LINKÖPINGS UNIVERSITET

TDIU16: Process- och operativsystemprogrammering

Lab 5: Säkra systemanrop

Filip Strömbäck, Klas Arvidsson, Daniel Thorén

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2023

2023-03-15

Mål

Alla program i Pintos (och i andra operativsystem) körs i en nedlåst miljö som ofta kallas för usermode. Operativsystemet ser till så att processer i usermode inte får göra mycket mer än att läsa från och skriva till det minne som är reserverat för processen. Däremot kan den med hjälp av systemanrop be operativsystemet att göra saker som processen normalt inte får göra. Eftersom operativsystemet inte körs i usermode så får det göra vad som helst med systemet. Därför är det viktigt att operativsystemet är mycket noga met att kontrollera så att programmet i usermode inte försöker göra något konstigt.

Den här labben fokuserar på fallet då ett program i usermode ber operativsystemet att läsa från eller skriva till minne (exempelvis i systemanropen `read` eller `write`, kom ihåg problem 4 från challenge 4 i TDIU11). Från operativsystemets synvinkel är då två saker viktiga att kontrollera:

1. Att den virtuella adress vi försöker komma åt faktiskt finns tillgänglig i RAM. Annars kommer ett pagefault genereras, och hela operativsystemet kraschar.
2. Att den process som har anropat systemanropet skulle få komma åt adressen i normala fall. Annars skulle det exempelvis vara möjligt att läsa känslig information som bara operativsystemet har tillgång till i normala fall, eller skriva över viktiga datastrukturer som gör att operativsystemet kraschar eller på annat sätt slutar fungera.

Målet är alltså att se till att ett program i usermode inte kan få kernel att krascha eller bete sig felaktigt genom att skicka ”felaktiga” argument till systemanropen. I stället ska Pintos detektera dessa felaktiga argument och returnera ett felmeddelande från systemanropet, eller stänga av processen beroende på vad som är lämpligt.

Mer information om hur detta fungerar finns i Pintos-Wiki under rubriken ”Minne”, se särskilt underrubrikerna ”Accesskontroll” och ”Paging i Pintos”.

Del A Accesskontroll

Först behöver vi logik för att kontrollera om en adress är giltig eller inte. Vi börjar med att implementera detta utanför Pintos för att förenkla testning.

I mappen `src/standalone/lab5a` finns filen `verify_adr.c`. Den innehåller två funktioner som ska kontrollera olika typer av argument: `verify_fix_length` och `verify_variable_length`. Filen innehåller också en uppsättning tester (arrayen `test_case` och `main`).

Målet är alltså att implementera funktionerna `verify_fix_length` och `verify_variable_length`. De ska fungera enligt nedan:

- `bool verify_fix_length (void *start, unsigned length);`

  Ska kontrollera att alla adresser från och med `start` till `start + length` (men inte inklusive) är giltiga och går att läsa från usermode.

- `bool verify_variable_length (char *start);`

  Ska kontrollera att alla tecken (bytes) från och med `start` och till och med det första 0-tecknet är giltiga och går att läsa från usermode. Funktionen kontrollerar alltså att det går att läsa en C-sträng som startar i `start`.

För att kontrollera ifall en adress är giltig måste vi inspektera de page-tables som används av processen. Till vår hjälp finns ett antal funktioner och konstanter i headerfilen `pagedir.h`:

- `PGSIZE`

  Storleken av en page. I testprogrammet är page-storleken 100 bytes för att göra felsökning enklare (då är hundratalssiffran pagenummer och resten är offset). I Pintos är PGSIZE 4096 (vilket är `0x100` hexadecimalt) eftersom x86 har en page-storlek på 4 KiB.

  På grund av detta är det viktigt att du använder hjälpfunktionerna i `pagedir.h` för att manipulera adresser. Annars kommer din kod inte att fungera i Pintos senare.

- `void *pg_round_down (const void *addr);`

  Avrundar adressen `addr` nedåt, till början av den page som `addr` refererar till.

- `unsigned pg_no (const void *addr);`

  Beräknar vilket page-nummer som `addr` refererar till. Första sidan är 0 som allt annat i C.

- `void *pagedir_get_page (void *pd, const void *addr);`

  Använder page-tabellen `pd` för att slå upp vilken fysisk address som den virtuella adressen `addr` refererar till. Om uppslagningen misslyckas så returneras `NULL`. Översättningstabellen finns i `struct thread` och heter `pagedir`.

  Notera att denna funktion är relativt dyr. Du vill alltså se till att inte anropa den fler gånger än vad du faktiskt behöver! I testprogrammet tar varje anrop till `pagedir_get_page` några hundra millisekunder, så om testarna tar lång tid att köra anropar du antagligen funktionen för många gånger!

- `bool is_end_of_string (char *addr);`

  Returnerar `true` om `addr` är ett 0-tecken, och därmed indikerar slutet av en sträng. Denna funktion finns endast i testprogrammet, och behövs eftersom att testprogrammet bara simulerar adresser. Det går därför inte att läsa direkt från pekaren.

  I Pintos måste du ersätta anropet med att kontrollera om tecknet som `addr` pekar på är `'\0'`.

Del B Säkra systemanropen

När testprogrammet i del A fungerar som det ska är det dags att flytta implementationen av `verify_fix_length` och `verify_variable_length` till Pintos (de kan exempelvis ligga i `src/userprog/syscall.c`). Vi vill nu använda dessa funktioner för att verifiera att all data som kommer från usermode är korrekt, så att kernel inte kraschar eller beter sig felaktigt.

Du kommer ofta behöva anropa någon av `verify`-funktionerna, och avsluta processen ifall den returnerade `false`. Det kan därför vara en bra idé att skapa hjälpfunktioner för detta för att förenkla implementationen.

B.1 Vad ska kontrolleras?

Grundidén är att kernel inte kan lita på data som kommer från usermode. Detta innebär att om vi tar emot en pekare från usermode (exempelvis `buffer`-parametern från `read`), så måste vi försäkra oss om att den innehåller en virtuell adress som är giltigt (dvs. den finns i page-table för processen) och att den inte refererar till minne som bara kernel borde kunna komma åt.

En bra idé är att rita upp hur minnet ser ut när `syscall_handler` anropas, och markera de delar av minnet som ni har kontrollerat att de är säkra. Målet är då att bara läsa från minne som ni tidigare har kontrollerat att det är säkert (dvs. ni vet att ni inte kommer att krascha eller komma åt minne som hör till kernel).

B.2 Testprogram

Vid detta tillfälle ska *alla* automatiska tester passera. Se ”Automatiska tester” i Pintos-Wiki för information om hur de körs.

När du arbetar med systemanropen som testar valideringen av parametrar till systemanropen är det viktigt att förstå vad testerna gör (läs källkoden i `src/tests/...`), så att du förstår *varför* testet misslyckas. Att testa sig fram genom att försöka verifiera olika saker till testet går igenom är sällan en bra idé, och gör ofta att något annat test misslyckas i stället.

Några av testerna kan hitta buggar i tidigare uppgifter, så du kan behöva gå tillbaka och fundera på tidigare laborationer också.

Det är också värt att notera att testerna inte är heltäckande. Att alla tester går igenom betyder inte nödvändigtvis att ni har verifierat allt som behöver verifieras (men testerna är en bra approximation). Fundera därför igenom vad som behöver kontrolleras, och dubbelkolla att ni inte har missat något i er implementation!

När alla tester verkar fungera är det dags att försöka framkalla synkroniseringsfel. I och med att synkroniseringsfel inte visar sig vid varje körning, utan bara ibland, så måste testerna köras många gånger. Detta kan göras med kommandot:

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

I och med att trådproblem inte alltid visar sig så finns det egentligen aldrig någon garanti att implementationen är korrekt, oavsett hur många körningar som implementationen klarar. Däremot ökar sannolikheten att den är korrekt ju fler körningar du har gjort. Ett rimligt riktmärke kan vara 30–40 iterationer för någon form av konfidens.