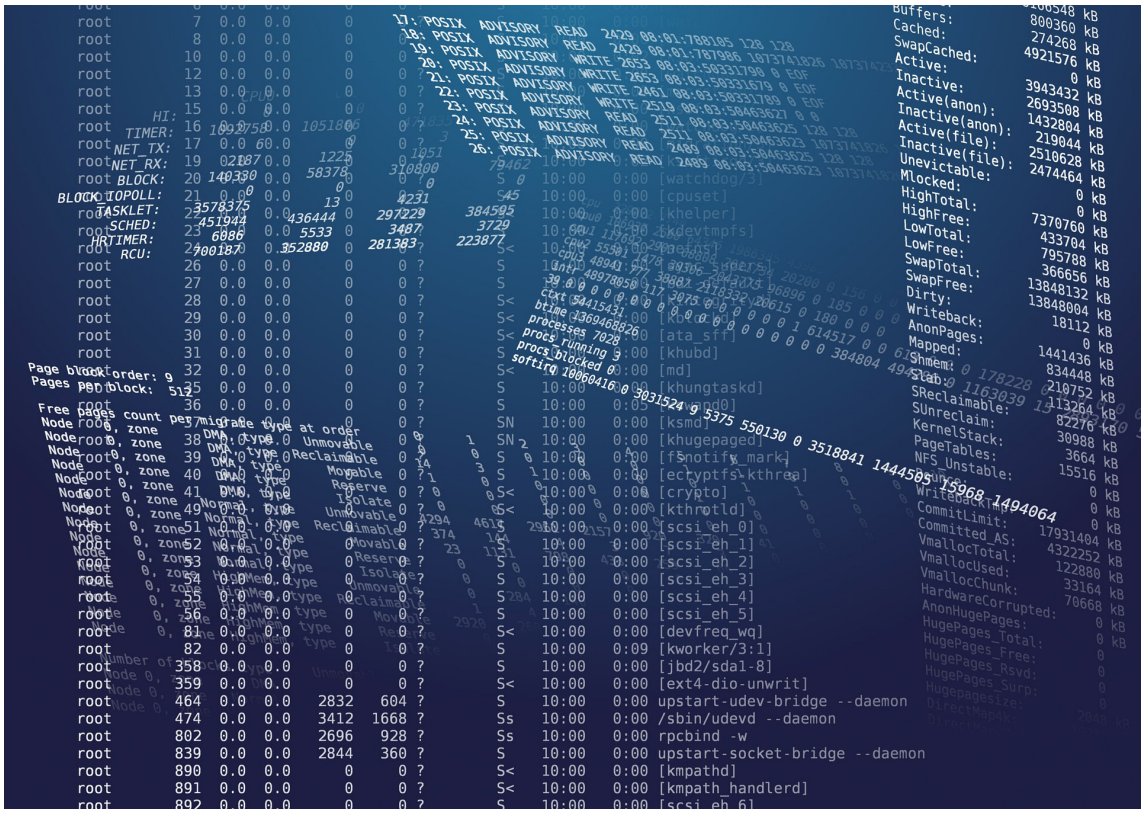
Automatisk attackkodsgenerering

En skanning av forskningsfronten

JACOB LÖFVENBERG, TEODOR SOMMESTAD, CAROLINE BILDSTEN



FOI-R--4737--SE
ISSN 1650-1942 Mars 2019

Jacob Löfvenberg, Teodor Sommestad, Caroline Bildsten

Automatisk attackkodsgenerering

En skanning av forskningsfronten

Bild/Cover: Pixabay – Antoine Pernot

Titel	Automatisk attackkodsgenerering – En skanning av forskningsfronten
Title	Automatic Exploit Generation – A Survey of the Research Frontier
Rapportnr/Report no	FOI-R--4737--SE
Månad/Month	Mars
Utgivningsår/Year	2019
Antal sidor/Pages	48
ISSN	1650-1942
Forskningsområde	4. Informationssäkerhet och kommunikation4. Informationssäkerhet och kommunikation
Projektnr/Project no	I70161
Godkänd av/Approved by	
Ansvarig avdelning	Ledningssystem

Sammanfattning

Automatiserad attackkodsgenerering (automated exploit generation) innehåller två steg: (1) olika sorters automatiserad kodanalys används för att identifiera programkodsfel som har säkerhetsimplikationer, (2) ett annat automatiserat verktyg analyserar de identifierade felen och skapar färdiga IT-attacker som använder dessa fel som attackvektorer. Framgångsrika experiment med denna typ av system har gjorts och finns beskrivna. Företag med detta som inriktning har startats, bland annat av personerna bakom vinnarbidraget till Darpas Cyber Grand Challenge.

Denna rapport bygger på en litteraturgenomgång av publikt tillgängliga beskrivningar av fungerande system för automatiserad attackkodsgenerering. Fokus har legat på vetenskapliga publikationer och har resulterat i att en handfull existerande system har identifierats. De beskrivna systemen är resultat av relativt omfattande, ingenjörsmässiga insatser, huvudsakligen bestående av att sätta samman existerande mjukvarukomponenter till välfungerande system för automatiserad attackkodsgenerering.

De attacker som de beskrivna systemen genererar verkar uteslutande baseras på relativt lätt exploaterbara svagheter, särskilt buffertöverskridningar och formatsträngssårbarheter. De beskrivna systemen har mycket begränsad möjlighet att kringgå allmänt använda skyddsåtgärder och kan därmed beskrivas som enkla men fungerande.

Nyckelord: automatiserad attackkodsgenerering, exploit

Summary

Automated exploit generation includes two steps: (1) different kinds of automated code analysis are used to identify software errors that have security implications, (2) another automated tool analyses the identified errors and creates exploits that use these errors as attack vectors. Successful experiments with this type of systems have been described. Companies with this as their focus have been started, among others by those behind the winning contribution to DARPA's Cyber Grand Challenge.

This report is based on a literature review of publicly available descriptions of working systems for automated exploit generation. Focus has been on scientific publications and has resulted in a handful of existing systems being identified. The identified systems are the result of relatively extensive engineering efforts, mainly consisting of combining existing software components to create well-functioning systems for automated exploit generation.

The exploits generated by the identified systems appear to be based solely on relatively easily exploitable vulnerabilities, particularly buffer overflows and format string attacks. The identified systems have very limited ability to circumvent widely used protective measures and can thus be described as working but simple.

Keywords: automated exploit generation, exploit

Innehållsförteckning

1 Inledning

Totalförsvarets forskningsinstitut FOI genomför varje år ett antal *skannande projekt* där forskningsläget i ett speciellt ämne undersöks för att öka organisationens samlade kompetens och främja enskilda individers kompetensutveckling. Projekten finansieras indirekt av Försvarsmaktens samlingsbeställning och väljs utifrån kriterier som säger att projektet (1) ska undersöka forskning utförd utanför FOI, (2) ska titta på områden eller tekniker kopplade till militär förmåga och (3) inte överlappa med pågående forskningsprojektverksamhet. Denna rapport beskriver resultatet av ett skannande projekt om automatisk attackkodsgenerering som utfördes under 2018.

1.1 Attackkodsgenerering

Många IT-angrepp exponerar datorprogram för särskilda koder som gör att de beter sig på ett (för systemägaren) oönskat sätt. Dessa attackkoder kan exempelvis möjliggöra att data kan skrivas till otillåtna platser eller att en obehörig användare kan ta över kontrollen över datorn programmet körs i. Det går att skapa sådana attackkoder eftersom mjukvaror har buggar, och en del buggar har säkerhetsimplikationer, så kallade sårbarheter. En ansenlig del av arbetet med att skapa säkra IT-system handlar om att förebygga dessa sårbarheter, men det läggs också mycket tid på att göra det svårt att utnyttja dem, eller på att ta bort dem med hjälp av mjukvaruuppdateringar.

Produktion av attackkoder kan motiveras av behovet för utvecklare, systemadministratörer och försvarare att prioritera vilka buggar som ska hanteras. Mjukvaror kan ha många olika typer av fel och det är långt ifrån alla dessa som har säkerhetsimplikationer, dvs. kan utnyttjas av någon för att påverka sekretess, tillgänglighet eller riktighet. Om det med enkla medel går att förutse vilka buggar som utgör sårbarheter kan mjukvaruutvecklare prioritera dessa framför andra buggar och systemadministratörer kan prioritera uppdateringar eller andra systemförändringar som hanterar dessa säkerhetsrelevanta buggar.

Att producera en attackkod kan ses som en trestegsprocess (se figur 1) där det först behöver identifieras en bugg i form av ett oönskat beteende, exempelvis en systemkrasch i en webbtjänst. För att detta ska ha säkerhetsimplikationer krävs det i regel att denna bugg kan aktiveras av en angripare, exempelvis en godtycklig användare av webbtjänsten. En sådan bugg klassas typiskt som en sårbarhet. Sårbarheter kan dock ha olika säkerhetsimplikationer och vara olika svåra att utnyttja. I allmänhet är de allvarligaste sårbarheterna de som kan utnyttjas av angripare för att ta kontroll över ett system, exempelvis om en användare av webbtjänsten kan bli administratör på webbservern.

Figur 1. Processen för att producera attackkoder.

Domänexperter har skattat att medianarbetsinsatsen som krävs för att producera en attackkod som exploaterar en allvarlig[1] sårbarhet varierar mellan en och fjorton dagar beroende på om koden inspekterats av granskare, testats med analysverktyg, är otillgänglig för angriparen och är skriven i ett typsäkert språk [1]. Denna rapport handlar om forskning för att automatisera processen, men rapporten kommer i viss utsträckning även att beröra forskning relaterad till enbart de två första stegen eftersom det finns flera intressanta lösningar med denna inriktning. Andra FOI-rapporter [2][3] presenterar översikter över vissa metoder för att identifiera sårbarheter varför inga detaljer ges i denna rapport.

Forskning om att automatisera attackkodsgenerering har pågått i mer än ett decennium. Området fick nyligen en knuff framåt i och med att USA:s militära forskningsorganisation Defense Advanced Research Projects Agency (Darpa) organiserade en tävling där automatisering av processen var central. I den allmänt tillgängliga litteraturen finns trots detta bara två översiktsbeskrivningar av forskningen inom området [4][5]. Dessa värderar dock inte hur mogna och användbara de föreslagna lösningarna är.

1.2 Darpas Cyber Grand Challenge

Darpa genomförde år 2015–2016 en tävling med namnet Cyber Grand Challenge (CGC). Syftet var att skapa automatiska system som minskar behovet av de stora arbetsinsatser som normalt krävs för att hitta och laga programvarufel som kan utgöra sårbarheter i mjukvara. Genom att erbjuda stora prispengar lockades många att delta. Mer än hundra lag registrerade sig, tjugoåtta lag nådde kvalificeringsomgången och de sju lag som presterade bäst fick delta i finalen.

Programmen som lagen skulle bearbeta var inte vanliga Linux- eller Windows-program. Istället hade Darpa skapat ett mycket enkelt operativsystem, Decree,

[1] I studien frågades det om en sårbarhet som enligt Common Vulnerability Scoring System rankats som allvarlig (High Severity), vilket så gott som alla angreppskoder som denna rapport berör gör.

som var basen för hela tävlingen. Detta förenklade för lagen och gjorde det möjligt att i större utsträckning fokusera på programanalysen istället för implementationsdetaljer i ett stort och komplext operativsystem [6].

Lagens uppgift var att bygga automatiska system som hittade, utnyttjade och lagade sårbarheter i de program som tävlingsledningen tagit fram. Poäng gavs efter hur väl detta genomfördes. I kvalificeringsomgången fick lagen 131 program att behandla. I finalen ställdes de sju lagen mot varandra. Deras uppgift var att hitta, utnyttja och laga programvarufel i de program som tävlingsledningen tagit fram. De fick också poäng för att hitta fel i program som andra lag redan lagat.

Att utnyttja ett fel definierades i CGC som att göra minst en av två saker:

1. Krascha programmet (orsaka *segmentation fault*) med åtminstone partiell kontroll över vissa processorregister.
2. Läsa minst fyra på varandra följande byte data från en viss minnesarea omfattande 4 096 byte.

CGC kan sägas utgöra en milstolpe inom området. Tävlingen motiverade flera lag att göra storskaligt, ingenjörsmässigt arbete vilket lyfte lösningarna till en mer produktionsmässigt användbar nivå.

Flera av bidragen beskrivs i den vetenskapliga litteraturen, antingen i artiklar skrivna av lagen bakom respektive bidrag eller i områdesbeskrivande avsnitt i artiklar med angränsande innehåll. Ett av bidragen till CGC har gjorts till öppen källkod efter att tävlingen avgjorts. Denna öppna källkod har legat till grund för fortsatta försök, som i sig beskrivits i vetenskapliga artiklar. De program som de tävlande lagen bearbetade under CGC har i viss mån använts som testobjekt när nya, liknande lösningar presenterats. Detta är värdefullt eftersom en etablerad testuppsättning inom ett område möjliggör jämförelse mellan olika forskares resultat och lösningar.

Det vinnande bidraget, Mayhem, har kommersialiserats och sålts bland annat till amerikanska försvarsdepartementet (DOD)[2]. Det finns också ett tidigt arbete med att erbjuda Mayhem som en service för att analysera och laga binär programkod hos en inledningsvis begränsad krets av kunder[3].

[2] https://www.cyberscoop.com/mayhem-darpa-cyber-grand-challenge-dod-voltron/ (läst 2019-01-11)
[3] https://forallsecure.com/pilot/ (läst 2019-01-11)

1.3 Syfte och avgränsningar

Denna rapport syftar till att sammanfatta forskningsläget inom automatisk attackkodsgenerering. Detta görs genom att identifiera existerande lösningar och söka svar på tre frågor:

1. Vilka lösningar finns?
2. Vilka attackkoder klarar de av att generera?
3. Vilka testfall har de utsatts för?

Sökningen, och därmed svaren, har flera avgränsningar. Till att börja med ges bara svar på automatisering av angrepp mot datorers minneshantering. Denna avgränsning motiveras av att (1) den absoluta merparten av de existerande lösningarna är inriktade på denna typ av angrepp, (2) det är en kraftfull typ av angrepp eftersom den ger angriparen kontroll över datorn och (3) det är en typ av angrepp som är mer komplicerad än många andra typer. Angående det sista kan det konstateras att vissa andra typer av sårbarheter är triviala att utnyttja när de väl hittats. En så kallad filinkluderingssårbarhet, som låter en användare peka på en fil som ska exekveras, kräver exempelvis ingen avancerad ingenjörskonst för att kunna användas i ett angrepp.

En annan betydande begränsning är att bara öppen och publicerad vetenskaplig forskning har inkluderats. Detta innebär att forskning som är ingenjörsmässigt avancerad men inte publicerad i vetenskapliga forum kan ha förbisetts. Exempelvis finns ett antal praktiskt orienterade presentationer, filmer och texter om bidrag från CGC. Dessa har inte inkluderats i genomgången. Avgränsningen till publicerad vetenskaplig forskning innebär också att forskning som omgärdas av sekretess inte tagits med. Det finns såväl företagsintressen som försvars-intressen kopplade till området och det kan antas att inte alla framsteg är öppet redovisade.

1.4 Rapportöversikt

Som framgår ovan är denna rapport avgränsad till angrepp mot datorers minnes-hantering. Läsare som inte är bekanta med begrepp som returadresser, strängar, buffertar och minnesrandomisering bör läsa den korta introduktion som ges i kapitel 2. Kapitel 3 beskriver hur litteraturgenomgången genomfördes och hur svaren på de olika frågorna söktes i artiklarna. Kapitel 4 presenterar svaren på de tre frågorna: Vilka lösningar finns? Vilka attackkoder klarar de av att generera? Vilka testfall har de utsatts för? Kapitel 5 diskuterar tillförlitligheten i dessa svar och presenterar slutsatserna från studien.

2 Datorers minneshantering

Detta kapitel ger en översiktlig beskrivning av hur minneshantering fungerar i moderna datorer av skrivbords- och servertyp. Minneshantering är en komplicerad uppgift. Syftet här är inte att ge en fullständig beskrivning utan bara att ge en tillräcklig bakgrund för att på en principiell nivå kunna förstå de attacker och de skydd som diskuteras i rapporten.

2.1 Grundläggande funktioner

För att kunna diskutera minnesbaserade attacker behövs först en grundläggande beskrivning av hur datorer hanterar minne. Nedan beskrivs hur minnet fördelas mellan olika användningsområden och en något mer noggrann beskrivning ges av stacken och heapen samt deras funktion.

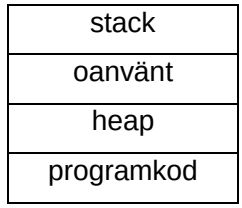
2.1.1 Minnesrymden i en dator

Tack vare virtuell minneshantering upplever varje process i datorn att den får tillgång till hela datorns minnesrymd. Operativsystemets minneshanterare döljer det faktum att hela minnesrymden inte motsvaras av verkligt, fysiskt minne och att det finns flera processer som kör samtidigt.

Processens minnesrymd delas upp i olika delar med olika funktioner. Den exakta fördelningen beror på detaljer i operativsystem, programspråk och hårdvara. Viktiga områden är:

- programkodsavsnittet, där processens körbara program finns
- heapen, där minne allokeras på allt högre adresser efterhand som heapen växer
- stacken, där lokala funktionsvariabler och återhoppsadresser vid funktionsanrop lagras på allt lägre adresser efterhand som stacken växer.

I figur 2 visas en bild av minnesrymden i Linux. I Windows ser det lite annorlunda ut, men för den principiella förståelsen är skillnaden oväsentlig. Det finns fler områden i minnesrymden, men dessa behövs inte för förståelsen i denna rapport.



Figur 2: Minnesrymden i Linux. Notera att stacken växer nedåt och heapen växer uppåt.

2.1.2 Stacken

Framställningen i detta avsnitt baseras på hur en x86-processor i 32-bitarsläge fungerar. Beskrivningarna är dock förenklade. I verkligheten finns fler detaljer i stackhanteringen som vi väljer att bortse från för att göra det lättare att förstå den grundläggande funktionen, vilket räcker för en principiell förståelse för de stackbaserade minnesattackerna i avsnitt 2.2.

Stacken har flera viktiga funktioner vid funktionsanrop. När en funktion anropas behöver det vara möjligt att skicka parametrar till funktionen, det behöver gå att använda lokala variabler i funktionen och när funktionen är färdig måste programflödet kunna återvända till punkten direkt efter funktionsanropet. Stacken används till allt detta.

Nya saker som läggs till stacken placeras på minnesadresser nedanför det som fanns där tidigare, så när stacken blir större växer den nedåt. Den nedersta adressen i den nuvarande stacken finns lagrad i ett särskilt register, stackpekaren. När något ska läggas till stacken används stackpekaren av processorn för att hitta nästa lediga utrymme. I samband med att ett objekt läggs till stacken minskas stackpekaren så att den återigen pekar på den nedersta adressen i stacken.

När vi i den följande framställningen beskriver hur stacken utnyttjas används orden *ovanför* och *nedanför* för att beteckna högre respektive lägre minnesadresser i stacken. I alla exempel gäller att rader långt upp är placerade på högre minnesadresser och stacken växer nedåt, mot lägre minnesadresser. Det som är senast tillagt till stacken ligger således nederst, både i betydelsen på att det ligger på stackens lägsta adress i minnen och i betydelsen att det är raden längst ner i exemplen.

För att se hur stacken fungerar i ett sammanhang, betrakta programkoden i exempel 1, där funktionen `main` anropar funktionen `myfunction` med argumentet `mfarg`.

```
void main() {...
  myval = myfunction(myarg);
  nextval = do*some+math;
...}

  int myfunction(int mfarg) {
  int localvar=0;
...}
```

Exempel 1: Programkod som gör ett funktionsanrop.

När funktionen `myfunction` anropas händer flera saker med stacken. Det som är viktigt för denna framställning är att returadressen läggs till stacken, följt av parametern `myarg` och den lokala funktionsvariabeln `localvar`, se exempel 2 nedan. Som beskrivits ovan så ligger det som är tillagt senast, i detta fall `localvar`, längst ner i stackens minne.

```
<resten av stacken>
<returadress>
<myarg>
<localvar>
```

Exempel 2: Ordningen på de parametrar som skrivs till stacken vid ett funktionsanrop. Beskrivningen är inte fullständig men visar de parametrar vi behöver i den fortsatta framställningen.

När funktionen `myfunction` är färdig används den sparade returadressen för att återföra programflödet till instruktionen direkt efter funktionsanropet, som i exempel 1 är raden med `nextval = do*some+math;`.

2.1.3 Heapen

Heapen är ett minnesutrymme som är avsett för att låta program göra dynamisk minnesallokering under körning. När ett C-program allokerar minne med funktionen `malloc` används minne från heapen. Samma sak gäller när ett C++-program skapar ett nytt objekt med operatorn `new`. I båda fallen allokeras ett lämpligt minnesblock på heapen. Detta innebär att en pekare till det allokerade minnet returneras och den tillgängliga minnesmängden på heapen minskas. Det finns också motsvarande möjligheter att lämna tillbaka minne till heapen när minnet inte längre används. Hur heapen hanteras när data skrivs till, flyttas runt och tas bort är komplicerat jämfört med hur stacken används. Det finns exempelvis flera olika sätt att hantera heapen som används av olika system, där vissa sätt använder slump för att bestämma var data ska placeras.

2.1.4 Variation i var data placeras

Var den buffert som skrivs över är placerad påverkar hur komplicerat angreppet blir. Detta kan bland annat skilja mellan olika versioner av samma mjukvara, mellan olika operativsystem och mellan olika språkpaket i samma operativsystem. Genomgående är angrepp mot data placerad på heapen svårare att lyckas med. Detta eftersom data på heapen placeras ut dynamiskt, vilket innebär att de hamnar på mer svårförutsägbara platser i datorns minne. Platsen beror på vilken heaphanterare som används, hur data finns allokerad innan angreppet, storleken på data och andra allokeringar i samma heap som sker under angreppet. Att lyckas identifiera den faktiska adressen för var data hamnar är därmed svårare. Det kan dock åstadkommas genom att angriparen identifierar sekvenser av operationer som gör att placeringen av data blir förutsägbar, något som brukar kallas *heap feng shui*. Angriparen kan också fylla stora delar av heapen med sin attackkod för att ha större möjlighet att träffa rätt. Merparten av denna programkod kan bestå av en så kallad nop-släde[4] som följs av den faktiska attackkoden.

2.2 Stackbaserade minnesattacker

För en angripare är det lockande att manipulera stacken eftersom den regelmässigt innehåller returadresser, som används för att styra programflödet, se avsnitt 2.1.2. Attacker via stackmanipulation kan göras på många sätt och nedan beskrivs två typer, buffertöverskridningar och formatsträngsattacker. Notera att även om detta är mycket vanliga attacktyper så finns det en stor uppsättning av andra sorters stackbaserade attacker.

2.2.1 Enkla buffertöverskridningar

Stackbaserad buffertöverskridning förutsätter att angriparen har möjlighet att förse programmet med indata som sedan, utan säkerhetskontroller, skrivs till en buffert (allokerat minnesutrymme) i stacken. Den grundläggande principen är att angriparen behöver få programmet att skriva mer data än vad bufferten rymmer, vilket resulterar i att minnet ovanför bufferten blir överskrivet. Ett vanligt mål är att skriva över en returadress med adressen till programkod som utför något för angriparen önskvärt. Ett program som är sårbart för buffertöverskridning visas i exempel 3.

[4] Nop (förkortning av engelskans no operation) är namnet på en maskininstruktion som inte gör något mer än tar plats i minnet och tar tid att utföra. Nop finns i de flesta processorarkitekturer. En nop-släde är en sekvens av nop som gör det lättare att hoppa till attackkoden när dess adress inte är exakt känd.

```
void opengate() {
    // Open the magic gate only if the user knows the secret
    password
    ...
}

int checkpassword() {
    char pwdbuffer[20];

    printf(”Ange hemligt lösenord: ”);
    scanf(”%s”,pwdbuffer);

    return strncmp(pwdbuffer,”mypass”,6);
}

void main() {

    if (checkpassword()==0)
        opengate();
    else exit();
}
```

Exempel 3: Ett program som kan angripas med hjälp av buffertöverskridning.

Programmet anropar funktionen `checkpassword` som frågar användaren efter lösenord. De sex första tecknen[5] i det angivna lösenordet och det korrekta lösenordet jämförs och om dessa överensstämmer så anropas funktionen `opengate`, som vi antar är vad angriparen vill uppnå trots att denne inte har det korrekta lösenordet.

När funktionen `checkpassword` anropas får stacken innehållet som visas i exempel 4.

```
<resten av stacken>
<returadress> [4 byte]
<pwdbuffer> [20 byte]
```

Exempel 4: Stackinnehåll efter anrop till funktionen `checkpassword`.

När funktionen `scanf` används för att ta emot lösenordet görs ingen kontroll av hur lång sträng användaren skriver in. En angripare kan utnyttja detta och skriva in något som inte får plats i bufferten som allokerats i stacken. Det som inte får plats kommer då att skrivas vidare uppåt i minnet, över det som kommer ovanför

[5] Funktionen `strncmp(x,y,n)` jämför strängarna `x` och `y`, men högst `n` tecken.

bufferten i stacken. I vårt fall är det returadressen som ligger närmast ovanför bufferten, se exempel 4. Genom att studera programkoden med analysmjukvara kan en angripare enkelt ta reda på adressen för funktionen `opengate`. Den sträng som angriparen skriver in kan sedan utformas så att den är 24 byte lång. De första 20 bytens innehåll har ingen betydelse utan används bara för att fylla `pwdbuffer`. De resterande fyra byten innehåller adressen till funktionen `opengate` och skriver över returadressen. Stacken får då innehållet som visas i exempel 5.

```
<resten av stacken>
<returadress> [adress till opengate(), 4 byte]
<pwdbuffer> [20 byte]
```

Exempel 5: Stackinnehåll efter lyckad buffertöverskridning med funktionspekning.

Efter att angriparen gjorts sin inmatning och returadressen skrivits över fortsätter programmet vidare och jämför de första sex tecknen med det korrekta lösenordet. Resultatet av jämförelsen returneras och därefter är funktionen `checkpassword` färdig och programflödet ska återföras till huvudfunktionen där nästa steg är att jämföra resultatet med noll för att avgöra om användaren ska släppas in eller inte. I vårt fall blir det dock inte så, eftersom returadressen skrivits över. Istället hämtas adressen till funktionen `opengate` från stacken och programflödet styrs dit istället för till den korrekta positionen i huvudfunktionen `main`. I och med detta så har angriparen framgångsrikt genomfört en attack och lyckats uppnå något som bara skulle varit möjligt för den som innehar det korrekta lösenordet.

Detta exempel beskriver hur en buffertöverskridningsattack kan se ut i sin allra enklaste form. Hur en sådan attack ser ut i verkligheten beror på det exakta utseendet på det program som angrips, men grundidén är att programmet inte kontrollerar storleken på innehållet i en buffert, vilket leder till att närliggande delar av minnet skrivs över. Om användaren kan påverka innehållet i bufferten så utgör programmet en möjlig kandidat för att angripas med en buffertöverskridningsattack. En vanlig variation är att angriparen genom sin inmatning även förser programmet med den programkod som ska köras när programflödet styrs om, exempelvis programkod som öppnar ett skalfönster och därmed tillåter angriparen att ge godtyckliga kommandon till datorn. I exempel 3 skulle denna programkod tillsammans med den ändrade återhoppsadressen kunna skrivas in vid lösenordsinmantingen. Stacken får då innehållet som visas i exempel 6.

```
<resten av stacken>
<returadress> [adress till pwdbuffer, 4 byte]
<pwdbuffer> [programkod, 20 byte]
```

Exempel 6: Stackinnehåll efter lyckad buffertöverskridning med pekning till lösenordsbufferten. Adressen till pwdbuffer är minnesadressen till starten på bufferten, dvs. den lägsta minnesadressen.

2.2.2 Formatsträngsattacker

En vanlig funktion för att låta ett program skriva text till skärmen i programspråket C är `printf`. Det finns även flera besläktade funktioner som kan användas på motsvarande sätt för att skriva till strängar eller filer, samt varianter som begränsar längden på det som skrivs. Gemensamt för dessa funktioner är att det första argumentet är en formatsträng som innehåller dels den vanliga text som ska skrivas ut, dels specialsymboler för variabler vars innehåll ska infogas i texten. Specialsymbolerna beskriver hur variablerna ska formateras, exempelvis om heltal ska anges decimalt eller hexadecimalt, hur bred spalt de ska uppta[6], hur många decimaler som ska anges på ett flyttal och så vidare. I tabell 1 visas några vanliga specialtecknen som kan användas i formatsträngar.

Tabell 1: Ett urval av specialtecken i formatsträngar.

Specialtecken	Betydelse	Parameter
%d	heltal i decimal form	heltal
%x	heltal i hexadecimal form	heltal
%s	sträng	pekare
%n	antal tecken som skrivits hittills – sparas på angiven adress	pekare
%hn	antal tecken som skrivits hittills – sparas på angiven adress – tolkat med 16-bitarsrepresentation	pekare

Om exempelvis `printf(”Hello world number %d”,worldcounter)` körs får stacken innehållet som visas i exempel 7.

```
<resten av stacken>
<returadress>
<worldcounter>
<pekare till formatsträngen>
```

Exempel 7: Stackinnehåll vid ett anrop till funktionen printf.

Funktionen `printf` behandlar formatsträngen ett tecken i sänder. Om det är ett vanligt tecken så skrivs det ut och om det är ett specialtecken så hanteras det på särskilt sätt. I exemplet ovan finns ett `%d`, vilket gör att nästa värde, ovanför i

[6] Exempelvis om talet är tre siffror och den angivna spaltbredden är åtta tecken så infogas fem mellanslag före talet så att det totalt blir åtta tecken brett.

stackens minne, hämtas och skrivs ut som ett heltal i decimal form. I exemplet ovan så innebär det att `worldcounter` hämtas och skrivs ut.

Möjligheterna med formatsträngar är större än det kan verka vid första påseendet. Om en angripare har möjlighet att styra innehållet i en formatsträng går detta att använda för olika typer av attacker. Den enklaste formen utnyttjar det faktum att det inte görs någon kontroll av om de använda specialtecknen matchar de bifogade parametrarna, vare sig med avseende på antal eller typ. Ett funktionsanrop `printf(”I stacken finns: %x %x %x %x”)` kommer att skriva ut den inledande texten, följt av de hexadecimala representationerna av de fyra heltal som ligger i stacken direkt ovanför pekaren till formatsträngen. Eftersom funktionsanropet inte bifogat några parametrar utöver formatsträngen finns inget mer i stacken som tillhör funktionen `printf`, varför istället andra värden med annan betydelse kommer att skrivas ut. På detta sätt kan en angripare se delar av stacken som inte ska vara tillgängliga för denne.

Ett något mer avancerat exempel tillåter angriparen att göra mer komplexa saker, se exempel 8.

```
void reprint() {
    char inputbuffer[100];

    printf(”Ange text: ”);
    scanf(”%s”,inputbuffer);

    printf(inputbuffer);
}
```

Exempel 8: Programkod som tillåter en mer avancerad formatsträngsattack.

När funktionen `reprint` anropats, och när den i sin tur anropat den sista `printf`-funktionen, så har stacken utseende som visas i exempel 9.

```
<resten av stacken>
<returadress för reprint()>
<inputbuffer[99]>
<inputbuffer[98]>
...
<inputbuffer[0]>
<returadress för printf()>
<pekare till formatsträng, dvs. inputbuffer>
```

Exempel 9: Stackinnehåll efter anropet till den sista `printf`-funktionen exempel 8.

I exempel 9 ovan har `inputbuffer` delats upp i sina 100 beståndsdelar för att förtydliga utseendet hos stacken. Pekaren till `inputbuffer` är adressen till det första elementet i bufferten, det vill säga adressen för `inputbuffer[0]`.

Som framgår av exempel 8 kan angriparen styra innehållet i formatsträngen som används i den sista `printf`-funktionen. Antag att angriparen anger format-strängen `”\05\05\05\05 %x %s”`. De inledande fyra `\05` är ett sätt att få strängen att innehålla fyra element med värdet `05`[7]. Specialtecknet `%x` kommer att skriva ut innehållet i stacken ovanför formatsträngen i hexadecimal form, vilket i vårt fall är returadressen för funktionen `printf` och `%s` kommer att tolka det som ligger ovanför denna returadress som en pekare till en sträng som ska skrivas ut. Men stackinnehållet i exemplet ovan visar att i stacken ovanför returadressen till `printf` finns `inputbuffer`, som angriparen själv har bestämt innehållet i. Det `printf` vill hämta är en strängpekare, vilket motsvarar precis de fyra första elementen i bufferten. Resultatet blir att funktionen `printf` kommer att tolka innehållet på adress `05050505` (hexadecimalt) som en sträng och skriva ut den, fram till första strängtermineringstecknet – som är noll i de flesta fall. Om formatsträngen väljs till andra inledande värden än `05050505` (hexadecimalt) kan valfria delar av minnet skrivas ut. Exempelvis kan en angripare använda denna teknik för att krascha programmet genom att ange en del av minnet som processen inte har tillgång till.

Beskrivningen ovan visar principen för hur formatsträngsattacker fungerar. Attackerna går att anpassa så att de tillåter inte bara läsning utan även skrivning på godtyckliga adresser. Det finns många detaljer, knep och variationer som inte beskrivits här[8].

2.3 Skydd och varianter på attacker

Som nämnts visar de exempel som presenterats ovan endast enklare fall av buffertöverskridningsattacker. I praktiken är det sällan så lätt som i exemplen. Det finns många omständigheter som begränsar angriparens möjligheter att överta kontrollflödet och som påverkar vilken information som krävs för att lyckas. Några exempel på saker som påverkar attackernas svårighet beskrivs nedan.

[7] I formatsträngar går det generellt att ange \(num) för att få strängen att innehålla ett element med värdet (num). Detta gör det möjligt att lägga till värden som inte har något vanligt tecken kopplat till sig.

[8] Se exempelvis https://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf (läst 2019-01-11)

2.3.1 Stackkakor

Moderna operativsystem och kompilatorer har utrustats med att antal skydd mot buffertöverskridningsattacker som explicit syftar till att göra angrepp svårare. Ett sådant skydd är stackkakor[9]. Dessa består av särskild data som placeras i stacken nedanför varje returadress. Värdet hos stackkakan kontrolleras sedan innan returadressen används för att säkerställa att det inte har förändrats. För att angriparen ska få sin kod exekverad krävs att stackkakorna, vars innehåll ofta är slumpvis valda, har korrekta värden. Vid en vanlig buffertöverskridningsattack går det inte att undvika att skriva över även stackkakorna, varför angriparen måste lyckas skriva över dessa med de ursprungliga, slumpmässiga värdena för att programflödet ska följa returadressen. Är stackkakorna inte korrekta anses det vara ett fel som hanteras på lämpligt sätt i en felhanteringsfunktion. Detta gör det svårt för angriparen att använda buffertöverskridningar för att styra om programflödet.

I vissa versioner av Windows har angripare lyckats använda felhanteringen i sig som ett sätt att ta kontroll över programflödet genom att skriva över den del av minnet där felhanteringen utförs eller genom att ersätta pekaren till felhanteringsfunktionen. Som en följd av detta har även felhanteringen säkrats upp, och angreppen behöver ta hänsyn till ytterligare faktorer för att lyckas.

2.3.2 Dataexekveringsskydd

Ett annat skydd mot buffertöverskridningsattacker är att datorn känner till vilka delar av minnet som innehåller data och förbjuder exekvering av program i dessa delar. Det angriparen kan placera i minnet är typiskt avsett att vara en sträng med bokstäver (som i exemplen ovan) och sådana minnesadresser ska inte innehålla programinstruktioner. Med dataexekveringsskydd går det att förhindra attacken om returadressen pekar på en plats in minnet som inte ska innehålla programkod, som i exempel 6 ovan. Detta skyddar dock inte mot angreppet som ges i exempel 3, eftersom returadressen i detta fall pekar på en del av minnet där programkod ska finnas, nämligen funktionen `opengate`.

Finns det inga funktioner i programmet som angriparen har nytta av kan istället allmänna biblioteksfunktioner användas. Programspråk innehåller ofta funktioner för att starta godtyckligt program på datorn, exempelvis funktionen `system` i programspråket C. Angriparen kan placera en sträng med önskade instruktioner, exempelvis instruktioner för att addera en användare, på den plats där funktionen letar efter sina argument och sedan peka returadressen till funktionen `system`.

[9] Dessa kallas även kanariefåglar, med hänsyftning på de kanariefåglar som förr användes i gruvor för att varna om luften var farlig.

En annan teknik för att kringgå dataexekveringsskydd är att identifiera platser i minnet som innehåller programkod som motsvarar vissa instruktioner eller effekter, och sedan peka på dessa. Om angriparen lyckas manipulera stacken så att den ser ut som i exempel 10 så går det att skriva godtyckligt värde till godtycklig adress.

```
<resten av stacken>
<returadress> [pekare till koden ”mov (ecx),eax; ret;”]
<värdet som ska skrivas, hamnar i eax >
<adressen värdet ska skrivas till, hamnar i ecx>
<returadress> [pekare till koden ”pop ecx; pop eax; ret;”]
<data som skrivs över vid attacken>
<sårbar buffert>
```

Exempel 10: Angrepp med så kallade gadgets.

Exemplet använder två så kallade register (`eax` och `ecx`), som är små väldigt snabba minnesutrymmen i datorer. Den nedersta returadressen i stacken i Exempel 10 flyttar programflödet till ett stycke exekverbar kod som hämtar data från stacken[10], som angriparen kunnat skriva till med en buffertöverskridning. Data från stacken placeras i registren `eax` och `ecx`, för att sedan returnera till adressen som utpekas av den översta returadressen i exempel 10. Denna flyttar programflödet till ett stycke exekverbar kod som skriver värdet i registret `eax` till adressen dit registret `ecx` pekar.

Exempel 10 visar hur det går att skriva valfritt värde till valfri adress i minnet, även i ett fall där angriparen inte kan skicka med sin egen programkod. I exempel 3 går det exempelvis att ändra hoppet till funktionen `checkpassword` så att istället funktionen `opengate` anropas. Andra exempel på hur denna typ av korta programsekvenser kan användas är för att anropa funktioner med vissa argument eller för att slå av dataexekveringsskyddet. En fullständig beskrivning av denna teknik är omfattande och komplicerad och lämnas därför utanför denna text.

2.3.3 Minnesrandomisering

När angriparen skriver in en egen returadress i stacken behöver denna peka på en bestämd plats i minnet. Minnesrandomisering är ett skydd som med avsikt placerar saker (exempelvis stacken) på olika platser i minnet vid olika körningar, vilket gör det mycket svårt att förutse var intressant kod och data (exempelvis funktionen `opengate` eller instruktionssekvensen “`pop ecx; pop eax; ret;`”) är placerade.

[10] Instruktionen `pop` hämtar ett värde från stacken och sparar det i det register som anges.

Det är dock inte alltid hela minnet som randomiseras. Exempelvis kan äldre programbibliotek innehålla absoluta referenser i koden och dessa program-bibliotek behöver undantas från minnesrandomisering. Placeringen av dessa kan då förutses av angriparen och refereras med en returpekare. I sådan fast placerad programkod kan det gå att hitta en trampolin, det vill säga en redan existerande sekvens av instruktioner som leder till att programflödet flyttas till en del av stackens minne som kan påverkas av buffertöverskridningen.

Som beskrivits tidigare så uppdateras stackpekaren, `esp`, så att den alltid pekar på stackens nedersta element. Instruktionen `jmp esp` flyttar därmed exekveringen till nedersta delen av stackens minne. Om angriparen kan hitta ett fast placerat kodavsnitt i minnet som innehåller instruktionen `jmp esp` kan denna användas för att rikta om exekveringen till skadlig kod som finns längst ner i stackens minne. En stack manipulerad som den till vänster i exempel 11 åstadkommer detta. Den sårbara funktionens returadress har här skrivits över med adressen till instruktionen `jmp esp` samtidigt som angriparen placerat sin attackkod i stacken ovanför returadressen. När funktionen returnerar flyttar datorn automatiskt programflödet till `jmp esp` samtidigt som den ändrar `esp` till att peka på adressen närmast ovanför returadressen. Det är just där angriparens attackkod finns, så när datorn utför instruktionen `jmp esp` flyttas programflödet dit.

Före funktionen returnerat

```
<resten av stacken>
<angriparens attackkod>
<adressen till ”jmp esp”>
<data som attacken skrivit över>
<sårbar buffert>
<diverse data>            <-esp
```

Efter funktionen returnerat

```
<angriparens attackkod>    <-esp
```

Exempel 11: Angrepp med trampolin.

2.3.4 Andra omständigheter och skydd

Om alla ovanstående skydd används krävs mer avancerade angrepp än de som beskrivits ovan. Angriparen kan exempelvis behöva kombinera sårbarheter med information om minnet för att gissa adresser som behövs när det finns minnes-randomisering. Ingen av de artiklar som identifierats i denna skanning behandlar dock in sådana metoder och de lämnas därmed utanför denna rapport. Det finns också andra kombinationer av system och skydd som påverkar hur angripare behöver göra som inte heller behandlas i de identifierade artiklarna. Exempelvis finns skydd som begränsar hur minne får adresseras, vilket påverkar hur format-strängssårbarheter kan användas. Dessutom finns minnesrandomisering av olika styrka.

3 Litteraturgenomgången

Litteraturgenomgången inleddes med att forskarna bekantade sig med området och den forskning som skett. Detta följdes av en systematisk (repeterbar) litteratursökning som syftade till att identifiera all relevant forskning publicerad på engelska i akademiska forum. Efter detta extraherades information för att kunna svara på de tre frågorna. Hur litteratursökningen gjordes beskrivs i detalj i avsnitt 3.1. Hur artiklar valdes ut beskrivs i avsnitt 3.2. Vilken information som extraherades och hur denna analyserades beskrivs i avsnitt 3.3.

3.1 Litteratursökning

Det konstaterades tidigt att detta område är ungt, omoget och saknar egna forum där resultat presenteras. Variationen i forum och terminologi gör det svårt att med en enkel metod, exempelvis en söksträng, identifiera all forskning av relevans. Istället valdes en metod som baseras på de referenser som finns i vetenskapliga artiklar. Metoden var som följer, där steg två och tre upprepades tills dess att inga nya artiklar identifierades.

1. En lista med potentiellt relevanta bidrag skapades genom att söka efter ”auto* exploit generation” i databasen Scopus. Termen ”exploit generation” är den term som använts sedan åtminstone 2002 för forskning som handlar om att skapa attackkoder och ”auto*” kräver att uttrycket inleds med ”automatic”, ”automated” eller något liknande. Den initiala listan kom att innehålla 12 bidrag.
2. För varje bidrag i listan med potentiellt relevanta bidrag undersöktes referenslistan för att hitta ytterligare bidrag av potentiell relevans. Detta bedömdes i huvudsak baserat på i vilken kontext bidraget refererades och på dess titel. Identifierade bidrag lades till i listan med potentiellt relevanta bidrag.
3. För varje bidrag i listan med potentiellt relevanta bidrag undersöktes Google Scholars lista med andra bidrag som *refererar till* det. Vilka av dessa bidrag som var potentiellt relevanta bedömdes även i detta fall främst baserat på titel och i vilken kontext det refererades. Identifierade bidrag lades till i listan med potentiellt relevanta bidrag.

Metoden gav 94 potentiellt relevanta bidrag, där en handfull var presentationer eller icke-akademiska texter. Det bidrag som refererades mest frekvent hade refererats 35 gånger från de andra bidragen. Därpå följde två bidrag som refererats åtta respektive nio gånger. Alla dessa tre hörde till de tolv som sökningen utgick från. Hela 75 bidrag refererades inte av någon annan av de 94 bidragen utan identifierades utifrån att de refererade till andra potentiellt

relevanta bidrag. De bidrag som litteratursökningen identifierade finns i appendix A.

3.2 Urval av artiklar

De två förstaförfattarna av denna rapport gick igenom de 94 potentiellt relevanta bidragen och bedömde om dessa

1. var en artikel (88 st)
2. dessutom beskrev eller testade en teknisk lösning (70 st)
3. dessutom handlade om sårbarheter i minneshantering (55 st)
4. dessutom producerade belägg för att sårbarheter är exploaterbara (46 st)
5. dessutom producerade attackkoder som går att använda för att öka sina privilegier på en dator (22 st).

Urvalet gjordes först på ett inkluderande sätt. Bland artiklarna som inte handlade om sårbarheter i minneshantering (kriterium 3) fanns bland annat en artikel som behandlade sårbarheter i hårdvaran i CPU:er och flera artiklar som analyserade webbapplikationer eller funktionsanrop till programvaror. Det är värt att notera att flera av artiklarna som producerade belägg för att sårbarheter är exploaterbara (kriterium 4) kan ses som gränsfall eftersom de beskriver lösningar som tar fram det som kallas *exploit primitives*. Det kan exempelvis vara information om anrop och minnesadresser som skulle göra det enklare för en människa att skapa en attackkod. Flera av bidragen i CGC hamnar i den kategorin. De inkluderades dock inte eftersom de inte producerade användbara attackkoder. Totalt mötte 22 artiklar kriterierna. Alla artiklarna finns listade i appendix A, där de inkluderade har markerats.

3.3 Analysmetod

De 22 utvalda artiklarna analyserades noggrannare. Artiklarna lästes för att besvara frågorna i tabell 2. Dessa frågor bildade ett ramverk vars mål var att kunna beskriva alla artiklarna på ett enhetligt sätt så att det blev möjligt att se mönster och jämföra artiklarnas lösningar. Alla frågorna i tabell 2 har svar som varierar mellan de 22 utvalda artiklarnas presenterade lösningar, vilket kan tolkas som att frågorna är relevanta för att beskriva de olikheter som finns. Om frågorna är tillräckliga, i betydelsen att de differentierar artiklarna i tillräcklig grad, beror på vad behovet är. Det finns inget rättframt sätt att objektivt avgöra om så är fallet, men författarna är av åsikten att de befintliga frågesvaren räcker för den typ av analys och diskussion som görs i denna rapport.

Tabell 2. Information som extraherades från artiklarna.

Information	**Alternativ**
Namn, kort beskrivning och relation till andra artiklar och verktyg?	(Fritextbeskrivning innehållande exempelvis referenser till andra artiklar och analysteknik lösningen byggts på).
Vilka indata behöver lösningen?	Källkod; binärkod; patch; systemkonfiguration; exploit-primitiv; indataexempel som kraschar programmet; information om angreppet; annan (beskrivning).
Vilken eller vilka plattformar fungerar lösningen för?	Windows; Linux; Decree (CGC); annan (beskrivning).
Vilka minnesdelar sker attackerna i?	Stacken; heapen; annan (beskrivning).
Vilken pekare angrips?	Returadress; funktionspekare; annan (beskrivning).
Hur sker överskrivningen?	Direkt; indirekt; annat (beskrivning).
Vilken attackkod används?	Shellkod med nop-släde; shellkod utan nop-släde; return-to-lib; annan (beskrivning).
Angrips någon särskild typ av funktion?	Funktion med formatsträng; kopieringsfunktion i libc; annan (beskrivning).
Vilka skydd kan attacken kringgå?	Stackkaka, dataexekveringsskydd minnesrandomisering, annat (beskrivning).
Hur har lösningen testats?	På specialdesignade exempel; på utvalda mjukvaror med kända, lämpliga sårbarheter; enstaka, allmänt spridda mjukvaror; flera, allmänt spridda mjukvaror.
Hur många sårbarheter kunde utnyttjas med hjälp av lösningen vid testerna?	Totalt antal angrepp som lyckades; antal angrepp som var tidigare okända.

Vid genomläsningen visade det sig att alla frågor inte gick att besvara för alla artiklar. Orsakerna varierade, men handlade väsentligen om att den nödvändiga informationen saknades i artikeln, att artikelns beskrivning inte gick att tolka, eller att frågorna inte passade artikelns innehåll eller framställning. Dessutom var svaren ofta behäftade med villkor i form av specialfall, förenklingar eller avgränsningar. I resultatkapitlet nedan återges därför inte enskilda lösningars svar på frågorna i tabell 2 utan den analys som gjorts baseras på svaren som helhet.

4 Resultat

Den äldsta artikeln av de utvalda artiklarna var från 2006 och den nyaste från 2018 (litteratursökningen gjordes under oktober och november 2018). En tabell över publikationsåren för de utvalda artiklarna visas i tabell 3 nedan.

Tabell 3: Publikationsår för utvalda artiklar.

Publikationsår	Antal
–2009	1
2010–2012	4
2013–2015	8
2016–2018	9

Av tabellen framgår en tydlig tillväxt i antalet artiklar som publicerats, även om det ser ut att ha skett en avmattning och ungefär lika många artiklar har publicerats de två senaste tidsintervallen.

4.1 Lösningar som finns

Artiklarna ger flera bra översikter över de olika metoder som finns att tillgå när sårbarheter ska identifieras eller analyseras, och när tillhörande attackkod ska genereras. I [7] delas dessa in *statiska* analysmetoder som inspekterar kod och *dynamiska* analysmetoder som dessutom kör programkod. Statiska analysmetoder bygger på anropsgrafer och/eller använder modeller av programmet för att identifiera data kopplad till sårbarheter. Dynamiska analysmetoder tycks vara en förutsättning för att kunna göra attackkod. Dessa analysmetoder kan grovt delas in i så kallad *fuzzing*, där indata genereras och prövas mot det körande programmet, och *symbolisk exekvering*, där programmet exekveras i en abstrakt modell där faktiska datavärden ersatts med symboliska värden. Tabell 4 sammanfattar de varianter som finns inom dynamisk analys.

Tabell 4. Olika dynamiska analysmetoder, sammanfattade från [7].

Analysmetod	**Beskrivning**
Fuzzing optimerad för kodtäckning	Indata produceras och testas så att så mycket som möjligt av programkoden exekveras. Detta ger konkreta testfall som kraschar programmet.
Fuzzing med kontamineringsanalys (eng. taint-based fuzzing)	Indata produceras och testas samtidigt som det hålls koll på hur indata propageras under exekveringen. Den ”kontaminering” som indata ger upphov till är relevant eftersom den visar vilken indata som påverkar vilka delar av programmets inre logik. På detta sätt går det att noggrannare styra vilken del av koden som analyseras.
Klassisk dynamisk symbolisk exekvering	Sårbarheter identifieras genom att härleda hur olika tillstånd kan uppnås. Detta ger information om vilka kombinationer av indata som kan påverka olika variabler, exempelvis instruktionspekare eller returadresser.
Underbegränsad symbolisk exekvering (eng. under-constrained symbolic execution)	För att klara större program analyseras endast valda delar av programmet. Detta leder till begränsad helhets-förståelse och riskerar att ge falsklarm men ger konkret information om vilka indata som kan påverka olika symboler
Symbolisk exekvering i kombination med fuzzing	Fuzzing är snabb på att testa olika indata men kan begränsas av sin brist på förståelse för programmets logik. Fuzzing kan göras smartare om den kombineras med symbolisk exekvering som förstår programmet. Resultatet är konkret körbar indata som kan påverka viktiga variabler, exempelvis instruktionspekare.

Några av de 22 utvalda artiklarna beskriver samma lösning, men i olika vetenskapliga bidrag. Det finns 17 distinkta lösningar presenterade i artiklarna. De mer framstående beskrivs nedan.

- AEG [8], som är den mest citerade lösningen, använder symbolisk exekvering i kombination med fuzzing och vissa heuristiker när den letar buggar. AEG letar exempelvis extra noga i de delar av programmet där det hittats sårbarheter tidigare.
- CGC-vinnaren Mayhem [9][10][11] har utvecklats av samma personer som gjorde AEG och bygger på samma principer. Mayhem tillhandahålls av företaget For All Secure för testning av mjukvara.

- Crax [12][13] [14] tittar på ”symboliska register” (för returadresser) och kombinerar likt AEG detta med fuzzing. Crax tittar bland annat på om det går att skicka in indata av den storlek som behövs för att kontaminera rätt plats i minnet. Lösningen finns tillgänglig för nedladdning[11].
- Angr [7] löser i huvudsak problemet med att verifiera att buggar är sårbarheter och skapa information som behövs för att manuellt skapa attackkoder. Det gör så genom att implementera analys av kontrollflödet, statisk analys, dynamisk symbolisk exekvering, underbegränsad symbolisk exekvering och symbolisk exekvering i kombination med fuzzing. Dessutom demonstrerar Angr möjligheten att skapa angrepp med samma metoder som AEG. Lösningen finns tillgänglig för nedladdning[12].
- Revery [15] bygger på Angr och gör en enkel symbolisk exekvering i kombination med fuzzing. Revery gör något författarna kallar *control flow stitching*, som går ut på att identifiera hur krascher ska omvandlas till kontroll över exekveringsflödet, exempelvis genom att villkorssatser får värden som styr exekveringsflödet på lämpligt sätt.
- Lösningen i [16] bygger på Angr och tycks använda underbegränsad symbolisk exekvering innan attackkoden verifieras mot binärkoden.
- Flowstitch [17][18] gör dataorienterade angrepp för att läcka eller manipulera minnesinnehåll utan att ta kontroll över exekveringsflödet.

Utöver dessa finns artiklar med liknande bidrag som citerats i mindre utsträckning, inte tycks implementerats i en namngiven produkt eller endast löser ett avgränsat problem inom attackkodsgenerering. Dessa är: [19] som inte lyckades få den genererade attackkoden att fungera ordentligt; [20] som inte innehåller några konkreta tester; [21] och [22] som är inriktade på enklare inbäddade system men annars påminner om ovanstående; [23] som verkar kräva lite handpåläggning av en analytiker för att det ska fungera; [24] och [25] som skapar lösningar för attackkoder att träffa rätt på heapen; [26] som demonstrerar sårbarheter i filsystem med en lösning för att skapa attackkoder; [27] som beskriver olika typer av angrepp och tycks automatgenerera några av dem; och [28] som skapar många olika varianter av attackkoden som gör samma sak.

[11] https://github.com/SQLab/CRAX
[12] https://github.com/angr/angr

4.2 Attackkoder som kan genereras

Som förklarats i avsnitt 2 finns många olika varianter av minnesangrepp. Vissa varianter är enklare att få till, vissa kan kringgå skydd och vissa varianter kräver att den angripna koden är beskaffad på ett visst sätt för att fungera. Wilander et al. [29] beskriver exempelvis in angrepp efter vilken målpekare de har (exempelvis returadressen eller en funktionspekare), tekniken som används (direkt eller indirekt), platsen buffertöverskridningen görs (exempelvis stacken eller datasegmentet), attackkoden som exekveras (exempelvis programkod eller biblioteksanrop) och funktionen som används (exempelvis en formatsträngsfunktion). Alla dessa dimensioner och deras ingående varianter är inte relevanta för att jämföra angreppen som kan genereras i de automatiska lösningarna i denna rapport. Alla identifierade lösningar använder direkta tekniker och gör överskridningar mot stacken och/eller heapen. Tabell 5 ger en översikt över vilka olika typer av angrepp de klarar av utifrån kategorierna i [29]. Kolumnen *Funktion* anger om de funktionstyper som lösningarna stödjer angrepp mot nämns explicit i artikeln. Exempelvis använder Crax det fristående verktyget Libfmtb v0.3 vid arbete med formatsträngar och AEG letar explicit efter formatsträngar.

Nästan alla lösningar fokuserar på enklare former av buffertöverskridningsangrepp. Exempelvis klarar AEG att utföra den enklare formen av stackbaserade buffertöverskridningar där returadressen skrivs över och hanterar formatsträngar i särskild ordning. AEG skriver över returadressen och håller koll på eventuell data i närheten (exempelvis pekare som används efter formatsträngsangreppet). Lösningen är byggd för att fungera på Linux och AEG:s biblioteksanrop görs med en symbolisk länk skriven i en fil, vilket innebär att lösningen kan kringgå minnesrandomisering. Skapandet av filen kräver dock att angriparen har vissa skrivrättigheter på datorn som ska angripas.

Mayhem använder binärkod istället för källkod och fungerar på både Windows och Linux, men kan inte föra in attackkod för att göra biblioteksanrop. Mayhem skriver dock inte bara över returadresser utan kan också använda programhopp av typen `jmp esp` för att hoppa till en känd plats i minnet, en teknik som även används av flera andra lösningar.

Crax kan skapa angrepp av samma typer som Mayhem. Skaparna av Crax visar dessutom att de kan exploatera sårbarheter i tillrättalagda exempel som innebär manipulation av funktionspekare.

Tabell 5. Översikt över vilka typer av angrepp olika lösningar klarar av. ”Ja” betyder att lösningen stödjer angreppsvarianten och”?” betyder att det är möjligt att lösningen stödjer angreppsvarianten men att det är svårt att utläsa ur artikeln.

		Målpekare			**Attackkod**			**Plats**		**Funktion**	
Namn	**Referens(er)**	Direktreferens	Med en trampolin (t.ex. jmp esp)	Funktionspekare	Programkod	Biblioteksanrop	Returorienterad programmering	Heap	Stack	Formatsträngar	Minneskopiering (memcpy)
AEG	[8]	Ja			Ja	Ja			Ja	Ja	
Mayhem	[9][10][11]	Ja	Ja		Ja				Ja		
Crax	[13] [12][14]	Ja	Ja	Ja	Ja	Ja		?	Ja	Ja	
Angr	[7]	Ja			Ja	Ja	Ja		Ja		
Revery	[15]		Ja						Ja		
Xu et al.	[16]	Ja	Ja			Ja			Ja		
Flowstitch	[17][18]	Ja				?		?	Ja		
Honig	[19]			Ja	?				?		
Capella & Jochen	[20]	Ja			Ja				Ja	Ja	
Ruffell et al.	[21] [22]	Ja							Ja		
Padaryan et al.	[23]	?	?						?		
Subramanian	[27]	?				?		?	?	?	?
Wang et al.	[28]		Ja	Ja	Ja				Ja	Ja	Ja

Som framgår i tabell 5 genomför de flesta lösningarna angrepp mot stacken. De som har markeringar i kolumnen för angrepp mot heapen (Crax, Flowstitch och Subramanian) har otydliga beskrivningar av faktisk förmåga och beskriver inte på något tydligt sätt några genomförda tester med angrepp mot heapen. Som nämndes ovan finns andra bidrag [24] [25] som explicit hanterar problemet att träffa rätt på heapen. I [24] presenteras en lösning som kan få heapangrepp att

fungera under antagandet att (1) startläget på heapen är känt, (2) ingen annan arbetar med heapen parallellt och (3) heaphanteraren arbetar deterministiskt. Artikelförfattarna menar att detta kan vara rimliga antaganden vid lokala angrepp. I [25] beskrivs en lösning som tar kraschdata och söker efter sårbarheter i heaphanteraren som kan få angrepp att fungera. Båda dessa skulle kunna integreras i någon av ovanstående lösningar eller utökas för att även leta sårbarheter, men idag erbjuder inte lösningarna någon automatisering av hela processen. Lösningarna i [12] och [14] tycks klara av att skapa hantera angrepp mot heapen givet att en gammal version av ett programbibliotek (glibc 2.3.2) som saknar kontroller för hur pekarvärden används.

Det ska noteras att det var svårt att identifiera vad flera av lösningarna faktiskt klarar av att automatisera. Flera av artiklarna diskuterade olika angreppstekniker, beskrev hur vissa av dem kunde automatiseras och testade någon enstaka variant i ett faktiskt test. Vad som är implementerat och vad som är kvar på ritbordet är oklart i dessa fall. Likaså tycks det ibland krävas handpåläggning för att få allt att fungera, som i lösningen av Padaryan et al. [23] där en analytikers uppgifter ibland nämns. I andra fall redovisas lyckade tester utan att det i lösnings-beskrivningen framgår hur dessa har genomförts görs. I beskrivningarna av Flowstitch [17][18] står exempelvis att angrepp lyckats mot heapen, men det är svårt att avgöra om lösningen för att hitta adresser på heapen är generellt användbar eller om handpåläggning krävts.

4.3 Vilka testfall har de utsatts för

När lösningarna som beskrivs i artiklarna testas varierar det stort vilka mjukvaror som används som testobjekt. Drygt en handfull av artiklarna testar sina lösningar på vad som skulle kunna kallas riktiga programvaror. Med detta avses att det är program som har viss allmän spridning. Tabell 6 listar dessa program och de sårbarheter som hittats i dem.

Eftersom kunskapen om IT-attacker och skydd mot IT-attacker hela tiden utvecklas är det generellt sett lättare att göra fungerande attacker mot äldre mjukvaror. I tabellens kolumn *Ålder på sårbarhet* anges hur många år det skiljer mellan artikelns publicering och sårbarhetens publicering. När detta har räknats fram har ingen hänsyn tagits till när på året sårbarheten eller artikeln publicerades, utan skillnaden som anges är en enkel differens mellan årtalen. Medelvärdet i tabellen är 6,14 år, vilket innebär att sårbarheterna som hittats i programmen i genomsnitt var gamla.

Äldre mjukvaror är kompilerade med äldre kompilatorer och saknar de skydd som modernare kompilatorer för in i koden. De är också mer sällan skrivna för att vara kompatibla med skydd som minnesrandomisering och använder äldre och mer sårbara biblioteksfunktioner. Testerna av Crax [12] görs exempelvis i huvudsak på mjukvara som kompilerats med GCC 4.3.2 (släppt 2008) och använde biblioteket glibc 2.7 (släppt 2007). Skälet är att nyare versioner skyddar main-funktionen mot buffertöverskridning och gör riktighetskontroller på heapen. I de flesta fall beskriver artiklarna hur skydd som minnesrandomisering och dataexekveringsskydd stängts av och flera artiklar anger explicit att de betraktar hanteringen av sådana skydd som framtida arbete.

De sårbarheter som används i artiklarna är sådana som klassificerats som lätta att exploatera i US National Vulnerability Database. Det innebär att de har bedömts ha en attackkomplexitet (eng. *access complexity*) som bland annat innebär att angreppet endast kräver lite skicklighet eller extra informationsinsamling [30].

Kolumnen *exploit redan tillgänglig* visar att det i de flesta fall fanns attackkoder publikt tillgängliga redan. Till det kan det tilläggas att endast AEG [8] faktiskt hittade tidigare okända sårbarheter: en i mjukvaran htget 0.93 och en i expect 5.45. Båda dessa sårbarheter kan exploateras av den som kan bestämma miljövariabler på datorn och kräver alltså någon form av privilegier på det angripna systemet. I [17] genereras angrepp för sårbarheter som andra forskare [30] tidigare identifierat manuellt i SSHD, wu-ftpd och null httpd. Dessa varianter presenteras som tidigare okända angrepp, men var alltså inte helt nya.

Lösningarna har alltså testats på sårbarheter som är kända sedan tidigare, mjukvaror som inte skyddas av moderna skydd och sårbarheter som är relativt enkla att utnyttja.

Tabell 6: Tabell med de program som har en allmän spridning och som varit testobjekt för de mest kvalificerade angreppen.

Program	Antal publika sårbarheter[13]	Testad i artikel	Angripen sårbarhet	Ålder på sårbarhet[14]	Minne	Exploit redan tillgänglig[15]	Används skydd vid test	Attackkomplexitet
Foxit Reader	345	[13][12]	CVE-2009-0837	3	Stack	Ja	Nej	Låg
GhostScript	94	[9]	CVE-2010-2055	2	Stack	Ja	Nej	Låg
htget	1	[12][13] [14] [8][9]	CVE-2004-0852 (Ej inkluderad)	7	Stack	Ja/ Nej	Nej	Låg
IrfanView	136	[28]	CVE-2007-2363	6	Stack	Ja	Nej	Mellan
iwconfig	3	[12][13] [14][8][9]	CVE-2003-0947	8	Stack	Ja	Nej	Låg
KingView (Scada HMI)	13	[25]	CVE-2011-0406	6	Heap	Ja	Nej	Låg
libpng	65	[23][25]	CVE-2004-0597	10	Stack	Ja	Nej	Låg
Mplayer	49	[13][12]	BID 46926	1	Stack	Ja	Nej	?[16]
nginx	47	[17]	CVE-2013-2028	3	Stack	Ja	Nej	Låg
PHP 2013	6075	[31]	CVE-2013-2110	5	Heap	Nej	?	Låg
SSHD	(?[17])	[17]	CVE-2001-0144	15	Stack	Ja	Nej	Låg
sudo	101	[17]	CVE-2012-0809	4	Stack	Ja	Ja[18]	Low
Windows GDI WinXP	20	[25]	CVE-2004-0209	6	Heap	Ja	Nej	Low

[13] Antal CVE-poster i Common Vulnerabilities and Exposures-databasen, oberoende av programversion.

[14] Detta avser tiden från det att en sårbarhet blivit allmänt känd tills det att artikeln publicerades med en lösning som skapar en attackkod för sårbarheten. När en artikel är en vidareutveckling av en annan används den äldsta för att göra jämförelsen.

[15] Anger om det fanns publikt tillgänglig attackkod i exempelvis https://www.exploit-db.com vid tidpunkten då den automatiska lösningen användes.

[16] Komplexitetsmått finns ej då sårbarheten inte är inkluderad in US National Vulnerability Database.

[17] I CVE-databasen är det svårt att urskilja specifika SSH-mjukvaror vilket gör det svårt att räkna antalet relevanta poster. Antalet poster som är kopplade till ”SSH” oberoende av vad det syftar på är 491.

[18] Minnesrandomisering.

Utöver detta tycks det som att noggrant utvalda program med kända sårbarheter som lösningarna klarar av valts som testobjekt. Det indikeras av att så gott som alla sårbarheter i allmänt spridda mjukvaror var kända på förhand och att de inte representerar en uppsättning mjukvaror som är vanliga idag. En annan indikator är att det inte redovisas fall där lösningarna misslyckats med att hitta och exploatera sårbarheter. En tredje indikator är att angrepp endast skapats för en sårbarhet i varje mjukvara. Exempelvis har den version av Nginx som testades i [17] ytterligare en känd sårbarhet (CVE-2013-2070), och för denna skapades ingen attackkod trots att den är lik den sårbarhet som faktiskt exploaterades (CVE-2013-2028). Likaså innehöll den version av Foxit reader som testades åtminstone ett dussin buffertöverskridningssårbarheter som inte hittades av den lösning som testades. Det ska noteras att det utöver mjukvarorna i tabell 6 har utförts tester på specialskrivna programvaror som uttryckligen är anpassade till lösningens funktion och förmåga. I resterande fall var testfallen skapade just för konceptuell testning, som de mjukvaror som användes i Darpas CGC eller liknande.

5 Diskussion och slutsatser

Den genomförda litteraturstudiens resultat har presenterats tillsammans med en grundläggande analys i kapitel 4, tillsammans med en grundläggande analys. Med detta som utgångspunkt finns det några intressanta observationer att göra och ett antal vinklar ur vilka det genomförda arbetet kan belysas.

5.1 Begränsningar i litteratursökningen

I denna studie har det endast tagits med artiklar som behandlar automatisk produktion av fungerande attackkod. Denna typ av artikel beskriver en insats som i stor utsträckning är ingenjörsmässig. Detta framgår exempelvis genom att flera av bidragen beskriver hur de utvecklat lösningar genom att integrera befintliga, tillgängliga komponenter. Ur ett akademiskt perspektiv är ett sådant urval påtagligt begränsande. Den vetenskapliga arbetsmodellen premierar ett djupt, men smalt fokus. Det finns därför en mycket större mängd artiklar som behandlar delar och detaljer hos den typ av lösningar som denna studie intresserar sig för. Exempelvis valdes två tredjedelar av de potentiellt relevanta artiklarna som identifierades i litteraturgenomgången bort, trots att de baserat på titel och kontext bedömdes vara intressanta. Den vanligaste orsaken var att innehållet visserligen var relevant, men att fokus för artikeln var för smalt och innehållet inte sträckte sig hela vägen till att producera fungerande attackkod. Särskilt vanligt var det att ingen faktisk attackkod genererades utan lösningen eller analysen stannade vid att påvisa en sårbarhet. Exempel på ingenjörsmässiga produkter som sorterades bort av denna orsak var:

- Driller [32] som fuzzar och utför dynamisk, symbolisk exekvering iterativt för att hitta indata som kan krascha en mjukvara.
- Exe [33] som kompilerar uppmarkerad källkod och hjälper användare att utföra symbolisk exekvering för att identifiera indata som orsakar en krasch eller fel, som sedan verifieras mot originalprogrammet.
- PrimGen [34] som byggts för att utifrån känd, kraschande indata identifiera vilken indata som ger angripare möjlighet att skriva till minnet och/eller kontrollera instruktionspekaren.
- Semfuzz [35] som utgår från sårbarhetsbeskrivningar i naturligt språk som kan hittas i sårbarhetsdatabaser eller bugglistor. Med denna information som stöd försöker lösningen hitta indata som kraschar mjukvaran.

- Pangr [36] som letar efter sårbarheter med angr (beskriven ovan), analyserar dom med symbolisk exekvering och sedan lagar dem med *patchkit* [19].

Som nämnts i inledningen finns också ett mycket stort antal förslag på hur potentiella sårbarheter ska identifieras utan att verifieras. Sökmetoden som använts i denna studie har inte försökt ringa in alla dessa.

Ytterligare begränsningar är de som tagits upp i avsnitt 1.3, nämligen att (1) endast automation av angrepp mot minneshantering har studerats och (2) endast lösningar som rapporterats i publik forskning har inkluderats. Den första av dessa begränsningar kan adresseras genom ytterligare studier av forskningsfronten. Det tycks finnas en uppsjö lösningar som automatiserar angrepp mot webbsidor, exempelvis Craxweb [13], och en liknande översikt kan göras över dessa. Den andra begränsningen är svårare att hantera, speciellt utan egen forskning i området och möjlighet att erbjuda ett kunskapsutbyte.

5.2 Forskningens kvalitet

De begränsningar som beskrivs avsnitt 5.1 gjorde att de artiklar som studerats i litteraturstudien i hög utsträckning behandlade ingenjörsmässiga lösningar. Bidragen handlar i stor utsträckning om integration och anpassning av befintliga komponenter med målet att bygga ett system som erbjuder faktisk och praktisk funktionalitet. Det är därför naturligt och bra att lösningarnas kvalitet utvärderades genom tester snarare än med teoretisk analys och bevis.

Det är positivt att artiklarna innehåller praktiska tester som på ett tydligt sätt visar både hur svårt problemet är och vilken mognadsgrad området har, sett ur ett tillämpningsperspektiv. Testerna ger dock intrycket av att vara tillrättalagda. Testfallen verkar valda för att ge ett bra resultat. Exempelvis noterades att inga testfall redovisas där lösningarna misslyckats med att generera fungerande attacker, samtidigt som lösningarna lyckas generera exakt en attackkod per testad mjukvara. Detta synes vara ett osannolikt resultat. Om testfallen hade valts förutsättningslöst och alla testfall redovisats i efterhand, borde det funnits både fall där ingen sårbarhet kunde utnyttjas och fall där flera sårbarheter kunde utnyttjas.

Trots den ovan beskrivna ingenjörsmässigheten så var artiklarna publicerade som vetenskapliga artiklar. De flesta innehåller, kanske av just detta skäl, omfattande beskrivningar av vad som skulle vara principiellt möjligt, sammanställningar av värdefulla tekniker och detaljerade beskrivningar av hur specifika sårbarheter går att utnyttja. Tyvärr gör detta det svårt att avgöra om det som beskrivs faktiskt är implementerat och fungerande, eller bara en beskrivning av möjligheter eller

[19] https://github.com/lunixbochs/patchkit

önskvärdheter. Artiklar som vid en initial genomläsning verkade beskriva en mycket kompetent lösning kunde vid en noggrannare genomgång visa sig använda en mycket begränsad repertoar när den testades. Detta gjorde analysen av lösningarnas förmåga betydligt mer krävande än den kunde ha varit och resultatet blev också mindre tydligt än om artiklarna varit tydligare i sin framställning. Avsaknaden av allmänt vedertagen terminologi och taxonomi för buffertöverskridningsattacker bidrar också till svårigheter i tolkningen av resultat. Vilka kolumner som bör finnas i tabell 5 för att beskriva vilka typer av attacker lösningarna klarar är exempelvis inte uppenbart.

5.3 Områdets mognad

Området automatiserad attackkodsgenerering har behandlats inom den akademiska arenan sedan åtminstone 2002. Det fanns flera stora, ingenjörsmässiga projekt beskrivna som satt samman lösningar för att automatiskt skapa attackkoder. Trots detta var de lösningar som beskrevs relativt omogna. De riktade sig mot de allra mest grundläggande och bäst förstådda sårbarheterna och de kunde i allmänhet inte hantera ens de vanligaste skydden – skydd som har använts i både Windows och Linux sedan flera år.

När artiklarna beskrev hur lösningarna hade testats var programmen som hade använts som testfall nästan alltid tillrättalagda på något sätt. I sin tydligaste form tog detta sig uttryck i att testprogrammen hade specialskrivits för lösningarna, med precis rätt sorts sårbarheter och andra förutsättningar. Det kunde också handla om att testprogrammen hade kompilerats och länkats med föråldrade kompilatorer eller bibliotek. När allmänt spridda program hade använts för tester så hade nästan uteslutande redan kända sårbarheter angripits. Den äldsta sårbarheten som hade angripits var 15 år gammal vid artikelns publiceringstillfälle! Den förmåga hos lösningarna som artiklarnas författare själva hade valt att visa upp, och som rimligen valts för att visa lösningarna ur en positiv synvinkel, kunde alltså sammanfattas med att de hanterade

- sårbarheter som var kända sedan tidigare
- mjukvaror som inte skyddades av allmänt tillgängliga, moderna skydd
- sårbarheter som var relativt enkla att utnyttja.

Försöker man att se styrkan i de bästa av de presenterade lösningarna går det istället att säga att de visserligen är svagare än en mänsklig penetrationstestare, men att de

- kan analysera stora mängder kod snabbt
- kan vara en helautomatiserad lösning i en miljö där inga penetrationstestare finns tillgängliga.

Författarna bedömer att de tre lösningar som var mest mogna var CRAX, Angr och Mayhem/AEG och att de låg ungefär på TRL-nivå 5 (tekniken validerad i relevant miljö) när artiklarna publicerades. Denna bedömning gäller dock just dessa lösningar, just som de beskrevs i de artiklar som analyserats. Åtminstone Mayhem/AEG har kommersialiserats och därmed sannolikt vidareutvecklats påtagligt. Det är också tänkbart att det finns lösningar som inte har beskrivits i den öppna litteraturen. Exempelvis tyder Darpas visade intresse på att området är relevant för militära aktörer och dessa har inte samma behov av öppenhet som akademiska aktörer. Det kan således finnas en betydande kunskapsmassa inom området som inte blottlagts i denna skannande studie.

En observation som gjorts under studien är att det inte finns några nyckelfärdiga lösningar för automatisk attackkodsgenerering för vanliga operativsystem att hämta på internet. Det finns en del tillgängligt, men det har mer karaktär av ramverk eller delkomponenter som skulle kunna utgöra delar av ett fullständigt system. Biblioteket Libfmtb v0.3 som nämndes ovan är ett exempel på en sådan delkomponent. En annan observation är att den tävling Darpa höll (Cyber Grand Challenge) gett en rejäl knuff framåt, både för denna typ av forskning och för kommersiell verksamhet inom området. Samtidigt var de testfall och mjukvaror som användes i tävlingen förenklade och tillrättalagda. Det krävdes inte heller att lösningarna faktiskt tog kontroll över den angripna datorn utan räckte att det påvisades att förutsättningar för sådan kontroll fanns.

Sammantaget gör författarna bedömningen att det krävs en gedigen och omfattande insats för att skapa en kompetent lösning som kan skapa kvalificerad attackkod för riktig mjukvara. Det som finns beskrivet i den akademiska litteraturen visar på principiella möjligheter, men det är långt kvar innan allmänt tillgängliga lösningar utgör en viktig källa till effektiva attackkoder.

5.4 Framtida arbete

Av den genomförda studien framgår att teknikområdet automatisk attackkods-generering har nått en mognadsnivå som ligger på gränsen till att vara praktiskt användbar. De publikt beskrivna lösningarna är relativt enkla, men det framstår samtidigt som möjligt att utveckla och förbättra lösningarna så att de också klarar att generera mer avancerade angrepp för mer komplexa sårbarheter.

Som forskningsområde är automatisk attackkodsgenerering förhållandevis omoget och det finns stor förbättringspotential i hur lösningar beskrivs, testas och relateras till varandra. Ett allmänt vedertaget ramverk för att klassificera och testa olika lösningar skulle troligtvis leda till avsevärda förbättringar både i hur forskningen inriktas och presenteras. I det goda fallet skulle ett sådant ramverk klassificera angrepp enligt viktiga dimensioner och erbjuda meningsfulla metriker för att jämföra dem med varandra och manuella lösningar, exempelvis genom att mäta hur komplexa sårbarheter de lyckas utnyttja.

Det finns flera skäl till varför Försvarsmakten borde vara intresserad av fortsatt forskning inom detta område.

- Området kräver konkret och riktig kunskap om angrepp. Sådan kunskap är värdefull i många sammanhang och skulle därför generera nytta långt utanför ett specifikt forskningsprojekt. Kunskapen kan exempelvis vara till nytta när attackkoder ska analyseras eller sårbarheter ska prioriteras.
- Det finns inga uppenbara teoretiska eller praktiska hinder för att det ska gå nå framgång i denna typ av forskning. De lösningar som identifierats i den förevarande studien fångar delar av de heuristiker, trix och procedurer som används i manuellt arbete och detta går sannolikt att fånga i betydligt större omfattning.
- För test och förmågeutvärdering finns lättillgänglig och bra data i form av sårbarhetsdatabaser med tiotusentals väldokumenterade sårbarheter i allmänt tillgänglig mjukvara. God tillgång till testdata är ovanligt i IT-säkerhetssammanhang och ger forskningen ett bra utgångsläge för praktisk validering. I det här fallet är det dessutom verklig data.
- Forskning inom området kan resultera i en teknisk artefakt i form av program som automatiskt genererar attackkod som är anpassad till den mjukvara som analyseras. Detta gör kunskapen konkret, lättanvänd och enkel att föra över till andra.

Dessutom torde förmågan till automatisk generering av attackkoder vara av stort intresse för Försvarsmakten. Ur ett defensivt perspektiv kan automatiserad attackkodsgenerering exempelvis vara värdefull genom att vara ett stöd vid prioriteringar av sårbarheter. De fall där attackkoder kan genereras automatiskt bör, allt annat lika, prioriteras framför de fall där attackkoder inte kan genereras. Ur ett offensivt perspektiv skulle automatiserad attackkodsgenerering kunna ge stora mängder fungerande attackkoder på kort tid, vilket skulle ge ett försprång framför dem som vill försvara program och system.

6 Författarnas tack

Författarna vill tacka Bodo Endres för dennes arbete med att kartlägga vilka mjukvaror som testats.

Referenser

[1] T. Sommestad, H. Holm, and M. Ekstedt, “Effort Estimates for Vulnerability Discovery Projects,” in *2012 45th Hawaii International Conference on System Sciences*, 2012, pp. 5564–5573.

[2] I. Rodhe and M. Karresand, “Overview of formal methods in software engineering (FOI-R--4156--SE),” Linköping, Sweden, 2015.

[3] C. Bildsten, M. Cohen, D. Eidenskog, and I. Rodhe, “Praktisk mjukvaruverifiering med formella metoder. Ett hjälpmedel i granskningsprocessen (FOI-R--4642--SE),” Linköping, 2018.

[4] T. N. Brooks, “Survey of Automated Vulnerability Detection and Exploit Generation Techniques in Cyber Reasoning Systems,” in *Intelligent Computing*, K. Arai, S. and Kapoor, and R. and Bhatia, Eds. Cham: Springer International Publishing, 2017, pp. 1083--1102.

[5] T. Ji, Y. Wu, C. Wang, X. Zhang, and Z. Wang, “The Coming Era of AlphaHacking?: A Survey of Automatic Software Vulnerability Detection, Exploitation and Patching Techniques,” in *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*, 2018, pp. 53–60.

[6] Zardus et al., “Cyber Grand Shellphish,” 2007. [Online]. Available: http://phrack.org/papers/cyber_grand_shellphish.html. [Accessed: 05-Feb-2019].

[7] Y. Shoshitaishvili *et al.*, “SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis,” in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157.

[8] T. Avgerinos, S. K. Cha, B. Lim, T. Hao, and D. Brumley, “AEG : Automatic Exploit Generation,” in *Network and Distributed System Security Symposium*, 2011.

[9] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, “Unleashing Mayhem on Binary Code,” in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 380–394.

[10] T. Avgerinos *et al.*, “The Mayhem Cyber Reasoning System,” *IEEE Secur. Priv.*, vol. 16, no. 2, pp. 52–60, Mar. 2018.

[11] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, “Automatic exploit generation,” *Commun. ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[12] S. K. Huang, M. H. Huang, P. Y. Huang, H. L. Lu, and C. W. Lai, “Software crash analysis for automatic exploit generation on binary programs,” *IEEE Trans. Reliab.*, vol. 63, no. 1, pp. 270–289, 2014.

[13] S.-K. Huang, M.-H. Huang, P.-Y. Huang, C.-W. Lai, H.-L. Lu, and W.-M. Leong, “CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations,” in *2012 IEEE Sixth International Conference on Software Security and Reliability*, 2012, pp. 78–87.

[14] P.-Y. Huang, “Automated Exploit Generation for Control-Flow Hijacking Attacks,” National Chiao Tung University, 2011.

[15] Y. Wang *et al.*, “Revery,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS ’18*, 2018, pp. 1914–1927.

[16] L. Xu, W. Jia, W. Dong, and Y. Li, “Automatic Exploit Generation for Buffer Overflow Vulnerabilities,” in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2018, pp. 463–468.

[17] H. Hong, “Systematic Methods for Memory Error Detection And Exploitation,” National University of Singapore, 2016.

[18] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, “Automatic Generation of Data-Oriented Exploits,” in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 177–192.

[19] J. Honig, “Autonomous Exploitation of System Binaries using Symbolic Analysis,” Twente, 2017.

[20] S. Cappella and M. Jochen, “Automated Exploit Generation of Binary Targets By Leveraging a Custom Fuzzing and Debugging Framework,” in *Annual Spring conference of the Pennsylvania Computer and Information Science Educators (PACISE)*, 2013.

[21] M. Ruffell, J. B. Hong, H. Kim, and D. S. Kim, “Towards Automated Exploit Generation for Embedded Systems,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10144 LNCS, 2017, pp. 161–173.

[22] M. Ruffell, “Applying Bytecode Level Automatic Exploit Generation to Embedded Systems,” University of Canterbury, 2015.

[23] V. A. Padaryan, V. V. Kaushan, and A. N. Fedotov, “Automated exploit generation for stack buffer overflow vulnerabilities,” *Program. Comput. Softw.*, vol. 41, no. 6, pp. 373–380, 2015.

[24] S. Heelan, T. Melham, and D. Kroening, “Automatic Heap Layout Manipulation for Exploitation,” in *Proceedings of the 27th USENIX Conference on Security Symposium*, 2018, pp. 763–779.

[25] D. Repel, J. Kinder, and L. Cavallaro, “Modular Synthesis of Heap Exploits,”

in *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security - PLAS '17*, 2017.

[26] Junfeng Yang, Can Sar, P. Twohey, C. Cadar, and D. Engler, "Automatically generating malicious disks using symbolic execution," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*, 2006.

[27] S. E. Subramanian, "Bit-vector Support in Z3-str2 Solver and Automated Exploit Synthesis," University of Waterloo, Canada.

[28] M. Wang, P. Su, Q. Li, L. Ying, Y. Yang, and D. Feng, "Automatic Polymorphic Exploit Generation for Software Vulnerabilities," in *Security and Privacy in Communication Networks.*, Cham: Springer, 2013, pp. 216–233.

[29] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: Runtime Intrusion Prevention Evaluator," in *Proceedings of the 27th Annual Computer Security Applications Conference on - ACSAC '11*, 2011, pp. 41–50.

[30] P. Mell, K. Scarfone, and S. Romanosky, "A Complete Guide to the Common Vulnerability Scoring System (CVSS), Version 2.0," 2007.

[31] S. Heelan, T. Melham, and D. Kroening, "Automatic Heap Layout Manipulation for Exploitation," in *Proceedings of the 27th USENIX Conference on Security Symposium*, 2018, pp. 763–779.

[32] N. Stephens *et al.*, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Proceedings 2016 Network and Distributed System Security Symposium*, 2016.

[33] C. Cadar, P. Twohey, V. Ganesh, and D. Engler, "EXE: A System for Automatically Generating Inputs of Death Using Symbolic Execution," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2006, vol. 43, pp. 199–209.

[34] B. Garmany, M. Stoffel, R. Gawlik, P. Koppe, T. Blazytko, and T. Holz, "Towards Automated Generation of Exploitation Primitives for Web Browsers," in *Proceedings of the 34th Annual Computer Security Applications Conference on - ACSAC '18*, 2018, pp. 300–312.

[35] W. You *et al.*, "SemFuzz," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*, 2017, pp. 2139–2154.

[36] D. Liu *et al.*, "Pangr: A Behavior-Based Automatic Vulnerability Detection and Exploitation Framework," in *Proceedings - 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications and 12th IEEE International Conference on Big Data Science and Engineering, Trustcom/BigDataSE 2018*, 2018, pp. 705–712.

Appendix A – Litteratursökningsresultat

I tabellen nedan visas de 94 bidrag som identifierades i litteratursökningen. De 22 artiklar som valdes ut som relevanta är markerade med grå bakgrund.

Författare	Titel	Årtal
G. Anapathy et al.	Automatic discovery of API-level exploits	2005
J. Yang et al.	Automatically generating malicious disks using symbolic execution.	2006
C. Cadar et al.	EXE: A system for automatically generating inputs of death using symbolic execution.	2006
C. Cadar et al.	EXE: automatically generating inputs of death	2006
D. Brumley	A Binary-Centric Approach to Vulnerability Analysis and Defense	2007
J. Medeiros.	Automated exploit development: The future of exploitation is here	2007
Grenier, L. (Pusscat and Lin0xx).	Byakugan Automating exploitation. In ToorCon Seattle (Seattle, WA, May 2007); http://seattle.toorcon.net/	2007
D. Engler, D. Dunba	Under-constrained execution: making automatic code destruction easy and scalable.	2007
D. Brumley	Analysis and defense of vulnerabilities in binary code	2008
D. Brumley et al.	Automatic patch-based exploit generation is possible: Techniques and implications	2008
M. Martin M. S. Lam.	Automatic generation of xss and sql injection attacks with goal-directed model checking.	2008
D. Song et al.	BitBlaze: A New Approach to Computer Security via Binary Analysis	2008
Z. Lin, X. Zhang. D. Xu	Convicting exploitable software vulnerabilities: An efficient input provenance based approach	2008
C Cadar et al.	EXE: automatically generating inputs of death	2008
A. Kieyzun et al.	Automatic Creation of SQL Injection and Cross-Site Scripting Attacks.	2009
S. Heelan D. Kroening	Automatic Generation of Control Flow Hi-jacking Exploits for Software Vulnerabilities.	2009
Y.H. Wang, C.H. Mao, H.M. Lee	Structural learning of attack vectors for generating mutated xss attacks	2010
T. Avgerinos et al.	AEG: Automatic exploit generation	2011
P-Y. Huang	Automated Exploit Generation for Control-Flow Hijacking Attacks	2011

Chipounov, V., Kuznetsov, V., Candea, G.	S2E: A platform for in-vivo multi-path analysis of software systems	2011
J. DeMott, R. Enbody, W. Punch,	Towards an automatic exploit pipeline	2011
B. Lim, T. Hao, D. Brumley	Automatic heap exploit generation	2012
S. Huang et al.	CRAX: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations	2012
V. Chipounov,.V. Kuznetsov, G. Candea	S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems	2012
S. Cha et al.	Unleashing Mayhem on binary code	2012
S. Cappella, M. Jochen	Automated exploit generation of binary targets by leveraging a custom fuzzing and debugging framework	2013
T. Houtenbos, D. Pellikaan	Automated vulnerability scanning and exploitation	2013
M. Wang et al.	Automatic Polymorphic Exploit Generation for Software Vulnerabilities	2013
S. Huang et al.	CRAXweb: Automatic Web Application Testing and Attack Generation.	2013
D. Caselden et al.	HI-CFG: Construction by Binary Analysis and Application to Attack Polymorphism	2013
J. Vanegue	The automated exploitation grand challenge. InH2HC 2013 (2013) https://www.youtube.com/watch?v=AJKp0C5EHww	2013
D. Caselden et al.	Transformation-aware Exploit Generation using a HI-CFG.	2013
H. Li, et al.	A scalable approach to vulnerability discovery based on security patches	2014
M. Zhang, H. Yin.	AppSealer: Automatic Generation of VulnerabilitySpecific Patches for Preventing Component Hijacking Attacks in Android Applications.	2014
T. Avgerionos et al.	Automatic exploit generation	2014
S. Letian et al.	Pvdf: an automatic patch-based vulnerability description and fuzzing method	2014
S. Huang et al	Software crash analysis for automatic exploit generation on binary programs	2014
D. Gallingani	Static detection and automatic exploitation of intent message vulnerabilities in android applications	2014
M. Ruffell	Applying Bytecode Level Automatic Exploit Generation to Embedded Systems	2015

V. A. Padaryan, V. V. Kaushan, A. N. Fedotov	Automated exploit generation for stack buffer overflow vulnerabilities	2015
H. Hu et al.	Automatic Generation of Data-Oriented Exploits	2015
S. Subramanian	Bit-vector support in z3-str2 solver and automated exploit synthesis	2015
O. Oswaldo, I. Dilling, C. Lin	Detecting and Exploiting Second Order Denial-of-Service Vulnerabilities in Web Applications	2015
D. Brumley et al.	Detecting exploitable bugs in binary code	2015
J. Vanegue	Heap models for exploit systems. In LangSec Workshop 2015 (2015). Invited talk.	2015
Trail of Bits	https://blog.trailofbits.com/2015/07/15/how-we-fared-in-the-cyber-grand-challenge/	2015
M. Walker	Machine vs. Machine: Lessons from the First Year of Cyber Grand Challenge	2015
J. Song, J. Alves-Foss	The DARPA Cyber Grand Challenge: A Competitor's Perspective	2015
Y. Shoshitaishvili et al.	(State of) The art of war: offensive techniques in Binary Analysis	2016
A. Alhuzali et al.	Chainsaw: Chained automated workflow-based exploit generation	2016
L. Luo et al.	Context-aware System Service Call-oriented Symbolic Execution of Android Framework with Application to Exploit Generation	2016
N. Stephens et al.	Driller: Augmenting Fuzzing Through Selective Symbolic Execution	2016
FORALLSECURE	https://forallsecure.com/blog/2016/02/09/ unleashing-mayhem/,	2016
B.D. Brumley	https://forallsecure.com/blog/2016/07/26/why-cgc-matters-to-me/	2016
S. D'Antoine	Practical Uses of Program Analysis: Automatic Exploit Generation https://www.youtube.com/watch?v=d3fy4se4JO0	2016
Team Shellphish	Shellphish https://www.youtube.com/watch?v=tEtlFUEWrEY	2016
H. Hong	Systematic methods for memory error detection and exploitation	2016
J. Song, J. Alves-Foss	The DARPA Cyber Grand Challenge: A Competitor's Perspective, Part 2	2016
M. Ruffell, J. Hong, H. Kim, D. Kim	Towards Automated Exploit Generation for Embedded Systems	2016
Y. P. Wan et al.	Automatic exploit generation system based on symbolic execution	2017

J .Garcia, M. Hammad, N. Ghorbani	Automatic generation of inter-component communication exploits for Android applications	2017
J. Honig	Autonomous Exploitation of System Binaries using Symbolic Analysis	2017
F. Tang , C.Feng, C. Tang	Binary vulnerability exploitability analysis	2017
D. Brumley et al.	Detecting exploitable bugs in binary code	2017
G. Yan et al	Exploitmeter combining fuzzing with machine learning for automated evalutaion of software exploitability	2017
Team Shellphish	http://phrack.org/papers/cyber_grand_shellphish.html	2017
D. Repel, J. Kinder, L. Cavallaro	Modular Synthesis of Heap Exploits	2017
Y. Shoshitaishvili et al.	Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance	2017
W. You et al.	Semfuzz: Semantics-based automatic generation of proof-of-concept exploits	2017
Z. Xu et al.	Spain: security patch analysis for binaries towards understanding the pain and pills	2017
T. N. Brooks	Survey of automated vulnerability detection and ex-ploit generation techniques in cyber reasoning systems	2017
L. Luo et al	System service call-oriented symbolic execution of android framework with applications to vulnerability discovery and exploit generation	2017
K. Lu et al.	Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying	2017
K. Ispoglou et al.	Block Oriented Programming: Automating Data-Only Attacks	2018
R Zhang, C Sturton	A recursive strategy for symbolic execution to find exploits in hardware designs	2018
M Schwarz et al.	Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features	2018
G. Yang, J. Huang, G. Gu	Automated generation of event-oriented exploits in android hybrid apps	2018
L. Xu et al.	Automatic Exploit Generation for Buffer Overflow Vulnerabilities	2018
S. Heelan, T. Melham, D. Kroening	Automatic Heap Layout Manipulation for Exploitation	2018
R Zhang, et al.	End-to-End Automated Exploit Generation for Validating the Security of Processor Designs	2018
W. Wu, X. Xing, J. Su	From Thousands of Hours to a Couple of Minutes: Towards Automating Exploit Generation for Arbitrary Types of Kernel Vulnerabilities	2018

W. Wu, et al.	FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities	2018
S. Heelan, T Melham, D Kroening	Heap layout optimisation for exploitation	2018
S. D'Antoine	https://www.sophia.re/ML/index.html	2018
Y. Shoshitaishvili et al.	Mechanical Phish: Resilient Autonomous Hacking	2018
A. Alhuzali et al.	NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications	2018
D. Liu et al.	Pangr: A Behavior-Based Automatic Vulnerability Detection and Exploitation Framework	2018
Y. Wang et al.	Revery: From Proof-of-Concept to Exploitable	2018
J. Krupp, C. Rossow	TEETHER: Gnawing at Ethereum to Automatically Exploit Smart Contracts	2018
T. Ji, et al.	The Coming Era of AlphaHacking?	2018
T. Avgerinos et al.	The Mayhem Cyber Reasoning System	2018
B. Garmany et al.	Towards Automated Generation of Exploitation Primitives for Web Browsers	2018
A. Nguyen-Tuong, D. Melski, J.W. Davidson	Xandra: An Autonomous Cyber Battle System for the Cyber Grand Challenge	2018
Team Shellphish	Rex - Automated Exploitation Engine." [Online]. Available: https://github.com/shellphish/rex	u.å.

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