Cryptography Gearup

Goals

Attack some insecure "systems" of the fictional Blue University

- Learn cryptographic principles by attacking some insecure systems
- Understand what attacks can look like
- Learn why you should never implement your own crypto
  - ⇒In reality: best practices, libraries are your friend!

Overview

NOTES AT END

1660 students	CS1620/2660 students
1. Grades 2. Ivy 3. Passwords	(Everything from part 1) + Padding

All problems in each part are **separate/self-contained** => you can work on them in any order

=> Move on if you get stuck!

Getting started: dev environment, your repo

Clone your repo here:

```
- ...
 |--DEV-ENVIRONMENT
 |  |--docker/
 |  |--home/
 |  |  |--.etc/
 |  |  |--p01-cryptography-yourname/  # <--- Clone your stencil here!
 |  |--run-container
 |  |-- ...
 ...
```

Repository layout

- One directory for each problem => "problem directory
- Some problems have multiple stencils

```
p01-cryptography-yourname
   |- grades/          # <--- Problem directory for ivy
   |  |- stencil/    # <--- Stencil code for grades
   |      |- go/
   |      |   |- STENCIL.md  # Guide for using this stencil
   |      |   |- sol.go
   |      |   |- ...
   |      |- python/
   |      |   |- STENCIL.md
   |      |   |- ...
   |      |- ...
   |- ivy/            # <--- Problem directory for ivy
   |  |- stencil/  # <--- Stencil code for ivy
   |      |- ...
```

How the stencils work

Grades, Ivy, Padding have stencils in Python and Go => you pick which one

To start: must copy the stencil you want to the directory for that problem:

```
cs1660-user@container: ~/repo$ cp -Trv stencil/grades/python grades
```

Useful stuff for each stencil

- STENCIL.md: Super helpful stuff about this stencil
- (Go only) Makefile: run make to compile

What you should submit

For each problem, your repo should have:

- Your solution program (usually `sol`)
- (Any other required stencil files)

- README
  - Describe the attack, how you did it, what you might change
  - See handout for per-problem details
  - Anything else we should know (what you tried, feedback, issues, etc.)

Your README is important—we're interested in your discussion/analysis!

Grades

You have

- Database of grades, encrypted with ECB mode
- Some statistics
  - 100000 students
  - 30 grades/student
  - Distribution of all grades: 50% As, 30% Bs, …

What can you learn from this?

Grades

What you have

- Database of grades, encrypted with ECB mode
- Some statistics
  - 100000 students, 30 grades/student
  - Across all grades: 50% As, 30% Bs, …

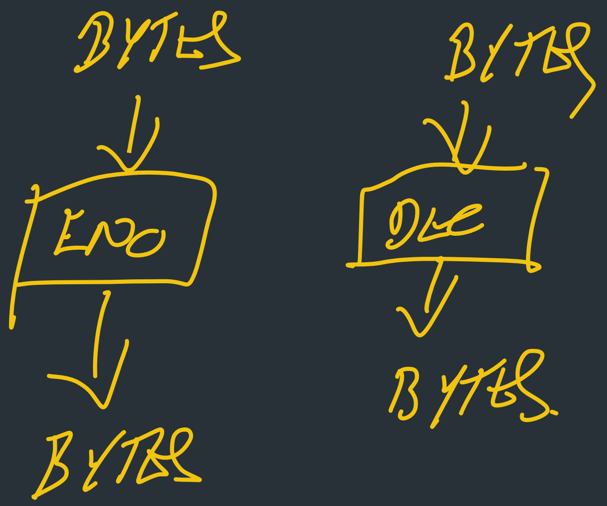
What you need: script to gather some info about database, without decrypting anything

eg. “What ciphertext block corresponds to grade of A?”

(see handout for full list of questions)

Types and bytes

- What type is a ciphertext? It's just bytes
- At this level, consider data as just an array of bytes, rather than as string/integer/etc data



Types and bytes

- What type is a ciphertext? It's just bytes
- At this level, consider data as just an array of bytes, rather than as string/integer/etc data

```
# Get a string as bytes
str_as_bytes = “hello”.encode("utf-8") # b'hello’


# Construct arbitrary bytes
b = bytes([0xaa, 0xbb, 0xcc, 0xdd])
b'\xaa\xbb\xcc\xdd’


# Common to print in “hex-encoded” form
b.hex()              # ‘aabbccdd’
str_as_bytes.hex()   # ‘68656c6f’
```

See examples for hex strings in handout

Ivy Wireless

- Attacking a weak protocol, a way to exchange data between parties
- Ivy Protocol: fictional wireless network, want to recover wifi key k

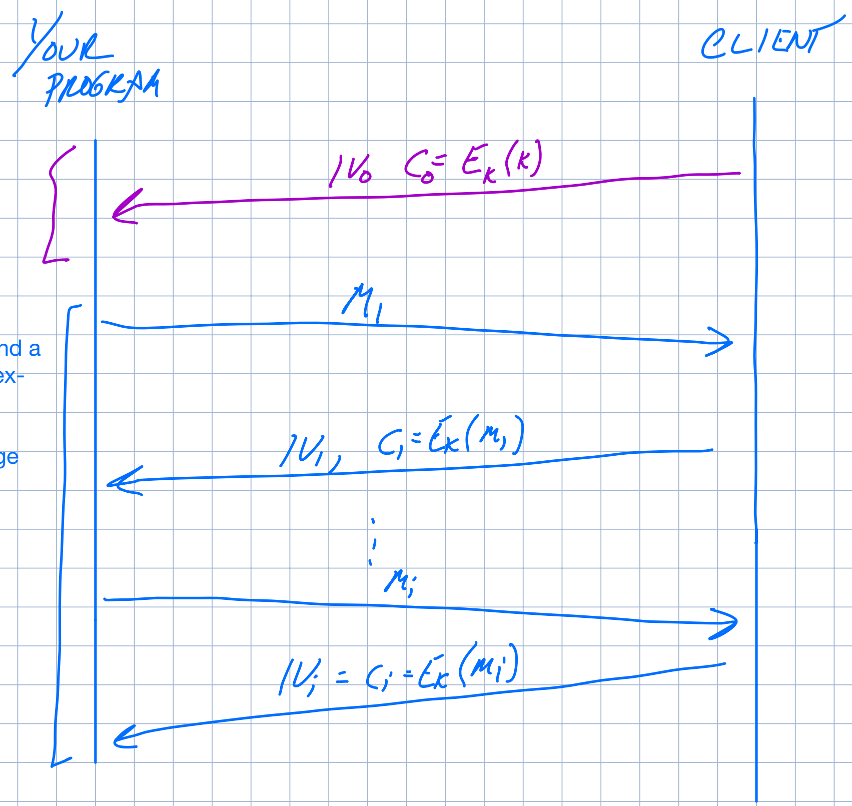
You have

- *Encryption oracle: given plaintext m, returns (iv, c)*
- *Can send as many plaintexts as you want => chosen plaintext attack*

HOW YOU'LL INTERACT WITH THE IVY CLIENT

Setup phase: on startup, client sends encrypted key

Normal operation: you send a plaintext message (as a hex-encoded string), client response with an (IV, ciphertext) for that message



Your goal: recover the key k

The client is called an *encryption oracle*—can encrypt as many plaintexts as you want and send the output.

Idea: if you can send as many plaintexts as you want, can you learn something about the key?

See the handout for more details on the protocol and encryption mechanics.

Demo

Passwords

Implement two methods of “secure” password storage

Operates on ”databases” of passwords…

```
{
 "method": "plain",
 "users": {
  "user0399": {
   "password": "7vxd"
  },
  "user0449": {
   "password": "hb5s"
  },
  . . .
 }
}
```

← CLEARTEXT PASSWORD

<u>Our “super secure” password policy</u>

- Passwords are <= 4 characters long
- Contains only lower case letters + numbers

Passwords, "better"

More secure to store a hash of the user's password, h = hash(password)

```
{
 "method": "sha1-nosalt",
 "users": {
  "user3234": {
   "password": "1cc33637bdd3b586d89d259d719e8ad9a5e4f42e"
  }
}
```

H(PW), NOT A PASSWORD

But… can we guess it?

↳ YES! => INPUT SPACE IS SMALL

4 CHARS: A-Z, 0-9 = 36 POSSIBLE VALUES

36 36 36 36 = $36^4 \approx 2^{20}$

With such a restrictive password policy, this is feasible to quickly compute on a modest system. "Real" passwords have higher complexity, but there are optimizations and heuristics that can improve guessing (see lectures for details).

Passwords

Implement two methods of “secure” password storage:

- Single hash (sha1-nosalt)
- Salted hash (sha1-salt4)

Two stencil programs

- login: Simulate a login
- pwfind: Crack all passwords in the database

Assignment guides you through extending each program and using them to crack passwords => comment on performance tradeoffs

SEE RECORDING FOR DEMO!

Questions

- We're here to help! Ask us in hours/on Ed
- See FAQ/reading list post for common issues!
- Collaborative hours: just come and work, collaborate with others!

Padding (CS1620/CS2660 ONLY)

Based on real attack on TLS (the "s" in "https")

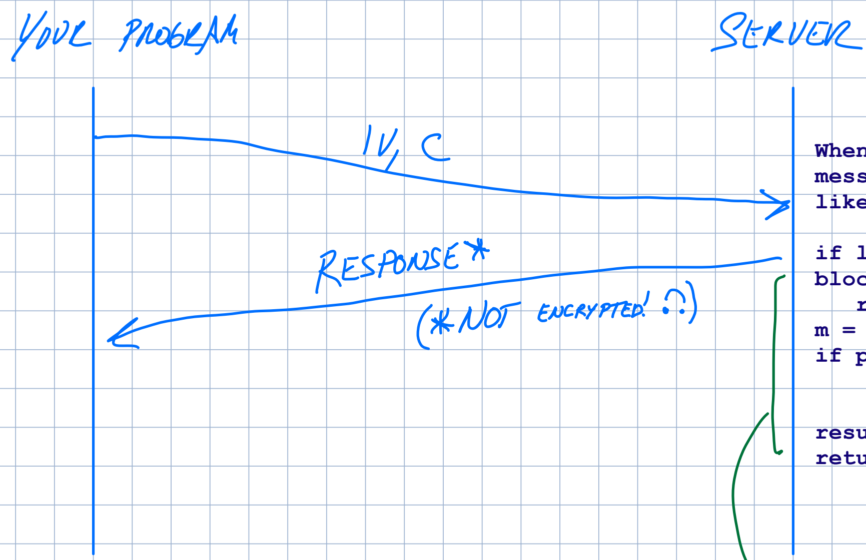
Setup: "grading server"

you send (iv, ciphertext) encrypted with CBC mode

get back plaintext message, or error

Turns out the error feedback is enough to break the system!

How to think about padding



When server receives a message, it does something like this:

```
if len(c) not multiple of
block size (16 bytes):
    return error
m = decrypt(c)
if paddingIsInvalid():
    return "incorrect padding"

result = run_command(m)
return result
```

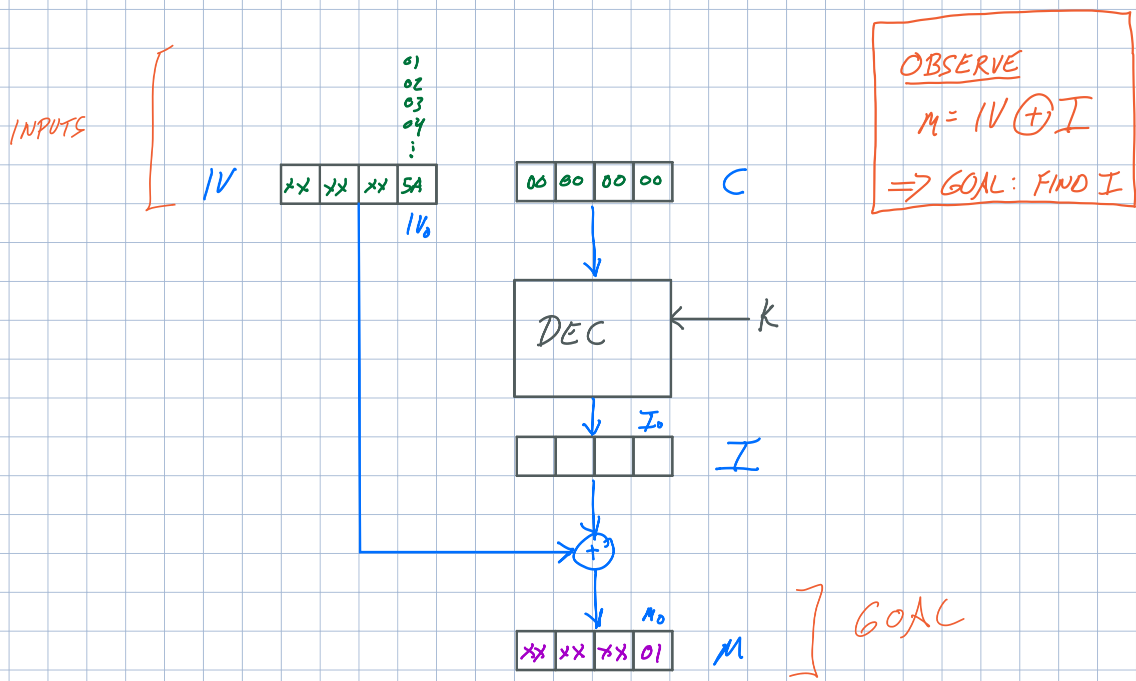
(might also be error if invalid command)

Since these are different errors, can determine how far we get!

PADDING: THE ATTACK

LET'S CONSIDER ONE BLOCK OF THE DECRYPT PROCESS

Idea: can find an (IV, ciphertext) pair such that m ends in 0x01 => this would have valid padding, and thus return a different error during decryption than others
- Set c to a constant value
- Try different values for last byte of IV until you know message had valid padding



From this, we can figure out last byte of intermediate state I:

$$IV_0 \oplus I_0 = M_0$$

$$5A \oplus I_0 = 01$$

$$I_0 = 5B$$

So now what happens if we set the last byte of the IV to I0?

$$IV_0 \oplus I_0 = M_0$$

$$5B \oplus 5B = 0$$

This means we can set this byte of the plaintext to zero! This is great because it means we can set this byte to <u>any value</u> by xor'ing something it into the IV:

EG. SET M_0 TO 0x02

$$\underbrace{02 \oplus 5B}_{IV_0} \oplus \underbrace{5B}_{I_0} = 02$$

So now what? Here's a sketch:

— Can now set last byte of m to an arbitrary value—how about 0x02, for a messate with 2 bytes of padding? Then, try same attack again for second to last byte to determine next-to-last byte of I.

— Repeat for larger amounts of padding until you have recovered entire value of I.

— With I fully recovered, how can you use this to get m to decrypt to a plaintext of your choosing?
=> Per handout: try to send command "help"

—After sending help, you should get info about another command you can send. This command will need you to send more than one ciphertext block
=> Can't just set c to zero anymore! Consider how c is used for next block…