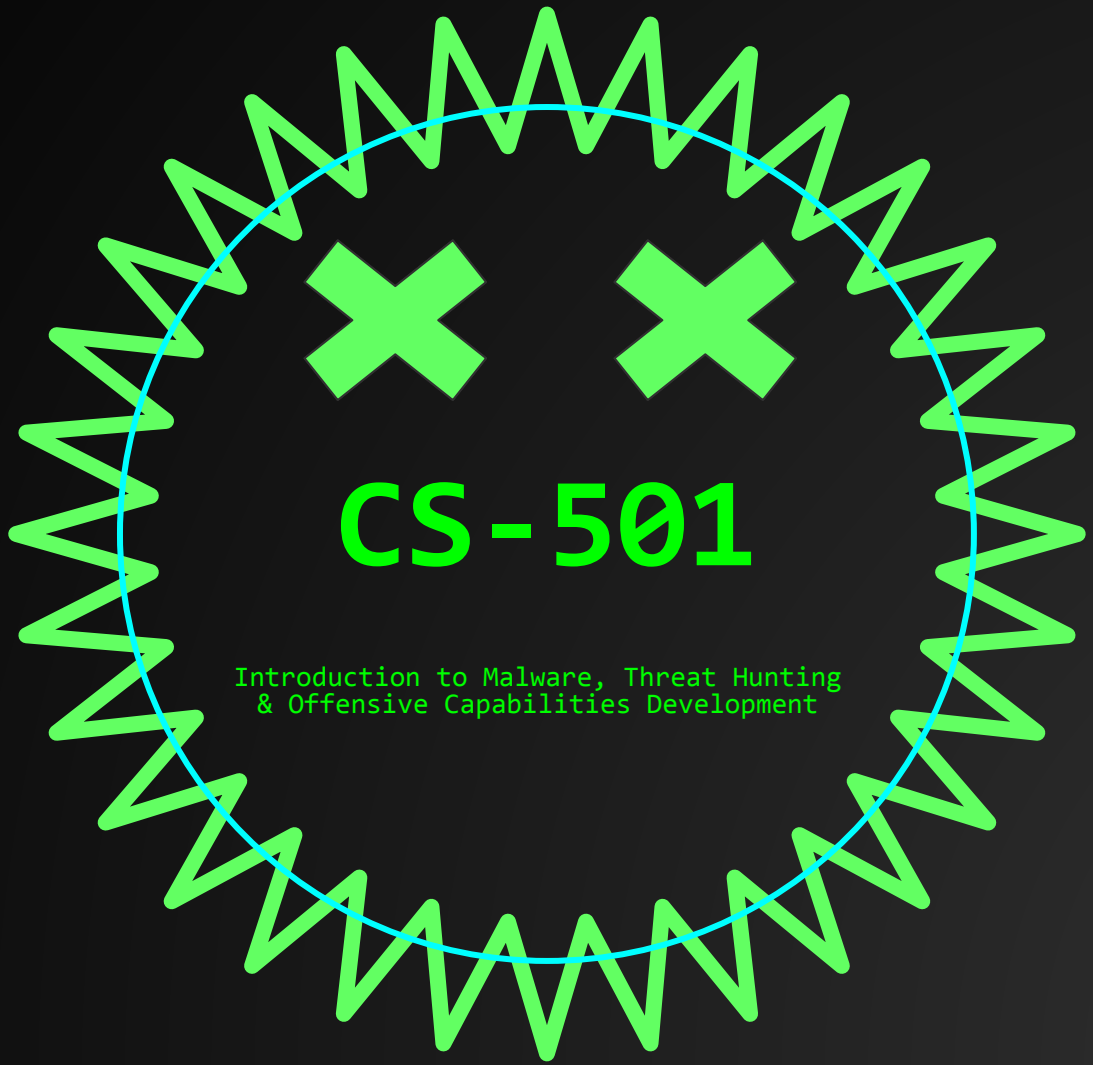


CS-501

Introduction to Malware, Threat Hunting
& Offensive Capabilities Development

Lecture 15
Symmetric Crypto

Ransomware

- Malware that encrypts sensitive data, and extorts victims for the decryption key
- If done properly, the data is at the mercy of the ransomware operator

How does ransomware work?

- Enumerate files
- Filter files that are sensitive/not critical to OS function
- Encrypt the files
- Make your presence known (Leave a ransom note)
- Extract payment, and provide decryptor

Background we are going to need

A crash course in cryptography!

Basic Definitions

Plaintext: the message transmitted

Encryption: A reversible algorithm designed to provide privacy between two communicating parties.

Ciphertext: the result of encrypting a plaintext using an encryption algorithm

Computationally bounded adversary: an adversary with limited resources

Security and Communication

What does it mean for a method of communication to be “secure”?

Well, *secure from what?*

Communicating over an “Unsafe” Channel

When talking about security, it is essential to to describe the type of Adversary you are secure from!

Over the past few decades, Cryptography has evolved into an offshoot of Complexity Theory.

Security definitions are defined in terms of the *adversary* that cannot invalidate some property given their abilities.

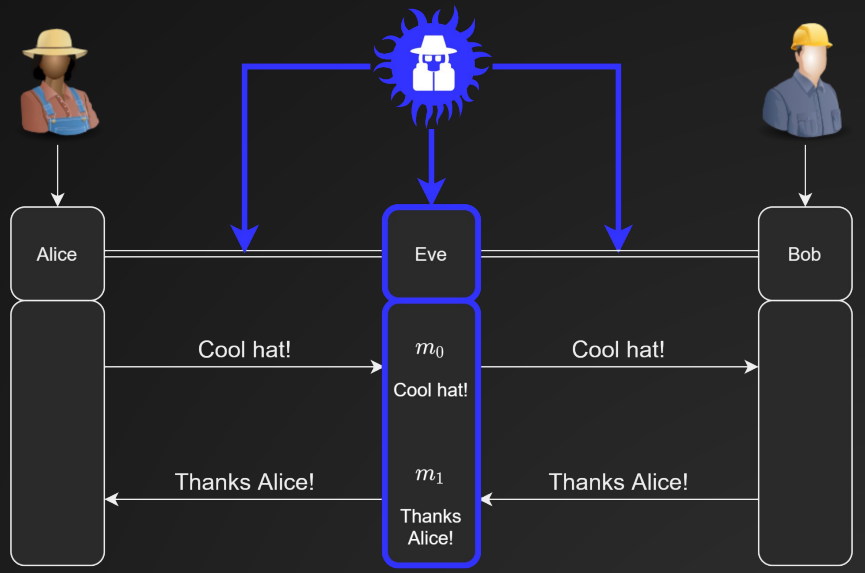
Symmetric Cryptography (simple case)

Cryptographic protocols between two parties that have agreed on a shared secret.

Scenario 1: Eve

Consider two Entities “Alice” and “Bob” that wish to communicate over a reliable communication channel.

Suppose that an adversary *Eve* is able to install a *tap* in their channel, and can eavesdrop on all messages exchanged



Informal Definitions: Secure Against Eve

When sending data across a communication channel, an adversary who is able to position themselves as a tap in the network should not be able to learn anything “meaningful” about the contents of the data they see.

In particular, even if they were able to view the encryption of some polynomial number of messages, they should not be able to distinguish encrypted data from random noise

I.e. I.I.D. Bernoulli(p=.5)

Informal Definitions: Secure Against Eve

Cryptographic Primitive to Enable this: Encryption

See Chosen Plaintext Attack (CPA) security for a more rigorous treatment of this.

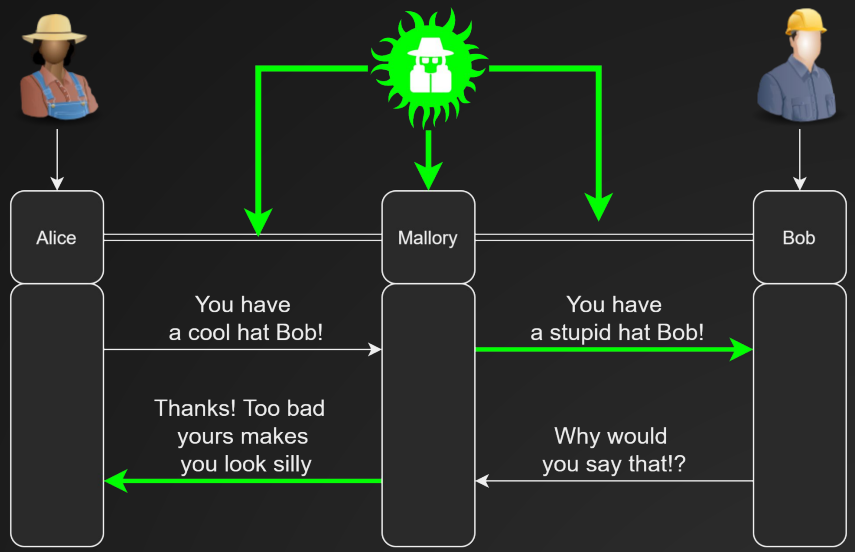
Limitations of Eve

- Eve is only able to sniff the contents of messages.
- In the event that Eve is discovered as Tap by one party, say Bob, they are unable to prevent the other prevent Bob from warning Alice
- In the context of an implant and a C2 server, it might not matter that Eve can see our communication if we can achieve a particular goal unhindered

Scenario 2: Mallory

Consider now an adversary that has all the powers of Eve and one addition:

- Mallory is able to modify messages in transit!
- This could involve dropping, modifying and/or replaying messages



Secure against Mallory

When Mallory is able to position themself as an active MITM,

1) Alice and Bob should be able to detect when a message has been modified
2) Alice and Bob should be able to detect when a message has been replayed
3) Alice and Bob should be able to detect when a message been dropped
4) Mallory should be (computationally) unable to forge a message that Alice and Bob will verify
5)

Cryptographic primitive to enable this: Message Authentication Codes (MACs)

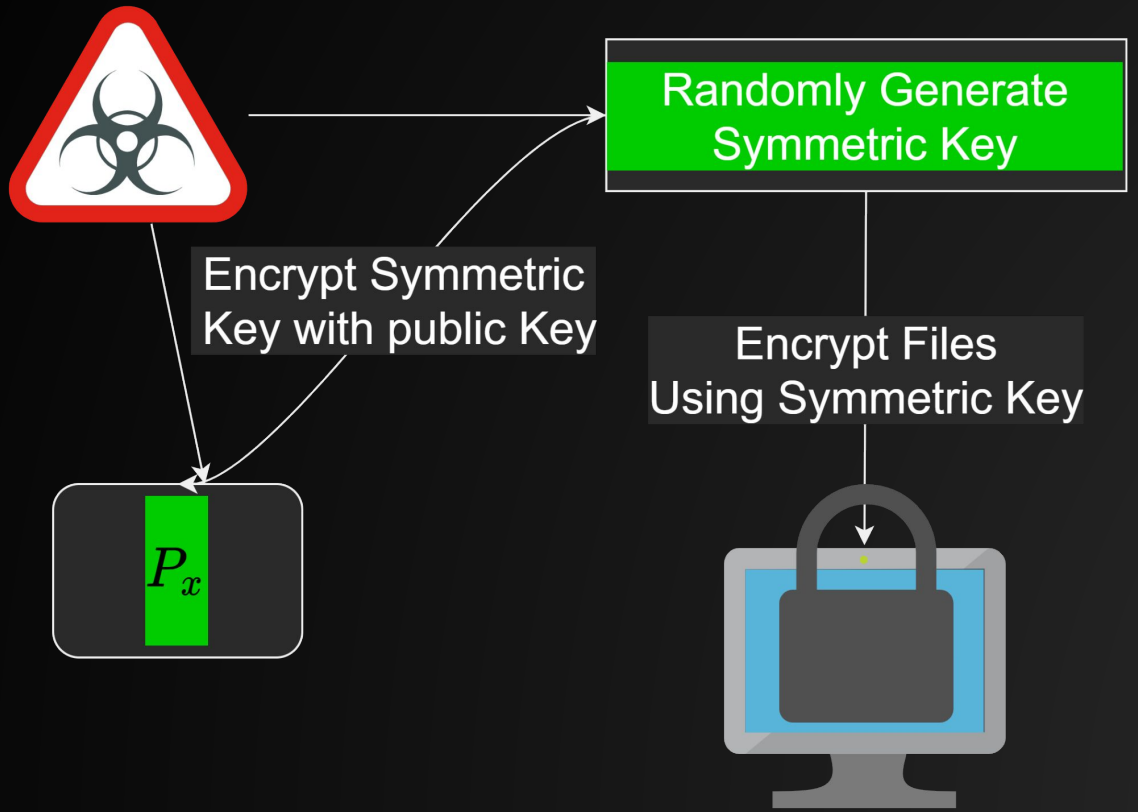
For a more rigorous treatment, see existential unforgeability

https://cseweb.ucsd.edu/~mihir/papers/gb.pdf

Summary:

- Privacy: nobody can read the contents of your messages
- Integrity: if anyone modifies/spoofs a message in transit, you will detect it

Ransomware


Randomly Generate Symmetric Key
Encrypt Symmetric Key with public Key
Encrypt Files Using Symmetric Key
P_x

Caesar Cipher

Start with an alphabet

Messages are composed of characters contained in the alphabet

A secret key k is a random value between 0 and len(alphabet)-1

The ciphertext is created by shifting each character of the alphabet k positions clockwise after assigning every character in the alphabet with a position on a clock.

The plaintext is recovered by shifting each character counter clockwise k positions

Security of the Caesar Cipher

When the length of the message > 1: it is not great

We can check all possible keys if the size of the alphabet is reasonable

I.e., it is vulnerable to a brute force search

One Time Pad

Message: bytes of length n

Key: random bytes of length n

Encrypt: compute ciphertext= message xor key

Decrypt: compute plaintext = ciphertext xor key

Limitations of the One Time Pad

- As the name suggests, you should only use the random byte stream once
- Encrypting large volumes of data requires a large pool of entropy
- Ciphertexts are “malleable”

Note:

Remember, any data that you leave in a binary can be recovered by a skilled reverse engineer

In other words, you should assume that they get leaked!

Symmetric Cryptography (simple case)

Cryptographic protocols between two parties that have agreed on a shared secret.

Perfect Secrecy (informal)

- No amount of computational power can give you odds better than just guessing
- Only the One-time pad has this property

Security: Computationally Bounded Adversary

The “advantage” an adversary gets in predicting your plaintext after seeing it’s ciphertext is exponentially small (negligible)

A more rigorous treatment of this topic is beyond the scope of our class, but the idea here is we want to create an algorithm that without knowledge of the secret key, the adversary has a negligible change of recovering the plaintext.

Security Vs Obscurity

As we will see, it is usually easy to identify implementation of common block ciphers and stream ciphers. In the context of extracting configuration data from an implant, is it better to use a robust, easy to identify cryptographic primitive, or an ad hoc weaker primitive?

Threat model: Malicious OS

From the perspective of the malware author, when leveraging cryptography we must assume that the OS the implant is executing on is malicious.

Stream Ciphers

- A cryptographic algorithm that acts a lot like the One time pad, with the exception that the stream of pseudo-random bytes are generated by using a cryptographically secure Pseudo Random Number Generator (csPRNG)
- Stream Ciphers are Synchronous

Common Stream Ciphers

RC4

Salsa20

ChaCha20

Block Ciphers

Encrypt data in chunks called blocks

If the size of the data is not a multiple of the block size, it needs to be “padded”

If the size of the data is larger than the block size, the algorithm needs a “mode of operation”

Those two requirements have created a pretty massive headache for cryptographers and are hard to get right.

Common Block Ciphers

Rijndael/AES

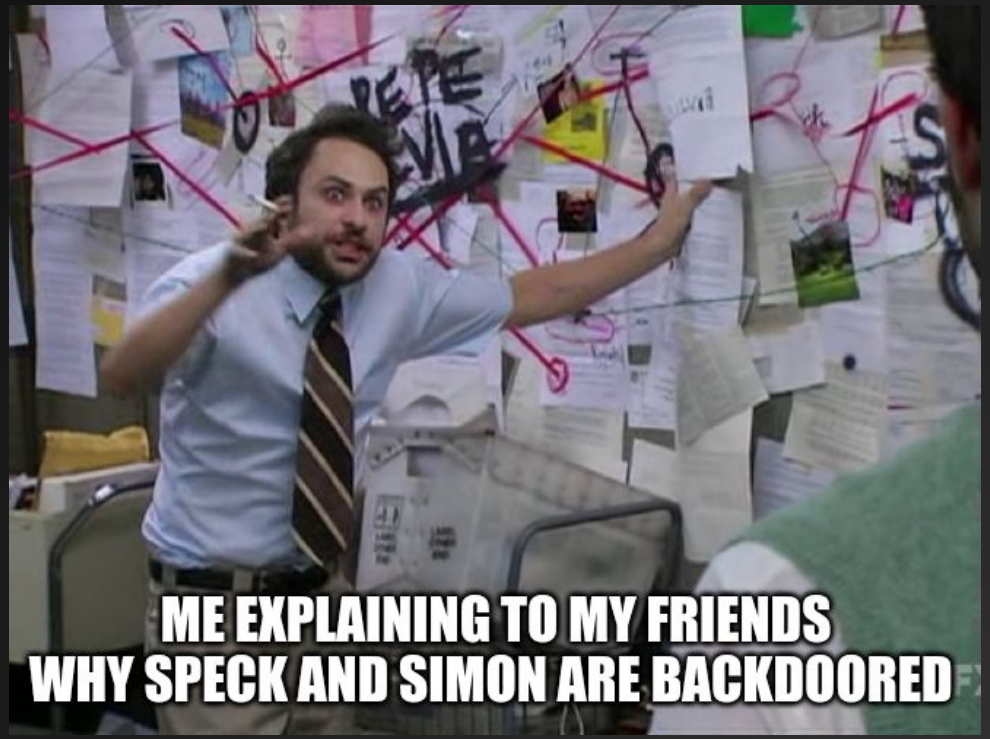
Blowfish

TEA/xTEA

SIMON/SPECK (Sus :-)

DES 3DES

RC5



Common Mistakes

Messing up the padding. There are a lot of ways to do this.

Using ECB mode for a block cipher

Reusing nonces in CBC mode

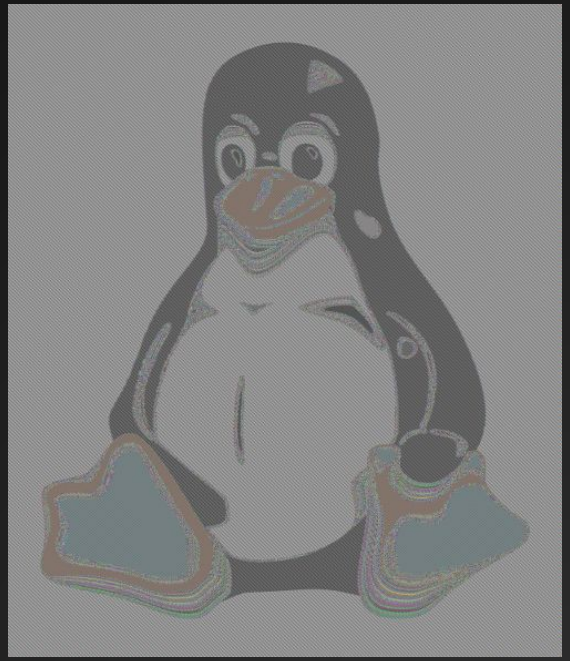
Using weak algorithms (rot ciphers, substitution ciphers)

Inadequate authentication to the server

Inadequate server authentication to the client

Failure to use MACs

Using weak hash functions



Practical Attacks on Crypto

This class will focus on attacking crypto used by malware from the perspective of a reverse engineer.

In particular, we will look at how we can recover keys, replay messages, inject our own data, and decrypt traffic.

Recovering the Key From the Malware

Statically: hard(ish?)

Dynamically: usually easy. Find the function that encrypts/decrypts and just set a breakpoint.

How do you find that function? Set breakpoints on common WinCrypt API functions, look for constants, trace your way backwards after network communication...etc

Example: RC4

RC4 is very likely broken.

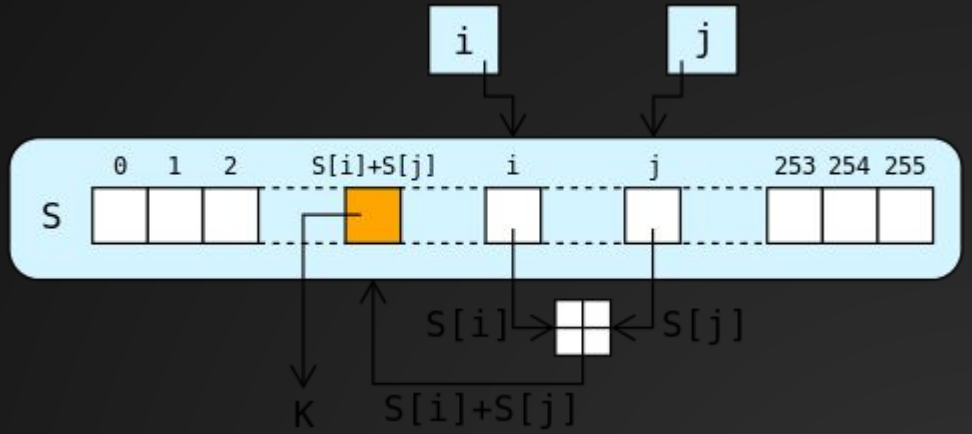
It should not be used in any production environment and was removed from TLS. So why are we talking about it?

Well, RC4 is an incredibly simple cipher to implement, and a fan favorite of malware authors.

RC4

- Key scheduling algorithm
- Stream generation

https://github.com/kbsec/misc_crypto/blob/master/rc4/rc4.py



RC4: Rivest Cipher 4

- Stream cipher commonly used by malware authors due to its ease of implementation
  - Cryptography poses an engineering challenge, as any code you write client side needs to have a counterpart server side. Implementing RC4 is trivial in any language!
- Two components:
  - Key-Scheduling Algorithm (KSA)
  - Pseudo Random Generation Algorithm
- PRNG that generates random bytes from an initial seed (the key)
- Algorithm is synchronous, meaning Client and server need to maintain state information.

KSA

- Initialize an array S of 256 bytes where each byte is set to its index in the array
- We then use the key to create a random permutation of 256 bytes
- That is, we can view S as defining a function, where the input is the index in the array, and the output is the value stored in the array
- That is, initially, S[i] = i for all i=0,...,255, S[input] = output is the identity function
- Using the Key, we randomly swap values inside of the S array
- S is usually called an “S-box”

Generating Psuedo random bytes

Pull data from the Sbox according to the algorithm and update.

Looking at RC4 in Assembly

- Let’s spend some time implementing it!
- We can compile the end result with the -g flag to generate a PDB
- Load it into Ghidra and see if we can identify the relevant portions

Demo

Identifying Common Crypto

- Constants
- Common Round Functions
- Example, using a ghidra plugin to identify sha256 in crackme 4.

https://github.com/TorgoTorgo/ghidra-findcrypt

Integrity Vs Privacy

- Privacy: nobody can read what your messages are
- Integrity: if anyone modifies your data in transit, you will detect it

Message Authentication Codes

- Symmetric protocol to detect data corruption/tampering
- Simplest example of this is HMAC (Hash MAC)

HMAC: Fun fact, invented by a BU professor: Ran Canetti

What should you use for C2 Sessions?

Generally Believed to be secure

AES-GCM

CHACHA20-Poly1305

Remember, all of this is moot if they simple pull the symmetric key out of your binary or dump it from memory. You should assume your session keys will eventually get leaked the same way you should plan for your implant eventually getting caught!

Solid Libraries

OpenSSL, LibreSSL, python Cryptography, Rust Crypto, LibSodium, Monocypher, NACL

For your implant, I would recommend using a combination if Windows Crypto API functions along with a statically linked cryptography library. OpenSSL is very large, as is LibreSSL. An experimental library that is targeted for embedded devices is LibHydrogen and WolfCrypt are good options.

I have also personally made use of https://github.com/jedisct1/libhydrogen/tree/v0 (old branch)

Note that the cryptography you use is something that can be used to identify you

What is missing?

Problem: If your key gets leaked, then all previous communications are also easily decrypted.

Problem: How do we get keys distributed?

Problem: How do we authenticate to the server? How does the server authenticate to us?

Opsec Considerations

- Using custom crypto can identify you
- Your implant is executing on a malicious OS. This makes privacy very difficult.
- Your goal should be to blind the defenders until you have achieved your goal
- Cryptography can hide your data, but it is harder to hide the fact that you are using cryptography!

KASPERSKY

27

16. What kind of encryption algorithms are used by the EQUATION group?

The Equation group uses the RC5 and RC6 encryption algorithms quite extensively throughout their creations. They also use simple XOR, substitution tables, RC4 and AES.

RC5 and RC6 are two encryption algorithms designed by Ronald Rivest in 1994 and 1998. They are very similar to each other, with RC6 introducing an additional multiplication in the cypher to make it more resistant. Both cyphers use the same key setup mechanism and the same magical constants named P and Q.

The RC5/6 implementation from Equation group's malware is particularly interesting and deserves special attention because of its specifics.

```
.10010119: C745F884000000    mov     d,[ebp][-8],000000084 ;'    ä'
.10010120: C7006351E1B7      mov     d,[eax],0B7E15163 ;'▒8Qc'
.10010126: 41                inc     ecx
.10010127: 8B5488FC          mov     edx,[eax][ecx]*4[-4]
.1001012B: 81EA4786C861      sub     edx,061C88647 ;'a╩╪G'
.10010131: 891488            mov     [eax][ecx]*4,edx
.10010134: 41                inc     ecx
.10010135: 83F92C            cmp     ecx,02C ;','
.10010138: 7CED              jl      .010010127 --↑2
.1001013A: 33D2              xor     edx,edx
.1001013C: 33DB              xor     ebx,ebx
.1001013E: 8955FC            mov     [ebp][-4],edx
.10010141: 33FF              xor     edi,edi
.10010143: EB03              jmps    .010010148 --↓3
.10010145: 8B4508            mov     eax,[ebp][8]
.10010148: 8B75FC            mov     esi,[ebp][-4]
```

Encryption-related code in a DoubleFantasy sample