Mjukvarusårbarheter och skyddstekniker

Analys av Pythonmjukvara

Casper Jensen, David Ekman, Henrik Karlzén,
Daniel Eidenskog och Jerry Falkcrona

FOI-R--5829--SE

December 2025

FOI

Casper Jensen, David Ekman, Henrik Karlzén,
Daniel Eidenskog och Jerry Falkcrona

Mjukvarusårbarheter och skyddstekniker

Analys av Pythonmjukvara

Titel	Mjukvarusårbarheter och skyddstekniker – Analys av Pythonmjukvara
Title	Software vulnerabilities and mitigations – Analysis of Python software
Rapportnr/Report no	FOI-R--5829--SE
Månad/Month	December
Utgivningsår/Year	2025
Antal sidor/Pages	28
ISSN	1650-1942
Uppdragsgivare/Client	Försvarsmakten
Forskningsområde	Cyberförsvar och cybersäkerhet
FoT-område	Operationer i cyberdomänen
Projektnr/Project no	E38556
Godkänd av/Approved by	Emil Hjalmarson
Ansvarig avdelning	Cyberförsvar och ledningsteknik

Bild/Cover: Shutterstock

Sammanfattning

Mjukvara har en central roll i många verksamheter. Samtidigt har det visat sig att det är mycket svårt att utveckla mjukvara utan sårbarheter. Rapporten beskriver resultatet av en studie som undersöker vilka som är de vanligaste sårbarhetstyperna i kod skriven i programmeringsspråket Python. Studien undersöker också vilka som är de vanligaste skyddsteknikerna för att åtgärda sårbarheterna. Studien presenterar en taxonomi för skyddstekniker samt sambandet mellan sårbarhetstyper och skyddstekniker.

Resultaten visar att de vanligast förekommande sårbarhetstyperna är bristande hantering av indata samt bristande autentisering och åtkomstkontroll. Resultaten visar även att skyddstekniken indatavalidering är vanligast vid åtgärdandet av sårbarheter.

Nyckelord: sårbarheter, sårbarhetstyper, skyddstekniker, Python

Summary

Software plays a central role in many businesses. At the same time, it turns out that it is very hard to develop software that has no vulnerabilities. This report describes the results of a study that examines the most common vulnerability types in code written in the Python programming language. The study also investigates the most common mitigations for the vulnerabilities. The study presents a taxonomy for mitigations and the connection between vulnerability types and mitigations.

The results show that the most frequent vulnerability types are insufficient handling of input data and insufficient authentication and access control. The results also show that the most common mitigation is input validation.

Keywords: vulnerabilities, vulnerability types, mitigations, Python

Innehållsförteckning

1 Inledning 7
1.1 Syfte och mål 7
1.2 Avgränsningar 8
2 Bakgrund 9
2.1 Sårbarheter och skyddstekniker 9
2.2 Ett belysande exempel 9
3 Metod 11
3.1 Val av data 11
3.2 Extrahering av data 11
3.3 Framtagande av en taxonomi 12
3.4 Analys av extraherade data 13
4 Resultat 14
4.1 De vanligaste sårbarhetstyperna 14
4.2 De vanligaste skyddsteknikerna 15
4.3 Samband mellan skyddstekniker och sårbarhetstyper 16
5 Diskussion 20
5.1 Praktisk påverkan 20
5.2 Forskningsmässiga begränsningar 21
5.3 Framtida forskningsmöjligheter 21
6 Slutsatser 24
6.1 Vilka sårbarhetstyper är vanligast? 24
6.2 Vilka skyddstekniker är vanligast? 24
6.3 Vilka samband finns det mellan sårbarhetstyperna och skyddsteknikerna? 25
7 Referenser 26
Bilaga A: Taxonomi av skyddstekniker 27

1 Inledning

Mjukvara har en central roll i såväl civil som militär verksamhet. Denna centrala roll gör det viktigt att upprätthålla en godtagbar säkerhetsnivå hos mjukvarorna. Samtidigt har det visat sig svårt att utveckla mjukvara utan sårbarheter. En anledning till detta är förmodligen att programmeringsspråk är avsedda att kunna skapa i stort sett vilken mjukvara som helst. Om de val som görs vid utvecklingen inte är perfekta kan säkerhetsmässiga sårbarheter införas.

Olika programmeringsspråk har i sin tur olika begränsningar som påverkar hur mjukvaran kan utformas. Vilka sårbarheter som typiskt uppstår beror därmed även på vilket språk som används (Karlzén m.fl., 2023). Det vanligaste språket är Python, enligt återkommande rankningar av IEEE Spectrum (Cass, 2024). Python används på alla större operativsystem för många typer av mjukvara, däribland webbtjänster och maskininlärning (Bogaerts m.fl., 2024). Det finns därmed ett särskilt intresse att studera sårbarheter i Pythonmjukvara.

Med tanke på hur vanlig Pythonmjukvara är, och hur många mjukvaror som Försvarsmakten använder, blir det uppenbart att det svenska försvaret behöver kunskap om sårbarheter och skyddstekniker för Pythonmjukvara. Exempelvis behöver Försvarsmakten veta vilka typer av sårbarheter de kan förvänta sig att hitta vid granskning av Pythonkod eller vid egenutveckling av sådan kod. Dessutom behövs kunskap om hur sådana sårbarheter kan lösas. Det är också väsentligt att etablera sambanden mellan sårbarheterna och skyddsteknikerna samt förstå i vilken mån detta är speciellt för mjukvara skriven i just Python. Det finns många idéer på området samt rentav vedertagna vägledningar om vilka sårbarheter som uppstår när och hur sårbarheterna bör rättas till. Men dessa vedertagna vägledningar saknar generellt sett stöd. Här kan rapporten bidra.

Denna rapport kartlägger sårbarheter i mjukvara skriven i Python. Rapporten beskriver också vilka skyddstekniker som används för att åtgärda olika typer av sårbarheter.

1.1 Syfte och mål

Syftet med studien är att öka kunskapen om det bästa sättet att åtgärda olika typer av mjukvarusårbarheter. Målet med studien är att undersöka vilka skyddstekniker som är vanligast i praktiken för att åtgärda vanliga sårbarheter i mjukvara skriven i Python. Vad gäller skyddsteknikerna är det inte uppenbart att det finns en enda som är rätt för alla sårbarheter av en viss typ, eller tvärtom att en viss skyddsteknik bara kan åtgärda sårbarheter av en viss typ. Sambanden mellan sårbarheter och skyddstekniker är därmed också relevanta att undersöka.

Målet skiljer sig från syftet på två viktiga sätt. För det första tar målet en deskriptiv ansats, eftersom vanliga skyddstekniker undersöks utan att uttala sig

om vilka skyddstekniker som är bäst. För det andra begränsar sig målet till mjukvaror skrivna i Python. Framtida forskning kan komplettera detta för mjukvaror i andra språk samt försöka överbrygga gapet från mest vanliga skyddsteknik till mest önskvärd skyddsteknik.

För att uppnå målet besvarar rapporten följande forskningsfrågor:

1. Vilka sårbarhetstyper är vanligast?
2. Vilka skyddstekniker är vanligast?
3. Vilka samband finns det mellan sårbarhetstyperna och skyddsteknikerna?

Svaren ges utifrån en analys av en befintlig datamängd med sårbar och åtgärdad kod.

1.2 Avgränsningar

Rapporten gör inga praktiska tester av huruvida koden innehåller sårbarheter före åtgärd med skyddstekniker och att skyddsteknikerna faktiskt avlägsnar sårbarheterna. Rapporten bedömer inte heller huruvida åtgärderna har negativa bieffekter, eller om sårbarheterna istället hade kunnat hanteras genom åtgärder utanför mjukvaran, exempelvis i operativsystem eller övergripande arkitektur.

2 Bakgrund

I detta kapitel förklaras några av rapportens grundläggande begrepp: sårbarhet, sårbarhetstyp, åtgärd, kodrevidering och skyddsteknik. Först ges övergripande förklaringar. Därefter illustreras begreppen med ett exempel.

2.1 Sårbarheter och skyddstekniker

Mitre har en databas med information om sårbarheter i mjukvara.[1] Varje sårbarhet rör en specifik mjukvara och får ID-nummer enligt Common Vulnerabilities and Exposures (CVE). Dessa sårbarheter delas in i typer, av snarlika sårbarheter, enligt Common Weakness Enumeration (CWE). Det finns hundratals olika kategorier i CWE och bland kategorierna finns exempelvis olika typer av buffertöverskridningar (Karlzén m.fl., 2023).

Resultaten i denna rapport analyserar sårbarheter och sårbarhetstyper. Dessutom analyseras hur sårbar kod revideras för att åtgärda sårbarheten. Kodrevideringen, även benämnd åtgärd, applicerar en viss skyddsteknik. Dessa skyddstekniker utgör en kategorisering av åtgärder. Tillsammans bildar skyddsteknikerna en taxonomi som återges i Bilaga A. I detta bakgrundskapitel ges nu ett belysande exempel för att läsaren lättare ska förstå rapportens resultat.

2.2 Ett belysande exempel

Som exempel tas en kodrevidering för sårbarhetstypen *CWE-94 kodinjektion* och mer specifikt sårbarheten *CVE-2022-29216*.[2] Sårbarheten är i Pythonkod i TensorFlow, vilket är ett mjukvarubibliotek som används för att skapa olika modeller för maskininlärning. Kodrevideringen åtgärdar sårbarheten på ett sätt som innebär att skyddstekniken *indatavalidering* används. Skyddstekniken utgörs av att validera att alla indata följer förväntade format och regler.

I exemplet inträffar den bristande indatavalideringen när inmatade strängar ska konverteras från indataformat till programmets format, det vill säga parsas. Som ofta i Pythonkod används evalueringsfunktionen *eval(),* vilken är inbyggd i språket. Den reviderade koden byter denna funktion till det säkrare alternativet *ast.literal_eval(),* från biblioteket *ast* (Abstract Syntax Tree).[3] Denna funktion validerar inmatningen och ger felmeddelanden vid vissa typer av potentiellt skadliga indata och är designad för att läsa men inte exekvera kod. Figur 1

[1] https://cve.mitre.org

[2] https://www.cve.org/CVERecord?id=CVE-2022-29216 [hämtad 2025-11-26]

[3] https://github.com/tensorflow/tensorflow/commit/8b202f08d52e8206af2bdb2112a62fafbc546ec7 [hämtad 2025-06-02]

återger relevanta delar av kodrevideringen för den specifika mjukvaran. De kodrader i figuren som är rödmarkerade kommer att ersättas med de som är grönmarkerade.

```
535  538        Returns:
536  539          A dictionary that maps input keys to their values.
                @@ -545,8 +548,15 @@ def preprocess_input_exprs_arg_string(input_exprs_str):
545  548            raise RuntimeError('--input_exprs "%s" format is incorrect. Please follow'
546  549                               '"<input_key>=<python expression>"' % input_exprs_str)
547  550          input_key, expr = input_raw.split('=', 1)
548       -      # ast.literal_eval does not work with numpy expressions
549       -      input_dict[input_key] = eval(expr)  # pylint: disable=eval-used
          551  +      if safe:
          552  +        try:
          553  +          input_dict[input_key] = ast.literal_eval(expr)
          554  +        except:
          555  +          raise RuntimeError(
          556  +              f'Expression "{expr}" is not a valid python literal.')
          557  +      else:
          558  +        # ast.literal_eval does not work with numpy expressions
          559  +        input_dict[input_key] = eval(expr)  # pylint: disable=eval-used
550  560        return input_dict
```

Figur 1 Del av en kodrevidering där den osäkra funktionen *eval()* ersätts med alternativet *ast.literal_eval()* för kompatibla uttryck (läs: om variabeln *safe* är sann) i syfte att validera indata. Upphovsrätt enligt Apache-2.0-licens.

Att ett programmeringsspråk har inbyggda funktioner som inte är säkra kan verka märkligt. Att sådana funktioner finns kan bland annat bero på ogenomtänkt design av språket samt en avvägning mellan säkerhet och funktionalitet (Karlzén m.fl., 2023). Att sådana funktioner används i praktiken kan bland annat, som i det specifika exemplet ovan, bero på att koden bara var avsedd för testning men sedan lämnades kvar vid driftsättning.

Det finns också skäl att vara försiktig med den åtgärdande funktionen från ast-biblioteket. Dokumentationen till ast-funktionen avråder till att den används vid behandling av ej betrodda data. Detta beror på att en angripare då kan konstruera indata med hjälp av djupt nästlade strukturer för att på så sätt göra att mjukvaran kräver omfattande mängd arbetsminne.[4] Ett alternativ är att inte alls använda eval-funktioner i sådana situationer. Istället föreslår Mitre varianter som att applicera heltalskonvertering i ett try-except-block när programmet förväntar sig heltal i indatasträngar. Om användarens inmatning inte är siffror kommer då felmeddelandet ValueError uppstå.

[4] https://cwe.mitre.org/data/definitions/95.html [hämtad 2025-06-02]

3 Metod

I de följande avsnitten beskrivs forskningsmetoden. Denna metod används för att besvara forskningsfrågorna och då specifikt för Pythonkod. Frågorna handlar om vilka de vanligaste sårbarhetstyperna är, vilka de vanligaste skyddsteknikerna är för att möta dessa sårbarhetstyper samt vilka samband som finns mellan skyddstekniker och sårbarhetstyper.

3.1 Val av data

FOI-memot av Jensen m.fl. (2024) sammanställde datamängder med sårbarheter, vilka är en sorts dataset eller databaser med kod och sårbarheter. Den enda av datamängderna som det framgår täcker Pythonkod är Reposvul. Den datamängden passar därför denna studie bra. Reposvul skapades av kinesiska forskare (Wang m.fl., 2024) och finns på github.com/Eshe0922/ReposVul. Reposvul sammanställer dokumenterade sårbarheter i en rad olika mjukvaror samt länkar till kodrevideringar som åtgärdar sårbarheterna. Länkarna leder till Github, där mjukvarornas källkod finns öppet tillgänglig tillsammans med ändringshistorik, vilken beskriver hur koden såg ut före och efter sårbarheten åtgärdades.

Vid skapandet av Reposvul kontrollerade Wang m.fl. (2024) att inhämtade data (exempelvis sårbarhetstyp och länk till kod) verkligen innehöll sårbarheter och var kopplade till åtgärdande kodrevideringar. Deras kontroll skedde genom att mata in data till både ett AI-verktyg och till två klassiska verktyg för statisk kodanalys (SAT). När AI och SAT båda intygade att det rörde sig om sårbarheter och kodrevideringar så togs data med i Reposvul. Stickprov gjordes också där några forskare med erfarenhet av sårbarhetsupptäckt gick igenom källkoderna manuellt. Denna rapport har inte upprepat dessa kontroller.

Rapportens analys utgick från Reposvul för data om sårbar mjukvara. Detta kombinerades med de länkade kodrevideringarna från Github. Tillsammans utgörs rapportens datamängd av dessa data från Reposvul och Github.

3.2 Extrahering av data

Data extraherades enbart för Python. Reposvul anger bland annat vilka språk de olika mjukvarorna är skrivna i. Detta kan vara mer än ett språk. Wang m.fl. (2024) anger att Reposvul innehåller 6134 sårbarheter, varav de flesta är i mjukvara skriven i C++. 1152 sårbarheter anges vara i Python (indelade i 159 sårbarhetstyper). Författarna kontrollerade på Github vilka av dessa Pythonsårbarheter och kodrevideringar som verkligen rör Python. Det visade sig att Reposvul felaktigt klassificerat vissa av sårbarheterna och att de i själva verket är i ett annat språk än Python. Dessutom fanns en del dubbletter. Efter

exkludering av dubbletter och felklassificeringar kvarstod 746 sårbarheter i Python.

Från Reposvul extraherades sårbarhetstyp (CWE-ID), sårbarhet (CVE-ID) samt commit-länk till respektive mjukvaras Github-projektsida. Från Github extraherades själva kodrevideringarna, inklusive skillnaden mot den tidigare koden.

Reposvul innehåller visserligen också viss information om kodrevideringarna, men inte lika fullständigt som Github. Varken Github eller Reposvul innehåller tydlig information om skyddsteknikerna (fältet CWE Solution var tomt för alla poster i Reposvul).

3.3 Framtagande av en taxonomi

Taxonomin togs fram med utgångspunkt i den kategorisering av sårbarhetstyper i Python som gjorts av Bogaerts m.fl. (2024) samt i sårbarhetstyper som framgick i Reposvul. Ingen av dessa existerande kategoriseringar kunde användas rakt av eftersom de inte hade samma fokus som analysen i denna rapport. Exempelvis inkluderas bara sårbarheter i Pythonkod snarare än sårbarheter som består av att Pythonkod anropar C-kod (som i Bogaerts buffertöverskridningar). Därtill utgick Bogaerts från sårbarheter snarare än skyddstekniker. Alla taxonomier kan dock påverkas av framtagarens bias och det är inte tänkt att taxonomin som används i denna rapport ska nyttjas för mer än analysen här.

Författarna genomförde en tematisk analys för att identifiera kategorier av skyddstekniker som bäst matchade sårbarhetstyperna. Resultatet blev taxonomin, vilken presenteras mer i Bilaga A.

Vissa sårbarhetstyper fick inte egna kategorier eller exkluderades vid framtagandet av taxonomin. Det gäller de typer som Bogaerts m.fl. (2024) kopplade till konfigurationsfel, designdefekter, buffertöverskridningar och numeriska fel. Därtill exkluderades andra minnesrelaterade sårbarheter, eftersom sådana sårbarheter uppstår i språk med mer manuell minneshantering som exempelvis C/C++. Kategorin Övrigt i Reposvul täckte dock en del av dessa exkluderade typer från Bogaerts.

Tabell 1 anger vilka skyddstekniker som ingår i taxonomin och varifrån de identifierats.

Tabell 1 Skyddstekniker och källa.

Skyddsteknik	**Källa**
Indatavalidering	Bogaerts
Indatasanering	Bogaerts
Säker deserialisering	Bogaerts
Utdatasanering	Bogaerts

Skyddsteknik	Källa
Parametriserade frågor	Reposvul
Åtkomstkontroll	Bogaerts
Verifiering av autentisering	Bogaerts
Sandboxing	Reposvul
Optimering av resursanvändning	Bogaerts
Ökade krav på säkrare autentiseringsuppgifter	Reposvul
Säkrare lagring av autentiseringsuppgifter	Reposvul
Användning av säkra kryptografiska parametrar	Bogaerts
Användning av konstanta operationer	Reposvul
Säker pakethantering och beroendehantering	Reposvul
Synkroniserad parallellkörning	Bogaerts
Övrigt	Reposvul

3.4 Analys av extraherade data

Analysen utgick från sårbarheter och sårbarhetstyper i Reposvul samt från relaterade kodrevideringar i Github. Varje kodrevidering analyserades och tilldelades en skyddsteknik enligt taxonomin.

I flera fall bedömdes en skyddsteknik täcka flera sårbarhetstyper. En sårbarhetstyp kunde även täckas av flera olika skyddstekniker. Därtill fanns det vissa sårbarheter som i Reposvul saknade koppling till en sårbarhetstyp. I hälften av dessa fall kunde en sådan sårbarhetstyp ändå identifieras, genom en kontroll av kodrevideringarna i Github. I den andra hälften hjälpte inte detta och de sårbarheterna placerades därför i kategorin Övrigt.

Tilldelning av skyddstekniker till sårbarheterna kan bero på den person som utför tilldelningen. Två av rapportförfattarna utförde tilldelningarna och testade först sin bedömningsförmåga genom ett pilottest. I pilottestet analyserades de tjugo första posterna i Reposvul. Bedömarna var helt samstämmiga i ungefär 60 % av fallen, delvis samstämmiga i drygt 20 % samt inte samstämmiga i knappt 20 %. All bristande samstämmighet diskuterades tills enighet nåddes för dessa sårbarheter och för att bättre positionera sig för resterande bedömningar av övriga sårbarheter.

4 Resultat

I detta kapitel redovisas resultatet från analysen av de 746 sårbarheterna (CVE:erna) från Reposvul samt relaterade kodrevideringar från Github.

4.1 De vanligaste sårbarhetstyperna

Antalet sårbarheter per sårbarhetstyp redovisas i Tabell 2. Namnen som anges är översatta från cwe.mitre.org. När det funnits flera alternativa namn har det kortaste namnet valts. I vissa fall har ingen översättning gjorts. Det gäller fall där den engelska termen bedömts vara mer vedertagen än en eventuell svensk översättning.

De flesta av sårbarhetstyperna rör antingen hanteringen av indata eller autentisering och åtkomstkontroll. Den vanligaste sårbarhetstypen är *CWE-79 cross-site scripting* och den täcker ungefär en sjundedel av antalet i tabellen. Detta följs av *CWE-20 bristande indatavalidering*. Den tredje vanligaste är en övrigt-grupp för de sårbarheter som saknar kategorisering. Ganska vanliga sårbarhetstyper är även *CWE-200 exponering av känslig information* samt *CWE-22 sökvägstraversering*. Överlag finns det många sårbarhetstyper med vardera några procent av det totala antalet. I tabellen syns inte heller de sårbarhetstyper med lägst antal i datamängden. Totalt finns 159 sårbarhetstyper, men i tabellen syns bara de som är kopplade till minst 10 sårbarheter.

Tabell 2 Antal sårbarheter (CVE:er) per sårbarhetstyp (CWE:er).

ID	**Namn**	**CVE:er**
CWE-79	Cross-site scripting (XSS)	87
CWE-20	Bristande indatavalidering	61
NVD-CWE-noinfo	–	55
CWE-200	Exponering av känslig information till ej autentiserad aktör	48
CWE-22	Sökvägstraversering	45
CWE-601	Öppen omdirigering	33
CWE-400	Okontrollerad resursanvändning	32
CWE-863	Felaktig auktorisering	24
CWE-352	Cross-site request forgery (CSRF)	22
CWE-287	Felaktig autentisering	22
CWE-77	Kommandoinjektion	21
CWE-770	Allokering av resurser utan gränser eller strypning (eng. throttling)	21
CWE-94	Kodinjektion	21
CWE-74	Injektion	19

ID	Namn	CVE:er
CWE-78	OS-kommandoinjektion	18
CWE-264	Behörigheter, privilegier och åtkomstkontroll	18
CWE-918	Server-side request forgery (SSRF)	15
CWE-89	SQL-injektion	15
CWE-1333	Ineffektiv reguljäruttryckskomplexitet	14
CWE-502	Deserialisering av icke-betrodda data	13
CWE-269	Olämplig privilegiehantering	12
CWE-532	Införande av känslig information i loggfil	11
CWE-444	HTTP-smuggling	11

4.2 De vanligaste skyddsteknikerna

Antal sårbarheter och sårbarhetstyper som kan åtgärdas av varje skyddsteknik presenteras i Tabell 3. I vissa fall är en sårbarhet kopplad till flera skyddstekniker, varför det totala antalet sårbarheter är större än 746. Att en skyddsteknik kan åtgärda en sårbarhetstyp innebär att skyddstekniken kan åtgärda minst en sårbarhet av den typen, men inte nödvändigtvis alla sårbarheter av den typen.

Tabell 3 Antal sårbarheter (CVE:er) och antal sårbarhetsyper (CWE:er) per skyddsteknik.

Skyddsteknik	CVE:er	CWE:er
Indatavalidering	202	14
Övrigt	136	–
Indatasanering	106	5
Utdatasanering	77	3
Verifiering av autentisering	71	4
Optimering av resursanvändning	54	3
Åtkomstkontroll	53	9
Användning av säkra kryptografiska parametrar	29	3
Ökade krav på säkrare autentiseringsuppgifter	13	1
Användning av konstanta operationer	11	2
Sandboxing	11	4
Säkrare lagring av autentiseringsuppgifter	10	1
Säker pakethantering och beroendehantering	10	1
Säker deserialisering	9	1
Parametriserade frågor	8	1
Synkroniserad parallellkörning	5	2

Indatavalidering är den skyddsteknik som används mest. Främst används skyddstekniken för sårbarheter med bristande indatavalidering (CWE-20) och

problem med sökvägstraversering (CWE-22). Till skillnad mot de flesta andra skyddstekniker används indatavalidering mot en bred mängd av olika sårbarhetstyper (14 CWE:er). Indatavalidering används även mot en bred mängd av sårbarheter (202 CVE:er). Snarlikt indatavalidering är *indatasanering*. Detta är den tredje vanligaste skyddstekniken och den täcker tillsammans med indatavalidering en dryg tredjedel av sårbarheterna. Andra, närbesläktade, skyddstekniker är säker *deserialisering* och *parametriserade* frågor. Av dessa kan noteras att säker *deserialisering* är av särskild betydelse för Python där standardbiblioteket pickle ofta används med följden att godtycklig kod kan exekveras.

Den näst vanligaste skyddstekniken är *övrigt*, vilket representerar en spretig mängd olika tekniker som inte enkelt kan klassificeras på något tydligt sätt. Den fjärde vanligaste skyddstekniken är *utdatasanering*, vilket kan ses som utdatas motsvarighet till skyddsteknikerna för indata.

Flera skyddstekniker rör autentisering, åtkomstkontroll och liknande. Tillsammans täcks ungefär en femtedel av sårbarheterna av skyddsteknikerna *verifiering av autentisering, åtkomstkontroll, ökade krav på säkrare autentiseringsuppgifter* samt *säkrare lagring av autentiseringsuppgifter*. Närbesläktade skyddstekniker är *användning av säkra kryptografiska parametrar* samt *användning av konstanta operationer*.

Vissa skyddstekniker rör prestanda och separering av beräkningar. Detta omfattar *optimering av resursanvändning, sandboxing* samt *synkroniserad parallellkörning*. Tillsammans står dessa skyddstekniker för knappt en tiondel.

En lite speciell skyddsteknik är *säker pakethantering och beroendehantering*. Den har inte så mycket med övriga skyddstekniker att göra och åtgärdar bara någon procent av alla sårbarheter. För Python kan detta dock vara anmärkningsvärt med tanke på den stora mängden tredjepartsbibliotek som finns tillgängliga att använda.

4.3 Samband mellan skyddstekniker och sårbarhetstyper

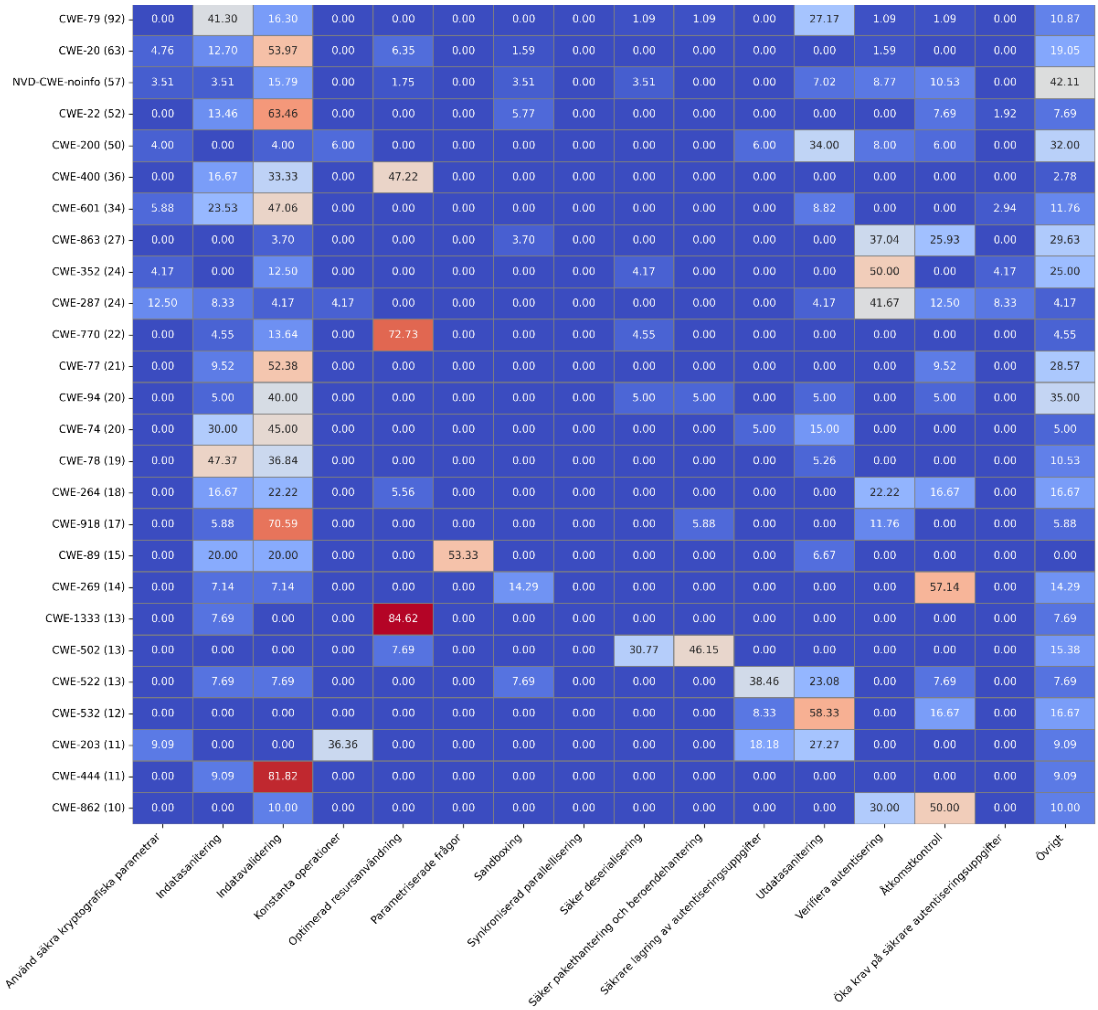
Detta avsnitt beskriver samband mellan skyddstekniker och sårbarhetstyper. Det rör sig om hur viktiga skyddstekniker är för en sårbarhetstyp, hur många sårbarhetstyper som kan åtgärdas av en viss skyddsteknik samt huruvida en del skyddstekniker är ovanliga men viktiga för enstaka sårbarhetstyper.

För vissa sårbarhetstyper finns det bara ett fåtal skyddstekniker som kan åtgärda sårbarheterna. I extremfall står en skyddsteknik för mer än hälften av åtgärderna för sårbarhetstypen. Sådana extremfall framgår av Tabell 4.

Tabell 4 Sårbarhetstyper (CWE:er) där en viss skyddsteknik är klart viktigast av alla skyddstekniker.

CWE-ID	**Sårbarhetstyp**	**Skyddsteknik**	**%**
CWE-1333	Ineffektiv reguljäruttrycks-komplexitet	Optimering av resursanvändning	85
CWE-444	HTTP-smuggling	Indatavalidering	82
CWE-770	Allokering av resurser utan gränser eller strypning (eng. throttling)	Optimering av resursanvändning	73
CWE-918	Server-side request forgery (SSRF)	Indatavalidering	71
CWE-22	Sökvägstraversering	Indatavalidering	63
CWE-532	Införande av känslig information i loggfil	Utdatasanering	58
CWE-269	Olämplig privilegiehantering	Åtkomstkontroll	57
CWE-20	Bristande indatavalidering	Indatavalidering	54
CWE-89	SQL-injektion	Parametriserade frågor	53
CWE-77	Kommandoinjektion	Indatavalidering	52

Urvalet i tabellen motsvarar de röda och mörkorange cellerna i Figur 2 nedan. I figuren är varje rad en sårbarhetstyp. Sårbarhetstypens CWE-ID anges först på raden. I parentes direkt efter står antal åtgärder som gäller för sårbarhetstypen. På raden längst ned listas de olika skyddsteknikerna. De färglagda fälten visar hur många procent av åtgärderna som hör till respektive skyddsteknik. Antalet åtgärder kan vara större än antal sårbarheter som kopplas till sårbarhetstypen (i avsnitt 4.1) eftersom en sårbarhet kan åtgärdas av flera skyddstekniker. Av utrymmesskäl har sårbarhetstyper med lägre antal åtgärder än tio utelämnats.



Figur 2 För varje sårbarhetstyp visas hur många procent av dess förekomster som åtgärdas av olika skyddstekniker. Ju blåare desto lägre procent står den skyddstekniken för vad gäller åtgärdandet av radens sårbarhetstyp; ju rödare desto högre procent.

För de flesta sårbarhetstyper finns det en handfull alternativa skyddstekniker som åtgärdar sårbarhetstypen. I ett fall finns ingen skyddsteknik som står för ens 30 procent av åtgärderna. Det gäller *CWE-264 behörigheter, privilegier och åtkomstkontroll*. Orsaken, som framgår av Mitres databas, är att denna sårbarhetstyp är tänkt som en bredare kategori vilket också får konsekvensen att

en bredare palett med skyddstekniker behöves för att adressera dessa sårbarheter.[5]

Indatavalidering är en vanlig skyddsteknik som adresserar många sårbarhetstyper. Den åtgärdar dessutom en stor andel av sårbarheterna för respektive sårbarhetstyp. Något mindre vanlig och viktig är *indatasanering*. Tillsammans har dessa två skyddstekniker åtgärder för alla sårbarhetstyper utom tre: *CWE-502 deserialisering av icke-betrodda data, CWE-532 införande av känslig information i loggfil* samt *CWE-203 observerbar diskrepans.* Dessutom förekommer övrigt-kategorin för alla sårbarhetstyper utom en.

Några skyddstekniker är ovanliga men trots allt viktiga för enstaka sårbarhetstyper i rapportens datamängd. Det gäller framförallt *optimerad resursanvändning*, men också *parametriserade frågor* samt *konstanta operationer*. Dessutom är *säker deserialisering* samt *säker pakethantering och beroendehantering* båda ovanliga som skyddstekniker men används mycket för just *CWE-502 deserialisering av icke-betrodda data*. Denna sårbarhetstyp uppstår bland annat på grund av inbyggda funktioner i vissa programmeringsspråk, däribland Python. Det är anmärkningsvärt att *säker pakethantering* inte är särskilt viktig som skyddsteknik, trots den stora mängden tredjepartsbibliotek som är tillgängliga för Python.

En hel del av skyddsteknikerna skyddar antingen mot skadlig inmatning eller mot att avslöja information vid utmatning. I vissa fall vill angripare utföra båda dessa led: först en skadlig inmatning och sedan få mjukvaran att utmata känslig information. Ett förenklat synsätt är då att skyddsteknikerna antingen bör åtgärda inmatningen eller utmatningen. Åtgärder av inmatningen är mycket vanligare än åtgärder av utmatningen, i rapportens datamängd. Ett undantag som inriktas på utmatningen är skyddstekniken *utdatasanering*. Den är särskilt viktig för *CWE-532 införande av känslig information i loggfil.* Anmärkningsvärt nog är den dessutom viktig för att åtgärda *CWE-79 Cross-site scripting (XSS)*, där även skyddstekniker mot inmatning är relevanta. Detta är anmärkningsvärt eftersom åtgärder av utmatningen tillsammans med åtgärder av inmatningen (vilket ofta är en angripares primära attackvektor) inte är vanligt för andra sårbarhetstyper.

[5] https://cwe.mitre.org/data/definitions/264.html [hämtad 2025-11-06]

5 Diskussion

I avsnitten som följer diskuteras resultatens praktiska påverkan, de forskningsmässiga begränsningarna samt framtida forskningsmöjligheter.

5.1 Praktisk påverkan

Rapporten visar att det finns sårbarheter som uppkommer oftare än andra. Det är rimligt att praktiskt ta höjd för dessa sårbarheter. Det är visserligen vedertaget att mjukvaruutveckling bör ske på ett visst sätt för att undvika sårbarheter. Men det saknas ofta empiriskt stöd för dessa vedertagna sätt. Denna rapport kan i huvudsak ge just stöd till sådana vedertagna sätt. Det rör sig dock om att undvika sårbarheternas fortsatta verkan, snarare än att förebygga sårbarheterna från att alls uppstå. Att undvika att sårbarheterna införs från början vore förstås det bästa, men har visat sig svårt. Denna rapport fokuserar istället på att identifiera vilka vanliga sårbarheter som brukar åtgärdas med vilka skyddstekniker. Det är förmodligen ändå en god idé att vara väl förtrogen med åtminstone de vanligaste skyddsteknikerna. Detta gäller både för utvecklare och för beställare samt till viss del driftorganisationer.

De vanligaste skyddsteknikerna är framförallt sådana som rör hanteringen av indata till mjukvaran. Samtidigt är inmatning typiskt en grundförutsättning för att ha nytta av sin mjukvara. Förutom att applicera programmeringsmässiga skyddstekniker vid inmatningen kan det vara värt att undersöka möjligheterna till att göra vissa övergripande förenklingar av mjukvaran, exempelvis vad gäller antalet funktioner och då minska angreppsytan. Att skyddstekniker vid utmatning är mindre vanliga kan tyda på att det är mindre lämpligt eller möjligt att införa skydd där. Det kan även tyda på en överdriven tro på att det som finns på insidan av mjukvaran går att lita på. Detta ger mer stöd till de idéer som handlar om att inte lita på mjukvaror och istället hitta arkitektoniska möjligheter att låsa in mjukvaror, eller att förlita sig på försvar på djupet såsom säkerhetskopiering och annan kontinuitetshantering. I rapportens data finns det också gott om skyddstekniker som rör autentisering och åtkomstkontroll. För dessa skyddstekniker kan det behövas en avvägning mellan vad den enskilda mjukvaran ska åstadkomma för skydd och vad som ska hanteras av kringliggande systemkomponenter såsom operativsystemet.

Resultaten visar att det ofta saknas ett uppenbart rätt val av skyddsteknik. Detta innebär ett behov av flexibilitet och förmåga att i utvecklingsteam diskutera sig fram till en lämplig åtgärd. Det finns också skyddstekniker som sällan används men som är kritiska för vissa sårbarhetstyper. Det är viktigt att sådana ovanligare konstruktioner inte faller i glömska bland granskare och utvecklare. Därtill finns det nästan inget i rapportens resultat som tyder på att skyddsteknikerna är

specifika för det enskilda programmeringsspråket, Python. Exempelvis är både deserialisering och säker pakethantering ovanliga för att åtgärda sårbarheter.

5.2 Forskningsmässiga begränsningar

Den forskning som presenteras i rapporten har två huvudsakliga begränsningar: data är inte fullt representativa och analyserna är inte helt objektiva.

Den ena begränsningen med forskningen i rapporten är att den baseras på en datamängd som kanske inte är representativ för alla sårbarheter och skyddstekniker. Exempelvis inkluderar den bara mjukvara med öppen källkod och då bara mjukvara publicerad på vissa ställen och skriven i Python samt där enstaka verktyg bekräftat att det rört sig om sårbarheter och kodrevideringar. Annan mjukvara kanske drabbas av andra typer av sårbarheter. Datamängden kan sakna mjukvaruspecifik information som gör att en viss skyddsteknik är olämplig i just den mjukvaran eller tvärtom. Studien utgår från språket Python utan att välja en viss version av språket. Men det är osannolikt att detta spelar en betydande roll för studiens resultat. Valet av språk verkar inte vara särskilt viktigt, med tanke på att de utmärkande egenskaper som tas upp för språket knappt märks i vilka skyddstekniker som är relevanta. Det är i och för sig möjligt att andra språk, eller versioner av Python, har fler egenheter och är mer riskabla att använda. Utöver detta behöver de valda skyddsteknikerna för denna studie ses som en generell mappning mot vissa typer av sårbarheter. Det kan förekomma undantagsfall där sårbarheter inte kan täckas av dessa skyddstekniker. Det är också möjligt att datamängden har brister som inte hunnit upptäckas än, med tanke på att den publicerades förra året.

Den andra begränsningen med forskningen i rapporten är att resultaten baseras på subjektiva bedömningar. Forskargruppen har diskuterat och försökt ensa resonemangen för att minska personberoendet, men en del subjektivitet kan ändå prägla resultaten. Det är också känt att manuell kodgranskning har begränsningar i vilka sårbarheter som kan upptäckas (exempelvis språkspecifika sårbarheter, enligt di Biase m.fl., 2016). Sådan kodgranskning brukar ske för att upptäcka sårbarheter. De analyser som är gjorda i studien har istället utvärderat valda skyddstekniker för redan upptäckta sårbarheter. De begränsningar som finns med kodgranskning kan dock vara tillämpliga även för denna studie. Begränsningarna gäller både vårt resultat och de andra forskarnas framtagande av datamängden, där manuell kodgranskning ingick. Därtill kan subjektiva bedömningar ha spelat roll när själva kategorierna av skyddstekniker skapades i denna rapport.

5.3 Framtida forskningsmöjligheter

För forskare är de mest uppenbara framtida forskningsmöjligheterna att bemöta de begränsningar som diskuterades i avsnitt 5.2. Då kan resultaten valideras och

få starkare stöd. Exempelvis kan studien replikeras för andra versioner av Python, eller för andra språk som tros ha större betydelse för vilka sårbarhetstyper som uppstår. Därtill kan praktiska tester komplettera de subjektiva bedömningar som gjorts. Det går också att samla mer data om utvecklares bedömningar. Sådana data kan utgöras av de diskussioner som ibland publiceras på Github i samband med kodrevideringar. Det finns också möjligheter att sammanföra flera olika kategoriseringar av sårbarheter och skyddstekniker. Exempelvis kan de skyddstekniker som presenteras i denna rapport jämföras med skyddsteknikerna i Mitres D3fend-ramverk och då specifikt avseende källkodshärdning.

För Försvarsmakten finns det förmodligen mindre intresse av de nyss nämnda valideringarna. Mer relevanta är då forskningsmöjligheter som fokuserar på att bredda de praktiska tillämpningarna av resultaten. En möjlighet är då att använda datamängden för att träna AI till att bli bättre på att upptäcka sårbarheter och införa kodrevideringar, vilket föreslogs av Wang m.fl. (2024) och utfördes för C/C++ i Wen m.fl. (2024). Detta kan öka möjligheterna att granska mjukvaras säkerhetsnivå. Ännu bättre blir det förmodligen om AI:n tränas på data (mjukvara) som är mer lik den som så småningom ska granskas.

Förutom att använda AI för mjukvarugranskning, eller i allmänhet fokusera på kodrevidering, kan andra säkerhetsåtgärder användas för att hantera potentiellt osäker kod. Ett sätt är att låta funktioner utanför mjukvaran hantera sårbarheterna. Redan i dag finns till exempel säkerhetsmekanismer i operativsystem. Ju fler mjukvaror som körs desto mer lockande blir det att låta operativsystemet eller andra omkringliggande mekanismer stå för säkerheten. Med tanke på det allt större utnyttjandet av kodbibliotek i mjukvara blir det också allt svårare att betrakta en viss mjukvara isolerat från andra mjukvaror. Forskning skulle kunna undersöka vilka faktorer som avgör när det är lämpligare att bemöta säkerhetshot med säkerhetshöjande åtgärder på annan nivå än på mjukvarunivå. Inspiration kan tas från kaosteori där små skillnader i en enskild entitet kan påverka systemet på ett svårförutsägbart sätt. Forskningen kan också fokusera på organisatoriska aspekter med försvar-på-djupet.

Ett annat alternativ till kodrevideringar är att lägga mer fokus på att undvika att införa sårbarheter i koden. För Försvarsmakten skulle detta innebära mer fokus på leverantörstillit. En utgångspunkt är då att utvecklare använder många olika metoder som kallas bästa praxis och som sägs höja säkerheten vid utveckling. Det finns dock begränsat med vetenskaplig evidens för att kora en viss praxis till bäst eller ens kunna konstatera att koden blir säkrare med vissa etablerade metoder. Kudriavtseva och Gadyatskaya (2022) sammanställde metoder för säker mjukvaruutveckling och frågade sig varför säkerhetsproblemen fortsätter trots att metoderna används. Det finns goda möjligheter att experimentellt utvärdera metoderna, vilket kan kombineras med att studera vilka steg (eller avsteg) som har samband med vilka uppkomna sårbarhetstyper. Etablerandet av vetenskapligt

stöd för en viss utvecklingsmetod skulle underlätta möjligheterna att ställa välgrundade krav på leverantörer.

En annan aspekt är att beställare och slutanvändare inte ska tro att säkerhet är ett uppnåeligt slutgiltigt tillstånd, snarare än något arbeta med över tid. Med denna vetskap kan man flytta en del av fokuset från att försöka skapa eller köpa helt säker mjukvara, till att försöka förutse vilka sårbarheter som kommer uppkomma och positionera sig för dessa. Tidigare forskning har visat att det finns vissa samband mellan tidigare uppkomna sårbarheter i Pythonkod och senare sårbarheter i koden (Ruohonen, 2018). Därtill verkar buffertoperationer vara särskilt svåra att få till på rätt sätt, vilket gäller framförallt C-kod (Wang m.fl., 2024). Denna forskning kan utökas samt kompletteras med utvärdering av kodrevideringars långvariga effekter på säkerheten och mjukvarans prestanda i övrigt. Här är det relevant hur långvariga organisationens system förväntas bli. För organisationer där mjukvaror lever kvar under lång tid är det viktigare att se säkerhet som något att arbeta med över tid.

6 Slutsatser

I detta kapitel redovisas rapportens slutsatser genom att kortfattat besvara forskningsfrågorna. Frågorna besvaras utifrån hur olika skyddstekniker åtgärdar sårbarheter i mjukvara skriven i Python, där rapportens data kommer från Reposvul och Github.

6.1 Vilka sårbarhetstyper är vanligast?

I den valda datamängden rör de flesta av sårbarhetstyperna antingen hantering av indata eller autentisering och åtkomstkontroll. Den vanligaste sårbarhetstypen (CWE) är *CWE-79 cross-site scripting* och den förekommer i ungefär en sjundedel av alla förekomster. Denna följs av *CWE-20 bristande indatavalidering*. Den tredje vanligaste är en övrigt-grupp för de konkreta sårbarheter (CVE:er) som saknar kategorisering. Ganska vanliga sårbarhetstyper är även *CWE-200 exponering av känslig information* samt *CWE-22 sökvägstraversering*.

6.2 Vilka skyddstekniker är vanligast?

Den vanligaste skyddstekniken är *indatavalidering*. Den är vanligast både vad gäller att täcka flest sårbarhetstyper och flest specifika sårbarheter. Indatavalidering är lik skyddstekniken *indatasanering*, vilket är den tredje vanligaste skyddstekniken. Tillsammans med indatavalidering täcker de drygt en tredjedel av sårbarheterna. Den näst vanligaste skyddstekniken är *övrigt*, vilket representerar en spretig mängd olika tekniker som inte enkelt kan klassificeras på något tydligt sätt. Den fjärde vanligaste skyddstekniken är *utdatasanering*, vilket kan ses som utdatas motsvarighet till skyddsteknikerna för indata.

Ungefär en femtedel av sårbarheterna täcks av skyddstekniker som rör autentisering och åtkomstkontroll: *verifiering av autentisering, åtkomstkontroll, ökade krav på säkrare autentiseringsuppgifter* samt *säkrare lagring av autentiseringsuppgifter*.

Knappt en tiondel av sårbarheterna täcks av skyddstekniker som rör prestanda och separering av beräkningar: *optimering av resursanvändning*, *sandboxing* samt *synkroniserad parallellkörning*.

Bland de ovanligare skyddsteknikerna finns *säker deserialisering av dataobjekt* samt *säker pakethantering och beroendehantering*. Dessa skyddstekniker borde vara betydligt relevantare för Python än de flesta andra programmeringsspråk men är alltså ändå inte vanlig i detta urval för Python.

6.3 Vilka samband finns det mellan sårbarhetstyperna och skyddsteknikerna?

I vissa fall står en skyddsteknik för mer än hälften av åtgärderna för sårbarhetstypen. För de flesta sårbarhetstyper finns däremot en handfull alternativa skyddstekniker som åtgärdar sårbarhetstypen. I ett fall finns ingen skyddsteknik som står för ens 30 procent av åtgärderna.

Bland skyddsteknikerna är *indatavalidering* både väldigt vanlig i hur många sårbarhetstyper den är relevant för och hur relevant den är för dem. Något mindre vanlig och viktig är *indatasanering*. Bara tre sårbarhetstyper åtgärdas aldrig av dessa två skyddstekniker. Åtgärder vid utmatning är betydligt mindre vanliga.

Några skyddstekniker är ovanliga men viktiga för enstaka sårbarhetstyper. Det gäller framförallt skyddstekniken *optimerad resursanvändning*, men också *parametriserade frågor* samt *konstanta operationer*. Dessutom är både *säker deserialisering* samt *säker pakethantering och beroendehantering* ovanliga men används mycket för just *CWE-502 deserialisering av icke-betrodda data*.

7 Referenser

Bogaerts F.C.G., Ivaki N., Fonseca J. 2024. A Taxonomy for Python Vulnerabilities. IEEE Open Journal of the Computer Society, vol. 5, pp. 368-379. https://doi.org/10.1109/OJCS.2024.3422686

Cass S. 2024. The Top Programming Languages 2024. Typescript and Rust are among the rising stars. IEEE Spectrum. https://spectrum.ieee.org/top-programming-languages-2024

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. https://doi.org/10.1109/SCAM.2016.30

Jensen C., Vestlund C., Gustavsson C., Andersson V., Nyholm L., Eidenskog D. 2024. Inventering av testfall för verktyg som identifierar mjukvarusårbarheter. FOI Memo 8673. Totalförsvarets forskningsinstitut. https://www.foi.se/rest-api/report/FOI%20Memo%208673

Karlzén H., Eidenskog D., Falkcrona J., Valassi C. 2023. Varför har mjukvaror sårbarheter?. FOI-R--5550--SE. Totalförsvarets forskningsinstitut. https://www.foi.se/rest-api/report/FOI-R--5550--SE

Kudriavtseva A., Gadyatskaya O. 2022. Secure Software Development Methodologies: A Multivocal Literature Review. arXiv preprint arXiv:2211.16987. https://arxiv.org/abs/2211.16987

Ruohonen J. 2018. An empirical analysis of vulnerabilities in python packages for web applications. 9th International Workshop on Empirical Software Engineering in Practice (IWESEP) (pp. 25-30). IEEE. https://doi.org/10.1109/IWESEP.2018.00013

Wang X., Hu R., Gao C., Wen X.-C., Chen Y., Liao Q. 2024. ReposVul: A Repository-Level High-Quality Vulnerability Dataset. IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Lisbon, Portugal, pp. 472-483, https://doi.org/10.1145/3639478.3647634

Wen X.C., Wang X., Chen Y., Hu R., Lo D., Gao C. 2024. Vuleval: Towards repository-level evaluation of software vulnerability detection. arXiv preprint arXiv:2404.15596. https://doi.org/10.48550/arXiv.2404.15596

Bilaga A: Taxonomi av skyddstekniker

De skyddstekniker som ingår i rapportens taxonomi beskrivs i Tabell 5. Beskrivningarna av skyddsteknikerna utgår delvis från rapportförfattarnas tidigare kunskap, delvis från de åtgärdsförslag som i vissa fall fanns i Mitres databas.

Tabell 5 Taxonomins skyddstekniker med förklaringar.

Skyddsteknik	**Förklaring**
Indatavalidering	Validera att alla indata följer förväntade format och regler.
Indatasanering	Filtrera och sanera (vid behov ändra) alla indata för att säkerställa att de följer förväntade format och regler.
Säker deserialisering	Använd säker serialisering och deserialisering vid konvertering av objekt.
Utdatasanering	Säkerställ att loggar och utskrifter inte innehåller känslig information såsom filsökvägar, användarlösenord, tokens eller API-nycklar.
Parametriserade frågor	Använd bindningsparametrar vid databasanrop med användarindata för att se till att indata inte kan tolkas som kod.
Åtkomstkontroll	Säkerställ att användare bara får tillgång till system och tjänster som de är behöriga till. Tillämpa även principen om minsta privilegium, dvs. ge endast de rättigheter som krävs för att utföra en viss uppgift och inget mer. Undvik att använda förinställda standardval av autentiseringsuppgifter, t.ex. lösenordet password eller admin.
Verifiering av autentisering	Verifiera att användaren är den rättmätiga entiteten den utger sig för att vara. Utför även lösenordsbyten när så behövs.
Sandboxing	Begränsa kodexekvering till en isolerad miljö (sandlåda) genom att t.ex. definiera externa komponenter som otillgängliga, för att minska potentiella konsekvenser vid hantering av osäker eller opålitlig kod.
Optimering av resursanvändning	Förbättra minneshanteringen för att undvika extrem resursanvändning som annars kan följa av undermålig minneshantering eller tillgänglighetsangrepp.

Skyddsteknik	**Förklaring**
Ökade krav på säkrare autentiseringsuppgifter	Öka kravet på säkrare inloggningsuppgifter, t.ex. att använda längre lösenord.
Säkrare lagring av autentiseringsuppgifter	Lagra lösenord och andra känsliga uppgifter på ett säkrare sätt.
Användning av säkra kryptografiska parametrar	Använd lämpliga krypteringslägen (eng. modes of operation), nyckellängder, algoritmer, pseudoslumptalsgeneratorer och implementeringar.
Användning av konstanta operationer	Utför operationer med konstant exekveringstid, oberoende av ingångsvärde, för att minimera risken för sidokanalattacker.
Säker pakethantering och beroendehantering	Håll Pythonversionen och biblioteken uppdaterade. Samt begränsa användningen av osäkra tredjepartsbibliotek.
Synkroniserad parallellkörning	Förhindra kapplöpning (eng. race condition) genom att använda lås, semaforer och synkroniseringsmekanismer i multitrådade och parallella Pythonapplikationer. Undvik även delade globala variabler i multitrådade program när korrekt synkronisering saknas.
Övrigt	(Övriga som inte passar i de andra kategorierna.)

ISSN 1650-1942