CRYPTOGRAPHY

HASHING, ENCRYPTION, AND ALGORITHMS

LOGISTICS

Class Hours:

- Instructor will set class start and end times.
- There will be regular breaks in class.

Telecommunication:

- Turn off or set electronic devices to silent (not vibrate)
- Reading or attending to devices can be distracting to other students
- Try to delay until breaks or after class

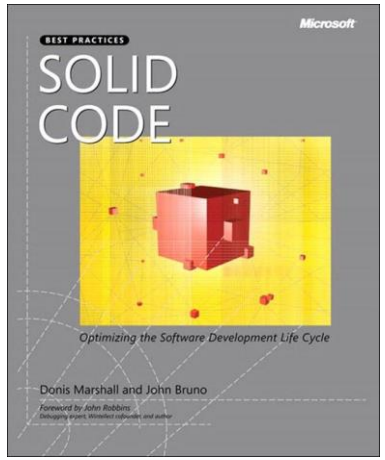
Miscellaneous:

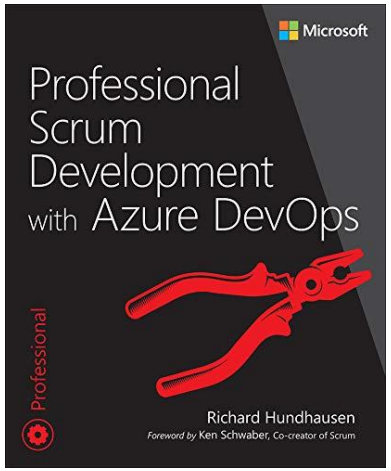
- Courseware
- Bathroom
- Fire drills

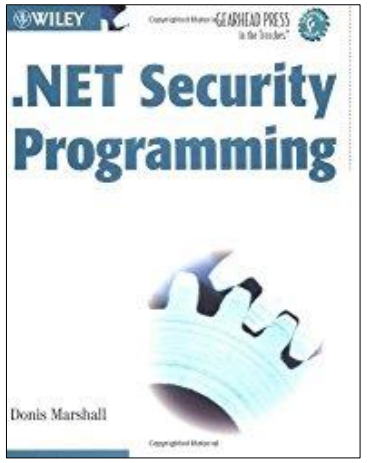
DONIS MARSHALL

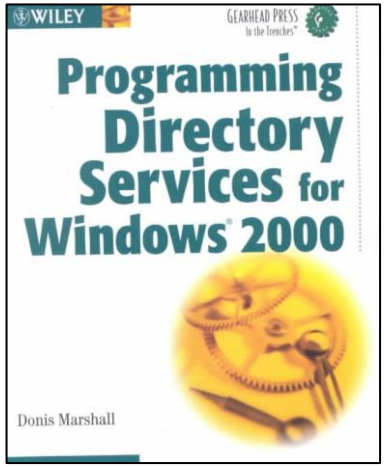
Security Professional
Microsoft MVP
Microsoft Certified
Cryptography Artisan
Author

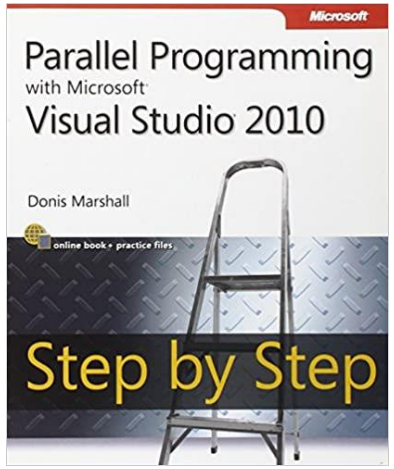
dmarshall@innovationinsoftware.com

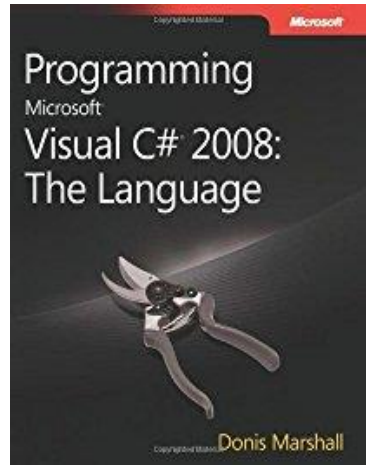












INTRODUCE YOURSELF

Time to introduce yourself:

- Name
- What is your role in the organization
- Indicate cryptography experience

INTRODUCTION

According to the Handbook of Theoretical Computer Science, this is the definition of Cryptography :

Cryptography or cryptology (from Ancient Greek: **κρυπτός**, translit. kryptós "hidden, secret"; and **γράφειν** graphein, "to write", or **-λογία** -logia, "study") is the practice and study of techniques for secure communication in the presence of third parties called adversaries.

The goal of cryptography is protecting sensitive data or resources from tampering or disclosure. The two most important components of cryptography are hashing and encryption:

- Hashing creates a fixed length digest used to detect tampering.
- Encryption encrypts data to prevent disclosure of secrets. Later, the data can be decrypted to reveal the original information.

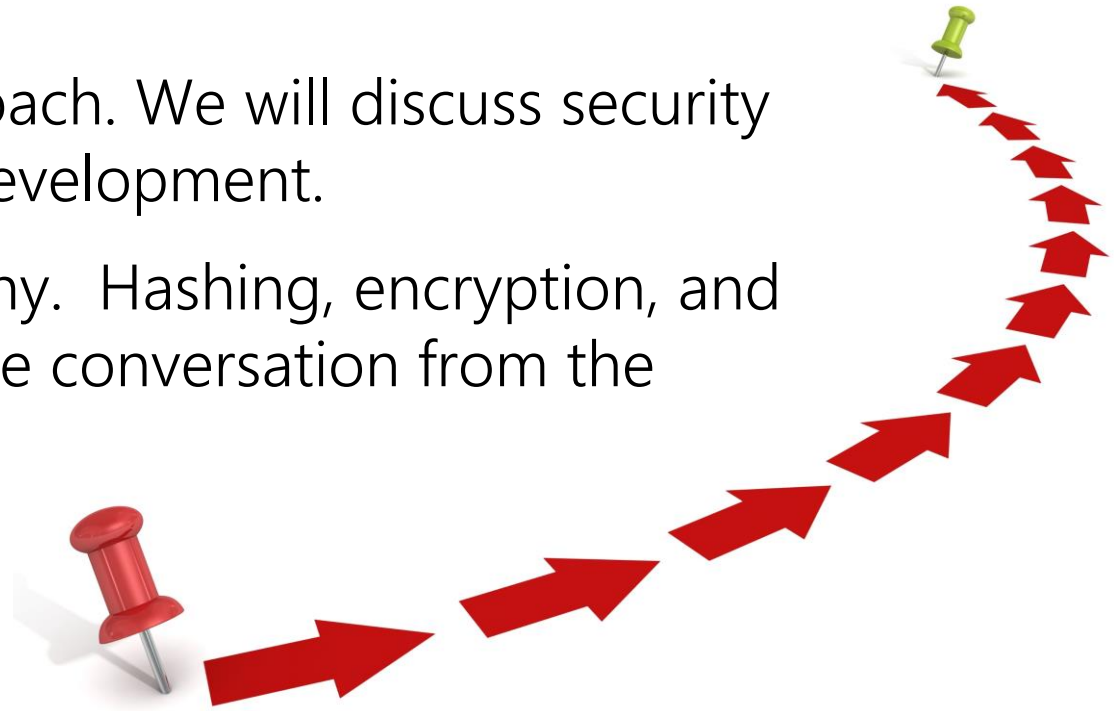
END TO END

"Security" should not be the final task at the end of product development, but an end-to-end endeavor.

Microsoft Secure Development Lifecycle (SDL) maps cybersecurity onto each phase of the traditional product development lifecycle.

This class takes the same approach. We will discuss security during all aspects of product development.

This also applies to cryptography. Hashing, encryption, and algorithms should be part of the conversation from the beginning.



ATTACKERS GOAL

The most frequent goal of the attackers is data. This includes:

- Identity theft
- Stealing data
- Modifying data
- Ransomware
- Information disclosure

Cryptography can mitigate these attack and more.

STANDARDS AND PRACTICES

In this class, we will discuss various algorithms, techniques, and standards. This includes occasional recommendations. However, the most important recommendation is from your organizations. Many organization publish a standards and practices document for cryptography. Refer to this document for the standards for your organization.

If standards and practices document does not exist, we recommend that you work within your team to establish some guidelines.

NOT A MATH CLASS

Everyone within an organization is responsible for security, not just developers and architects.

This includes:

- Executive team
- Management
- Designers
- Operations
- DevOps
- Testers

A competent security strategy spans the entire software development life cycle. For this reason, only a collaborative and iterative approach towards security is likely to be successful.

LIFECYCLE

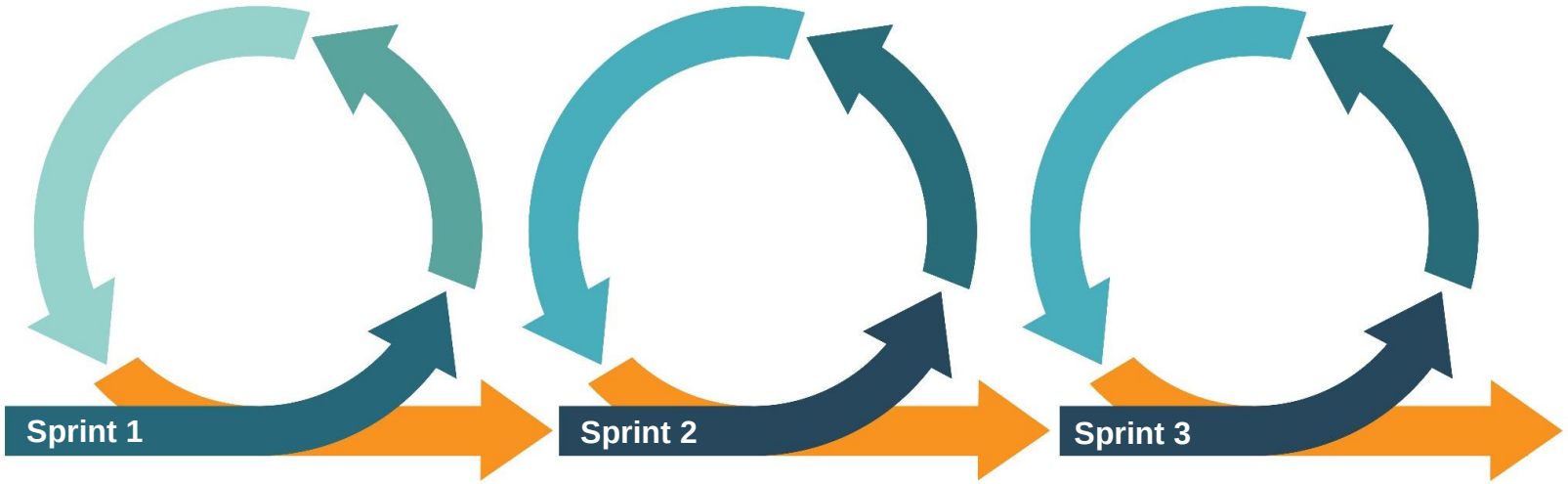
Cryptography should be a consideration throughout the product lifecycle – end to end. This is defined in the Microsoft Security Development Lifecycle (SDL).

- Requirements phase. Security features are decided in this phase.
- Design phase. The design phase is where architectural decisions are made, such as the proper security algorithm.
- Implementation phase. The cryptography decisions are implemented by the development team.

LIFECYCLE - 2

- Deployment phase. In the deployment phase, verified certificates are installed, secure configuration is implemented, and other tasks are completed.

After the release, the development team must remain current on the latest in cryptography, suspicious behavior, and monitoring for Zero-day attacks.



ADVERSARIAL

For security modeling, understanding the adversarial perspective is the best approach. Apply this approach to all aspects of threat modeling:

- Attack surface
- Vulnerabilities
- Threats

Thinking like an adversary may not be natural. You are the good guys! However, thinking like that is one of the primary goals of this class.

VULNERABILITIES

Hackers don't attack just *anywhere*. First they identify vulnerabilities. Vulnerabilities are attack points on an attack surface. These are often the target of attackers. For that reason, vulnerabilities are usually the pivot point in security analysis, even with cryptography. As such, it is also an important topic in this class.

THINK SECURITY

Security is more than a technical set of requirements or policies that must be adhered to or implemented. It is a frame of mind.

Application security is a perpetual game of chess. Unlike chess however, you have multiple adversaries. For success, you must be able to "see" two or three moves ahead of every adversary.

This requires vision and the ability to *think* security.

SECURITY STRONG

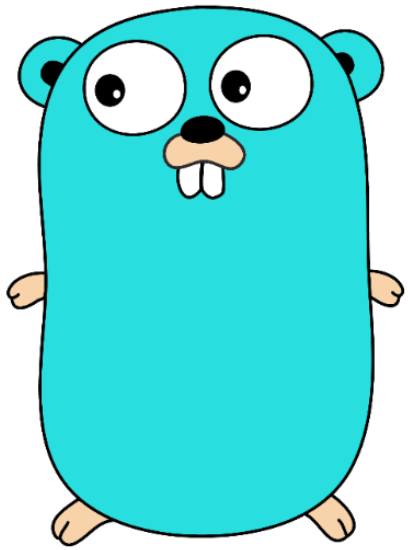
The goal of this course is creating secure application. Cryptography is an additional tool in your toolbox for securing systems.

Combined with other tools such as fuzzing, defense in depth, security scanning, attack trees, and more your systems will be “security strong”.

ATTACKS

At the end of each module, a cyberattack is introduced. Reviewing security attacks will help you better understand and adopt the adversarial perspective.

GOLANG / PYTHON



Code examples are presented in Go and Python language.

There are various IDEs (integrated development environment) available for both programming languages. For class, use your preferred IDE.

There are also several online playgrounds. Here are two:

- https://go.dev/play/
- https://replit.com/languages/python3

YOUR CLASS!

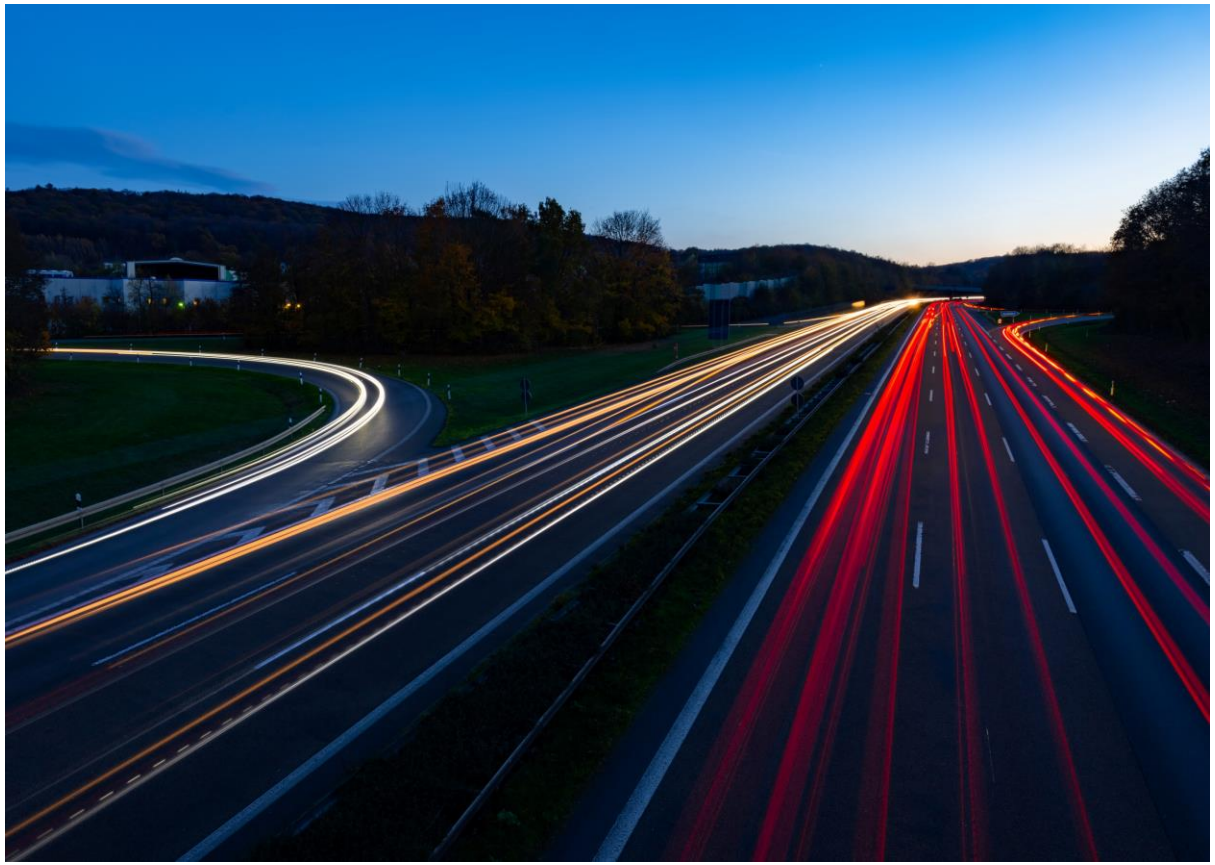
Yes, this is your class. What does this mean? You define the value.

- What is the most important ingredient of class – your participation!
- Your feedback and questions are always welcomed.
- There is no protocol in class. Speak up anytime!
- We welcome your comments during and after class. Just email dmarshall@innovationinsoftware.com.

COURSE AGENDA

1. Vulnerabilities
2. Crypto concepts
3. Crypto terminology
4. Algorithms
5. Hashing
6. Passwords
7. Encryption
8. Digital signatures
9. Adversarial perspective
10. Attacks

SIDE CHANNEL ATTACK



Side channel attacks are often the most creative sort of exploitation. A side channel attack exploits the unintended information exhaust of an application or device. For example, the electrical emissions of a computer monitor or integrated chip (IC). Rowhammer attacks are an example of a side channel attack.

POP QUIZ: SIDE EFFECTS

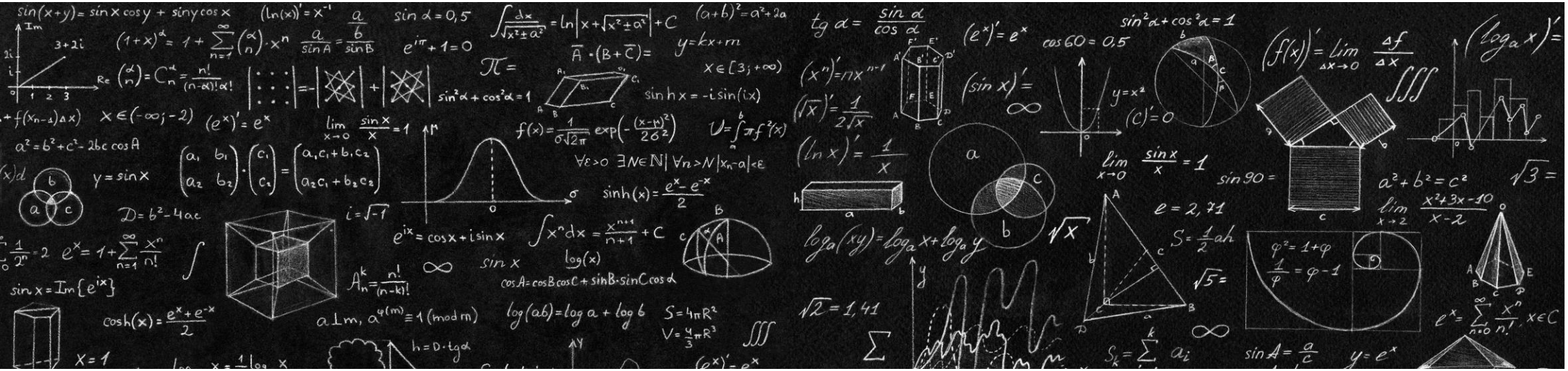
If standing adjacent to a highway, what can you learn or discern from the side effects of being near the thoroughfare?

5 MINUTES

SIDE CHANNEL ATTACK - 2

When securing applications, the focus is on inputs and outputs, the conventional attack points. This is the traditional approach even for hardware and UEFI.

However, hardware exists at a physical level. Computers can sometimes be attacked based on physical side effects, such as power, time, and sound. Hackers use Side Channel attacks to interpret these side effects. They can then obtain sensitive data and even secrets.

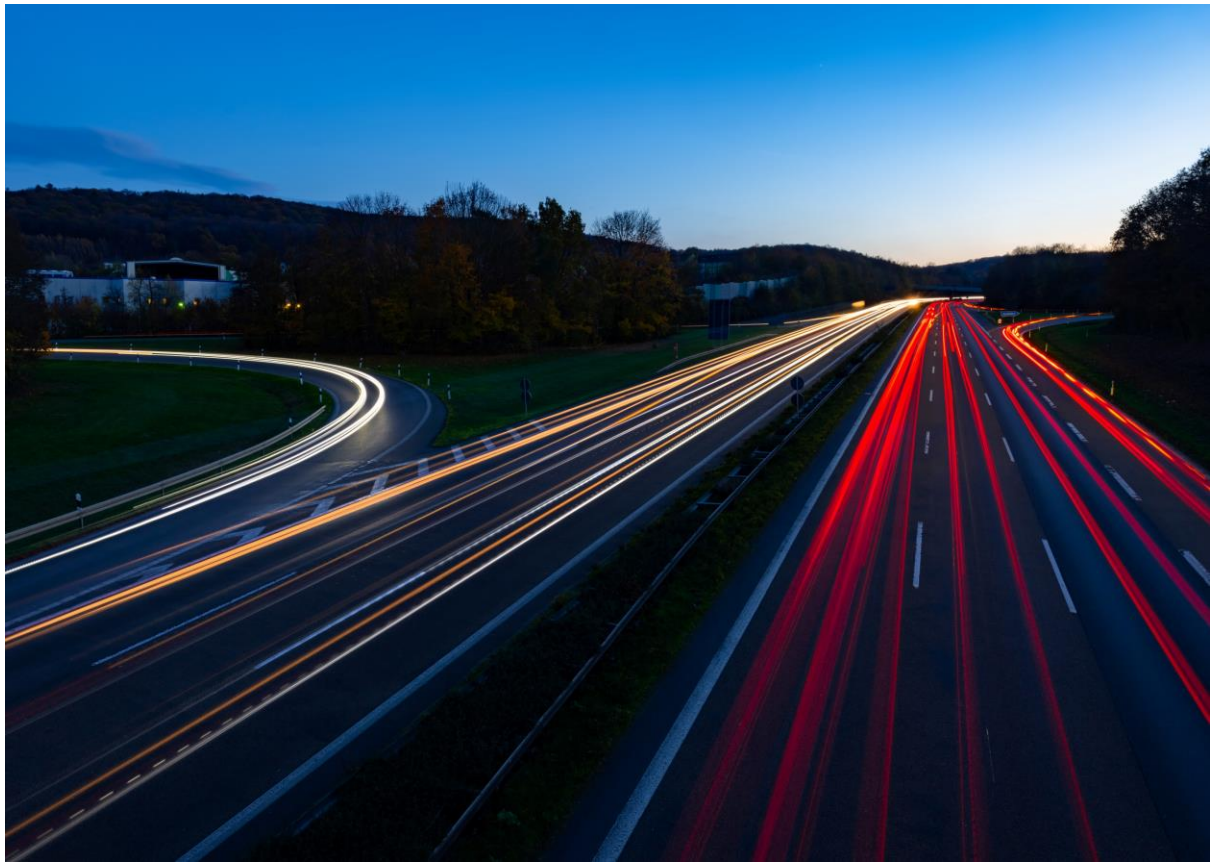


SIDE CHANNEL ATTACK - 3

The most common Side Channel attacks are:

- Timing attack. An attack that measures the amount of time to complete a calculation or algorithm.
- Power analysis attack. An attack that measures power consumption for various calculations, algorithms, or activities.
- Electromagnetic attack. An attack that measures leaked electromagnetic radiation.
- Acoustic attack. An attack that measures amount or deviation of sound related to a calculation, algorithm, or activity.
- And many more

SIDE CHANNEL ATTACK - 4



Tel Aviv University has done significant research on side channel attacks. For example:

https://bit.ly/3xOfuLt

What offers effective protection against a side channel attack? Cryptography!

LAB 1 – SIMPLE

SIMPLE

Most programming classes start with a “Hello, world” example application. Let’s start there also.

Several programming languages have a simple method for creating a hash.

In this lab, we will hash “Hello, world” with that method

PYTHON

Create a Python project.

Within the project, create new source file.

Store the string “Hello, World!” into a variable

Convert the variable into a hash using the hash function. Save the results in a new variable

```
variable=hash(input)
```

PYTHON - 2

Format the result (hash) as a hexadecimal string using the format function. Here is the function prototype for converting to hexadecimal:

format(value, ‘X’)

Print the result.

Run the program. Congratulations!

For now, hashing is just a transformation of data. Much more to come.

JAVA

Create a Java project.

Find the source file for the project. Implement the following steps in that file.

Import the entire java.security package.

In main, create a global try / catch.

JAVA – 2

In try:

1. Create a MessageDigest instance called digest. Initialize with MessageDigest.getInstance. The only parameter is “SHA_256”, which is the most popular hashing algorithm.
2. For creating the hash, call the digest function on the digest object. The only parameter are the bytes of “Hello, World!”. Save the return that is a byte array (i.e., the hash).
3. Print the hash to the screen.

Lab completed

TALKING SECURITY

TERMINOLOGY

2

INTRODUCTION

Cryptography is the science of keeping secrets, where hashing and encryption are the primary components.

Cryptography supports these important security principles:

- Confidentiality
- Integrity
- Authentication

Cryptography has various applications: X.509 certificates, token authentication, secure boot, blockchain security, root of trust, and more.

CIA TRIAD

CIA Triad is a reference to confidentiality, integrity, and availability, which are the pillars of security. Each is a category of security:

- Confidentiality: Confidentiality is the heart of cryptography, which is protecting sensitive data.

  Applying encryption to data provides confidentiality.

CIA TRIAD - 2

- Integrity: Integrity is about trust. Knowing whether data is reliable is essential in preventing attacks.

  Hashing algorithms are used to confirm integrity.

- Availability: Availability means resources and services are available as expected.

  Digital signatures, a combination of hashing and encryption, can help protect against non-availability by preventing unauthorized users.

STRIDE

STRIDE is an acronym and the categorization of vulnerabilities. It useful in deciding, based on the categorization, the best practices of how to defend or mitigate against current or future problems.

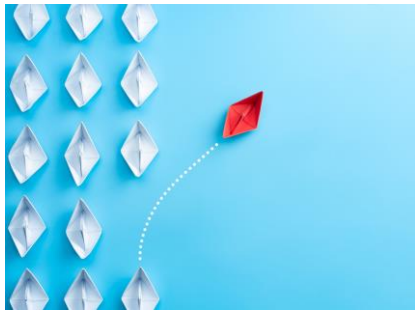
- Spoofing: theft of user credentials. Compromise of authentication
- Tampering: unauthorized data modification. Compromise of integrity.
- Repudiation: repudiation of activity. Compromise of non-repudiation.
- Information Disclosure: unauthorized access to data. Compromise of confidentiality.
- Denial of Service: prevent correct access to data or services. Compromise of availability.
- Elevation of Privilege: unauthorized upgrade in privilege. Compromise of authorization.

POP QUIZ:

Label the per picture boxes with STRIDE elements (s, t, r, i, d, e).





5 MINUTES

HASHING

Hashing is a one-way algorithm that creates a fixed length digest (transformation) from input data.

Features of hashing include:

- Hashing is conducted in sequential blocks
- There is no detectable pattern
- The results are both unpredictable and predictable
- Longer digests tend to be more secure

HASHING - 2

Hashing algorithms are also referred to as ciphers, where block ciphers are the most common.

There are various approaches to creating hashes:

- Block cipher
- Stream cipher
- Keyed hash

ENCRYPTION

Encryption is an important tool in cryptography and provides confidentiality. Encryption protects from unauthorized information disclosure.

This is a two-way algorithm: encryption and decryption. Encrypt data using an encryption algorithm, which is a different type of transformation. Decryption decodes encrypted data to reveal the original contents.

ENCRYPTION - 2

Encryption is a component in other cryptography features, including:

- Digital signatures
- Key exchange
- Root of trust

ENCRYPTION - 2

There are two types of encryption:

- Symmetric encryption: encrypts plaintext using a single key to create the ciphertext.
- Asymmetric encryption: encrypts using paired keys: public key and private key.

MESSAGE AUTHENTICATION CODE (MAC)

A MAC verifies both the integrity and identification of a message. The MAC consists of a keyed hash. This is a hash seeded with a private key.

There are three aspects of a Mac:

- Creating a private key
- Signing a hash
- Verifying the MAC

FINGERPRINT

What is a digital fingerprint?

There is often confusion about the definition of a digital fingerprint versus digital signature.

A digital fingerprint is a hash of a public key (discussed shortly).

Two parties with shared knowledge of a public key can exchange the key using the hash. The hash becomes a factor for authentication.

DIGITAL SIGNATURE

Digital signatures are a combination of hashing and encryption to verify the identity of a sender and integrity of the message.

Digital signatures potentially defend against several vulnerabilities in STRIDE:

- Spoofing
- Tampering
- Repudiation

DIGITAL SIGNATURE

Digital signatures use asymmetric encryption.

The sender creates a digital signature while the receiver verifies the signature.

DATA

The most frequent goal of the attackers is data. This includes:

- Identity theft
- Stealing data
- Modifying data
- Ransomware
- Information disclosure

DATA - 2

Protect data in all states

- At rest
- At transit
- In memory

P1

P1 data is personally identifiable information (PII). This is highly sensitive personal data that identifies a specific individual.

- Government ID
- Credit card
- Email address
- Ocular

Secure with the highest level of security.

P2

P2 is personally anonymous data. These are attributes that can identify you but also others.

- Hair color
- Address
- Mother maiden name
- Name of High School

This data is semi sensitive.

P3

P3 data does not have personal significance. Be sure that the data is not miscategorized, which has been the source of several attacks.

POP QUIZ: P2

Can enough P2 data be considered P1?

5 MINUTES

INITIALIZATION VECTOR

An initialization vector (IV) adds another ingredient to the asymmetric encryption algorithm, which makes the digest less predictable. Adversaries can use predictability to glean information from patterns and exploit security vulnerabilities. For that reason, an IV makes encryption more secure.

PADDING

Encrypted data has a definitive start and end, based on the input. Hackers can potentially exploit this predictability. Yup, predictability again!

Encryption padding adds "random stuffing," which further obfuscates the result. These are the two most popular padding schemes:

- **OAEP (Optimal Asymmetric Encryption Padding)**
- PKCS1 (Public-Key Cryptography Standards)

The details of padding algorithms are beyond the scope of this course.

https://bit.ly/3z6rMzq

BLOCK CIPHER MODES

There are different block cipher modes (also called chaining modes) that are used to combine cipher blocks:

- Electronic Codebook (ECB): encrypt each block and concatenate the individual results.
- Cipher Block Chaining (CBC): encrypt the first block with a seed. The initial seed is the IV vector. The remaining blocks are encrypted with the previous cipher block as the seed.

BLOCK CIPHER MODES - 2

- Cipher Feedback (CFB): a mode where data is being provided in a stream versus a block.
- Counter Feedback Mode (CTR): this mode is the combination of encrypting a counter which is then XOR'd with plain text.

RANDOM NUMBERS

Random numbers are frequent input for cryptography. However, improper randomness is often a vulnerability. A common problem are random numbers that are not sufficiently random, such as pseudo-random numbers.

The solution is random numbers generated with entropy or created specifically for cryptography.

For example, the Random class in C# should not be used with cryptography. The RNGCryptoServiceProvider class is the correct solution.

ENTROPY

Entropy is an uncertain random value. Here are attributes of a value with Entropy.

- Disordered
- Unpredictable
- Not discoverable
- Not repeatable

For example, mouse location is often a source of entropy.

https://bit.ly/3aJX7RU

BOB / ALICE

Bob and Alice are common characters used to document cryptography scenarios. Bob is typically the recipient of an encrypted message. Alice is the sender:

> Bob creates a public / private key pair. Bob forwards Alice the public key. Alice encrypts a message with the public key and sends to Bob. Bob receives the message and decrypts the message with his private key. Bob can now read Alice's message – "Hello, Bob!"

https://bit.ly/3HrueGm

CRYPTOGRAPHY - VULNERABILITIES

Cryptography is sometimes the source of security vulnerabilities. Common vulnerabilities include:

- Weak random number generation
- Improper key storage
- Deprecated algorithms
- Misconfiguration
- Poor policies
- Self-signed certificates
- Invalid certificates

COLD BOOT

Never underestimate hackers! Their attacks are often ingenuous. Unfortunately, some of the best engineers are cyber attackers.

Imaginative attacks are appearing every day. This is yet another excellent reason to implement defense in depth.

The cold boot attack is a perfect example of creative from the evil empire.

COLD BOOT - 2

Cold boot attack is another example of a side-channel attack.

There are actually two forms of cold-boot attacks. For one of the attacks, the term “cold-boot” is figurative.

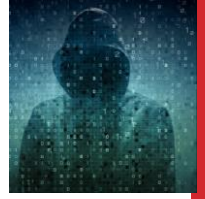
For the other, the term “cold book” is a literal description. Brrrr…

Both categories of cold boot attacks requires physical access to the device.

COLD BOOT (FIGURATIVE) ATTACK

When a computer is shutdown, data in the memory chips may persists for a few seconds, even minutes after the abrupt shutdown. Memory in the DRAM and SRAM are especially vulnerable to this sort of attack.

Most often the system is rebooted from a operating system on a secondary device, such as a usb stick. The hacker can then employ various techniques for dumping the sensitive information.



COLD BOOT (LITERAL) ATTACK

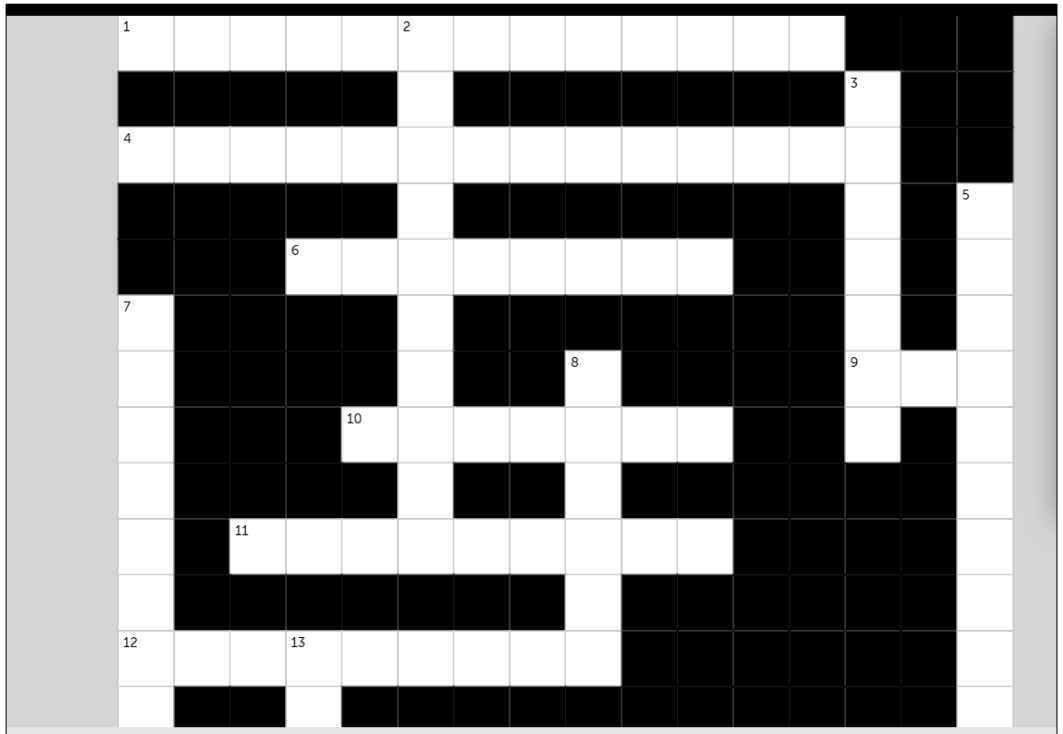
As an extension to the cold boot attack, you can actually freeze memory chips using liquid nitrogen, freeze sprays, and other techniques. When cooled, the memory will take longer to erode. This provides the hacker more time to remove sensitive data from the impacted memory.

LAB 2 – PUZZLE

PUZZLE

In this module, you have learned several terms related to cryptography. As a learning exercise, in this lab you will complete a crossword puzzle with the terminology you have learned in the first two modules.

SECURE PASSWORDS



ACROSS

1 Raw data input

4 Component of CIA

6 S in STRIDE

9 Personally Indentifiable Information

10 Completes a block

11 Hashes provide...

12 Compromise of integerity

14 Unknown attack

DOWN

2 Attack that freezes assets

3 Uncertain random value

5 DoS impacts...

7 Two-way algorithm

8 Side channel attack

13 High collision randon number

Here is the puzzle.

Have fun completing!

Lab completed

HASHING

CHECKSUMS

HASHING

Hashing protects the integrity of data. With a hash, you can detect tampering

A hash algorithm will convert variable length data into a fixed length encoded result, which is called a digest. Any change to the data, even a single bit, can cause a radically different digest being generated using the hash algorithm.

Here are the attributes of a secure hash algorithm:

- One way – must use brute force to reverse
- Should be fast
- Memory efficient

Hash algorithms do not encrypt data.

HASHING - JAVA

Hashing protects the integrity of data. With a hash, you can detect tampering

A hash algorithm will convert variable length data into a fixed length encoded result, which is called a digest. Any change to the data, even a single bit, can cause a radically different digest being generated using the hash algorithm.

Here are the attributes of a secure hash algorithm:

- One way – must use brute force to reverse
- Should be fast
- Memory efficient

Hash algorithms do not encrypt data.

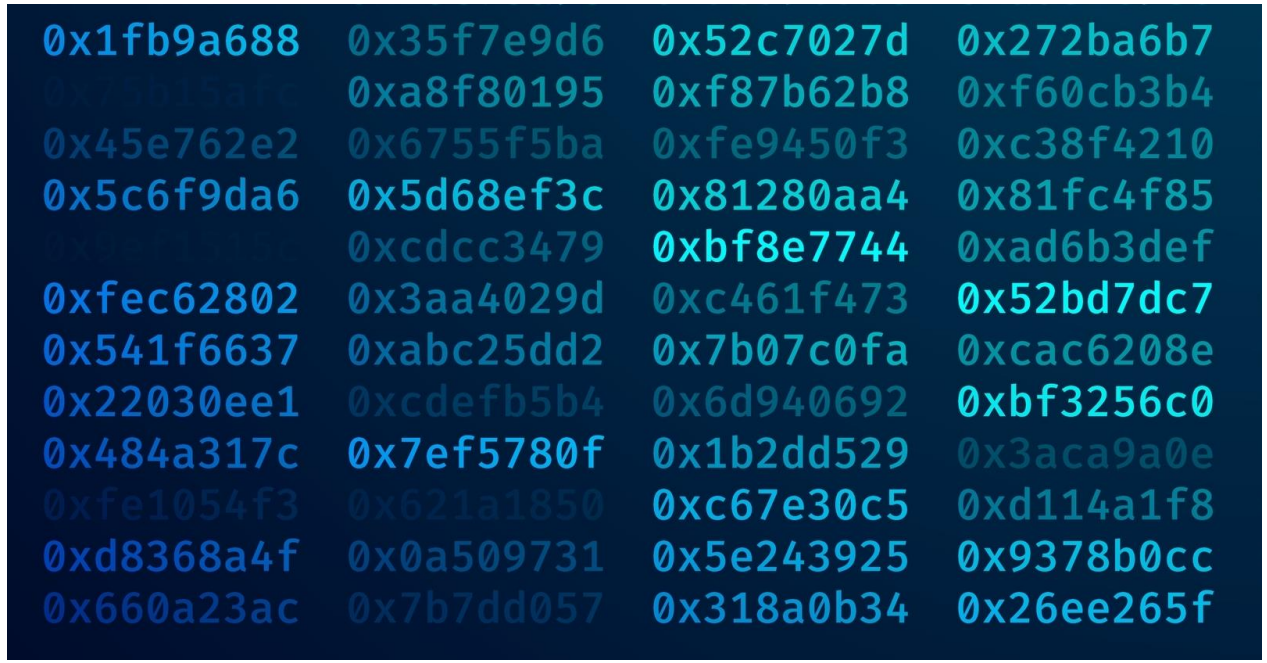
HASHING - 2

Hash algorithms can hash (transform) a variety of input. However, some hash algorithms may require that the input format is binary. The input for a hash is often called a message, while the result is a digest.

Here are features of a hash algorithm:

- Deterministic: the result of a hash, based on a specific value, should never change.
- Fixed-length: regardless of the message size, the result of applying the hash algorithm is fixed length.
- Avalanche Effort: any change, even a minor change, can result in a radically different hash.

HASHING - 3



- One-way: as mentioned hashing algorithms are one way. More on the next slide.
- Collision resistant: it is unlikely, if not impossible, for an attacker to uncover collision, where two inputs have the same hash.
- Performant. hashes must be highly performant. Slow hashes are hard to scale which is necessary for many use cases.

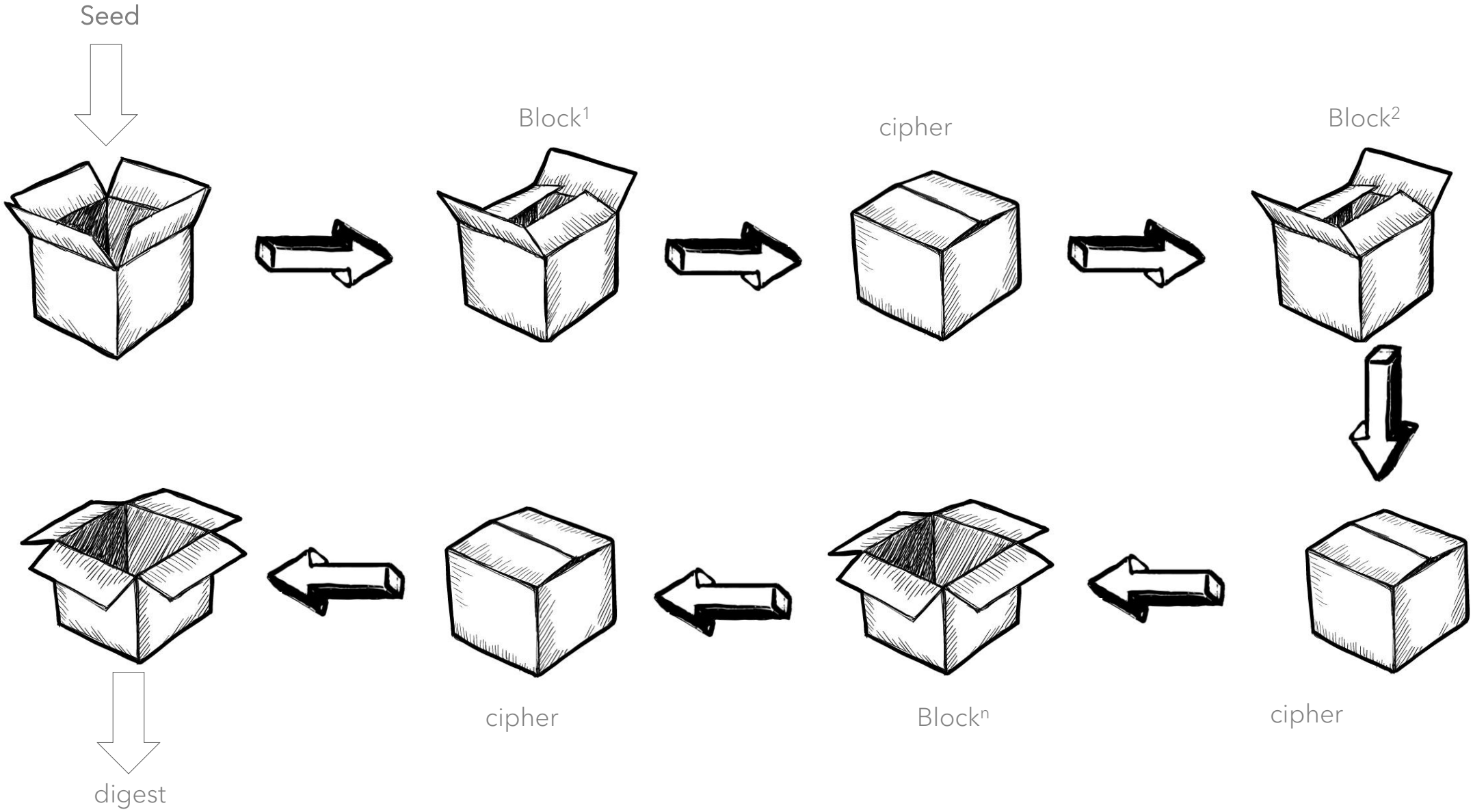
BLOCK CIPHER

A block cipher is an algorithm, when applied to a data source, decomposes the input and processes one chunk (block) at a time. The algorithm processes the blocks in order.

Basically, a block takes nibbles instead of eating an entire meal at one.

Would you like to add some salt to that sandwich? You can always seed your input with a unique value. When mixed in with normal hashing, this adds another level of unpredictability.

BLOCK CIPHER



POP QUIZ: HASH ALGORITHMS

What is the major security shortcoming of hash algorithms?

10 MINUTES

BRUTE FORCE

No hash is computationally impenetrable. Even a hash can be broken with a brute force attack – it is just a matter of time. The goal of a hash algorithm is to make a brute force attack unfeasible, not impossible.

For example, MD5 is no longer considered secure. A brute force attack takes less than 30 minutes to resolve MD5 encryption.

The current standard is SHA-256. Sometime in the future, SHA-256 will also become insecure. At that time, a different hash algorithm must be chosen. Fortunately, SHA-3 and other algorithms are already completed and in the queue.

POLICIES

Policies can help or hurt the securability of a hash.

Here are policies that can hurt:

- Length of a password
- Periodic renewal of secure resources
- Throttling
- Limitations to character set

SECURE HASH ALGORITHM (SHA-256)

At present, the most common hashing algorithm is SHA-256. SHA-256 is a member of the SHA 2 family of algorithms and creates a 256-bit digest. For example, SHA-384 and SHA-512 are also included in the SHA 2 family. Although more secure, there is a performance trade-off when using SHA-384 and SHA-512.

SHA-3 is available but not widely used.

HASHING ALGORITHMS

Here are the details of the most prevalent hashing algorithms.

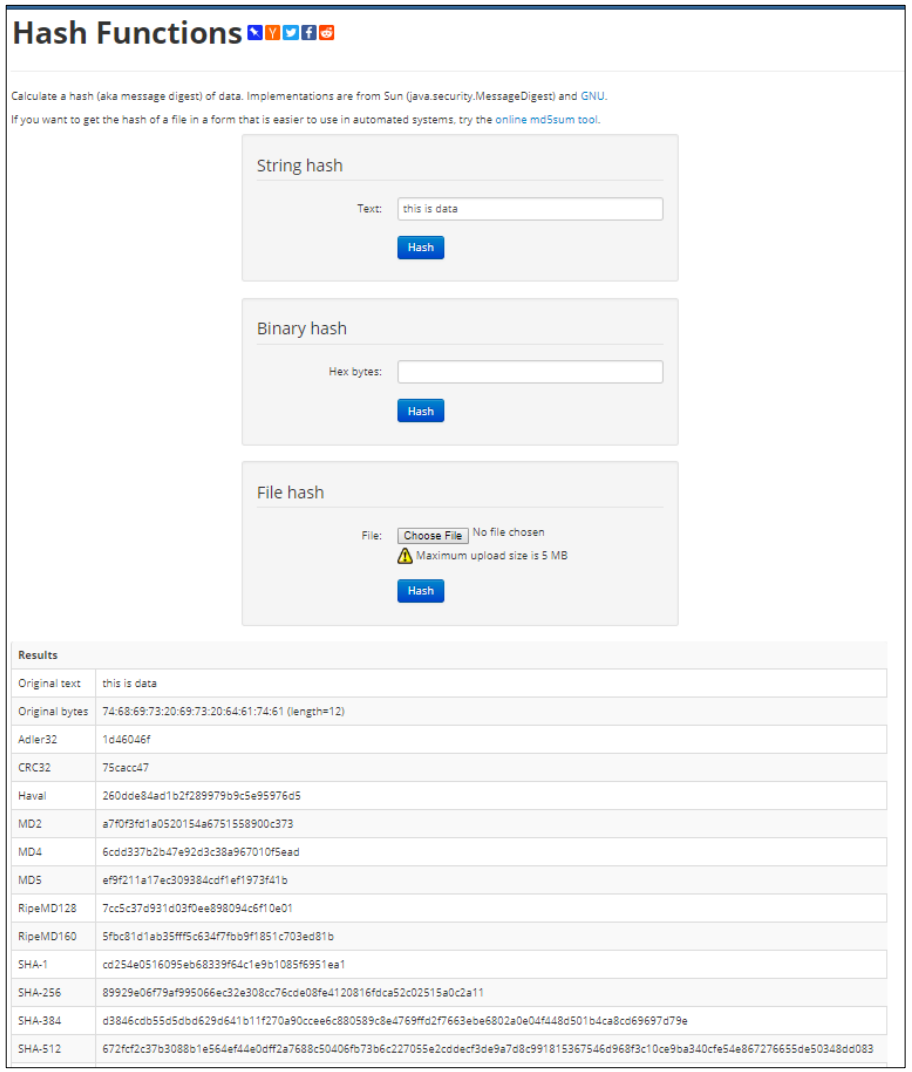
Name	Message Size	Block Size	Hash Size
MD5	2^{64}	512	128
SHA-1	2^{64}	512	160
SHA-256	2^{64}	512	256
SHA-384	2^{128}	1024	384
SHA-512	2^{128}	1024	512
SHA3-256*	Infinite	1088	256

* SHA-256 and SHA3-256 render digests of the same size. However, the SHA3 digest is more secure.

ONLINE HASHING TOOL

FileFormat.info has a great tool for comparing hashes generated from various algorithms.

https://www.fileformat.info/tool/hash.htm



HASHLIB

For Python, the Hashlib library provides a common interface for the various hash algorithms that are supported.

These algorithms are supported.

Md5	Sha1	Sha224
Sha256	Sha384	Sha512
Sha3_256	Sha3_512	Blake2b
blake3s		

NAMED CONSTRUCTORS

```
from hashlib import sha256

message=b'42'

hashobj=sha256(message)

result=hashobj.hexdigest();

print(result)
```

You can create a hash algorithm using named constructor, such as sha256 or sha3_512.

Named constructors create hashing objects. You can initialize the hashing objects with a message.

Call hash.hexdigest function returns a hash of the message, as a bytes object.

You can feed extra data into the hash using the update method.

NAMED CONSTRUCTORS - JAVA

GENERIC CONSTRUCTOR

```
import hashlib

from hashlib import sha256

message=b'42'
hashobj=hashlib.new('sha256')
hashobj.update(message)
print(hashobj.hexdigest())
```

As an alternative to a named constructor, you can use a generic constructor (new method). In the constructor, name the representative hash algorithm. The algorithm is described within a string. The new function returns a hash object.

You can add a message to the hash object with the update message.

GENERIC CONSTRUCTOR – JAVA

```
import java.security.*;

public class HashingSHA {
    public static void main(String[] args) {
      try {
        MessageDigest digest =
           MessageDigest.getInstance("SHA-256");
        byte[] encodedhash = digest.digest(
        "test".getBytes());
        System.out.println(encodedhash);
      }catch(Exception ex) {

      }
    }
}
```

For Java, there is one basic model for hashing data. This is identical to your code in the first library.

There is only an intermediate solution for the standard solution for creating hashes.

This is it!

HASHING – PARTIAL INPUT

```
from hashlib import sha256

message=b'42'
hashobj1=sha256(message)
result=hashobj1.hexdigest();
print(result)

message=b'4'
hashobj2=sha256(message)
hashobj2.update('2'.encode('ascii')  )
result=hashobj2.hexdigest();
print(result)
```

After creating, you can add messages continuously to the hash object with the update function.

In the first code snippet, we create a hashing object initialized with the entire input.

In the second, the buffer is filled in portions.

HASHING – PARTIAL INPUT – JAVA

```
import java.security.*;
public class App {
    public static void main(String[] args) throws Exception {
        try {
            String char1="4";
            String char2="2";
            MessageDigest digest =
                MessageDigest.getInstance("SHA-256");
            digest.update(char1.getBytes());
            digest.update(char2.getBytes());
            System.out.println(digest.digest());
          }catch(Exception ex) {
            System.out.println("Exception");
          }
    }
}
```

After creating, you can add messages continuously to the hash object with the update function.

In the first code snippet, we create a hashing object initialized with the entire input.

In the second, the buffer is filled in portions.

STREAMING HASH

A streaming hash does not require that the plaintext is memory resident. You can collect the plaintext in partial increments until complete. With everything upfront, you can add to the plaintext in chunks, as the data arrives.

This is convenient for a variety of data:

- Network
- File storage
- Calculated data

STREAMING HASH -2

Here are the basic steps:

1. Create a hash
2. Generate the nth chunk
3. Call hash.update to add the current chunk to the hash
4. Return to Step 2 until data is completed (no additional chunks)
5. Call hash.hexdigest or other related method to create the hash of all the chunks

STREAMING HASH – IMPLEMENTATION

```
import hashlib # hashlib module
import os.path # For file handling
from os import path

hash = hashlib.sha256()

file=open('readme.md','rb')
chunk=''
```

In this example, we will read 512 byte chunks from a file until complete. At each iteration, we will add the current chunk into the plaintext cached in the hash.

This is the setup for the sample code. The imports are added.

We also initialize the important ingredients for this code:

- Hash object
- File object for the input file
- Chunk initialized to nothing at the start

STREAMING HASH – IMPLEMENTATION – 2

```
while True:
    chunk = file.read(512)
    if chunk == b'':
        break
    hash.update(chunk)

hash_value=hash.hexdigest()
print(hash_value)

file.close()
```

In the infinite while loop:

- Read a chunk.
- If empty, break from loop.
- If not empty, add chuck to the hash with the update function.

Generate the hash for the combined chunks.

Display the result.

STREAMING HASH – IMPLEMENTATION JAVA

```
import java.io.File;
import java.util.Scanner;
import java.security.*;

public class App {
    public static void main(String[] args)
                    throws Exception {
        try
        {
            File file = new File("…count.txt");
            Scanner sc = new Scanner(file);

            MessageDigest digest =
               MessageDigest.getInstance("SHA-256");
```

In this example, we will read lines from a text file. At each iteration, we will add the current line to the hash object, MessageDigest.

This is included in the code snippet;

- Open a file
- Initialize a Scanner object
- Create a hashing object

STREAMING HASH – IMPLEMENTATION – JAVA – 2

```
while (sc.hasNextLine()) {
          String partial=sc.nextLine();
          digest.update(partial.getBytes());
        }

        System.out.println(digest.digest());
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
    }
}
}
```

In the while loop:

- Check ahead for the next line of input. If there is additional input, do the while loop.
- Read the next line
- Convert the line to bytes
- Add to cache in hash object

Hash everything with the digest function.

Display the results.

DOUBLE SHA

```
from hashlib import sha256


message=b'42'
first_sha=sha256(message)
temp=bin(int.from_bytes(
    first_sha.digest(), 'big'))
double_sha=sha256(temp.encode('utf-8'))


result=double_sha.hexdigest();
print(result)
```

Double SHA is a simple protocol for increasing security and making a hash harder to reverse.

It is a simple process. You call SHA256 algorithm twice.

1. Apply SHA256 to the plaintext and create the first hash
2. Apply SHA256 to the first hash – creating a second hash

You then use the second hash, which has additional security.

DOUBLE SHA - JAVA

```
try {
            MessageDigest digest =
                MessageDigest.getInstance("SHA-256");
            byte[] encodedhash = digest.digest(
                    "test".getBytes());

            encodedhash = digest.digest(
                encodedhash);

          System.out.println(encodedhash);
        }catch(Exception ex) {
          System.out.println("Exception");
        }
    }
```

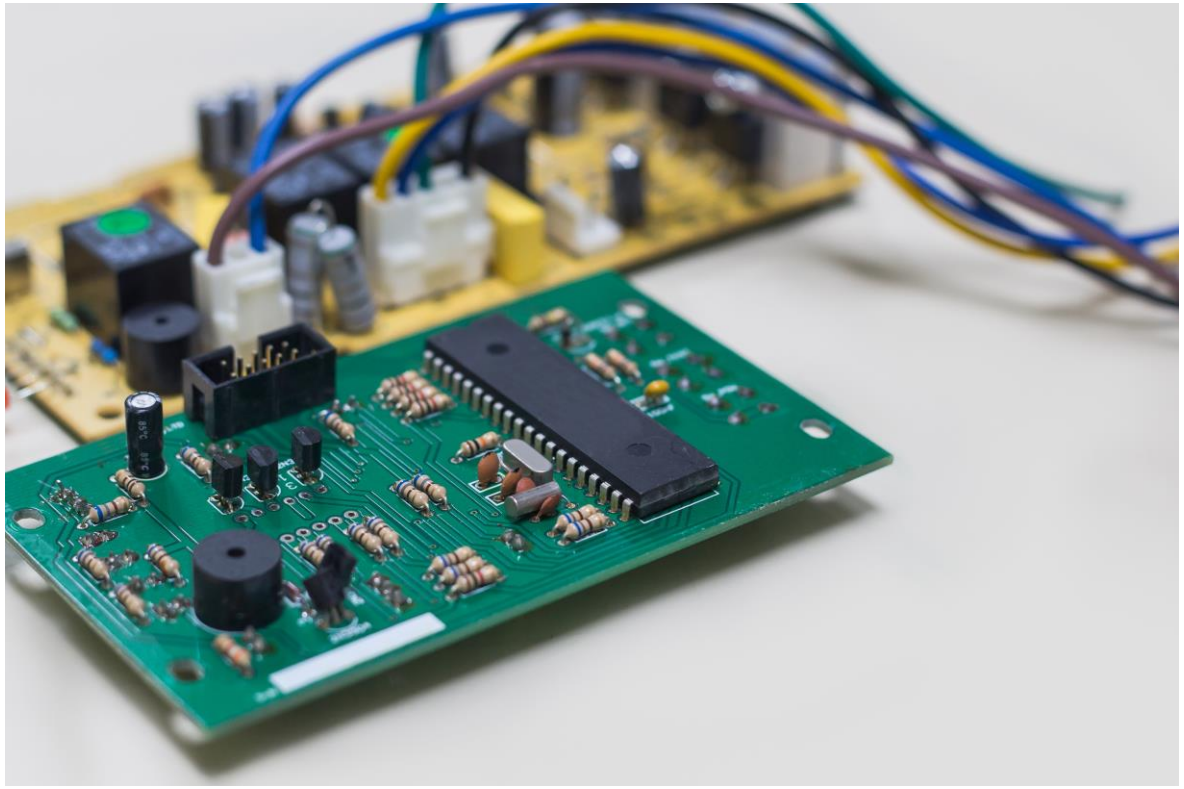
Double SHA is a simple protocol for increasing security and making a hash harder to reverse.

It is a simple process. You call SHA256 algorithm twice.

1. Apply SHA256 to the plaintext and create the first hash
2. Apply SHA256 to the first hash – creating a second hash

You then use the second hash, which has additional security.

ROWHAMMER ATTACK



"Necessity is the mother of invention." – Plato

Rowhammer attacks are a perfect example of this adage. Adversaries are increasingly inventive in finding creative solutions to attack your systems and applications.

In this attack, a program is executed repeatedly on a row of transistors on a memory chip. The row is hammered until electricity leaks into another row and flips some bits. The adversary can combine bit transformations for a larger purpose: to execute malicious code or otherwise attack the system.

This behavior is more predictable with standard RAM. However, this attack has now been replicated with ECC memory. Surprise!

ROWHAMMER ATTACK - 2

Error Correcting Code Memory (ECC memory) is supposedly resilient to Rowhammer attacks. ECC memory detects bit tampering and amazingly corrects the change. This would seem to be an effective mitigation.

ECC is used in sensitive systems, such as financial systems and IoT devices, because this sort of exploitation is especially troublesome.

ROWHAMMER ATTACK - 3

Here are the steps of a Rowhammer attack on ECC memory:

1. Rowhammer does templating to locate vulnerable bits. This is possible because ECC memory does not detect tampering of three consecutive bits.
2. Access time for unaltered bits versus tampered bits is different. In poker, this is called a *tell*. Based on this, you know where to hammer.
3. This is not a quick process. However, the process of templating is virtually undetectable. The adversary can map the attack for days, even weeks, before launching the Rowhammer attack.

More detail on Rowhammer attacks on ECC: https://bit.ly/2U4EYTg

LAB 3 – CHECKSUM

INTRODUCE

Checksums are created to provide data integrity but are considered lightweight hashes. Not every target of hashing requires a security-level hash, such as SHA-256.

Checksums are a highly performant and insecure hash.

LOST BITS

Lost of bits during some sort of transformation is a frequent scenario for employing checksums.

- Compressing / Decompression
- Network transmission
- File transfer
- Data transformation

Note that these use cases do not raise security issues. Nonetheless, integrity is still required.

GETTING STARTED

In this lab, you will read a file with no intention of making any changes. The checksum confirms there are no inadvertent changes between reads of the file. Changes could occur from a variety of reasons:

- Inadvertent changes from your application (theoretically)
- Media defect
- Electrical surge
- And so on

GETTING STARTED

Create a new Python project.

In the project directory, create a new text file.

Enter the following text and save the file:

"Cryptography is the science of keeping secrets."

The checksum functions are in the zlib library. For that reason, import that library.

CHECKSUM

- Define a prev_checksum variable. Initialize to 0.
- With the open function, open the text file you created in read-only mode.
- With the read function, read the entire file into a variable.
- Create a checksum with the zlib.crc32 function. The only parameter is the content of the file, as bytes.

COMPARISON

- Compare the prev_checksum and checksum. If different, display “Verified”. Otherwise, display “Not Verified”.
- Close the file.
- Display the file text
- Display the checksum

COMPARISON - 2

If checksum not equal 0:

- Compare the prev_checksum and checksum. If different, display “Verified”. Otherwise, display “Not Verified”.
- Close the file.
- Display the file text
- Display the checksum

RUN PROGRAM

Run the application. It should display the file text and checksum. Record the checksum.

- In the program, manually change the prev_checksum to the recorded checksum.
- Run the program. What is the result?
- Make a minor change in the text file.
- Run the program again. What is the result?

You are done. Great work!

JAVA

GETTING STARTED

Create a new Java project.

In the project directory, create a new text file.

Enter the following text and save the file:

"one two three"

The checksum functions are in the java.util.zip package. For that reason, import these packages.

```
import java.util.zip.*;

import java.nio.file.*;
```

READ FILE

1. Define a prev_checksum variable. Initialize to 0.

```
long checksum=Long.parseUnsignedLong("0");
```

2. Get the path to your file using the Paths.get method. The only parameter is the file path. The function returns a Path object.
3. With the path, call Files.readAllBytes. It will read the entire file into a byte array.
4. Create a checksum with the zlib.crc32 function. The only parameter is the contents of the file, as bytes.

COMPARE CHECKSUM

1. If checksum is 0:
   - Display the checksum
   - When you run the program, copy the checksum. Manually change the initial value of the checksum to that value. It should no longer be zero.
2. If not zero
   - If the current and initial checksum match, display "Verified".
   - Otherwise, display "Not Verified".

RUN PROGRAM

Run the application. It should display the file text and checksum. Record the checksum.

- In the program, manually change the checksum to the recorded checksum as a string.
- Run the program. What is the result?
- Make a minor change in the text file.
- Run the program again. What is the result?

You are done. Great work!

Lab completed

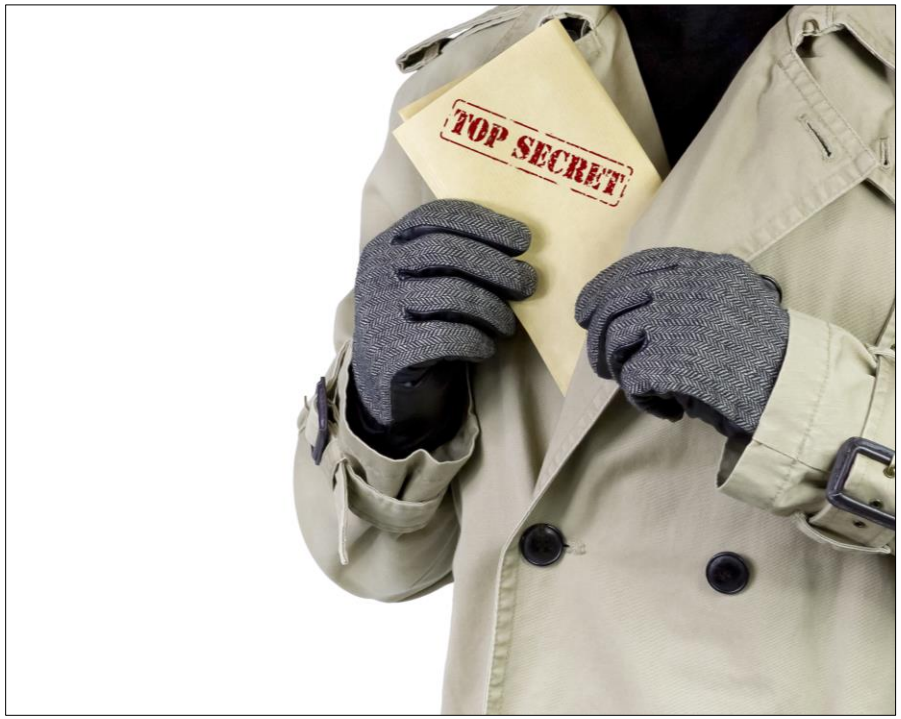
ENCRYPTION

SYMMETRIC / ASYMMETRIC

ENCRYPTION

Encryption is about keeping secrets confidential. Conversely, decryption discloses the secret.

Encryption algorithms with larger key sizes are more secure. But this is a relative concept. Future increases in computing power will eventually compromise the current “secure” key size.

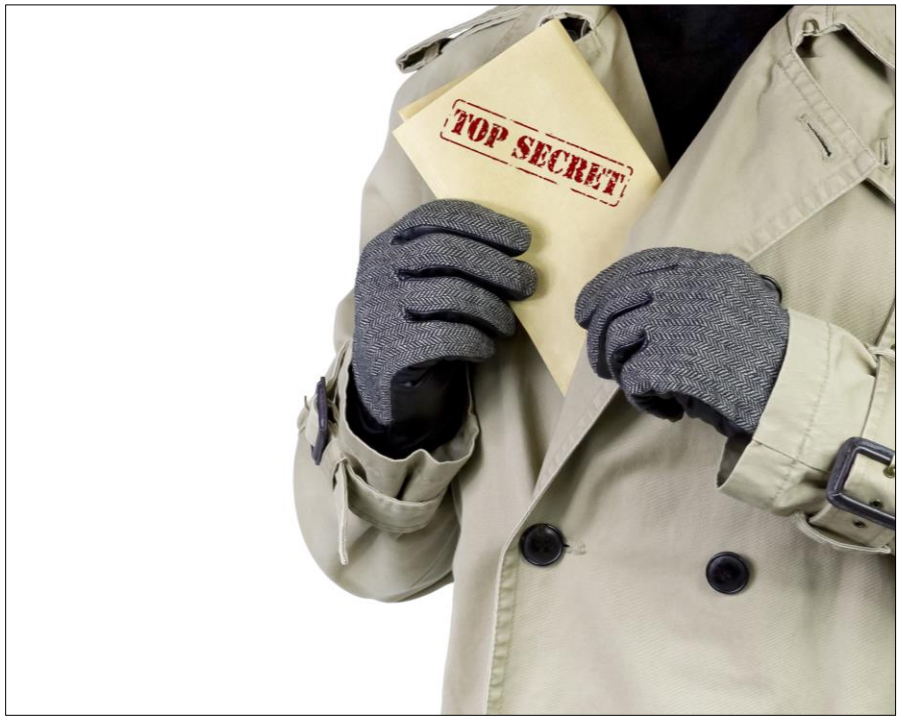


ENCRYPTION

Encryption is about keeping secrets *secret*.

First encrypt the plaintext containing the sensitive data. The result is ciphertext which is undecipherable.

Later you can recover the original plaintext by decrypting the ciphertext.

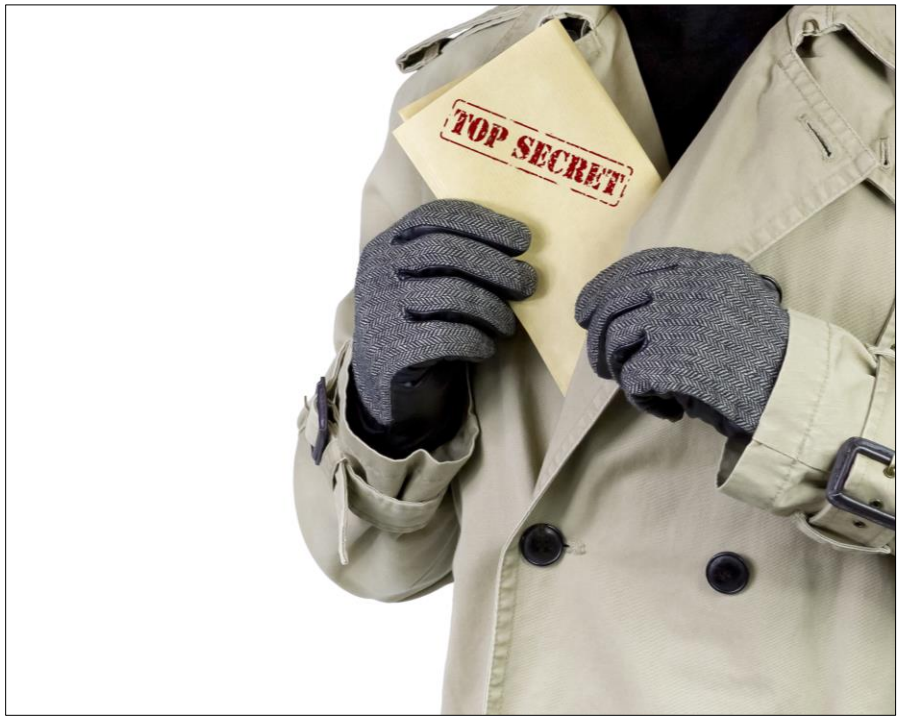


ENCRYPTION VERSUS HASH

Encryption is about preserving confidentiality / hash is about preserving integrity.

Encryption employs two-way algorithms / hash employs one-way algorithms

Encryption requires keys / hash does not require keys



SYMMETRIC VERSUS ASYMMETRIC

A symmetric key is employed to both encrypt (code) and decrypt (decode) data. Conversely, asymmetric keys require two keys: public key and private key.

The challenge with symmetric key encryption is both the sender and receiver must agree upon the same key. This needs to be done in a secure manner.

With asymmetric key encryption, the receiver keeps a private key and provides senders a public key. Since the public key is *public*, it does not have to be transmitted securely. There is an algorithmic relationship between the public and private key, where only the holder of the private key can decrypt data that was encrypted with the public key. And vice versa.

SYMMETRIC VERSUS ASYMMETRIC

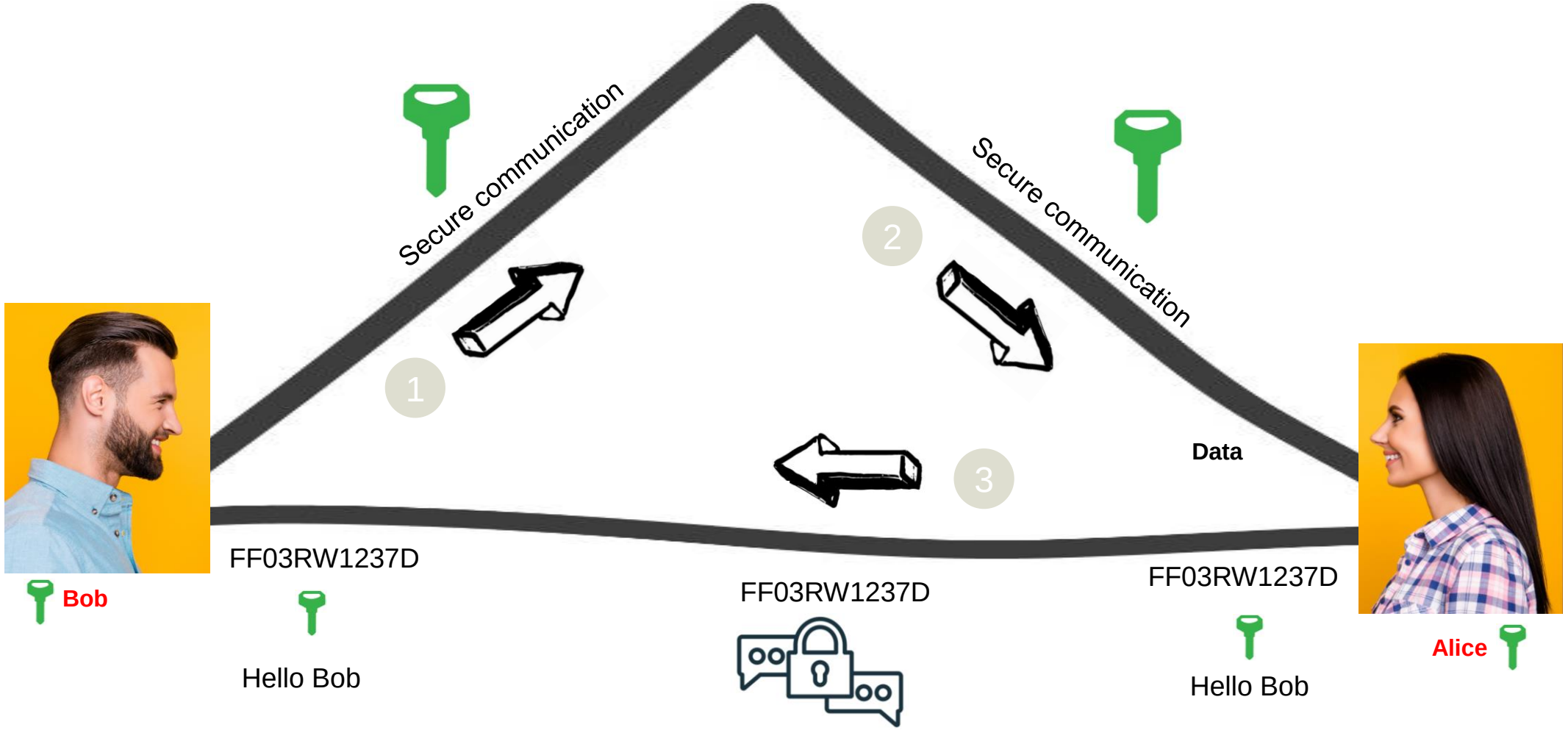
There are two types of encryption: symmetric version asymmetric.

For symmetric-key encryption, a single key is used for both encrypting and decrypting data.

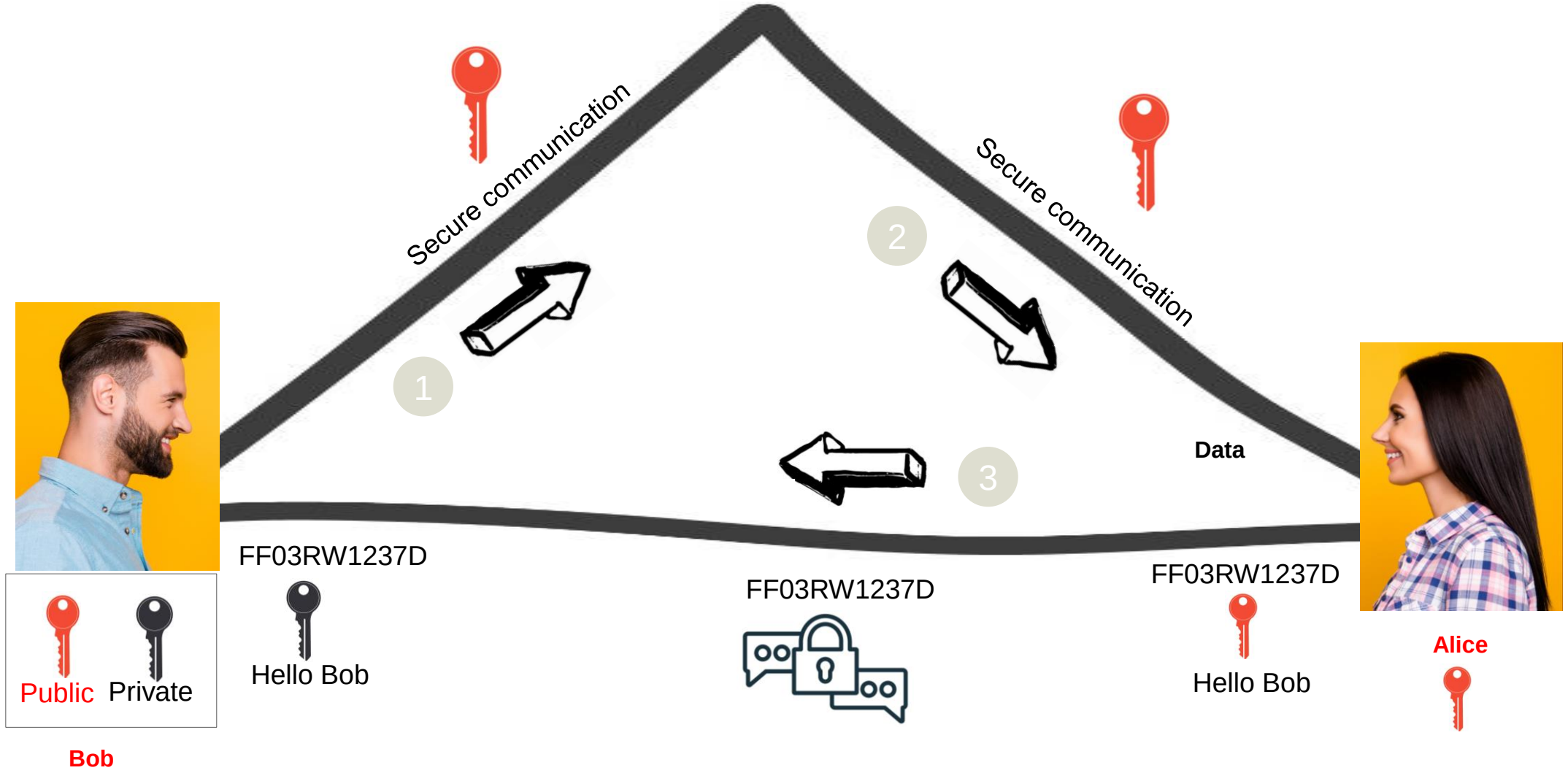
The challenge with symmetric key encryption is both the sender and receiver must agree upon the same key. And this must be done in a secure manner.

With asymmetric key encryption, there is a private and public key. The receiver keeps the private key and publishes the public key. Since the public key is *public*, it does not have to be transmitted securely. There is an algorithmic relationship between the public and private key, where only the holder of the private key can decrypt data that was encrypted with the public key. And vice versa.

SYMMETRIC - DIAGRAM



ASYMMETRIC - DIAGRAM



INITIALIZATION VECTOR

How many of you have the same password account for various accounts? Be honest!

This creates predictability. Predictabilities can be exploited by hackers. The initialization vector (IV) adds another factor of randomness during the encryption process. The initialization vector is combined with a random key for an extra level of protection.

The initialization vector is not necessary a secret. However, it should not be repeatable. How to determine the initialization vector must be known between the participating parties.

CIPHER FEEDBACK MODE

As shown earlier, a block cipher is an algorithm that encrypts segments of the data (i.e., a block) instead of everything at once. The encrypted segments must be combined in some manner. That process should not be predictable. A cipher feedback mode is the methodology used to merge the results of a block cipher.

Cipher Block Chaining (CBC) is the most popular cipher feedback mode and available in Golang. With CBC, the cipher text of the previous block is XORed with the plaintext of the current block. That becomes the input for the next block cipher. This process will continue until all data blocks have been processed.

ENCRYPTION INPUT

Here are various factors for symmetric encryption:

- Symmetric encryption algorithm
- Chain mode
- Cryptography key
- IV Vector

The two primary algorithms for symmetric encryption is American Encryption Standard (AES) and Data Encryption Standard (DES). AES is more efficient and flexible than DEC.

ENCRYPTION INPUT

AES is flexible and accepts three key lengths.

- AES-128: 128 bits
- AES-192: 192 bits
- AES-256: 256 bits

CODE - ENCRYPT

```
from cryptography.fernet import Fernet

key = Fernet.generate_key()

f = Fernet(key)
data_encrypted = f.encrypt(
  b"So long and thanks for all the business")

data_decrypted=f.decrypt(data_encrypted)

print(data_encrypted)
print(data_decrypted)
```

For symmetric encryption, download the cryptography library.

pip install cryptography

Here are the steps for standard encryption:

1. The generate_key function creates a random key (128-bit key)
2. Create a fernet instance
3. The encrypted function encrypts the plaintext

CODE – ENCRYPT - 2

```
from cryptography.fernet import Fernet

key = Fernet.generate_key()

f = Fernet(key)
data_encrypted = f.encrypt(
  b"So long and thanks for all the business")

data_decrypted=f.decrypt(data_encrypted)

print(data_encrypted)
print(data_decrypted)
```

4. The decrypt method decrypts the ciphertext
5. Display both ciphertext and plaintext

Fernet implements AES:

- 128-bit key
- CBC chain mode
- PKCS7 padding.

CODE – ENCRYPT – JAVA - 2

```
import com.macasaet.fernet.*;
public class Decrypt
  {
    public static void main(String args[])
    {
     final Key key = Key.generateKey();
     final Token token = Token.generate(key, "secret message");
```

Import all classes from com.macasaet.fernet.

Here are the steps for standard encryption:

1. Create a new key instance
2. Create a token

CODE – ENCRYPT – JAVA - 2

```
final Validator<String> validator = new StringValidator() {
  public TemporalAmount getTimeToLive() {
    return Duration.ofHours(4);
  }
};
final String payload = token.validateAndDecrypt(key, validator);
```

3. Validate the tokenDisplay both ciphertext and plaintext

4. Generate a Payload String

Fernet implements AES:

- 128-bit key
- CBC chain mode
- PKCS7 padding.

MULTIFERNET

```
from cryptography.fernet import Fernet,
MultiFernet

key1 = Fernet(Fernet.generate_key())

key2 = Fernet(Fernet.generate_key())

f = MultiFernet([key1, key2])

data_encrypted = f.encrypt(b"Secret message!")

print(data_encrypted)
```

MultiFernet allows you manage the keys. For example, it provides the ability to gradually remove the use of a key.

Initialize the MultiFernet with a sequence of keys. The first key in the list is used for encryption. For decryption, each key is presented to the function until the proper key is found or an exception is raised.

MULTIFERNET - JAVA

ROTATE

```
from cryptography.fernet import Fernet, MultiFernet
key1 = Fernet(Fernet.generate_key())
key2 = Fernet(Fernet.generate_key())
f = MultiFernet([key1, key2])
data_encrypted = f.encrypt(b"Secret message!")

f.decrypt(data_encrypted)
key3 = Fernet(Fernet.generate_key())
f2 = MultiFernet([key3, key2])
rotated = f2.rotate(data_encrypted)
result=f2.decrypt(rotated)
print(result)
```

Rotating the key used for encrypting data is sometimes helpful. The rotate function reencrypts ciphertexts using other key options.

This is useful for example if the administrator managing the keys leaves the company.

ROTATE - JAVA

POP QUIZ: ROTATE - 2

Does the previous code work? If not, why?

10 MINUTES

ASYMMETRIC ENCRYPTION

With asymmetric encryption, a public / private key pair is created. The future receiver provide the sender a public key. The sender uses the public key to encrypt the data (plaintext). The result is cipher text or encrypted data. You receive the cipher text and decrypt with the private key. Asymmetric encryption resolves the major problem of symmetric encryption of how to transfer a key safely?

RSA (Rivest–Shamir–Adleman) is a popular algorithm for asymmetric encryption.

RSA functionality is found in the Crypto/RSA package.

ASYMMETRIC CODE

```
from Crypto import Random
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP

def generate_keys():
            modulus_length = 256*4
            privatekey = RSA.generate(modulus_length,
                        Random.new().read)
            publickey = privatekey.publickey()
            return privatekey, publickey
```

This is generic code that works as a template for conducting asymmetric encryption in Python.

First of all, install pycryptodome:

pip install pycryptodome

Make sure to import the three required libraries: Crypto, Crypto.PublicKey, and Crypto.Cipher.

ASYMMETRIC CODE - 2

```
from Crypto import Random
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP

def generate_keys():
            modulus_length = 256*4
            privatekey = RSA.generate(modulus_length,
                        Random.new().read)
            publickey = privatekey.publickey()
            return privatekey, publickey
```

In the generate_keys functions create the public and private key.

1. Define the key length
2. The generate function returns the private key

   RSA.generate(length, random function)
3. Extract the public key from the private key
4. Return the private and public key

ASYMMETRIC CODE - 3

```
privatekey, publickey=generate_keys()


encryptor = PKCS1_OAEP.new(publickey)
encrypted = encryptor.encrypt(b'encrypt this message')


decryptor = PKCS1_OAEP.new(privatekey)
decrypted = decryptor.decrypt(ast.literal_eval(str(encrypted)))
print(decrypted)
```

Call the generate_key function to return both the public and private keys.

Create an encryption engine (encryptor) with the PKS1_OAEP.new function. The public key is the only parameter. Call the encrypt function to encrypt the message.

Create a decryption engine (decryptor) with the PKS1_OAEP.new function. The private key is the only parameter. Call the decrypt function to decrypt the message.

Display the decrypted (plaintext) message.

ASYMMETRIC CODE - JAVA

```
SecureRandom random = new SecureRandom();


KeyPairGenerator KPGenerator =
KeyPairGenerator.getInstance("RSA");
```

1. Generate public & private key using the *SecureRandom class*. SecureRandom class is used to generate random number.
2. Use the KeyGenerator class to call *getInstance()* to pass a string variable which is the Key Generation Algorithm. (It returns KeyGenerator Object.)

ASYMMETRIC CODE – JAVA - 2

```
keyPairGenerator.initialize(
          2048, secureRandom);


KeyPair keypair = keypair.genKeyPair();
System.out.println(
            "Private Key is: "
            + DatatypeConverter.printHexBinary(
                  keypair.getPrivate().getEncoded()))
```

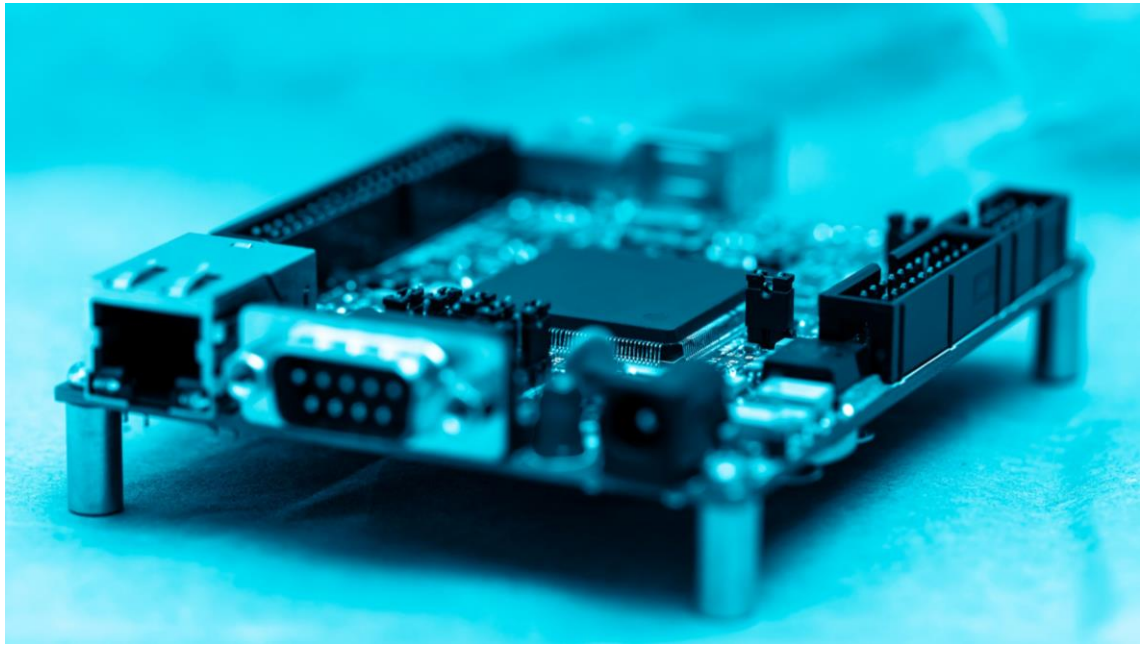
3. Initialize the keyGenerator object with 2048 bits key size and pass the random number.

POP QUIZ: ASYMMETRIC

Does the previous code provide authentication?

10 MINUTES

LOJAX: UEFI ROOTKIT



SEDNIT, a cyberattack group, used LoJax malware to launch attacks against governments in Europe.

Here is more information on SEDNIT:

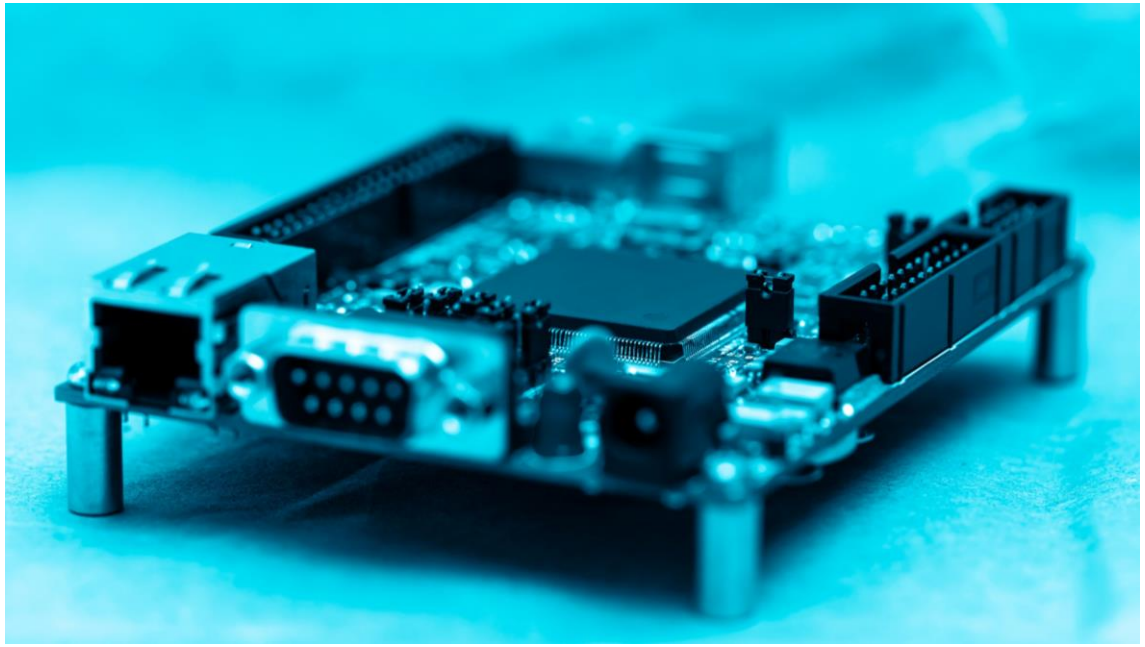
https://attack.mitre.org/groups/G0007/

LoJax attacks are based on a legitimate product: Absolute Software LoJack product, originally called Computrace. LoJack allowed consumers to locate their computer to prevent theft. LoJax compromises the LoJack product.

Unbeknownst to consumers, Lojack was preinstalled on computers from a variety of vendors.

https://bit.ly/3ACevzG

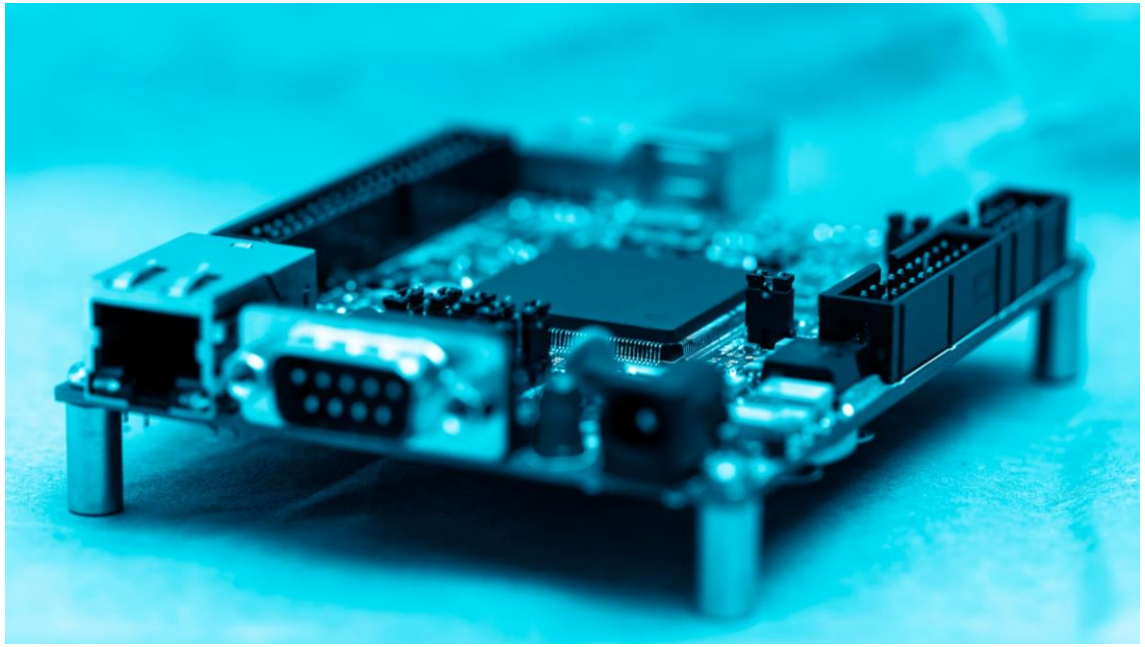
LOJAX: UEFI ROOTKIT - 2



Lojack is installed in the UEFI firmware of a computer or device. It relies on networking to “call home” and provide location information. There are several vulnerabilities

- UEFI is network enabled
- Lojack is persistent
- Since it is based on an actual product, most anti-virus programs and scanners do not detect a LoJax attack
- And much more

LOJAX: UEFI ROOTKIT – 3

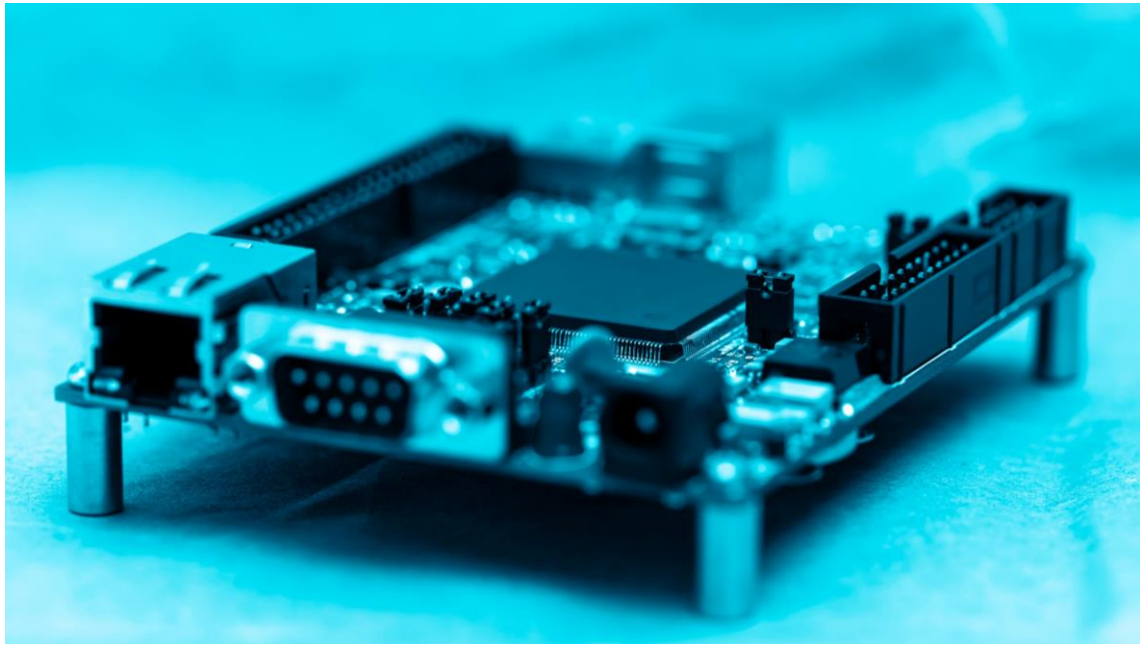


The security strategy of Absolute Software appeared to be –

"Security by obscurity"

Security by obscurity is rarely an adequate security defense. Worst of all, Absolute Software was made aware of potential problems in 2009. However, Absolute Software downplayed the problem for several years.

LOJAX: UEFI ROOTKIT - 4



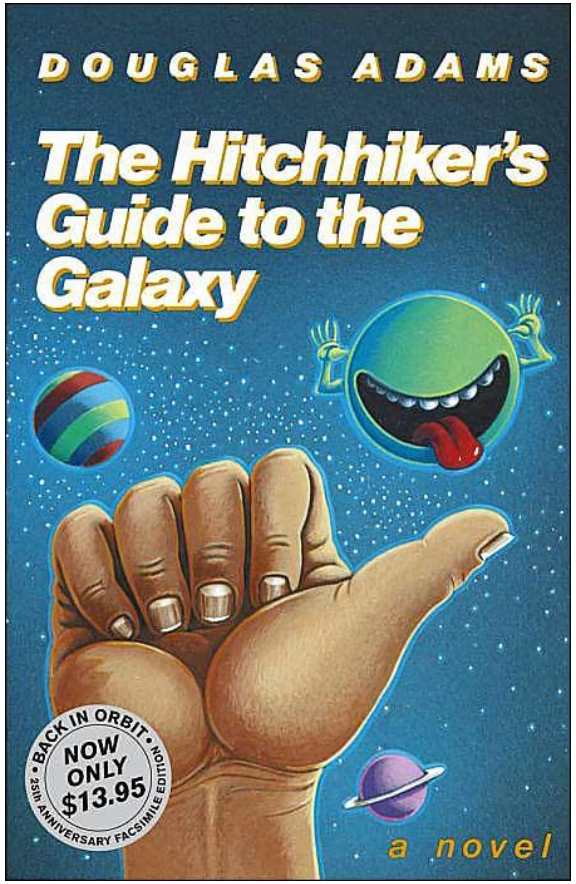
SEDNIT leveraged the network capability of LoJack.

- Dump system information to a file
- Redirected data from the device to a server controlled by attackers
- Save the image of the system firmware
- Wrote a malicious module into the image and saved back to the SPI flash memory

Important: Secure Boot may not prevent this attack.

LAB 4 – PHRASES

PHRASES



The Hitchhikers Guide to the Galaxy is renowned for its fantastical stories and the imaginative use of words. Many phrases from the book have seeped into popular culture.

"Goodbye and thanks for all of the fish" is one of those phrases. It is also the focus of this lab.

SETUP

With the information from this module, encrypt and decrypt these two phrases:

1) Goodbye and thanks for all of the fish
2) Goodbye and thanks for all of the blue fish

The instructions for this labs is purposely *thin*.

IMPLEMENTATION

1. Create a new project
2. Open the source file
3. Declare and initialize both phrases
4. Display the number of bytes in each phrase
5. Implement the code to encrypt both phrases

IMPLEMENTATION – 2

6. Display the ciphertext
7. Display the number of bytes for each ciphertext
8. Implement the code to decrypt the phrase
9. Display the plaintext for both

SAVE CIPHERTEXT

Run the application!

Copy or manually enter the ciphertext into a text file.

QUESTIONS

Despite the minor change are the ciphertext for both phrases significantly different?

Is the size of the plaintext and ciphertext the same or close?

Run it again, has the ciphertext changes for both phrases. If so, be prepared to provide an explanation.

Lab completed

DIGITAL SIGNATURES

DIGITAL SIGNATURE (REPEAT)

Digital signatures are a combination of hashing and encryption. This potentially resolves several vulnerabilities identified within STRIDE:

- Spoofing
- Tampering
- Information disclosure*
- Repudiation

*Optional

SENDER

Digital signatures are useful for sending data securely with identity. It is a combination of a hash and asymmetric encryption.

Here are the steps to create a digital signature for a message / data:

1. Create a hash of the message
2. Encrypt the hash with the private key. The result is a digital signature.
3. Send the digital signature to the receiver with the message

RECEIVER

The recipient receives the signature and plaintext. Before using the plaintext, verify the signature.

1. Hash (hash 1) the plaintext
2. Decrypt the signature and extricate the hash (hash 2)
3. Compare the two hashes. If the same, the signature has been verified.

WALKTHROUGH – MANUAL DIGITAL SIGNATURE

This is a walkthrough of creating a digital signature.

LIBRARIES

```
import hashlib
from hashlib import sha256
from Crypto import Random
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
import ast
```

This is code for manually creating and verifying a digital signature (almost).

These are the necessary libraries for:

- Generating hashes
- Creating private / public keys
- Performing asynchronous encryprtion

FUNCTIONS

```
def get_hash(value):
    hashobj=hashlib.new('sha256')
    hashobj.update(value)
    return hashobj.digest()

def generate_keys():
            modulus_length = 256*4
            privatekey = RSA.generate(
                  modulus_length, Random.new().read)
            publickey = privatekey.publickey()
            return privatekey, publickey
```

Several helper functions are created in the program. The code included in both functions has been demonstrated before:

- get_hash: accepts bytes as input and returns the derived hash.
- generate_keys: this code returns a public / private key pair.

FUNCTIONS - 2

```
def encrypt_data(key, plaintext):
    encryptor = PKCS1_OAEP.new(key)
    encrypted = encryptor.encrypt(plaintext)
    return encrypted

def decrypt_data(key, ciphertext):
    decryptor = PKCS1_OAEP.new(key)
    decrypted = decryptor.decrypt(
           ast.literal_eval(str(ciphertext)))
    return decrypted
```

- encrypt_data: this function receives a key and plaintext. Create a new encryptor. The function then encrypts and returns the plaintext.
- decrypt_data: this function receives a key and plaintext. Create a new decryptor. The function then decrypts and returns the ciphertext.

SIGNATURE - JAVA

```
#1
data=b'42'
hash1=get_hash(data)



#2
privatekey, publickey=generate_keys()
encrypted=encrypt_data(publickey, hash1)


#3
hash2=get_hash(data)


#4
hash3=decrypt_data(privatekey,
     ast.literal_eval(str(encrypted)))
```

In this code, you create a signature:

1. Create hash for data.
2. Generate the public / private keys. Encrypt the hash to create the signature.
3. Recreate a hash2 using the original data.
4. Decrypt the digital signature and recover a copy of the original hash (hash3).

VERIFICATION

```
if hash2==hash3:
    print("Verified")
else:
    print("Not Verified")
```

This code compares the original and decrypted hash. If the same, the digital signature is verified. If not, the digital signature is not verified.

POP QUIZ: MANUAL DIGITAL SIGNATURE

Is there anything wrong in the previous code?

5 MINUTES

CRYPTO.SIGNATURE

Creating a digital signature, as shown, requires several steps. For this reason, it is error prone.

The crypto signature module exports functionality that makes creating a digital signature easier and more convenient.

- pkcs1_15.new: this function returns a signature object:PKCS115_SigScheme.
- PKCS115_SigScheme.sign: the sign function signs a hash.
- PKCS115_SigScheme.verify: verifies a digital signature. If not verified, an exception is thrown.

CRYPTO.SIGNATURE - 2

Here are the ingredients of a creating and verifying a digital signature:

- Plaintext
- Digest
- Public key
- Private key

DIGITAL SIGNATURES - IMPLEMENTATION

```
from Crypto.Signature import pkcs1_15
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA


from Crypto import Random


def generate_keys():
      modulus_length = 256*4
      privatekey = RSA.generate(modulus_length,
            Random.new().read)
      publickey = privatekey.publickey()
      return privatekey, publickey
```

The pkcs1_15 module contains the components for creating and verifying a digital signature.

In this sample code, the pkcs1_15 module is imported.

DIGITAL SIGNATURES – IMPLEMENTATION - 2

Much of the code for creating a digital signature with the pkcs1_15 library is similar to the code shown previously.

1. Create public / private key in the generate_keys function.
2. Hash the message with SHA256.new function.
3. Create an instance of the signing object with the private key.
4. Call the sign function on the signing object to create a digital signature from the hash.

```
privatekey, publickey=generate_keys()
message = b'To be signed'

h = SHA256.new(message)
signature = pkcs1_15.new(privatekey).sign(h)
```

DIGITAL SIGNATURES – IMPLEMENTATION - 3

This is the code to verify the signature..

1. Recreate the hash from the original message with SHA256.new function.
2. Create a new instance of the signature object using the public key.
3. Verify the signature
4. If verification fails, throw an exception.

```
h2 = SHA256.new(message)
try:
        pkcs1_15.new(publickey).verify(h2, signature)
        print("The signature is valid.")
except (ValueError, TypeError):
        print("The signature is not valid.")
```

HMAC

HMAC is a hash that is seeded with a secret key. The two elements of HMAC.

- Hashing algorithm
- Authentication

HMAC can use a variety of hashing algorithms:

md5	sha1
sha224	sha256
sha384	black2b
black2s	and others

HMAC – IMPLEMENTATION

Here is the implementation for HMAC in Python.

Call the hmac.new function to great new hashing used the password and the designated hashing algorithm.

hmac.new(plaintext, password, hashing algorithm)

```
import hashlib
import hmac

hmac1 = hmac.new(b'info 1', b'1234', hashlib.md5)
hmac2 = hmac.new(b'info 2', b'1234', hashlib.sha1)

print("MD5", hmac1.hexdigest())
print("SHA1", hmac2.hexdigest())
```

HMAC – JAVA

Here is the implementation for HMAC in Java.

```
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import org.apache.commons.codec.binary.Base64;

public class ApiSecurityExample {
  public static void main(String[] args) {
    try {
     String secret = "secret";
     String message = "Message";

     Mac sha256_HMAC = Mac.getInstance("HmacSHA256");
     SecretKeySpec secret_key = new SecretKeySpec(secret.getBytes(), "HmacSHA256");
     sha256_HMAC.init(secret_key);
```

HMAC – JAVA - 2

```
      String hash = Base64.encodeBase64String(sha256_HMAC.doFinal(
                  message.getBytes()));
      System.out.println(hash);
     }
     catch (Exception e){
      System.out.println("Error");
     }
    }
}
```

DLL PRELOADING ATTACK

Stephen King wrote, “Sooner or later, everything old is new again.”

For that reason, adversaries are returning to the DLL Preloading Attack, which is a historical threat that has made a comeback.

The DLL Preloading Attack is simple to implement.

DLL PRELOADING ATTACK - 2

The attacker relies on probing for a dynamic link library (DLL) to launch an alternate library as a trojan. When loaded at runtime without a fully qualified name, Windows probes the location of a dynamic link library:

1. Application directory
2. System directory
3. System directory (16-bit)
4. Windows directory
5. Current directory

DLL PRELOADING ATTACK - 3

When the dynamic link library is launched at runtime, without the full path, the operating system will load the first similarly named library based on probing. Adversaries can leverage DLL Preloading as the first step of another vulnerability, including an Escalation of Privilege attack.

DLL Preloading attacks have been effective on a variety of well-known products, including Internet Explorer (IE7). Here is the story: https://bit.ly/3p5Kt2l

LAB 5 – BLOOM FILTER

BLOOM FILTER

A bloom filter is a hash driven algorithm. It is an algorithm that confirms a probable membership. Conversely, the algorithm confirms that a value is not a member.

Bloom filters are an example of non-secure cryptography but a high-performance methodology for identifying members of list, even if not sorted.

You can use non-secure algorithms with bloom filters, such as MD5 and SHA1. Bloom filters typically rely on 2 or more hash algorithms.

BLOOM FILTER – 3

With bloom filters, arbitrarily set the size of the filter. Larger filters more accurately assert membership but performance is slower.

You typically choose two or more hash algorithms to create distribution of bits within the filter. The more algorithms the more accurate the bloom filter. Two or three different algorithms is reasonably accurate for a modest list.

SETTING BITS

Each bit of the bloom filter is a binary bucket.

When added, hash the element. Get the remainder of the hash using the number of bits in the filter. The result is the next bit set in the filter.

1		**1**				**1**			

VERIFYING

How do you confirm membership?

Calculate the bits of the element using the hashing algorithms.

If the bits are found in the filter, the element is probably an element.

If a bit is missing (an empty bucket) in the filter, the element is definitely not a member.

BLOOM FILTER – 3

Create a new project called bloom and add a new source file.

- The filter for this example is 8 bits.
- We will use two hashing algorithms: MD5 and SHA1.

In the source file, add this code.

```
#Import libraries
import hashlib
import hmac

def set_bit(value, bit):
      return value | (1<<bit)

def get_bit(value, bit):
      return value & ~(1<<bit)
```

CALCULATE_BITS

Create a function calculate_bits. It accepts a value:

1. Create the first hash using hashlib.md5
2. Create the second hash using hashlib.sha1
3. Convert the first hash to an integer value as shown already in the course. Calculate the bit (hash % 8) : bit1.
4. Convert the second hash to a binary value as shown already in the course. Calculate the bit (hash % 8) : bit2.
5. Return bit1, bit2

ADD_TO_BLOOM(FILTER, BIT1, BIT2)

Create a function add_to_bloom. This is the prototype: add_to_boom(filter, bit1, bit2).

1. Call the set_bit function and set bit1 in the filter. Assign the new value back to filter.
2. Call the set_bit function and set bit2 in the filter. Assign the new value back to filter.
3. Return the filter.

MEMBERSHIP(FILTER, BIT1, BIT2)

Create a function membership. This is the prototype: membership(filter, bit1, bit2).

The membership function returns True or False to indicate membership, based on the bits. Remember, each bit represents a hashing algorithm: MD5 and SHA1. Both bits must be set to imply membership.

1. Confirm whether the bit is set.

```
f1=bin(filter)[8-b1]

f2=bin(filter)[8-b2]
```

2. If either bit is not set (empty bucket),returns false.
3. If both bit set, return true.

ADD AN ELEMENT TO THE FILTER

```
data = b'bob young'
print("data", data)


b1, b2=calculate_bits(data)
filter=add_to_bloom(filter, b1, b2)
result=membership(filter, b1, b2)
print(result)
```

This code adds an element to the bloom filter.

1. Create a new element called “bob young”.
2. Identify the bits for this element with calculate_bits.
3. Add the bits to the bloom filter with add_to_bloom.
4. Confirm the membership “bob young” using the membership function.

CONFIRM MEMBERSHIP

Confirm the membership of a second item *not* in the list. Use the code on the previous slide as a template.

1. Create a second element: 'Your name'.
2. Calculate the bits for the second element.
3. Check the membership of this element, using the new bits.
4. Display the results.

Run the program and confirm the results.

Add more elements if you desire and the check membership.

Have fun!

Lab completed

MORE CRYPTOGRAPHY

BCRYPT, ELLIPTIC CURVE CRYPTOGRAPHY, AND GCM

SECURE PASSWORDS

Securing a password is a frequent requirement for trustworthy applications.

The Bcrypt library exports the capability to secure passwords. Various languages support this library, including Python, Java, Go, and more.

Secure passwords created with Bcrypt where the salt can be iterated multiple times to prevent brute force attacks. This does have a performance cost.

SECURE PASSWORDS – STEPS

Here are the steps for creating a secure password with the Bcrypt package.

1. Convert the password to a byte slice
2. Hash the password with the brcypt.GenerateFromPassword function.
3. Now you can safely store the password
4. The password can be verified as correct using the bcrypt.CompareHashAndPassword function.

BCRYPT – EXAMPLE

```
import bcrypt


password = "password"

password = password.encode('utf-8')


hashedPassword = bcrypt.hashpw(

      password, bcrypt.gensalt())

print(hashedPassword)


if bcrypt.checkpw(password,
           hashedPassword):

    print("Matched")

else:

    print("Not matched")
```

In this code:

1. Define the password
2. Encode the password as Unicode
3. Create a hash for the password with a salt.
4. Bcypt.gensalt creates the random salt.
5. Call checkpw function to validate a password against the hash.
6. If validated, checkpw returns true. Otherwise, false is returned.

BCRYPT – EXAMPLE - 2

```
import bcrypt

password = "password"

password = password.encode('utf-8')

# salt = bcrypt.gensalt(rounds=16)

hashedPassword = bcrypt.hashpw(password, salt)

print(hashedPassword)

if bcrypt.checkpw(password, hashedPassword):
    print("Matched")
else:
    print("Not matched")
```

This code is similar to the previous code except that the cost has been increased to make the password hash more secure.

Increased cost can however impact performance, as demonstrated in this example.

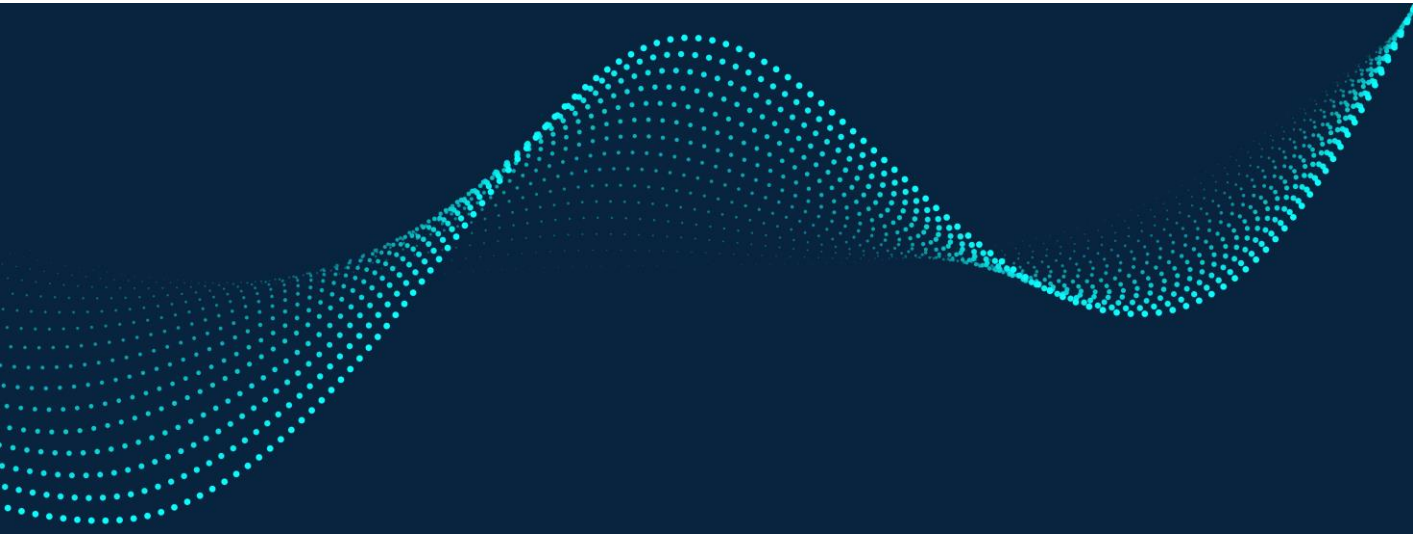
Set the cost with bcrypt.gensalt. The default cost is 12. You then use the resulting salt in the hashpw function.

ELLIPTIC CURVE CRYPTOGRAPHY

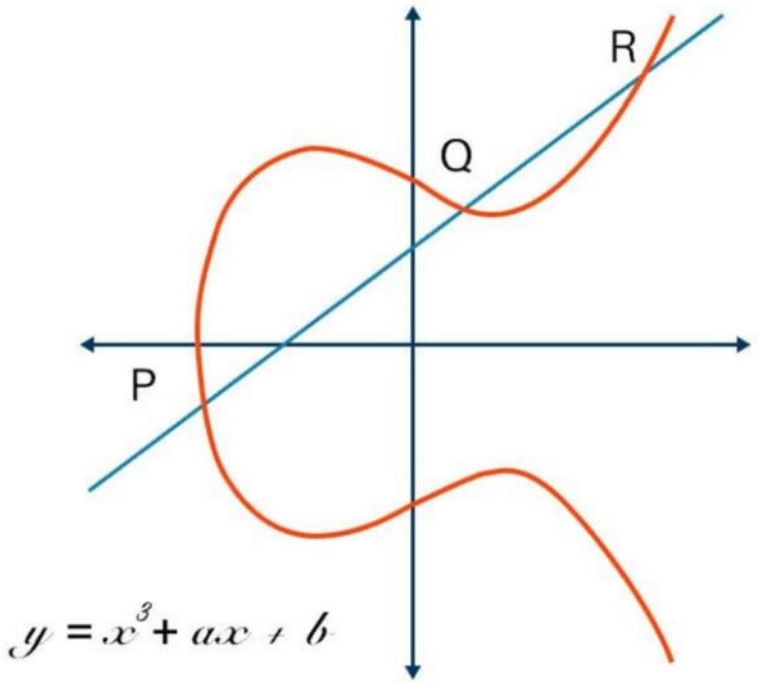
Elliptic Curve Cryptography (ECC) is modern public-key encryption based on mathematical elliptic curves. This is an improvement on first generation public-key encryption, such as RSA, which is based on prime numbers.

ECC is lightweight when compared to RSA. Benefits include:

- Smaller keys
- Quicker
- More efficient



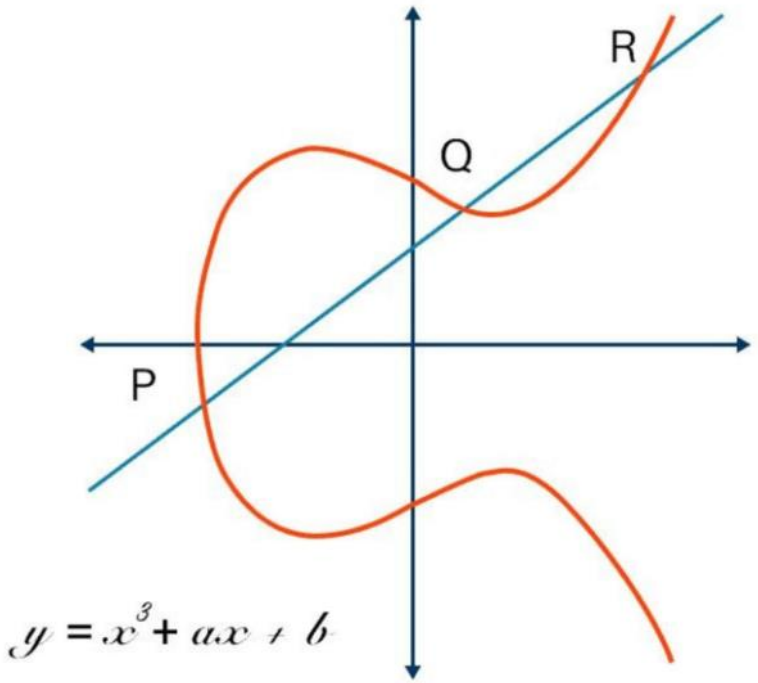
ELLIPTIC CURVE CRYPTOGRAPHY (ECC)



Elliptic Curve Cryptography is a mathematical curve based on this formula: $y^2 = x^3 + a*x^2 + b$, where 'a' and 'b' are constants.

Multiplying points will produce another point on the curve. The new point is difficult to reverse, even if you know the original point. For that reason, elliptic curves algorithms are relatively easy to perform, and extremely difficult to reverse.

ELLIPTIC CURVE CRYPTOGRAPHY (ECC) - 2



For Elliptic Curve Cryptography, a point is chosen on the elliptic curve and multiplied with itself '*n*' times. You can easily find the destination point with a starting point and '*n*'. However, the reverse, with the destination point and starting point, it is difficult to find the value of 'n'.

- The public key is a combination of the starting and ending point.
- The private key is the number of steps from the starting and ending point.

Here is a detailed description of Elliptic Curve Cryptography curves:

https://bit.ly/395l7uH

ELLIPTIC CURVE CRYPTOGRAPHY - IMPLEMENTATION

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import ec
private_key = ec.generate_private_key(
    ec.SECP384R1()
)
data = b"this is some data I'd like to sign"
signature = private_key.sign(
    data,
    ec.ECDSA(hashes.SHA256())
)

data = b"this is some data I'd like to sign"

public_key = private_key.public_key()
public_key.verify(signature, data, ec.ECDSA(hashes.SHA256()))
```

In this example, we mainly rely on the cryptography.hazmat.asymmetric.ec module for access to elliptic curve cryptography within Python for signatures. .

1. Create a private key with the generate_private_key function.
2. Generate a signature with the *private_key*.sign function. The parameters are:
   - The plaintext
   - The signing and hash algorithm
3. Later extract the public key
4. The *public_key*.verify function confirms the signature. If not confirmed, there is an exception.

ANOTHER ECC EXAMPLE

```
func main() {

    pubkeyCurve := elliptic.P256()
    privatekey, _ := ecdsa.GenerateKey(pubkeyCurve, rand.Reader)

    var pubkey ecdsa.PublicKey
    pubkey = privatekey.PublicKey

    l := big.NewInt(0)
    r := big.NewInt(0)
```

This section of code includes the necessary setup:

- The elliptic.P256 function returns a Elliptic Curve Cryptography algorithm that implements secp256r1
- Generate a private key with the ecdsa.GenerateKey function
- Derive the public key from the private key
- Create two big integers that will hold the digital signature

GALOIS/COUNTER MODE (GCM)

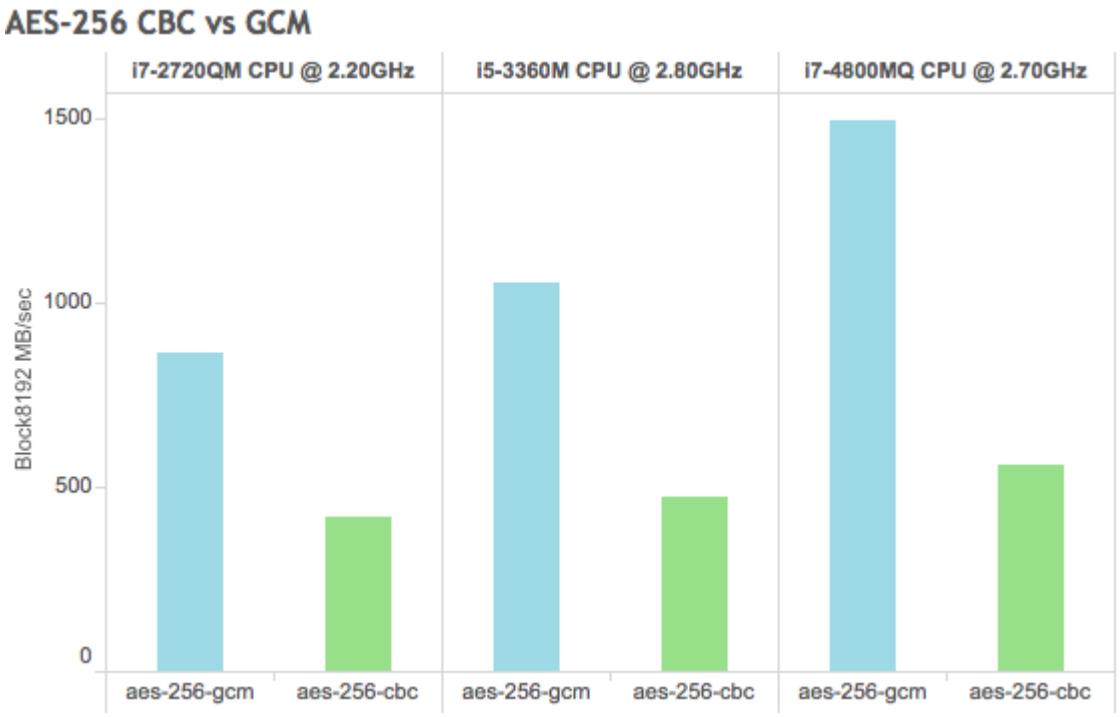
Galois Counter Mode (GCM) is a cryptography standard and a block cipher for symmetric key encryption. The primary advantage is efficiency and performance. For this reason, it is popular in environments where minimal resources are available, such as mobile devices.

AES-GCM is the most popular implementation.

For example, here is documentation on Android devices for developers. AES-GCM is an integral part of this document.

https://bit.ly/396iZ5X

GALOIS/COUNTER MODE (GCM) - 2



https://bit.ly/2XivGYC

Here is a performance comparison between AES-GCM and standard AES for 256 symmetric encryption.

GSM – EXAMPLE CODE

```
from Crypto.Cipher import AES


data=b'42'

key = b'Sixteen byte key'

cipher = AES.new(key, AES. MODE_GCM)


nonce = cipher.nonce

ciphertext, tag = cipher.encrypt_and_digest(data)
```

Here is sample code for using GCM or other block cipher mode. In this example, we will encrypt data.

1. Define a variable, data, that contains binary information.
2. Next, define a 16-byte key.
3. Create a new cipher initialized with the key and block cipher mode.
4. Set the nonce as an additional unique factor. The nonce is optional with some block chain mode.
5. Call the encrypt_and_digest function to encrypt the data. It returns the ciphertext for the data and the MAC tag.

GSM – EXAMPLE CODE - 2

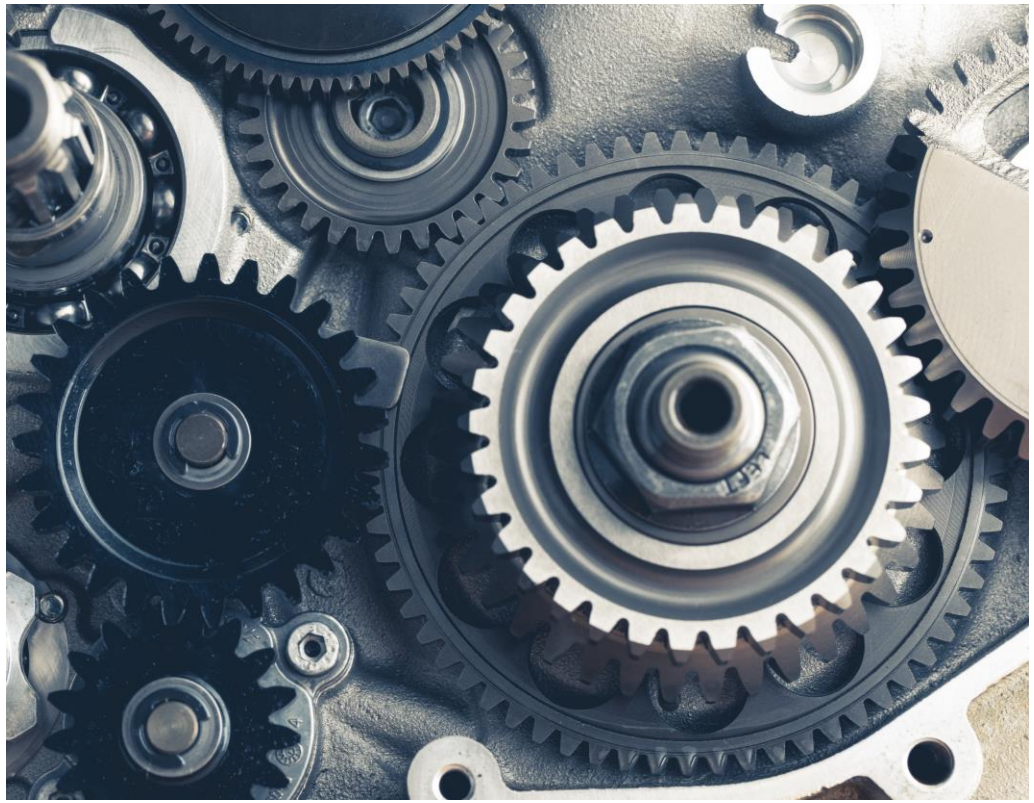
```
key = b'Sixteen byte key'
cipher = AES.new(key, AES. MODE_GCM, nonce=nonce)
plaintext = cipher.decrypt(ciphertext)
try:
    cipher.verify(tag)
    print("The message is authentic:", plaintext)
except ValueError:
    print("Key incorrect or message corrupted")
```

In this code, you will decrypt and verify the validity of the cipher.

1. Create a new cipher initialized with the key and block cipher mode. This creates the same cipher as used for encrypting.
2. Call decrypt to decrypt the ciphertext and return plaintext.
3. The verify function confirms the credibility of the data. If not valid, throw exception.

https://bit.ly/3HvtKPy

API ATTACKS



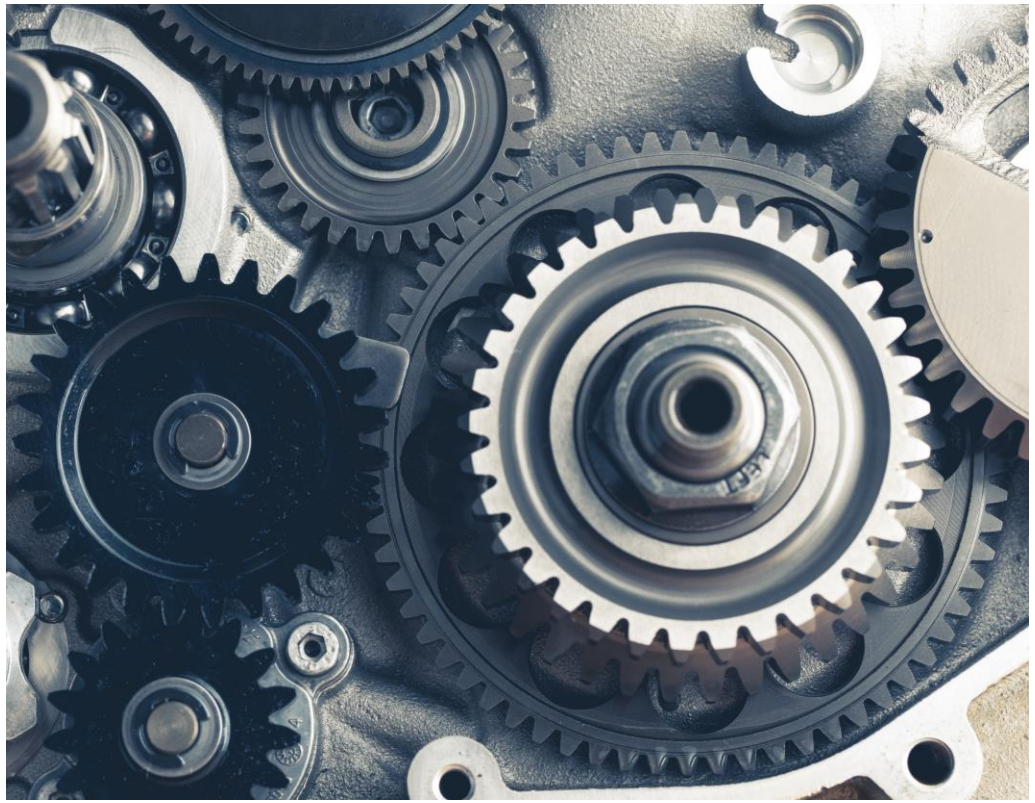
U. S. Postal Service

An attacker exploited a vulnerability in a U. S. Postal Service application. The application tracked mail in real time allowing users to retrieve delivery updates. The vulnerability allows the current user to view data from other accounts, including personal data. 60 million accounts were exposed.

The vulnerability was a SQL injection attack, where the attacker queried the backend database with wildcard parameters.

Even worse, a white hat notified the U. S. Postal Service of the vulnerability more than a year earlier.

API ATTACKS - 2



Steam Marketplace

This is an attack that was not actually exploited but discovered and reported by a white hat for a bounty.

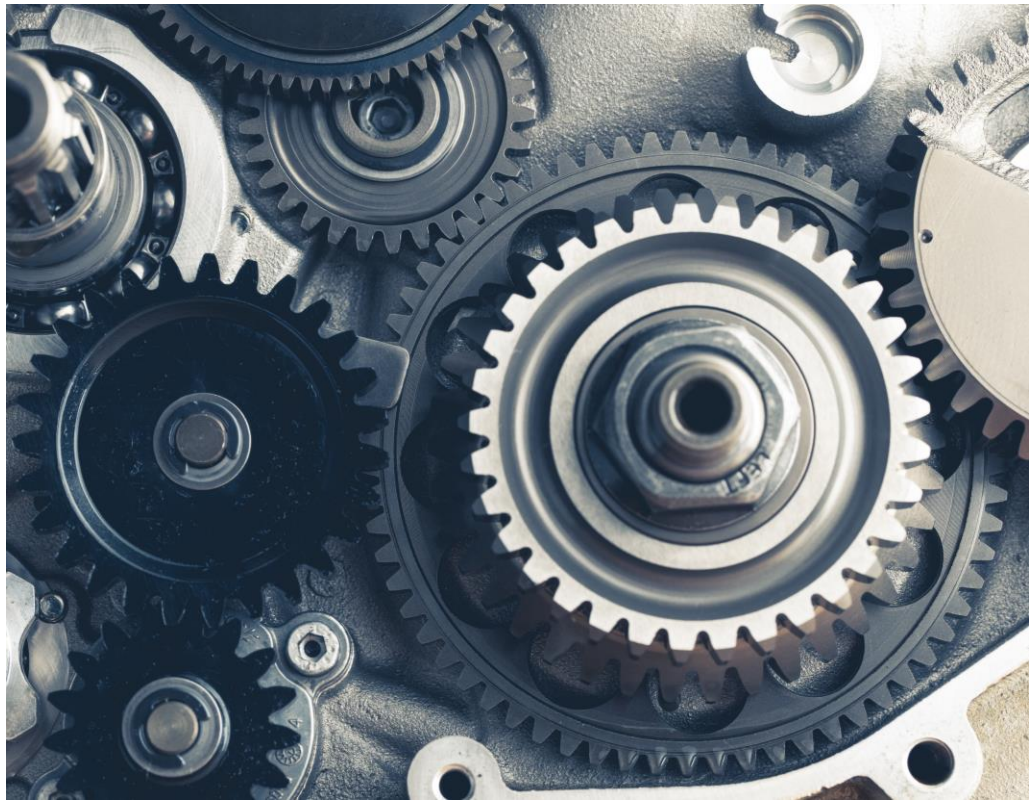
The vulnerability was only accessible to Steam developers and within the developer portal.

Internal or semi-internal applications, such as those found in the developer portal, often are not exposed to the same security inspection as a public application.

For the Steam Marketplace, calling the assignkeys API, with zero as the key count, returns a bucket of valid activation keys.

As a demonstration, the white hat stole 36,000 activation keys. Each key was worth $9.99.

API ATTACKS - 3



Facebook

Similar to the previous attack, the vulnerability was found in the Facebook developer APIs. The attackers were able to obtain user profile data, such as name, gender, and partial addresses. This affected 50 million Facebook users.

The vulnerability (actually three) was found in the video uploader. When using the *View As* feature to emulate a user, the video uploader would unexpectedly start and create a user access token for the target user. The developer could then login to the user account using the access token and view unauthorized sensitive data.

Fortunately, Facebook uncovered the problem while auditing for malicious behavior.

Lab completed

Excellent

You have completed the course successfully!

“Develop a passion for learning.”