HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
KHOA AN TOÀN THÔNG TIN

BÁO CÁO BÀI TẬP LỚN
HỌC PHẦN: MẬT MÃ HỌC CƠ SỞ
MÃ HỌC PHẦN: INT1344

XÂY DỰNG BÀI THỰC HÀNH
MÔ PHỎNG TẤN CÔNG BEAST
BEAST_LAB

Sinh viên thực hiện:

B22DCAT034 Trương Quốc Bảo

B22DCAT097 Lê Đức Giang

Giảng viên hướng dẫn: PGS.TS Đỗ Xuân Chợ

HỌC KỲ 2 NĂM HỌC 2024-2025

NỘI DUNG THỰC HÀNH

Tải bài thực hành:

imodule https://github.com/Baodeptraii/Labtainer/raw/refs/heads/main/imodule.tar

(Chú ý: sinh viên sử dụng MÃ SINH VIÊN của mình để nhập thông tin người thực hiện bài lab khi có yêu cầu, để sử dụng khi chấm điểm)

- Sinh viên khởi động bài lab:

Chạy lệnh:

labtainer -r beast_lab

(Chú ý: sinh viên sử dụng <TÊN_TÀI_KHOẢN> của mình để nhập thông tin người thực hiện bài lab khi có yêu cầu, để sử dụng khi chấm điểm.)

Sau khi khởi động bài lab, một container hiện lên sinh viên thực hiện làm theo yêu cầu.

TASK 1: Cài đặt và kiểm tra môi trường

- Kiểm tra các file đang có bằng lệnh: ***ls***
- Bài lab gồm 6 file python dùng để mô phỏng quy trình tấn công BEAST như sau:

messenger.py: Mô phỏng tiện ích để nhập message và key từ người dùng, sinh ra file **alice_message.txt** để Alice đọc

alice.py: Đóng vai trò là victim, đọc message và key từ file **alice_message.txt** rồi gửi yêu cầu mã hóa tới Server. Cung cấp phương thức **forceRequestandIntercept**() để giả lập tấn công MITM.

server.py: Mô phỏng server để thực hiện mã hóa/ giải mã AES-CBC. Cung cấp API **httpRequestForEncryptedText**() để mã hóa dữ liệu, hộ trợ tùy chỉnh hoặc tạo vector IV ngẫu nhiên

beast_malice.py: Đóng vai trò là attacker, thực hiện tấn công BEAST, khai thác lỗ hổng trong AES-CBC khi IV có thể dự đoán được. Sử dụng kỹ thuật brute-force từng byte để tiến hành giải mã.

cbc.py: Minh họa mã hoá CBC theo cách cũ với vector IV là bản mã của khối mã hoá trước, do vậy dễ dàng bị tấn công BEAST

cbc_iv.py: Mã hoá CBC với vector IV được sinh ngẫu nhiên mỗi lần mã hoá, đảm bảo an toàn hơn và khó có thể tấn công BEAST.

- Các để chạy các file python cần phải đảm bảo phiên bản Python 3.* và cài đầy đủ thư viện hỗ trợ mã hóa.

Gõ lệnh :

python --version

để kiểm tra phiên bản.

Gõ lệnh :

pip install pycryptodome

để cài thư viện Crypto cho việc mã hóa/ giải mã.

TASK 2: Mã hoá CBC với IV được lấy từ bản mã trước

Giả sử người dùng Alice muốn truyền đi một thông điệp. Chạy file **messenger.py** để gửi đi thông điệp kèm khóa. Gõ lệnh :

python messenger.py

File sẽ yêu cầu người dùng nhập thông điệp: "*xin chao moi nguoi toi la* ***<họ và tên>*** *toi dang hoc o Hoc vien Cong nghe Buu chinh Vien Thong - PTIT".*

Tiếp tục nhập khoá là **<mã sinh viên>**.

(Thay thế ***<họ và tên>*** *bằng tên sinh viên, thông điệp phải đảm bảo đủ dài để có thể thấy các khối mã hoá khác nhau của thông điệp)*

Kết quả sẽ tạo ra file **alice_message.txt** chứa thông điệp và khóa. Thông điệp này sẽ là mục tiêu của cuộc tấn công.

Sinh viên có thể tùy chỉnh kích thước của khối mã hóa bằng lệnh:

sudo nano cbc.py

sudo nano cbc_iv.py

Mặc định *BLOCK_SIZE = 16*, sinh viên có thể tự tùy chỉnh kích thước sao cho hợp lí nếu cần thiết.

Chạy file **cbc.py** để thông điệp được mã hoá với phương pháp CBC:

python cbc.py

quan sát thấy thông điệp được chia thành các khối mã hoá, với **vector IV mã hoá của khối sau được lấy 1 phần từ chính bản mã ciphertext của khối trước**.

Ví dụ:

ciphertext C1:1679ef632d7be6292260b96df66ea2a92a351bf18656437aba5d7

➔ **IV2**: 92a351bf18656437aba5d7

Điều này chính là lỗ hổng khiến cho giao thức SSL/TLS bị tấn công BEAST nếu kẻ tấn công có thể tấn công MitM, vector IV có thể dễ đoán được do nó chính là bản mã của khối mã hoá cuối cùng trước đó.

TASK 3: Mã hoá CBC với IV ngẫu nhiên

Để an toàn hơn khi mã hoá, một biện pháp tạm thời để khắc phục lỗ hổng đó chính là tạo ra vector IV ngẫu nhiên cho mỗi lần mã hoá. Chạy file thực hiện mã hoá:

python cbc_iv.py

có thể thấy các khối mã hoá đã được mã hoá với IV khác nhau được sinh ngẫu nhiên, do đó kẻ tấn công khó có thể đoán được IV để thực hiện tấn công BEAST đảm bảo cho giao thực SSL/TLS an toàn hơn.

Tại sao IV quan trọng?

- Bảo mật: IV ngẫu nhiên ngăn chặn tấn công như BEAST.
- Tính duy nhất: Cùng plaintext + key nhưng IV khác → ciphertext khác.
- Không được lặp lại: IV chỉ dùng một lần để tránh rò rỉ thông tin.

(Lưu ý: Trong thực tế, luôn dùng IV ngẫu nhiên và không lặp lại (như TLS 1.2+). TLS 1.0 đã bị BEAST khai thác vì sử dụng mode CBC và IV không đúng cách.)

TASK 4: Thực hiện mô phỏng tấn công BEAST

Dữ liệu trong file **alice_message.txt** sẽ được Alice đọc và gửi yêu cầu mã hóa đến Server (**server.py**). Server sẽ trả về ciphertext (IV + dữ liệu đã mã hóa)

Để có thể chạy được file **alice.py**, hãy thực hiện việc gửi yêu cầu đến server. Gõ lệnh:

sudo nano alice.py

để chỉnh sửa. Hãy bỏ dấu **"#"** ở trước hàm **forceRequestandIntercept**() để có thể gửi yêu cầu mã hóa tới Server mô phỏng.

Giải thích:

Hàm **forceRequestandIntercept()** sẽ đóng vai trò là MITM để hỗ trợ tấn công BEAST. Hàm này sẽ cho phép attacker (Malice) "chèn" dữ liệu vào yêu cầu mã hóa của Alice và "bắt" Alice gửi yêu cầu mã hóa theo ý muốn nhằm thu nhập ciphertext phục vụ tấn công. Attacker can thiệp nhằm mục đích thay đổi plaintext trước khi mã hóa và kiểm soát vector IV giúp brute-force từng byte plaintext dựa trên ciphertext thu được. Trong thực tế, kẻ tấn công cũng sẽ thực hiện MITM ở bước này, tức là bước Alice gửi yêu cầu mã hóa đến Server.

Tiếp theo, cần cấu hình lại Server để lựa chọn phương thức mã hóa và cho phép thực hiện mã hóa. Gõ lệnh:

sudo nano server.py

để thực hiện cấu hình. Ta sẽ cho phép sử dụng CBC mode để mô phỏng tấn công, bỏ dấu **"#"** trước dòng :

cipher = AES.new(self._key_bytes, AES.MODE_CBC, iv)

Kéo xuống bên dưới ta thấy có hàm **httpRequestForEncryptedText**(), hàm này là chức năng chính của Server, đóng vai trò mô phỏng quá trình gửi dữ liệu qua HTTP và mã

hóa dữ liệu plaintext thành ciphertext bằng AES-CBC. Hàm sẽ nhận vào plaintext (dạng str hoặc bytes) và một IV tùy chọn và trả về ciphertext dạng bytes (gồm IV + dữ liệu đã mã hóa). Attacker sẽ lợi dụng IV có thể kiểm soát để đoán IV của các block mã hóa, tính toán XOR giữa IV, ciphertext, và plaintext đã biết nhằm triển khau brute-force từng byte của plaintext.

Sau khi cấu hình xong, ta có thể gửi yêu cầu mã hóa của Alice đến với Server và kiểm tra. Gõ lệnh:

python alice.py

để chạy. Output sẽ là bản mã của thông kiệp sau khi sử dụng chế độ AES-CBC. Attacker sẽ thu thập bản mã (ciphertext) này để thực hiện quá trình tấn công.

Sau khi cấu hình các file **alice.py** và **server.py** chính xác và không có lỗi, tiến hành mô phỏng tấn công BEAST bằng cách chạy file **beast_malice.py**. Gõ lệnh:

python beast_malice.py

Nhấn **"y"** để thực hiện tấn công. File sẽ thực hiện việc brute-force để đoán được từng byte của bản rõ. Vì ta đã có sẵn bản rõ (plaintext) trước đó, chỉ cần so sánh là sẽ đoán được đúng ký tự tại byte đang xét. Thông điệp được giải mã sẽ tiếp tục được tìm dần. Nhấn Enter khi được hỏi **"ok?"** để tiếp tục brute-force cho đến khi thu được hoàn toàn bản rõ trước đó.

Gõ lệnh :

sudo nano beast_malice.py

để hiểu rõ qua trình tấn công.

Giải thích quá trình tấn công chi tiết:

- Khởi tạo:
    - Attacker chọn Alice làm mục tiêu.
    - Khởi tạo *known_prefix = ""* (ban đầu chưa biết gì về plaintext).
- Lặp từng byte của plaintext: Với mỗi vị trí i trong plaintext:

Gửi yêu cầu mã hóa: *Prefix = known_prefix[1:]* (bỏ đi 1 byte đầu, giữ 15 byte đã biết). Thêm 1 byte từ plaintext tại vị trí i vào prefix.

Ví dụ: *Nếu known_prefix = "aaaaaaaaaaaaaaa"* (15 ký tự 'a'), Thì plaintext được mã hóa là*: "aaaaaaaaaaaaaaa" + plaintext[i].*

Nhận ciphertext:

- *ciphertext = IV + encrypted_data.*
- *target_block = ciphertext[16:32]* (block thứ 2, chứa mã hóa của prefix + 1_byte_plaintext).

Brute-force từng ký tự (0-255):

- Dự đoán *guess_char* và *tạo modified_block = known_prefix[1:] + guess_char*.
- Tính *xored_block = XOR(IV, prev_cipher, modified_block).*
- Gửi *xored_block* để mã hóa và kiểm tra xem ciphertext có khớp với *target_block* không.
- Nếu khớp → tìm được *plaintext[i] = guess_char*, cập nhật *known_prefix.*

- Lỗ hổng:

AES-CBC sử dụng IV để XOR với plaintext trước khi mã hóa. Nếu IV có thể dự đoán (hoặc bị kiểm soát), attacker có thể thao túng để khôi phục plaintext.

- Cách khai thác :

Attacker lợi dụng tính chất:

Ciphertext_block[i] = Encrypt(Plaintext_block[i] XOR Ciphertext_block[i-1])

bằng cách gửi nhiều yêu cầu mã hóa với prefix đã biết và brute-force từng byte, attacker dần khôi phục plaintext.

(Chú ý: Trong trường hợp checkwork thiếu Y ở phần CBC mode, sinh viên hãy thử chạy file ***messenger.py*** *với thông điệp ngắn hơn.)*

TASK 5: Thay đổi chế độ mã hóa

Sau khi thực hiện phần trên, ta đã biết kẻ tấn công lợi dụng điểm yếu của AES-CBC và IV để thực hiện quá trình brute-force. Thay vì sử dụng AES-CBC để mã hóa dữ liệu ta sẽ dùng chế độ khác để chống lại replay attack, đó là chế độ AES-GCM.

Gõ lệnh :

sudo nano server.py

để chỉnh sửa cấu hình sử dụng GCM, bỏ dấu **"#"** ở đoạn

cipher = AES.new(self._key_bytes, AES.MODE_GCM, iv)

và thêm lại dấu "#" trước CBC mode để tránh xung đột.

Tiến hành chạy lại file **beast_malice.py** một lần nữa và xem kết quả. Ta thấy thông báo không tấn công thành công dù đã brute-force hết toàn bộ chữ cái. Do chế độ GCM sử dụng IV ngẫu nhiên (12 byte khác với 16 byte của CBC) cho mỗi lần mã hóa và thêm cơ thế xác thực (Encryption + Authentication).

- Kết thúc lab:

Trên terminal khởi động lab, sinh viên sử dụng lệnh:

stoplab

Khi bài lab kết thúc, một tệp lưu kết quả được tạo và lưu vào một vị trí được hiển thị bên dưới stoplab.

- Sinh viên cần nộp file .lab để chấm điểm.
- Để kiểm tra kết quả khi trong khi làm bài thực hành sử dụng lệnh:

checkwork <tên bài thực hành>

- Khởi động lại bài lab: Trong quá trình làm bài sinh viên cần thực hiện lại bài lab, dùng câu lệnh:

labtainer -r beast_lab