# Klassiske hackerteknikker

Det fantes hackere lenge før internett, men hacking ble uten tvil mer interessant da hele verden ble koblet sammen.

Med tiden har hackertrusselen endret karakter, og mottiltakene har variert. Grovt forenklet kan man kalle 70- og 80-tallet for *passord-epoken*. De første virusene dukker også opp. På 90-tallet kom langt kraftigere PC-er, men ikke mange var på internett enda. Sikkerhetstrenden var *antivirus*.

Når vi kommer til år 2000 så er "alle" på nett. *Antivirus* fortsetter å dominere som sikkerhetsprodukt. *Brannmuren* får også mye oppmerksomhet. De første store *ormene* herjer med dårlig sikrede servere på nett, og programvareleverandører blir tvunget til å øke takten på *sikkerhetspatcher*. På denne tiden leser vi også de første nyhetssakene om kinesiske etterretningsoperasjoner mot amerikanske interesser. I tillegg begynner alle å bruke internett til bank og varehandel, og med det kommer også vinningskriminalitet, ID-tyveri og nettbank-trojanere.

Utover 2000-tallet ser vi en eksplosjon av nye hackerteknikker og mottiltak. Hackerne endrer fokus bort fra *nettverksprotokoller* og inn mot *sårbare filformater* og *rootkits*. På sikkerhetssiden blomstrer *nettverksmonitorering*, "forsvarbare nett", *incident response*, *penetrasjonstesting*. *CERT-er* og andre sikkerhetsmiljøer begynner å samarbeide internasjonalt om *attribusjon* av trusselaktører. Begrepet *APT* (*Advanced Persistent Threat*) blir tatt i bruk.

Rundt 2010 kommer *passordsikkerhet* tilbake på agendaen, etter en lang rekke datainnbrudd hvor hundrevis av millioner med brukernavn og passord kommer på avveie. Nå rammes man altså av hackere uten selv å bli hacket. *Passord-managere* og *flerfaktor-autentisering* er ny *best practice*, men ikke alle henger med.

Destruktive *løsepengevirus* florerer, og betaling i Bitcoin forlanges for å låse opp filene dine. De første sabotasjeoperasjonene skremmer verden; urananrikingsanlegget i Natanz rammes av *Stuxnet*, hele Saudi Aramcos IT-infrastruktur ødelegges av *Shamoon*, *Black Energy* forårsaker strømutfall i Ukraina i 2015 og 2016. En rekke sivile bedrifter rammes i enorm skala av bl.a. *WannaCry* og *NotPetya*.

Også Norge rammes, eksempelvis datainnbruddene ved Helse Sør-Øst, Sykehuspartner, Hydro og Stortinget.

I dag finnes det gode sikkerhetsprodukter for nær sagt alle nisjer. Man kan kjøpe *managed security*, leie *penetrasjonstestere*, programvaren vi kjører er gjennomtestet, utviklere har lært, administratorer vet å segmentere nettene og holde programvaren oppdatert, nettverkstrafikken er kryptert, brukere er mer oppmerksomme og IT-sikkerhet har egne utdanninger.

> Systemene våre er godt sikret – mot gårsdagens trusler.

Er du kreativ og dyktig, finner du fremdeles sårbarheter. Det er uten tvil blitt betydelig vanskeligere, men sikkerhetshullene er der og kan utnyttes. Kanskje trenger man ikke én, men å utnytte en håndfull sårbarheter, finurlig satt sammen for å komme forbi alle sikkerhetsmekanismene.

Har du lyst til å reise tilbake i tid og se på noen klassiske sårbarheter og hackerteknikker?

## 1983

Dette året blir *TCP/IP* påkrevd på *ARPANET* og det markerer i manges øyne internetts fødsel. *DNS* introduseres og toppnivådomenet *.no* tildeles Televerkets forskningsinstitutt. Internett er forbeholdt militæret og forskning. De første kommersielle mobiltelefonene lages dette året. Richard Stallman kunngjør GNU-prosjektet, mens Microsoft kunngjør Windows og Word 1.0. Apple IIe lanseres. Michael Jacksons *Thriller* spilles på radio.

David Lightman er nær ved å starte atomkrig mellom Sovjetunionen og USA, etter å ha hacket superdatamaskinen *WOPR* ved et amerikansk militæranlegg. Heldigvis skjedde akkurat dette kun på kino, i filmen WarGames.

I den virkelige verden arresterer FBI hackergruppen *414s* for hacking av bl.a. *Los Alamos National Laboratory*, *Sloan-Kettering Cancer Center* og *Security Pacific Bank*.

## 1984: EDB-sikkerhet

Industriens sikkerhetsutvalg (nå: Næringslivets Sikkerhetsråd) gir i 1984 ut boken *EDB-sikkerhet*. Man kan i forordet lese:

> "Den stadig økende bruk og avhengighet av EDB medfører et betydelig sikkerhetsproblem"

Om utvalget var mest bekymret for sabotasje eller spionasje er uvisst. Boken gir mange gode råd, både om passord, kryptering, *programtekniske sikringstiltak*, fysisk sikkerhet, organisering, drift og jus.

> "Et passordsystem må ikke være så komplisert i bruk at det skaper operasjonsmessige problemer. Passordene bør sammenstilles på en slik måte at de ikke blir for vanskelig å huske. Antall tegn i et passord bør være minimum 5, men heller ikke bestå av for mange."

Fem tegn lange passord? Dette er i dag latterlig lett å forsere, men var kanskje tilstrekkelig i en tid med oppringte forbindelser og trege modemer. Hackeren kunne derfor ikke teste mer enn 1-3 passord per minutt.

### Passordstyrke

La oss regne litt på *passordstyrke*. Antall permutasjoner av sifre (10 tegn) og bokstaver i det engelske alfabetet (26 tegn), ved passordlengde 5 er:

> (10 + 26)^5 = 36^5 ≈ 60 millioner

60 millioner passordforsøk er mye. Og selv om man i snitt vil gjette rett passord halvveis i prosessen, vil det allikevel ta forferdelig lang tid å teste alle kombinasjoner. Ved tre forsøk per minutt tar det over 38 år.

> 60000000 / (3 * 60 * 24 * 365) ≈ 38

Men passord består sjelden av helt tilfeldig sammensatte bokstaver og tall. Oftest er det ord, navn, datoer eller andre mønstre. Om du som hacker først prøver *sannsynlige* passordkombinasjoner, vil det neppe ta lang tid å finne et gyldig passord.

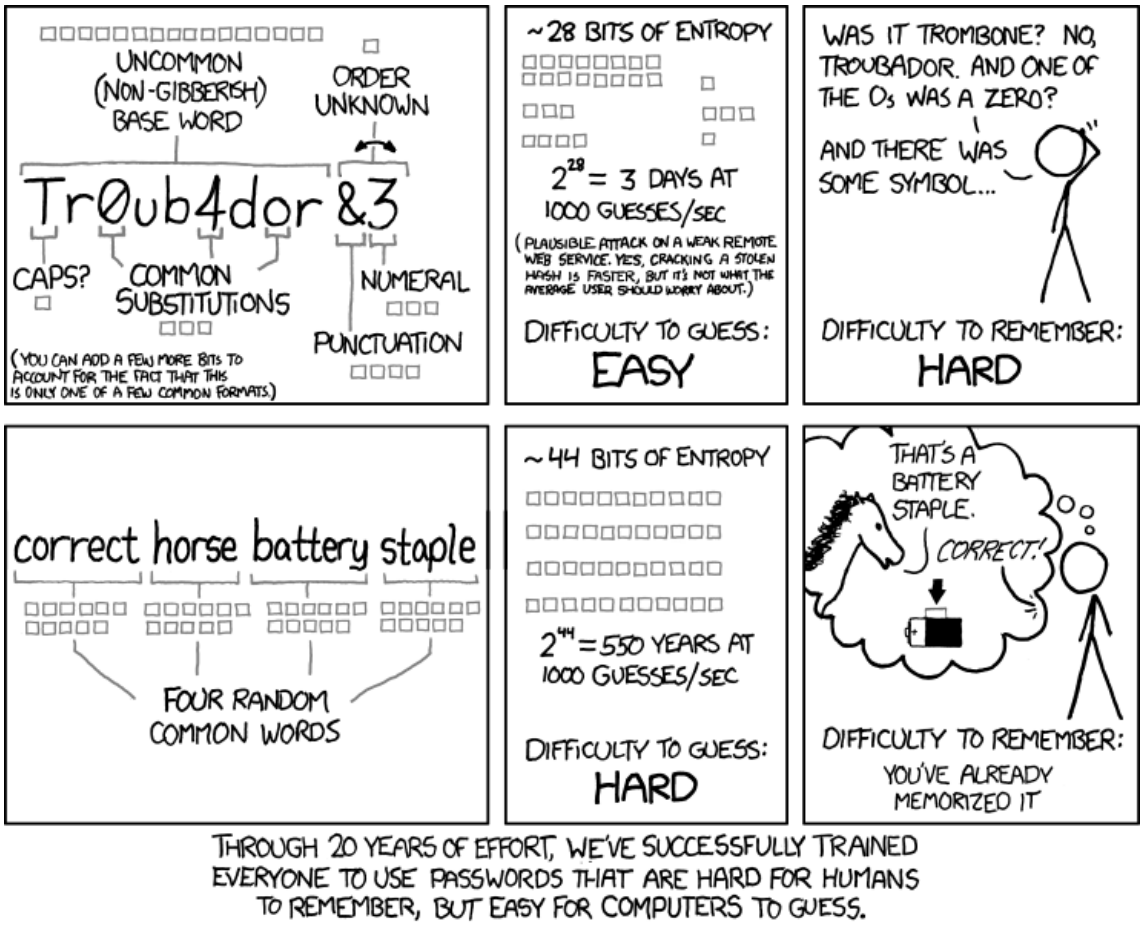
For å øke passordstyrken må man lage lengre passord, tillate flere tegn (f.eks. store og små bokstaver, spesialtegn, mellomrom, osv.), eller kreve større variasjon/uforutsigbarhet i passordet.

Så hvor *sterkt* er egentlig et passord på fem tegn? Man måler passordstyrke i *bits entropi*, der man kan tenke på entropi som uforutsigbarhet. Ett bit med entropi er som et myntkast; sannsynlighet: 50%, eller *1 av 2*. To bits er som å forutsi to myntkast: 25%, eller *1 av 4*. Sannsynligheten halveres altså for hvert myntkast/bit. I dag er det vanlig å anbefale passordstyrke på minst 42 bits entropi (2^42 ≈ 4,4 billioner). Et tilfeldig generert femtegnspassord ga 60 millioner kombinasjoner, omregnet til bits entropi blir det:

$\log_2(36^5) = 25.85 \approx 26$ bits entropi

Dagens minimumsanbefaling er altså å benytte passord som krever **73 tusen ganger** mer ressurser å forsere enn anbefalingen fra boken *EDB-sikkerhet*.

2^42 / 2^25.85 = 72717




Figur 1: XKCD 936

Ofte stjeler hackere databaser med brukernavn og passord. Når du som hacker kan teste passordene på egen maskin, går det nemlig an å teste milliarder av passord per sekund.

I slike databaser er ikke passordene lagret i klartekst, men kryptert med en matematisk enveis-funksjon – en hash-funksjon. Passord kryptert med en hash-funksjon kan ikke dekrypteres, men en passordkandidat kan krypteres med samme hash-funksjon, og så ser man om man får samme svar. *MD5* og *SHA* brukes ofte som hash-funksjon.

Ved å benytte en hash-funksjon som egner seg bedre for passord, eksempelvis *PBKDF2*, *BCRYPT*, *SCRYPT*, eller *ARGON2*, i kombinasjon med *salt*, og en fornuftig passord-policy, blir hackerens jobb veldig vanskelig.

De nevnte algoritmene skiller seg fra andre kryptografiske hash-funksjoner ved å være designet for å *kreve mye ressurser å regne ut*. Hash-funksjoner som *MD5* og *SHA* er, i motsetning, designet for å være svært raske.

## 1984: Hackers

Tilbake til åttitallet. I 1984 utgis også den norske boken *Hackers - datasnokere*, som forklarer hva hacking er, hvordan en hacker går frem og hva slags utstyr og programmer som trengs. Boken har endel programlistinger, blant annet for å generere passordkombinasjoner.

> "Hacking - mer spennende enn Detektimen"

Dette er også året Apple lanserer Macintosh, en 8MHz maskin med 128K RAM og diskettstasjon. På kino går Ghostbusters, Gremlins og Terminator. Treholt arresteres for spionasje.

## 1985

*Databladet "C"* spekulerer i artikkelen *Treholt: en avleggs spion* om *databasetapping* vil bli den nye store spionsaken, og forklarer hvordan fotografering er ut, og at en moderne spion kun trenger en Commodore 64 og et modem.

> "Klarer en hacker å bryte seg inn i en datamaskin, så klarer selvfølgelig en etterretningsorganisasjon det langt raskere"

## 1986

Sommeren 1986 oppdager Cliff Stoll en hacker i datasystemene til *Lawrence Berkeley National Laboratory*. Det markerer starten på en to år lang jakt på en gruppe øst-tyske hackere som bryter seg inn i hundrevis av systemer til forskningsinstitusjoner og forsvarsanlegg over hele USA. Hacker: Markus Hess m.fl. fra Chaos Computer Club. Oppdragsgiver: KGB. Betaling: kokain. Les hele den spennende historien i Stolls bok *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage*.

Hackeren får tilgang til en ubrukt brukerkonto via telenettet og *Tymnet*. Deretter skaffer han *superuser*-rettigheter ved å utnytte en sårbarhet i *movemail*-programmet (en del av Emacs på datidens UNIX-systemer). Cliff Stoll:

> "Over the past decade Stallman created a powerful editing program called Gnu-Emacs. But Gnu's much more than just a text editor. It's easy to customize to your personal preferences. It's a foundation upon which other programs can be built. It even has its own mail facility built in. Naturally, our physicists demanded Gnu; with an eye to selling more computing cycles, we installed it happily."

*Movemail* blir brukt, som navnet antyder, for å flytte epost fra *spool* (innkommende epost) til brukerens *INBOX*. For å gjøre det trenger programmet ekstra rettigheter på systemet, og kjøres *SUID root*.

### SUID – set-user-id

Ved å sette *suid*-rettigheten på et program blir det kjørt som programfilens eier, i stedet for som pålogget bruker. Når eieren er superbrukeren *root*, vil programmet alltid kjøre som *root*. Dette er samme mekanisme som *sudo*-programmet benytter på moderne systemer for å opptre på vegne av andre brukere. Se her:

```
$ stat `which sudo`
  File: /usr/bin/sudo
Access: (4755/-rwsr-xr-x)  Uid: (0/root)   Gid: (0/root)
         ^                          ^^^^


$ ls -l `which sudo`
-rwsr-xr-x 1 root root 161448 Jan.  1  13:37 /usr/bin/sudo
   ^         ^^^^
```

`/usr/bin/sudo` eies av *root* og har satt *suid*-bitet (4000). Når man som ordinær bruker kjører kommandoen *sudo*, vil *sudo* kjøre med rettighetene til superbrukeren *root*.

```
$ id
uid=1000(n00b) gid=1000(n00b) groups= ...

$ sudo id
uid=0(root) gid=0(root) groups= ...
```

> Du kan selv utforske *suid*. Ta en titt på øvelsene i katalogene `~/1_grunnleggende/2_setuid/` og `~/1_grunnleggende/3_injection/`.

En sårbarhet i et program med *suid*-bitet satt lar oss altså gjøre ting på systemet man normalt ikke ville hatt rettigheter til. Derfor vil ikke suid-programmer være skrivbare for andre enn eieren. Hvis du klarer å lure tilgangskontrollen er det trivielt å kjøre vilkårlig kode med samme rettigheter som eieren av filen. Et interessant eksempel på en sårbarhet som muliggjør dette er *Dirty COW* (CVE-2016-5195). Denne sårbarheten er av typen *race condition* eller *TOCTTOU* (*time-of-check to time-of-use*) i *virtual memory*-subsystemet i Linux-kjernen.

Problemet er at *movemail* ikke bare kopierer epost, men kan brukes til å kopiere nær sagt hva som helst – fra hvor som helst – til hvor som helst på systemet. Hackeren bruker

*movemail* til å installere en *cron*-jobb for *root*, og skaffer seg på den måten full tilgang til maskinen.

### Cron

*Cron* brukes på UNIX-type systemer til å kjøre kommandoer på et angitt tidspunkt, eller ved faste intervaller. Cron-jobber kan opprettes for både ordinære brukere og superbrukeren *root*. Kommandoene kjøres da med brukerens rettigheter.

Man trenger å være *root* for å opprette en *cron*-jobb for *root*.

Hvordan misbruker du dette i praksis? Hva med å få installert en *cron*-jobb for *root* som kjører dette scriptet?

```
#!/bin/sh
cp `which bash` /tmp/x
chmod u+s /tmp/x
```

Når *cron* har kjørt scriptet kan hackeren starte programmet `/tmp/x` og ha et interaktivt shell med *root* sine rettigheter. Dette er en type sårbarhet som kalles *LPE - Local Privilege Escalation*.

Moderne shell som kjøres *suid* vil av sikkerhetsgrunner droppe *suid*-rettighetene, med mindre man angir `-p` ved start.

```
$ stat /tmp/x
Access: (4755/-rwsr-xr-x)  Uid: (0/root)   Gid: (0/root)
         ^                          ^^^^


$ /tmp/x
x-5.0$ id
uid=1000(n00b) gid=1000(n00b) groups= ...

$ /tmp/x -p
x-5.0# id
uid=1000(n00b) gid=1000(n00b) euid=0(root) groups= ...
                              ^^^^^^^^^^^^
```

Spennende. Som Cliff Stoll sier:

> "Why is it drug addicts and computer aficionados are both called users?"

## 1988

Tidlig i november 1988 opplever internett noe nytt: *Morris*-ormen sprer seg fra UNIX-system til UNIX-system. I løpet av timer blir internett ubrukelig, og det tar dager før det blir rettet opp i. Når ormen stoppes har den angrepet rundt 10% av de 60000 systemene som er tilkoblet internett. Den første *CERT*-en opprettes av DARPA i kjølvannet av hendelsen.

Den norske delen av internett reddes ved at FFI-forskeren Pål Spilling (*Skal vi danse-Helene*'s farfar) rett og slett kobler hele Norge av nett; Det var enkelt å gjøre i 1988 – det var nok å trekke ut en kontakt :)


Figur 2: Forhindret norsk data-katastrofe

Verden er allerede godt kjent med *virus*, men dette er første gangen vi ser en *orm* spre seg på internett. En orm er altså et virus som sprer seg fra system til system uten brukerinteraksjon.

Ormer som sprer seg på denne måten utnytter en type sårbarhet som kalles *RCE* (Remote Code Execution). Morris-ormen utnytter flere slike, bl.a. en *stack overflow* i nettverkstjenesten *fingerd* (en tjeneste som forteller hvem som er logget på hvilken terminal).

### Buffer overflow

> Du kan selv utforske *stack overflows* i `~/1_grunnleggende/4_overflow/`.

*Stack overflow* er en spesiell type *buffer overflow*. Et *buffer* er bare et annet navn på plass avsatt i et programs arbeidsminne. *Overflow* betyr ganske enkelt at bufferet ved en feil fylles utover sin kapasitet. Det finnes flere typer *overflows*, som f.eks *heap overflow*, men vi skal nå konsentrere oss om stack. Er man riktig heldig gir nemlig en *stack overflow* dobbel gevinst; hvor man både kan fylle bufferet med hva man vil **og** overskrive data som *endrer programflyten*. Dette kalles en *stack smash*.

Å prøve å forstå hvordan en *stack smash* virker kan gjøre enhver svimmel, men når man for første gang innser hvordan puslespillbrikkene passer sammen går svimmelheten ganske fort over i gåsehud! To ting man må forstå er hvordan en CPU utfører instruksjoner og hvordan en stack virker:

1. En CPU utfører blindt enkle instruksjoner, eller *maskinkode*. Instruksjoner grupperes i *funksjoner*, og programmer bygges opp av funksjoner som kaller

funksjoner som kaller funksjoner. Å *kalle en funksjon* betyr at CPU-en hopper fra funksjonen den holder på med til en annen. For å komme tilbake til den første, må den ha notert seg hva den holdt på med før den hoppet. Den noterer da *adressen* til instruksjonen den skal utføre *etter den kommer tilbake*. *Adresse* (eller *peker*) er nummereringen av plassene i arbeidsminnet til et program. Hvilken instruksjon som utføres bestemmes av CPU-ens *instruksjonspeker*.

2. En stack er en datastruktur hvor man kan *putte et element på toppen* (push) eller *ta et element av toppen* (pop). CPU-en bruker bl.a. stacken når den trenger å notere hva den holdt på med før funksjonskall. Av praktiske årsaker "vokser stacken nedover", altså at neste ledige plass på stacken har lavere adresse etter en *push*.

Her er et minimalt program skrevet i *assembly*:

```
    [section .text]
    [bits 64]       ; dette er et 64-bit program.
                    ; pekere er 64 bits = 8 bytes ≈ 8 bokstaver.

    extern gets     ; vi trenger funksjonen 'gets' fra systembiblioteket.
    global _start

_start:             ; programmet starter i funksjonen '_start'
    call main       ; hopp til funksjonen 'main':
                    ; for å huske at programmet senere skal fortsette på
                    ; neste linje, er instruksjonen 'call' i praksis en
                    ; kombinasjon av instruksjonene:
                    ; - push rip ; lagre instruksjonspekeren på stacken
                    ; - jmp main ; hopp til main
    call exit       ; samme igjen:
                    ; - push rip
                    ; - jmp exit

main:               ; funksjonen 'main'
    sub rsp, 32     ; reserver et buffer på 32 bytes på stacken.
                    ; rsp er stack-pekeren.
                    ; husk: den vokser nedover, derfor sub rsp (subtract).
    mov rdi, rsp    ; ta vare på stack-pekeren i et register.
    call gets       ; be operativsystemet lese inn en tekst fra brukeren
                    ; og putte teksten i bufferet rdi peker på.
                    ; rdi er destination index.
                    ; legg merke til at vi ikke forteller 'gets'-funksjonen
                    ; hvor stort bufferet er! hvis brukeren taster mer enn
                    ; 32 tegn så skrives det videre utenfor bufferet!
                    ; spørsmål: hva overskrives i så fall?

    mov rdi, rax    ; ta vare på returverdien fra 'gets' i rdi.
    add rsp, 32     ; gi slipp på buffer-reservasjonen.
```

```
    ret             ; instruksjonen 'ret' returnerer CPU-en tilbake til det
                    ; den jobbet med før den kalte 'main'. det gjør den ved
                    ; å lese instruksjonspekeren av stacken. altså i praksis:
                    ; - pop rip

exit:               ; funksjonen 'exit'
    mov eax, 60     ; systemkall-nummeret i linux for å avslutte et program.
    syscall         ; utfør systemkall.
                    ; programmet avsluttes av operativsystemet.
```

Klarer du å følge flyten i programmet?

Hva blir overskrevet dersom brukeren taster mer enn 32 tegn? For eksempel 32 B-er og 8 A-er: "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAA"

Stacken vokser nedover, og det siste som ble pushet på stacken før buffer-allokeringen er... *retur-adressen* – altså der `main` skulle returnere tilbake til `_start`.

| 32 byte buffer | rip |
|---|---|
| BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB | AAAAAAAA |

Ved å utføre instruksjonen `ret` (som er det samme som `pop rip`), vil CPU-en nå hoppe til *adressen AAAAAAAA*. Den adressen er ugyldig, men se for deg at du i stedet for *AAAAAAAA* kan putte inn en *peker til bufferet* i returadressen, og så fylle bufferet med instruksjoner **du** ønsker å utføre (ofte kalt *shellcode*). Elegant, er det ikke?

Dessverre finner man veldig sjelden sårbarheter som er så enkle å utnytte (men det hender fremdeles, les om *Cable Haunt* fra 2020). Både leverandører av CPU-er, operativsystemer, biblioteker og kompilatorer har bidratt til å stenge denne typen hull.

CPU-er kan nå nekte å utføre instruksjoner som er lagret på stacken. Denne beskyttelsen har mange navn, men kalles oftest *NX*, for *No Execute*, eller i Windows *DEP* (*Data Execution Prevention*). At man ikke kan putte instruksjoner direkte i stack-bufferet begrenser mulighetene noe.

Men man kan komme rundt det ved å returnere til *eksisterende og kjent kode* som har instruksjoner som allerede gjør det man ønsker å oppnå. Dette ble opprinnelig kalt *ret2libc* fordi man returnerte til en av systembibliotekets (*libc*) funksjoner for starte programmer. Denne teknikken har senere blitt videreutviklet til *ROP* (*return-oriented programming*), der man kan sette sammen mange små deler av *eksisterende kode*, og så returnere fra den ene til den neste. Hver del, eller *gadget*, består da av én del som utfører instruksjonen man ønsker, og én del som returnerer til neste *gadget*.

Operativsystemer prøver også å beskytte mot dette ved å plassere programmer på tilfeldige adresser hver gang de startes. Dette kalles *ASLR* (*Address Space Layout Randomization*). Sammen med leverandører av biblioteker og kompilatorer er det også laget beskyttelsesmekanismer som *stack canary* (en snubletråd som er plassert mellom buffer og returadresse på stacken) og mange teknikker for å hindre tukling med

programflyten, såkalt *Control-flow integrity*. I tillegg frarådes bruk av usikre funksjoner som `gets()`.

Det var altså en slik sårbarhet *Morris-ormen* utnyttet – *stack overflow* etter usikker bruk av `gets()`. I 1988 var det ingen *NX*, ingen *stack canary*, ingen *ASLR* og ingen *CFI*, så utnyttelse av sårbarheten var ikke ulik eksempelet over.

Selv om teknikken var kjent allerede i 1972, var det først på midten av 90-tallet den fikk sitt store gjennombrudd. I 1995 skriver Thomas Lopatic om den på *bugtraq*. Året etter kommer den legendariske artikkelen *Smashing the Stack for Fun and Profit* i *Phrack magazine* #49, hvor Elias Levy (alias: *Aleph One*) gir oss en detaljert trinn for trinn gjennomgang av teknikken.

I årene som følger blir også teknikker for utnyttelse av heap overflows utviklet. Heap er *dynamisk allokert* minne. Altså minne man kan be opertivsystemet om å reservere, og man får en peker til bufferet tilbake. Ved overflyt av buffer på heap skriver man ikke over returadresse, men kan bl.a skrive over metadata som heap-systemet trenger for å holde orden på heapen, samt skrive over objekter som ligger etter ditt eget buffer. Dette kan også brukes til å påvirke eller ta kontroll over programflyten. En relatert type bugs kalles for *dangling pointer* eller *use-after-free*, hvor minne frigjøres, men pekeren brukes videre. Minnebufferet blir nemlig ikke automatisk tømt og utilgjengeliggjort; feilen oppdages først når programmet gjenbruker heap-minnet.

Eller som *ChatGPT* beskriver det:

> A use-after-free vulnerability is a type of computer security vulnerability that occurs when a program tries to access memory that has previously been freed. This can happen if the program has a pointer to memory that has been freed, but the program does not realize that the memory has been freed. When the program tries to access this memory, it can cause the program to crash or allow an attacker to execute arbitrary code on the system. This type of vulnerability is particularly dangerous because it can be difficult to detect.

## 1993: Command injection over Web

På 90-tallet er datamaskinen allemannseie og stadig flere husstander er tilkoblet internett. Hackerkulturen opplever eventyrlig vekst, og detaljerte beskrivelser av sårbarheter florerer åpent på *bugtraq* og *full-disclosure*-listene. Teknikker for å utnytte disse utvikles på løpende bånd og deles i tekstfiler samt magasiner, som Phrack.

Allerede fire år etter Tim Berners-Lee finner opp *World Wide Web*, og lenge før javascript ser dagens lys, dukker behovet for dynamiske nettsider opp. Dynamisk nettside betyr at et program på serveren lager en nettside basert på brukerens forespørsel, i stedet for å sende statiske HTML-sider. Grensesnittet får navnet *CGI – Common Gateway Interface*.

Typiske eksempler på dynamiske nettsteder er søkemotorer, aviser, og portaler. På serveren brukes ofte *LAMP*: Linux, Apache, MySQL, Perl/PHP/Python. På Windows bruker man *IIS* (*Internet Information Services*) og Microsofts egen *SQL Server*.

Hackere oppdager fort at man kan få CGI-programmer til å gjøre mye mer enn tiltenkt ved å snike inn kommandoer som en del av eksempelvis søkeord til søkemotorer. Hva skjer dersom søkescriptet mottar `sokeord` fra brukere, og sender det til denne kommandoen?

```
find /database/ -type f | \
  xargs grep ${sokeord}`
```

... og søkeordet er `tulleord ; cat /etc/passwd` ?

> Du kan eksperimentere med *command injections* i `~/1_grunnleggende/3_injection/`.

## 1998: SQL injection

Julen 1998 tar Jeff Forristal (alias: *rain.forrest.puppy*) *injection*-teknikker til neste nivå med artikkelen *NT Web Technology Vulnerabilities* i Phrack #54. Denne artikkelen var en glimrende innføring i grunnleggende *SQL injections* og er like relevant den dag i dag.

SQL er et digert spørrespråk som brukes til å snakke med databaser for å lese og skrive data. Kommunikasjonen foregår ved at man skriver en *spørring* som man programmatisk sender til databasen. Databasen tolker spørringen og bestemmer seg for hvordan den skal utføres.

En *spørring* kan se slik ut:

```
SELECT username, password FROM users;
```

Nå vil databasen sende tilbake alle brukernavn og passord i tabellen `users`. Se for deg at en bruker skal logge inn på et nettsted; brukernavn og passord er tastet inn, og *logg inn*-knappen er trykket. Serveren må nå verifisere at kombinasjonen av brukernavn og passord er gyldig. Man kan for eksempel gjøre noe som dette:

```
$u = $_GET["username"];
$p = $_GET["password"];
$query = "SELECT username, password
            FROM users
            WHERE username='$u'
              AND password='$p'";
```

Gitt at *u* er `AzureDiamond` og *p* er `hunter2` blir spørringen:

```
SELECT username, password
    FROM users
    WHERE username='AzureDiamond'
      AND password='hunter2'
```

Dersom databasespørringen lykkes tolker serveren det som at kombinasjonen av brukernavn og passord er korrekt, og brukeren kan logges inn. Hva skjer dersom *p* er `' OR 1=1; -- -`? Det gir oss følgende spørring:

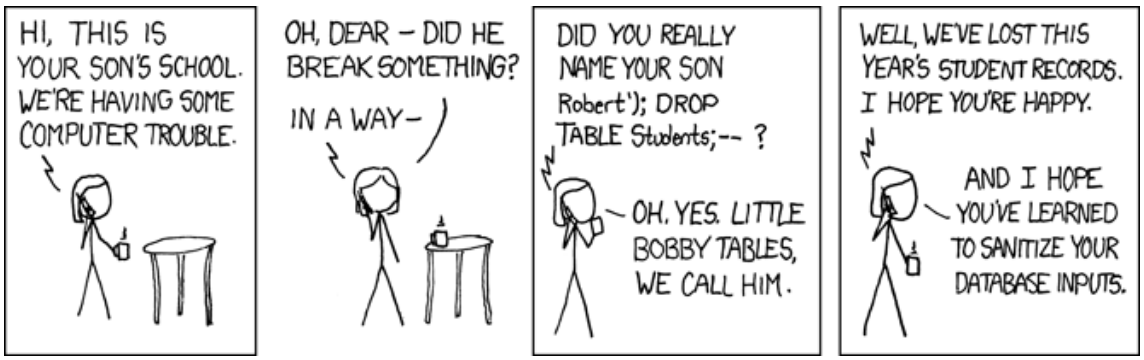
```
SELECT username, password
    FROM users
```

```
WHERE username='AzureDiamond'
  AND password='' OR 1=1; -- -'
```

Tallet 1 vil alltid være lik seg selv, så det spiller ingen rolle om passordet er riktig her. Brukeren logges rett inn som en vilkårlig valgt bruker, gjerne `Administrator` eller `root`.

SQL- og andre injections har i mange år toppet OWASP sin topp 10 liste over sårbarheter, og denne sårbarheten er nesten like aktuell i dag som den var i 1998.

> "1. Injection. Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization."




Figur 3: XKCD 327

I dag er det veldig mange slike injection-teknikker. Grovt sett kategoriseres disse som *insecure deserialization*. Et moderne eksempel på denne type bugs er *log4shell*, som involverer deserialisering av java-objekter.

## 2022: Etterretningstjenestens talentprogram cyberoperasjoner

Det har skjedd utrolig mye spennende innenfor offensive teknikker de siste tiårene, og det var på 80- og 90-tallet fundamentet ble lagt. De klassiske teknikkene har en helt egen sjarm!

La oss hoppe i tid til i dag. Hvordan er det å jobbe med offensive teknikker i 2022?

> "Er du kreativ og dyktig, finner du fremdeles sårbarheter. Det er uten tvil blitt betydelig vanskeligere, men sikkerhetshullene er der og kan utnyttes. Kanskje trenger man ikke én, men å utnytte en håndfull sårbarheter, finurlig satt sammen for å komme forbi alle sikkerhetsmekanismene."

Men offensive teknikker består av veldig mye mer enn å finne og utnytte sårbarheter. I Etterretningstjenesten jobber vi med programmering, elektronikk, kryptografi, infrastruktur, matematikk, nettverk og operativsystemer. Vi er kreative problemløsere.

Vi håper du synes dette er spennende og at du ønsker å lære mer moderne teknikker. Dette er noe av det du vil lære i vårt talentprogram cyberoperasjoner. Les mer på

`https://www.etterretningstjenesten.no/cybertalent`. Kanskje du en dag kan bruke dine ferdigheter til innhenting av informasjon til forsvar av Norge!

## Referanser

- Antivirus software
- Firewall
- Computer worm
- Titan Rain
- Rootkit
- Intrusion detection system
- Computer security incident management
- Penetration test
- Computer emergency response team
- Cyber threat intelligence
- APT, Advanced persistent threat
- Password manager
- Multi-factor authentication
- Ransomware
- Stuxnet
- Shamoon
- BlackEnergy
- WannaCry ransomware attack
- Petya
- GNU
- Wargames
- 414s
- Edb-sikkerhet, industriens sikkerhetsutvalg
- Reflections on Trusting Trust ¤
- Password strength
- Key derivation function
- Hackers - datasnokere
- Treholt: en avleggs spion
- Clifford Stoll
- Cuckoo's egg
- Tymnet
- Emacs
- Setuid
- Sudo
- Dirty COW
- Time-of-check to time-of-use
- Cron
- Morris worm
- Robert T. Morris
- Pål Spilling
- Remote code execution

- Stack buffer overflow
- Smashing The Stack For Fun And Profit
- Finger
- Instruksjonspeker
- Stack canaries
- Assembly language
- Shellcode
- Cable Haunt
- NX bit
- DEP
- Return-to-libc attack
- Return-oriented programming
- Address space layout randomization
- Control-flow integrity
- Bugtraq
- Aleph One / Elias Levy
- Heap overflow
- Dangling pointer
- ChatGPT
- CGI
- SQL
- SQL injection
- OWASP
- OWASP top 10
- log4shell