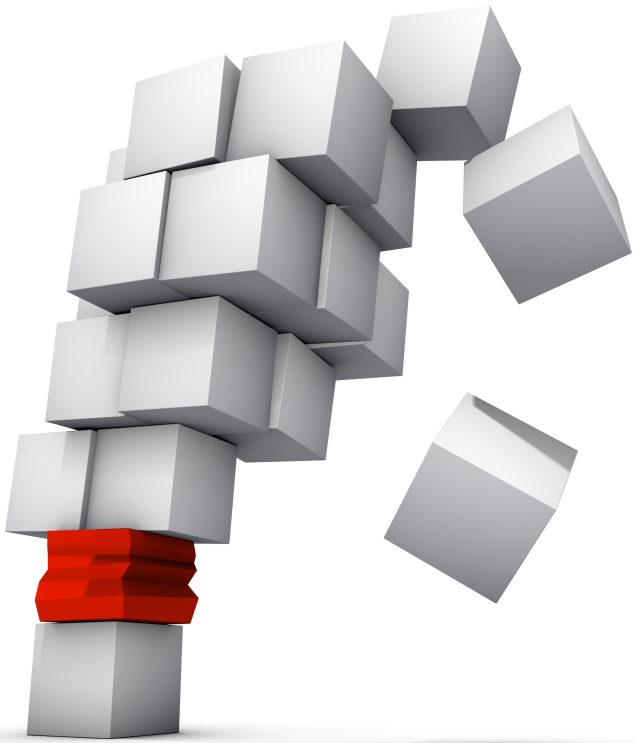
Varför har mjukvaror sårbarheter?

HENRIK KARLZÉN, DANIEL EIDENSKOG,
JERRY FALKCRONA, CHRISTIAN VALASSI



FOI-R--5550--SE
ISSN 1650-1942 December 2023

Henrik Karlzén, Daniel Eidenskog, Jerry Falkcrona,
Christian Valassi

Varför har mjukvaror sårbarheter?

Titel	Varför har mjukvaror sårbarheter?
Title	Why does software have vulnerabilities?
Rapportnr/Report no	FOI-R--5550--SE
Månad/Month	December
Utgivningsår/Year	2023
Antal sidor/Pages	44
ISSN	1650-1942
Uppdragsgivare/Client	Försvarsmakten
Forskningsområde	Informationssäkerhet
FoT-område	Operationer i cyberdomänen
Projektnr/Project no	E38509
Godkänd av/Approved by	Emil Hjalmarson
Ansvarig avdelning	Cyberförsvar och ledningsteknik

Bild/Cover: Shutterstock

Sammanfattning

Trots årtionden av forskning och utveckling inom mjukvarusäkerhet fortsätter sårbarheter att förekomma i stor mängd. Denna förstudie utgör en sammanställning av forskning och annan litteratur som undersöker olika typer av orsaker till varför sårbarheter fortfarande är så vanliga. Syftet med studien är att ge stöd i att förstå faktorerna bakom uppkomsten av sårbarheter och därigenom kunna undvika att de uppstår.

I princip förekommer mjukvarusårbarheter i all mjukvara och det finns många olika typer av sårbarheter. De bakomliggande orsakerna till att mjukvarusårbarheter uppstår är många och bland dessa finns organisatoriska faktorer, osäkra programmeringsspråk, svåranvända eller otillräckliga verktyg, brister i utvecklingsmetoder, avsaknad av motivation hos utvecklare och bristande säkerhetskompetens. Utmaningen med att förhindra att mjukvarusårbarheter uppstår är således inte enbart en teknisk sådan; även den mänskliga faktorn behöver beaktas.

Trots att det finns en ansenlig mängd befintlig forskning som identifierar potentiella orsaker till att mjukvarusårbarheter uppstår fortsätter nya sårbarheter av samma typ att rapporteras. Detta tyder på att det återstår mycket forskning inom sårbarheternas orsaker och uppkomst. Denna studie identifierar ett antal förslag på vidare forskningsområden.

Nyckelord:

Mjukvara, mjukvaruutveckling, mjukvarusäkerhet, cybersäkerhet, sårbarheter

Summary

Despite decades of research and development in software security, vulnerabilities continue to emerge in large quantities. This prestudy provides an overview of research and other literature that investigates the reasons behind the occurrence of vulnerabilities in software. The aim of the study is to support understanding of the factors behind the occurrence of vulnerabilities and this aiding in avoiding them.

Vulnerabilities occur in practically all software and there are many diverse types of vulnerabilities. The causes of the vulnerabilities are many and includes, among others, organizational factors, insecure programming languages, tools that are hard to use or inadequate, flawed development methods, lack of motivation among developers, and lack of security knowledge. The challenge in preventing software vulnerabilities is thus not only technical; it is also important to account for the human factor.

Despite a significant amount of research already performed to identify the potential factors behind the occurrence of software vulnerabilities, the same types of vulnerabilities are still being reported. This indicates that there is much research related to causes for vulnerabilities still to be done. This report identifies a number of suggestions for further research.

Keywords:

Software, software development, software security, cyber security, vulnerabilities

Innehållsförteckning

1 Inledning

I februari 2023 gav Jen Easterly, direktör för USA:s cyberförsvarsmyndighet CISA[1], en mörk bild av cybersäkerhetsläget i ett tal:

> *Vi har normaliserat det faktum att tekniska produkter släpps ut på marknaden med dussintals, hundratals eller tusentals defekter, trots att en så dålig konstruktion skulle vara oacceptabel inom alla andra kritiska områden.*
>
> *Vi har normaliserat det faktum att cybersäkerhetsbördan placeras oproportionerligt på konsumenters och små organisationers axlar, hos de som oftast är minst medvetna om hoten och minst kapabla att skydda sig själva.*
>
> *Det är som om vi normaliserat det avvikande beteendet att verka vid den yttersta framkanten av olycksgränsen. […] Som tur är bevisar historien att vi kan – och även måste – ändra sättet som vi kollektivt värdesätter säkerhet över andra marknadsincitament såsom kostnad, funktioner och ledtid. […] (Easterly, 2023)*

Utöver de generella uttalandena ovan tog Easterly även upp mer specifika aspekter av teknikval och utbildningar, bland annat följande exempel:

> *C och C++ ska, när de lärs ut, behandlas som farliga […]*
>
> *Bara ett av de tjugo bästa kandidatprogrammen för datavetenskap kräver en säkerhetskurs som examineringskrav.*

Easterlys synsätt är inte nytt, vilket framgår av uppmaningen i den nästan tjugo år gamla rapporten till DHS av Davis m.fl. (2004):

> *Använd ett programmeringsspråk med betydligt färre möjligheter för misstag än C eller C++ när möjligt.*
>
> *Uppmuntra och finansiera universitet som lär ut datavetenskap och närliggande ämnen för att erbjuda kurser och utföra forskning om utvecklingsprocesser för säkra mjukvaror.*

Av citaten framgår en bild av osäkra produkter med osäkra mjukvaror, framtagna i osäkra programmeringsspråk av utvecklare som saknar säkerhetskunskap. Citaten nämner C och C++ specifikt, då de är programmeringsspråk där det är lätt att begå programmeringsmisstag, men det finns mjukvarusårbarheter i många olika programmeringsspråk. Därtill är mjukvaror väldigt vanliga och hotbilden

[1] CISA är förkortningen för Cybersecurity and Infrastructure Security Agency vilket är en del av USA:s Department of Homeland Security (DHS). Citatet är fritt översatt från ett tal hållet hos universitetet Carnegie Mellon.

stor vilket innebär att bristande säkerhet kan få stora konsekvenser. Den bristande säkerheten kan ses i form av sårbarheter i mjukvaran och det finns många välkända och allvarliga exempel på mjukvarusårbarheter och deras utnyttjande, däribland Log4Shell i Log4j.

Log4Shell är en sårbarhet i Java-loggramverket Log4j som vid sårbarhetens upptäckt fanns i nästan alla Javabaserade produkter (Check Point, 2021). Något förenklat kunde sårbarheten utnyttjas av en angripare för att tillåta fjärrkörning av kod (Graham-Cunning, 2021). Enligt Check Point (2021) fanns det inom ett dygn efter att sårbarheten blev allmänt känd över 60 varianter av angrepp som utnyttjade sårbarheten. Inom 72 timmar hade det skett nästan en miljon angreppsförsök mot nätverk över hela världen och efter en månad hade nästan hälften av världens alla företagsnätverk utsatts för försök till utnyttjande av sårbarheten.

Log4Shell illustrerar vilka katastrofala konsekvenser mjukvarusårbarheter kan medföra. Eftersom mjukvarusårbarheter inte är nya företeelser finns det redan många försök till att motverka deras uppkomst. Bland annat finns säkerhetsfokuserade utvecklingsmetoder och programmeringsspråk, testverktyg för att hitta och åtgärda sårbarheter samt säkerhetsfunktioner som ska skydda systemet mot konsekvenserna av att kvarstående sårbarheter utnyttjas. Trots dessa försök att undvika sårbarheter är mjukvara fortsatt osäker. Denna rapport beskriver ett antal orsaker till att det fortfarande är så.

1.1 Syfte och mål

Rapporten sammanställer befintlig forskning om varför mjukvaror har sårbarheter. Sammanställningen syftar till att ge stöd i att undvika sårbarheter i mjukvara. Källorna är i huvudsak dokument producerade av forskare och andra auktoriteter, inklusive tongivande myndigheter. Rapporten utgör en förstudie och det lämnas inga garantier om att rapporten visar en komplett bild av området. Rapporten undersöker brett mjukvarusårbarhetsområdet och har tagits fram i projektet *Analys av koncept och teknik för cyberförsvar och informationssäkerhet (AKTCI)* inom ramen för Försvarsmaktens samlingsbeställning inom forskning och teknikutveckling (FoT). I projektet görs analyser av olika cybersäkerhetsproblem för att bland annat inrikta framtida forskningssatsningar.

De mjukvarusårbarheter som anses intressanta i rapporten är främst sådana som på något sätt rör Försvarsmakten, men de är i de flesta fall även relevanta för andra större organisationer. Källorna som används i rapporten omfattar däremot både stora och små projekt, där de minsta i vissa fall drivs av enskilda personer.

Rapporten svarar på följande frågor:

1. *Vad är mjukvarusårbarheter?*
2. *Vilka sårbarheter har mjukvaror?*

3. *Varför har mjukvaror sårbarheter?*
4. *Vilken forskning behöver utföras om mjukvarusårbarheter?*

1.2 Avgränsning

Denna rapport beskriver mjukvarusårbarheter och hur dessa kan undvikas. De avgränsningar som gjorts i studien framgår av tabell 1. Däremot görs ingen avgränsning av vilka typer av mjukvaror som inkluderas, vare sig det är sådana som ingår i mikrotjänster, webbtjänster, molntjänster, databassystem, kontorssystem, fristående datorer, cyberfysiska system eller AI-system.

Tabell 1: Vad som är inkluderat och exkluderat i rapportens fokus samt källor med mer information om det exkluderade ämnet.

Inkluderat	**Exkluderat**	**Källor med mer information om det exkluderade**
Skanning av mjukvaror för att upptäcka nya sårbarheter.	Skanning av system för att upptäcka känt sårbara mjukvaror.	Holm m.fl. (2011)
Avlägsnande av sårbarheter.	Döljandet av sårbarheter genom rörlig minnesallokering eller kodobfuskering.	Holm m.fl. (2014) samt Hosseinzadeh m.fl. (2018)

Det bör noteras att den största delen av forskningen om problem med mjukvaror rör buggar och defekter i allmänhet, snarare än specifikt sårbarheter. Det har dock visats att sårbarheter är väsensskilda från andra buggar, exempelvis vad gäller hur sårbarheter ska undvikas eller åtgärdas (Clark m.fl., 2010 samt Camilo m.fl., 2015). Denna rapport är därför avgränsad till till domänen mjukvarusårbarheter och drar inte några slutsatser från den mer allmänna domänen mjukvarubuggar.

1.3 Läsanvisning

Rapporten riktar sig huvudsakligen till de tre målgrupperna utvecklare, utvecklare av just säkerhetsmjukvara samt forskare. Läsanvisningen är indelad per målgrupp:

- *Utvecklare*, som vill utveckla säkra mjukvaror. Här ryms inte bara programmerare, utan även sårbarhetstestare, arkitekter och projektledare. För denna målgrupp är det främst bakgrundskapitlen 3–4 och resultaten i

kapitel 5 som är av intresse. I det senare kapitlet är det framför allt de sammanfattande delarna som är relevanta.

- *Utvecklare av säkerhetsmjukvara*, som vill veta vilka nya säkerhetsmekanismer som säkerhetsmjukvarorna behöver för att möta sårbarheter. För denna målgrupp är detaljer om brister i kapitel 5 mest relevanta att läsa.
- *Forskare*, som vill veta vad de behöver inrikta sin forskning mot för att undvika sårbarheter. För denna målgrupp är kapitel 2 om rapportens metod relevant att läsa, liksom både sammanfattningar och detaljer om resultaten i kapitel 5 samt framtida forskningsförslag i kapitel 6.

Därtill kan slutanvändare ha viss nytta av rapporten genom att få en första insikt i vad de säkerhetsmässigt kan förvänta sig av mjukvaran de använder. I samtliga fall gäller att läsarna bör ha ett stort säkerhetsintresse. Dessutom kräver många resonemang en generell förståelse för mjukvaruutveckling och säkerhet, även om en del förhållanden klargörs i kapitel 3.

2 Metod

Studien bygger på en semistrukturerad litteraturstudie huvudsakligen baserad på forskningslitteratur och auktoritativa rapporter från myndigheter och liknande instanser. Detta kompletteras med material från intervjuer med erfarna mjukvaruutvecklare på FOI för att undersöka litteraturens relevans. Det här kapitlet beskriverdetaljerna kring hur data samlats in och sammanställts.

2.1 Söksträngar och källor

Det har inte varit rapportens ambition att vara fullständigt uttömmande när det gäller relevanta källor. Däremot har avsikten varit att täcka in de allra flesta relevanta källorna. För att identifiera källorna användes söksträngar uppbyggda av söktermer som var kända sedan tidigare, exempelvis ”mjukvara”, ”säkerhet” och ”sårbarheter” samt mer specifika söktermer som ”best practice”, ”testning”, ”användbarhet” och ”komplexitet”. Därtill lades fler söktermer till utifrån de översiktsartiklar som hittades i preliminära sökningar samt utifrån boken *Software Security: Building Security In* (McGraw, 2006). Till viss del användes också söktermer som hade militär koppling, t.ex. ”military” eller ”defen*e”. Söktermerna översattes till engelska och lades ihop till söksträngar varpå sökningar i olika databaser gjordes. Därtill gjordes för forskningsartiklar kompletterande sökningar (snöbollsurval) efter forskningsartiklar som citerade redan identifierade forskningsartiklar eller som citerades av desamma.

Den databas som framförallt användes var Google Scholar, vilken sammanställer framförallt mycket stora mängder forskningsartiklar, akademiska avhandlingar, tekniska rapporter och patent. Vid sökningarna i Google Scholar var fokus på vad som där klassas som översiktsartiklar samt i övrigt de mest citerade forskningsartiklarna från (och framförallt då från perioden 2021–2023). Några översiktliga sökningar gjordes också i databaser för de tidskrifter och konferenser som identifierats som mest framstående på forskningsområdet enligt rankningar av Scopus och liknande databaser.[2] Anledningen till detta vara att öka sannolikheten till att hitta resultat av hög kvalitet. Sökningarna i dessa mindre databaser gjordes med de enkla söksträngarna ”security OR vulnerability” i publikationer rörande mjukvara samt med ”software” och dylikt i publikationer som fokuserade på säkerhet.

[2] Tidskrifterna och konferenserna var bland annat IEEE Transactions on Dependable and Secure Computing, IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), IEEE Security & Privacy, Computers & Security, Computer and Communications Security, IEEE Symposium on Security and Privacy, International Conference on Information Security Practice and Experience.

Sökningar gjordes även efter andra typer av dokument än forskningsartiklar. Detta gällde framförallt uttalanden från auktoriteter inom området, t.ex. best practice från tongivande myndigheter som NIST och CISA, men även befintliga rapporter och memon utgivna av FOI. När befintliga FOI-dokument fanns inom ett delområde gjordes enbart minimala sökningar, så länge FOI-dokumenten inte var mer än tio år gamla. Det är tänkbart att det även finns andra mindre formella källor till relevant information om mjukvarusårbarheter. Exempelvis förekommer mycket diskussion om sårbarheter i olika webbforum (såsom Stack Overflow). Sådana forum har inte gåtts igenom i denna rapport men däremot ingår flera forskningsartiklar som studerat sådana forum.

Utöver detta gjordes också sökningar efter sammanställningar av sårbarheter i form av taxonomier och databaser för kapitel 4 om vilka sårbarheter mjukvaror har. Parallellt med litteratursökningarna gjordes också ett fåtal intervjuer av erfarna mjukvaruutvecklare verksamma vid FOI. Syftet med intervjuerna var att kontinuerligt utvärdera om det insamlade materialet var relevant utifrån intervjupersonernas åsikter.

2.2 Extrahering och sammanställning

Identifierad litteratur lästes igenom och viktiga utsagor extraherades, med fokus på artiklars sammanfattningar, slutsatser, tabeller och figurer. I vissa fall extraherades även förslag på framtida forskning ur artiklarna. Eftersom denna rapport utgjorde en förstudie var ambitionen att enbart återge de grova dragen från den viktigaste litteraturen inom varje delområde.

Det extraherade materialet sammanställdes genom att materialets huvuddrag kortfattat beskrevs, utan onödiga tekniska detaljer. För de delområden det redan fanns FOI-dokument gjordes bara kortare sammanställningar med hänvisning till ytterligare läsning.

3 Vad är en mjukvarusårbarhet?

Mjukvarusårbarheter kan definieras på flera olika sätt. Ett exempel ges av amerikanska NIST: *”En säkerhetsbrist, fel eller svaghet i programvarukod som kan utnyttjas av en angripare”* (Quirolgico m.fl., 2015).[3] Det här kapitlet beskriver detaljer kring vad som krävs av en angripare för att utnyttja en mjukvarusårbarhet och vad en angripare har möjlighet att göra när en sårbarhet har utnyttjats.

3.1 Vad krävs för att utnyttja sårbarheter?

Det finns olika kompetensnivåer hos angripare, vilka inkluderar hela spannet från uttråkade tonåringar som läst en hackerguide eller laddat ned ett angreppsverktyg från internet till statsunderstödda Advanced Persistent Threat (APT)-grupper med många år av praktisk erfarenhet och en stor budget. För mjuksårbarheter som redan publicerats finns i regel redan uppdateringar tillgängliga som tar bort sårbarheten. Det betyder dock inte att alla sårbara system har uppdaterats, vilket innebär att sårbarheten i vissa fall fortfarande kan utnyttjas. Sådana gamla sårbarheter finns det ofta färdig attackkod för vilket gör att kompetenskraven för en angripare inte är så höga. För mjukvarusårbarheter som inte är allmänt kända krävs högre kompetens för att en angripare ska kunna utnyttja dem framgångsrikt. Därtill krävs ofta tid för att utveckla och testa relevant attackkod.

Generellt behöver en angripare gå igenom flera steg för att utnyttja en sårbarhet: spaning (där angriparen identifierar vilken mjukvara målsystemet kör och vilka sårbarheter mjukvaran har), resursutveckling (där angriparen utvecklar eller införskaffar skadlig kod som utnyttjar sårbarheten), exekvering av skadlig kod (där angriparen kör den skadliga koden på målsystemet). Vart och ett av dessa steg medför svårigheter för angriparen. Exempelvis är det ofta svårt att ta fram en attackkod som fungerar på målsystemet. Ibland finns det visserligen redan färdiga skadliga koder att använda, men de fungerar inte nödvändigtvis på just den mjukvaruversion och systemkonfiguration som angriparen vill angripa. Mer om svårigheten att gå från sårbarhet till användbar skadlig kod kan läsas i FOI-rapporterna Sommestad och Holm (2017) samt Löfvenberg m.fl. (2019).

[3] Översatt från: *“A security flaw, glitch or weakness found in software code that could be exploited by an attacker”*

3.2 Vad kan en angripare göra när sårbarheterna utnyttjats?

När en angripare kör sin attackkod kan angriparen få mjukvaran att göra saker som inte förväntas eller som angriparen inte borde kunna göra. Exempelvis kan det röra sig om att få indata att tolkas och behandlas som instruktioner i ett buffertöverskridningsangrepp (eng. buffer overflow attack). Sådant är dock typiskt bara ett delmål för angriparen. För att nå slutmålet behöver angriparen ofta skaffa sig fler privilegier på en dator eller angripa datorer i andra delar av nätverket datorn är ansluten till. Även då kan mjukvarusårbarheter utnyttjas. Själva slutmålet i cyberdomänen kan sedan vara att exempelvis manipulera, stjäla eller förstöra information, eller stänga av systemet. Dessa effekter kan i sin tur få stor påverkan på en organisations förmåga att exempelvis hemlighålla affärshemligheter, att agera utifrån oförvanskad information och att upprätthålla kritiska industriella processer.

4 Vilka sårbarheter har mjukvaror?

Detta kapitel beskriver vilka sårbarheter mjukvaror har. Detta görs genom att beskriva kategoriseringar av sårbarheter, vilka sårbarheter som är kända samt vilka sårbarheter som nyttjats i kända angrepp.

4.1 Vilka kategoriseringar av sårbarheter finns?

Det finns många försök till att kategorisera sårbarheter i taxonomier. Här beskrivs de taxonomier och andra kategoriseringar av sårbarheter som är vanligast i forskningsartiklar och i andra publikationer om sårbarheter. Det bör dock nämnas att taxonomierna inte nödvändigtvis enbart tar upp den typ av sårbarheter som denna rapport fokuserar på.

En sammanställning av taxonomier för sårbarheter gjordes av Krsul (1998). Det finns till exempel taxonomier om vilken typ av mjukvara en sårbarhet finns i, vad som krävs för att utnyttja sårbarheten, svårighetsgrad för att utnyttja sårbarheten samt vad utnyttjandet kan ge för effekter. Joshi m.fl. (2015) sammanställde taxonomier och då främst sådana som föreslagits av forskare. Det finns också taxonomier över hur sårbarheterna uppkommit. Exempelvis beskriver Kuhn m.fl (2017) administrativa och konfigurationsmässiga fel, fundamentala designrelaterade fel samt vanliga programmerings- eller implementationsrelaterade fel.

Vägledningen ISO/IEC 24772-1:2019 beskriver olika typer av programmeringsspråksnära sårbarheter för att underlätta för programmerare att undvika dessa. Ett annat sätt att kategorisera sårbarheter är Common Weakness Enumeration (CWE) (Bojanova m.fl., 2016). Det finns hundratals olika kategorier i CWE där varje kategori utgör en typ av sårbarhet i mjukvara eller hårdvara. Bland kategorierna finns exempelvis olika typer av buffertöverskridningar och webbkodinjektion (eng. cross-site scripting). Närbesläktad med CWE är Common Vulnerabilities and Exposures (CVE) som kategoriserar mer specifika sårbarheter där varje konkret sårbarhet i en viss mjukvara får ett eget CVE-nummer. Det finns också ett tidigt försök att koppla ihop å ena sidan CWE och CVE samt å andra sidan (vissa) ATT&CK-tekniker.[4] CVE:er finns bland annat sökbara i amerikanska NIST:s nationella sårbarhetsdatabas NVD[5], tillsammans med en rankning av allvarlighetsgraden för respektive CVE genom

[4] Se https://github.com/center-for-threat-informed-defense/attack_to_cve/blob/master/methodology.md. MITRE:s ATT&CK-ramverk är en kunskapsbas för antagonistiska taktiker och tekniker. Se https://attack.mitre.org för mer information.

[5] Se https://nvd.nist.gov för mer information.

betygsystemet Common Vulnerability Scoring System (CVSS)[6]. NVD innehåller även en Common Platform Enumeration (CPE) för respektive CVE som beskriver mer detaljerat vilken mjukvara som innehåller sårbarheten. Exempel på aspekter som tas upp är vilken applikation det gäller, vilket operativsystem och eventuell hårdvara som den körs på och vilket versionsnummer som berörs.

4.2 Vilka sårbarheter är kända i mjukvaror?

Schryen (2011) utgick från NVD och jämförde sårbarheter i öppen källkod och sårbarheter i proprietär källkod. Sjutton stora och välkända mjukvaror jämfördes. Inga signifikanta skillnader hittades vad gäller när sårbarheter publicerats, hur allvarliga de var eller hur de åtgärdats (buggrättats).

Johnson m.fl. (2016) undersökte variationer över tid för vilka individer som rapporterar upptäckta sårbarheter i en mjukvara för att uppskatta genomsnittlig tid för nya publicerade sårbarheter mjukvaran. Författarna beräknar med hjälp av sannolikhetsmodeller fram medeltid för publicering av nya sårbarheter för ett antal välkända mjukvaror och operativsystem.

NIST (Kuhn m.fl., 2017) redovisar trender över tid för olika typer av sårbarheter baserat på NVD. De beskriver att den vanligaste sårbarhetstypen över tid är olika typer av implementationsfel och noterar att många av dessa bör kunna undvikas med hjälp av utvecklingsverktyg eller utvecklingsmetodik.

Gorbenko m.fl. (2017) studerade olika operativsystems sårbarheter, huruvida sårbarheterna åtgärdats och hur allvarliga sårbarheterna är i form av värdering enligt CVSS, både i medel och i antal sårbarheter per CVSS-nivå. Författarna presenterar även statistik vad gäller tiden det tar från att en sårbarhet rapporteras i CVE-databasen till det att sårbarheten åtgärdas. Studien visar att sårbarheter med hög CVSS-nivå inte verkar åtgärdas snabbare än de sårbarheter som har låg CVSS-nivå. Författarna presenterar också de vanligaste typerna av sårbarheter (enligt CWE-kategoriseringen), samt vilka gemensamma sårbarheter som operativsystem har på grund av att de använder sig av gemensamma kodbibliotek.

Williams m.fl. (2018) använde statistiska modeller för att hitta mönster i NVD vad gäller vilka de mest sårbara och populära mjukvarorna var över tid, vilka typer av sårbarheter som var vanligast över tid, samt hur allvarlighetsgraden för (vissa) sårbarheter varierar över tid.

Anwar m.fl. (2021) påtalade att statistik och innehåll från NVD inte är helt tillförlitliga eftersom klassningen av sårbarheter i databasen inte alltid är rätt eller komplett.

[6] Se https://nvd.nist.gov/vuln-metrics/cvss för mer information.

Li m.fl. (2023) sammanställde sårbarhetsdatabaser som använts i forskningsartiklar. Utöver de vanligaste sårbarhetsdatabaserna NVD och CVE är även de kompletterande databaserna OSVDB och X-Force populära.

Det finns även andra typer av sammanställningar av sårbarheter. Ett exempel är OWASP topp-10 som utgör en sammanställning av vad organisationen OWASP anser vara de tio viktigaste webbsårbarhetstyperna (på CWE-nivå). Ett annat exempel är sökmotorn Shodan som söker igenom internet efter sårbarheter i industriella kontroll- och styrsystem. Genge och Enăchescu (2016) använde Shodan[7] för att extrahera och analysera CPE:er och sårbarheter.

Det är generellt sett svårt att avgöra vilka sårbarheter som är viktigast att fokusera på för en försvarare. Exempelvis beskriver Le m.fl. (2021) att databasernas vanligaste sårbarhetskategorier inte nödvändigtvis är de mest diskuterade sårbarheterna på utvecklarforum. Därtill anser Gueye m.fl. (2021) att det är oklart hur man bör mäta vilken sårbarhet som är mest signifikant. De nämner bland annat det som tveksamt att utgå från CWE eftersom det fokuserar för mycket på antalet förekomster. Det finns också avvikelser mellan allvarlighetsbedömningen som gjorts i CVSS och den som experter gör, vilket rapporterades av Holm och Afridi (2015). CISA m.fl. (2023) varnar för att bedöma säkerheten i en mjukvara baserat på antal CVE:er eftersom antalet också är ett tecken på hälsosam kodanalys och testning. Det verkar också finnas nationell hänsyn som tas inför publicering i sårbarhetsdatabaser. Exempelvis hävdar amerikanska Recorded Future att kinesiska databasen CNNVD dröjt med publiceringen av sårbarheter som kinesiska hackergrupper nyttjat (Moriuchi och Ladd, 2017).

4.3 Vilka sårbarheter utnyttjas?

Databaser över kända sårbarheter redovisar vanligtvis inte vilka sårbarheter som *utnyttjats* av illasinnade. Information om vilka sårbarheter som nyttjas kan däremot erhållas på andra sätt. Ett sätt är att utgå från databaser över cyberoperationer och använda deras källor för att hitta information om nyttjade sårbarheter. Sådana databaser sammanställdes i en FOI-rapport av Karlzén (2019). CISA har vid några tillfällen tillgängliggjort sammanställningar över vilka kända sårbarheter som de sett användas. Tabell 2 beskriver de tolv mest utnyttjade sårbarheterna under 2022 enligt CISA.

[7] Shodan är en sökmotor för att hitta Internet-anslutna enheter. Se https://www.shodan.io för mer information.

Tabell 2: De tolv mest utnyttjade sårbarheterna 2022. (CISA, 2023)

CVE	**Produkt**	**Konsekvens**
CVE-2018-13379	Fortinet FortiOS/FortiProxy	Exponering av behörighetsnycklar (eng. credentials)
CVE-2021-34473	Microsoft Exchange Server	Fjärrkodsexekvering
CVE-2021-31207	Microsoft Exchange Server	Möjlighet att kringgå säkerhetsfunktioner
CVE-2021-34523	Microsoft Exchange Server	Möjlighet att utöka privilegier
CVE-2021-40539	Zoho ManageEngine ADSelfService Plus	Fjärrkodsexekvering, möjlighet att kringgå autentisering
CVE-2021-26084	Atlassian Confluence Server/Data Center	Kodexekvering
CVE-2021-44228	Apache Log4j2	Fjärrkodsexekvering
CVE-2022-22954	VMware Workspace ONE Access and Identity Manager	Fjärrkodsexekvering
CVE-2022-22960	VMware Workspace ONE Access, Identity Manager/vRealize Automation	Felaktig hantering av privilegier
CVE-2022-1388	F5 Networks BIG-IP	Avsaknad av autentisering
CVE-2022-30190	Microsoft Flertalet produkter	Fjärrkodsexekvering
CVE-2022-26134	Atlassian Confluence Server / Data Center	Fjärrkodsexekvering

Förutom kända angrepp finns det förmodligen ett stort mörkertal när det gäller angrepp där även andra sårbarheter utnyttjas. Vilka dessa sårbarheter är kan till viss del sammanfalla med de sårbarheter som utnyttjas i angreppsövningar (såsom Pwn2Own) och som upptäcks vid penetrationstester, men det kan också röra sig om helt andra sårbarheter. Exempelvis kan en person som upptäcker en mjukvarusårbarhet välja att rapportera den till tillverkaren eller tipsa om den till någon som snarare vill att den används i angrepp. I det senare fallet kan det bland annat röra sig om företag som köper information om sårbarheter för att sälja dem vidare.

5 Varför har mjukvaror sårbarheter?

Det här kapitlet beskriver de studerade forskningsartiklarnas olika svar på varför mjukvaror har sårbarheter. Svaren är indelade i tre kategorier. Den första kategorin beskriver mänskliga och organisatoriska faktorer, såsom utvecklares attityder gentemot säkerheten, organisatoriska normer samt svårigheten i att utveckla på ett säkert sätt. Den andra kategorin beskriver programmeringsspråkens betydelse, såsom olika språks styrkor och svagheter samt försök att etablera språk som tagits fram med säkerhet som fokus. Den tredje kategorin beskriver hur införda sårbarheter kan upptäckas, såsom genom olika typer av granskningar och tester av mjukvaran.

Det kan tilläggas att en grundläggande anledning till att mjukvaror har sårbarheter är mjukvarornas komplexitet, vilket i sig kan antas (delvis) bero på att mjukvarorna löser komplexa problem. Exempelvis har forskning visat att en mjukvaras antal sårbarheter är förknippad med mängden komplexitet i koden och hur den tagits fram. Ett exempel på sådan forskning är Zimmermann m.fl. (2010) som undersökte de 66 sårbarheter som då rapporterats till NVD för operativsystemet Windows Vista. Antalet sårbarheter visade sig ha samband med faktorer som antal rader kod, antal ändringar, antal möjliga exekveringsvägar, antal funktioner och binärer som anropar varandra, antal globala variabler, antal rader kod som påverkats (lagts till, tagits bort eller ändrats) samt antal utvecklare och antal utvecklare som slutat. Ett nyare exempel på sådan forskning utgörs av Gkortzis m.fl. (2021). Där undersöktes drygt tusen mjukvaror på Github och det visades att antal sårbarheter var starkt kopplat till antal rader kod och antal beroenden. Att komplexare mjukvara tenderar att ha fler sårbarheter säger dock i sig inget om huruvida komplexiteten kan avlägsnas eller om sårbarheterna orsakas av komplexiteten eller av något annat som varierar med både komplexitet och sårbarhet.

5.1 Varför råkar utvecklare införa sårbarheter?

I intervjustudien av Haney m.fl. (2018) genomfördes 21 intervjuer med väldigt erfarna representanter för organisationer som utvecklar kryptografiska produkter. Deras resultat tyder på att kryptoutvecklare har en annan medvetenhet, organisationskultur och kunskap än andra utvecklare. Dessa skillnader påverkar i sin tur valet av vilka verktyg, utvecklings- och testmetoder som anses användbara. Om vi gör antagandet att mjukvaruutveckling av kryptoprodukter generellt präglas av högre säkerhet än annan mjukvaruutveckling kan de nyss nämnda resultaten vägleda beskrivningarna här. Därför tar de kommande

avsnitten upp aspekterna medvetenhet, attityder och normer, kunskap samt användbarhet hos kodbibliotek och verktyg.

5.1.1 Bristande medvetenhet

Medvetenhet av säkerheten och hot mot densamma studeras ofta inom cybersäkerhetsforskningen och detta gäller även forskningen om mjukvarusårbarheter. Flera forskningsartiklar har i experiment visat att utvecklare, testare och AI-bottar tillverkar säkrare mjukvara när de uppmanas att göra det eller på annat sätt medvetandegörs eller uppmärksammas om säkerhetsaspekten. Det varierar i artiklarna hur utvecklarna medvetandegörs om säkerheten. I vissa fall rör det sig om tydliga instruktioner, medan det i andra fall rör sig om att snarare påverka det undermedvetna hos utvecklarna. Nedan beskrivs mer detaljer om forskningen om medvetenhet rörande mjukvarusårbarheter.

Frilansutvecklare skriver säkrare kod när de uppmanas att göra det, men skriver osäkert annars.
Naiakshina m.fl. (2019) undersökte om 43 frilansutvecklare kunde fås att skriva säker mjukvara. Forskarna utgav sig för att vara företräda ett startup-företag och bad utvecklarna skriva kod för att lagra lösenord. Utvecklarna var i olika utsträckning benägna att välja osäkra programmeringsfunktioner för att lagra lösenord samt hävda att de skulle skriva säkert utan särskild uppmaning trots att de inte gjorde så. En uppmaning att skriva säkert hade en statistiskt säkerställd effekt på att lagra lösenord på ett säkert sätt.

AI-chattbotten ChatGPT behöver specifika instruktioner för att skriva säker kod samt är naiv vad gäller hotet från hotaktörer.
Khoury m.fl. (2023) undersökte om chattbotten ChatGPT (version 3.5) kunde skriva säkra program. Artikeln undersökte också om en direkt fråga räckte för att botten skulle erkänna (eller inse) att den skrev osäkra program, när den skrev sådana. Botten gavs uppgiften att skriva drygt tjugo program. Av dessa var bara fem säkra från början och i flera fall var dess osäkra program långt under normal standard för mänskliga utvecklare. Exempelvis skrev botten ett program med en gemensam (hårdkodad) kryptonyckel trots att programmet skulle användas för kommunikation av tre olika användare. På direkta frågor svarade botten å andra sidan ja på om programmen den skapat var osäkra. I sju fall lyckades den dessutom skapa förbättrade versioner som undvek de sårbarheter som forskarna ville testa. Samtidigt fanns andra typer av sårbarheter kvar i de förbättrade versionerna. Detta visade att botten behöver instrueras väldigt specifikt, varför den som frågar redan behöver känna till säkerhetsproblemet som ska åtgärdas, eller undvikas från början. Därtill trodde forskarna att en viktig anledning till de osäkra programmen var att botten antog att det inte fanns några hotaktörer. Ett argument var att den flera gånger föreslog att säkerhetsproblemen helt enkelt

kunde undvikas genom att användaren såg till att inte ge ogiltig input. En hotaktör skulle inte följa en sådan uppmaning.

Utvecklare saknar medvetenhet om skillnader mellan antagonistisk och icke-antagonistisk säkerhet.
Gasiba m.fl. (2021) utförde en enkätstudie om utvecklares syn på säker mjukvaruutveckling. Utifrån studien fastslog artikeln bland annat att utvecklare behöver medvetandegöras om skillnader mellan säker kodning (som är till för att förhindra uppkomsten av sårbarheter) och andra programmeringsaspekter såsom funktionell säkerhet och prestanda.

Uppmaning gör utvecklarna mer uppmärksamma på sårbarheter vid kodgranskning.
Oliveira m.fl. (2014) testade utvecklares förmåga att hitta sårbarheter i kod. Knappt femtio personer med liten till stor erfarenhet av mjukvaruutveckling fick i uppgift att på någon timme förbättra kod. Utvecklarna blev i grundfallet instruerade att ändra koden på ett visst sätt (som inte hade med säkerhet att göra) och samtidigt fundera på om koden kunde förbättras på något annat sätt. Utvecklarna fick olika uppmaningar. En uppmaning var att svara på om det fanns något ovanligt med koden och vad det i så fall kunde resultera i. En mer detaljerad uppmaning bestod av att utvecklarna fick information om att koden hade en sårbarhet och ombads hitta sårbarheten. Efter uppmaningarna fick utvecklarna frågor om när de funderade på sårbarheter. Resultaten visade att utvecklare normalt sett inte tänker på säkerhet när de programmerar men att uppmaningar (eng. priming) gör utvecklarna mer uppmärksamma på sårbarheter. Den uttryckliga uppmaningen gav särskilt bra resultat.

Uppmärksamhet på sårbarheter gör testare något bättre.
Braz m.fl. (2021) undersökte 146 personers förmåga att upptäcka sårbarheter. Deltagarna var i de flesta fall mjukvaruutvecklare. Deltagarna ombads kontrollera en kodändring där det, utan deltagarnas vetskap, införts en indatavalideringssårbarhet av en av två typer (SQL-injektion eller felaktig hantering av användarinmatning) samt en icke-säkerhetsrelaterad algoritmisk bugg. Knappt hälften av deltagarna uppgav att de upptäckte sårbarheten, varav knappt fyrtio procent när sårbarheten var tydlig, medan det för tio procent räckte att sårbarheten var mindre tydlig. Därefter fick deltagarna information om att koden var sårbar och en chans att kontrollera koden igen. Av de deltagare som inte redan upptäckt sårbarheten upptäcktes sårbarheten då av ytterligare nästan femton procent.

Möten ökar medvetenheten för säkerhet i vissa agila utvecklingsprojekt.
Tøndel och Cruzes (2022) undersökte små och medelstora företags utvecklingsmöten och fastslog att möten i lag av utvecklare är en populär

aktivitet i utvecklingsprojekt. Studien fastslog också att möten är särskilt frekvent använda i agil utveckling. Det visades också att möten med fokus på säkerhet kan göra att säkerhet upplevs som viktigt i organisationen och i allmänhet medvetandegöra säkerheten där. Därtill visade artikeln att det är främst i mindre och medelstora utvecklingsorganisationer med låg säkerhetskompetens som mötena får effekt.

5.1.2 Dåliga attityder och normer

Etablerade teorier inom informationssäkerhetsområdet talar om att säkerhetsbeteenden beror på vilka attityder och normer som finns i organisationen. Detta kan också antas gälla arbetet med att förebygga och hantera mjukvarusårbarheter. Attityder berör sådant som personlig drivkraft för säkerhet. Normer täcker in sådant som organisationens prioritet av och stöd för säkerhet samt kollegors åsikter om att använda säkerhetsrelaterade verktyg. Mer detaljer om attityder och normer som rör mjukvarusårbarheter beskrivs nedan.

Utvecklares drivkrafter för mjukvarusäkerhet är flera.
Assal och Chiasson (2019) använde en enkät för att undersöka utvecklares drivkrafter för mjukvarusäkerhet. De viktigaste drivkrafterna var att vara medveten om att mjukvaran har implikationer för säkerheten, gilla säkerhetsarbete, anse att säkerhetsarbete är ett delat ansvar, bry sig om sina användares säkerhet, bry sig om sin organisations rykte samt att gilla utmaningar i utvecklingsarbetet.

Organisationsfaktorer kan både öka och minska säkerhetsprioriteten.
Tøndel m.fl. (2022) undersökte säkerhetsprioriteten hos ett företag med åttio utvecklare som arbetade agilt. De viktigaste faktorerna för att öka säkerhetsprioriteten visade sig vara säkerhetskrav som gav synbar säkerhet, säkerhetsförkämpar, säkerhetskrav från kunden snarare än tillit, externa säkerhetsexperter, GDPR, en dedikerad budget och dedikerade roller för säkerhet samt att det var enkelt att implementera säkerheten av utvecklare utan att fråga om tillstånd. De viktigaste faktorerna som tvärtom minskade säkerhetsprioriteten var att fokusera på funktionalitet och att bli klar i tid, kostnadsbesparingar, tidspress, bristande kompetens, komplexitet, vana vid att ignorera säkerheten, att säkerhet ses som något extra, överväldigande dokumentation samt bristande ansvar och organisation.

Olika utvecklare har olika personligt intresse av säkerhet och olika stöd från omgivningen.
Ryan m.fl. (2021) skapade en teori som delade in utvecklare efter två dimensioner: personligt intresse av säkerhet och stöd från omgivningen. Tillsammans bildar de två dimensionerna fyra typer av utvecklare. *Förkämpar* är högmotiverade och väl understödda. Förkämparna sprider sin expertis och motivation till övriga organisationen. *Hjältar* är högmotiverade men dåligt understödda. Hjältarna kämpar för säkerheten i organisationer där säkerhet mest

ses som ett hinder. *Pragmatiker* är omotiverade men väl understödda. Pragmatikerna bryr sig inte om säkerheten men följer reglerna om det inte påverkar det övriga jobbet negativt. *Optimister* är omotiverade och dåligt understödda. Optimister undviker säkerhetsarbete med argument som att det kommer gå bra ändå, att någon annan tar hand om säkerheten, att deras mjukvara är alltför obetydlig för att angripas samt att det inte finns något av värde att stjäla i systemet.

Normer är viktiga anledningar till att använda säkerhetstestningsverktyg. Witschey m.fl. (2015) undersökte anledningar till att utvecklare använder statiska och dynamiska säkerhetstestningsverktyg. Förutom kostnadseffektivitet var de viktigaste faktorerna att se sina kollegor använda verktygen, att cheferna förväntade sig att verktygen användes (exempelvis genom företagspolicyer) och att utvecklarna var intresserade av verktygen.

Få Github-projekt kräver att säkerhetstestningsverktyg används för att kontrollera säkerheten. Vassallo m.fl. (2020) utförde en manuell inspektion av vägledningar för nästan tvåhundra mjukvaror med öppen källkod i Github. Slutsatsen var att bara en procent av Github-projekten kräver att verktygen används för att kontrollera säkerheten.

5.1.3 Okunskap

En viktig anledning till undermålig säkerhet är bristande kunskap hos utvecklarna. Mjukvaruutvecklarnas roll är bred och de måste ha kunskap om mycket. Därtill spelar utbildning och erfarenhet roll för att förebygga, upptäcka och rätta till sårbarheter. Det är också svårt att åtgärda kunskapsglappet i en organisation eftersom det är svårt att få tag i kompetenta utvecklare i säkrare språk, svårt att veta vilka utvecklingsmetoder som faktiskt fungerar samt avgöra vilka informationskällor det går att lita på. Mer detaljer om alla dessa aspekter följer nedan

Det ställs stora krav på utvecklaren för att uppnå säker mjukvara.
CISA (2020) beskriver olika cybersäkerhetsroller i NICE-ramverket, vilket tagits fram av en motsvarighet till informationssäkerhetsdelen hos MSB och industrin. Rollen mjukvaruutvecklare är en av rollerna inom området mjukvaruutveckling och beskrivs (fritt översatt) som ”den som utvecklar, skapar, underhåller och skriver nya (eller modifierar existerande) datorapplikationer, mjukvara eller systemverktyg”. Vad gäller säkerheten behöver mjukvaruutvecklare bland annat ha förmågan att utveckla enligt säkra metoder; ha kunskap om sårbarheter och säkra kodningstekniker; samt kompetens i att validera input, använda publika kryptoinfrastukturer samt använda kodanalysverktyg.

Oerfarna utvecklare ger mer sårbar kod relativt mängden kod som produceras. Bosu m.fl. (2014) studerade säkerhetsnivån på den kod som utvecklare med olika erfarenhet skrev i tio olika mjukvaror med öppen källkod. Mer erfarna utvecklare

skrev mer kod och orsakade därför flest sårbarheter. Men mindre erfarna utvecklare orsakade mellan knappt två och tjugofyra gånger så många sårbarheter per utvecklad mängd kod.

Erfarenhet är viktig för att ha nytta av statiska säkerhetsanalysverktyg.
Baca m.fl. (2009) beskrev att den typiska utvecklaren överlag är dålig på att klassificeras varningar från statiska kodanalysverktyg. I studien undersökte de hur bra utvecklarna kunde klassificera in varningar från ett kodanalysverktyg i de tre kategorierna falska positiver (påvisar inte en brist), sanna positiver (påvisar en brist som inte kan nyttjas av en angripare) och säkerhetsbrist (påvisar en brist som kan nyttjas av en angripare). Studien visade att utvecklarna klassificerade sanna positiver i hög utsträckning, medan de klassificerade falska positiver och säkerhetsbrister ungefär lika dåligt som slumpen. Mer erfarna utvecklare var bättre på att göra korrekta klassificeringar, där erfarenhet av verktygen fördubblade mängden korrekta bedömningar av verktygens output. Om användaren även hade ökad säkerhetserfarenhet blev den totala förbättringen trefaldig.

Utbildning och erfarenhet spelar roll för förmågan att upptäcka sårbarheter.
Allodi m.fl. (2020) undersökte hur bra studenter och yrkesverksamma är på att bedöma mjukvarusårbarheter som beskrivits i text. Masterstudenter med inriktning på informationssäkerhet var bättre än studenter med inriktning på datavetenskap. Studenterna med inriktning på informationssäkerhet var dessutom för det mesta lika bra som yrkesverksamma säkerhetsspecialister i att bedöma nödvändiga privilegier, komplexitet och attackvektorer. Specialisterna var dock bättre på att bedöma vilken användarinteraktion som krävdes för sårbarheternas utnyttjande.

Buggrättningar medför nya sårbarheter på grund av undermålig kunskap.
Yin m.fl. (2011) undersökte säkerhetsrelevanta buggrättningar i ett antal olika operativsystem och hur de kan medföra nya sårbarheter. Artikeln nämner några sådana historiska fall. Exempelvis släppte antivirusföretaget Mcafee 2010 en buggrättning som råkade identifiera en kritisk systemfil i Windows som virus varefter tusentals datorer förlorade sina nätverksanslutningar eller inte längre kunde starta. Därtill visade sig tio procent av Microsofts säkerhetsbuggrättningar under 00-talet vara felaktiga. Studien beskriver att runt 15–25 % av de utvalda buggrättningarna medförde nya sårbarheter, där det varierade mellan de utvalda operativsystemen. För olika operativsystem medförde knappt hälften av alla inkorrekta buggrättningar allvarliga konsekvenser i form av krascher, att datorer hängde sig, data blev korrupta eller att säkerheten äventyrades. Anledningen till problemen beskrivs delvis bero på att utvecklarna valde generellt sett tveksamma lösningar på de ursprungliga buggarna. Exempelvis var bara ett av tre återkommande sätt att lösa buffertöverskridningar att begränsa funktionens inputlängd (med t.ex. snprintf) medan de andra två sätten ansågs riskfyllda

eftersom de var svåra att utföra korrekt. De inkorrekta buggrättningarna uppstår också förmodligen på grund av undermålig kunskap om koden. Exempelvis utfördes en dryg fjärdedel av de inkorrekta buggrättningarna av utvecklare som aldrig tidigare rört den relevanta källkoden.

Korrekta buggrättningar är svåra ta fram.
Dissanayake m.fl. (2022) sammanställde sjuttiotvå artiklar om buggrättning av sårbarheter. Sammanställningen beskriver att den mesta forskningen fokuserat på att identifiera behov av buggrättningar och att lansera buggrättningarna. Däremot saknas det forskning om testning och verifiering av att buggrättningarna är korrekta.

Organisatoriskt anammande av nya säkrare språk begränsas av svårigheten att få tag i kompetenta utvecklare.
Fulton m.fl. (2021) undersökte i en enkät- och intervjustudie vad som begränsade användningen av språket Rust, vilket förespråkas ur säkerhetsperspektiv. En viktig begränsning från organisationernas håll var att det var tveksamt om det skulle gå att få tag i kompetenta utvecklare i framtiden. Därtill beskriver organisationerna det som negativt med ett relativt okänt språk, att onödiga ändringar borde undvikas samt att det kanske inte passar in i det existerande ekosystemet av språk och utvecklingsmiljöer.

Bristande evidens för goda säkerhetseffekter från etablerade metoder för säker mjukvaruutveckling.
Kudriavtseva och Gadyatskaya (2022) sammanställde i en ogranskad (publicerad på ArXiv) artikel 28 metoder för säker mjukvaruutveckling, vilka tagits fram av industri, myndigheter och forskare. Metoderna bygger bland annat på mjukvarors livscykler, där det typiskt beskrivs hur en mjukvara går från krav, riskbedömning och design till konstruktion och testning samt vidare till underhåll, konfiguration och avveckling. Sådana modeller kan vara linjära (som vattenfall) eller mer upprepade och agila.[8] I de flesta av metoderna antyddes det att deras användning skulle leda till ökad mjukvarusäkerhet. Samtidigt angav metoderna ingen evidens för deras påståenden. Artikeln frågade sig varför det fanns så många metoder, varför det kommer fler och fler metoder samt varför säkerhetsproblemen inte verkar avta med den ökade användningen av metoderna.

Webbforumet Stack Overflow ger generellt sett undermåliga säkerhetsrelaterade råd till utvecklare.
Meng m.fl. (2018) undersökte webbforumet Stack Overflow angående Java-

[8] Ett exempel på en modell för säker mjukvaruutveckling är NIST:s SP 800-218 Secure Software Development Framework (Souppaya m.fl., 2022). Förutom den typen av utvecklingsmodeller finns modeller och evalueringskriterier för att utvärdera mognaden i en viss implementering av livscyklerna och en organisations övergripande mjukvaruförmåga. Ett exempel på en modell för säkerhetsutvecklingsmognad är SSECMM (ISO/IEC 21827:2008). Den beskriver säkerhetsmässig praxis indelat i 22 processområden såsom ”bedöm hot”, ”administrera säkerhetsfunktioner”, och ”förse med färdigheter och kunskap” (översatt från engelska).

säkerhet och då vilka svar som frågeställare märkte som accepterade (eller inte). Många av de accepterade svaren visade sig ge mycket dåliga råd ur säkerhetssynvinkel. Exempelvis rekommenderades i nio av tio SSL/TLS-poster att irriterande fel skulle undvikas genom att låta koden lita på alla certifikat eller alla värdnamn, varigenom hela säkerheten kringgås. I ett fall gavs det osäkra svaret av en medlem med mycket bra rykte medan en nyare medlem som ifrågasatte svaret inte togs på allvar.

5.1.4 Låg användbarhet hos bibliotek och testverktyg

Det räcker inte med att det finns tillgängliga kodbibliotek och testverktyg som potentiellt kan ge utvecklare säker kod. Biblioteken och verktygen måste också vara ge tillräcklig användbarhet. Exempelvis är ett kryptografiskt kodbibliotek bara av nytta om utvecklare förstår hur biblioteken ska användas. På liknande sätt behöver testverktyg ge användbara varningar om sårbarheter och det i en hanterarbar mängd. Nedan beskrivs mer detaljer om användbarheten hos kodbibliotek och testverktyg.

Kodbibliotek för kryptografiska funktioner är svåranvända.
Nadi m.fl. (2016) samlade empiri utifrån Github, Stack Overflow och en enkät med utvecklare. Slutsatser drogs att utvecklare tycker att Javas kryptografiska kodbibliotek (API:er) är svåra att använda och dåligt dokumenterade. Därtill upplevs biblioteken alltför finmaskiga vilket till exempel gör att det är svårt att veta i vilken ordning olika anrop ska göras.

Flera saker kan göra kodbibliotek för kryptografiska funktioner mer lättanvända.
Green och Smith (2016) föreslog tio principer för att göra krypto-API:er mer användbara och säkra. Principerna inkluderar bland annat att integrera krypto-funktionalitet i standard-API:er (så att utvecklare inte behöver använda dem explicit), se till att förhandsval är säkra och tydliga, gör krypto-API:erna användbara även utan dokumentation samt att se till att krypto-API:er har en testinställning (för att undvika att utvecklarna avaktiverar krypto-funktionalitet under utveckling för att sedan glömma bort att aktivera den igen).

Verktyg för att utveckla och testa mjukvara saknar flera saker ur ett användbarhetsperspektiv.
Smith m.fl. (2020) undersökte flera verktyg för att analysera säkerheten i mjukvara. Undersökningen lät utvecklare använda verktygen och följde upp med intervjuer. Studien resulterade i bedömningen att verktygen hade flera användbarhetsbrister. Exempelvis saknades övergripande listor på alla varningar, definitioner av terminologi, förklaringar av hur koden kunde utnyttjas av angripare, exempel på hur koden kunde åtgärdas samt möjlighet att ändra koden direkt där varningen visades.

Verktyg för att utveckla och testa mjukvara behöver en avvägning mellan att producera snabba varningar om sårbarheter och arbetsro för utvecklarna.

Tahaei och Vaniea (2019) sammanställde forskningsartiklar om säker mjukvaruutveckling med fokus på utvecklaren. Sammanställningen undersökte bland annat hur feedback ska ges till utvecklare vid användning av utvecklings- och testverktyg. Å ena sidan kan snabb feedback möjliggöra att sårbarheter åtgärdas snabbt. Å andra sidan bör inte utvecklaren bli överöst med varningar utan ska istället kunna få programmera ifred. Tidigare forskning har ofta valt att använda sig av snabb feedback, dock utan att tillräckligt starkt ha underbyggt detta val enligt källan. Vidare beskrivs att studier visat att det är bättre att avbryta en person som skriver text efter att ett stycke är färdigskrivet. Detta är dock inte uppenbart direkt översättningsbart till mjukvara, varpå källan pekar på att vidare forskning inom området behövs.

5.2 Varför medger programmeringsspråk sårbarheter?

Själva programmeringsspråken har betydelse för mjukvarors sårbarheter. De följande avsnitten beskriver olika språks roller och språkens kombinationer samt utveckling av säkrare språk.

5.2.1 Språken har olika svagheter

Mängden sårbarheter som en mjukvara har beror delvis på vilket eller vilka programmeringsspråk som mjukvaran är skriven i. Framförallt är det typen av sårbarheter som varierar mellan språk, men till viss del skiljer sig även antalet sårbarheter. Därtill kan kombinationer av språk ge fler sårbarheter när mjukvara är skriven i flera språk. Nedan beskrivs mer om programmeringsspråkens roll för mjukvarusårbarheter.

I webbforum diskuteras det olika mycket om säkerhet för olika språk. Därtill skiljer det sig vilka säkerhetsaspekter som diskuteras.
Croft m.fl. (2022) undersökte vilka säkerhetsrelaterade utmaningar som utvecklare står inför när de använder olika programmeringsspråk och hämtar information om säker programmering på Stack Overflow och Github. Mer än 100 000 inlägg vardera på Stack Overflow och Github analyserades. För varje språk var diskussionerna säkerhetsrelaterade i 0–4 % av fallen. De språk med mest högst andel säkerhetsrelaterade diskussioner var Shell, PHP och Powershell på Stack Overflow samt C/C++, Erlang och PHP på Github. De mest lägst andel var sammanvägt Matlab och Fortran. Detta säger dock inte att vissa språk är säkrare än andra, utan enbart vad diskussionerna handlade om. För olika språk skilde det sig åt vilka typer av säkerhetsproblem som diskuterades mest. Några exempel är att det för Java, Javascript och PHP var vanligast med diskussioner om testning, implementering och autentisering. För C# var det vanligaste lösenordslagring, testning, bibliotek och autentisering. För Python var det vanligaste testning, implementering, nätverkssårbarheter och

autentiseringstokens. För Shell var det vanligaste nätverkskommunikation, autentisering och bibliotek. För C/C++ var det vanligaste kryptering, minneshantering, lösenordslagring och implementering. Författarna gör ingen vidare fördjupning i hur de säkerhetsrelaterade diskussionerna för respektive programmeringsspråk korrelerar med hur vanliga de olika sårbarheterna är för språket. Däremot noteras att de säkerhetsrelaterade diskussionerna ofta rör ämnen kopplade till de användningsområden som programmeringsspråket är populärt inom.

Mjukvara skriven i flera programmeringsspråk verkar leda till fler sårbarheter. Kochhar m.fl. (2016) undersökte 85 miljoner rader kod från totalt 628 mjukvaror (projekt) på Github. Ungefär hälften av mjukvarorna var skrivna i mer än ett programmeringsspråk. Resultaten visade att användandet av flera programmeringsspråk i en mjukvara var förknippat med fler sårbarheter (och buggar i allmänhet) jämfört med andra faktorer såsom kodbasens ålder, antalet utvecklare, storlek för kodbas och antal versionsändringar (eng. commits). Artikeln föreslår ytterligare forskning som undersöker vilka språk som passar bäst ihop.

5.2.2 Språkens utveckling har säkerhetsbrister

Att utveckla säkrare programmeringsspråk som kan användas för att utveckla säkrare mjukvara är ett sätt att minska problemet med mjukvarusårbarheter. I detta avsnitt beskrivs några exempel på påstått säkrare språk samt vilka begränsningar som trots allt finns med dessa språk. Det finns också försök att ta fram säkrare dialekter av språk och vägledningar för säkrare språkanvändning. Ett exempel är SEI CERT Coding Standards för språket C (SEI-CMU. 2022), där det bland annat föreslås sätt att undvika sådan riskabel hantering av heltal som kan leda till buffertöverskridningar vilka kan användas av angripare.

Rust som potentiellt säkrare språk men som tillåter avvikelser vilka kan äventyra säkerheten. Därtill utvecklares syn på språkets säkerhet och begränsningar. Fulton m.fl. (2021) beskriver språket Rust som skapades av Mozilla och då bland annat med avsikten att förebygga segmenteringsfel och trådosäkerhet. En egenhet med Rust är dess undvikande av nullpekare. För att hantera problemet med att pekare används efter att utpekat minne frigjorts har vissa språk ett inbyggt omhändertagande av pekarna. Rust saknar detta men ser istället till att varje variabel har en ägare, som i sin tur kan låna ut variabeln. Detta upplägg hindrar dock det funktionella skapandet av dubbelt länkade listor (för iterering bakifrån) och grafdatastrukturer. För att kunna använda sådana strukturer används det i vissa fall särskilda bibliotek som går runt det nyss nämnda ägartänkandet. Det går också att använda särskilda kodblock markerade som osäkra (eng. unsafe) för att exempelvis öka prestandan eller göra det lättare att interagera med externa funktioner. Dessa avvikelser i bibliotek och kodblock kan dock äventyra säkerheten. Samma artikel undersökte i en enkät- och intervjustudie med Rust-

utvecklare vad som var positivt och negativt med att använda språket Rust. Det upplevdes som positivt att det inte fanns nullpekare, att variabler är oföränderliga som grundinställning, att minneshanteringen var säker och att prestandakraven var låga. Dessutom kände utvecklarna tillit till att skapad mjukvara var säker. Därtill började utvecklarna tänka säkrare även när de åter utvecklade i andra språk. Det fanns också negativa aspekter med att använda Rust, såsom att språket upplevdes ha få bibliotek och att det tog lång tid att kompilera.

Det kan också nämnas att flera vanliga operativsystem nyligen fått (eller planerats få) nya versioner som viss mån baseras på Rust. Detta gäller både Windows (Claburn, 2023), Linux (Purdy, 2023) och Android (Stoep, 2022).

Go som potentiellt säkrare språk men där ytterligare säkerhetsaspekter kunde ha beaktats.
Cox m.fl. (2022) beskriver språket Go som skapades av Google och som nu är populärt för att skapa molninfrastrukturer, såsom containersystemen Docker och Kubernetes. I artikeln ger skaparna av språket sin syn på varför Go blivit populärt och jämför ett antal säkerhetsegenskaper hos Go med motsvarande egenskaper hos C och C++. De säger att Go är bättre genom bland annat skräpminnesinsamling (eng. garbage collection) snarare än manuell minneshantering, samt felmeddelanden vid anrop som går utanför definierade ramar (eng. out-of-bounds array index). Därtill har Go inbyggt stöd för SSL/TLS med en färdig klient och server för HTTPS i standardbiblioteket. Författarna säger sig dock i efterhand möjligen ha missat vissa säkerhetsaspekter, exemplifierat med att alltför höga heltal (eng. integer overflow) gör att mjukvaran tippar över (eng. wrap around) till stora negativa heltal snarare än ger ett körfel (eng. runtime error).

Mer om säkerhetsegenskaper hos programmeringsspråk finns att läsa i flera andra rapporter från FOI (och föregångaren FOA).[9]

5.3 Varför upptäcks inte sårbarheter av utvecklare och testare?

Med statisk analys (statiskt testande) utvärderas mjukvaran utan att koden exekveras. Därmed utgår analysen enbart på mjukvarans källkod (eller ibland dess binärkod). Analysen används ofta för att hitta sårbarheter och sker genom manuell källkodsgranskning eller delvis manuellt med verktygsstöd med statiska analysverktyg (Liu m.fl., 2012). Ett alternativ till statisk analys är dynamisk

[9] Palme (1969) beskriver den tidiga historiken för programmeringsspråk samt olika egenskaper hos språk exempelvis rörande säkerhet. Andersson (2001) beskriver grunderna om Javascript samt dess säkerhetsegenskaper och dokumenterade säkerhetsbrister. Rodhe och Karresand (2016) beskriver bland annat säkerhetstypade språk och utökningar av dessa.

analys, där mjukvaran utvärderas baserat på att den exekveras. Som komplement finns också mer rigorösa formella metoder. Dessa olika varianter av testning beskrivs närmare nedan.

5.3.1 Svårgenomförd manuell källkodsgranskning

Ett sätt att upptäcka sårbarheter är med manuell granskning av källkoden. Kodgranskning är en aktivitet där utvecklare (ofta annan än den ursprungliga kodförfattaren) manuellt kontrollerar koden. Nedan beskrivs några exempel på forskning om detta.

Forskning om kodgranskning fokuserar på bättre metoder och bättre sätt att välja granskare, men saknar generellt sett säkerhetsvinkel.
Davila och Nunes (2021) beskriver en systematisk litteraturgenomgång av 139 forskningsartiklar om kodgranskning. Genomgången visar att forskningen bland annat fokuserat på att ta fram bättre granskningsmetoder samt bättre sätt att välja granskare genom olika rekommendationssystem baserat på historik. De allra flesta artiklar i litteraturgenomgången saknar dock säkerhetsvinkel. Det saknas alltså i stor utsträckning forskning om säkerhetsrelaterad manuell källkodsgranskning. En av de säkerhetsrelaterade artiklar som trots allt ingår i genomgången är di Biase m. fl. (2016), som beskrivs i nästa stycke.

Flera olika typer av sårbarheter kan missas vid manuell kodgranskning.
di Biase m.fl. (2016) undersökte kodgranskningen i Chromium, som huvudsakligen är skriven i C++, avseende upptäckt av säkerhetsrelaterade brister. Enligt Chromiums policy ska varje incheckning granskas av minst två personer innan den accepteras.. En slutsats från studien var att de sårbarheter som missas mest var vad de kallar språkrelaterade problem (såsom användning av icke-initialiserade variabler), möjligheten till buffertöverskridningar samt webbkodinjektion. De brister som upptäcktes i granskningen var primärt av samma typer, vilket också indikerar att de är de vanligaste typerna av brister som uppstår i projektet. Andelen språkspecifika problem var dock märkbart högre bland de som missats vid granskning jämfört med de som upptäckts, vilket antyder att granskningen kan vara ineffektiv när det gäller att hitta den typen av problem.

5.3.2 Bristande verktyg för statisk säkerhetstestning

Statiska analysverktyg kan exempelvis bedöma vilka dataflöden en mjukvara kan ha och om vissa av dessa flöden kan utgöra sårbarheter och därmed möjliggöra angrepp såsom buffertöverskridningar. Statiska verktyg kan också leta efter sådant som kända sårbara programmeringsfunktioner eller programmeringsmönster. Dessa verktyg kan användas enskilt eller inbyggt i

utvecklingsmiljöer (IDE:er). Verktygen hittar defekter och graderar normalt dem i allvarlighetsgrad. Därtill kan verktygen i viss mån föreslå lösningar på defekterna i form av buggrättningar (Ayewah m.fl., 2008). Mer detaljer om statiska verktyg beskrivs nedan.

Verktyg för statisk säkerhetstestning är generellt sett av begränsad nytta. Inget sådant verktyg kan hitta sårbarheter av alla typer och olika verktyg är bra på att hitta olika typer.
Oyetoyan m.fl. (2018) undersökte vilken roll sex verktyg för statisk säkerhetstestning spelade i agil utveckling hos Telenor. Undersökningen utgick från NIST:s Software Assurance Reference Dataset för att testa säkerhetsverktyg. En särskild del av datasetet valdes, med drygt tjugotusen konstgjorda testfall kategoriserade i drygt hundra olika typer av sårbarheter (CWE:er). Verktygen visade sig generellt sett vara av begränsad nytta. Inget verktyg kunde hitta sårbarheter av alla typer och olika verktyg var bra på olika typer av sårbarheter. För nummerhanteringskategorin (t.ex. division med noll) hittade inte ett enda verktyg några sårbarheter.

AI-tekniker kan för närvarande inte upptäcka sårbarheter vid mindre kodändringar.
Lomio m.fl. (2022) undersökte om AI-tekniker kan vara av nytta för att upptäcka sårbarheter i mjukvarors versionsändringar. Undersökningen baserades på sårbarhetsdata från NVD om nio Java-mjukvaror med totalt knappt nio tusen versionsändringar och drygt nittiotusen rader kod, vilka testas med åtta maskininlärningstekniker. I sårbarhetsdata fanns information om varje versionsändring, såsom mängd ny kod, antal författare, antal tidigare ändringar samt antal rader kod. Slutsatsen var att AI-teknikerna är undermåliga i dagsläget och därmed inte kan användas för att förutsäga sårbarheter för enskilda versionsändringar (snarare än för större kodändringar).

5.3.3 Utmaningar med dynamisk analys och formella metoder

Dynamisk analys och formella metoder utgör två sätt att analysera en mjukvaras beteende. I en dynamisk analys utvärderas mjukvaran baserat på att den exekveras, medan användningen av formella metoder bygger på matematiska modeller av mjukvaran som sedan bevisas utifrån en eller flera egenskaper.

En form av dynamisk analys är fuzzing,[10] vilket är en i stora delar automatiserad testningsteknik som exekverar mjukvaran med olika ovanliga (eller oförutsedda)

[10] Fuzzing är snarlikt den formella metoden symbolisk exekvering och mer om det beskrivs i FOI-rapporten Cohen m.fl. (2016). Se även FOI-rapporten Löfvenberg m.fl. (2019).

indata på olika sätt. Dessa indata kan skapas genom att ändra på legitima data innan de används som indata, skapa helt nya indata utifrån kännedom om protokoll och filformat, eller genom att använda slumpade indata. Efter att indata införts kan det observeras hur mjukvaran (och systemet) beter sig för att identifiera avvikelser som bland annat kan ha relevans för säkerheten (Liu m.fl., 2012).

Ett annat sätt att upptäcka sårbarheter är med formella metoder, vilka kan ses som en mer rigorös form av statisk eller dynamisk analys. Med formella metoder resoneras det om i princip alla möjliga programexekveringar utifrån en formell specifikation och angivna resursbegränsningar (t.ex. för mjukvarans indatas storlek). Specifikationen kan bestå både av generella krav, såsom att division med noll är förbjuden, och av krav som är specifika för den mjukvara som testas.

Verktyg för formella metoder kan stödja sårbarhetsupptäckt, men ställer krav på att mjukvaran anpassas efter det.
Bildsten m.fl. (2018) har skrivit en FOI-rapport som beskriver flera varianter av formella metoder.[11] En populär variant är symbolisk exekvering vilken påminner om algebraiska ekvationer. Symbolisk exekvering går igenom program med abstrakta (symboliska) variabler istället för konkreta indata, grenar sig för varje villkor (t.ex. if-satser), för att på slutet lösa ut vilka konkreta värden på variablerna som skulle ge att vilken gren exekveras. Det finns olika begränsningar med formella metoder, t.ex. på grund av falsklarm, att verktygen kräver manuell handpåläggning samt att den mjukvara som testas måste vara skriven på lämpligt sätt och med tillräckligt detaljerad dokumentation. Studien menar att användningen av verktyg för formella metoder ger ett mervärde vid verifiering av säkerhetskritisk mjukvara, speciellt när metoderna används redan under programmeringsfasen, men att de kan vara svåra att applicera på befintlig källkod.

[11] FOI-rapporten Bildsten m.fl. (2018) utvärderar även ett antal verktyg för formella metoder med avseende på verktygens användbarhet för att verifiera säkerhetskritisk mjukvara i Försvarsmaktens IT-system. Därtill har det skrivits om formella metoder i FOI-rapporten Rodhe och Karresand (2015). Där beskrivs hur testare ska välja bland alla formella metoder (t.ex. beroende på domän och säkerhetsprioritering), hur formella metoder har använts samt vilka som forskar om formella metoder i Sverige.

6 Vilken forskning behöver utföras om mjukvarusårbarheter?

Utifrån resultaten som sammanställs i de föregående kapitlen går det att konstatera att mycket forskning ännu återstår inom säker utveckling av mjukvara. Dels tar de citerade artiklarna ibland upp möjlig framtida forskning, dels har forskningsbehov identifierats genom de analyser som den här rapportens författare gjort. Resultaten från analyserna ger dock inte någon komplett eller säkerställd bild av forskningsbehoven då underlaget inte utgörs av en fullständig systematisk litteraturgenomgång av redan utförd forskning. Sammanställningen nedan ska därmed ses som en indikation på vilken forskning som kan behövas i framtiden och som är relevant för Försvarsmakten.

Förslagen på forskning som tas upp i detta kapitel är indelade i områden som ungefär motsvarar de rubriker som återfinns i kapitel 5. Förslagen är tänkta att vara någorlunda konkreta, och inte fokusera på allmänna förbättringar av forskningen i stil med att utföra större studier, att ha mer realistiska studiemiljöer eller att upprepa forskning om allmän mjukvarukvalitet fast specifikt inriktat på buggar som utgör sårbarheter. Det har inte utförts någon realiserbarhetskontroll för förslagen och de innehåller generellt sett inte heller förslag på hur forskningen bör utföras. Förslagen har inte heller getts någon prioriteringsordning, utan återges i en ordning som framför allt är tänkt att öka läsbarheten.

6.1 Utvecklare och organisationer

Utifrån resultaten i avsnitt 5.1 har vi identifierat följande forskningsbehov.

Säkerhetsmedvetenheten bland de inblandade personerna tycks vara en viktig faktor för den utvecklade mjukvarans säkerhetsnivå. Olika former av mätningar av medvetenheten kan användas såväl som underlag för bedömning av faktorn i branschen som helhet samt som basnivå för jämförelser inom enskilda organisationer eller för effekten av utbildningsinsatser. Följande frågor identifierar forskning som saknas. *Hur ser säkerhetsmedvetenheten ut bland svenska mjukvaruutvecklare i allmänhet? Hur är säkerhetsmedvetenheten bland mjukvaruutvecklare inom svensk försvarsindustri? Hur ser svenska utvecklares drivkrafter och attityder ut kring säkerhet?*

Med föregående styckes utgångspunkt bör en förbättring av säkerhets-medvetenheten även kunna förbättra säkerheten i de utvecklade mjukvarorna. Följande frågor är därför intressanta ur forskningsperspektiv. *Hur kan utvecklare göras medvetna om säkerhetsaspekterna? Hur kan de bli medvetna om skillnaderna mellan antagonistiska och icke-antagonistiska säkerhetsrisker?*

Ett näraliggande område till säkerhetsmedvetenhet är effekten av säkerhetsrelevanta utbildningar för utvecklare och annan personal som är involverade i utvecklingsarbetet. Följande frågor är exempel på relevanta framtida forskningsområden inom detta. *Vilken effekt får vidareutbildning inom säker programmering på produkternas säkerhet? Vilken effekt får vidareutbildning inom exempelvis mjukvarusäkerhet och utvecklingsmetodik för övriga inblandade personer (såsom projektledare, produktägare och testare) på produkternas säkerhet?*

När en organisation anställer personal finns det många olika typer av egenskaper och kompetenser som behöver bedömas. Exempelvis är kandidaternas säkerhetsmedvetenhet och säkerhetsrelevanta kunskaper sådant som kan behöva undersökas i samband med anställning. Följande frågor är exempel på relevant forskning inom detta. *Hur kan anställningstester utformas för att underlätta urvalet av utvecklare som har goda egenskaper när det gäller säker utveckling (såsom insikt, motivation eller anpassningsbarhet)? Hur kan tester utformas för att identifiera exempelvis lovande förkämpar och hjältar?*

Vissa organisationer tycks vara bättre än andra på att producera säkra produkter och system. Det vore relevant att undersöka vilka faktorer som påverkar säkerheten för mjukvaruprodukter och mjukvaruprojekt som ska användas inom Försvarsmakten. Följande frågor är exempel på relevant forskning. *Vilka organisationsfaktorer påverkar mjukvarusäkerhetsarbetet inom Försvarsmakten och FMV? Vilka organisationsfaktorer påverkar mjukvarusäkerhetsarbetet hos Försvarsmaktens och FMV:s leverantörer?*

En kultur som främjar säkerhet och säkra produkter kan exempelvis uppstå hos organisationer som tar fram produkter med särskilda krav, exempelvis för säkerhetsskydd eller sekretesskydd. Följande frågor är exempel på intressanta områden där mer forskning behövs. *Hur kan utvecklingsorganisationer lära sig av andra organisationer med särskilt säker kultur inom utveckling?*

Trots årtionden av forskning och utveckling inom såväl utvecklingsmetodik som säkrare programmeringsspråk och säkerhetsverktyg tycks antalet sårbarheter inte avta. Ett sätt att undersöka detta kan vara genom långsiktiga studier som inkluderar datainsamling såväl före som efter införandet av säkrare utvecklingsmetoder inom en eller flera organisationer. Det till synes paradoxala förhållandet mellan antalet sårbarheter och utveckling av metodik, programmeringsspråk och säkerhetsverktyg väcker exempelvis följande fråga som behöver fortsatt forskning. *Varför verkar inte mängden sårbarheter avta med den ökade användningen av väletablerade metoder för säker mjukvaruutveckling?*

Det finns många olika områden inom mjukvarusäkerhet som behöver fortsatt forskning. Utöver ovanstående områden har vi exempelvis sett följande intressanta frågeställningar. *Hur kan utvecklingsteamen säkerställa att*

buggrättningar inte inför nya sårbarheter? Hur bra är AI-bottar (såsom ChatGPT) jämfört med mänskliga utvecklare avseende säker utveckling när de får rätt förutsättningar? Hur kan säkerhet bli en större prioritet i svaren på programmeringsfrågor i webbforum såsom Slashdot och Reddit och hur kan de bästa (säkraste) svaren bli mer synliga?

6.2 Programmeringsspråk

Utifrån resultaten i avsnitt 5.2 har vi identifierat följande forskningsbehov.

Valet av programmeringsspråk kan potentiellt ha stor effekt på säkerheten i en produkt. Numera finns det en uppsjö av olika språk med olika egenskaper som kan påverka produktens kvalitet. Följande frågor är exempel på relevant forskning angående språkval. *Hur kan utvecklare välja programmeringsspråk för bästa säkerhet i sin domän? När är det lämpligt att kombinera olika språk för att stärka säkerheten samtidigt som produktiviteten hålls hög? Vilka kombinationer av språk passar bäst ihop ur säkerhetssynvinkel?*

Att hålla kvar vid äldre, men osäkrare, programmeringsspråk kan vara bekvämt jämfört med att byta till nyare, säkrare språk. Tyvärr övertrumfar fortfarande bekvämlighet säkerhet, trots att det osäkrare språket kan leda till fler sårbarheter. Ekonomiska förutsättningar, brist på erfarenhet av det nyare språket, vidareutveckling av befintliga produkter skrivna i det äldre språket och den mänskliga motviljan till förändring kan vara potentiella faktorer som bromsar. Samtidigt kan ett byte av språk ge upphov till sårbarheter om förutsättningarna är fel. Följande frågor är exempel på relevant forskning. *I vilken utsträckning förekommer språkkonservatism inom mjukvarubranschen, det vill säga att utvecklare väljer invanda programmeringsspråk snarare än språk som är lämpliga ur säkerhetssynvinkel? Vilka effekter får byte av programmeringsspråk på den säkerhetsmässiga kvaliteten, exempelvis då sämre språkvana leder till lägre kvalitet eller när sårbarheter undviks genom det säkrare språket?*

Väletablerade programmeringsspråk såsom C/C++ och Java används i stor utsträckning och kommer säkerligen även fortsatt användas mycket. Med utgångspunkten att språken kommer användas mycket även i framtiden så finns stor potential för förbättringar genom att stärka språkens säkerhetsegenskaper. Försvarsmakten skulle till exempel kunna vara delaktig i standardiseringsarbete för att vara med och förbättra språken. Detta väcker följande forskningsfråga. *Hur kan befintliga programmeringsspråk utvecklas för att bli mer säkra?*

Det finns ett antal nyare programmeringsspråk som har skapats med större hänsyn till säkerhetsaspekterna. Användningen av sådana språk ökar men det är fortfarande oklart vilka säkerhetseffekter den ökande användningen får. Följande frågor är exempel på relevant forskning om säkrare språk. *Vilka positiva och negativa effekter kan förväntas på säkerheten när stora operativsystem går över till att (delvis) använda Rust? Hur kan organisationer fås att våga satsa på nya*

säkrare språk trots ovisshet om kompetensförsörjning? Hur kan språken utvecklas vidare för att ytterligare förbättra säkerheten? Hur kan AI stödja arbetet med att utveckla nya säkrare språk?

6.3 Sårbarhetsupptäckt

Utifrån resultaten i avsnitt 5.3 har vi identifierat följande forskningsbehov.

En viktig del i arbetet med att skapa säkrare mjukvara är att underlätta upptäckt av potentiella brister och sårbarheter under utvecklingsfasen. I detta ingår tillgång till effektivt verktygsstöd med hög användbarhet i praktiskt arbete. En viktig aspekt är att verktygen inte bör överösa utvecklaren med varningar som inte går att undvika, t.ex. på grund av att de beror på något inneboende i språket eller mjukvaruspecifikationen. En forskningsfråga kopplat till verktygens användbarhet är följande. *Hur och när kan utvecklings- och testverktyg ge feedback till utvecklarna för att påskynda åtgärder av sårbarheter?*

I dagsläget är formella metoder ofta svåra att använda i praktiken. Svårhanterade, komplicerade formella specifikationer passar många gånger inte in i agila och föränderliga utvecklingsprojekt. En forskningsfråga inom detta är följande. *Hur kan specifikationer skrivas för att (formella) säkerhetstester ska ha något tillräckligt rigoröst att jämföra koden med?*

Nya verktyg och ändrad användning av befintliga metoder kan potentiellt hantera olika typer av brister och sårbarheter. Här finns ett stort antal forskningsfrågor som behöver fortsatt arbete. Exempel på forskningsfrågor relaterat till verktyg inkluderar följande. *Hur kan verktyg för upptäckt av sårbarheter i mjukvara dra nytta av verktyg för upptäckt av sårbara mjukvaror (i system) och tvärtom? Hur effektiv kan en sårbarhetssökande AI bli när den är upplärd på en massiv sårbarhetsdatamängd (för ett eller flera språk)? Finns det behov av mer riktade varianter av statiska testverktyg för att hitta en viss typ av sårbarhet med rimligt låga falska positiva och falska negativa? Hur kan sådana verktyg tas fram på ett effektivt sätt?*

Samtidigt som kodgranskning är en värdefull metod för att kvalitetssäkra mjukvaror så har forskningen visat att det finns utmaningar inom granskning av källkod. Följande utgör några exempel på forskningsfrågor relaterade till kodgranskning. *Vad gör att en person blir en bra kodgranskare?* En sådan studie bör undersöka frågan ur ett brett spektrum av potentiella faktorer, såsom kunskap, erfarenhet, utbildning och säkerhetskultur. *Hur kan manuella kodgranskare bli bättre på att upptäcka sårbarheter som är svårupptäckta vid manuell kodgranskning, såsom möjligheten till buffertöverskridningar?*

7 Referenser

Allodi, L., Cremonini, M., Massacci, F., & Shim, W. (2020). Measuring the accuracy of software vulnerability assessments: experiments with students and professionals. Empirical Software Engineering, 25, 1063-1094.

Andersson, C. (2001). Säkerhetsbrister i Javascript. FOI-R--0124--SE. Totalförsvarets forskningsinstitut.

Anwar, A., Abusnaina, A., Chen, S., Li, F., & Mohaisen, D. (2021). Cleaning the NVD: Comprehensive quality assessment, improvements, and analyses. IEEE Transactions on Dependable and Secure Computing, 19(6), 4255-4269.

Assal, H., & Chiasson, S. (2019). 'Think secure from the beginning' A Survey with Software Developers. In Proceedings of the 2019 CHI conference on human factors in computing systems (pp. 1-13).

Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J. D., & Penix, J. (2008). Using static analysis to find bugs. IEEE software, 25(5), 22-29.

Baca, D., Petersen, K., Carlsson, B., & Lundberg, L. (2009). Static code analysis to detect software security vulnerabilities-does experience matter?. In 2009 International Conference on Availability, Reliability and Security (pp. 804-810). IEEE.

Bildsten, C., Cohen, M., Eidenskog, D., & Rodhe, I. (2018). Praktisk mjukvaruverifiering med formella metoder. Ett hjälpmedel i granskningsprocessen. FOI-R--4642--SE. Totalförsvarets forskningsinstitut.

Bojanova, I., Black, P. E., Yesha, Y., & Wu, Y. (2016). The bugs framework (BF): A structured approach to express bugs. In 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS) (pp. 175-182). IEEE.

Bosu, A., Carver, J. C., Hafiz, M., Hilley, P., & Janni, D. (2014). Identifying the characteristics of vulnerable code changes: An empirical study. In Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering (pp. 257-268).

Braz, L., Fregnan, E., Çalikli, G., & Bacchelli, A. (2021). Why Don't Developers Detect Improper Input Validation?'; DROP TABLE Papers;--. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (pp. 499-511). IEEE.

Camilo, F., Meneely, A., & Nagappan, M. (2015). Do bugs foreshadow vulnerabilities? a study of the chromium project. In 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (pp. 269-279). IEEE.

Check Point. (2021). The Numbers Behind Log4j Vulnerability CVE-2021-44228. Publicerad 21 december 2021. https://blog.checkpoint.com/security/the-numbers-behind-a-cyber-pandemic-detailed-dive/

CISA. (2020). Software Development. NICE Framework. National Initiative for Cybersecurity Careers and Studies. Besökt 7 juni 2023. Cybersecurity and Infrastructure Security Agency. https://niccs.cisa.gov/workforce-development/nice-framework

CISA. (2023). 2022 Top Routinely Exploited Vulnerabilities. CISA Cybersecurity Advisory. Publicerad 3 augusti 2023. https://www.cisa.gov/news-events/cybersecurity-advisories/aa23-215a

CISA, NSA, FBI, ACSC, NCSC-UK, CCCS, BSI, NCSC-NL, CERT NZ, & NCSC-NZ. (2023). Shifting the Balance of Cybersecurity Risk: Principles and Approaches for Security-by-Design and -Default.

Claburn, T. (2023). Microsoft is busy rewriting core Windows code in memory-safe Rust. The Register. Publicerad 27 april 2023. https://www.theregister.com/2023/04/27/microsoft_windows_rust/

Clark, S., Frei, S., Blaze, M., & Smith, J. (2010). Familiarity breeds contempt: The honeymoon effect and the role of legacy code in zero-day vulnerabilities. In Proceedings of the 26th annual computer security applications conference (pp. 251-260).

Cohen, M., Rodhe, I., & Löfvenberg, J. (2016). White-box fuzzing. FOI-R--4329--SE. Totalförsvarets forskningsinstitut.

Cox, R., Griesemer, R., Pike, R., Taylor, I. L., & Thompson, K. (2022). The Go programming language and environment. Communications of the ACM, 65(5), 70-78.

Croft, R., Xie, Y., Zahedi, M., Babar, M. A., & Treude, C. (2022). An empirical study of developers' discussions about security challenges of different programming languages. Empirical Software Engineering, 27, 1-52.

Davila, N., & Nunes, I. (2021). A systematic literature review and taxonomy of modern code review. Journal of Systems and Software, 177, 110951.

Davis, N., Humphrey, W., Redwine, S. T., Zibulski, G., & McGraw, G. (2004). Processes for producing secure software. IEEE Security & Privacy, 2(3), 18-25.

di Biase, M., Bruntink, M., & Bacchelli, A. (2016). A security perspective on code review: The case of chromium. In 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM) (pp. 21-30). IEEE.

Dissanayake, N., Jayatilaka, A., Zahedi, M., & Babar, M. A. (2022). Software security patch management-A systematic literature review of challenges,

approaches, tools and practices. Information and Software Technology, 144, 106771.

Easterly, J. (2023). CISA Director Easterly Remarks at Carnegie Mellon University. Unsafe at Any CPU Speed: The Designed-in Dangers of Technology and What We Can Do About It. Besökt 24 november 2023. https://www.cisa.gov/cisa-director-easterly-remarks-carnegie-mellon-university

Fulton, K. R., Chan, A., Votipka, D., Hicks, M., & Mazurek, M. L. (2021). Benefits and drawbacks of adopting a secure programming language: rust as a case study. In Symposium on Usable Privacy and Security.

Gasiba, T. E., Lechner, U., Pinto-Albuquerque, M., & Mendez, D. (2021). Is secure coding education in the industry needed? An investigation through a large scale survey. In 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET) (pp. 241-252). IEEE.

Genge, B., & Enachescu, C. (2016). ShoVAT: Shodan-based vulnerability assessment tool for Internet-facing services. Security and communication networks, 9(15), 2696-2714.

Gkortzis, A., Feitosa, D., & Spinellis, D. (2021). Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. Journal of Systems and Software, 172, 110653.

Gorbenko, A., Romanovsky, A., Tarasyuk, O., & Biloborodov, O. (2017). Experience report: Study of vulnerabilities of enterprise operating systems. In 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE) (pp. 205-215). IEEE.

Graham-Cunning J. (2021). Inside the Log4j2 vulnerability (CVE-2021-44228). Cloudflare. Publicerad 12 oktober 2021. https://blog.cloudflare.com/inside-the-log4j2-vulnerability-cve-2021-44228/

Green, M., & Smith, M. (2016). Developers are not the enemy!: The need for usable security apis. IEEE Security & Privacy, 14(5), 40-46.

Gueye, A., Galhardo, C. E., Bojanova, I., & Mell, P. (2021). A decade of reoccurring software weaknesses. IEEE Security & Privacy, 19(6), 74-82.

Haney, J. M., Theofanos, M., Acar, Y., & Prettyman, S. S. (2018). "We make it a big deal in the company": Security Mindsets in Organizations that Develop Cryptographic Products. In SOUPS@ USENIX Security Symposium (pp. 357-373).

Holm, H., Sommestad, T., Almroth, J., & Persson, M. (2011). A quantitative evaluation of vulnerability scanning. Information Management & Computer Security, 19(4), 231-247.

Holm, H., Bengtsson, J., Löfvenberg, J., Persson, M., & Sommestad, T. (2014). Moving Target Defense. En kartläggning av forskningsbidrag. FOI-R--3942--SE. Totalförsvarets forskningsinstitut.

Holm, H., & Afridi, K. K. (2015). An expert-based investigation of the common vulnerability scoring system. Computers & Security, 53, 18-30.

Hosseinzadeh, S., Rauti, S., Laurén, S., Mäkelä, J. M., Holvitie, J., Hyrynsalmi, S., & Leppänen, V. (2018). Diversification and obfuscation techniques for software security: A systematic literature review. Information and Software Technology, 104, 72-93.

ISO/IEC 21827. (2008). Information technology — Security techniques — Systems Security Engineering — Capability Maturity Model (SSE-CMM). Second edition.

ISO/IEC 24772-1:2019. (2019). Programming languages — Guidance to avoiding vulnerabilities in programming languages — Part 1: Language-independent guidance.

Johnson, P., Gorton, D., Lagerström, R., & Ekstedt, M. (2016). Time between vulnerability disclosures: A measure of software product vulnerability. Computers & Security, 62, 278-295.

Joshi, C., Singh, U. K., & Tarey, K. (2015). A review on taxonomies of attacks and vulnerability in computer and network system. International Journal, 5(1).

Karlzén, H. (2019). Cyberoperationers attribution, tillvägagångssätt och sofistikation. FOI-R--4834--SE. Totalförsvarets forskningsinstitut.

Khoury, R., Avila, A. R., Brunelle, J., & Camara, B. M. (2023). How Secure is Code Generated by ChatGPT?. arXiv preprint arXiv:2304.09655.

Kochhar, P. S., Wijedasa, D., & Lo, D. (2016). A large scale study of multiple programming languages and code quality. In 2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER) (Vol. 1, pp. 563-573). IEEE.

Krsul, I. V. (1998). Software vulnerability analysis. Purdue University.

Kudriavtseva, A., & Gadyatskaya, O. (2022). Secure Software Development Methodologies: A Multivocal Literature Review. arXiv preprint arXiv:2211.16987.

Kuhn, R., Raunak, M., & Kacker, R. (2017). It doesn't have to be like this: Cybersecurity vulnerability trends. IT professional, 19(6), 66-70.

Li, X., Moreschini, S., Zhang, Z., Palomba, F., & Taibi, D. (2023). The anatomy of a vulnerability database: A systematic mapping study. Journal of Systems and Software, 111679.

Le, T. H. M., Croft, R., Hin, D., & Babar, M. A. (2021). A large-scale study of security vulnerability support on developer q&a websites. In Evaluation and assessment in software engineering (pp. 109-118).

Liu, B., Shi, L., Cai, Z., & Li, M. (2012). Software vulnerability discovery techniques: A survey. In 2012 fourth international conference on multimedia information networking and security (pp. 152-156). IEEE.

Löfvenberg, J., Sommestad, T., & Bildsten, C. (2019). Automatisk attackkodsgenerering. En skanning av forskningsfronten. FOI-R--4737--SE. Totalförsvarets forskningsinstitut.

Lomio, F., Iannone, E., De Lucia, A., Palomba, F., & Lenarduzzi, V. (2022). Just-in-time software vulnerability detection: Are we there yet?. Journal of Systems and Software, 111283.

McGraw, G. (2006). Software Security: Building Security In. Addison-Wesley.

Meng, N., Nagy, S., Yao, D., Zhuang, W., & Argoty, G. A. (2018). Secure coding practices in java: Challenges and vulnerabilities. In Proceedings of the 40th International Conference on Software Engineering (pp. 372-383).

Moriuchi, P., & Ladd, B. (2017). China’s ministry of state security likely influences national network vulnerability publications. Recorded future blog. Publicerad 16 november 2017. https://www.recordedfuture.com/chinese-mss-vulnerability-influence

Nadi, S., Krüger, S., Mezini, M., & Bodden, E. (2016). Jumping through hoops: Why do Java developers struggle with cryptography APIs?. In Proceedings of the 38th International Conference on Software Engineering (pp. 935-946).

Naiakshina, A., Danilova, A., Gerlitz, E., Von Zezschwitz, E., & Smith, M. (2019). " If you want, I can store the encrypted password" A Password-Storage Field Study with Freelance Developers. In Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (pp. 1-12).

Oliveira, D., Rosenthal, M., Morin, N., Yeh, K. C., Cappos, J., & Zhuang, Y. (2014). It's the psychology stupid: how heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots. In Proceedings of the 30th Annual Computer Security Applications Conference (pp. 296-305).

Oyetoyan, T. D., Milosheska, B., Grini, M., & Soares Cruzes, D. (2018). Myths and facts about static application security testing tools: an action research at Telenor digital. In Agile Processes in Software Engineering and Extreme Programming: 19th International Conference, XP 2018, Porto, Portugal, May 21–25, 2018, Proceedings 19 (pp. 86-103). Springer International Publishing.

Palme, J. (1969). What is a good programming language?. FOA C 8231-64(11). Försvarets forskningsanstalt.

Purdy, K. (2023). Two core Unix-like utilities, sudo and su, are getting rewrites in Rust. Ars Technica. Publicerad 1 maj 2023. https://arstechnica.com/information-technology/2023/05/two-core-unix-like-utilities-sudo-and-su-are-getting-rewrites-in-rust/

Quirolgico, S., Voas, J., Karygiannis, T., Michael, C., & Scarfone, K. (2015). NIST Special Publication 800-163. NIST Special Publication, 800, 163.

Rodhe, I., & Karresand, M. (2015). Overview of formal methods in software engineering. FOI-R--4156--SE. Totalförsvarets forskningsinstitut.

Rodhe, I., & Karresand, M. (2016). Verktyg för att åstadkomma pålitlig programvara. FOI-R--4290--SE. Totalförsvarets forskningsinstitut.

Ryan, I., Roedig, U., & Stol, K. J. (2021). Understanding developer security archetypes. In 2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS) (pp. 37-40). IEEE.

Schryen, G. (2011). Is open source security a myth?. Communications of the ACM, 54(5), 130-140.

SEI-CMU. (2022). Rule 04. Integers (INT). Software Engineering Institute. Carnegie Mellon University. Besökt 24 november 2023. https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152052

Smith, J., Do, L. N., & Murphy-Hill, E. (2020). Why can't johnny fix vulnerabilities: A usability evaluation of static analysis tools for security. In Proceedings of the Sixteenth Symposium on Usable Privacy and Security.

Sommestad, T., & Holm, H. (2017). Publika angreppskoder och intrångssignaturer. Kvantitativa tester av träffsäkerhet. FOI-R--4499--SE. Totalförsvarets forskningsinstitut.

Souppaya, M., Scarfone, K., & Dodson, D. (2022). Secure software development framework (ssdf) version 1.1. NIST Special Publication, 800, 218.

Stoep, J. V. (2022). Memory Safe Languages in Android 13. Google Security Blog. Publicerad 1 december 2022. https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html

Tahaei, M., & Vaniea, K. (2019). A survey on developer-centred security. In 2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW) (pp. 129-138). IEEE.

Tøndel, I. A., & Cruzes, D. S. (2022). Continuous software security through security prioritisation meetings. Journal of Systems and Software, 194, 111477.

Tøndel, I. A., Cruzes, D. S., Jaatun, M. G., & Sindre, G. (2022). Influencing the security prioritisation of an agile software development project. Computers & Security, 118, 102744.

Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Gall, H. C., & Zaidman, A. (2020). How developers engage with static analysis tools in different contexts. Empirical Software Engineering, 25, 1419-1457.

Williams, M. A., Dey, S., Barranco, R. C., Naim, S. M., Hossain, M. S., & Akbar, M. (2018). Analyzing evolving trends of vulnerabilities in national vulnerability database. In 2018 IEEE International Conference on Big Data (Big Data) (pp. 3011-3020). IEEE.

Witschey, J., Zielinska, O., Welk, A., Murphy-Hill, E., Mayhorn, C., & Zimmermann, T. (2015). Quantifying developers' adoption of security tools. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (pp. 260-271).

Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., & Bairavasundaram, L. (2011). How do fixes become bugs?. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (pp. 26-36).

Zimmermann, T., Nagappan, N., & Williams, L. (2010). Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In 2010 Third international conference on software testing, verification and validation (pp. 421-428). IEEE.

FOI är en huvudsakligen uppdragsfinansierad myndighet under Försvarsdepartementet. Kärnverksamheten är forskning, metod- och teknikutveckling till nytta för försvar och säkerhet. Organisationen har cirka 1000 anställda varav ungefär 800 är forskare. Detta gör organisationen till Sveriges största forskningsinstitut. FOI ger kunderna tillgång till ledande expertis inom ett stort antal tillämpningsområden såsom säkerhetspolitiska studier och analyser inom försvar och säkerhet, bedömning av olika typer av hot, system för ledning och hantering av kriser, skydd mot och hantering av farliga ämnen, IT-säkerhet och nya sensorers möjligheter.

FOI
Totalförsvarets forskningsinstitut
164 90 Stockholm

Tel: 08-55 50 30 00
Fax: 08-55 50 31 00

www.foi.se