Verktyg som döljer skadlig kod

En systematisk granskning

HANNES HOLM, ERIK HYLLIENMARK, MATS PERSSON

FOI-R--5366--SE
ISSN 1650-1942 December 2022

Hannes Holm, Erik Hyllienmark, Mats Persson

Verktyg som döljer skadlig kod

En systematisk granskning

Titel	Verktyg som döljer skadlig kod – En systematisk granskning
Title	Tools that hide malicious code - A systematic review
Rapportnr/Report no	FOI-R--5366--SE
Månad/Month	December
Utgivningsår/Year	2022
Antal sidor/Pages	36
ISSN	1650-1942
Uppdragsgivare/Client	Försvarsmakten
Forskningsområde	Informationssäkerhet
FoT-område	Operationer i cyberdomänen
Projektnr/Project no	E716589
Godkänd av/Approved by	Christian Jönsson
Ansvarig avdelning	Ledningssystem

Bild/Cover: Midjourney

Sammanfattning

Denna rapport beskriver en systematisk granskning av verktyg som döljer skadlig kod. Verktygen identifierades via Github, en databas som huvudsakligen innefattar mjukvaruprojekt skrivna som öppen källkod.

Totalt kategoriserades 174 verktyg enligt fyra huvudkategorier – allmän projektinformation, vilka arkitekturer och filformat som stödjs, vilka försvar mot statisk granskning som erbjuds, samt vilka försvar mot dynamisk granskning som erbjuds.

Resultatet visade att de allra flesta verktyg tillämpade en applikation för att kryptera den skadliga koden, och en annan applikation för att dekryptera och exekvera den på en dator. Det visade också att de flesta verktyg troligen skapades i utbildningssyfte snarare än för att effektivt dölja skadlig kod.

Nyckelord: Systematisk granskning, skadlig kod, obfuskering

Summary

This report describes a systematic review of tools available on Github that can prevent the discovery of malicious code.

A total of 174 tools were categorized according to four overall categories: project metadata, supported architectures and file formats, defence against static analysis, and defence against dynamic analysis.

The results showed that most of the tools employed one application to encrypt the malicious code, and a loader application to decrypt and execute the obfuscated code on a target machine. They also showed that most tools were created for educational purposes rather than to be effective.

Keywords: Systematic review, malware, obfuscation

Innehållsförteckning

1 Inledning

Att delta i en cybersäkerhetsövning ökar förmågan gällande dator- och nätverksförsvar [1]. Det finns många varianter på cybersäkerhetsövningar, varav en av de vanligare varianterna är att utsätta försvarare av datorer och nätverk för cyberangrepp från simulerade hotaktörer.

Att bedriva en cybersäkerhetsövning är dock en mycket kostsam process. I synnerhet ställer utförande av cyberangrepp stora krav på specialistkompetens och arbetstid.

Inom projektet *Verktyg och Experiment för Computer Network Operations* (VECNO) har ett verktyg kallat Lore [1] utvecklats för automatisering av cyberangrepp under cybersäkerhetsövningar. Lore kräver ingen användarinteraktion och klarar de flesta typer av relevanta handlingar, såsom nätverkskartläggningar, serverangrepp, lösenordsgissning och skalkommandon på fjärrstyrda maskiner [1].

Hittills har Lore använts under cybersäkerhetsövningarna SAFE Cyber 2020 [2] (under mars 2021), Cybon[1] (under maj 2022) samt SAFE Cyber 2022[2] (under september 2022). Dessa övningar omfattade cirka 120 försvarare av datorer och nätverk.

Automatisering av simulerade hotaktörer är en komplicerad uppgift som både kräver omfattande forskning och utveckling av programkod. Merparten av koden som utvecklas av FOI för Lore involverar prioritering och exekvering av sekvenser med angreppsaktiviteter. Specifika verktyg som används för olika angreppsprocesser, såsom verktyg som kartlägger nätverkssegment eller utför angrepp mot mjukvarusårbarheter, inkorporeras i form av tredjeparts-projekt.

Ett viktigt tredjepartsverktyg som används av Lore är bakdörren Meterpreter[3], den i vår vetskap funktionsmässigt mest omfattande bakdörren som finns tillgänglig som öppen källkod. Meterpreter möjliggör enkel exekvering av diverse komplexa funktioner och har i skrivande stund stöd för cirka 1000 typer av specialiserade kommandon, såsom inbyggd nätverksrouting, extrahering av lösenordshashar, och eskalering av privilegier. Utan Meterpreter kan Lore enbart utföra ett fåtal enklare skalkommandon, och kan exempelvis inte eskalera behörigheter på, eller vidarebefordra trafik genom, en övertagen maskin.

[1] “Cyber Defence Exercise - Cybon 2022”, FOI Memo under arbete.

[2] https://www.forsvarsmakten.se/sv/aktuellt/2022/09/cybersoldaternas-forsta-repovning/

[3] https://github.com/rapid7/metasploit-payloads/tree/master/c/meterpreter

Moderna IT-miljöer innefattar i regel diverse skyddsmekanismer såsom antivirus, brandväggar och nätverksintrångsdetektionssystem för att identifiera och stoppa skadliga koder [3]. Att upptäcka Meterpreter är särskilt högt prioriterat för utvecklare av säkerhetsmekanismer på grund av dess popularitet. Detta medför att Lore inför 2022 inte fungerade väl i moderna maskiner med uppdaterade antivirusprogram, då dessa oftast upptäckte och stoppade dess Meterpreter-bakdörrar på fjärrstyrda maskiner.

Ett relativt enkelt sätt att undgå säkerhetsmekanismer är att bygga en helt ny bakdörr, och vid behov modifiera den tills den inte längre upptäcks. Att utveckla en helt ny bakdörr med liknande modulstöd som Meterpreter är dock ett omfattande projekt som kräver mycket tid och kunskap. Av denna anledning har praktiker och akademiker utvecklat diverse metoder och verktyg för att dölja bakdörrar på ett sådant sätt att de inte upptäcks och förhindras av säkerhetsmekanismer. Denna rapport beskriver det arbete som utfördes inom VECNO för att identifiera sådana verktyg som kan tillämpas av Lore under cybersäkerhetsövningar för att dölja Meterpreter. Detta arbete sammanfattas med forskningsfråga 1:

> **Fråga 1**: *Vilka verktyg finns det som kan dölja kända bakdörrar från säkerhetsmekanismer?*

Det finns många metoder för att motverka upptäckt av kända bakdörrar [3]–[9]. De identifierade verktygen kategoriserades enligt vanliga sådana metoder för att identifiera deras övergripande funktioner. Detta sammanfattas med forskningsfråga 2:

> **Fråga 2**: *Vilka metoder tillämpas av olika verktyg för att dölja kända bakdörrar från säkerhetsmekanismer?*

Rapportens struktur är följande: Kapitel 2 presenterar tidigare utförd forskning kring forskningsfrågorna. Kapitel 3 presenterar metoden som nyttjades för att identifiera och kategorisera verktyg. Kapitel 4 presenterar själva kategoriseringsmodellen. Kapitel 5 presenterar studiens resultat. Slutligen presenterar kapitel 6 slutsatser och framtida arbete.

Läsare av kapitel 2-5 förväntas ha god kunskap kring programvaruutveckling i allmänhet och programvaruutveckling med IT-säkerhetstillämpning i synnerhet. Övriga läsare rekommenderas läsa kapitel 6.

2 Tidigare forskning

Tidigare forskning kring att specifikt dölja bakdörrar är sparsmakad. Forskning kring att dölja skadlig kod i allmänhet är däremot omfattande. Medan det kan finnas principiella skillnader mellan olika typer av skadliga koder är processen för att upptäcka de likartad [6].

Ett rättframt sätt att kringgå skyddsmekanismer är att bygga en helt ny bakdörr och vid behov modifiera den tills den ej längre upptäcks. En sådan enklare bakdörr har dock inte samma funktionalitet som Meterpreter, och dess tillämpning under en cyberoperation kan därför ta anspråk på betydligt mer arbetstid. Som jämförelse kan ett enda Meterpreter-kommando involvera tusentals rader kod, och Lore kan tillämpa mer än 100 typer av kommandon under en operation.

Av resursskäl är utveckling av en helt ny bakdörr för varje ny cyberoperation sällan ett alternativ, och än mindre ett alternativ för Lore. Det är dyrt att producera bra kod och det är därför önskvärt att koden vidareutvecklas snarare än skrivs om inför varje nyttjandetillfälle.

Av denna anledning har diverse metoder och verktyg utvecklats för att dölja skadlig kod [3]. Det finns diverse översiktsstudier av sådana metoder (t.ex. [3]–[9]). Dessa översiktsstudier låg till grund för kategoriseringsmodellen som beskrivs i kapitel 4.

Resterande del av detta kapitel beskriver akademisk forskning som, likt denna rapport, innefattar någon typ av empirisk analys av sätt att kringgå säkerhetsmekanismer. Dessa identifierades genom sökningar på Google Scholar utförda baserade på forskarnas erfarenhet.

Chen m.fl. [10] granskade vilka tekniker som skadliga koder tillämpar för att försvåra analysarbetet. Sådana tekniker involverar generellt att detektera om avlusare (eng: debugger) övervakar exekveringen, samt att utvärdera huruvida exekveringen sker i en emulerad miljö. Författarna granskade 16 246 generella och 1037 riktade skadliga koder (eng: advanced persistent threat, APT), och identifierade bland annat att riktade skadliga koder mer sällan upptäcks, men att de inte använder fler eller mer avancerade tekniker för att försvåra analysarbetet.

Hammad m.fl. [11] studerade huruvida 22 olika antivirusprogram upptäckte skadlig kod på Android-plattformen. Författarna obfuskerade 3000 riktiga applikationer och 3000 skadliga koder genom 11 typer av tekniker tillhandahållna av sex olika verktyg. Exempel på tekniker var att signera filer med andra certifikat eller lägga till extrainstruktioner som inte påverkade applikationers tilltänkta funktionalitet. Totalt testades 29 kombinationer av de 11 teknikerna. Resultaten visade att kodobfuskering var ett effektivt sätt att kringgå antivirusprogram, även givet de enklaste obfuskeringsvarianterna. De visade

också att kombinationer av flera obfuskeringstekniker inte minskade sannolikheten för upptäckt.

Morales m.fl. [12] studerade huruvida sex olika antivirusprogram kunde detektera 500 olika skadliga koder. Deras resultat visade att enskilda antivirusprogram ofta misslyckas med att fullständigt förhindra skadliga koder (dvs., detektera och stoppa all skadlig funktionalitet).

Sukwong m.fl. [13] undersökte om antivirusprograms förmåga att upptäcka skadlig kod beror på huruvida antivirusprogrammets regelverk är uppdaterat. I det fall ett regelverk är äldre än den skadliga koden kan den senare betecknas som en ”zero-day”, då antivirusutvecklare haft 0 dagar på sig att analysera koden. Som grund i analysen tillämpade författarna 1115 skadliga koder och sex antivirus med regelverk uppdaterade till olika datum. Resultaten visade att antivirus detekterade zero-days med en sannolikhet mellan 40%-60%, och att detektionssannolikheten ökade baserat på tiden som en skadlig kod varit känd (givet ett uppdaterat regelverk). Skadliga koder som varit kända i 30 dagar upptäcktes nästan alltid.

Mills och Legg [14] utförde en studie lik [10] vilken involverade analyser av 251 skadliga koder med hjälp av ett verktyg utvecklat av författarna. Flera typer av egenskaper som skadlig kod nyttjar identifierades, såsom särskilda processer, filer och programmeringsgränssnitt (eng: Application Programming Interface, API).

Galloro m.fl. [15] granskade hur 45 375 skadliga koder tillämpade 92 olika tekniker som försvårar upptäckt. Dessa tekniker involverade t.ex. tillämpning av tester gällande felhantering, timing, minnesregioner, mänsklig interaktion och Windows-registerfunktionalitet. Resultaten visade att tillämpning av de studerade teknikerna hade ökat över tid, och att det fanns tydliga trender för när nya tekniker ökade, och gamla tekniker minskade, i popularitet. Resultaten visade också att 15 av de granskade teknikerna inte utnyttjades av någon skadlig kod.

Avllazagaj m.fl. [16] studerade 7,6 miljoner exekverade skadliga koder på 5,4 miljoner datorer i 113 länder. Författarna granskade sekvenser med API-anrop mot olika operativsystemsfunktioner för dessa exekveringar för att identifiera mönster. Resultaten visade att skadliga koder har ett mer varierat beteende än godartade koder. De visade också att skadlig kod som inte upptäcks har ett mindre varierat beteende än skadlig kod som upptäcks.

Barr-Smith m.fl. [17] studerade hur cirka 32 miljoner skadliga koder tillämpade binärer som redan tillhandahålls av målsystemen (t.ex. Powershell, rundll32 eller WMIC). Resultaten visade att 9,4% av de studerade skadliga koderna tillämpder binärer på målsystemet, och att siffran var högre (26,3%) för riktad skadlig kod (APT).

Som framgår av ovan beskrivna arbeten har tidigare forskning i regel granskat skadliga koder, snarare än de verktyg som nyttjas för att obfuskera dem. Ingen tidigare studie har systematiskt kategoriserat verktyg som obfuskerar skadliga koder. Detta är fokus för studien beskriven i denna rapport – att systematiskt kartlägga verktyg som kan nyttjas för att obfuskera skadliga koder.

3 Metod

En systematisk granskningsmetod inspirerad av [18] tillämpades för att identifiera och kategorisera verktyg som döljer kända bakdörrar från säkerhetsmekanismer. Kitchenham [18] rör systematisk granskning av vetenskaplig forskningslitteratur, men metoderna beskrivna i rapporten kan i de flesta fall även tillämpas på andra områden, såsom granskning av mjukvaruprojekt.

Det fanns två grundförutsättningar för att ett verktyg skulle inkluderas:

1. det behövde ämna till att dölja kända bakdörrar från säkerhetsmekanismer
2. det behövde kunna generera kod som kunde exekveras på ett målsystem

Github[4] användes som databas och Githubs REST API[5] för att nyttja dess sökmotor. Github valdes eftersom det är den största plattformen med öppen kod[6]. Medan Github innefattar ett stort antal öppna mjukvaruprojekt har dess API emellertid stora begränsningar:

- det tillåter söksträngar med operatorerna AND, OR och NOT, men med maximalt fem operatorer per söksträng
- det tillåter ej att söksträngar innehåller reguljära uttryck, *, eller andra liknande funktionella nyckelord.

Dessa begränsningar hanterades genom att tillämpa ett stort antal sökord i kombination med att dela upp söksträngarna på ett lämpligt sätt.

Målet med sökstrategin var att identifiera relevanta verktyg och samtidigt inte inkludera för många irrelevanta verktyg i resultatet. Forskarna bedömde att cirka 1000 verktyg var hanterbart givet studiens omfattning.

Först framställdes en lista med preliminära sökord. Denna lista framställdes genom att forskarna på egen hand sökte efter verktyg på Github med valfria sökord. Utöver preliminära sökord resulterade dessa sökningar i två listor:

1. en lista med verktyg som bör ingå i resultatet
2. en lista med verktyg som inte bör ingå i resultatet.

Dessa listor användes för att testa olika söksträngar mot. Forskarna genomförde sökningar mot Github med målet att träffa verktyg i lista 1 och utesluta verktyg i lista 2, och samtidigt hålla antalet träffar under 1000.

[4] https://github.com/

[5] https://docs.github.com/en/rest/search

[6] https://en.wikipedia.org/wiki/Comparison_of_source-code-hosting_facilities

Detta var en utmaning, och de första sökningarna gav cirka 30 000 resultat. Forskargruppen hade tre möten för att diskutera söksträngar och metoder för att uppnå målet. Den slutliga söksträngen (med en sammanfogad söksträng) finns i fotnot[7]. Utöver söksträngen exkluderades projekt som inte uppdaterats sedan 2018, samt projekt som hade färre än 26 stjärnor (en stjärna betyder att en användare på Github följer projektet).

Sökningen gjordes mot ett projekts titel, beskrivning, samt nyckelord. Detta förfarande resulterade i cirka 1200 verktyg. Samtliga projekts README-filer laddades ner. För att göra en första sållning av irrelevanta träffar gick forskarna igenom några slumpmässiga projekts README-filer för att identifiera irrelevanta träffar samt ord som förekom i README-filerna. Projekt med orden ”xss”, ”phishing”, ”fuzzing”, ”iot” eller ”owasp” i README-filen sorterades bort. Efter detta förfarande återstod cirka 900 verktyg.

Nästa steg involverade att snabbt gå igenom varje projekts Github-sida och granska dess beskrivning, rubriker och README-fil för att sålla bort irrelevanta träffar (givet de två grundförutsättningarna beskrivna i början av kapitlet). Exempel på exkluderade träffar var Github-projekt som enbart listade andra verktyg[8] och verktyg som hade syften som inte passade denna studie[9].

Först bedömde två forskare i projektet 25 slumpmässigt utvalda träffar. Konsensus mellan forskarnas bedömningar mättes med Cohens Kappa koefficient till 0,78. Detta kan ses som god konsensus [19]. Sedan delade forskarna upp de resterande cirka 875 verktygen mellan sig och bedömde varje verktyg som relevant eller inte enligt samma metod. Denna process resulterade i 189 verktyg som bedömdes som relevanta efter en ytlig granskning.

---

[7] (((bypass AND "anti virus") OR (bypass AND anti-virus) OR (bypass AND antivirus) OR (bypass AND av) OR (bypass AND avs) OR (evade AND "anti virus") OR (evade AND anti-virus) OR (evade AND antivirus) OR (evade AND av) OR (evade AND avs) OR (evasion AND "anti virus") OR (evasion AND anti-virus) OR (evasion AND antivirus) OR (evasion AND av) OR (evasion AND avs)) AND ((malware AND encode) OR (malware AND encoding) OR (malware AND encoder) OR (malware AND decode) OR (malware AND decoder) OR (malware AND obfuscate) OR (malware AND encrypt) OR (malware AND undetectable) OR (payload AND encode) OR (payload AND encoding) OR (payload AND encoder) OR (payload AND decode) OR (payload AND decoder) OR (payload AND obfuscate) OR (payload AND encrypt) OR (payload AND undetectable) OR (shell AND encode) OR (shell AND encoding) OR (shell AND encoder) OR (shell AND decode) OR (shell AND decoder) OR (shell AND obfuscate) OR (shell AND encrypt) OR (shell AND undetectable) OR (shellcode AND encode) OR (shellcode AND encoding) OR (shellcode AND encoder) OR (shellcode AND decode) OR (shellcode AND decoder) OR (shellcode AND obfuscate) OR (shellcode AND encrypt) OR (shellcode AND undetectable))) AND NOT (NOT vaccine NOT legal)

[8] Exempelvis https://github.com/cyberguideme/Tools och https://github.com/aress31/sci.

[9] Exempelvis https://github.com/ab77/black.box, som tillhandahåller routing via VPN.

Nästa steg var att skapa en kategoriseringsmodell för att kunna besvara forskningsfråga två. Denna modell skapades med grund i publicerad vetenskaplig forskning såsom [3]–[9] och beskrivs i kapitel 4.

De 189 kvarstående verktygen kategoriserades. Denna kategorisering gjordes genom att undersöka projektens Github-sidor samt granska projektens kod. Två av forskarna gick först igenom fem verktyg var för sig, kategoriserade dem och jämförde sedan resultatet. Vid enstaka fall skiljde sig bedömningen åt och då diskuterade forskarna varför. Efter denna kalibrering bedömde forskarna sin konsensus som god. Verktygen som var kvar delades upp mellan de två forskarna och kategoriserades enligt metoden i kapitel 4.

Av 189 ursprungligen utvalda verktyg bedömdes efter källkodsgranskningen 174 som relevanta. De verktyg som inte bedömdes som relevanta var i de flesta fall inte verktyg, utan dokumentation av olika slag. I ett fåtal fall handlade det om verktyg som hade ett användningsområde som inte rörde forskningsfrågan. Ett sådant exempel är OffensiveRust[10], vilket innefattar ett antal allmänna exempelverktyg skrivna i programspråket Rust. Statistik för de 174 relevanta verktygen analyseras i kapitel 5.

[10] https://github.com/trickster0/OffensiveRust

4 Kategoriseringsmodell

Kategoriseringsmodellen består av fyra delar. *Projektinformation* (avsnitt 4.1) innefattar övergripande information om varje verktyg samt huruvida forskarna bedömde de som användbara och möjliga att tekniskt evaluera. *Arkitekturer och filformat* (avsnitt 4.2) behandlar de typer av arkitekturer, filformat och filprotokoll (den binära datastrukturen för en sparad fil) som de olika verktygen kan hantera. *Försvar mot statisk granskning* (avsnitt 4.3) innefattar de metoder som verktygen tillämpar för att dölja bakdörrar från säkerhetsmekanismer innan de exekveras. *Försvar mot dynamisk granskning* (avsnitt 4.4) innefattar de metoder som verktygen tillämpar för att dölja bakdörrar från säkerhetsmekanismer när bakdörrarna exekveras.

4.1 Projektinformation

Kategorin projektinformation innefattar nio underkategorier:

1. huvudsakligt programmeringsspråk
2. när verktyget skapades
3. när verktyget senast uppdaterades
4. hur många ouppklarade fel som finns rapporterade för verktyget
5. hur många som skapat förgreningar av verktyget (Git forks)
6. hur många användare som följer verktyget
7. hur stort verktyget är (i kilobytes)
8. huruvida FOI-forskarna bedömer att verktyget enkelt kan testas rent tekniskt
9. ett av FOI-forskarna angivet betyg på skalan 1-10, där 1: oanvändbart och 10: fantastiskt.

Underkategori 1-7 samlades in via skript mot Githubs API. Programmeringsspråk (#1) identifieras automatiskt via verktyget Linguist[11]. Om ett projekt består av flera typer av språk anges det dominanta. Storleken för ett projekt (#7) är den packade storleken för projektet på Github-servern exklusive alla tredjepartsberoenden.

Dessa sju kategorier beskriver inte vilka metoder som verktyg tillämpar, men är ändock värdefulla att inkludera. Exempelvis är verktyg som stadigt uppdateras, och har många förgreningar och följare, sannolikt bättre än verktyg som sällan uppdateras och har få förgreningar och följare. Antal ouppklarade fel kan ses som beskrivande statistik för hur robust ett verktyg är, samt enkelt det är att tillämpa och underhålla det. Programmeringsspråk indikerar hur ett verktyg är

[11] https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-repository-languages

tänkt att vidareutvecklas. Det bör också beaktas att många populära bakdörrar, såsom Meterpreter[12] och Cobalt Strike:s Beacon[13], är skrivna i programmeringsspråket C. Vissa kodningsmetoder är betydligt mer komplicerade att tillämpa för språk längre från hårdvaran. Exempelvis är det i grundutförande inte möjligt för en utvecklare att tillämpa valfria nativa Windows-API-funktioner om applikationen är skriven i C#. Utvecklaren är istället begränsad till att tillämpa de funktioner som det byggts in stöd för i den tillämpade C#-varianten. Dessutom ökar storleken för kod i takt med antalet beroenden som måste skeppas med den för att den skall fungera.

Kategori 8-9 bedömdes av projektgruppens forskare. För att ett verktyg skulle få ett ”Ja” i underkategori 8 krävdes det att dess installation samt skapande av testskript för systematisk teknisk utvärdering max skulle bedömas kräva ett fåtal arbetstimmar. I grund handlade det om att verktyget inte skulle vara orimligt svårt att integrera i verktyget SVED [20]. SVED är ett verktyg som utvecklats av FOI för automatiserad exekvering av olika förutbestämda tekniska aktiviteter, såsom nätverkskartläggning och mjukvaruangrepp, under cybersäkerhetsövningar. SVED används av Lore för den faktiska exekveringen av utvalda tekniska aktiviteter.

4.2 Arkitekturer och filformat

Att stödja fler arkitekturer och filformat gör ett verktyg mer versatilt då det kan tillämpas för fler omständigheter. Fyra kategorier återfinnes gällande arkitekturer och filformat:

1. vilka filformat som kan produceras av ett verktyg:
    a. binärkod med huvud, exempelvis EXE, DLL eller SO
    b. positions-oberoende binärkod (PIC)
    c. skript, exempelvis Powershell eller Python
    d. data, exempelvis PNG, SVG eller MP4.
2. vilka typer av arkitekturer som verktyget stödjer:
    a. x86-32 (32-bit)
    b. x86-64 (64-bit)
    c. annan.
3. vilka typer av operativsystem som filer skapade av verktyget har stöd för:
    a. Windows
    b. Linux
    c. MacOS

[12] https://github.com/rapid7/metasploit-payloads/tree/master/c/meterpreter

[13] https://hstechdocs.helpsystems.com/manuals/cobaltstrike/current/userguide/content/topics/post-exploitation_main.htm.

d. annat.

4. huruvida verktyget kan obfuskera valfria användarangivna filer, eller enbart kan obfuskera ett antal särskilt utvalda filer

4.2.1 Filformat

Filformat är av vikt av flera anledningar. Dels ställer en cyberoperation ibland krav på ett visst filformat, såsom positions-oberoende binärkod (eng: position independent code, PIC). PIC betyder att en kod, till skillnad från en binärkod med huvud (#1a), inte inkluderar några hårdkodade adresser, såsom specifika adresser för kodfunktioner i delade bibliotek, och därmed kan exekveras i minnet på en annan process (#1b). I det fall att det behövs delade biblioteksfunktioner, såsom Windows API-funktioner, konstrueras import- och export-tabeller dynamiskt som ett steg under exekveringen av koden.

Valet av filformat kan också påverka huruvida en säkerhetsmekanism upptäcker den skadliga koden (#1d). Exempelvis är ett vanligt sätt att dölja skadlig kod att koda den till ett filformat som normalt enbart innefattar data, såsom en CSV-fil. [21]

4.2.2 Operativsystem och arkitekturer

Likt filformat kan en cyberoperation involvera målmaskiner som har olika operativsystem och processorarkitekturer. Kategori #2 fokuserar på de två vanligaste arkitekturerna för skrivbordsdatorer: x86-32 och x86-64. Kategori #3 fokuserar på de vanligaste operativsystemen för skrivbordsdatorer: Windows, Linux och Mac OS. Fokus ligger på skrivbordsdatorer eftersom detta är den typ av system Lore oftast tillämpas mot under cybersäkerhetsövningar.

4.2.3 Stöd för valfria filer

Kategori #4 involverar att vissa verktyg enbart tillåter obfuskering av ett fåtal särskilt utvalda koder. För ett ”Ja” i denna kategori krävs det att verktyget kan obfuskera en valfri kod. Det är godtagbart om koden behöver formateras enligt en viss specifikation, såsom PIC eller Powershell.

4.3 Försvar mot statisk granskning

Statisk granskning involverar att gå igenom programkod utan att koden körs, och är ett vanligt första steg för säkerhetsmekanismer såsom antivirusprogram för att bedöma om koden bör tillåtas exekvera [3].

Kategoriseringsmodellen innefattar fyra kategorier som rör olika sätt att obfuskera kod för att undgå upptäckt genom statisk granskning:

1. kodförändringmetod:

   a. polymorfism eller metamorfism
   b. kodning utan entropi.
2. kodförändringsgrund:
   a. källkod
   b. kompilerad kod.
3. kodexekveringsmetod:
   a. exekvering via kodens standardmetod
   b. exekvering via verktygsspecifik applikation
   c. exekvering via bootstrap-funktion i legitim applikation.
4. kodning till icke-exekverbart format.

4.3.1 Kodförändringsmetod

En metod att undgå upptäckt är att genomföra kodförändringar som förändrar kodens/filens signatur [6], [7], [22]. Ett sätt att genomföra kodförändringar involverar tillämpning av tekniker som medför att varje kod som genereras av verktyget får en ny unik signatur [6]. De två vanligaste teknikerna inom denna kategori är polymorfism och metamorfism (#1a). Polymorfism realiseras via kryptering såsom XOR eller AES [4], [23]. Metamorfism realiseras genom slumpmässiga kodändringar som inte påverkar kodens tänkta syfte vid exekvering, exempelvis att instruktioner som inte tillför mer än obetydliga beräkningar läggs till i den skadliga koden [4], [6], [23]. Till skillnad från polyforfism kräver metamorfism ingen dekrypteringsprocess.

En annan metod är att tillämpa kodning/packning utan entropi; att varje genererad kod/fil har samma signatur så länge indatat är samma [5], [24] (#1b). Kodförändringar utan entropi kan exempelvis tillämpas om kodningen i sig antas räcka för att kringgå säkerhetsmekanismer. Exempelvis kan det handla om att konvertera tillämpade Windows API-anrop (som antivirus ofta granskar) till direkta systemanrop[14] (som ej granskas). Vissa menar även att polymorfism i sig själv kan medföra upptäckt då entropin genererad av vanliga kryptografiska funktioner såsom XOR och AES är enkla att identifiera, och godartade exekverbara filer sällan inkluderar stora mängder krypterad programkod [25].

4.3.2 Kodförändringsgrund

Kodförändringar kan vidare antingen realiseras genom förändring av kompilerade binärkod eller genom förändring av källkod [6], [11] (#2).

[14] https://github.com/klezVirus/SysWhispers3

4.3.3 Kodexekveringsmetod

En obfuskerad bakdörr kan, beroende på kodförändringsmetod, vara möjlig att exekvera utan någon hjälpapplikation (#3a). Exempelvis kan det handla om en binärkod med huvud som har kodats genom att byta ut NOP-instruktioner mot registeroperationer som inte förändrat applikationens funktion. Kodexekvering kan också utföras via någon form av en specifik applikation skapad för ändamålet (#3b), eller genom boostrap-funktioner i en legitim applikation (t.ex. calc.exe) (#3c). [6], [11]

4.3.4 Icke-exekverbart format

Kodning till ett icke-exekverbart format innefattar att konvertera koden till ett filprotokoll som säkerhetsmekanismer inte förväntas studera lika väl, exempelvis en bildfil eller en textfil (#4). Sedan nyttjas en särskild uppacknings- och exekveringsrutin för att exekvera koden som ligger lagrad i denna fil [21]. Exempelvis placerade den skadliga koden Skywiper programkod i .OCX-filer [26].

4.4 Försvar mot dynamisk granskning

Dynamisk granskning innefattar att undersöka programkod under dess exekvering [3]. Flera av försvaren mot statisk granskning är mindre effektiva mot dynamisk granskning då säkerhetsmekanismen kan studera det faktiska programflödet.

Exempelvis kanske det finns särskilda tester som antivirusprogrammet tillämpar efter att dekrypteringsrutiner eller särskilda sekvenser av Windows API-anrop har utförts [16]. Ett exempel på en särskild sekvens av Windows-API-anrop som det skulle kunna skapas en signatur för är *GetProcessWindowStation*() → *GetUserObjectInformation*() → *GetCurrentThreadId*() → *GetThreadDesktop*() → *GetUserObjectInformation*(). Denna API-sekvens körs alltid vid start av C-versionen av bakdörren Meterpreter[15].

Det finns diverse metoder för att undgå upptäckt av säkerhetsmekanismer som utför dynamisk granskning:

1. kodinjektion:
    a. injicera kod i ny trovärdig process
    b. injicera kod i körande legitim process.
2. nedladdning av kod från extern plats
3. tester för om koden körs i en sandlåda, såsom försenad exekvering eller försök att kalla på funktioner i grafiska gränssnitt

[15] https://GitHub.com/rapid7/metasploit-payloads/blob/master/c/meterpreter/source/metsrv/server_setup.c

4. hindra övervakningsfunktioner från att kunna observera processen
5. stänga av säkerhetsfunktioner.

4.4.1 Kodinjektion

En vanlig metod för att undgå upptäckt är att flytta den skadliga koden till en annan plats som inte övervakas [23]. Det finns flera sätt att injicera kod från en process till en annan. Ett vanligt sätt är att skapa en ny process, se till att denna process verkar trovärdig, och sedan injicera koden till den (#1a). Den nya processen kan exempelvis fås att se ut som om den exekverats av en användare genom utforskaren i Windows (explorer.exe). Ett annat sätt är att injicera koden till en redan körande process som säkerhetsmekanismer förväntas lita på (#1b). Dessa metoder har både för- och nackdelar. En fördel med att skapa en ny process är att det finns komplett kunskap kring dess exekvering. En nackdel är att dess skapande kan vara en anomali för systemet. Det kan även finnas brandväggsregler för hur olika applikationer får kommunicera, vilket kan skapa problem för kommunikation till servern som tillhandhåller fjärrstyrningssessionen (eng: Command and Control, C2). Exempelvis kanske kommunikationen till C2-servern sker via HTTP, vilket enbart en särskild webbläsare på maskinen har tillstånd för. En nackdel med att injicera i en existerande process är att det kan kräva behörigheter som inte finns. Det kan ibland också vara svårt att hitta en lämplig plats för den injicerande koden. Dessutom kan det finnas övervakningsfunktioner som granskar de API:er som kodinjicering involverar. [23]

4.4.2 Nedladdning av kod från fjärran plats

En metod för att undgå övervakningsfunktioner är att först exekvera en liten kod med mycket begränsad funktionalitet som bedöms ha mindre chans att upptäckas av säkerhetsmekanismer (#2). Denna kod ansvarar för att hämta hem och exekvera den egentliga skadliga koden från en extern plats såsom en webbserver. [6], [23]

4.4.3 Sandlådetester

Av säkerhetsskäl genomför säkerhetsmekanismer ofta dynamiska granskningar i särskilda isolerade ”sandlådor” där exekvering av en skadlig kod antas kunna utföras utan risk för systemet (#3). Av kostnadsskäl täcker sandlådor dock inte in alla tänkbara externa funktioner som olika program kan tänkas behöva. Detta utnyttjas av skadliga koder, t.ex. genom att ställa frågor mot externa funktioner som inte förväntas vara inkluderade på ett korrekt sätt i sandlådan. Om fel svar ges erhålls avbryts kodexekveringen [10], [14], [15], [23]. Av prestandaskäl granskar de flesta säkerhetsmekanismer en process noggrant i sandlådan samt vid särskilda observerade beteendemönster, men relativt sällan emellan. Detta

mönster utnyttjas av skadliga koder på diverse sätt, i synnerhet genom att försena exekveringen. [10], [15], [23]

4.4.4 Hindra övervakningsfunktioner

När en process startas registrerar olika säkerhetsmekanismer typiskt sig som lyssnare på diverse av processens nyttjade API-funktioner som bedöms vara värdefulla för upptäckten av skadlig kod [9]. Utöver att lyssna på vanliga API-funktioner såsom *WriteFile*() erbjuder Windows även Antimalware Scan Interface (AMSI)[16], som är ämnat för detektion av skadlig kod. Genom AMSI kan säkerhetsmekanismer enkelt utföra olika tester av andra processer.

Två vanliga metoder som skadliga koder nyttjar för att lösa detta problem är att avregistrera sådana lyssnare från processen samt att motverka injektion av andra processer i sig (#4).

En tredje möjlighet är att helt undvika Windows API-anrop och istället nyttja direktandrop mot olika valda systemfunktioner[17]. Exempelvis, normalt tillämpas Windows API-funktionen *WriteFile*() i kernelbase.dll för att skriva till en fil. *WriteFile*() kallar vidare på Windows native-funktionen *NtWriteFile*() i ntdll.dll. Båda dessa funktioner lever i applikationens egna virtuella addressutrymme[18] (eng: user mode). *NtWriteFile* kallar vidare på funktioner i operativsystemets delade virtuella adressutrymme (eng: kernel mode) för att genomföra den faktiska filsparningen. Säkerhetsfunktioner övervakar typiskt user mode-funktioner, men ej kernel mode-funktioner. En hotaktör kan utnyttja detta genom att kalla på kernel mode-implementationen för en funktion, såsom den för *WriteFile*(). Nackdelen med metoden är att olika Windows-versioner har olika implementationer för kernel mode-funktioner, vilket gör att metoden inte är helt reliabel. Ett direkt kernel-anrop är också en tydlig anomali för hur program normalt beter sig.

4.4.5 Stänga av övervakningsfunktioner

Slutligen kan vissa skadliga koder ta till mer invasiva metoder, såsom att ta bort kritiska filer för antivirusprogram som gör att de ej fungerar (#5) [6]. Det finns flera anledningar till varför detta sällan görs. I synnerhet är det i regel svårt för en hotaktör att veta vilka övervakningsmekanismer som finns för ett system. Detta är problematiskt då avstängning av en säkerhetsmekanism är en stor anomali för ett system, och därmed enkelt upptäcks om det inte görs med kirurgisk precision.

---

[16] https://learn.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-portal

[17] https://outflank.nl/blog/2019/06/19/red-team-tactics-combining-direct-system-calls-and-srdi-to-bypass-av-edr/

[18] https://learn.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode

Det är inte heller säkert att hotaktören har de behörigheter som krävs för ändamålet.

5 Resultat

Statistik för de 174 relevanta verktygen analyseras i detta kapitel. Läsaren bör beakta att det saknas en möjlighet att utvärdera hur effektiva olika tekniker är för målet att dölja en bakdörr från säkerhetsmekanismer, då effektiviteten för de studerade verktygen och deras tillämpade tekniker ej har analyserats rent praktiskt. Ett teoretiskt alternativ till att utföra praktiska tester vore att koppla verktyg och tekniker till funktionaliteten för olika säkerhetsmekanismer, så som antivirus-signaturer. Detta är dock generellt inte en bra metod då sådan funktionalitet i bästa fall är svår att härleda till särskilda tekniker, och i värsta fall är proprietär och okänd.

Av denna anledning är merparten av analysarbetet utförd på den deskriptiva statistik som erhållits genom kartläggningsarbetet. Det åligger framtida arbete att genomföra praktiska tester av verktyg för att på så sätt kunna utvärdera effekten för olika verktyg och tekniker.

Kapitel 5.1-5.4 beskriver resultatet för de fyra studerade huvudkategorierna. Kapitel 5.5 presenterar verktygen som gavs högst betyg av de granskande forskarna.

5.1 Projektinformation

En översikt av programmeringsspråken för de studerade verktygen (#1) beskrivs av Tabell 1. I Tabell 1 är alla kolumner utom ”Antal” medelvärden.
Som kan ses var flest verktyg huvudsakligen skrivna i språket Python. Tre verktyg kunde ej tilldelas ett språk av Github. Ett av dessa var en kollektion skript skrivna direkt i README-filen; de andra två innehöll enbart kompilerad kod. Rangordningen gällande programmeringsspråk skiljer sig en del från rangordningen för projekt på Github i allmänhet[19]. Exempelvis är JavaScript det vanligaste språket på Github, men används inte alls av verktygen. Anledningen är att verktygen generellt består av två delar – en del som obfuskerar kod, och en annan del som exekverar den obfuskerade koden på ett målsystem.

Delen som exekverar kod är med fördel skriven i ett språk som ställer så få krav på målsystemet som möjligt. Ett exempel på ett sådant krav är gällande särskild tredjepartskod på målsystemet, såsom Java runtime (JRE) för applikationer skrivna i Java och Visual C++ runtime (MSCV)[20] för många applikationer skrivna i C++. Dessutom är det generellt fördelaktigt att skriva kod i ett språk som har ”kortare avstånd” till hårdvaran eftersom det då finns bättre möjligheter

[19] https://octoverse.github.com/2022/top-programming-languages

[20] https://learn.microsoft.com/en-US/cpp/windows/latest-supported-vc-redist

att skräddarsy koden för att kringgå olika lyssnare som säkerhetsmekanismer kan tänkas tillämpa för olika API:er (se kapitel 4.4.4).

Applikationen som obfuskerar kod körs på hotaktörens system och ställer därför egentligen inga direkta krav gällande språk. Dock är forskarnas erfarenhet att vissa språk, i synnerhet då Python, har betydligt bättre tredjepartsstöd för IT-säkerhetsrelevanta tillämpningar. Python ses dessutom i allmänhet som ett nybörjarvänligt språk som det är enkelt att skriva kod i [27].

Tabell 1. Översikt av programmeringsspråk (kategori #1).

Språk (#1)	Antal	Betyg (1-10)	Storlek (kilobytes)	Följare	Förgreningar	Ouppklarade fel
Python	51	3,8	8274	543	134	5
C#	31	3,8	5653	289	69	2
C++	24	4,3	17937	287	70	1
C	20	4,1	7101	341	80	3
Go	18	4,0	8678	332	71	2
PowerShell	12	4,3	856	266	68	2
Assembly	4	5,3	6202	821	142	4
Nim	4	2,8	213	199	33	1
(Inget)	3	4,3	12998	168	45	10
Shell	2	5,5	1077	860	218	3
VBScript	2	3,5	481	463	100	2
Pascal	1	5,0	2870	857	158	0
Ruby	1	5,0	456	121	24	0
Visual Basic	1	4,0	110	757	203	4

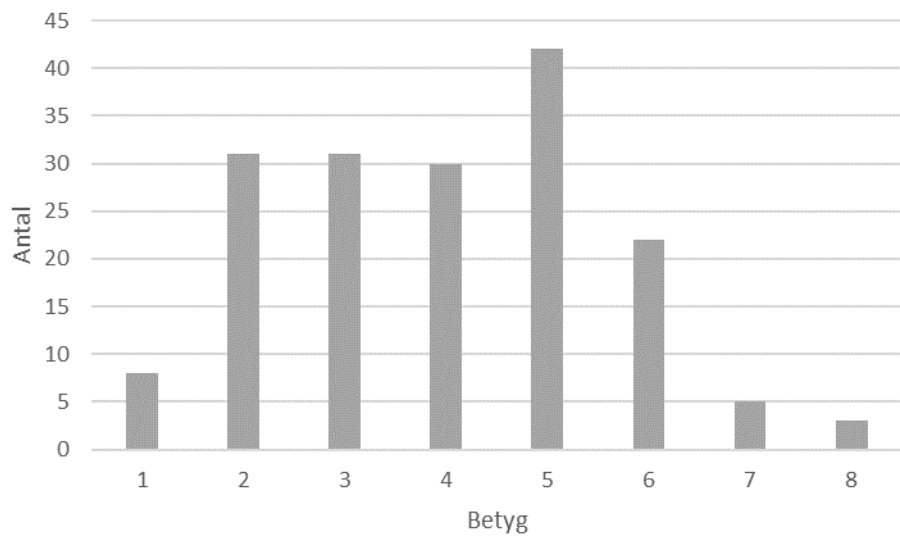
En översikt av de resterande kategorierna ges av Tabell 2. Medelvärden och standardavvikelse presenteras för alla kategorier utom #8, vars utfall var binärt: 114 verktyg bedömdes kunna testas enkelt och 58 med en större ansträngning. Det är anmärkningsvärt att medeltiden för den senaste uppdateringen av verktygen var mer än ett år innan sökningen gjordes mot Github. Ett år ger utvecklare av säkerhetsmekanismer mycket goda möjligheter att anpassa sina system efter verktygens uppdateringar.

Tabell 2. Översikt av kategori #2-#9.

#	Kategori	Medel	Standardavvikelse
2	Skapelse	2019-10-18	689 dagar
3	Senaste uppdateringen	2021-02-24	465 dagar
4	Ouppklarade fel	2,9	6,4
5	Förgreningar	93,4	174,9
6	Följare	394,3	702,1
7	Storlek (kilobytes)	8031,7	23932,1
9	Betyg	4,0	1,6

Betygen som gruppens forskare gav de olika verktygen beskrivs i Figur 1. Av 174 verktyg fick 30 betyget 6 eller högre. Sju av dessa fick ett betyg högre än 6. Dessa verktyg beskrivs i Kapitel 5.5.

Valet av Python i samband med undermåliga uppdateringar och låga betyg föranleder ett antagande om att många av verktygen skapades i utbildningssyfte snarare än för att vara dugliga.



Figur 1. Fördelning av betyg för studerade verktyg.

5.2 Arkitekturer och filformat

En översikt av de typer av arkitekturer och filformat som kan hanteras av de olika verktygen presenteras i Tabell 3.

Tabell 3. Filformat som stödjs

#	Kategori	Stöds	Stöds ej
1a	Binärkod med huvud	113	60
1b	PIC	18	155
1c	Skript	34	138
1d	Data	27	146
2a	x86-32	154	20
2b	x86-64	161	13
2c	Annan arkitektur	6	168
3a	Windows	170	4
3b	Linux	30	144
3c	MacOS	22	152
3d	Annat operativ	11	163
4	Valfria filer	131	43

Som kan ses är binärkod med huvud (t.ex. PE/EXE) klart vanligast, medan positionsoberoende binärkod (PIC) är ovanligast. 16 verktyg stödjer inget av formaten. Dessa verktyg är kodbibliotek och stödverktyg av olika slag som erbjuder relevant funktionalitet, såsom minnesinjektion eller kryptering. Exempelvis kan verktyget SysWhisperer[21] generera assemblerkod för de kernel-anrop som i slutändan tillämpas av olika vanliga Windows API-funktioner, vilka kan nyttjas för att undgå de lyssnare som säkerhetsmekanismer kan ha registrerat för API-funktionerna. Tre verktyg kan generera filer för både binärkod med huvud, PIC, skript och data. Det första är en kollektion av verktyg skapade inom ramen för en kurs; det andra är donut (se kapitel 5.5); det tredje är ett ramverk som innefattar flera andra verktyg – i synnerhet då donut.

Att stödet för x86-64 är något större är inte särskilt förvånande då många operativsystem och applikationer är i färd med att avsluta stödet för x86-32.

De allra flesta verktyg stödjer Windows. Betydligt färre stödjer Linux eller Mac. Orsaken är sannolikt att antivirus sällan nyttjas på dessa plattformar. Vidare tillåter 43 av verktygen enbart obfuskering av ett fåtal utvalda skadliga koder

[21] https://GitHub.com/jthuraisamy/SysWhispers2

(#4). En analys av #4 visade för övrigt att PIC var det vanligaste kravet på formatering av den skadliga koden som skall obfuskeras.

5.3 Försvar mot statisk granskning

En översikt av resultatet gällande försvar mot statisk granskning ges av Tabell 4.

Tabell 4. Resultat gällande försvar mot statisk granskning.

#	Kategori	Stöds	Stöds ej
1a	Polymorfism/metamorfism	98	76
1b	Kodning utan entropi	20	154
2a	Källkodsförändringar	3	171
2b	Förändringar av kompilerad kod	117	57
3a	Exekvering via kodens standardmetod	7	164
3b	Exekvering via verktygs-specifik applikation	150	21
3c	Exekvering via bootstrap-funktion i legitim applikation	3	168
4	Kodning till icke-exekverbart format	24	150

Som kan ses var stödet större för polymorfism/metamorfism (#1a) än för kodning utan entropi (#1b). Av dessa två tekniker tillämpade de allra flesta verktyg polymorfism genom XOR. Det kan också noteras att 56 verktyg inte ändrade på koden över huvud taget. I detta fall gjorde verktyget i regel enbart en enklare inladdning av kod. Exempelvis består verktyget DotNetAVBypass[22] av ett fåtal rader C#-kod som laddar hem och exekverar en användarangiven PIC.

Enbart tre verktyg hade stöd för källkodsförändringar (#2a). Två av dessa verktyg rörde bakdörrar skapade i skriptspråk (Powershell och Python), och det tredje C-kod. Den huvudsakliga anledningen till detta är troligen att flera vanliga bakdörrar, såsom Cobalt Strike Beacon, enbart finns tillgängliga i binär form.

Cirka 70% av alla verktyg hade stöd för att förändra kompilerad kod (#2b). Typiskt involverade ändringen polymorfism och exekvering via någon typ av hjälpapplikation (#3b). Ett exempel är foolavc[23], som krypterar en användargiven .EXE-fil med XOR-kodning. Den resulterande filen laddas in och exekveras via en hjälpapplikation (av typen .EXE) skapad av författarna för ändamålet.

Enbart sju verktyg genomförde obfuskeringar som medförde att applikationen kunde exekveras utan någon hjälpkod (#3a). Nästan 90% av alla verktyg kunde

[22] https://GitHub.com/lockfale/DotNetAVBypass-Master

[23] https://GitHub.com/hvqzao/foolavc

exekvera kod via en verktygs-specifik hjälpapplikation (#3b). Enbart tre verktyg kunde exekvera kod genom en användargiven applikation på disken (#3c).

Cirka 14% av alla verktyg hade stöd för att koda till ett icke-exekverbart format (#4). Ett sådant exempel är Bluffy[24], som kodar till filformaten UUID, CLSID, SVG, CSS och CSV. De faktiska filprotokollen nyttjas på ett sådant sätt att den genererade filen kan laddas in i vanliga programvaror som är byggda för att tolka dem (t.ex. Excel för CSV). En verktygspecifik hjälpapplikation (av typen .EXE) nyttjas sedan för att packa upp och exekvera koden.

5.4 Försvar mot dynamisk granskning

En översikt av resultatet gällande försvar mot dynamisk granskning ges av Tabell 5.

Tabell 5. Resultat gällande försvar mot dynamisk granskning.

#	Kategori	Stöds	Stöds ej
1a	Injicera kod i ny trovärdig process	35	139
1b	Injicera kod i körande legitim process	19	155
2	Nedladdning av kod från extern plats	32	140
3	Sandlådetester	30	144
4	Hindra övervakningsfunktioner	22	152
5	Stänga av säkerhetsfunktioner	2	172

Som kan ses har få verktyg valt att tillämpa kodinjektion över huvud taget (#1). De allra flesta verktyg (74%) exekverar koden i huvudprocessen, om än ibland i en separat tråd. Ett exempel på ett verktyg som både har stöd för att injicera kod i en ny process, och för att injicera kod i en körande process, är Phantom-Evasion[25]. Phantom-Evasion stödjer flera kodinjektionsmetoder (MITRE ATT&CK teknik T1055[26]), såsom thread execution hijacking (T1055.003[27]),

[24] https://GitHub.com/preemptdev/bluffy

[25] https://GitHub.com/oddcod3/Phantom-Evasion

[26] https://attack.mitre.org/techniques/T1055/

[27] Injicera kod i en körande process på ett sätt som byter ut dess egentliga exekverade kod https://attack.mitre.org/techniques/T1055/003/

Asynchronous Process Call (APC) spraying (T1055.004[28]) och process hollowing (T1055.012[29]).

Cirka 19% av alla verktyg stödjer någon form av nedladdning av användargiven kod från fjärran plats (#2). I de flesta fall handlar det om någon typ av TCP-baserad överföring över IPv4, i synnerhet via HTTP. Ett exempel är verktyget JPGtoMalware[30], som tillämpar två steg: först genomförs kodning till en bildfil (.JPG). Sedan körs ett Python-skript på målmaskinen som hämtar bildfilen via HTTP, avkodar den, och exekverar den.

Cirka 17% av alla verktyg stödjer någon typ av sandlådetester (#3). Exempelvis har verktyget uru[31] stöd för att avbryta kodexekveringen om maskinen inte är med i en domän, samt för att sova en viss tid innan dess huvudfunktionalitet börjar.

Cirka 13% av alla verktyg inkluderar funktionalitet som kan hindra övervakningsfunktioner (#4). Det handlar uteslutande om hantering av Windows API-hooks. Dessa tas i regel hand om genom tre övergripande metoder:

- genom att motverka injicering av extern kod (t.ex. Alaris[32] som blockerar injicering av osignerade DLL:er)
- genom att ta bort inladdade callback-funktioner för övervakningssystem (t.ex. manipulerar Ivy[33] minnet i den egna processen genom Windows API-funktionerna *WriteProcessMemory*(), *ZWQueryVirtualMemory*() och *NtProtectVirtualMemory*() för Ntdll.dll, Kernel32.dll, Kernelbase.dll, Advapi32.dll, Sechost.dll, Ws2_32.dll och Winmmbase.dll)
- genom att kalla på funktionalitet genom direkta kernel-anrop som undgår callback-funktioner istället för de tänkta API-ändpunkterna (t.ex. SysWhisperer[34])

Enbart två verktyg tar bort säkerhetsfunktioner (#5). Båda fallen handlar om verktyg som stänger av Microsoft Windows Defender (ett skrivet i C och ett i Powershell). Som beskrivits tidigare är det i regel inte ett bra alternativ att stänga

[28] Injicera kod i en körande process asynkront via Windows APC-kö, https://attack.mitre.org/techniques/T1055/004/

[29] Starta en ny process i pausat tillstånd, och sedan byta ut dess kod mot användargiven kod https://attack.mitre.org/techniques/T1055/012/

[30] https://GitHub.com/abdulkadir-gungor/JPGtoMalware/

[31] https://GitHub.com/guervild/uru

[32] https://GitHub.com/cribdragg3r/Alaris

[33] https://GitHub.com/optiv/Ivy

[34] https://GitHub.com/jthuraisamy/SysWhispers2

av säkerhetsfunktioner då det ger ett stort avtryck på målet och kräver höga privilegier.

5.5 Verktyg med höga betyg

Av de 174 verktygen fick 30 betyget 6 eller högre. Sju av dessa fick ett betyg högre än sex:

- **Ivy**[35] (betyget 8)
- **Phantom-Evasion**[36] (betyget 8)
- **avet**[37] (betyget 8)
- **donut**[38] (betyget 7)
- **Shhhloader**[39] (betyget 7)
- **Sgn**[40] (betyget 7)
- **Veil**[41] (betyget 7)
- **Wraith**[42] (betyget 7).

Egenskaperna för dessa verktyg beskrivs i Tabell 6 - Tabell 9.

Som kan ses i Tabell 6 finns det inga tydliga samband mellan de olika kategorierna gällande projektinformation. Phantom-Evasion, avet och Wraith hade inte uppdaterats på över ett år när sökningen genomfördes. Det är dock oklart hur detta påverkar verktygens effektivitet, mer än att utvecklare av säkerhetsmekanismer inte behöver uppdatera sina system mot dem lika ofta.

[35] https://GitHub.com/optiv/Ivy

[36] https://GitHub.com/oddcod3/Phantom-Evasion

[37] https://GitHub.com/govolution/avet

[38] https://GitHub.com/TheWover/donut

[39] https://GitHub.com/icyguider/Shhhloader

[40] https://GitHub.com/EgeBalci/sgn

[41] https://GitHub.com/Veil-Framework/Veil

[42] https://GitHub.com/slaeryan/AQUARMOURY/tree/master/Wraith

Tabell 6. Projektinformation för de verktyg som bedömdes som bäst.

#	Kategori	Ivy	Phantom	avet	donut	Shhhloader	Sgn	Veil	Wraith
1	Programmeringsspråk	Go	Python	C	C	Python	Go	Python	C++
2	Skapelse	2021-11-18	2017-10-31	2017-01-28	2019-03-27	2021-09-28	2019-10-30	2017-03-02	2020-10-22
3	Senaste uppdateringen	2022-01-31	2021-01-26	2020-11-24	2022-03-22	2022-04-19	2022-06-13	2022-05-20	2020-10-24
4	Ouppklarade fel	3	39	1	27	2	3	52	1
5	Förgreningar	104	349	339	433	118	114	807	109
6	Följare	581	1226	1424	2081	603	708	3206	507
7	Storlek (kilobytes)	419	367	671	9770	1174	851	712	10712
8	Kan testas	J	J	J	J	J	J	J	J
9	Betyg	8	8	8	7	7	7	7	7

Som kan ses i Tabell 7 exekverar alla verktyg utom Ivy skadlig kod via en särskilt utvecklad binär. Ivy placerar istället skadlig binärkod i makron i Microsoft Office-dokument (t.ex. .docx eller .xlsx).

Tabell 7. Arkitekturer och filformat för de verktyg som bedömdes som bäst.

#	Kategori	Ivy	Phan	avet	donut	Shhh	Sgn	Veil	Wrai
1a	Binärkod med huvud	N	J	J	J	J	N	J	J
1b	PIC	N	N	N	J	N	J	J	N
1c	Skript	N	N	N	J	N	N	N	N
1d	Data	J	N	N	J	N	N	N	N
2a	x86-32	J	J	J	J	N	J	J	J
2b	x86-64	J	J	J	J	J	J	J	N
2c	Annan arkitektur	N	N	N	N	N	N	N	N
3a	Windows	J	J	J	J	J	J	J	J
3b	Linux	N	J	N	N	N	J	J	N
3c	MacOS	N	N	N	N	N	J	J	N
3d	Annat operativ	N	N	N	N	N	N	N	N
4	Valfria filer	J	J	J	J	J	J	J	N

Som kan ses i Tabell 8 kan avet och donut utöver polymorfism/metamorfism också tillhandahålla kodning utan att filens signatur ändras mellan kompileringar. Alla verktygen arbetar med indata i form av kompilerad kod. Dokumenten som genereras av Ivy är tänkta att exekveras av vanliga Windows-applikationer för ändamålet, såsom Excel, men verktyget har också förmågan att generera ren VBA-kod som kan exekveras via en särskild hjälpapplikation.

Tabell 8. Försvar mot statisk granskning för de verktyg som bedömdes som bäst.

#	Kategori	Ivy	Phan	avet	donut	Shhh	Sgn	Veil	Wrai
1a	Polymorfism/metamorfism	J	J	J	J	J	J	J	J
1b	Kodning utan entropi	N	N	J	J	N	N	N	N
2a	Källkodsförändringar	N	N	N	N	N	N	N	N
2b	Förändringar av kompilerad kod	J	J	J	J	J	J	J	J
3a	Exekvering via kodens standardmetod	N	N	N	N	N	N	N	N
3b	Exekvering via verktygs-specifik applikation	J	J	J	J	J	J	J	J
3c	Exekvering via bootstrap-funktion i legitim applikation	N	N	N	N	N	N	N	N
4	Kodning till icke-exekverbart format	J	N	N	J	N	N	N	N

Som kan ses i Tabell 9 finns det inget verktyg som är kapabelt att utföra alla studerade metoder för att undgå dynamisk granskning. Närmast är Phantom-Evasion, vilken är kapabel till allt utom att stänga av säkerhetsfunktioner. Detta betyder dock inte att Phantom-Evasion nödvändigtvis är bättre än de andra verktygen på att undvika upptäckt via dynamisk granskning – för att studera denna frågeställning krävs det mer praktiska tester.

Tabell 9. Försvar mot dynamisk granskning för de verktyg som bedömdes som bäst.

#	Kategori	Ivy	Phan	avet	donut	Shhh	Sgn	Veil	Wrai
1a	Skapa ny trovärdig process och injicera kod till denna	J	J	J	J	J	N	N	J
1b	Injicera kod till körande legitim process	N	J	J	N	N	N	N	N
2	Nedladdning av kod från extern plats	N	J	J	N	N	N	N	J
3	Sandlådetester	J	J	J	J	J	N	N	J
4	Hindra övervakningsfunktioner	J	J	N	J	N	N	N	N
5	Stänga av säkerhetsfunktioner	N	N	N	N	N	N	N	N

6 Slutsatser och framtida arbete

Denna rapport studerade forskningsfrågorna ”*Vilka verktyg finns det som kan dölja kända bakdörrar från säkerhetsmekanismer?*” samt ”*Vilka metoder tillämpas av olika verktyg för att dölja kända bakdörrar från säkerhetsmekanismer?*”

Som grund i analysen granskades 900 verktyg från databasen Github. Av de 900 verktygen bedömdes 174 som relevanta för forskningsfrågorna. Dessa 174 verktyg kategoriserades enligt fyra huvudkategorier – projektinformation, vilka filformat arkitekturer som stödjs, vilka försvar mot statisk granskning som erbjuds, samt vilka försvar mot dynamisk granskning som erbjuds.

Forskarna bedömer att många av verktygen skapades i utbildningssyfte snarare än för att tillföra nya metoder för att dölja bakdörrar. Denna tes stöds dels av faktumet att de flesta verktyg fick låga betyg, och dels genom hur de flesta verktyg designats: ett typiskt verktyg tillämpar polymorfism för att kryptera en PIC genom en Python-applikation, och en egenskapad hjälpapplikation i C för att dekryptera och exekvera binärkoden på en måldator. Python ses av de flesta som ett av de enklaste språken att skriva kod i, och de tillämpade krypteringsformerna består i de flesta fall bara av ett fåtal rader kod. XOR, en av de enklaste krypteringsmetoderna, är också den metod som oftast tillämpas av verktygen. Hjälpapplikationen består typiskt av ett fåtal rader kod som helt enkelt dekrypterar och exekverar den användarangivna koden, utan några egentliga försök att dölja dessa funktioner.

Gällande Lore ställs fyra huvudsakliga krav på obfuskeringsverktyg:

1. det måste kunna dölja Meterpreter
2. det måste vara effektivt
3. det måste vara robust
4. det skall klara att hantera x86-32- och x86-64-baserade Windows-operativsystem.

Krav #1 gäller Meterpreter-varianter i allmänhet, och C-kodsvarianten i synnerhet då denna har störst funktionsstöd. Krav #2 medför att verktyg oftast måste dölja Meterpreter från säkerhetsmekanismer. Krav #3 medför att koden producerad av ett verktyg inte får krascha eller liknande på målsystem. Krav #4 härrör från faktumet att antivirus under FOI:s cybersäkerhetsövningar primärt varit installerat på x86-baserade Windows-system.

I teorin uppfyller de allra flesta verktyg dessa krav, särskilt då ett och samma verktyg egentligen inte måste kunna hantera både x86-32 och x86-64. Det är dock inte möjligt att relatera verktygens egenskaper till olika säkerhetsmekanismers funktioner på ett tillfredsställande sätt. Exempelvis blir bra verktyg ofta populära, och populära verktyg är såklart attraktiva att upptäcka

för utvecklare av säkerhetsmekanismer. Faktiska tekniska tester av hur väl olika verktyg döljer bakdörrar krävs för att bedöma vilket, eller vilka, verktyg som är bäst lämpade för Lore.

Det finns två huvudsakliga problem som kan påverka studiens validitet (om resultatet speglar det som ämnas mätas) och reliabilitet (resultatets tillförlitlighet). Läsaren bör beakta dessa vid tolkning av studiens resultat.

- *Diverse verktyg som ofta används av verkliga skadliga koder täcktes inte in av sökningarna*. Exempelvis nämner Roundy och Miller [24] packningsverktygen Yoda's Protector, UPX, Upack, PolyEne, MEW, PECompact, NSPack, nPack, ASPACK, Petite och ASProtect. Inget av dessa verktyg täcktes in av studien. Anledningen till detta är huvudsakligen att de inte skapades för att kringgå säkerhetsmekanismer, utan för att helt enkelt minimera binärkoder. Detta medförde att de mer IT-säkerhetsinriktade sökorden för studien inte täckte in dem. En sekundär anledning är att vissa av verktygen inte finns tillgängliga på Github. Däremot finns exempelvis Yoda's Protector istället tillgängligt på SourceForge[43]. Samma problem finns för kommersiella verktyg som inte har sin kod på Github.
- *Olika forskare gör olika bedömningar*. Den systematiska granskningen tog höjd för att olika forskare kan göra olika bedömningar av samma verktyg genom en kombination av möten, workshops och dubbelgranskningar. Dessa metoder bidrog till en ökad konsensus, men resultatet hade troligen trots detta sett annorlunda ut om alla bedömningar gjorts av andra forskare.

Slutligen, medan studien bidrar med värdefull information om en tidigare outforskad forskningsfråga, kvarstår det ändock mycket arbete. Utöver att studera relevanta verktyg som inte täcktes in av granskningen (t.ex. UPX) är det även planerat att under 2023 genomföras tekniska tester av olika verktyg för att utvärdera hur väl de fungerar i praktiken, samt hur effektiva olika tekniker är. Det finns också många intressanta studier som kan göras genom fördjupade analyser av olika delar av kategoriseringsmodellen. Exempelvis vore det värdefullt att granska hur olika verktyg tillhandahåller kodinjicering (kategori #1 inom *försvar mot dynamisk granskning*), såsom hur deras implementation av APC injicering ser ut rent kodmässigt, samt hur de säkerställer att injicerad kod ser legitim ut (t.ex. hur import/exporttabeller konstrueras).

[43] https://sourceforge.net/projects/yodap/

Referenser

[1] H. Holm, ”Lore A Red Team Emulation Tool”, *IEEE Transactions on Dependable and Secure Computing*, 2022.

[2] Sommestad, Teodor, ”SAFE Cyber 2020, genomförande av distribuerad övning (FOI Memo 7522)”, Totalförsvarets forskningsinstitut (FOI), apr. 2021.

[3] A. P. Namanya, A. Cullen, I. U. Awan, och J. P. Disso, ”The world of Malware: An overview”, i *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*, 2018, s. 420–427.

[4] Ö. A. Aslan och R. Samet, ”A comprehensive review on malware detection approaches”, *IEEE Access*, vol. 8, s. 6249–6271, 2020.

[5] F. Guo, P. Ferrie, och T.-C. Chiueh, ”A study of the packer problem and its solutions”, i *International Workshop on Recent Advances in Intrusion Detection*, 2008, s. 98–115.

[6] I. A. Saeed, A. Selamat, och A. M. Abuagoub, ”A survey on malware and malware detection systems”, *International Journal of Computer Applications*, vol. 67, nr 16, 2013.

[7] S. S. Chakkaravarthy, D. Sangeetha, och V. Vaidehi, ”A survey on malware analysis and mitigation techniques”, *Computer Science Review*, vol. 32, s. 1–23, 2019.

[8] F. A. Aboaoja, A. Zainal, F. A. Ghaleb, B. A. S. Al-rimy, T. A. E. Eisa, och A. A. H. Elnour, ”Malware Detection Issues, Challenges, and Future Directions: A Survey”, *Applied Sciences*, vol. 12, nr 17, s. 8482, 2022.

[9] M. N. Olaimat, M. A. Maarof, och B. A. S. Al-rimy, ”Ransomware anti-analysis and evasion techniques: A survey and research directions”, i *2021 3rd international cyber resilience conference (CRC)*, 2021, s. 1–6.

[10] P. Chen, C. Huygens, L. Desmet, och W. Joosen, ”Advanced or not? A comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware”, i *IFIP International Conference on ICT Systems Security and Privacy Protection*, 2016, s. 323–336.

[11] M. Hammad, J. Garcia, och S. Malek, ”A large-scale empirical study on the effects of code obfuscations on Android apps and anti-malware products”, i *Proceedings of the 40th international conference on software engineering*, 2018, s. 421–431.

[12] J. Morales, S. Xu, och R. Sandhu, ”Analyzing malware detection efficiency with multiple anti-malware programs”, *ASE Science Journal*, vol. 1, nr 2, s. 56–66, 2012.

[13] O. Sukwong, H. Kim, och J. Hoe, ”Commercial antivirus software effectiveness: an empirical study”, *Computer*, vol. 44, nr 03, s. 63–70, 2011.

[14] A. Mills och P. Legg, ”Investigating anti-evasion malware triggers using automated sandbox reconfiguration techniques”, *Journal of Cybersecurity and Privacy*, vol. 1, nr 1, s. 19–39, 2020.
[15] N. Galloro, M. Polino, M. Carminati, A. Continella, och S. Zanero, ”A Systematical and longitudinal study of evasive behaviors in windows malware”, *Computers & Security*, vol. 113, s. 102550, 2022.
[16] E. Avllazagaj, Z. Zhu, L. Bilge, D. Balzarotti, och T. Dumitraș, ”When Malware Changed Its Mind: An Empirical Study of Variable Program Behaviors in the Real World”, i *30th USENIX Security Symposium (USENIX Security 21)*, 2021, s. 3487–3504.
[17] F. Barr-Smith, X. Ugarte-Pedrero, M. Graziano, R. Spolaor, och I. Martinovic, ”Survivalism: Systematic analysis of windows malware living-off-the-land”, i *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, s. 1557–1574.
[18] B. Kitchenham, ”Guidelines for performing systematic literature reviews in software engineering v2.3”, Keele University, Technical Report, 2007.
[19] M. L. McHugh, ”Interrater reliability: the kappa statistic”, *Biochemia medica*, vol. 22, nr 3, s. 276–282, 2012.
[20] H. Holm och T. Sommestad, ”Sved: Scanning, vulnerabilities, exploits and detection”, i *MILCOM 2016-2016 IEEE Military Communications Conference*, 2016, s. 976–981.
[21] M. Ramilli och M. Bishop, ”Multi-stage delivery of malware”, i *2010 5th International Conference on Malicious and Unwanted Software*, 2010, s. 91–97.
[22] A. Swinnen och A. Mesbahi, ”One packer to rule them all: Empirical identification, comparison and circumvention of current antivirus detection techniques”, *BlackHat USA*, 2014.
[23] O. Or-Meir, N. Nissim, Y. Elovici, och L. Rokach, ”Dynamic malware analysis in the modern era—A state of the art survey”, *ACM Computing Surveys (CSUR)*, vol. 52, nr 5, s. 1–48, 2019.
[24] K. A. Roundy och B. P. Miller, ”Binary-code obfuscations in prevalent packer tools”, *ACM Computing Surveys (CSUR)*, vol. 46, nr 1, s. 1–32, 2013.
[25] S. Euh, H. Lee, D. Kim, och D. Hwang, ”Comparative analysis of low-dimensional features and tree-based ensembles for malware detection systems”, *IEEE Access*, vol. 8, s. 76796–76808, 2020.
[26] B. Bencsáth, G. Pék, L. Buttyán, och M. Félegyházi, ”sKyWIper (aka Flame aka Flamer): A complex malware for targeted attacks”, *CrySyS Lab Technical Report, No. CTR-2012-05-31*, 2012.
[27] R. T. Narayanan, ”Novice programmer to new-age application developer: What makes python their first choice?”, i *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2019, s. 1–7.

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