LINUX SOFTWARE PROTECTION AGAINST REVERSE ENGINEERING

Tomáš Korec
Bachelor Degree Programme (3), FIT BUT
E-mail: xkorec00@stud.fit.vutbr.cz

Supervised by: Tomáš Hruška
E-mail: hruska@fit.vutbr.cz

Abstract: This thesis deals with an implementation of Linux software protection against reverse engineering. Protection consists of two layers. The first layer is represented by methods that aggravate the usability of tools, that enable the application of reverse engineering methods. Protection based on hashing functions represents the second layer which protects the application against changes. Possible future extensions are discussed at the end of the thesis.

Keywords: software protection, reverse engineering, cracking, Linux

1 ÚVOD

V této práci se rozebírá problematika ochrany softwaru před reverzním (též zpětným) inženýrstvím pro operační systémy postavené na linuxovém jádře (dále označované jako linuxové operační systémy nebo Linux). Zaobírá se způsoby, jak zhoršit, případně znemožnit, použitelnost nástrojů běžně dostupných v Linuxu, prostředníctvím kterých mohou být aplikovány jednotlivé postupy a metody reverzních inženýrů.

Ochrana se skládá ze dvou vrstev. První vrstvu představují metody chránící softwarový produkt proti analýze. Druhou vrstvu představuje metoda založená na hashování funkcí, která chrání aplikaci proti změnám a tedy proti vyřazení ochran první úrovně.

2 REVERZNÍ INŽENÝRSTVÍ

Reverzní inženýrství můžeme definovat jako: „Proces analýzy systému s cílem identifikovat jeho komponenty a vztahy mezi nimi a vytvořit reprezentaci systému v jiné formě nebo na vyšší úrovni abstrakce“ [1]. Toto bývá zneužíváno při crackingu, což je proces, jehož cílem je vyřadit nebo odstranit ochranné prvky, rutiny ověřující platnost a pravost licence a omezení daná licencí u proprietárního softwaru. Analýzu softwaru dělíme na statickou a dynamickou. Statická analýza je analýza binárních souborů aplikace bez jejich vykonání. Dynamickou analýzou nazýváme sbírání informací o aplikaci sledováním jejího vykonávání pomocí nástrojů určených k ladění (debugging) a sledování (tracing) aplikací.

3 NÁVRH ŘEŠENÍ

Ochranné rutiny nesmí být centralizované a volané jako funkce. Potom musí útočník odstavit rutinu na každém místě, kde je vložena. Nejvhodnějším způsobem implementace je využití maker. Inline funkce nejsou vhodné, protože klíčové slovo *inline* kompilátory berou pouze jako doporučení a nemusí tedy funkci vložit jako inline.

Následující metody chrání aplikaci proti sledování při jejím běhu. Tvoří první vrstvu ochrany. Metoda využívající hashování chrání aplikaci proti změnám a představuje druhou vrstvu ochrany.

3.1 Detekce breakpointů

Některé typy breakpointů jsou realizované přepsáním prvního bytu adresy, kam je breakpoint umísťovaný, operačním kódem 0xCC. Když tedy chceme odhalit breakpointy na funkce v aplikaci, stačí adresy funkcí porovnat s operačním kódem 0xCC [3].

Na této technice se dá postavit detekce linuxového nástroje *ltrace* tak, že budeme hledat operační kód breakpointu na nějaké knihovní funkci, kterou *ltrace* sleduje (například printf).

3.2 Falešné breakpointy

Do aplikace je možné vložit falešné breakpointy umístěním instrukce int3 [3]. To samo o sobě nemá žádný ochranný účel a navíc v kódech psaných ve vyšších programovacích jazycích vložení falešného breakpointu vyžaduje inline assembler. Na falešných breakpointech se však dá postavit ochrana.

Instrukce int3 se dá nahradit vysláním signálu SIGTRAP. Ochrana potom využívá toho, že debugger signál SIGTRAP zachytí a když jej nepošle procesu, obslužná rutina k tomuto signálu se nevykoná. Potom už jen stačí v obslužné rutině signálu nastavovat proměnnou a později zkontrolovat její obsah.

3.3 Detekce ladicích nástrojů pomocí *ptrace*

Systémové volání *ptrace* používá mnoho ladicích nástrojů a nástroje na sledování běhu aplikace. Z běžně dostupných nástrojů v Linuxu to jsou například *gdb*, *ltrace* a *strace*. Když proces zavolá *ptrace* sám na sebe a je již monitorovaný, skončí toto volání neúspěšně. Proces totiž může být v jeden čas monitorován pomocí *ptrace* jen jednou [3].

3.4 Detekce založená na časové značce

Ochrana je postavena na porovnávání dvou časových značek. Uložíme si první časovou značku a vyšleme nějaký signál, ke kterému budeme mít připravenou (prázdnou) obslužnou funkci. Poté si uložíme druhou časovou značku a porovnáme s první. Pokud bude aplikace spuštěná spolu s ladicím nástrojem, bude rozdíl časových značek několikanásobně větší než při normálním spuštění aplikace [2]. U ochrany je zapotřebí vhodně zvolit práh a brát v úvahu I/O zařízení, která by mohla při změně kontextu dobu běhu aplikace prodloužit.

3.5 Hashování funkcí

V ochraně tohoto typu je důležitá rychlost hashování a hlavně dobré ukrytí správného hashe (otisku). Pokud správný hash útočník odhalí, je jedno, jakým způsobem jsme hashovali, protože jej může jednoduše nahradit vlastní hodnotou. Jednoduchý a rychlý způsob, jak spočítat hash funkce, je využít logickou funkci xor. Stačí xorovat všechny bajty funkce, čímž dostaneme originální výsledek pro danou funkci. Kdyby se útočník pokusil funkci pozměnit, hodnota hashe bude zcela odlišná [2]. Metoda taktéž odhalí breakpointy vkládané pomocí přepisování operačním kódem breakpointu.

Jelikož je potřeba hash testovat na správnost, musí být někde v kódu uložený. Je tedy nutné, aby byl nějakým způsobem ukrytý nebo zamaskovaný. Logická funkce xor při dvou stejných vstupních operandech dává nulový výsledek. Když tedy při počítání hashe nad výsledkem vykonáme logický xor se správným výsledkem [2], funkce vracející hash při jeho správné hodnotě bude vracet nulu. V opačném případě bude vždy výsledek nenulový.

3.6 Odhalení útočníka

Co se bude dít při odhalení útočníka, je závislé na konkrétní chráněné aplikaci. Není vhodné aplikaci ihned ukončit a útočníkovi tak okamžitě prozradit místo, kde se kontrolní rutina nachází. Lepším

řešením je aplikaci pozměnit tak, že po čase dojde k jejímu pádu (například zavoláním destruktoru nějakého objektu, změněním hodnoty ukazatele, který aplikace používá).

3.7 EXPERIMENTÁLNE VÝSLEDKY

V tabulce 1 je vidět vliv ochrany na rychlost vykonávání chráněné aplikace. Bylo provedeno měření rychlosti aplikace počítající prvních 45 čísel fibonacciho posloupnosti pomocí rekurze.

Počet volání ochran	Naměřený čas [s]
0	81,503
1	81,750
46	81,791

Tabulka 1: Vliv ochrany na rychlost

3.8 MOŽNÁ VYLEPŠENÍ A POKRAČOVÁNÍ PRÁCE

Pokud by byla možnost větší integrace ochranné rutiny s chráněnou aplikací, metoda hashování by se dala rozšířit. Jednotlivé funkce by mohly počítat hashe některých dalších funkcí, čímž by došlo k vytvoření sítě, ve které by se funkce kontrolovaly navzájem. Dalším vylepšením této metody by bylo umístění kopií chráněných funkcí do aplikace. Kdyby funkce zjistila změnu jiné, jednoduše by tuto funkci opravila jejím přepsáním správným kódem[2].

Pokračováním práce by bylo vytvořit aplikaci, která by přeložený binární soubor zašifrovala. Šifrování by představovalo třetí vrstvu ochrany. Takto zašifrovaný binární soubor by znemožňoval statickou analýzu. Šifrování by doplňovaly již implementované ochranné rutiny zamezující změně a odhalující nástroje útočníka.

4 ZÁVĚR

Ochrana, která by se nedala obejít nebo odstranit, neexistuje. Vše je pouze otázkou času, možností a trpělivosti útočníka. Cílem ochrany softwaru je co nejvíce ztížit útočníkům práci. V ideálním případě do té míry, že bude odstranění ochrany pro útočníka časově neúnosné.

Výše uvedené metody se pokoušejí aplikaci chránit proti sledování ladicími nástroji a proti změnám. V kombinaci se šifrováním binárního souboru by mohly tvořit spolehlivou ochranu pro aplikaci.

5 PODĚKOVÁNÍ

Tento příspěvek vznikl za podpory grantů MPO FR-TI1/038, TAČR Alfa TA01010667 a Výzkumnému záměru MSM 21630528.

REFERENCE

[1] EILAM, Eldad. Reversing : Secrets of Reverse Engineering. Canada : Wiley Publishing, Inc., 2005. 589 s. ISBN 0-7645-7481-7, ISBN-13 978-0-7645-7481-8.

[2] COLLBERG, Christian; NAGRA, Jasvir. Surreptitious software : obfuscation, watermarking, and tamperproofing for software protection. 1st ed. United States : Addison-Wesley, 2010. 792 s. ISBN 0-321-54925-2.

[3] CESARE, Silvio. Linux anti-debugging techniques : (fooling the debugger). VX Heavens [online]. January 1999, [cit. 2011-03-04]. Dostupný z WWW: <http://vxheavens.com/lib/vsc04.html>.