Deluppgift 19 Säkra systemanropen

Bakgrund

Som du vet så tilldelar operativsystemet en egen minnesrymd till varje användarprogram. Detta är för att ett program inte skall kunna mixtra och trixa med minne som tillhör andra program eller rent av tillhör operativsystemet. Det är operativsystemets uppgift att säkra systemet så inga möjligheter finns att kringgå minnesskyddet.

Ett möjligt sätt för användarprogram att komma runt minnesskyddet är att låta operativsystemet självt utföra de förbjudna åtgärderna (det har ju tillgång och tillåtelse till allt). Detta kan göras genom att utnyttja parametrarna till systemanrop. Alla systemanrop använder en pekare (minnesadress!) till användarprogrammets stack för att komma åt parametrarna till systemanropet. Genom att ange en falsk stackpekare innan systemanrop kan man alltså få operativsystemet att läsa förbjudna adresser.

Vidare så anger vissa parametrar till vissa systemanrop adressen där filnamn, kommando eller data startar, eller adressen där operativsystemet skall placera returnerad data. Om operativsystemet okritiskt läser eller skriver till någon av dessa pekare kan ett användarprogram alltså komma åt förbjudet minne genom att skicka in falska pekare som parametrar till systemanropen. Framförallt är det systemanropen `read` och `write` som är farliga, men även ett par andra.

Som exempel kan du tänka dig att vi vill ändra på uppgifterna lagrade på en godtycklig (förbjuden) adress X och framåt. Först öppnar vi en fil för att temporärt lagra dessa uppgifter. Därefter anropar vi systemanropet `write` för att skriva från godtycklig adress till filen. Nu kan de förbjudna uppgifterna läsas från filen som vanligt, ändras och skrivas tillbaka till filen. Därefter anropas systemanropet read för att läsa från filen till en godtycklig adress (i detta fall tillbaka till X). Pseudokod följer:

```
/* arrange temporary storage for the data */
char data[SIZE];
int fd = open(“mem.dump“);

/* save content of forbidden address X to file */
write(fd, X, SIZE);
/* read it to our memory space */
read(fd, data, SIZE);

/* do something with data */

/* save the data back to our file */
write(fd, data, SIZE);
/* read it to the forbidden memory address X */
read(fd, X, SIZE);
```

Varken detta eller något annat scenario skall vara möjligt när du säkrat din implementering.

Det gäller att noga utföra alla kontroller du kan komma på för varje parameter till ett systemanrop. Det får aldrig bli fel i operativsystemet för att ett program skickar konstiga data till systemanrop. Alla data måste kontrolleras så att operativsystemet kan använda dem utan att fel uppstår i operativsystemet.

Om användarprogrammet däremot råkar skicka en felaktig pekare till ett systemanrop som gör att operativsystemet skriver över viktig data i användarprogrammets *eget* minne så får användar-programmets programmerare skylla sig själv. *Operativsystemet skyddar bara sitt eget och andra processers minne.*

Saker som är bra att kontrollera är stackpekaren (att inga data läses utanför stacken), systemanropsnummer, alla parametrar som är pekare, pekare till strängar (alla adresser inom strängen), pekare till buffer (alla adresser i buffern), fildeskriptorer (fd), processid (pid), och filstorlek.

Saker som kan vara bra att känna till

Pintos använder *paging* i två nivåer. Varje tråd (se `threads/thread.h`) i kernel håller reda på en så kallad *pagedir*, som innehåller pekare till *pagetables*. Pagedir är alltså en pagetable för en större sidindelad pagetable. Filerna `threads/vaddr.h` och `userprog/pagedir.h` innehåller deklarationer för att hantera pages och pagetable. Observera att en manuell uppslagning i pagetable med `pagedir_get_page` är långsam jämfört med processorns automatiska uppslagning. Antalet uppslagningar skall alltså minimeras. Lyckligtvis räcker det med att slå upp *en* adress inom en sida för att kontrollera om *alla* adresser inom sidan är giltiga. Är sidan markerad som ogiltig i pagetable skulle ju uppslagningen misslyckas på varje adress inom den sidan.

Den logiska minnesrymd som via paging tilldelas en tråd (process) i Pintos är (som processen ser det) sekventiell och uppdelad i två delar. Användarminnet finns från adress 0 upp till adressen `PHYS_BASE (0xc0000000)`. Resterande minne, inklusive `PHYS_BASE`, tillhör kernel. Processen få endast använda användarminnet. Observera speciellt att processens stack är placerad så den växer från `PHYS_BASE` och nedåt mot lägre logisk adress, samt att processens kod är placerad med start på adress `0x08048000`. Det ger ett spann av en storlek om ungefär 3GB, där alla adresser däremellan knappast är knutna till fysiska adresser. Du måste alltså förutsätta att det mellan två giltiga adresser kan finnas sidor med adresser som är ogiltiga.

Reglerna för om en viss adress går att läsa på ett säkert sätt är som följer:

- Adress 0 (`NULL`) är aldrig giltig.
- Data som återfinns på en (logisk) adress större än eller lika med PHYS_BASE får inte läsas eller skrivas av processen, det är reserverat för kernel. Eftersom vi fortfarande pratar om logiska adresser i processens logiska (sekventiella) minnesrymd räcker det naturligtvis att testa den högsta adressen av dem som skall användas. Är den åtkomlig är naturligtvis även alla lägre adresser åtkomliga.
- Data som är lagrat på en logisk adress som återfinns inom en ogiltig sida i pagetable är förbjuden att läsa då det skulle ge upphov till pagefault. Om `pagedir_get_page` returnerar `NULL` är samtliga adresser inom samma sida ogiltiga. Annars är samtliga adresser inom samma sida giltiga. Huruvida en logisk adress högre eller lägre än den testade är giltig går inte att avgöra, om den inte infaller inom samma page som den testade. Kontroll av denna punkt givet en sekvens adresser har du löst i lab 5.

Skapa testfall i Pintos (endast för referens)

Kopiera och modifiera ett befintligt testfall (*.c och *.ck) i katalogen `tests/userprog` eller `tests/filesys/base` (beroende på test). Lägg sedan till det i Make.tests genom att ange värden för `_TESTS`, `_SRC`, och `_PUTFILES`.

Uppgift

Verifiera att all indata till varje systemanrop är korrekt och inte kan orsaka säkerhetsbrister eller krasher i kernel. Om någon parameter är ogiltig eller konstig skall systemanropet misslyckas. Om det går att misslyckas genom att returnera en felkod (oftast `-1`) från systemanropen skall detta göras. annars dödas det användarprogram som gjorde det felaktiga systemanropet. Vid allvarliga fel, såsom att en ogiltig pekare gavs som parameter, skall processen alltid dödas. En process som dödades av kernel skall alltid avsluta med `-1` som `exit_status`.

Speciellt uppmärksam måste du vara på alla instruktioner som använder användarprogrammets minne (läser eller skriver). Flera pekare dit existerar, se avsnitten ovan. Du kan återanvända kod från uppgift 5, men du måste lägga till några kontroller, och du måste skicka in korrekt `pagedir` till `pagedir_get_page`.

Testa din implementering

I pintos finns en uppsättning “inbyggda” test för att verifiera att implementeringen är någotsånär korrekt. Hur du använder dessa beskrevs i uppgift 18. Efter denna laboration skall alla dessa tester fungera.

Observera att testerna måste köras MÅNGA gånger för att vara någotsånär säker på att allt fungerar. Synkroniseringsfel uppstår inte vid varje körning, utan någon gång ibland, och på olika ställen. Prova följande script för att sluttesta. Kör ssh till `astmatix.ida.liu.se` och kör sluttestningen på den servern. Hur många vändor klarar din implementation?

```
pintos-check-forever -a
```