigned with each iteration (see next example).

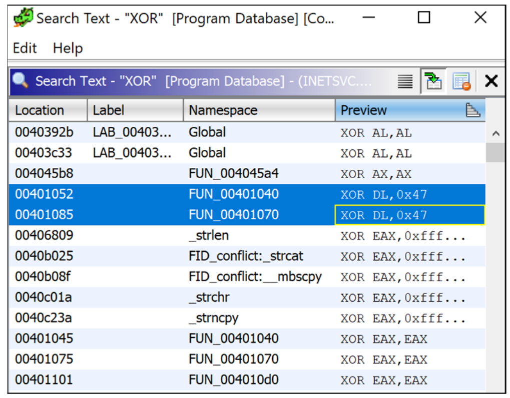
There are several variations you may encounter in the implementation of RC4 when performing code analysis. First, it's possible that the S-box initialization code will include more than just the identity permutation. In the example on this slide from malware with SHA-256 hash 58e923ff158fb5aecd293b7a0e0d305296110b83c6e270786edcc4fea1c8404c, the first loop within the KSA function that initializes the S-box also expands the key to the size of the S-box (i.e., it repeats the key bytes in memory for 256 bytes). One important indicator that key expansion code may reside in the initialization loop is the presence of a mod operator. As a result, the next for loop that is responsible for mixing up the S-box based on the key does not need to perform a mod operation involving the key length; it simply references the expanded key.

The last two variations will be covered in an upcoming example.

Communications with a C2 Server May Be Encrypted or Encoded

Searching for the XOR instruction mnemonic is one simple approach to identifying potential encoding/decoding routines.



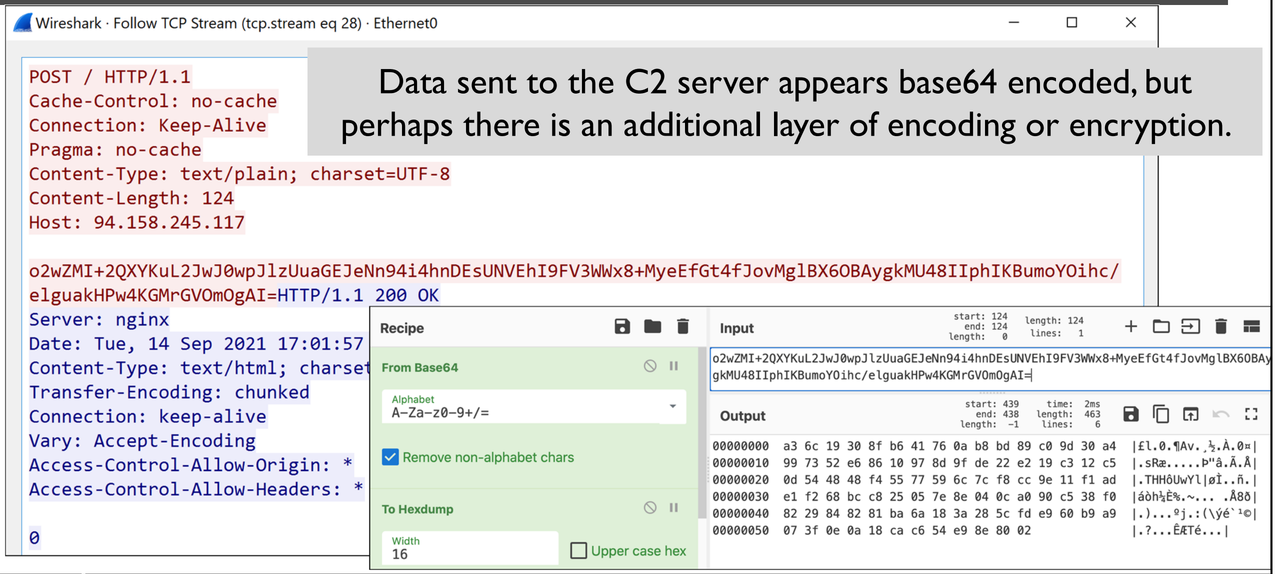


Communications with a command and control (C2) server are often encoded or encrypted to prevent eavesdropping. Using HTTPS is one well-known approach to encrypting traffic between a target and C2 server, but some malware families take a more straightforward approach of manually encoding communications.

The screenshot on this slide is from a malware sample allegedly used by the HIDDEN COBRA actor attributed to the North Korean government. The file is named INETSVC.EXE and has SHA-256 hash d1f3b9372a6be9c02430b6e4526202974179a674ce94fe22028d7212ae6be9e7. You can find this sample in your Malware\Section2 folder. For more information on this malware and threat actor, see https://for710.com/hiddencobra.

One simple approach to identifying potential encoding/decoding routines is to search for the XOR instruction mnemonic. This operation is often used to zero out a register, so we can ignore its use when the source and destination operand are identical. However, when the two operands are different, this *may* indicate encoding or decoding activity. Within Ghidra, use the file menu and browse to Search > Program Text. Choose the options shown on this slide and search all code for the XOR instruction mnemonic. There are 285 results, but most are cases where a register is zeroed out. In two results, the XOR operation is performed against one byte, which is common for encoding and decoding routines. Let’s investigate.

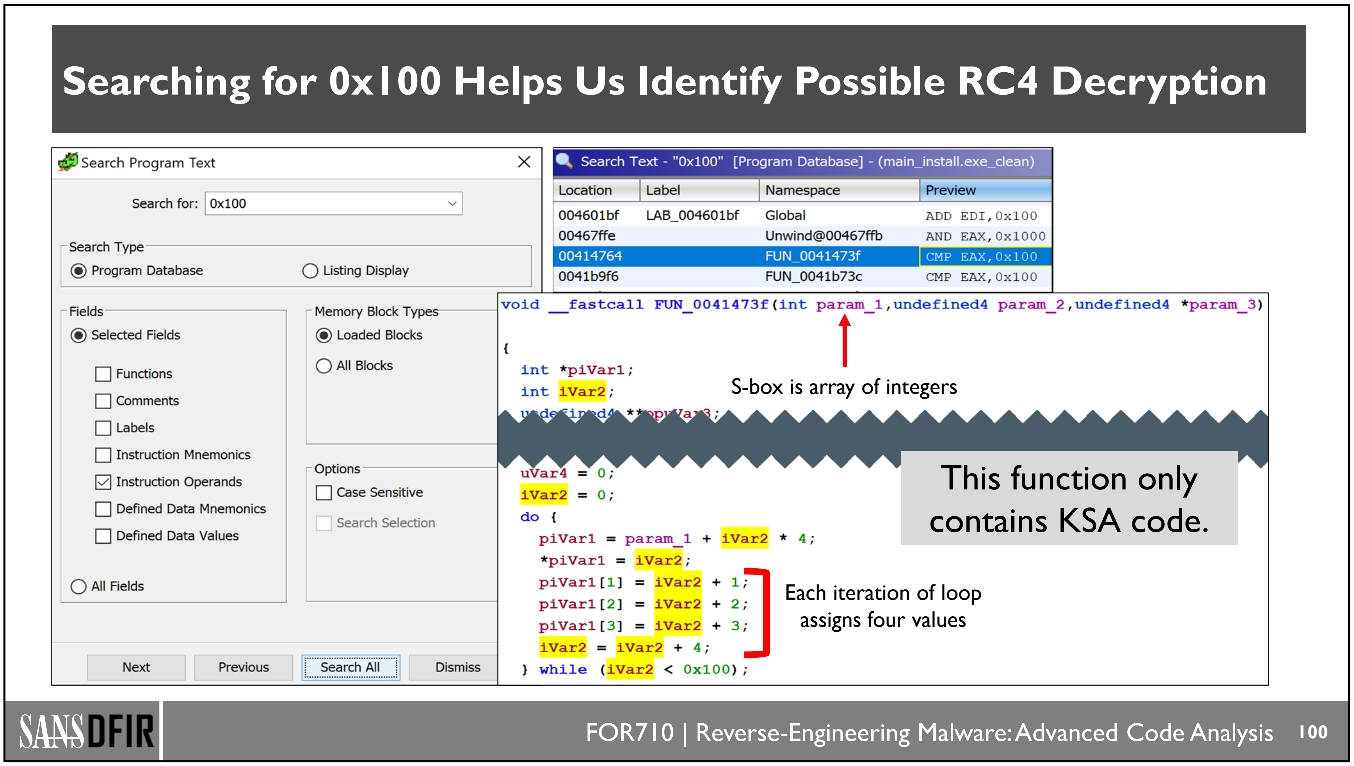
This Malware Has Multiple Layers of Encoding and Encryption to Communicate with a C2 Server



Some malware samples apply multiple layers of encoding and encryption within the malicious code to evade detection and analysis.

This discussion uses a sample named main_install.exe that belongs to the Raccoon family of infostealers. The sample has SHA-256 hash 15e98815867e8e122a0e75a6af0c0dbacace0fc09ab4023eeae360ca42bb92e9, and you can find it in your Malware\Section2 folder.

Upon execution, the program attempts to contact an IP address and sends some data via an HTTP POST request. The “=” padding at the end of the string hints at base64 encoding but decoding this content in CyberChef does not yield readable data. Perhaps the content is encrypted before it is base64 encoded. This would make sense as base64 encoding alone offers little protection.



As we did in the previous example, we can search for indicators of RC4 to assess if this program uses the algorithm. Searching for the 0x100 value returns multiple results. If we sort by "Preview", the first CMP instruction takes us to code in function FUN_0041473f that resembles the RC4 KSA function. There are a few differences, however, when we compare this code to the prior example. First, the S-box is an array of 4-byte integers instead of 1-byte values. This is a variation we may encounter in malware. Also, observe that each iteration of the loop assigns four values within the S-box.

Note that address of the S-box is passed via the first parameter. Because this function follows the `fastcall` calling convention, this value is placed into ECX before the function call (this is important for the discussion on the next slide).

If we scroll down to view the remaining code in FUN_0041473f, it is clear this function does not contain the PRGA code. This second component of RC4 encryption must be in a different function called after FUN_0041473f. Let's return to the code that calls FUN_0041473f.

Code Analysis Reveals a Function That Accepts the S-Box as an Argument

```
004148da    MOV     ECX, EDI                        Address of S-box
004148dc    CALL    FUN_0041473f                    RC4 KSA code

004148e1    PUSH    ESI
004148e2    SUB     ESP, 0x18
004148e5    LEA     EAX, [EBP + 0xc]
004148e8    MOV     ECX, ESP
004148ea    PUSH    EAX
004148eb    MOV     dword ptr [ECX + local_1c], EBX
004148ee    MOV     dword ptr [ECX + local_18], EBX
004148f1    CALL    FUN_0041295c

004148f6    PUSH    dword ptr [EBP + 0x8]
004148f9    MOV     ECX, EDI                        Address of S-box
004148fb    CALL    FUN_004147c3                    PRGA function?
```

- The S-box must be provided to the PRGA code for encryption.
- FUN_004147c3 appears to take the S-box as an input.
- We could use a debugger to confirm the S-box is passed to FUN_004147c3.

If we review the function that calls FUN_0041473f, we see addition function calls soon after the KSA code executes. Which of these upcoming functions might contain the PRGA code?

Recall that the PRGA must have access to the S-box to perform encryption or decryption. With this in mind, we can look for a function that accepts the S-box as an argument. Based on our analysis on the previous slide, we know the S-box address is passed to FUN_0041473f via ECX. Within the code on this slide, observe the instruction at 4148da that moves a value from EDI to ECX. If we highlight references to EDI, we can see the same value is placed into ECX at 4148f9 before the call to FUN_004147c3. Could this function contain the PRGA code?

FUN_004147c3 Likely Contains the RC4 PRGA Code

PRGA

```
i := 0
j := 0
while GeneratingOutput:
    i := (i + 1) mod 256
    j := (j + S[i]) mod 256
    swap values of S[i] and S[j]
    K := S[(S[i] + S[j]) mod 256]
    K ^ data byte
endwhile
```

We do not need to scrutinize each line of code to assess if the function likely contains PRGA code.

```
iVar5 = 0;
iVar9 = 0;
*(unaff_EBP + -0x10) = 0;
uVar6 = extraout_EDX;
if (0 < uVar1) {
  do {
    uVar6 = iVar5 + 1U & 0x800000ff;        <- mod 256
    if (uVar6 < 0) {
      uVar6 = (uVar6 - 1 | 0xffffff00) + 1;
    }
    iVar5 = *(unaff_EBP + -0x14);
    *(unaff_EBP + -0x18) = uVar6;
    iVar2 = *(iVar5 + uVar6 * 4);
    uVar6 = *(unaff_EBP + -0x10) + iVar2 & 0x800000ff;        <- mod 256
    [redacted]
    }
    uVar6 = *(*(unaff_EBP + -0x14) + iVar3 * 4) + iVar2 & 0x800000ff;        <- mod 256
    if (uVar6 < 0) {
      uVar6 = (uVar6 - 1 | 0xffffff00) + 1;
    }
    iVar2 = *(unaff_EBP + 0x24);
    bVar8 = *(*(unaff_EBP + -0x14) + uVar6 * 4) ^ *(iVar5 + iVar9);        <- XOR
    [redacted]
    iVar9 = iVar9 + 1;
    iVar5 = *(unaff_EBP + -0x18);
  } while (iVar9 < iVar2);
```


If we view FUN_004147c3, at first it may seem overwhelming. It definitely has more code than our prior example of an RC4 PRGA. However, if we review overarching flow of this function, we can identify a do-while loop. Within the loop, we also observe three mod operations and an XOR near the end of the loop.

Note that the AND operations highlighted on this slide appear different than the previous example. Specifically, the AND uses a value of 0x800000ff instead of 0xff. If you spent more time with this sample, you would determine that some of the additional code in this function evaluates if the array index is signed (i.e., the most significant bit is negative). It is unclear why this check is performed, and the result is that the AND operation with 0x800000ff is effectively the same as performing an AND operation with 0xff.

Combining these notes with our earlier observation that the S-box is passed to this function, we are likely looking at PRGA code. Let’s proceed to test this theory.

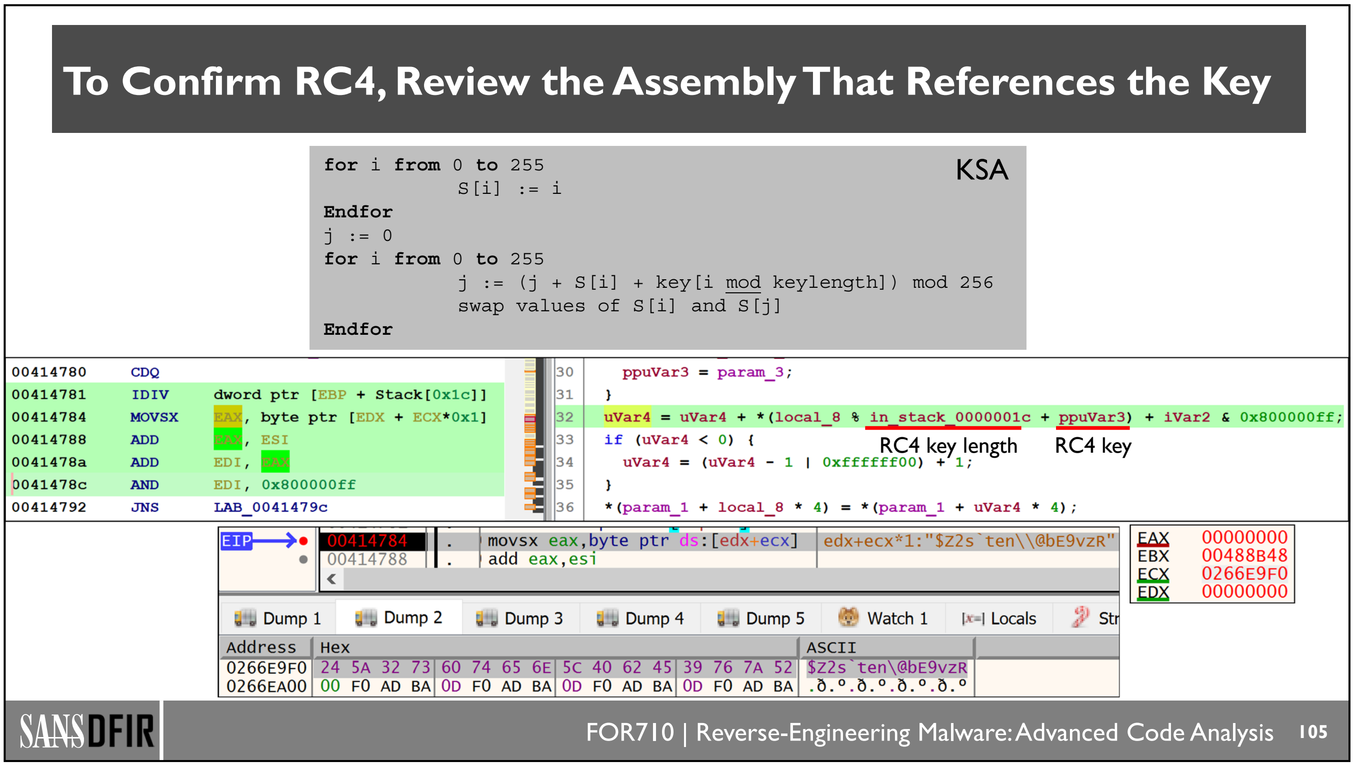
Allow the Function to Execute Until It Returns to View Decrypted Content

Several calls to the PRGA function reveal network IOCs.

Address	Hex																ASCII
0269EA28	68	74	74	70	73	3A	2F	2F	74	74	74	74	74	74	2E	6D	https://tttttt.m
0269EA38	65	2F	6A	64	69	61	6D	6F	6E	64	31	33	00	F0	AD	BA	e/jdiamond13.ð.º
0269EA48	AB	AB	AB	AB	AB	AB	AB	AB	00	00	00	00	00	00	00	00	««««««««........

Address	Hex																ASCII
0254E960	68	74	74	70	3A	2F	2F	39	34	2E	31	35	38	2E	32	34	http://94.158.24
0254E970	35	2E	31	31	37	2F	00	BA	0D	F0	AD	BA	0D	F0	AD	BA	5.117/.º.ð.º.ð.º

If we continue running the program and allow the PRGA function to execute until it returns, we observe a network IOC in the dump window. If we continue running the program and view future addressed of decoded content using the same approach, we can decrypt additional IOCs of interest as shown on this slide.



If we want to confirm that the target program performs RC4 decryption, we can perform a test using CyberChef as we demonstrated in the earlier example. This requires identifying a key and encrypted data.

The key is referenced in the KSA code, so we should return to FUN_0041473f to identify where it is located. If we review the decompiled code on this slide, we can apply our knowledge of the RC4 KSA to determine where the key and key length are referenced. In line 32 of the decompiled code on this slide, we understand that second operand in the mod operation specifies the key length, and the remainder is added to the RC4 key to arrive at the appropriate offset within the key.

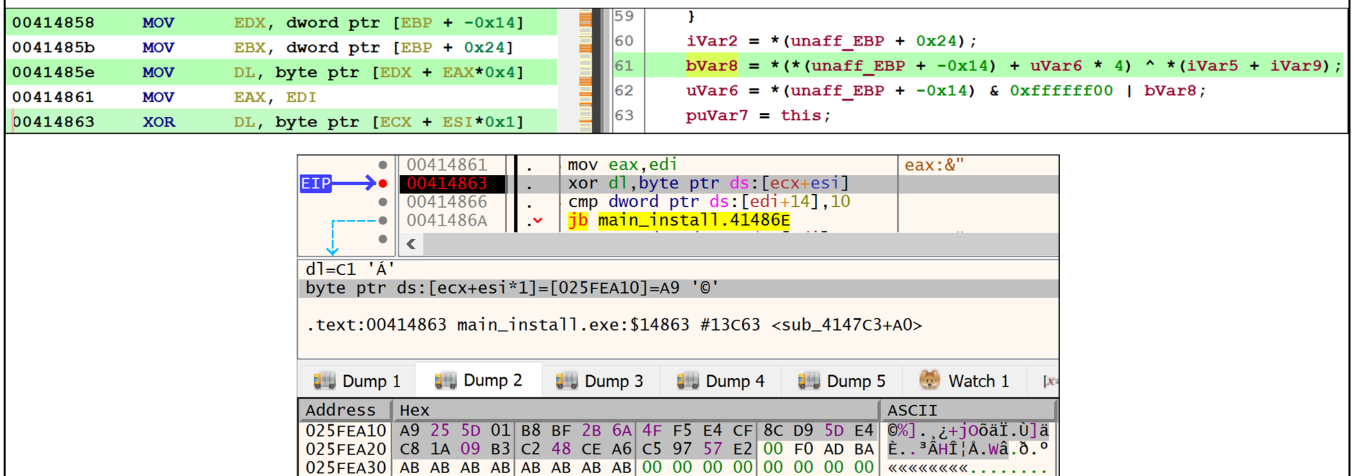
At 414781, the IDIV (signed divide) instruction can be treated as a DIV instruction. It places the remainder in EDX.

At 414784, if EDX contains a remainder that represents an offset (i.e., index) within the key, ECX must contain a pointer to the RC4 key.

In the x64dbg output shown on the bottom of this slide, we set a breakpoint on 414784 and run the program until that address. If we view a dump of the address stored in ECX, we can identify the key $Z2s`ten\@bE9vzR.

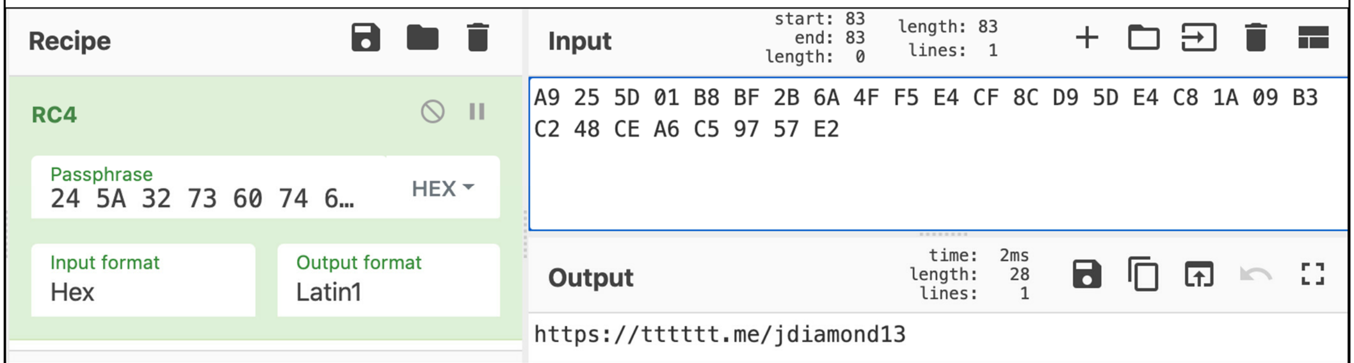
Note that the key appears to be null terminated. While not shown on this slide, we could confirm the key length by setting a breakpoint on the IDIV instruction and viewing the value of the operand. Debugging this code indicates the key length is 0x10, or decimal 16. This matches the length of the key mentioned above.

Next, We Need Encrypted Data to Test RC4 Decryption



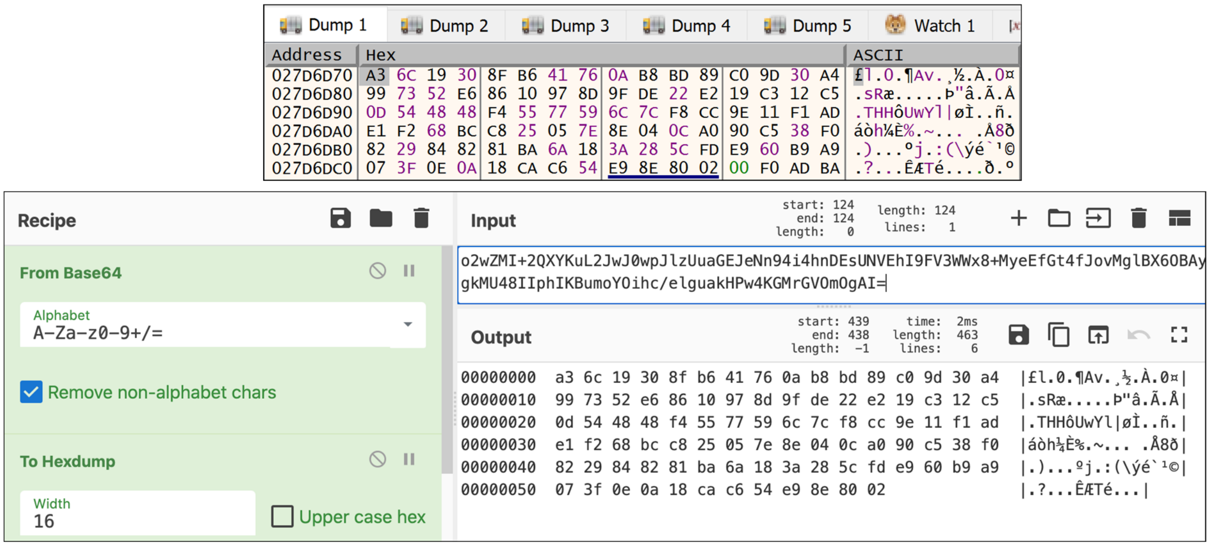
Now that we have an encryption key, we need some encrypted data to test RC4 decryption. Based on our knowledge of RC4, the encrypted data is referenced in the PRGA during the XOR operation. If set a breakpoint on the XOR operation and examine the data referenced by the pointer in the source operand, we see some unreadable null terminated data. Could this be RC4 encrypted contents?

CyberChef Confirms RC4 Decryption



We can now use CyberChef to confirm the presence of RC4 decryption. Inputting the key and encrypted data from earlier slides reveals the URL we encountered while debugging the decryption function.

The Encrypted Contents Match the Base64 Decoded Data We Identified in Network Traffic



The encrypted content matches the base64 decoded data we identified earlier. As a reminder, this content was sent to a server via an HTTP POST request. This confirms this sample likely uses RC4 encryption to protect its command and control (C2) traffic.

Course Roadmap

- FOR710.1: Code Deobfuscation and Execution
- FOR710.2: Encryption in Malware
- FOR710.3: Automating Malware Analysis
- FOR710.4: Correlating Malware and Building Rules

SECTION 2

- Encryption Essentials
  - Lab 2.1: Encryption Essentials: Quiz
- File Encryption and Key Protection
  - Lab 2.2: Identifying File Encryption and Key Protection in Ransomware
- Data Encryption in Malware
  - ***Lab 2.3: Analyzing Data Encryption in Malware***

This page intentionally left blank.

Lab 2.3

Analyzing Data Encryption in Malware

Please begin Lab 2.3 now.

Data Encryption in Malware: Module Objectives, Revisited

✓Understand common use cases for data encryption in malware.

✓Gain familiarity with the critical aspects of RC4, a common symmetric cipher for encrypting data in malware.

✓Learn how to identify the use of RC4 in malware.

✓Debug malware to decrypt relevant data.

✓Identify the RC4 key and length in malware to decrypt content outside of the target program.

This slide describes the objectives of the module we just completed.

Course Roadmap

- FOR710.1: Code Deobfuscation and Execution
- **FOR710.2: Encryption in Malware**
- FOR710.3: Automating Malware Analysis
- FOR710.4: Correlating Malware and Building Rules

SECTION 2

- Encryption Essentials
  - Lab 2.1: Encryption Essentials: Quiz
- File Encryption and Key Protection
  - Lab 2.2: Identifying File Encryption and Key Protection in Ransomware
- Data Encryption in Malware
  - Lab 2.3: Analyzing Data Encryption in Malware

SANS DFIR

This page intentionally left blank.