2023

TCP Flag Attack Detection – Bachelor Opgave - KEA

CHRISTOFFER BRYDENSHOLT – ITS22 – 28/07/1996 / CHRI23CM

Indhold

Indledning og valg af projekt

Muninn er et IT-Sikkerhedsvirksomhed der beskæftiger sig med at udvikle løsninger til kunder, der kan hjælpe dem med at holde deres interne IT-miljø sikkert på flere forskellige måder. Deres hovedprodukt er en NDR (network detection and response) sensor, der alarmerer og beskytter imod angreb. Firmaet blev grundlagt i 2016, og har i dag knapt 30 medarbejdere. Trods at de er en mindre virksomhed, har de store kunder i mange af landets største brancher, som de hjælper med at beskytte imod trusler på deres netværk. Jeg har i 10 uger haft mulighed for at arbejde sammen med dem, og hjælpe dem med at udvikle deres løsninger samt at give dem et overblik over deres nuværende produkter, og komme med min egen ekspertise hvad angår deres nuværende og fremtidige idéer og udfordringer.

Gennem mit ophold hos Muninn fik jeg indblik i, og ansvar for at være med til at vælge hvilke nye teknologier der skulle udvikles i fremtiden. Der er i IT-Sikkerhedsverdenen konstant nye huller der skal dækkes, i den forstand at der hele tiden udvikles nye angrebsmetoder som man skal kunne beskytte sig imod.

Et emne der gik igen og igen var dog et større projekt, som lød på at Muninn enormt gerne vil have en måde at observere og analysere ”ulovlige” TCP flag kombinationer i de forskellige data streams på. Dette ville ikke bare dække ét problem, men det vil kunne bruges til at dække rigtig mange angrebsoverflader som TCP flags kan bruges til, og derudover vil det kunne opfange noget så banalt som defekte devices der fylder og sender støj ud på netværket.

Dette produkt er lignende CISCO’s ”Stealthwatch” produkt, som blandt andet analyserer traffik for mistænkelige TCP flags og kombinationer[1].

Udover selve produktudviklingen er omfattende forskning også nødvendig for at indhente tilstrækkelig information til at designe det på en effektiv måde. Jeg skal i første omgang forstå de forskellige TCP flag, deres tiltænkte anvendelsesmåde, hvorfor de er vigtige for TCP protokolen, og i sidste ende hvordan de kan misbruges som et angreb. Herefter skal jeg kigge

---

1 https://www.cisco.com/c/dam/en/us/td/docs/security/stealthwatch/management_console/securit_events_alarm_categories/SW_7_2_1_Security_Events_and_Alarm_Categories_DV_1_0.pdf

på kombinationer af TCP flagene, hyppigheden af angreb baseret på specifikke flag og kombinationer, og til sidste udvikle et produkt der kan analysere traffik og gøre opmærksom på potentielle angreb, baseret på min research.

Problemstilling:

TCP protokollen benyttes i høj grad til en bredde af forskellige angreb i disse dage. Alle virksomheder er ofre for netværks scans, port scans, MiTM angreb mm. Som hackere udfører ved hjælp af forskellige TCP flag angreb, og som ofte aldrig bliver opdaget i tide.

Problemformulering:

Det er mit mål med dette projekt at forske og udvikle et produkt der kan hjælpe virksomheder med at opdage disse angreb, og det vil jeg udarbejde baseret på ét hovedspørgsmål og to underspørgsmål.

- **Hvordan kan man bruge deep packet inspection til at sikre netværk mod TCP flag angreb?**
    - Hvilke værktøjer kan bruges til dette, og hvordan?
    - Hvordan adskiller disse værktøjer fra hinanden, og hvornår skal de hver især bruges?

Fremgangsmåde:

For at besvare denne problemformulering vil jeg følge denne fremgangsmåde:

- Jeg vil først og fremmest researche TPC protokolens header flags, og blive fuldstændig klar over hvad de hver især har af funktion.
- Derefter vil jeg undersøge de forskellige angreb der hyppigst bruges i forhold til TCP flags, og forstå hvordan og hvorfor de virker på den måde gennem TCP protokollen.
- Jeg vil undersøge hvorfor det er vigtigt, i form af System Sikkerhed, at vi opdager disse typer angreb.

- Herefter vil jeg undersøge hvilke værktøjer der ville være relevante at tage i betragtning til at udvikle mit produkt
- Til sidst vil jeg sammenligne værktøjer, og udvikle mit produkt baseret på min sammenligning og research.
- Dette produkt vil jeg give en demo på, samt bevise brugbarheden ved at teste den op imod simulerede angreb.

Teori:

TCP Flag, en forklaring:

For at komme i gang med opgaven om at forstå hvordan TCP flag kan bruges på ondsindet vis, skal vi i forstå omgang have en konkret forståelse for hvad flag er, hvilke der eksisterer og hvad de bruges til. Dette vil jeg gøre på kortfattet vis, da det er basal viden.

Hvad er TCP flag?

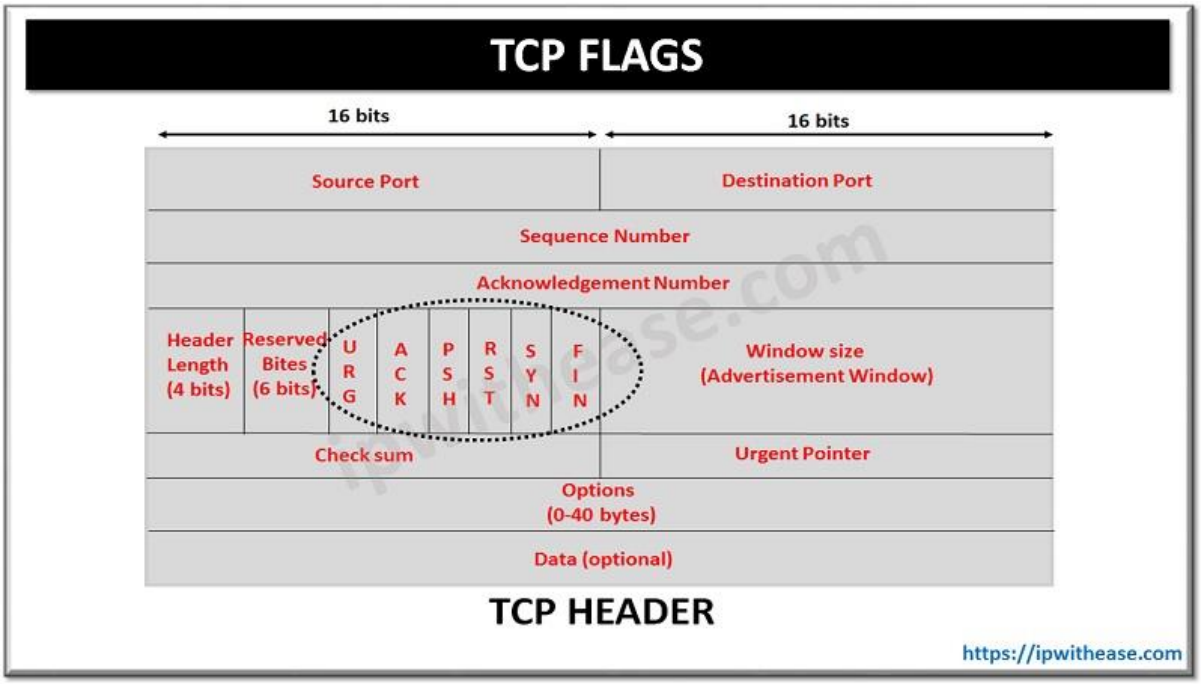
TCP flag kan ses som et hjælpemiddel til TCP protokollen, for at pakkerne kan specificere afsenderes intention, ved at sætte forskellige typer flag, der nemt kan forståes af modtageren. De kan ses som indstillinger der kan sættes i et TCP segment, som kan forståes og bruges i en større helhed for TCP forbindelsen (eller forsøget på et oprette en).

De 6 standard flag i TCP protokollen er:

1. URG (Urgent Pointer)
2. ACK (Acknowledgement)

3. PUSH
4. RST (Reset)
5. SYN (Synchronization)
6. FIN (Finished)

Derudover findes ECH og CWR flagene.



For at forstå hvordan disse forskellige flag bruges, og hvordan de kan bruges på ondsindet vis, vil jeg gennemgå deres tekniske grundlag, hvad de normalvis bruges til og hvornår vi ser dem blive brugt. Dette vil give os den fundamentale forståelse for hvordan flags på teknisk plan kan udnyttes til en angribers fordel, sammen med vores forståelse indenfor TCP protokolen og vores viden indenfor netværk.

URG (Urgent Pointer):

URG flaget bruges til at informere den modtagene del af TCP forbindelsen om at vis data er akut. Urgent pointer specificerer hvilken del af dataen i segmentet der er ”akut”, og som

derfor sendes videre til den modtagne applikation først. Det skal dog pointeres at det er applikationen selv der håndterer den akutte data, og at det ikke er TCP's ansvar.

Med de hastigheder vores netværk undestøtter disse dage, kan det virke lidt overflødigt hvorvidt data sendes først eller sidst, når det allerede er ankommet til modtageren i samme segment. Pointen er dog stadig at det stadig understøttes af TCP, og derved fortsat er en funktion der kan misbruges på trods af at det knapt nok benyttes længere, på samme måde som f.eks. telnet.

| Data |R|S|F| **U** |A|P|RSF| Window Size (e.g., 4096) |

| Offset|S|Y| | **R** |C|S|SYN| |

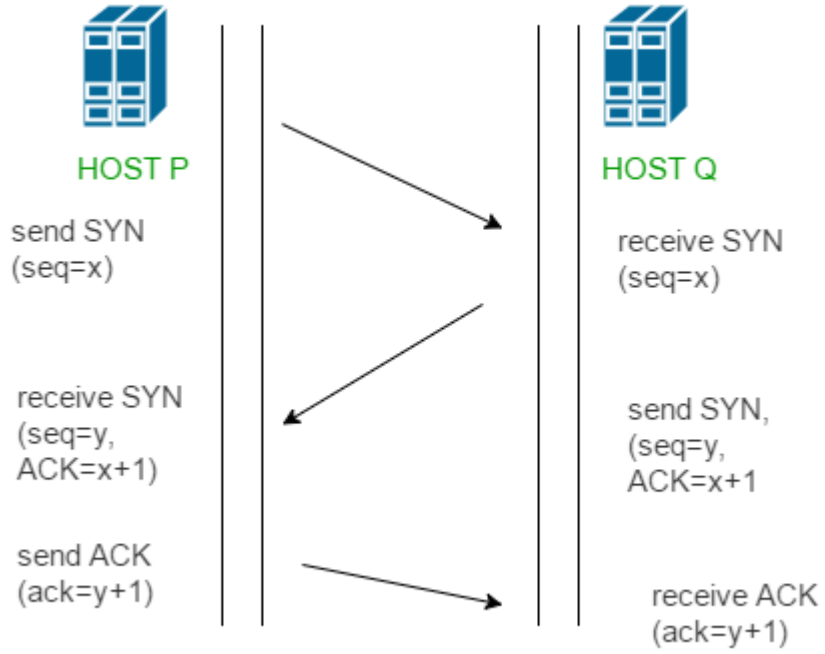
+-+

| Checksum (e.g., 0xABCD) | Urgent Pointer (e.g., 150)|

Dette eksempel specificerer altså at "urgent data" begynder 150 bytes efter starten af segmentets data segment.

SYN (Synchronization):

SYN flaget bruges til at oprette en forbindelse mellem to parter. For at oprette en forbindelse foretager TCP et "3-way-handshake", som betyder at begge parter er villige til at oprette en forbindelse, og som er påkrævet før at TCP data kan sendes frem og tilbage mellem dem.



[2]

Den initierende vært sender en SYN-pakke til vedkommende den ønsker at oprette forbindelse med. Hvis modtageren ønsker at acceptere, sender vedkommende en SYN,ACK pakke tilbage (acknowledgement af forbindelse), hvorved den initierende part sender en ACK tilbage (acknowledgement af at vedkommende har accepteret).

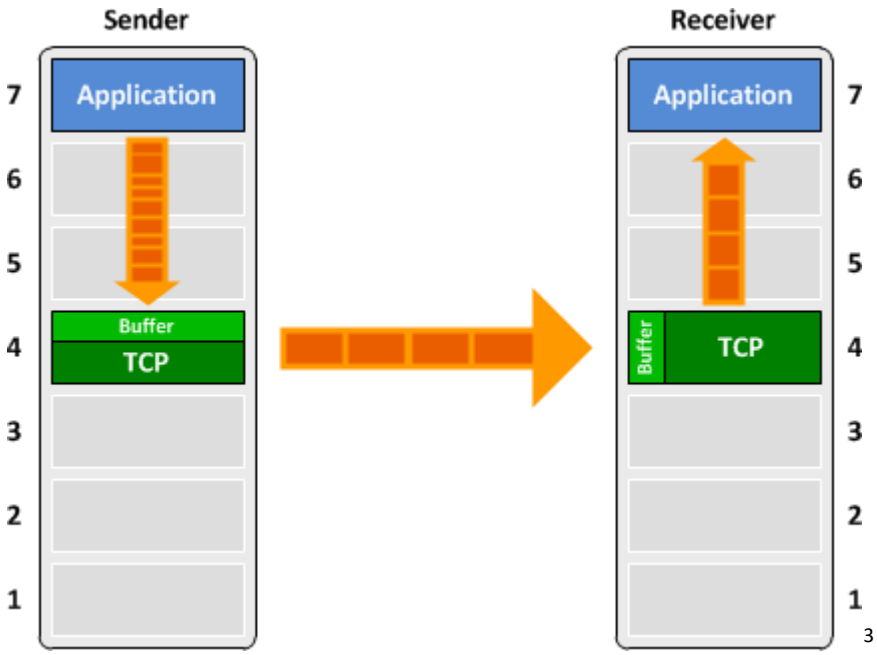
ACK (Acknowledgement):

ACK flaget bruges altså til anerkendelse af oprettelse af en forbindelse, men også som en form for kvittering på at data er modtaget. Eftersom at dette flag er velkendt, vil jeg ikke gå i nærmere detaljer.

PSH (PUSH):

I TCP protokollen bruges der en buffer. Denne buffer implementeres på begge sider af TCP forbindelsen.

[2] https://media.geeksforgeeks.org/wp-content/uploads/TCP-connection-1.png



[3]

En sådan buffer gør en forbindelse mere effektiv i tilfælde af afsendelse af data med større størrelse end det afsendte segments maksimum. Idéen her er altså at bufferen konstant bliver ”fyldt op” og er klar med det næste segments data med det samme, så der ikke er noget ventetid fra applikation til TCP segmentet.

Problem er dog i ”real time applications”, hvor der konstant sendes mindre segmenter frem og tilbage, at vi i dette tilfælde selvfølgelig ikke har noget behov for at fylde segmentet op. Disse former for applikationer har behov for konstant at sende segmenter, og så hurtigt som muligt, her er bufferen altså overflødig.

Det er her at PSH flaget er nødvendigt.

Applikationer der bruger TCP kan vælge at fortælle den socket de gør brug af, at PSH flaget skal bruges. Dette gør altså at alt data denne socket modtager kan blive videresendt til TCP forbindelsen med det samme, og den den modtagne forbindelse ved, eftersom at PSH er sat til 1, at den skal videresende dataen til den modtagende applikation med det samme.

Kort sagt bruges PSH til at fremskynde data udveksling mellem to parter, da visse applikationer har et behov for dette for at fungere optimalt.

[3] https://packetlife.net/blog/2011/mar/2/tcp-flags-psh-and-urg/

RST (Reset):

RST flaget har flere forskellige brugbarheder, men bruges oftest til enten at resette en fejlagtig forbindelse, eller til at afslå forbindelser, f.eks. fra en bruger der prøver at tilgå en lukket port. Lokale firewalls kan også, baseret på deres konfiguration, sende RST pakker tilbage på modtagne pakker der overtræder dets regler.

FIN(Finished):

FIN bruges som en mere effektiv måde at lukke inaktive forbindelser. Nogle applikationer lader en TCP forbindelse ”hænge” på trods af at den ikke bruges længere, og den lukkes derfor først når en vilkårlig host ikke længere er online. FIN kan derimod bruges til at afslutte en forbindelse på en ordentlig måde. Dette sker ved at en host sender en TCP fin besked, hvorved modtageren sender en FIN ACK pakke tilbage, og modsat.

Flag attacks

For at løse min problemstilling er jeg nødt til at tage udgangspunkt i en vis mængde angreb. Angreb baseret på TCP flags findes der mange af, og de kan bruges til mange forskellige ting. Jeg kan derfor ikke dække dem alle. Jeg vil derfor definere nogle hyppigt brugte angreb i dette afsnit, og hovedsageligt teste mit produkt op imod disse angreb i led med udviklingen af det. Disse typer angreb jeg vil dække er derudover også defineret ud fra Muninns behov, der i største omfang består af hvad CISCO’s Stealthwatch dækker.

Først vil jeg kigge på flood attacks. I tilfælde af disse angreb er vi interesseret i stateful tracking, altså hvor vi holder øje med en vis forbindelse, og hvor mange specifikke typer pakker der bliver sendt gennem denne forbindelse.

Flooding angreb:

Syn Flood Attack:

Syn flood angrebet er et af de bedst kendte angreb når det kommer til at misbruge TCP flags.

Tanken bag dette angreb er samme koncept som andre DOS/DDOS angreb, altså at overloade en givent netværk i den grad at det ikke længere kan bruges, eller i det mindste gøres ubrugeligt langsomt. Hvorvidet dette er succesfuldt afhænger både af hvor mange resourcer angriberen har til rådighed, i form af botnets mm. Og hvor stærkt et netværk ofret har, samt deres beskyttende teknologier.

Billedet ovenfor er et eksempel på hvordan et simpelt SYN flood script kunne se ud. Her spoofes vores egen IP, og spammer en række IP adresser med SYN pakker på port 80.

Hvad kigger vi efter her?

Usædvanligt mange SYN flag

SYN sendt men SYN-ACK ikke sendt.

ACK SCAN:

ACK scan er kendt som en teknik der bruges til at definere firewall regler for en maskine.

ACK er ikke et flood attack, men eftersom ACK pakker ikke kan identificeres som ondsindede kun på baggrund af et ACK flaget, dækker det også ind under stateful tracking.

Per RFC 793[4] skal en modtager sende en RST besked tilbage til afsenderen, hvis vedkommende sender en ACK besked til en port, omend den er åben eller lukket. Logisk set

[4] https://www.rfc-editor.org/rfc/rfc793.txt

kan man så regne ud, at hvis dette ikke sker, betyder det at pakken aldrig når så langt, og derfor er blevet blokkeret af en firewall eller et intrusion prevention system. En del af problemet for ofret ved dette angreb er, at det er nemt at filtrere forbindelser fra på SYN pakker, fordi det viser hvem der starter forbindelse. SYN pakken kommer altid fra den initierende part, og kan derfor nemt beskyttes mod. Vi ved derimod ikke, baseret på ACK flaget, hvem der har initieret forbindelsen og om den er legitim, og derfor kræver det en stateful firewall.

Hvad kigger vi efter her?

Usædvanligt mange ACK flag uden en egentlig funktion i forbindelsen.

Flag kombinationer:

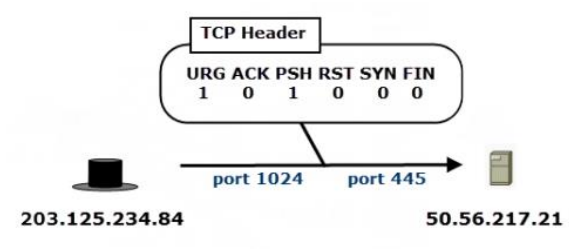
Herefter bevæger vi os ind i TCP flooding og scanning angreb, som primært består af en række TCP Flag kombinationer som vi kan definere som ondsindede. Dette gør det nemmere at opdage, eftersom at der ikke skal kigges på kontekst i en forbindelse, vi skal i stedet bare kigge på ”ulovlige” flag kombinationer.

De flag kombinationer vi er interesserede i at opdage er:

Bad flag ACK:

Denne type packet kendes ved at ACK ikke er sat, men URG, FIN, eller PUSH er.

Denne type packet, ligesom mange andre jeg kommer ind på, består af en ”ulovlig” kombination af flags, altså en kombination der ved normal brug af TCP protokollen aldrig burde opstå. Disse typer angreb kan bruges til fingerprinting, port scanning, vulnerability detection mm.

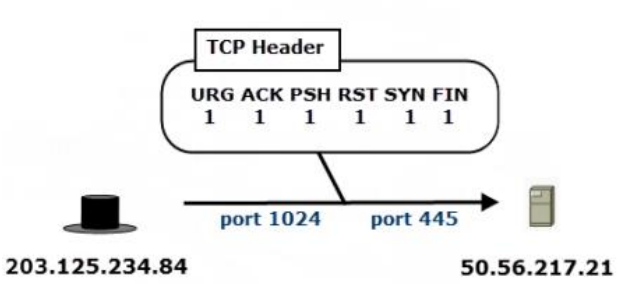


[5]

Bad flag All:

Denne type packet kendes ved at alle flags er sat.

Denne kombination er selvfølgelig ikke ”lovlig”, og er næsten altid sendt på denne måde med vilje. Intentionen er at se hvordan en sådan kombination vil interagere med modtagerens maskine, og prøve at fingerprinte deres system, baseret på det svar der returneres.



Bad flag no flag:

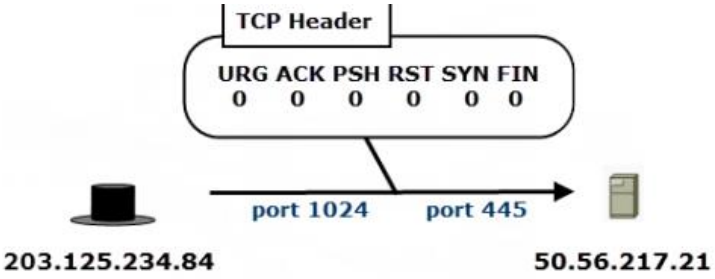
Denne type packet kendes ved at ingen flags er sat (også kendt som null scan).

Denne type packet bruges til at scanne for åbne porte. Returneres der ikke noget svar på denne packet, indikere det at porten er åben. Returneres der en RST packet indikerer det at porten ikke er åben, per RFC 793.[6] Denne type scan er både hurtig og kan ofte passere igennem stateless firewalls og ACL regler.

---

[5] https://www.cisco.com/c/dam/en/us/td/docs/security/stealthwatch/management_console/securit_events_alarm_categories/SW_7_0_Stealthwatch_Security_Events_and_Alarm_Categories_DV_1_0.pdf

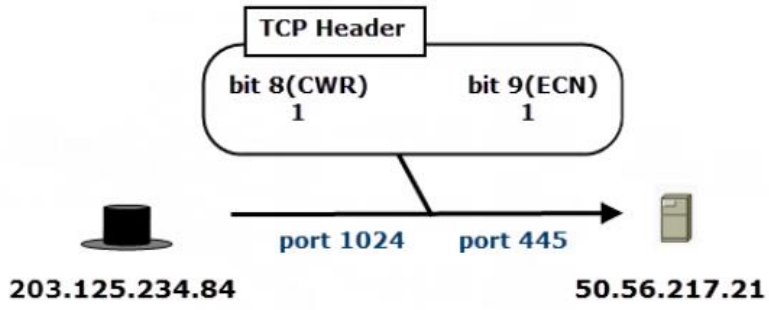
[6] https://www.rfc-editor.org/rfc/rfc793



Bad flag reserved:

Denne type packet kendes ved at ECN eller CWR flags er sat:

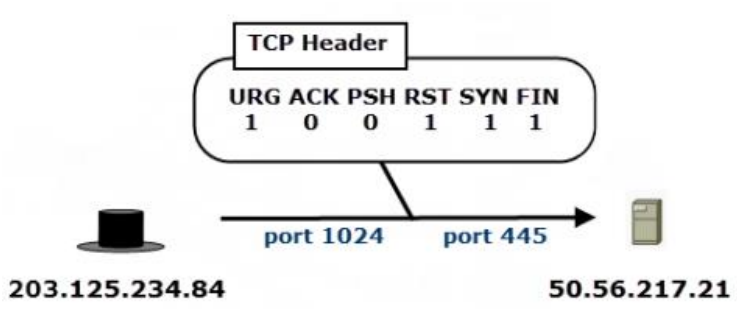
ECN og CWR anses som ”invalid flags”, og bruges meget sjældent i dag. Af denne grund er det mistænkeligt at se disse flags optræde, og det anses som at være onsidede packets der bruges som fingerprinting til at skaffe information om ofrets system. Dette fungerer ved at angriberen analysere svarene på disse type packets, og matcher dem op mod en database af svar fra forskellige styresystemer.



Bad flag RST:

Denne type packet kendes ved at RST flag er sat, en eller flere af FIN, URG, PSH.

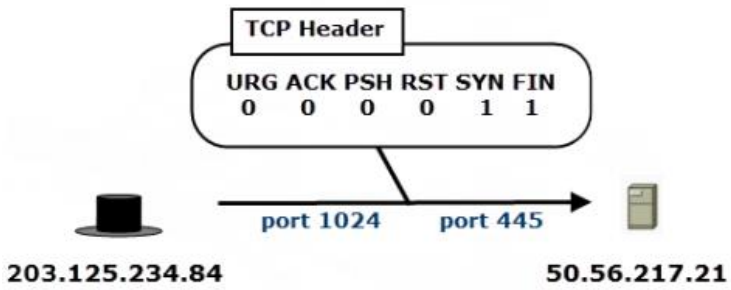
Disse packets kan angrebsmæssigt bruges til at afslutte legitime TCP forbindelser, hvilket kan resultere i tabt data og denial of service. En adversary kan altså i princippet spoofe RST pakker til at komme fra forskellige addresser ofret opretter forbindelse til, og derved afslutte disse forbindelser så ofret ikke længere modtager data fra dem.



Bad flag SYN:

Denne type packet kendes ved at både FIN og SYN flags er sat, en ”ulovlig” kombination i TCP protokollen.

Syn-Fin packets bruges som flooding, og modsat syn-flood som jeg undersøgte tidligere, giver denne version af flooding ofrets forsvarsmekanismer både opstart og afslutning af forbindelse at håndtere. Dette gøres i håb om at ofrets TCP/IP stack ikke kan håndtere denne kombination, og at forsvarsmekanismerne ikke er indstillet til at håndtere denne type flooding, da den er brugt knapt så hyppigt som andre versioner.

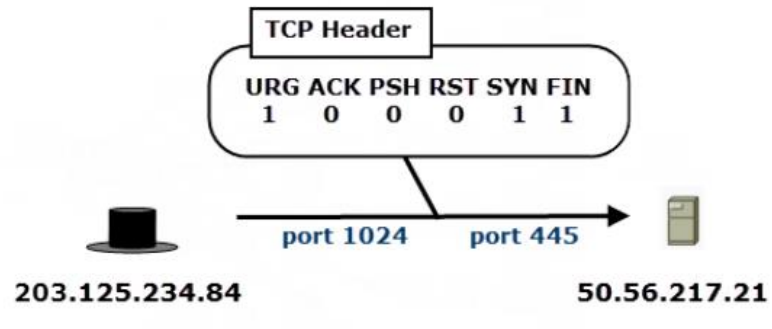


Bad flag URG:

Denne type packet kendes ved at URG er sat i kombinationer med alle andre flags end ACK.

URG er i de fleste tilfælde et forældet TCP flag som kun ældre systemer benytter sig af. Eftersom at det stadig er en del af protokollen, kan det stadig bruges som angreb, og det hænder at dette bliver brugt i forbindelse med DDOS. Det er min opfattelse at disse typer

angreb bruges i håb om at eventuelle sikkerhedsforanstalninger ikke beskytter mod mindre kendte angreb som en URG flood.



Hvorfor er Muninn og andre firmaer interesserede i en sådan slags beskyttelse?

I dette afsnit vil jeg fortælle om hvorfor vi er interesserede i TCP flagene, hvad vi er bange for ved dem, og hvad hackere typisk vil bruge disse typer angreb til. Dette vil give indsigt i hvorfor Muninn intereseret i at jeg beskæftiger mig med at udvikle dette produkt, i stedet for så mange andre ting jeg kunne lave for dem.

I sagens natur handler det om at beskytte sig imod videregående angreb der kan påvirke maskinerne på netværket, altså system sikkerhed.

Når en hacker først har information angående styresystemer gennem fingerprinting, firewall regler, åbne porte og services gennem port scanning, så har de derefter mulighed for at prøve at udnytte disse sikkerhedshuller til at hacke sig ind på de forskellige maskiner der kører på netværket. Produktet jeg udvikler er derfor ikke kun vigtigt i forhold til netværkssikerhed, men i lige så stor grad system sikkerheden for, for eksempel, en virksomhed.

Vi har selv lært på første semester hvordan man kan udnytte eksempeltvis åbne porte eller kørende services til at tvinge sig adgang til maskiner der kører på et netværk. Dette kan for

eksempel gøres ved hjælp af Metasploit, et framework der indeholder en større mængde software der er lavet til at udnytte forskellige vulnerabilities. Der findes nye vulnerabilities på forskellige styresystemer konstant, og det kan være alt fra unpatched services til ældre protokoler på forskellige porte der kan give hackere adgang. Et eksempel man ofte ser er at port 23 står åben, og det derfor kan være muligt at tvinge sig adgang til telnet på en maskine og bruge det som indgangsvinkel. Herfra kan man så forsøge sig med bruteforcing af login credentials, password guessing, password stuffing samt andre metoder, og hvis det skulle lykkedes hackeren at få adgang, kan værktøjer som mimikatz bruges til yderligere privilege escalation ved at "hente" passwords fra maskinens memory.

Herved støder vi på et andet punkt fra vores undervisning i system sikkerhed, nemlig system hardening. Dette princip omhandler oprustning af systemet til at kunne stå imod eventuelle angreb som tidligere beskrevet, og i denne kontekst er følgende punkter vigtige:

1. Patch management. Når vi netop snakker om TCP flag angreb som port og service scanning, er patch management et enormt vigtigt punkt. For værktøjer som metasploit, Empire og canvas, som altså er frameworks der indeholder hundredevis af forskellige exploits som kan udnytte services der kører på din maskine baseret på de porte og hvad der kører på dem, er det nødvendigt at holde disse opdateret.
2. Least privilege principle. Dette punkt er nødvendigt i tilfælde af at en maskine på netværket faktisk bliver kompromitteret, og det fungerer ved at holde skaden så lille som mulig. Ved at følge least privilege principle sikrer man sig at hver enkelt bruger har minimale rettigheder, baseret på deres rolle og behov. En virksomhed er for eksempel ikke interesseret i at en udvikler har adgang til ansattes privatdata, ligesom at HR medarbejdere ikke skal have adgang til kodebasen. På denne måde mindsker man hackeres potentielle datatilgang, og derved hvor meget skade de kan forvolde en virksomhed.
3. Logging og monitoring. Dette punkt er relevant når vi i forvejen snakker om netværksovervågning, fordi det er det samme princip, bare på maskine-niveau. I stedet for at holde øje med netværks traffik, holder vi med system monitoring øje med handlinger der bliver udført på maskiner. Et eksempel er Tripwire, et monitoring program der laver et hash på alle filer på et defineret område af maskinen, og gemmer det i en database. Når tripwire opdaterer og udregner hashes på disse filer

igen, ville det kunne se hvorvidt alle hashes matcher de gamle, og hvis de ikke gør det vil man kunne se hvilke filer der er blevet ændret, slettet eller oprettet. På den måde kan man alarmere om ændringer, og handle ud efter det. For at se hvordan en hacker er kommet frem til disse filer, kan man bruge logging til at trace de handliner hackeren har foretaget sig.

Et eksempel på hvordan svag netværkssikerhed kan gå ud over svag systemsikkerhed:

Nmap benyttes til at scanne maskiner på et netværk:

```
┌──(kali㉿kali)-[~/Desktop]
└─$ sudo nmap -vv -n -sn -T4 192.168.244.1/24
```

```
Host is up, received arp-response (0.00021s latency).
MAC Address: 00:0C:29:FA:DD:2A (VMware)
```

Vi scanner herefter den fundne maskine for porte og services.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ sudo nmap -vv -Pn -sS -A 192.168.244.136
```

Nmap returnerer resultat som blandt andet indeholder følgende:

```
PORT     STATE SERVICE     REASON         VERSION
21/tcp   open  ftp         syn-ack ttl 64 vsftpd 2.3.4
```

Vi bruger metasploit eller andre frameworks til at finde en passende moduler der kan bruges som exploit:

```
Matching Modules
================

   #   Name                                                Disclosure Date  Rank    Check  Description
   -   ----                                                ---------------  ----    -----  -----------
   0   auxiliary/scanner/smb/impacket/dcomexec             2018-03-19       normal  No     DCOM Exec
   1   auxiliary/scanner/smb/impacket/secretsdump                           normal  No     DCOM Exec
   2   auxiliary/scanner/smb/smb_ms17_010                                   normal  No     MS17-010 SMB RCE Detection
   3   auxiliary/scanner/smb/psexec_loggedin_users                          normal  No     Microsoft Windows Authenticated Logged In Users Enumera
   4   auxiliary/scanner/smb/smb_enumusers_domain                           normal  No     SMB Domain User Enumeration
   5   auxiliary/scanner/smb/smb_enum_gpp                                   normal  No     SMB Group Policy Preference Saved Passwords Enumeration
   6   auxiliary/scanner/smb/smb_login                                      normal  No     SMB Login Check Scanner
   7   auxiliary/scanner/smb/smb_lookupsid                                  normal  No     SMB SID User Enumeration (LookupSid)
   8   auxiliary/scanner/smb/pipe_auditor                                   normal  No     SMB Session Pipe Auditor
   9   auxiliary/scanner/smb/pipe_dcerpc_auditor                            normal  No     SMB Session Pipe DCERPC Auditor
   10  auxiliary/scanner/smb/smb_enumshares                                 normal  No     SMB Share Enumeration
   11  auxiliary/scanner/smb/smb_enumusers                                  normal  No     SMB User Enumeration (SAM EnumUsers)
   12  auxiliary/scanner/smb/smb_version                                    normal  No     SMB Version Detection
```

Vi spawner en reverse shell gennem vores exploit og har nu adgang:

```
su root
pwd
/root
whoami
root
```

Dette er et eksempel på først og fremmest dårlig netværkssikerhed, idet at scanning først og fremmest kan forekomme og ikke bliver blokkeret. Derefter eksemplificere det hvor vigtigt

det er at alle services kørende på ports er patchede, og hvor vigtigt det er at alarmere om disse scanninger, eller helt at blokkere dem.

Hackere kan ikke bare bruge disse angreb til at tilgå enkelte brugere, det kan være første led i at overtage mange maskiner på et netværk. Mitre frameworket[7] giver et godt og detaljeret indblik i hvordan et netværksangreb kan lede til adgang til maskiner, som så kan føre til privilege escalation, credential access af flere maskiner, lateral movement og til sidst exfiltration og impact som kan være katastrofalt på flere måder. Systemsikkerhed er altså afhængigt af dybdegående netværkssikerhed, og de to dele hænger derfor sammen. Det er altså vigtigt at have systemsikkerhed og eventuelle efterfølgende angreb i tankerne, når man udvikler og implementere netværkssikring.

Af samme grund er det vigtigt at udvikle disse netværkssikerhedsprodukter på en sikker og grundig måde.

Værktøjs overvejelser og valg:

Efter at have defineret hvilke angreb Muninn ønsker beskyttelse overfor, og hvorfor det er vigtigt, er næste del af fremgangsmåden at definere hvordan man skal beskytte sig over for netop de angreb.

I vores tilfælde har vi behov for et værktøj der kan analysere på deep packet-level samt stateful connections, hvilket vil kunne give os indblik i hvilke bits der er sat for de forskellige TCP flag, samt at holde overblik over TCP forbindelser og sikre mod flooding. Dette skal som sagt ske på to forskellige måder.

I første omgang er vi interesserede i at se på ”statiske” pakker, altså enkelte pakker i TCP forbindelser hvor flagene er sat på specifikke måder. For at kunne dette har vi brug for et værktøj der kan inspicere alle TCP segmenter på netværket, der kan læse deres flag og som

[7] https://attack.mitre.org/

kan programmeres til at gøre opmærksom på specifikke kombinationer af flag vi gerne vil advares om.

I anden omgang er vores mål at kunne analysere aktive TCP forbindelser og segmenterne der bliver sendt frem og tilbage, for at kunne opdage ondsidede forbindelser, baseret på TCP flag ved deres sekvenser eller hyppigheder, som tidligere forklaret.

I samarbejde med Muninn kiggede jeg på mange forskellige værktøjer og potentielle muligheder for at bygge noget der kunne håndtere denne opgave.

Det har været denne del af projektet der har været det sværeste og mest tidskrævende, da der har været *mange* forskellige ideer og foreslag, men få der har vist sig faktisk at være mulige, effektive eller relevante.

Der har været tre hovedpunkter der har været vigtige i overvejelserne over hvilket værktøj der skal bruges:

- Performance
- Sværhed (i form af udvikling)
- Implementering i Muninns pipeline

Værktøjs muligheder:

Rust: Rust er et programmeringssprog hvis styrker ligger i ydeevne og sikkerhed. Hvis vores værktøj skal skrives i et programmeringssprog, er det vigtigste aspekt hvor effektivit og ressourcestærkt sproget er, eftersom at mængden af data der skal gennemgåes og analyseres er ekstremt høj, og selv få procenters optimisering vil kunne gøre en stor forskel. Af denne grund er Rust's concurrency, altså evne til at multi-threade endnu et vigtigt egenskab og grund til at Rust kunne være en optimal mulighed. Det er dog vigtigt at pointere at Rust er et svært sprog, og ikke begyndervenligt hvis man ikke har kendskab til lignende sprog, som c++.

Pros:	Cons:

Høj ydeevne	Kompliceret
Multi-threading / parralel computing	Kræver forhåndsviden af low-level programmering
Low level programming	Nyt/småt økosystem
Memory management	Svært at integrere i Muninns pipeline
Sikkerhed	
Real-Time analyse	

Zeek: Zeek er, som tidligere nævnt, det program som Muninn primært bruger i deres sensor. Dette open-source sprog gør det muligt at anvende forskellige protocol-analyzers, der altså kan genkende og analysere netværkstraffik på et meget detaljeret niveau. Her kan der kigges på alt fra IP addresser til porte, protokoller, TCP flags, payload og mere. Zeek gør det muligt at definere ”events”, som kan triggers baseret på de parametre man sætter, for eksempel at et vidst antal telnet protokol pakker mellem to IP addresser er blevet set i netværkstraffikken. Eftersom det er et programmeringssprog i sig selv, er det muligt at definere den logik og de datastrukturer vi har brug for. Zeek er, i modsætning til Rust, ikke et ”hurtigt” sprog, i den forstand at det ikke vil kunne håndtere sammen mængde netværkstraffik og yde samme effektivitet som Rust.

Pros	Cons
Stærk dokumentation	Analyse baseres på logs
Bredt udvalg af protocol analyzers	Ikke real-time analyse
Nemt at implementere hos Muninn	Ikke effektiv header-analyse
Open source	Kræver kendskab til sproget

Suricata: Suricata er et open-source IDS/IPS system. Suricata er signatur-baseret, hvilket vil sige at der defineres nogle regelsæt for netværkstraffik, og suricata, som overvåger data på

ens interface, matcher trafikken med det definerede regelsæt, og genererer en alarm hvis disse regler bliver brudt. Suricata er skrevet i C, og det er derfor muligt selv at udvide koden eller udvikle plugins til det, men det burde ikke være nødvendigt til at udvikle regelsæt der passer det vi har brug for. Det vil også være et stort projekt selv at skulle udvikle videre på Suricatas codebase. Programmet er lavet til at være resourcenemt og hurtigt, men i stedet gå på kompromis med muligheden for selv at kunne programmere videre på det, som vi for eksempel nemmere kan i Zeek eller Rust. Suricata er programmet og designet til at læse data og sammenligne det med regelsæt på en specifik måde, der gør det mere effektivt end mange andre valgmuligheder.

Pros:	Cons:
Real-Time analyse	Kræver kendskab til signaturudvikling
Performance	Kompleks syntax
Protocol support	Svært at implementere hos Muninn
Open source	

Implementering i Zeek:

I mit samarbejde med Muninn var målet i først omgang at kunne implementere scripts i Zeek der skulle kunne notificere os om potentielt farlige TCP flag kombinationer og flooding.

Eftersom at Muninn primært bruger Zeek til at opdage problemer i netværkssikerheden, var det oplagt at jeg også skulle udvikle mit værktøj på samme måde, hvilket også var Muninns ønske. Dette ville både gøre det nemmere at implementere i deres NDR, men det er også nemmere for mig, eftersom at jeg har erfaring med sproget fra min praktikperiode hos dem.

Zeek tilbyder mange løsninger der hjælper med stateful forbindelses tracking. Dette er meget relevant for os når vi skal alarmere imod noget som flooding, hvilket er en stor del af denne opgave. Dette vil være nemt at implementere, derfor kiggede jeg først og fremmest på om flag kombinationerne var noget jeg kunne løse med Zeek.

Zeek løsning 1:

Idéen her er at bruge Zeeks protocol-analyzers til at analysere hver enkel Zeek pakke.

Zeeks protocol-analyzers kan altså analysere TCP packets på baggrund af header-information, og indeholder derudover også en liste af "events"[8], som Zeek koden reagerer på.

```
event tcp_packet(c: connection, is_orig: bool, flags: string, seq: count, ack: count, len: count, payload: string) {

    if (is_orig) {
        if ("F" in flags && "R" in flags) {
            print fmt("RST and FIN flags set in an outgoing packet: %s", c$id$orig_h);
            # Add your custom logic or alerting here
        }
    } else {
        if ("F" in flags && "R" in flags) {
            print fmt("RST and FIN flags set in an incoming packet: %s", c$id$orig_h);
        }
    }
}
```

Koden her, som også er inkluderet i bilag, er et bud på hvornår man kan bruge Zeek til at håndtere TCP flags, som fungerer således:

> Når Zeek fanger en TCP packet på interfacet det lytter til, trigger det eventet "tcp_packet"[9]. Eventet fanger parametrene der ses i paranteserne, som altså indeholder vigtig information som sessiondata, payload og andet header-data fra den givne packet. Det er på baggrund af dette at jeg har kunne udvikle scripts der kan alarmere baseret på flags. Som det ses i koden overfor, kan vi benytte de data objekter eventet returnere til os til at kigge på de forskellige fields af data de indeholder, heriblandt Flags som er en form for datastruktur vi kan inspicere.

```
brunoven3@brunoven3-virtual-machine:~/IdeaProjects/untitled/ZeekScripts$ zeek -r UPRF.pcapng test.zeek
RST and FIN flags set in an outgoing packet: 192.168.0.4
I
S
^dDa
Aa
^aADT
```

Ud fra dette event kan vi altså definere vores egen logik, som i dette tilfælde er at fange packets der indeholder både FIN og R flags, ligegyldigt hvilken vej de kommer fra (if-else statement). For at teste dette har jeg kørt scriptet mod en pcap der indeholder en sådan

[8] https://docs.zeek.org/en/master/scripts/base/bif/plugins/Zeek_TCP.events.bif.zeek.html
[9] https://docs.zeek.org/en/master/scripts/base/bif/plugins/Zeek_TCP.events.bif.zeek.html#id-tcp_packet

packet, som jeg har sendt gennem Scapy. Vi kan altså se at det virker, og protocol-analyzeren opfanger den ”ulovlige” packet.

Denne løsning virker altså, og kunne meget nemt implementeres i Muninns NDR. Problemet med produktet jeg udvikler er dog at det ikke bare skal kunne virke, men det skal kunne virke på enorme netværke, da Muninn sælger deres produkt til nogle af de største firmaer i Danmark og Europa. Det er derfor også vigtigt at have i tankerne at jeg skal udvikle et produkt der gør dets formål på effektiv vis, i forhold til komputerende resourcer.

Gennem yderligere research af Zeek documentation, fandt jeg dog frem til denne beskrivelse af TCP_Packet eventet:

“Generated for every TCP packet. This is a very low-level and expensive event that should be avoided when at all possible. It’s usually infeasible to handle when processing even medium volumes of traffic in real-time. It’s slightly better than new_packet because it affects only TCP, but not much. That said, if you work from a trace and want to do some packet-level analysis, it may come in handy.”

Dette information gjorde det tydeligt at det ikke var en holdbar møde at udvikle produktet på baggrund af. Zeek er altså ikke effektiv til deep packet inspection, på trods af at kunne håndtere det, men er derimod enormt effektiv til at håndtere protocol analyse på et højere niveau, ved for eksempel at overvåge brug af specifikke protokoller, IP addresser, porte og så videre.

Zeek løsning 2:

Eftersom at Zeeks mest åbenlyse funktion til at lave deep-packet analyse på TCP ikke virkede, måtte jeg kigge efter alternativer. Jeg havde mange overvejelser, for eksempel at bruge Zeeks connection logs til at analysere hver enkel TCP forbindelse og packets, og på trods af at det formentligt ville virke, ville det ikke give real-time alarmer, da det består af logs fra tidligere forbindelser, og det ville meget ueffektivt.

En anden løsning jeg kom på var at bruge Zeek eventet ”Connection state remove”[10]. Dette event indeholdte en real time ”history” af en aktiv TCP forbindelse, og indeholdte et data field der undersøgte inconsistent packets, altså ”ulovlige” kombinationer af packets.

Ud fra mine pcap tests, viste resultat at Zeek fandt alle de ulovlige kombinationer, som ses ovenfor i det printede resultat, hvor ”I” står for inconsistent packets. Dette event løste altså opgaven med at alarmere om ulovlige kombinationer, men problemet var at det ikke viste hvilket kombinationer det var.

Som tidligere nævnt i mit afsnit om Mitre, er der mange forskellige tilgange hackere kan bruge til at angribe netværk og maskiner. På trods af at det er vigtigt at vide at man er under angreb, er det ligeså vigtigt at vide hvordan man er blevet angrebet, og som Mitre frameworket viser, kan man bruge information om typen af angrebet til at beskytte sig mod det næsten trin i en hackers angrebsplan. Jeg var derfor ikke tilfreds med resultatet, og ville finde en anden mulighed for at kunne alarmere og give en detaljeret til ofret om hvilket angreb der var forekommet.

Implementering ved hjælp af programmeirng (Rust):

Eftersom at Zeek var udelukket til udvikling af opgaven, blev jeg i samarbejde med Muninn opfordret til at undersøge hvorvidt produktet kunne programmeres uafhængigt af andet software, som Zeek.

Idéen var her at bruge Rust til at udvikle et program der lyttede til netværksinterfacet og på baggrund af det kunne jeg implementere min egen logik der skulle holde øje med TCP header indstillingerne. Dette er, som tidligere nævnt, fordi at et velprogrammeret Rust program vil være meget effektivitet i det at det har en mindre indvirkning på brugen af systemets hukommelse, CPU og andre resourcer.

Det lykkedes mig at skrive et program der kunne lytte til interface, analysere TCP headere og udskrive advarsler, et eksempel på dette kan ses herunder og i bilag.

[10] https://docs.zeek.org/en/master/scripts/base/bif/event.bif.zeek.html#id-connection_state_remove

```rust
let mut cap = pcap::Capture::from_device(interface)
    .unwrap()
    .promisc(true)
    .snaplen(5000)
    .open()
    .unwrap();

while let Ok(packet) = cap.next() {
    if let Some(ethernet_packet) = EthernetPacket::new(&packet.data) {
        match ethernet_packet.get_ethertype() {
            IpNextHeaderProtocols::Tcp => {
                // Kig på TCP pakker
                let tcp_packet = TcpPacket::new(ethernet_packet.payload());
                if let Some(tcp_packet) = tcp_packet {
                    let flags = tcp_packet.get_flags();

                    // Check check for RSTFIN pakke
                    if flags & pnet::packet::tcp::TCP_FLAG_RST != 0
                        && flags & pnet::packet::tcp::TCP_FLAG_FIN != 0
                    {
                        println!(
                            "Illegal TCP Flag Combination (RST and FIN): {}:{} > {}:{}; Seq: {}, Ack: {}",
                            ethernet_packet.get_source(),
                            tcp_packet.get_source(),
                            ethernet_packet.get_destination(),
                            tcp_packet.get_destination(),
                            tcp_packet.get_sequence(),
                            tcp_packet.get_acknowledgment()
                        );
                    }
```

Det næste problem opstod nu i at mit produkt som sagt skal kunne håndtere stateless packets mod ulovlige flag kombinationer, men også statefull forbindelser der holder øje med parametre indenfor flooding. På trods af at været uddannet datamatiker, havde jeg ikke erfaring i low-level sprog som Rust, og jeg var ikke sikker på hvordan jeg skulle skrive kode der på optimal vis brugte datastrukturer til at holde på TCP forbindelser og dets header data.

Derudover viste det sig at for at implementere koden i Muninns pipeline, skulle det gå igennem ”Broker” protokollen, et andet trin der ummidelbart lød meget indviklet og tidskrævende.

Det var bestemt en effektiv måde at analysere TCP header data, men havde flere andre udfordringer.

Implementering i Suricata

Efter at have haft problemer med Zeeks deep packet inspection, og Rusts stateful connection overvågning, ville det give mening at udvikle mit produkt af flooding alarmer i Zeek og flag

kombinations alarmer i Rust. Dette ville være relativt nemt, men også tidskrævende for mig, og potentielt problematisk for Muninn at få tid til at integerere begge dele ind i deres NDR.

Jeg var meget opsat på at udvikle dette produkt igennem ét værktøj, der kunne håndtere det hele.

Det næste værktøj jeg ville udforske var Suricata. Suricata fungerer både som behaviourbaseret IDS men også med signaturer. Signaturer er regelsæt der definerer parametre for hvilke typer packets på et givent netværk der skal alarameres om. Disse parametre er alt fra porte og ipadresser, begge veje, til protokoller, header options, antal deraf, med mere.

Efter store overvejelser valgte jeg i sidste ende at udvikle mit produkt på baggrund af Suricata. På trods af at Zeek ville være meget nemmere at implementere i Muninns IT-infrastruktur, ville det derimod både være sværre at udvikle, og samtidigt enormt resourcekrævende som tidligere nævnt. Suricata bliver netop beskrevet som high-performance, og ved at besøge github kan vi se at det primært er skrevet i C og RUST, to ”low-level” programmeringssprog der giver adgang til memory-management og er derved programmeret på en mere ”effektiv” måde, der gør packet inspection hurtigere.

Opsætning af Suricata:

Suricata installeres på en given maskine, og derefter laves signaturer. I Suricata.yaml filen indstilles softwaren gennem variabler der kan benyttes i signaturer.

```
vars:
  # more specific is better for alert accuracy and performance
  address-groups:
    HOME_NET: "[192.168.0.0/16,10.0.0.0/8,172.16.0.0/12]"
    #HOME_NET: "[192.168.0.0/16]"
    #HOME_NET: "[10.0.0.0/8]"
    #HOME_NET: "[172.16.0.0/12]"
    #HOME_NET: "any"

    EXTERNAL_NET: "!$HOME_NET"
    #EXTERNAL_NET: "any"
```

Her kan man definere subnets og eksterne addresser som variabler der kan bruges i signaturer.

```
rule-files:
 ## - suricata.rules
  - /etc/suricata/local.rules
```

Derudover kan man linke til egne filer der indeholder signaturer man selv har oprettet.

Produkt:

Gennem suricata har jeg udviklet og testet signaturer der beskytter imod alle angreb jeg tidligere har defineret i samarbejde med Muninn. Mit produkt er altså en fil af suricata signaturer, vedhæftet i bilag. Jeg vil i dette afsnit vise signaturene, forklarer dem og bevise at de virker gennem testcases.

IP adresser samt porte er ikke definerede, eftersom at dette kommer an på virksomhedernes egne behov. Alle TCP test packets er hovedsageligt sendt til port 80, men er også testet på andre porte med f.eks. nmap scans.

Det skal derudover også pointeres at der er mange indstillinger jeg ikke har implementeret i mine regler, da det er noget virksomheder selv skal streamline. Dette er blandt andet:

Home/External networks

Flow_Established_To_Server: Signaturer der kun reagerer på oprettede forbindelser

Thresholds: Typer af pakker, tracking af IP addresser, tracking af packets til forskellige porte.

Limit: Hvor ofte signaturen skal udsende en alarm, for at suricata ikke udsender alarm for hver enkel ulovlig packet modtaget.

Disse parametre er enormt vigtige, men ikke noget jeg kan definere på egen hånd da det afhænger af virksomhedernes egne behov og størrelse på netværk.

Det har været vigtigt for Muninn at alt bliver testet gennem simulerede angreb, og ikke bare enkelte pakker fra f.eks. Scapy, hvis muligt.

De viste testcases til hver signatur er kun en af mange tests jeg har lavet, for at teste enhver mulighed for false positives fra forskellige kombinationer.

1) Bad Flag ACK

Denne type packet kendes ved at ACK ikke er sat, men URG, FIN, PUSH er.

```
#Bad ACK flag
alert tcp any any -> any any (msg:"Bad ACK flag PSH"; flags:P; sid:1997;)
alert tcp any any -> any any (msg:"Bad ACK flag URG"; flags:U; sid:1994;)
alert tcp any any -> any any (msg:"Bad ACK flag FIN"; flags:F; sid:1993;)
```

Denne signaturer tjekker hvorvidt PSH, URG eller FIN flags er sat uden andre flags.

Testcase:

For at teste denne regel har jeg brugt hping3 til at floode testmaskinen med packets der udelukkende indeholdte de relevante flags.

```
┌──(kali㉿new-hostname)-[~]
└─$ sudo hping3 -c 15000 -d 120 -U -w 64 -p 80 --flood --rand-source 192.168.244.131
sudo: unable to resolve host new-hostname: Name or service not known
[sudo] password for kali:
HPING 192.168.244.131 (eth0 192.168.244.131): U set, 40 headers + 120 data bytes
hping in flood mode, no replies will be shown
^C
--- 192.168.244.131 hping statistic ---
541090 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
```

No.	Time	Source	Destination	Protocol	Length	Info
19813	6.714381768	146.81.133.120	192.168.244.131	TCP	176	9764 → 80 [URG]
19814	6.714780172	12.25.28.3	192.168.244.131	TCP	176	9765 → 80 [URG]
19815	6.714962574	231.67.8.65	192.168.244.131	TCP	176	9766 → 80 [URG]
19816	6.714975374	218.9.7.176	192.168.244.131	TCP	176	9767 → 80 [URG]
19817	6.714988074	46.22.129.46	192.168.244.131	TCP	176	9768 → 80 [URG]
19818	6.715244577	45.68.138.24	192.168.244.131	TCP	176	9769 → 80 [URG]
19819	6.715257277	208.128.33.207	192.168.244.131	TCP	176	9770 → 80 [URG]
19820	6.715269977	229.255.133.81	192.168.244.131	TCP	176	9771 → 80 [URG]
19821	6.715282678	14.19.56.3	192.168.244.131	TCP	176	9772 → 80 [URG]
19822	6.715295378	212.80.51.14	192.168.244.131	TCP	176	9773 → 80 [URG]
19823	6.715308078	46.36.233.134	192.168.244.131	TCP	176	9774 → 80 [URG]
19824	6.715500680	246.151.134.125	192.168.244.131	TCP	176	9775 → 80 [URG]

Det ses at testmaskinen modtager floodingen af URG packets. Disse angreb har jeg foretaget for PSH, URG og FIN floodings.

Genererede alerts fra signaturer findes i Suricatas Fast.log

```
01/05/2024-04:42:55.551288  [**] [1:1994:0] Bad ACK flag URG [**]
```

```
01/05/2024-04:44:35.952983  [**] [1:1997:0] Bad ACK flag PSH [**]
 192.168.244.131:80
```

```
01/05/2024-04:27:18.544542  [**] [1:1993:0] Bad ACK flag FIN [**]
192.168.244.131:80
```

2) **Bad flag all:**

```
alert tcp any any -> any any (msg:"All Flags Set"; flags: RASPUF; sid:1006445;)
```

Denne type packet kendes ved at alle flags er sat.

Signaturen definerer i ”flags:” feltet at alle bits skal være sat.

Testcase:

```
┌──(kali㉿new-hostname)-[~]
└─$ sudo hping3 -c 15000 -d 120 -U -S -A -F -R -P -w 64 -p 80 --flood --rand-source 192.168.244.131
sudo: unable to resolve host new-hostname: Name or service not known
HPING 192.168.244.131 (eth0 192.168.244.131): RSAFPU set, 40 headers + 120 data bytes
```

```
01/05/2024-04:49:47.550158  [**] [1:1006445:0] All Flags Set [**]
P} 177.35.64.10:52959 -> 192.168.244.131:80
```

3) **Bad flag NoFlag (Null scan):**

```
alert tcp any any -> any any (msg:"No Flags Set"; flags: 0; sid:100003;)
```

Denne type packet kendes ved at ingen flags er sat (også kendt som null scan).

Testcase:

Nmap nullscan

```
┌──(kali㉿new-hostname)-[~]
└─$ sudo nmap -sN 192.168.244.131
```

4) **Bad Flag Reserved**

```
#bad flag reserved:
alert tcp any any -> any any (msg:"EC flags"; flags: E+; sid:108883255;)
alert tcp any any -> any any (msg:"CWR flags"; flags: C+; sid:108883255;)
```

E+ definerer at signaturen bliver triggered hvis der modtages en packet med ECN flaget sat, *eller* hvis der modtages en packet med ECN sat samt andre flags.

C+ definerer den samme regel, dog med CWR flaget i stedet.

Testcase:

```
>>> from scapy.all import IP, TCP, send
...:  ...:
...:  ...: # Craft a packet with both SYN and FIN flags set
...:  ...: packet = IP(dst="192.168.244.131") / TCP(dport=80, flags="C")
...:  ...:
...:  ...: # Send the packet
...:  ...: send(packet)
...:  ...:
.
Sent 1 packets.
>>> from scapy.all import IP, TCP, send
...:  ...:
...:  ...: # Craft a packet with both SYN and FIN flags set
...:  ...: packet = IP(dst="192.168.244.131") / TCP(dport=80, flags="E")
...:  ...:
...:  ...: # Send the packet
...:  ...: send(packet)
...:  ...:
.
Sent 1 packets.
```

```
01/05/2024-05:12:12.974504  [**] [1:1022353255:0] CWR flags [
[Priority: 3] {TCP} 192.168.244.138:20 -> 192.168.244.131:80
01/05/2024-05:12:35.205511  [**] [1:188:0] Bad URG flag [**]
ority: 3] {TCP} 192.168.244.138:20 -> 192.168.244.131:80
01/05/2024-05:12:35.205511  [**] [1:10255:0] EC flags [**] [C
ity: 3] {TCP} 192.168.244.138:20 -> 192.168.244.131:80
```

5) **Bad Flag RST:**

Denne type packet kendes ved at RST flag er sat samt en eller flere af FIN, URG, PSH.

```
#RST + any others (temp)
alert tcp any any -> any any (msg:"RST + ACK "; flags:RA; sid:15335;)
alert tcp any any -> any any (msg:"RST + SYN "; flags:RS; sid:15315;)
alert tcp any any -> any any (msg:"RST + PUSH "; flags:RP; sid:15235;)
alert tcp any any -> any any (msg:"RST + FIN "; flags:RF; sid:153225;)
alert tcp any any -> any any (msg:"RST + URG "; flags:RU; sid:15115;)
```

I dette tilfælde mangler jeg en operator der kan specificere bestemte flags der skal være sat, i stedet for alle andre flags som ”+” bruges til. Derfor har det været nødvendigt at lave regler for hvert enkelt.

Testcase:

```
01/05/2024-06:20:46.830787  [**] [1:15115:0] RST + URG  [**]
iority: 3] {TCP} 192.168.244.138:20 -> 192.168.244.131:80
01/05/2024-06:20:49.577175  [**] [1:1000234325:0] SA  packet
ted Information Leak] [Priority: 2] {TCP} 192.168.244.138:56
01/05/2024-06:20:49.577237  [**] [1:1000234325:0] SA  packet
ted Information Leak] [Priority: 2] {TCP} 192.168.244.138:56
01/05/2024-06:20:54.271401  [**] [1:15335:0] RST + ACK  [**]
iority: 3] {TCP} 192.168.244.138:20 -> 192.168.244.131:80
01/05/2024-06:20:54.271401  [**] [1:1002343255:0] RA TEST [*
Priority: 3] {TCP} 192.168.244.138:20 -> 192.168.244.131:80
01/05/2024-06:20:54.271462  [**] [1:1002343255:0] RA TEST [*
Priority: 3] {TCP} 192.168.244.138:20 -> 192.168.244.131:80
01/05/2024-06:21:06.224345  [**] [1:15315:0] RST + SYN  [**]
iority: 3] {TCP} 192.168.244.138:20 -> 192.168.244.131:80
01/05/2024-06:21:13.465605  [**] [1:153225:0] RST + FIN  [**
riority: 3] {TCP} 192.168.244.138:20 -> 192.168.244.131:80
01/05/2024-06:21:22.915904  [**] [1:15235:0] RST + PUSH  [**
riority: 3] {TCP} 192.168.244.138:20 -> 192.168.244.131:80
```

6) **Bad SynFin**

Denne type packet kendes ved at både FIN og SYN flags er sat, en ”ulovlig” kombination i TCP protokollen.

```
alert tcp any any -> any any (msg:"SynFin Detected"; flags: SF; sid:1053255;)
```

Testcase:

```
>>> from scapy.all import IP, TCP, send
... :
... : # Craft a packet with both SYN and FIN flags set
... : packet = IP(dst="192.168.244.131") / TCP(dport=80, flags="SF")
... :
... : # Send the packet
... : send(packet)
01/05/2024-06:25:06.741756  [**] [1:1053255:0] SynFin Detected [*
l)] [Priority: 3] {TCP} 192.168.244.138:20 -> 192.168.244.131:80
```

7) Bad Flag Urg

Denne type packet kendes ved at URG er sat i kombinationer med alle andre flags end ACK. Vi kan bruge "!" operatoren til at definere at der skal ses et URG flag, og ethvert flag bortset fra ACK.

```
#bad URG
alert tcp any any -> any any (msg:"Bad URG flag"; flags:U!A; sid:188;)
```

Testcase:

Til at teste dette kan vi bruge nmaps "-sX" scan, også kendt som xmas scan, eftersom at det scanner testmaskinen med FIN PSH URG packets. Dette burde altså trigger vores regel.

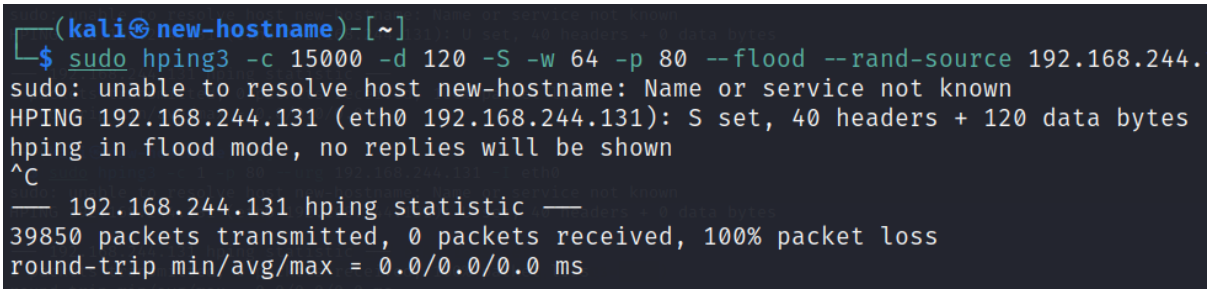
1379	13.438080700	192.168.244.138	192.168.244.131	TCP	62	48694 → 20222 [FIN, PSH, URG]
1380	13.438174499	192.168.244.131	192.168.244.138	TCP	56	20222 → 48694 [RST, ACK] Seq=1
1381	13.438093400	192.168.244.138	192.168.244.131	TCP	62	48694 → 987 [FIN, PSH, URG] Se
1382	13.438256298	192.168.244.131	192.168.244.138	TCP	56	987 → 48694 [RST, ACK] Seq=1 A
1383	13.438478295	192.168.244.138	192.168.244.131	TCP	62	48694 → 32776 [FIN, PSH, URG]

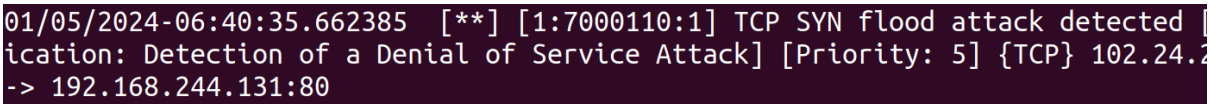
```
01/05/2024-06:30:15.590985  [**] [1:188:0] Bad URG flag [**] [
iority: 3] {TCP} 192.168.244.138:38462 -> 192.168.244.131:514
```

8) Syn Flood

```
#Synflood
alert tcp any any -> any any (msg:"TCP SYN flood attack detected"; flags:S;
threshold: type threshold, track by_dst, count 20, seconds 60;
classtype:denial-of-service; priority:5 ;sid:7000110; rev:1;)
```

Testcase:





Diskussion:

Undervejs i min opgave har en af pointerne været at mit produkt skulle hjælpe med at forhindre hackere at opnå adgang til maskiner på netværket produktet skal hjælpe med at beskytte. Eftersom at mine suricata signaturer fungerer ved at give advarsler til brugeren, kan man diskutere hvorvidt det er beskyttelse nok, eftersom at det fungerer som en IDS, og ikke en IPS der rent faktisk stopper angrebene.

Som jeg tidligere har nævnt er det selvfølgelig meget brugbar information at vide at der er foregået visse angreb, og det er også grunden til at Muninn har efterspurgt præcis denne funktion. Det giver muligheden for at tage handling og sikre sig mod fremtidige angreb der baseres på information bag for eksempel TCP flag scanninger af maskiner. Spørgsmålet er dog om der ikke skal bygges en prevent mekanisme videre på disse alarmer som Suricata genererer.

En idé er at implementere en form for machine learning der skal fungerer som en prevent funktion, og at denne machine learning kan drage data fra suricata alarmer, der baseres på hvem der angriber, hvordan de gør det, og hvad deres formål kunne være. Herfra kunne man optimalt udvikle chain of events, der kigger på hackerens valg af angreb baseret på TCP flagene, og hvad dette angreb har ført til at fremtidige angreb på systemerne. På trods af at det ville være svært at udvikle, kunne det både give Muninn et indblik i hvad det præcise

formål med angrebet er, men også at give prevent mekanismen en forebyggende effekt ved at formå at stoppe den angribende part før de når til det næste angreb.

Det ses ofte at IDS logs bliver brugt til machinelearning i andet software, og at konfigurere mit produkt på en måde der videregiver informationen til en prevent funktion baseret på machine learning virker som et meget værdifuldt tiltag. Man kunne også drage inspiration fra Mitre attack frameworket, der giver en mapping over de forskellige strategier, teknikker og underteknikker som hackere ofte bruger.

Som jeg har lært gennem min praktik periode har Muninn er spørgsmålet selvfølgelig hvor meget værdi det har. Det skal måles på flere forskellige parametre, blandt andet hvor effektivt det vil være, hvor lang tid det vil tage at udvikle, hvor meget data der er behov for og hvor godt det kan sælges til kunder.

Samtidigt kunne det være vigtigt at videreudvikle produktet ved at gøre brug af Suricatas log funktion. Suricata logger alle forbindelser, og man kunne bruge session data sammen med data fra alert logging til at generere en samlet log fil der giver et gennemgående overblik med vigtig data over angrebet, og eventuelt bruge det til at træne machinelearning.

Reflektering:

I påbegyndelsen af dette projekt var mit mål både at udvikle et brugtbart produkt, men derudover også at lære om forskellige IT-sikkerheds værktøjer og afprøve dem, så jeg havde erfaring med dem i min fremtidige karriere.

Jeg havde i tankerne at jeg relativt nemt kunne udvikle mit produkt både gennem Zeek, Rust og eventuelt andre værktøjer jeg ville finde henover projekt perioden. Jeg måtte dog hurtigt indse at når det kommer til IT-Sikkerheds værktøjer, er det ikke ligeså meget software og ikke lige så mange resourcer som i software udvikler verdenen. Projektet endte med at blive meget sværre end forudset, da jeg konstant manglede information og dokumentation på de

forskellige værktøjer jeg ville bruge, selv når det omhandlede noget så ”basalt” som TCP packet analyse.

Det var derudover en ny udfordring at skulle tilpasse sit arbejde til en virksomheds behov, og det betød at meget af det research og mange af de idéer jeg havde til projektet i sidste ende ikke kunne bruges, fordi det ikke passede til Muninns IT-infrastruktur. Zeek var besværligt på grund af effektivitets krav, Rust var svært eftersom at dataen skulle sendes gennem Broker protokollen, blandt mange andre udfordringer.

En anden overraskelse og udfordring var hvor tidskrævende det var at teste produktet i hver fase. På trods af at jeg selv synes at jeg er god til at finde værktøjer til at teste regelsæt, eller at oprette pcaps til samme formål, tog det enormt lang tid, da de hver især skulle passe til forskellige formater og angreb.

I mit afsnit om Suricata produktet har jeg vedhæftet testcases, men det er kun en lille del af enormt mange tests der skulle udføres for at sikre imod false positives. Eksempelvis har jeg i ”Bad flag URG” testcasen vedhæftet et xmas scan som test, men jeg har derudover også skulle teste det med over 20 andre kombinationer, for at sikre at alle flag kombinationer opfanges, og ikke generere forkerte resultater. Dette har været tilfældet med hvert eneste signatur jeg har lavet i Suricata. På trods af at have været enormt tidskrævende, har det været en god øvelse for mig, og givet mig endnu mere erfaring med wireshark, scapy, nmap mm.

Det har samtidigt givet mig ideen til at udvikle min egen software hvor man kan specificere hvilke typer packets man vil generere og hvor mange.

At arbejde med dette projekt har generelt givet mig det indtryk at IT-sikkerheds verdenen er langt mindre udviklet end man egentlig skulle tro, eftersom at det spiller så stor en rolle i vores verden. Der har igennem min praktik og min bachelor opgave været meget manuel arbejde, få værktøjer at vælge imellem og lidt eller ingen dokumentation at finde. Dette har dog lært mig at eksperimentere mere selv, for selv at finde frem til de svar der ikke er at finde på internettet, og lært mig at bruge en ”trial and error approach” som jeg synes har været med til at give mig meget ny viden på mange forskellige områder.

Hvad ville jeg have gjort anderledes?

I en større opgave, specielt når man arbejder for en virksomhed, lærer man rigtig meget, og som nævnt i dette afsnit har der været rigtig mange udfordringer. Jeg synes derfor det er vigtigt at reflektere over hvad jeg kunne have gjort anderledes igennem mit projekt. Ét punkt som jeg vil fokusere meget en anden gang, er at blive mere indforstået med kundens (Muninn) IT-infrastruktur, så jeg forstår præcist hvad kravene er, og hvilke muligheder jeg har når jeg skal udvikle mit produkt. Det var frustrerende, på trods af at jeg lærte meget af det, at skulle finde på nye idéer og tilgange til mit produkt, fordi at jeg ikke var blevet indviet i hvordan det skulle integreres i Muninns NDR. Jeg vil derfor bruge flere kræfter på at forstå virksomhedens krav, og samtidigt prøve at skabe bere dialog for at forstå hvilke muligheder jeg har.

På trods af de mange udfordringer har denne proces været helt enormt lærerigt for mig. Jeg er blevet klogere på:

- Værktøjer tilegnet netværksovervågning
- TCP protokollen
- TCP baserede angreb
- Zeek
- Signatur baserede IDS'er
- Indsamling af PCAPS
- Testing af sikkerhedssoftware

Konklusion:

- **Hvordan kan man bruge deep packet inspection til at sikre netværk mod TCP flag angreb?**
    - Hvilke værktøjer kan bruges til dette, og hvordan?
    - Hvordan adskiller disse værktøjer fra hinanden, og hvornår skal de hver især bruges

Der er mange forskellige måder at sikre netværk mod TCP flag angreb, og det kommer an på hvilke krav og tilgang du skal følge.

Du kan opnå dette ved at bruge værktøjer som Zeek, Suricata eller programmeringssprog, der adksiller sig ved hvor effektive de er til at beskytte mod forskellige typer angreb, og deres tilgang til det.

Zeek kan du bruge til at overvåge stateful angreb som floodings, eftersom at det er beregnet til ”high-level” overvågning. Det er dog ikke tilegnet TCP-header analyse, der gør det mindre effektivt til opgaver som at overvåge flag kombinationer.

Programmeringssprog som Rust kan du bruge til at overvåge flag kombinationer, eftersom at du nemt kan implementere din egen logik og hente den type data du har brug for. Det er dog mindre effektivt til stateful overvågning, der kræver effektive datastrulturer og algoritmer.

Suricata dækker begge punkter på effektiv vis da det er muligt at definere effektive signaturer til både stateful overvågning og deep-packet inspection.

Du skal derfor specificere kravene der skal opnåes, og derefter vælge et eller flere af disse værktøjer der på effektiv vis kan yde den beskyttelse der er behov for.

Jeg har udviklet mit produkt på baggrund af Suricata, og på baggrund af grundige tests ved jeg at det efter implementering hos Muninn vil kunne alarmere om de specificerede TCP flag angreb som virksomheden ønskede. Udover at have udviklet et brugbart produkt, har jeg også formået at dokumentere forskellige værktøjers styrker og svagheder, hvilket er enormt brugbart information for Muninn, der vil implementere nye teknologier i fremtiden.

Bilag

Vedhæftet findes følgende bilag:

1. Suricata signature fil
2. Suricata yaml fil
3. Suricata alarm log
4. Zeek kode
5. Rust kode

Litteraturliste

https://www.noction.com/blog/tcp-flags#:~:text=In%20TCP%2C%20flags%20indicate%20a,PSH%2C%20FIN%2C%20and%20RST.

https://www.geeksforgeeks.org/tcp-flags/

https://kb.mazebolt.com/knowledgebase/all-tcp-flags-flood-xmas-flood/

https://www.cisco.com/assets/sol/sb/Switches_Emulators_v2_3_5_xx/help/250/index.html#page/tesla_250_olh%2Fsecurity_top.html%23

https://nmap.org/

https://docs.zeek.org/en/master/

https://docs.suricata.io/en/latest/

https://www.linkedin.com/pulse/implementing-network-traffic-analyzer-rust-luis-soares-m-sc-/

https://blog.devgenius.io/implementing-a-network-traffic-analyzer-in-rust-50a772bb6564

https://github.com/zeek/zeek

https://github.com/bro