اجرای کدهای نامطمئن:
Sandboxing

اجرای کد های نا مطمئن

- نیاز به اجرای برنامه های نا مطمئن و اشکال دار(buggy)
  - برنامه های غیر قابل اطمینان د سایت های اینترنتی
    - مانند toolbar ها و codec ها و viewer ها
  - برنامه قدیمی نا امن مانند outlook و ghostview
  - سرور های قدیمی مانند bind و sendmail
- هدف: اگر برنامه به هنگام اجرا رفتارناهنجار نشان دهد، kill شود

روش :ایجاد محدودیت

- محدودکردن: اطمینان از اینکه کد ها از رفتار مجاز خود خارج نشوند
- در سطوح زیر قابل پیاده سازی هستند:
  - سخت افزار: اجرای برنامه در فضای سخت افزاری ایزوله شده
    - مشکل مدیریت سخت افزار و پیچیدگی
  - ماشین های مجازی: ایزوله کردن سیستم عامل ها در یک سخت افزار مشترک
  - اجرای مجزای برنامه در داخل یک سیستم عامل به کمک فراخوان های سیستمی
  - جدا سازی پروسس هائی که در یک فضای آدرس دهی اجرا می شوند
    - مانند روش SFI
  - خاص برنامه ها: مانند اجرای مجزا در مرورگر ها

پیاده سازی محدودیت

- اجزا اصلی ایجاد محدودیت در زمان اجرا
  - پایش در خواست های کاربرد ها توسط یک سیستم واسط
    - پیاده سازی قوانین و سیاست های حفاظت در سیستم واسط
    - ایجاد محدودیت در اجرا
  - در اجرای تمامی برنامه ها باید پایش و ایجائ محدودیت انجام شود
  - برنامه پایش نباید kill شود. در صورت kill شدن برنامه پایش برنامه کاربردی نیز باید kill شود.

یک مثال ساده: chroot

- معمولا برای حساب های guest در سایت های FTP استفاده می شود
- برای اجرای آنب باید کاربر root بود

chroot /tmp/guest root dir "/" is now "/tmp/guest"

su guest EUID set to "guest"

- دایرکتوری /tmp/guest به فایل سیستم اضافه شده و برای تمامی برنامه های موجود در Jail(محدوده /tmp/guest) قابل دسترس است.
- کاربر این ناحیه به دیگر نقاط فایل سیستم دسترسی ندارد.

open("/etc/passwd", "r") ⇒

open("/tmp/guest/etc/passwd", "r")

Jailkit

- مشکل : تمام برنامه های مورد نیاز کاربر باید یک کپی در محدوده **jail** داشته باشد.
- پروژه **jailkit**: یک کپی از فایل ها و دایرکتوری های مورد نیاز را داخل **jail** ایجاد می کند
  - **jk_init**: متغیر های محیطی را ایجاد می کند
  - **jk_check**: محیط **jail** را از نظر امنیتی بررسی می کند
    - از نظر تغییر برنامه های محیط **jail** بررسی می کند
    - از لحاظ وجود دایرکتوری های ایجاد شده قابل نوشتن بررسی می کند
  - **jk_lsh**: **shell** محدود شده مورد استفاده در داخل **jail**
- در استفاده از **chroot** دسترسی شبکه محدود نمی شود

دور زدن Jail

- خروج از Jail با استفاده از مسیر دهی نسبی

open("../../etc/passwd", "r") ⇒

open("/tmp/guest/../../etc/passwd", "r")

- **Chroot فقط باید توسط root اجرا شود**
  - **در غیر اینصورت برنامه jail می توانند**
    - **فایل** "/aaa/etc/passwd" ایجاد کند
    - **د ستور** "/aaa" chroot را اجرا کند
    - با اجرای su root ، root شود

(bug in Ultrix 4.0)

دور زدن Jail

- ایجاد یک device که بتواند به دیسک خام دسترسی پیدا کند
- ارسال سیگنا ل های سیستمی به پروسس های غیر chroot شده
- Reboot کردن سیستم
- دسترسی به پورت های با دسترسی های بالا

Freebsd jail

- قدرت مند تر از chroot ساده

- برای اجراء:

jail jail-path hostname IP-addr cmd

- **نمی توان با مسیر دهی نسبی jail را دور زد**
- **فقط می توان به یک پورت اجازه داده شده و با IP مشخص bind شد.**
- **فقط می توان با پروسس های داخل Jail ارتباط بر قرار کرد**
- **root داخل jail محدود است و نمی تواند Module های kernel را صدا بزند**

مشکلات Jail و chroot

- سیاست های اعمال شده برای کل jail است
  - یا همه به فایل سیستم دسترسی دارند یا هیچکس ندارد
  - برای کاربرد هائی مانند مرورگر وب مناسب نمی باشد
    - مثلا برای attach کردن فایل در gmail نیاز به دستری خواندن فایل خارج از jail است.
- نمی تواند از دسترسی فایل های مخرب به شبکه و حمله به کامپیوتر های دیگر جلو گیری کرد
- می توانند باعث crash کردن سیستم عامل شوند

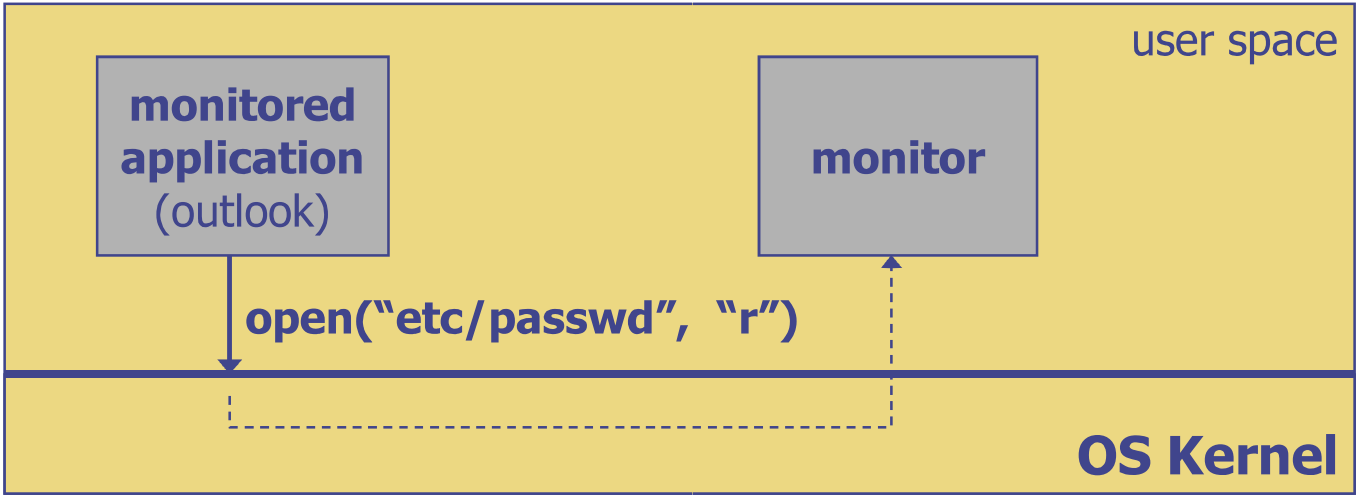
پایش اجرای فراخوانی های سیستمی

پایش اجرای فراخوانی های سیستمی

- برای ایجاد خرابی بر روی سیستم ها برنامه ها باید فراخواهای سیستم انجام دهند
  - برای پاک کردن یا نوشتن بر روی فایل ها توابع سیستمی unlike، open و write فراخوانی می شوند.
  - برای حمله از طریق شبکه توابع سیستمی socket، bind، connect و send فراخوانی می شود.
- ایده اصلی: پایش فراخوانی های توابع سیستمی توسط کاربرد ها و جلوگیری از فراخوانی های غیر مجاز
- پیاده سازی:
  - پیاده سازی در kernel سیستم عامل (GSWTK)
  - پیاده سازی در فضای کاربران (program shepherding)
  - پیاده سازی ترکیبی(Systrace)

پیاده سازی اولیه (Janus)

- ptrace در سیستم عامل linux: ردیابی پروسس ها
- تابع زیر پروسس را ردیابی کرده و درصورت فراخوانی تابع سیستمی بیدار می شود. **ptrace (... , pid_t pid , ...)**



- Ptrace درصورت در خواست غیر مجاز برنامه را kill می کند.

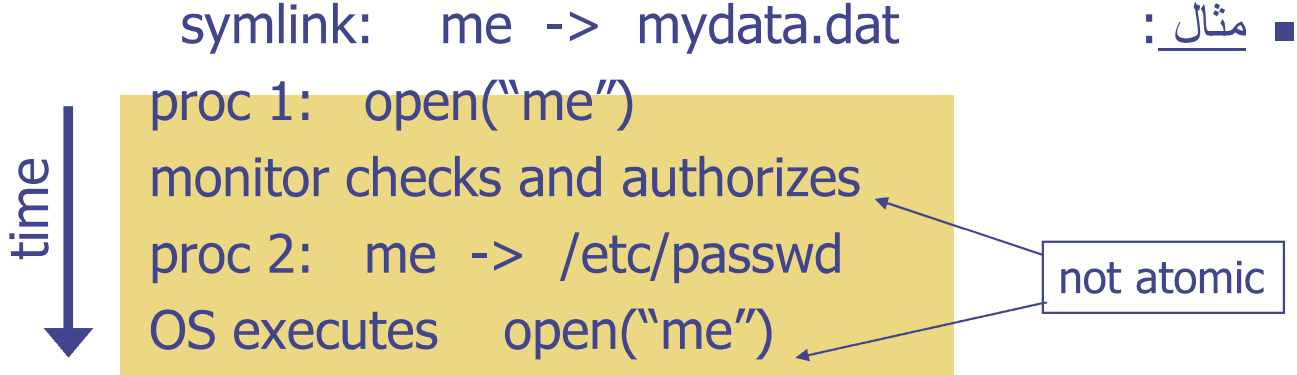
مشکلات

- اگر یک برنامه fork شود حتما برنامه پالایش نیز باید دو تا شود.
- اگر برنامه پالایش از بین برود در اینصورت برنامه اصلی هم باید kill شود.

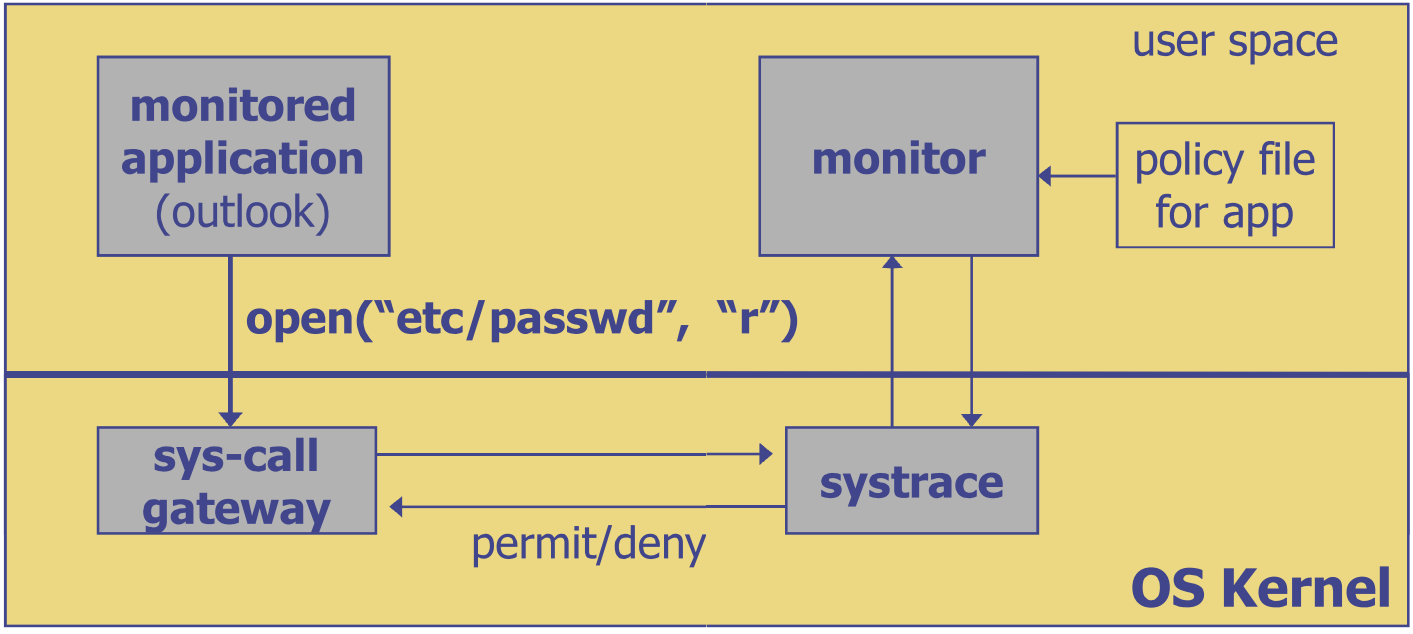
- برنامه پالایش لازم است تمام حالت های نگهداری شده در OS را باید نطارت کند.
- مثلا باید دایرکتوری جاری برنامه یا تغییرات دایرکتوری برنامه را باید بداند تا بتواند مثلا از تغییر نسبی دایرکتوری جلوگیری کند.

مشکلات ptrace

- Ptrace یا همه فراخوانی های سیستمی را ردیابی می کند یا هیچ کدام
  - به عنوان مثال نیازی به ردیابی تابع سیستمی close نیست.
  - برای abort کردن یک تابع سیستمی باید کل کاربرد kill شود.
- مشکل امنیتی در ptrace: **race conditions**
  - مثال:



systrace



- فقط فراخوانی توابعی را که قرار است پالایش شود به سیستم پالایش خبر می دهد
- برای حل مشکل race به جای symlink ها از آدرس دایرکتوری کامل استفاده می کند
- زمانی که برنامه قصد اجرا داشته باشد یک فایل سیاست گذاری بار می شود

فایل سیاست گذاری

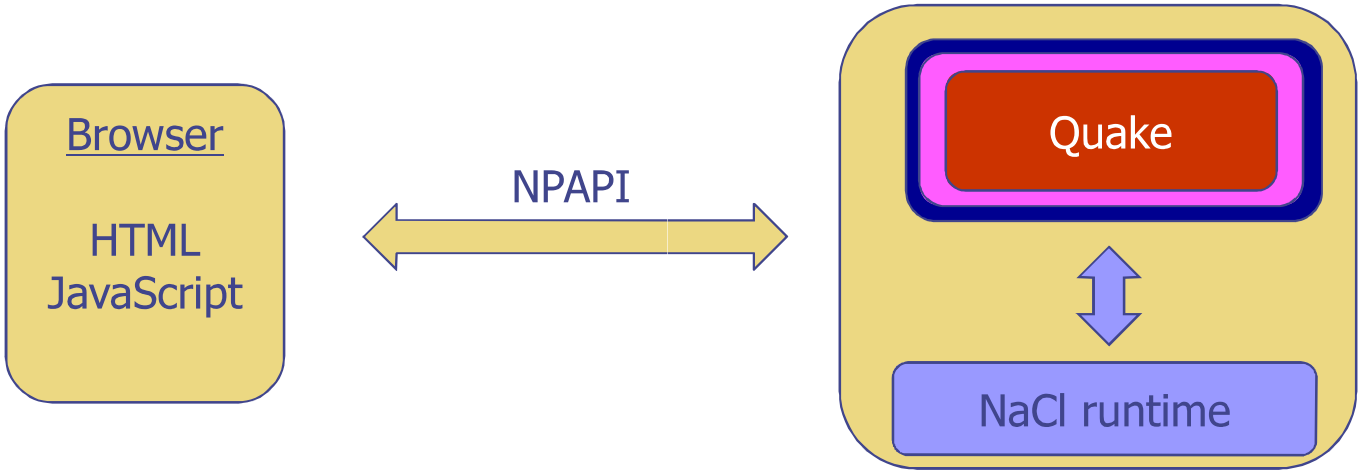
- یک نمونه فایل سیاستگذاری:

```
path allow  /tmp/*
path deny  /etc/passwd
network deny all
```

- تولید فایل سیایتگذاری کار مشکلی است.
  - Systrace با یادگیری رفتار خوب یک کاربرد می تواند فایل سیاستگذاری را تولید کند
  - اگر در فایل سیایتگذاری یک قانونی برای یک فراخوانی تابع سیستمی نبود از کاربر سئوال می شود ولی کاربر احتمالا نمی داند چکار کند

- مهمترین دلیل عدم استفاده از این روش سختی داشتن سیاست ها در مورد توابع سیستمی است

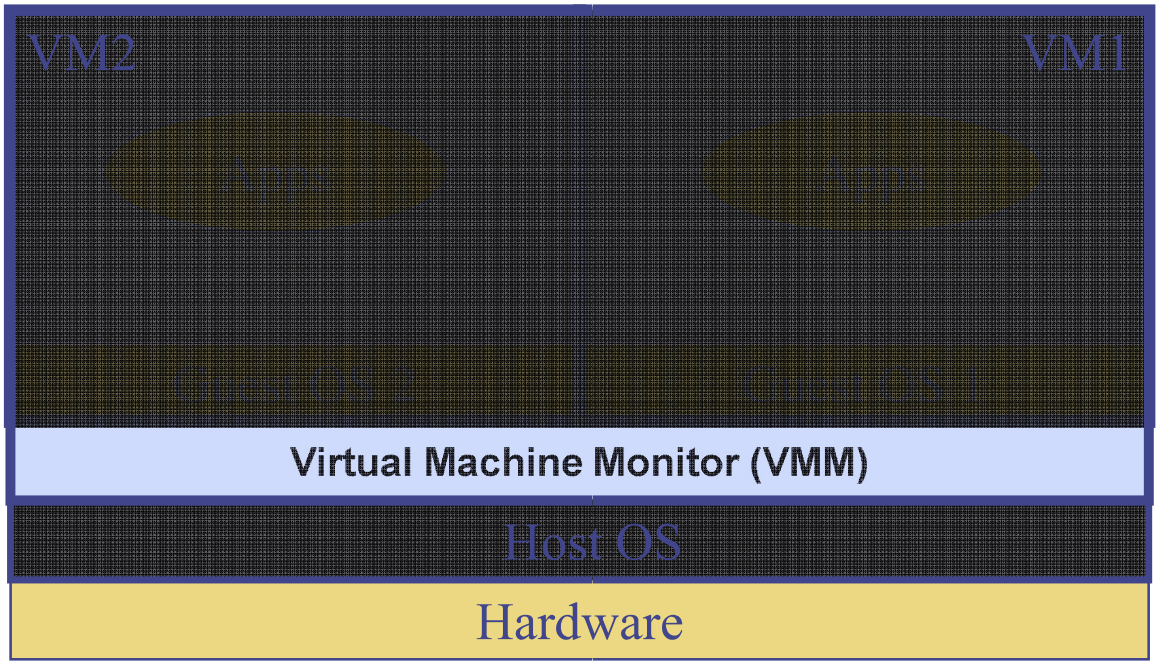
NaCl



- Quake: کد غیر قابل اعتماد بر روی x86
- دو عدد sandbox
  - Sandbox بیرونی: ایجاد محدودیت با استفاده از پایش فراخوانی توابع سیستمی
  - Sandbox درونی: با استفاده از بخش بندی حافظه توسط پردازنده X86 برنامه را از برنامه های دیگر ایزوله می کند.

محدود کردن با استفاده از ماشین مجازی
(Virtual Machine)

ماشین های مجازی



مثال: **NSA NetTop**

- یک سخت افزار برای چندین کاربرد

دلایل محبوبیت ماشین های مجازی

- اولین بار در سال 1960 مطرح شد
  - تعداد کم کامپیوتر، تعداد کاربر زیاد
  - ماشین مجازی امکان به اشتراک گذاری یک کامپیوتر بین چندین کاربر را فراهم می کند.
- بین سال های 1970 تا 2000 استفاده از ماشین مجازی کاهش یافت
- از سال 2000 به بعد:
  - تعداد سرویس ها نسبت به تعداد کاربران افزایش داشته است.
    - سرور وب، سرور ایمیل، سرور پرینتر، سرور پایگاه داده، سرور فایل
  - اجرای هر سرویس روی کامپیوتر جداگانه، هدر دادن منابع است.
    - استفاده از ماشین مجازی، علاوه بر صرفه جویی در سخت افزار، سرویس ها را ایزوله می کند.
  - امروزه ماشین های مجازی به طور گسترده در پردازش ابری (Cloud Computing) استفاده می شوند.

فرضیات امنیتی VMM

- فرضیات امنیتی
  - بدافزارها می توانند سیستم عامل مهمان (Guest) و برنامه های مهمان را آلوده کنند.
  - اما بدافزارها نمی توانند از محیط ماشین مجازی فراتر روند.
    - نمی توانند سیستم عامل میزبان را آلوده کنند.
    - نمی توانند ماشین های مجازی دیگری که روی همان سخت افزار نصب شده اند را آلوده کنند.
- اما خود VMM می تواند آسیب پذیر باشد.
  - VMM اصولاً ساده است و می تواند بصورت امن پیاده شود.
  - اما درایورها در سیستم عامل میزبان اجرا می شوند.

ایزوله کردن خطاهای نرم افزار
Software Fault Isolation

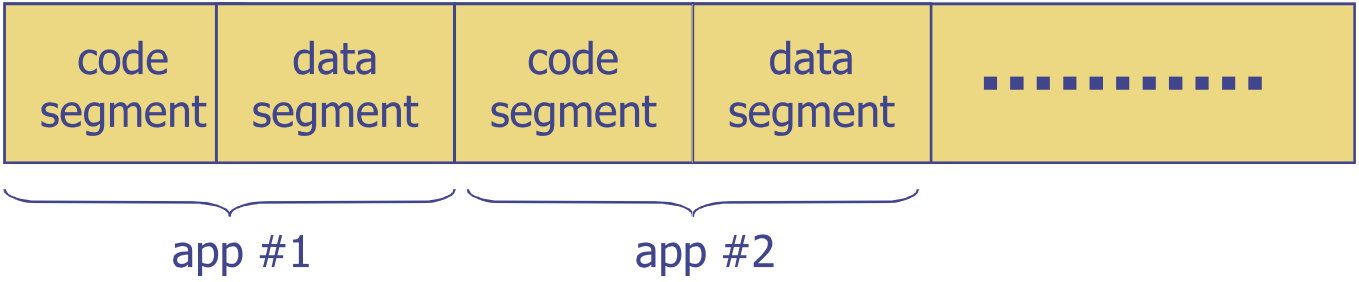
ایزوله کردن خطاهای نرم افزار

- هدف: تضمین کند تا برنامه های دارای یک فضای آدرس، دیگر برنامه های همان فضای آدرس را خراب نکنند.
  - مثال: دیوایس درایور نتواند کرنل را خراب کند.
- راه حل: برنامه ها را در فضاهای آدرس متفاوت اجرا کنیم.
  - مشکل: اگر برنامه ها رد و بدل داده زیادی داشته باشند، باعث کند شدن آن ها می شود.
    - به ازای هر پیام، یک Context Switch نیاز است.

ایزوله کردن خطاهای نرم افزار

ایده SFI

- حافظه را به بخش های مختلف تقسیم کنیم.



- دستورات ناامن مثل jmp، load و store را پیدا می کنیم:
  - هنگام کامپایل کد، حفاظ هایی قبل از دستورات ناامن می گذاریم.
  - هنگام بارکردن کد، از حضور تمام حفاظ ها مطمئن می شویم.

تکنیک segment matching

حفاظ اطمینان حاصل می کند که کد از سگمنت دیگری داده بارنکند.

```
dr1 ← addr
scratch-reg ← (dr1 >> 20)        : get segment ID
compare scratch-reg  and  dr2    : validate seg. ID
trap if not equal
R12 ← [addr]                     : do load
```

Address sandboxing تکنیک

- Dr2 حاوی شناسه سگمنت است.
- دستور **R12 ← [addr]** به دستورات زیر تبدیل می شود.

dr1 ← addr & segment-mask	: zero out seg bits
dr1 ← dr1 | dr2	: set valid seg ID
R12 ← [dr1]	: do load

- دستورات تولید شده کمتر نسبت به تکنیک segment matching

نتیجه گیری SFI

.

- کارایی
  - تا 4% باعث کندی می شود.
- مشکلات پیاده سازی در x86
  - طول دستورات متغیر بوده و مکان قرارگیری حفاظ ها نامشخص است.
  - تعداد پایین رجیسترها
  - تعداد زیادی دستور حافظه را تغییر می دهند و در نتیجه حفاظ های بیشتری نیاز است.

خلاصه

- تعداد زیادی تکنیک های sandboxing
  - Physical air gap,
  - Virtual air gap (VMMs),
  - System call interposition
  - ایزوله کردن خطای نرم افزار
- معمولاً ایزوله کردن کامل نامناسب است
  - برنامه ها باید با استفاده از واسط هایی با هم ارتباط برقرار کنند.