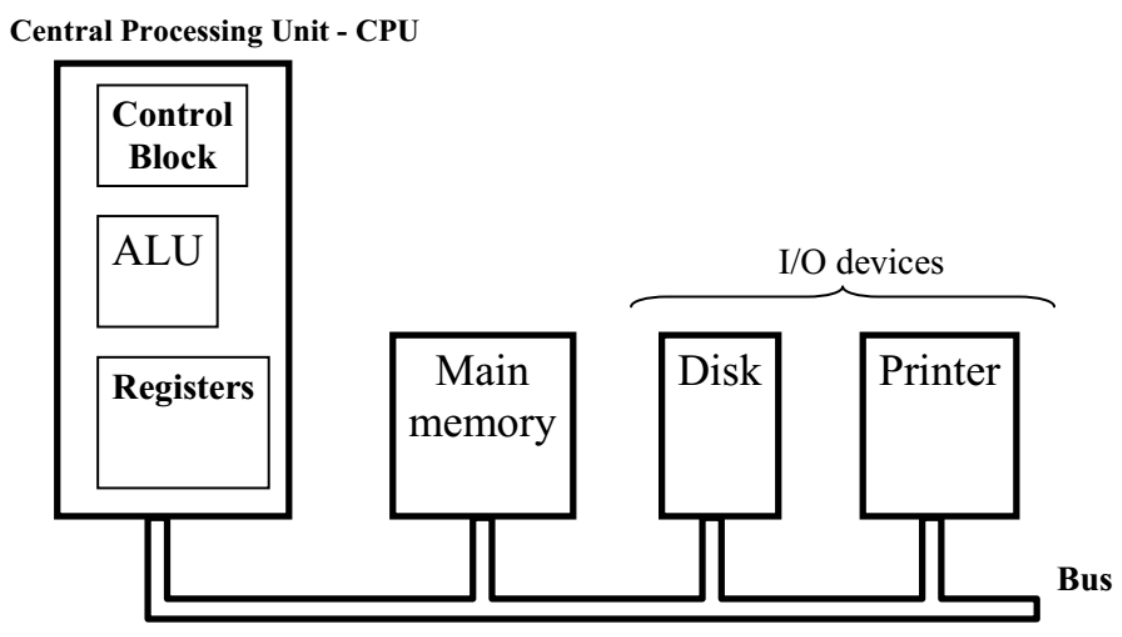
# Lý thuyết

## Kiến trúc máy tính

### Khái niệm

Kiến trúc máy tính là khoa học về việc lựa chọn và kết nối các thành phần phần cứng để tạo ra các máy tính đạt được yêu cầu về chức năng, hiệu năng và giá thành.



Kiến trúc dựa trên nguyên lý:

-Lệnh và dữ liệu được lưu trữ trong bộ nhớ đọc ghi, chia sẻ - 1 bộ nhớ duy nhất được sử dụng để lưu trữ cả lệnh và dữ liệu;

-Bộ nhớ được đánh địa chỉ theo vùng, không phụ thuộc vào nội dung nó lưu trữ

-Các lệnh của 1 chương trình phải đc thực hiện tuần tự

Qúa trính thực hiện được chia làm 3 giai đoạn:

-CPU đọc lệnh (fetch) từ bộ nhớ

-CPU giải mã và thực hiện lệnh; nếu lệnh yếu cầu dữ liệu, CPU đọc dữ liệu từ bộ nhớ

-CPU ghi kết quả thực hiện lệnh vào bộ nhớ

### Các thanh ghi

Thạnh ghi đa dụng: EAX, EBX, ECX, EDX

Thanh ghi xử lý chuỗi: EDI, ESI

Thanh ghi ngăn xếp: EBP, ESP

Thanh ghi con trỏ lệnh: EIP

Thanh ghi cờ: EFLAGS

Thanh ghi phân vùng: không còn được sử dụng ở kiến trúc 32 bit

### Bộ nhớ

Bộ nhớ chính: RAM

RAM chứa nhiều ô nhớ, mỗi ô 1 byte

RAM dùng để chứa 1 phần hệ điều hành, các lệnh chương trình, các dữ liệu, ...

Mỗi ô nhớ có địa chỉ duy nhất và địa chỉ này được đánh số từ 0 trở đi

### Mô hình bộ nhớ tuyến tính

Flat memory model

Là mô hình bộ nhớ mà các ô nhớ được đánh địa chỉ liên tiếp từ 0 đến MAXBYTE-1

Các mô hình khác:

-Mô hình phân đoạn

-Mô hình phân trang

Các chương trình 32 bit ở Protected Mode luôn sử dụng mô hình Flat

Mỗi chương trình có thể coi là nó có riêng 4GB RAM

Mã lệnh và dữ liệu nằm trong cùng 1 không gian địa chỉ

### Hướng ghi dữ liệu

Các hàm dữ liệu trong các ngôn ngữ lập trình luôn ghi dữ liệu vào RAM theo chiều tăng dần của địa chỉ

### Trật tự byte: little-endian

Các máy tính hiện đại sử dụng little-endian trong biểu diễn số

## Lỗ hổng phần mềm

### Khái niệm

**Lỗ hổng phần mềm** là khiếm khuyết trong thiết kế, lập trình phần mềm mà kẻ tấn công có thể lợi dụng để làm thay đổi hoạt động bình thường của phần mềm

Có lỗi chưa chắc đã có lỗ hôngr, có lỗ hổng thì chắc chắn có lỗi

### Phân loại lỗ hổng:

**-Theo nguyên nhân xuất hiện:**

+Do không kiểm tra đúng dữ liệu được cung cấp bởi người dùng

+Các lỗ hổng khác: Race condition, sử dụng thành tố mật mã không an toàn

**-Theo quy trình phát triển phần mềm:**

+Giai đoạn phân tích yêu cầu: do không có yêu cầu về tính năng an toàn

+Giai đoạn thiết kế: Thiết kế luồng thực thi không an toàn, Lựa chọn hoặc cho phép sử dụng các thành tố không an toàn

+Giai đoạn xây dựng: Do sử dụng các hàm, cấu trúc không an toàn; không kiểm tra thỏa đáng dữ liệu đầu vào

**-Theo mức độ nguy hiểm:**

+Loại C (Mức thấp): Lỗ hổng cho phép kẻ tấn công thực hiện tấn công từ chối dịch vụ (Dos)

+Loại B (Mức trung bình): Lỗ hổng cho phép người dùng nội bộ leo thang đặc quyền hoặc truy cập trái phép

+Loại A (Mức cao): Lỗ hổng cho phép kẻ tấn công truy cập trái phép từ xa vào hệ thống

## Stack

### Khái niệm

Stack là 1 vùng nhớ được hệ điều hành cấp phát cho chương trình khi nạp

Kích thước của stack được xác định khi biên dịch chương trình:

-Có thể chỉ định kích thước qua tham số cho trình biên dịch

-Mặc định khoảng 1Mb

### Chức năng

Chứa các biến cục bộ

Lưu địa chỉ trả về khi gọi hàm

Truyền tham số khi gọi hàm

Lưu trữ con trỏ “this” trong lập trình hướng đối tượng

### Thao tác

Trong x86, mỗi phần từ stack là 4 byte

Stack được quản lý qua ESP

Hai thao tác cơ bản: POP và PUSH

PUSH:

-Giảm giá trị của ESP: ESP =ESP-4

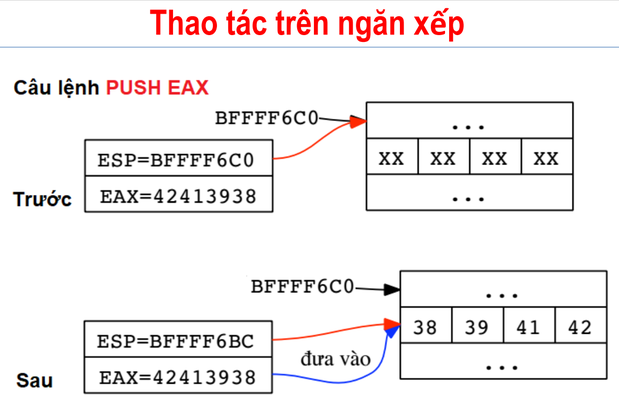
-Ghi đối số (4 byte) vào [ESP]

POP:

-Đọc 4 byte tại [ESP] vào đối số

-Tăng giá trị của ESP: ESP = ESP + 4

### Ví dụ



## Hàm và gọi hàm

### Khái niệm

Hàm là 1 đoạn chương trình con mà có thể được gọi bởi 1 chương trình khác để thực thi 1 nhiệm vụ bất định

Thông thường nếu hàm có trả về 1 kết quả, thì kết quả đó được đặt trong EAX trước khi hàm kết thúc

### Gọi hàm

Việc gọi hàm bao gồm:

-Nạp các tham số cần thiết

-Thực hiện lệnh CALL

Tham số có thể được nạp vào thanh ghi

-Ưu điểm: nhanh

-Nhược điểm: có thể không đủ thanh ghi

Cần kết hợp nạp tham số vào stack

Người xây dựng hàm có toàn quyền lựa chọn cách thức nạp tham số. Nhưng cần có quy ước chung:

-Mọi người hiểu mã của nhau

-Mọi người có thể sử dụng hàm của nhau

### Cấu trúc hàm

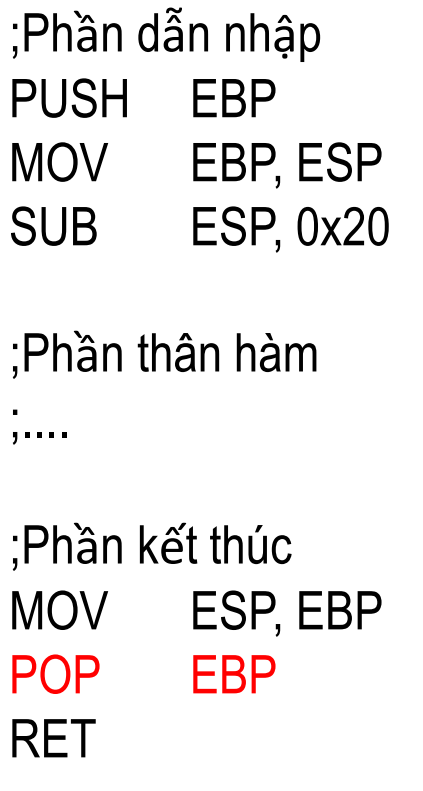
Hàm:

-Phần dẫn nhập;

-Phần thân hàm;

-Phần kết thúc;

### Ví dụ



## Shellcode

### Khái niệm

**Shellcode** là một đoạn mã máy được tiêm vào một phần mêmf có lỗ hổng và sau đó được thực thi cùng phần mềm đó

Shellcode trực tiếp can thiệp lên các thanh ghi và hướng thực thi chương trình, vì thế nó thường được viết bằng hợp ngữ rồi dịch sang mã máy

Shellcode được biểu diễn ở dạng chuỗi hex

**NOP Sled** là một chuỗi lệnh NOP (no-operation) có tác dụng dẫn dắt CPU tới đoạn lệnh mục tiêu mà ta mong muốn

Cần sử dụng NOP Sled khi ta không biết địa chỉ shellcode, mà chỉ áng chừng trong khoảng EIP +- delta, trong đó EIP-delta là địa chỉ của NOP Sled

Vị trí:

-Đặt phía trên stack frame: Shellcode ghi đè lên dữ liệu của caller

-Đặt trong stack frame: Buffer phải đủ lớn để chứa payload; Shellcode cũng sử dụng stack, vì thế cần đảm bảo rằng khi stack phát triển, nó sẽ không ghi đè lên chính shellcode

### Tạo shellcode

Lưu ý:

Shellcode chỉ có code, không có dữ liệu. Nhưng shellcode cũng cần có dữ liệu => phải tìm cách nhúng dữ liệu vào shellcode

Các bước tạo shellcode:

-Viết mã bằng hợp ngữ

-Biên dịch

-Trích xuất shellcode

-Kiểm tra shellcode

-Tinh chỉnh shellcode

Vấn đề định địa chỉ:

-Trong shellcode thường sử dụng dữ liệu kiểu chuỗi

-Tham chiếu đến dữ liệu trong .data được thực hiện qua địa chỉ tuyệt đối

-Không thể thực hiện địa chỉ tuyệt đối trong shellcode (ngoại trừ địa chỉ của hàm thư viện lõi như libc)

Định vị dữ liệu qua địa chỉ tương đối:

-Đưa dữ liệu vào stack =>lợi dụng ESP sau lệnh PUSH

-Đưa dữ liệu vào .text =>lợi dụng địa chỉ trả về sau lệnh CALL

# Lỗ hổng

## Lỗ hổng Buffer Overflow (tràn bộ đệm)

### Khái niệm

Lỗi tràn bộ đệm là lỗ hổng trong lập trình, cho phép dữ liệu được ghi vào 1 bộ đệm có thể tràn ra ngoài bộ đệm đó, ghi đè lên dữ liệu khác và dẫn tới hoạt động bất thường của chương trình

Lỗi này dễ tránh, nhưng phổ biến và nguy hiểm

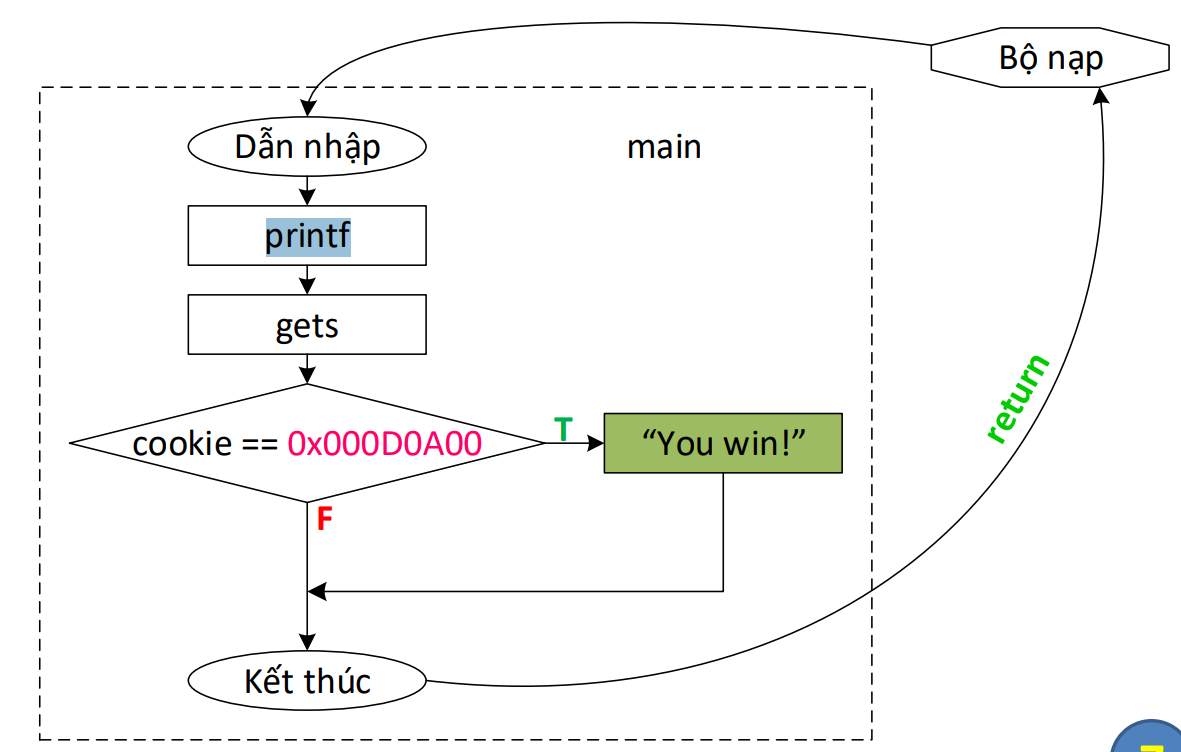
Lỗi này thường xuất hiện ở các dịch vụ web; chúng dduwoj viết bằng ngôn ngữ C, C++

Có 2 dạng tràn bộ đệm:

-Tràn bộ đệm trên Stack: biến cục bộ

-Tràn bộ đệm trên Heap: cấp phát động

### Luồng hoạt động



### Ghi đè biến cục bộ

### Ví dụ

|  |
| --- |
| /\* vuln1.c \*/  int main(int argc, char \*\*argv)  {  char buf[500];  if (argc>1) {  strcpy(buf, argv[1]);  printf("%s\n", buf);  }  } |

### Điều kiện để khai thác

Chúng ta chỉ cần ghi đề lên giá trị của “con trỏ lệnh bảo lưu” (saved instruction pointer) được lưu trên stack bằng địa chỉ mã lệnh mong muốn, đây là vị trí bắt đầu của shellcode. Như vậy, chúng ta cần phải xắp sếp shellcode ở đâu đó trên bộ nhớ stack và xác định địa chỉ bắt đầu của nó

### Cách khai thác

Vấn đề của tổ chức shellcode là làm sao xác định được vị trí bắt đầu của shellcode. Nhờ cách tổ chức shellcode với các NOP, địa chỉ này chỉ cần gần đúng sao cho rơi vào khoảng giữa các lệnh NOP trên bộ đệm shellcode

Các bước cơ bản cuar kỹ thuật tràn bộ đệm: Chuẩn bị bộ đệm làm tràn (shellcode); xác định địa chỉ trả về (RET); độ lệch do sắp biến, xác định địa chỉ của bộ đệm chứa shellcode, cuối cùng là gọi thực thi tràn bộ đệm

### Cách thức truyền dữ liệu vào chương trình

Nhập dữ liệu từ 1 tệp tin:

**-Chuyển hướng (direct)**

+Chuẩn bị file dữ liệu: data

+Gọi chương trình: app < data

|  |
| --- |
| #include <stdio.h>  int main(){  char st[ ]="aaaaaaaaaaaaaaaa\x04\x03\x02\x01";  FILE \*f = fopen("input.dat", "wb");  fwrite(st, 1, sizeof(st), f);  fclose(f);  return 0;  } |

-**Dùng đường ống (pipe line)**

+Viết ứng dụng xuất dữ liệu ra standard output: datagen

+Gọi chương trình: datagen | app

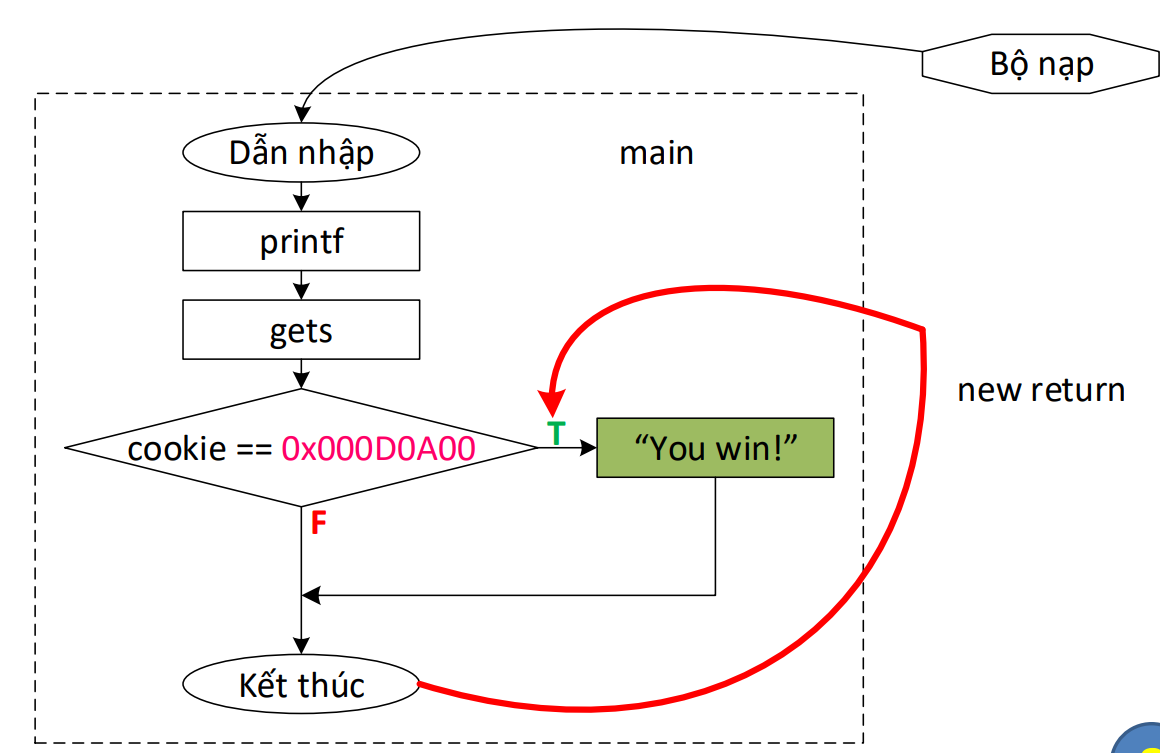
|  |
| --- |
| //File name: "redirect.c"  #include <stdio.h>  int main(){  char st[ ]="aaaaaaaaaaaaaaaa\x04\x03\x02\x01";  puts(st);  return 0;  } |

Nhập dữ liệu từ bàn phím:

|  |
| --- |
| #include <stdio.h>  int main(int argc, char \*argv[ ]){  printf("This progamm was run with %d parameter(s)\n"  ,  argc);  int i;  for(i=0; i<argc; i++)  printf("argv[%d] = %s\n", i, argv[i]);  return 0;  } |

### Ghi đè địa chỉ trả về

Luồng hoạt động sau khi thay đổi luồng thực thi: trở về than hàm



Sửa địa chỉ trả về:

-Xác định địa chỉ trả về mới

-Ghi đè lên vùng nhớ chứa địa chỉ trả về

+Tính khoảng cách từ buffer tới vùng nhớ chứa địa chỉ trả về

+Tạo dữ liệu thích hợp để ghi đè