SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 362

Neprimjetan nadzor aktivnosti u operacijskom sustavu

Robert Perica

Zagreb, lipanj 2012.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Da bi ste uklonili ovu stranicu obrišite naredbu `\izvornik`.

SADRŽAJ

1. Uvod

Računalna sigurnost predstavlja veliko područje računarske znanosti, koje se, kao i sva druga kritična područja, premalo istražuje i u koje se premalo ulaže. Svaka domena računarske znanosti i programskog inženjerstva u kojoj se stvaraju nove aplikacije, koje implementiraju staru ili novu funkcionalnost, teži za primjenom principa kvalitetnog programiranja, kako bi se moguće greške svele na najmanju moguću mjeru. Unatoč tome, programska podrška prebrzo izlazi na tržište, često nedovoljno testirana, jer uvijek postoji neki rok kojeg treba ispuniti.

Odavno je poznato da zloćudni programi (engl. *malware*) iskorištavaju propuste u programskoj podršci kako bi stekli ovlasti administratorskih korisnika, u svrhu iskorištavanja ili uništavanja računalnog sustava na kojem se nalaze. Prema Symantecovom godišnjem pregledu [60], tokom 2011. pojavilo se 4,500 novih ranjivosti. Broj uspješno obranjenih napada na računalne sustave je premašio 5.5 milijardi, dok se broj različitih vrsta zloćudnih aplikacija povećao na 403 milijuna. Napad se smatra uspješno obranjenim ukoliko je neki od alata za zaštitu (npr., antivirusni alat) pravilno uočio i spriječio prijetnju. Ovako "crna" statistika trebala bi ukazati na rastuću potrebu za razvojem naprednijih alata za obranu od zloćudnih aktivnosti.

Tržište virtualizacijskih tehnologija [24, 6, 15, 8] postaje sve veće, jer postaje očito kako instalacija samo jednog operacijskog sustava, na sustavima koji nude tolike performanse, ne iskorištava sav potencijal *hardvera*. Uvođenjem *hardverskih* proširenja za virtualizaciju na procesorima, virtualizirani sustavi su postali praktički ekvivalentni uobičajenim sustavima po performansama. Sustavi nadzora i detekcije napada [4, 3], te antivirusni alati [10, 18], koji se trenutno koriste u praksi, nalaze se na sustavu kojeg napadač pokušava osvojiti. Ukoliko napadač dobije administratorske ovlasti, on vrlo lako sve alate može ugasiti, izbjeći ili namjerno krivo konfigurirati [33, 43]. Osvanulo je vrijeme za sustave nadzora koji će se nalaziti izvan dohvata napadača, a virtualizirana okruženja upravo pružaju mogućnost da se takvi sustavi ostvare.

U ovom radu prikazano je trenutno stanje tehnologije, opće smjernice za izgradnju sustava nadzora, te testni prototip sustava nadzora. Prototip sustava služi kako bi se

lakše uočili trenutni propusti u tehnologiji, te kako bi se dobila jasna vizija smjera kojim treba ići da se identificirani propusti uklone. Unatoč brojnim pokušajima stvaranja sustava nadzora u virtualnim okruženjima, još nije poznato da postoji sustav koji se primjenjuje u produkcijskim okruženjima.

2. Stanje tehnologije

Svrha poglavlja je opisati tehnologije, metode i alate koji se spominju, a i koriste u ovom radu.

2.1. Operacijski sustav

Operacijski sustav je skup programa i alata koji omogućavaju korisnicima i korisničkim aplikacijama lakšu uporabu računalnih resursa i komponenti poput procesora, memorije, ili mrežnog priključka. Moderni operacijski sustavi najčešće imaju neki oblik grafičkog sučelja, kako bi se približili većem broju korisnika, ali grafičko sučelje nije neophodno za rad operacijskog sustava. U suštini, operacijski sustavi predstavljaju međusloj između *hardvera* računala i programske podrške, te omogućavaju svim programima siguran i jednoznačan pristup resursima sustava.

Tanenbaum [61] u uvodnom poglavlju opisuje razvoj operacijskih sustava kroz nekoliko generacija. U prvu generaciju pripadaju sva ona gigantska računala koja su nastala nakon Drugog svjetskog rata, poput *ENIAC-a*, ili računala *Z3*. Ta su računala koristila tehnologije vakuumskih cijevi, i obično su se programirala bušenim karticama. Nedugo, javljaju se računala druge generacije, koja su koristila tranzistorsku tehnologiju. Računala te generacije imala su naziv *mainframes*, i slijedno su obrađivala poslove (engl. *jobs*) koji su im bili predani u skupinama (engl. *batch*). Računala te generacije se zato ponekad nazivaju i *batch* računalima.

Pojavom integriranih krugova i multiprogramiranja, osvanula je i treća generacija računala. Za razliku od prethodne generacije, ova računala su bila brža, i omogućavala su izvršavanje više poslova (procesa) u prividno paralelnom načinu rada. Dok je jedan proces čekao na rezultat neke UI[1] operacije, procesorsko vrijeme se predalo nekom drugom procesu koji je bio spreman za izvršavanje. Zadnja generacija računala traje i dan danas, a obilježava ju masovno korištenje osobnih računala. Ovo razdoblje se još naziva i *PC erom*.

[1] ulazno-izlazne

2.2. Virtualizacija

Pojam virtualizacija se često odnosi na stvaranje nekakve vrste privida. Tako se u računarstvu koncept virtualizacije prvenstveno odnosi na stvaranje privida postojanja više računala na jednom fizičkom računalu. Jednostavnije rečeno, na jednom fizičkom računalu instalira se odgovarajuća programska podrška koja omogućava instalaciju nekoliko operacijskih sustava (engl. *guest operating system*) koji se izvršavaju "paralelno" i odvojeno na fizičkom stroju. Paralelno je namjerno smješteno u navodnike, jer pravi paralelizam u virtualnim okruženjima još nije moguće u potpunosti postići. Svaki sustav koji je instaliran u virtualnom okruženju naziva se virtualni stroj (engl. *virtual machine*), što zapravo predstavlja objekt koji enkapsulira funkcionalnost instaliranog operacijskog sustava i svih aplikacija unutar operacijskog sustava.

Zanimljivo je primijetiti da koncept virtualizacije postoji još od šezdesetih godina prošlog stoljeća, kad su se u IBM-u razvijali računalni sustav *CP-40* i eksperimentalni računalni sustav naziva *IBM M44/44X* [28]. Taj sustav je trebao simulirati nekoliko računalnih sustava *IBM 7044* na jednom stroju. Kao svaka druga dobra tehnologija, tako je i ova bila daleko ispred svog vremena. Virtualna okruženja su istinski zaživjela početkom ovog tisućljeća, jer su se performanse računala toliko povećale da je postalo nepraktično instalirati samo jedan operacijski sustav na računalo. Npr., u nekoj organizaciji može postojati potreba za većim brojem poslužitelja — svaki poslužitelj zauzima i grije prostor oko sebe, dok procesor svakog poslužitelja najveći dio vremena provodi u čekanju (engl. *idle*). Iskorištavanjem virtualizacije, moguće je smjestiti velik broj poslužitelja na jedno računalo. Time se povećava iskorištenost prostora, smanjuje se rizik od pregrijavanja, a procesor se više koristi, obzirom da mora posluživati nekoliko operacijskih sustava, sve aplikacije, te program nadzornik za virtualizaciju. Osim nabrojanih ekoloških i znanstvenih prednosti, ovakav princip iskorištavanja sustava donosi i veliku financijsku uštedu.

2.2.1. Svojstva virtualizacije

Popek i Goldberg [54] u svom radu navode tri svojstva koja kontrolni program za virtualizaciju (engl. *VMM*[2]*, virtual machine monitor*) mora ispuniti. Svojstva su:

Ekvivalencija: Program koji se izvršava pod *hypervisorom* treba pokazivati ponašanje identično programu izvršenom na ekvivalentnom računalu.

Kontrola resursa: *Hypervisor* mora biti u potpunoj kontroli nad resursima.

[2] u daljnjem tekstu *hypervisor*

Učinkovitost: Bez uplitanja *hypervisora* mora se moći izvršiti statistički dominantna količina instrukcija.

Ova tri svojstva moraju vrijediti za bilo koji *hypervisor* koji se koristi u virtualiziranim okruženjima. Osim ova tri svojstva, jedno svojstvo koje proizlazi iz same definicije virtualizacije jest izolacija. Pod izolacijom se misli na odvojenost virtualnih strojeva. Smisao je da napad ili oštećenje jednog od virtualnih strojeva ne utječe na normalan rad ostalih virtualnih strojeva, pod uvjetom da je program nadzornik virtualnih strojeva ispravan. Svojstvo izolacije, kao i druga svojstva koja proizlaze iz karakteristika *hypervisora* bit će obrađena u narednim poglavljima koja se odnose na sigurnost virtualnih strojeva.

Virtualizacija se može promatrati s dva stajališta:

1. gdje je smješten *hypervisor* u odnosu na *hardver* računala
2. radi li se o punoj, djelomičnoj ili paravirtualizaciji

2.2.2. Hypervisor

Nadzornik virtualnih strojeva (engl. *VMM, virtual machine monitor, virtual machine manager, hypervisor*[3]) je program koji omogućava stvaranje virtualnog okruženja. Operacijski sustavi zahtijevaju resurse, poput memorijskog prostora i procesorskog vremena. Oni moraju "živjeti" u uvjerenju da imaju potpuni nadzor nad tim resursima, što i jest najčešći slučaj, s obzirom da je i dalje dominantna paradigma *jedno računalo — jedan operacijski sustav*.

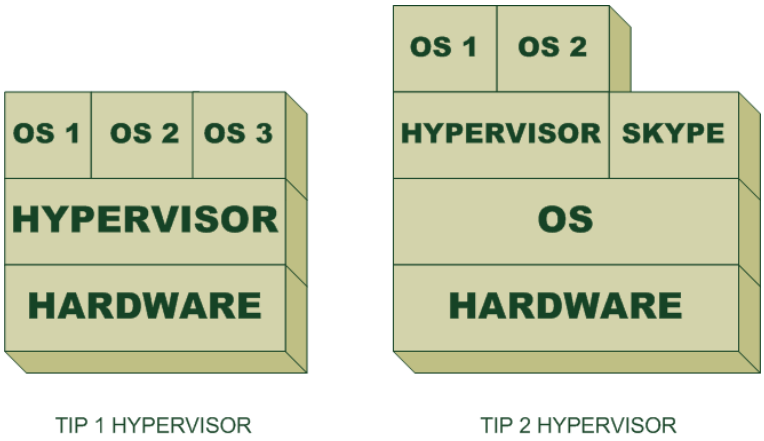
U virtualnom okruženju, više operacijskih sustava dijeli ograničeni skup resursa, pa kako se izvorni kod operacijskih sustava ne bi morao mijenjati, treba ih "uvjeriti" kako su oni jedini kontrolni program na računalu. Tu dužnost preslikavanja i raspodjele resursa ima *hypervisor*. Jednostavnije rečeno, *hypervisor* glumi operacijski sustav, a instalirani operacijski sustavi su njegove aplikacije. Pratimo li objektno-orijentiranu paradigmu, *hypervisor* je ujedno konstruktor, destruktor, *garbage collector* i dio programske logike koja dodjeljuje dodatne resurse objektu (u ovom slučaju, virtualnom stroju) na zahtjev. Dodatno, prema [41] postoje dvije vrste (dva tipa) *hypervisora*:

Tip 1 Izvršava se direktno iznad *hardvera* računala, prvi oblik *hypervisora* koji je bio u upotrebi (engl. *bare metal, native*)

[3] u tekstu se koristi ovaj izraz jer ne postoji službena riječ ili kovanica u hrvatskom jeziku

Tip 2 Izvršava se u nekom uobičajenom operacijskom sustavu, kao aplikacija (engl. *hosted*)

Ovakvo objašnjenje je dosta "suho", pa je grafički prikaz ova dva tipa predočen na slici 2.1.



Slika 2.1: Tipovi *hypervisora*

Tip 1 *hypervisor* se izvršava direktno iznad *hardvera* računala, glumeći operacijski sustav na tom računalu. Sučelje koje se oslanja na *hardver* koristi sve njegove mogućnosti, a sučelje koje se nudi operacijskim sustavima koji se instaliraju nad *hypervisorom* je zapravo identično sučelju *hardvera*. Korištenjem ove logičke odvojenosti, *hypervisor* može implementirati jedinicu za upravljanje memorijom (engl. *MMU, memory managment unit*) i raspoređivač procesa (engl. *scheduler*), koji je za instalirane operacijske sustave u potpunosti transparentan — operacijski sustavi smatraju da su sami na stroju, i potpuno su međusobno izolirani. Prvi pokušaji virtualizacijske tehnologije[4] koristili su upravo ovaj tip *hypervisora*. Današnji predstavnici *tip 1* arhitekture su *VMWare ESX/ESXi*[24] i Microsoft *Hyper-V*[15].

Tip 2 *hypervisor* mora postojati kao aplikacija unutar nekog operacijskog sustava. Operacijski sustav unutar kojeg je tip 2 *hypervisor* instaliran naziva se operacijski sustav domaćin (engl. *host OS*). Obzirom da *tip 2 hypervisori* rade u okruženju u kojem se nalaze brojni drugi procesi, a moraju prolaziti kroz još jednu dodatnu razinu apstrakcije (operacijski sustav domaćin) korištenjem pripadnih sistemskih poziva, za očekivati je da su sporiji u odnosu na *tip 1 hypervisore*. Otkad su uvedena virtualizacijska proširenja na procesorima (*Intel VT-x* [29], *AMD-V* [9]) razlike u performansama su praktički zanemarive [36], [63]. Predstavnici ove arhitekture su *VirtualBox* [22] i *VMWare Workstation*[26].

[4] IBM-ov *CP-40*

Potrebno je izdvojiti još jedan *hypervisor* imena *KVM* (engl. *kernel-based virtual machine*) [6]. *KVM hypervisor*, u obliku Linux jezgrenih modula, stvara virtualizacijsku infrastrukturu koja može koristiti procesorska virtualizacijska proširenja. Sam po sebi, *KVM* ne radi nikakvu emulaciju, nego samo omogućava pristup virtualizacijskim proširenjima procesora i korištenje `/dev/kvm` sučelja na kojem se može postaviti adresni prostor procesa ili operacijskog sustava. U kombinaciji s emulatorom procesora *QEMU* [7], omogućava jednostavnu, kvalitetnu i besplatnu virtualizaciju. Ova kombinacija alata je izdvojena jer je sustav GlassRoom implementiran korištenjem, između ostalih, i ta dva programska alata.

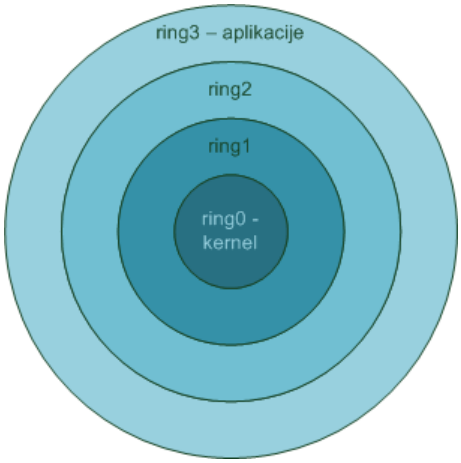
Osim tipova *hypervisora*, bitno je razjasniti i nazivlje pojedinih dijelova virtualiziranog okruženja. Operacijski sustav gost (engl. *guest OS*[5]) je operacijski sustav koji se nalazi u virtualnom stroju, odnosno operacijski sustav **koji je virtualiziran**. Operacijski sustav domaćin (engl. *host OS*) je operacijski sustav **u kojem** se nalazi virtualizacijski alat, odnosno u kojem se pokreće virtualizacijski alat kojim se virtualiziraju *guest OS-ovi*. U okruženju tip 1 *hypervisora* ne postoji koncept *host OS-a*, jer se takav *hypervisor*, kako je već navedeno, izvršava direktno iznad *hardvera*, te se, na neki način, može reći da je u ovom slučaju sam *hypervisor host OS*.

2.2.3. Puna virtualizacija

Puna virtualizacija (engl. *full virtualization*) je virtualizacijska tehnika koja pruža potpuno virtualizirano okruženje. Drugim riječima, sve mogućnosti *hardvera*, poput instrukcijskog seta procesora, memorijskih i ulazno-izlaznih operacija, su preslikane i omogućene da ih virtualizirani operacijski sustavi koriste. Problem kod pune virtualizacije x86 arhitekture procesora nastaje zbog tzv. *ringova*, odnosno sigurnosnih razina u kojima se izvršava kod programa.

Na slici 2.2 vidljivo je kako se korisničke aplikacije izvršavaju u *ring 3*, a jezgra (engl. *kernel*) u *ring 0*. Koncept prstena je dodatno objašnjen u poglavlju 2.3. *Ring 3* je najmanje, a *ring 0* najviše privilegirana razina. Svi operacijski sustavi pretpostavljaju da se izvode u *ring 0* razini, i da imaju pristup svim mogućnostima *hardvera*. **Tip 1** *hypervisor* se mora smjestiti na razinu *ring 0*, kako bi mogao ostvariti svoju funkcionalnost, ali se onda virtualizirani operacijski sustavi moraju smjestiti u neku od preostalih razina. Ostale razine nemaju ovlasti izvršavanja određenih procesorskih instrukcija, pa, prema tome, određene funkcionalnosti operacijskih sustava ne bi mogle funkcionirati. *VMWare ESX*[62] je bio jedan od prvih virtualizacijskih alata koji je riješio taj

[5] nadalje će se koristiti ovaj izraz



Slika 2.2: "Prsteni" privilegija

problem i omogućio punu virtualizaciju prije nego što su procesorska virtualizacijska proširenja postala dostupna. Problem je bio riješen tehnikom zvanom *binarna translacija*. Ta tehnika zamjenjuje naredbe u kodu, koje nije moguće izvršiti, naredbama "zamkama" (engl. *traps*) koje *hypervisor* prepoznaje i emulira u svojoj programskoj logici. Nakon što su procesorska virtualizacijska proširenja postala dostupna, više nije bilo potrebno pribjegavati ovakvim rješenjima. Osim toga, proširenja su omogućila pojavu **tip 2** *hypervisora* koji omogućava punu virtualizaciju.

Procesorska virtualizacijska proširenja

U poglavlju 2.2.2 navedeno je kako su Intel i AMD uveli virtualizacijska proširenja u svoje procesore. To se dogodilo oko 2006. godine, što znači da ranije nastali procesori nemaju podršku za virtualizaciju. Virtualizacijska proširenja su skup instrukcija i kontrolnih struktura koje omogućavaju *hypervisorima* iskorištavanje procesorske arhitekture za postizanje virtualizacije, umjesto da sami emuliraju ponašanje koje pripada procesoru i koje će inherentno biti sporije ako se emulira u programskoj logici. Virtualizacijske tehnologije koje iskorištavaju procesorska virtualizacijska proširenja se nazivaju *hardverski potpomognutim virtualizacijama* (engl. *hardware-assisted virtualization, native virtualization, accelerated virtualization*).

2.2.4. Djelomična virtualizacija

Djelomična virtualizacija (engl. *partial virtualization*) služi za virtualizaciju velikih količina istog resursa, kao što je primjerice memorijski prostor. Ovakva vrsta virtualizacije ne omogućava pokretanje operacijskih sustava, ali omogućava pokretanje

velikog broja aplikacija. Primjeri upotrebe ove tehnike bili su vidljivi u prvim *time-sharing* ili *memory-sharing* sustavima, te su predstavljali veliki korak naprijed prema ostvarenju pune virtualizacije.

2.2.5. Paravirtualizacija

Paravirtualizacija (engl. *paravirtualization, OS supported virtualization*) je vrsta virtualizacije za koju je potrebno modificirati operacijski sustav koji se namjerava instalirati, kako bi se mogao pokrenuti u virtualiziranom okruženju. Jedan od značajnijih virtualizacijskih alata koji koristi ovu metodu jest *Xen* [32]. Xen u originalu nije imao mogućnost preslikavanja *hardvera*, nego je pružao svoj *Xen API*, za koji je jezgra operacijskog sustava gosta morala biti promijenjena. Ova metoda virtualizacije nije baš zaživjela jer je kod *closed-source* jezgara, poput operacijskog sustava Windows, bilo jako teško napraviti potrebne preinake da se iskoriste sve mogućnosti *para-API-ja*[6].

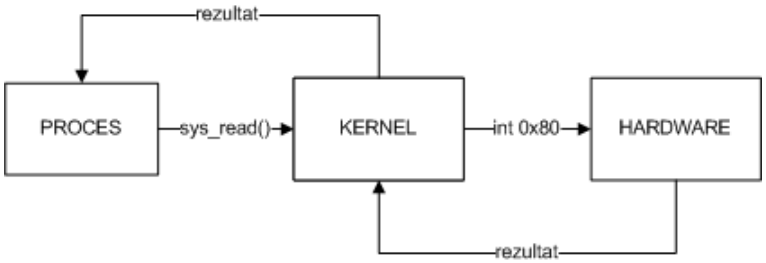
2.3. Sistemski pozivi

Operacijski sustav, kako je navedeno u poglavlju 2.1, je program koji predstavlja sloj između *hardvera* i programske podrške. Privilegija izvršavanja operacijskog sustava viša je od privilegije uobičajenih korisničkih programa. Naime, na x86 arhitekturi procesora, postoje četiri razine privilegija nazvane *prsteni* (engl. *rings*), navedeni u poglavlju 2.2.3. Prsteni su osmišljeni kako bi se razdvojile razine pristupa pojedinih aplikacija, jer nema smisla da neka korisnička aplikacija udvaja posao koji bi inače mogao obavljati operacijski sustav. Tako se operacijski sustav nalazi i izvršava u najvišoj razini ovlasti (*ring 0*), a korisničke aplikacije u najmanjoj razini ovlasti (*ring 3*). Prstenovi 1 i 2 su u pravilu neiskorišteni. Kako bi aplikacije mogle raditi i druge operacije osim matematičkog preračunavanja, bilo je potrebno osmisliti siguran način da aplikacije mogu pristupiti svim resursima računala. Rješenje se ostvarilo u obliku sistemskih poziva.

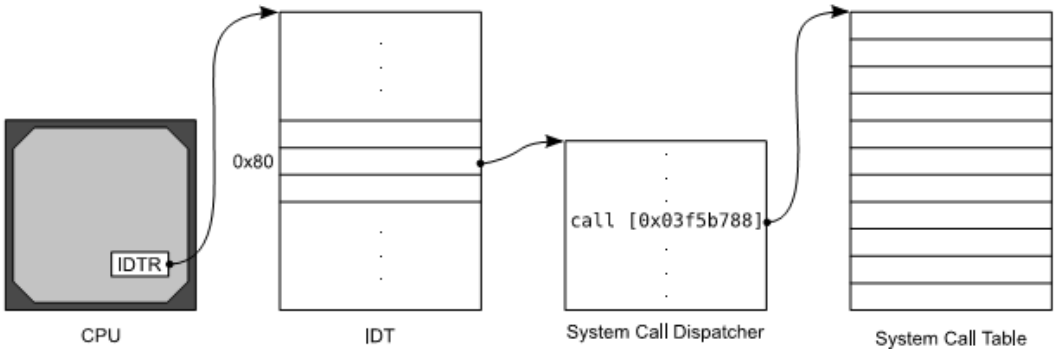
Sistemski pozivi korisničkim aplikacijama predstavljaju sučelje prema *hardveru*, kako bi mogli na jedinstven način pristupiti svim funkcionalnostima računala, ne zamarajući se detaljima implementacije (npr. koji niz bajtova treba poslati kako bi se kontroler čvrstog diska navelo da pomakne glavu čitača na određeni sektor). Zanimljivo je primijetiti kako operacijski sustav Linux uobičajeno pruža približno tristo sistemskih poziva, dok operacijski sustav Windows navodno broji u tisućama [48]. Na

[6] paravirtualizacijskog API-ja

slikama 2.3 i 2.4 moguće je vidjeti logički tok instrukcija prilikom pozivanja sistemskih poziva.



Slika 2.3: Logički tok sistemskog poziva



Slika 2.4: Adresiranje na razini procesora (uzeto iz [52])

Za pozivanje sistemskog poziva, korisnička aplikacija ima na raspolaganju tri mehanizma koje joj nudi x86 arhitektura. Sistemske pozive moguće je pozvati korištenjem naredbe `int`, naredbom `SYSCALL` i naredbom `SYSENTER`. Naredba `int` generira programski prekid (engl. *software interrupt*), te joj kao parametar treba predati broj koji predstavlja redni broj prekidnog vektora. Prekidni vektori se nalaze u tablici opisnika prekida (engl. *Interrupt Descriptor Table, IDT*), te je svaki unos u toj tablici veličine 8 bajtova. Primjerice, ako se naredbi `int` preda broj 128, to znači da se adresira 128. prekidni vektor u *IDT* tablici. Na toj lokaciji se nalazi adresa procedure koja se izvršava u slučaju generiranja prekida. x86 arhitektura podržava do 256 prekidnih vektora. U slučaju da se pokuša adresirati neki vektor izvan tog raspona, generirat će se opća povreda zaštite (engl. *general protection fault*). Druge dvije naredbe su uvedene zadnjih godina, a predstavljaju način ubrzavanja sistemskih poziva s obzirom na učestalost korištenja sistemskih poziva u računalnim programima.

Prilikom pozivanja sistemskih poziva, potrebno je predati parametre, jer su sistemski pozivi također funkcije. Arhitektura i386 ima konvenciju da se u registar `eax` stavlja broj sistemskog poziva, dok se u registre `ebx, ecx, edx, esi` i `edi` stavljaju

parametri pripadnog sistemskog poziva. Parametre koje sistemski pozivi prihvaćaju moguće je vidjeti u specifikaciji sistemskih poziva dotičnog operacijskog sustava.

2.4. Adresiranje memorije na x86 procesorima

Postoje dvije metode adresiranja memorije: segmentacija i straničenje [34]. **Segmentacija** pretpostavlja da se aplikacija može podijeliti u smislene cjeline (segmente) koji se prilikom izvršavanja aplikacije smještaju u memoriju. Postoje kodni (engl. *code*), podatkovni (engl. *data*) i stogovni (engl. *stack*) segmenti. Segmentacija omogućava iskorištavanje istog kodnog segmenta ukoliko se aplikacija pokrene veći broj puta, a svakoj instanci programa se pridružuju zasebni podatkovni i stogovni segmenti. Segmenti se adresiraju, odnosno indeksiraju, korištenjem segmentnih registara.

Straničenje (engl. *paging*) koristi memorijske stranice koje se smještaju u različite razine memorijske hijerarhije i zamjenjuju po potrebi. Zamjene se događaju ukoliko se dogodi promašaj, odnosno ako se stranica ne nalazi u memoriji te hijerarhije. Za razliku od segmentacije, straničenje omogućuje korištenje memorijskih blokova koji se međusobno ne "naslanjaju", odnosno mogu biti jako udaljeni u memoriji, a svejedno sadržavati slijedne podatke.

U operacijskom sustavu Linux dominantan način korištenja memorije je straničenje, dok se u operacijskom sustavu Windows pretežno koristi segmentacija. Usko vezano uz memorijska adresiranja, potrebno je spomenuti razliku između logičkih, linearnih (virtualnih) i fizičkih adresa. **Logičke** adrese su adrese koje korisnički programi koriste kako bi pristupili memoriji. Nakon postupka segmentacije, logičke adrese se pretvaraju u **linearne** adrese. Postupkom straničenja, linearne adrese se transformiraju u **fizičke** adrese, kojima je moguće adresirati stvarne lokacije u fizičkoj memoriji.

2.5. Sustavi za nadzor aktivnosti

Infrastrukturu, poput računalnog sustava ili lanca trgovina, moguće je pokušati napasti čim se ta infrastruktura "stvori", odnosno postane dostupna javnosti. Kako bi se vjerojatnost napada smanjila, ili kako bi se iz napada koji su se dogodili naučilo što više i uhvatilo krivce, počeli su se proizvoditi sustavi za nadzor aktivnosti. U fizičkom svijetu sustavi za nadzor aktivnosti najčešće predstavljaju *RFID*[7] čitači kartica ili kamere koje nadziru zaposlenike ili kritične dijelove organizacije.

[7] *Radio-Frequency IDentification*

U računalnoj domeni, na žalost, ne postoje ekvivalenti "opipljivih" zapisa kao što su snimke s kamere, već se sva aktivnost bilježi u datoteke na sustavu. Osim uobičajenih datoteka koje prate normalan rad sustava, obično se instaliraju posebni alati, poput sustava za detekciju ili prevenciju provale, koji bilježe i neke dodatne podatke oko uočenih aktivnosti. Takvi se sustavi konfiguriraju pomoću pravila kojima se određuje što se točno želi pratiti, te koje je ponašanje prihvatljivo, a koje nije. Najčešća se podjela ovih sustava radi po primjenjivosti, pa se tako govori o mrežnoj i lokalnoj primjenjivosti (engl. *network and host based*). Sažeti pregled tipova nadzora slijedi u idućim potpoglavljima.

2.5.1. Sustavi za detekciju provale

Sustav za detekciju provale (engl. *Intrusion Detection System, IDS*) je alat ili skup alata koji prate mrežne ili sistemske aktivnosti kako bi detektirali zlonamjerno ponašanje, koje se može klasificirati kao napad. Napadi koji se otkrivaju mogu biti učinjeni ili u tijeku. Obzirom da ponekad postoji vremensko kašnjenje u odnosu na vrijeme napada i vrijeme provjere (npr., ako je sustav namješten da radi intervalne provjere), to može utjecati na vrijeme detekcije napada. Kako im ime kaže, ovi sustavi služe samo za detekciju, i nemaju mogućnost odgovaranja na napad — napad se zabilježi, podiže se uzbuna i korisnik se obavijesti. Ovakvi sustavi ostvaruju funkcionalnost nadzora iskorištavanjem principa detekcije nepravilnog ponašanja, otkrivanjem poznatih "potpisa" (engl. *signature*) napada ili provjerom integriteta određenih datoteka.

Problem lokalnih alata je što se u pravilu nalaze na sustavu koji se napada, što znači da jednom kad napadač uspješno provali u sustav, on može uz vrlo malo truda isključiti ili krivo konfigurirati sustav nadzora. Situacija je nešto bolja za mrežne sustave za detekciju provale, koji se obično nalaze na nekom izdvojenom računalu koje analizira sav mrežni promet, ali se i takvi sustavi mogu zaobići korištenjem *DoS*[8] napada. Značajni primjeri ovakvih alata su Tripwire [4] i Snort [3].

2.5.2. Sustavi za prevenciju provale

Slijedeći ideju sustava za detekciju provale, sustavi za prevenciju provale (engl. *Intrusion Detection and Prevention Systems, IDPS*) su alati koji osim detekcije imaju ugrađene mehanizme za reagiranje na potencijalno zlonamjerne aktivnosti. Principi detekcije su slični, a u mnogim slučajevima isti, kao i u sustavima za detekciju pro-

[8] *Denial of Service*

vale. Mehanizmi prevencije obično funkcioniraju po principu zaustavljanja aplikacije koja bi mogla uzrokovati potencijalnu štetu, odbacivanjem mrežnih paketa koji su označeni kao zloćudni i zabranjivanjem veza prema računalima koja su identificirana kao izvori napada. Kombinacijom sustava za detekciju provale i *firewalla* može se ostvariti sustav za prevenciju provale.

2.5.3. Sustavi za privlačenje napadača

Sustavi za privlačenje napadača (engl. *honeypots*) su računala s posebno instaliranim alatima. Njihova je namjera namamiti napadače kako bi napali to računalo u svrhu istraživanja novih vrsta i statistika napada (poput izvorišta i učestalosti) [55]. Napadači se privlače prividnim otvaranjem različitih ranjivosti na sustavu. Ovakvi sustavi se mogu podijeliti na sustave za detekciju napadača i sustave za praćenje aktivnosti napadača. Iako se ova terminologija djelomično kosi s nazivima prošlih poglavlja, ipak će se koristiti u okviru ovog potpoglavlja.

Sustavi za detekciju napadača (engl. *low-interaction honeypots*) predstavljaju jednostavne sustave kojima je glavni cilj otkrivanje kvantitativnih podataka o napadačima, kao što su broj napada, broj napadača, učestalost napada, itd. Sjajan primjer ovakvog sustava je sustav HoneyD [2]. Postavljanje ovakvih sustava ne predstavlja veliki sigurnosni rizik, jer su sve ranjivosti na sustavu virtualne, što znači da ih nije moguće iskoristiti, jer pružaju nisku razinu interakcije.

Sustavi za praćenje aktivnosti napadača (engl. *high-interaction honeypots*) predstavljaju ozbiljniji mamac napadaču — u ovom je slučaju postavljene ranjivosti moguće iskoristiti. Ovakvi sustavi se obično koriste kako bi se pronašli najnoviji načini napada na usluge, ali zbog velikih zahtjeva na interaktivnost, predstavljaju i veliki sigurnosni rizik u slučaju uspješne provale. Kvaliteta analize napada ovisi količini podataka prikupljenih tokom napada, pa se u tu svrhu prate svi detalji oko ranjivih aplikacija. Primjer ovakvog sustava je Argos [1].

2.5.4. *Sandbox* sustavi

Sandbox sustavi su kontrolirana okruženja u kojima se izvršavaju potencijalno zloćudne aplikacije. Uobičajenim pristupom, postavilja se jedno računalo, koje nije spojeno na ostatak mrežne infrastrukture, na kojem se takvi programi mogu testirati. Na žalost, ovakvo ulaganje je obično preveliki trošak vremena i novčanih sredstava. Kao (financijski) prikladnija alternativa, koriste se virtualni strojevi. Virtualizacijski alati omogućavaju pohranjivanje trenutnih slika sustava (engl. *snapshots*), na koje se može

vratiti prilikom veće katastrofe na sustavu. Na taj način se izbjegava mukotrpna i nepotrebna reinstalacija sustava, i ubrzava proces analize.

3. Nadzor virtualnih strojeva

Ovo poglavlje opisuje osnovne ideje i metode za stvaranje okruženja koje podržava nadzor virtualnih strojeva (engl. *Virtual Machine Introspection, VMI*). Područje nadzora virtualnih strojeva smatra se relativno mladim, obzirom da je započelo oko 2003. godine. Osim već nekih poznatih svojstava i karakteristika sustava, predlažu se i neka poboljšanja i zahtjevi na buduće sustave. Od ovog poglavlja na dalje, promatrat će se (osim ako nije eksplicitno navedeno drugačije) **tip 2** ***hypervisori***. Iako se mnoga svojstva mogu odnositi i na tip 1 *hypervisore*, bilo bi potrebno promijeniti nazivlje komponenti nad kojima se *hypervisor* izvršava.

3.1. Svojstva nadzora

U ovom poglavlju nalaze se definicije koje se susreću kada se govori o nadzoru virtualnih strojeva, pravila koja treba poštovati, te zamke na koje se može naići pri pojedinom pristupu. Opisan je i koncept semantičke praznine, nakon čega slijedi opis metoda kako istu premostiti.

3.1.1. Opći zahtjevi za nadzor vezani za *hypervisor*

U svom radu o izgradnji sustava za detekciju provale temeljenom na nadzoru virtualnih strojeva, Garfinkel i Rosenblum [40] su naveli tri ključna svojstva koja moraju biti zadovoljena kako bi se mogao izgraditi kvalitetan i pouzdan sustav nadzora virtualnih strojeva. Točnije, implementacija *hypervisora* mora zadovoljavati ta svojstva, koja predstavljaju ključne prednosti ovakve vrste nadzora u odnosu na već poznate i korištene tehnike nadzora. Zbog tih prednosti, nadzor virtualnih strojeva pripada u superiorniju klasu nadzora. Svojstva koja je potrebno zadovoljiti su:

- izolacija,
- inspekcija,
- interpozicija.

Izolacija je karakteristika koja slijedi direktno iz definicije virtualnog stroja. Odnosi se na (virtualno) fizičku odvojenost pojedinih virtualnih strojeva, kao i odvojenost svih virtualnih strojeva od *host OS-a* u kojem se virtualizacijski alat nalazi. Drugim riječima, izolacija je svojstvo koje osigurava da *softver* koji se pokreće unutar virtualnog stroja ne može pristupiti ili promijeniti *softver* koji se izvršava u *hypervisoru* ili nekom drugom virtualnom stroju. To znači da u slučaju kad napadač uspije podvrgnuti *guest OS* svojoj kontroli, on tu kontrolu ne može iskoristiti na način da ugrozi drugi virtualni stroj ili sustav detekcije provale.

Inspekcija je svojstvo koje osigurava da je moguće dohvatiti sve relevantne podatke iz virtualnog stroja, koji su predstavljeni stanjem virtualnog stroja unutar *hypervisora*. *Hypervisor*, kako bi mogao pokretati više virtualnih strojeva istovremeno, mora imati sliku stanja sustava. U radu Pfoh et al. [51] se spominje kako je stanje nekog računala jednako zbroju *softverskog* i *hardverskog* stanja. *Softversko* stanje čini skup svih podataka iz promjenjive memorije i skup svih podataka u permanentnoj memoriji, a *hardversko* stanje čini stanje procesora (procesorski registri i zastavice) te ostalih fizičkih komponenti (npr., *pending interrupts*[1]). Jednostavniji prikaz ovog paragrafa dan je jednadžbama:

$$stanje\ sustava = softversko\ stanje + hardversko\ stanje, \tag{3.1}$$

$$softversko\ stanje = promjenjiva\ memorija + permanentna\ memorija, \tag{3.2}$$

$$hardversko\ stanje = stanje\ procesora + ostale\ komponente. \tag{3.3}$$

Zahvaljujući ovom svojstvu, moguće je razmotriti korištenje već gotovih *host-based IDS* rješenja za nadzor virtualnih strojeva [46], jer je moguće pristupiti svim podacima o sustavu "izvana". S obzirom na uvedena procesorska virtualizacijska proširenja, nije jednostavno pratiti svaku izvedenu instrukciju, jer se velik dio instrukcija šalje direktno procesoru na izvršavanje, a da se ne "love" u *hypervisoru*.

Zadnje svojstvo gornjeg trojca jest **interpozicija**, odnosno sposobnost *hypervisora* da učini neku aktivnost nad virtualnim strojem. Ta aktivnost može biti: zaustavljanje virtualnog stroja, zaustavljanje aplikacije, dojava sustavu nadzora ili bilo kakva druga aktivnost koju sustav za detekciju i prevenciju provale može iskoristiti kako bi spriječio potencijalni napad. Kako bi *hypervisor* mogao provoditi ovakve akcije, često je potrebno promijeniti njegov kod. U ovom trenutku nije poznato da neki od proizvođača virtualizacijskih alata pruža sve mogućnosti za izgradnju sustava nadzora virtualnih strojeva.

[1] prekidi vanjskih jedinica koji čekaju na obradu

U radu Payne et al. [50] je izneseno ukupno šest svojstava nadzora, ali nisu sva usko vezana za *hypervisor*. *Hypervisor* se uvjetuje još jednim ključnim svojstvom, iako je zapravo više riječ o naglašavanju poznatog principa *KISS*[2] — **u *hypervisor* ne treba dodavati više funkcionalnosti no što je potrebno**. Ovo svojstvo je posebno izdvojeno jer *hypervisor* predstavlja komponentu virtualizacijskog alata kojoj se potpuno vjeruje (engl. *trusted computing base, TCB*), i unošenjem bilo kakvih promjena riskira se otvaranje potencijalnih rupa koje bi napadač mogao iskoristiti. Iz tog razloga je promjene potrebno svesti na minimum i iscrpno testirati.

3.1.2. Ostala svojstva vezana uz sustave nadzora

Obzirom da kvaliteta sustava nadzora u velikoj mjeri ne ovisi samo o implementaciji *hypervisora*, već i o programskom rješenju sustava, bitno je spomenuti još neka svojstva koje bi ti sustavi trebali zadovoljiti. Kittel [46] spominje tri svojstva koje bi sustavi nadzora trebali ostvariti:

- pouzdanost,
- otpornost na "uplitanje"[3],
- minimalni utjecaj na *guest OS*.

Pouzdanost je svojstvo koje se očekuje od svih aplikacija na tržištu, a koje je najčešće prvo koje zakaže. Nitko ne vjeruje aplikacijama koje se stalno ruše. Primjerice, u slučaju aplikacija koje imaju veze sa sigurnošću, kao što je kriptiranje diska, može se dogoditi da aplikacije zakažu usred postupka, čime korisnike dovode u neugodnu situaciju da ne mogu odrediti koji je dio diska kriptiran. Na isti način, aplikacija koja pruža nadzor nad virtualnim strojevima mora imati mogućnost oporavka od jednostavnijih grešaka, ili mogućnost zaustavljanja nadziranog stroja dok se aplikacija u potpunosti ne oporavi.

Tamper resistance je svojstvo alata da se može oduprijeti pokušajima malverzacije, odnosno bilo kakvim pokušajima napadača da injektira lažne podatke u sustav za nadzor ili ga zaustavi. Zahvaljujući svojstvu izolacije iz poglavlja 3.1.1, *hypervisor* ne bi smio omogućiti ikakvu direktnu komunikaciju virtualnih strojeva, ili komunikaciju prema *host OS-u*. Zbog tog svojstva, napadač ne može zaustaviti sustav nadzora. No, injekcija lažnih podataka može predstavljati problem ukoliko se podaci ne izvlače iz pouzdanog izvora. Postoji nekoliko metoda dohvaćanja podataka, koji su opisani u

[2] *keep it simple stupid*
[3] *tamper resistance*

idućim poglavljima, ali također postoji i nekoliko metoda napada na određene metode dohvata. Napadi su opisani u poglavlju 3.4.

U virtualnom okruženju potrebno je ostvariti **minimalni utjecaj na *guest OS***, kako napadači ne bi posumnjali da ih netko promatra. Bilo kakva invazivna akcija sustava za nadzor u nekom trenutku može biti otkrivena, što znači da bi se napredniji zloćudni programi mogli prilagoditi prilikom detekcije. Osim invazivnih akcija, bitno je spomenuti i performanse sustava. Danas više nije uobičajena pojava da poslužitelju treba mnogo vremena kako bi odgovorio na upit, tako da bilo kakvo veće kašnjenje može biti znak postojanja nadzora u sustavu. Payne et al. [50] ovo svojstvo nazivaju ***mali utjecaj na performanse*** (engl. *small performance impact*). Kako bi se sustavi ubrzali, preporuča se razvoj sustava koji bi mogli djelovati u paraleliziranom okruženju.

3.1.3. Semantička praznina

Sustavi za nadzor, detekciju i prevenciju napada, koji su smješteni na računalu koje nadziru, imaju veliku prednost u odnosu na sustave nadzora u virtualnim okruženjima. Naime, ako se alat instalira na sustavu koji nadzire, on može vrlo jednostavno upotrijebiti druge alate dostupne na sustavu, kao što može iskoristiti i semantiku i API operacijskog sustava kako bi došao do željenih podataka. No, ta je prednost ujedno i veliki nedostatak ovakvih sustava, jer su poznate brojne metode zaobilaženja tih alata [43, 33] ako napadač ima administratorske ovlasti nad sustavom.

Jedno od mogućih rješenja jest premjestiti sustav nadzora izvan sustava koji se nadzire, za što virtualizacija predstavlja savršeno okruženje. No, u slučaju odvajanja nadzornog programa od semantike i API-ja operacijskog sustava, dolazi se do problema *semantičke praznine* (engl. *semantic gap*). Ovaj koncept objasnili su Chen i Noble [35], a radi se o problemu semantičke interpretacije podataka koje možemo dobiti iz *hypervisora*. Bez struktura operacijskog sustava ili *kernel* API-ja, parsiranje memorije (odnosno slike fizičke memorije) virtualnog stroja je praktički nemoguć zadatak — teško je ili nikako nije moguće odrediti gdje se nalaze bitne kontrolne strukture. Aplikacija samo vidi binarnu interpretaciju podataka, bez ikakve semantike. Situacija se dodatno komplicira, jer različite verzije jezgre operacijskog sustava smještaju identične strukture na druge memorijske lokacije, tako da čak ni u slučaju identične distribucije[4] operacijskog sustava, verzija jezgre može biti presudna.

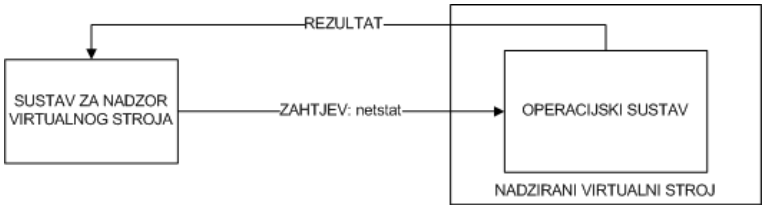
Osim memorije, moguće je dobiti i podatke o izvođenim instrukcijama na proce-

[4] ovaj pojam je usko vezan uz operacijski sustav Linux, ali se u ovom kontekstu odnosi i na različita izdanja operacijskog sustava Windows (npr. *Professional* ili *Home*)

soru, ali ako nije poznato u koje se registre smještaju koje vrijednosti prilikom sistemskih poziva (što ovisi o implementaciji operacijskog sustava), ponovo nije moguće jednoznačno odrediti semantiku. Pfoh et al. [51] u svom radu pobrojavaju tri pristupa kojima je moguće dobiti podatke u virtualiziranom okruženju, a opisani su u idućim poglavljima.

3.1.4. *In-band delivery*

Ukoliko se koriste podaci koje dostavlja operacijski sustav koji se nadzire, radi se o *in-band delivery* pristupu. Na 3.1 se može vidjeti primjer ovakvog pristupa, korištenjem naredbe *netstat*. Može se reći da ovaj pristup prati *klijent-poslužitelj* arhitekturu, gdje aplikacija za nadzor (klijent) šalje zahtjev internoj komponenti u sustavu koji se nadzire (poslužitelj). Time se, zapravo, ne premošćuje semantička praznina, već se zaobilazi.



Slika 3.1: *In-band* metoda dohvata

Ovaj pristup ima nekoliko nedostataka. Za početak, podaci koji se dohvaćaju ovise o komponenti koja se nalazi u sustavu koji se promatra. Napadač može ovladati sustavom koji se nadzire, te sve komponente unutar tog sustava mogu biti objekti malverzacija. Na taj način, svi podaci koji se dohvaćaju nakon napada mogu biti krivi, jer inherentno koriste iste podatke koje i operacijski sustav ima na raspolaganju. Drugi problem se pojavljuje ukoliko se virtualni stroj koji promatramo zaustavi[5] dok se obavlja neka logika u sustavu nadzora. Aplikacije koje se nalaze u sustavu koji se nadzire ne mogu biti izvršene dok je virtualni stroj zaustavljen. Ukoliko se prikupljaju podaci dok sustav radi, napadač može iskoristiti nekonzistentnosti u memoriji. Primjerice, izlistavanjem svih procesa, program *ps* iterira po cirkularnoj listi — napadač bi teoretski mogao smjestiti svoju strukturu procesa prije čvora koji je prethodno pročitan, čime se taj novi proces ne bi prikazao u ispisu. Konačno, izvršavanjem aplikacija na sustavu koji se nadzire mijenja se stanje sustava, što bi napadač mogao otkriti.

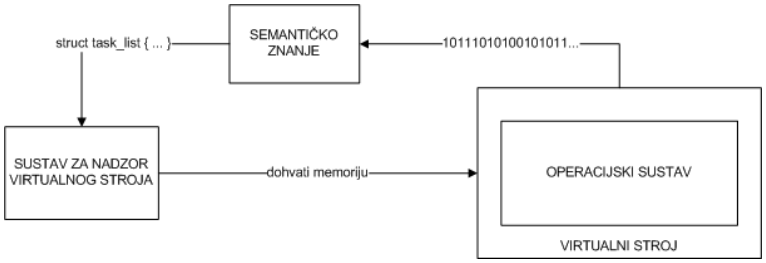
Ovakav pristup ne predstavlja nikakvu dodatnu vrijednost u odnosu na uobičajene sustave nadzora koji se nalaze na sustavima koje nadziru. Štoviše, ovaj pristup može

[5] suspendira

biti samo jednak, ili lošiji, od uobičajenih pristupa, jer može uvesti nekonzistentnosti u memoriji.

3.1.5. *Out-of-band delivery*

Korištenjem *out-of-band delivery* pristupa, semantičko znanje o sustavu koji se nadzire prenosi se preko eksterne funkcije. Drugim riječima, semantičko znanje se izvlači iz određenih sistemskih podataka prije nego što nadzor započne, te se parametri zaključivanja predaju funkciji kojom se dohvaćaju podaci prilikom nadzora. Ta funkcija je eksterna jer se ne nalazi na sustavu koji se promatra. Ovo je najčešće korišteni pristup pri nadzoru virtualnih strojeva.



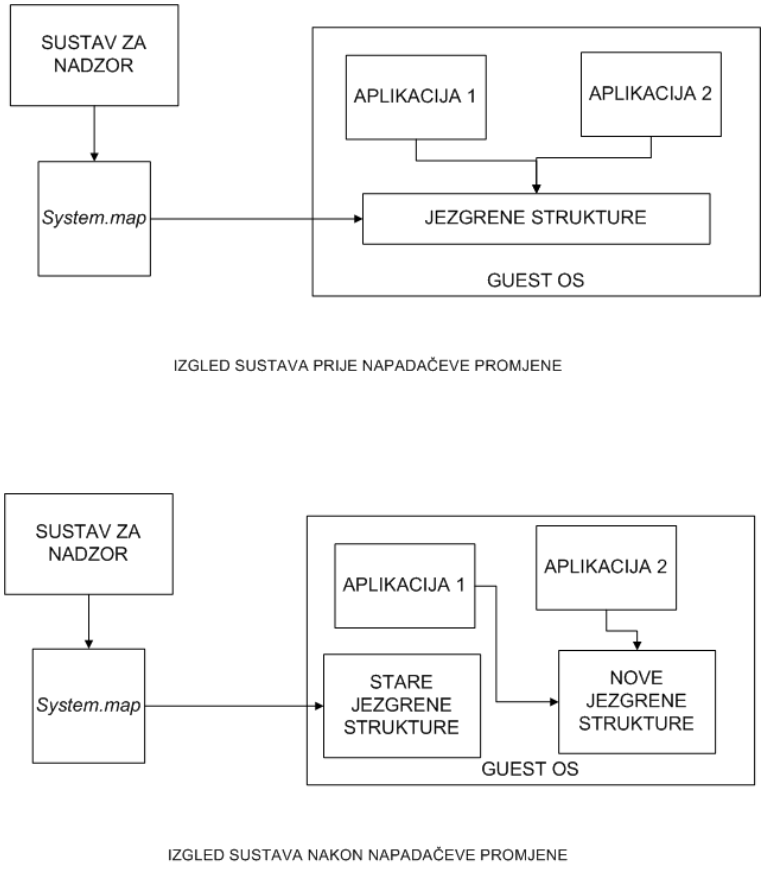
Slika 3.2: *Out-of-band* metoda dohvata

Na primjeru operacijskog sustava Linux, sistemski podaci iz kojih se može izvući semantičko znanje nalaze se u *System.map* datoteci, *debug* simbolima koji se mogu stvoriti prilikom prevođenja jezgre, i u nekim drugim datotekama. Ti podaci se daju funkciji koja prepoznaje memorijske lokacije ključnih sistemskih struktura u memoriji (npr., statičke[6] memorijske strukture), te se to ekstrahirano znanje koristi prilikom dohvaćanja podataka iz virtualnog stroja prilikom nadzora. Slikovni opis ovog procesa vidljiv je na 3.2. Kao i u *in-band delivery* pristupu, kako bi se pristupilo podacima u fizičkoj memoriji, potrebno je zaustaviti virtualni stroj kako bi memorija ostala konzistentna. No, u ovom slučaju, moguće je iz zaustavljenog virtualnog stroja dohvatiti podatke, jer se za dohvaćanje podataka koristi eksterna funkcija koja ne ovisi o radu virtualnog stroja. Velika prednost ovog pristupa je što se, za razliku od *in-band delivery* pristupa, ne generira nikakav dodatan memorijski "promet" koji nastaje prilikom pokretanja aplikacija na sustavu. Time se povećava razina skrivenosti sustava nadzora koji koristi ovakav pristup.

Kako se semantičko znanje ekstrahira korištenjem podataka koji su dostupni prije nadzora, može se pojaviti problem ukoliko se promijene podaci koji se koriste unu-

[6] pozicioniraju se uvijek na iste memorijske lokacije

tar virtualnog stroja. Naime, proces ekstrakcije semantičkog znanja odvija se samo jednom, i to prije početka nadzora. Ukoliko se podaci iz kojih je znanje ekstrahirano promijene, ili se doda neki novi podatak, napadač može izbjeći nadzor, jer sustav nadzora i dalje koristi staru sliku sustava, dok je napadač modificirao tu sliku. Izgled napada vidljiv je na slici 3.3. Specifični napadi su opisani u poglavlju 3.4. Uvijek postoji mogućnost ponovne ekstrakcije znanja, ali taj proces traje predugo da bi se provodio dok je virtualni stroj u normalnom radu.



Slika 3.3: Napad na *out-of-band* pristup

3.1.6. *Derivacija*

Treći pristup dohvata podataka iz nadziranog virtualnog stroja naziva se derivacija (engl. *derivation*), kako bi se naglasilo da se do podataka dolazi zaključivanjem o podacima korištenjem semantike *hardverske* arhitekture. U ovom kontekstu, *hardversku* arhitekturu predstavlja procesor, iako se može iskoristiti bilo koja druga komponenta koja može na neki način doprinijeti u semantici.

Očekivano ponašanje svih računalnih sustava je korištenje procesora (osim pri,

npr., prijenosu velikih količina podataka u memoriju ili iz memorije, pri čemu se koriste posebne *DMA*[7] jedinice), što znači da velika količina podataka prolazi kroz procesorske registre. Primjerice, pri sistemskim pozivima operacijskog sustava Linux, u procesorske registre se stavljaju argumenti sistemskih poziva, s tim da se u registru *eax* nalazi broj sistemskog poziva. Zatim se poziva instrukcija koja generira *softverski* prekid. Korištenjem derivacijskog pristupa, može se saznati koji su parametri bili predani kojem sistemskom pozivu, što gledano iz perspektive jednog sistemskog poziva ne znači mnogo. Ukoliko se za svaki proces prati koji su sistemski pozivi bili pozivani s kojim parametrima, moguće je rekonstruirati aktivnosti tih procesa. Primjer ovakve aplikacije je prikazan u radu [37].

Osim praćenja sistemskih poziva, moguće je pratiti određene sistemske registre, kao što je *CR3* registar. U taj se registar smješta memorijska lokacija direktorija memorijske stranice, koja je različita za svaki proces. Iskorištavanjem ove informacije, moguće je enumerirati sve procese na temelju različitih memorijskih lokacija. Mogu se pojaviti dvosmislenosti prilikom ponovne uporabe stranice, što se može izbjeći usporedbom sadržaja tih memorijskih lokacija i pripadnom logikom odlučivanja. Kako bi se odredile sve moguće primjene derivacijskog pristupa, potrebno je detaljno proučiti dokumentaciju procesora za koji se radi alat. U ovom slučaju je riječ o Intel x86 procesorima, za koje se dokumentacija može naći u [14].

Velika je prednost ovog pristupa što napadač ni na koji način ne može utjecati na tako nisku razinu, točnije procesor u izvođenju. Napadač ne može promijeniti podatke koji moraju proći kroz procesor kako bi se neki proces izvršio. Iako je ovaj pristup naizgled savršen, postoji veliko ograničenje na količinu informacija koja se može dobiti ovim pristupom. Osim dobivanja informacija, potrebna je velika stručnost u njihovoj interpretaciji. Npr., moguće je pobrojati sve procese korištenjem *CR3* registra, ali ih nije moguće dalje identificirati. Pfoh et al. [52] su u svom radu opisali kako se mogu izvući i interpretirati neki podaci korištenjem derivacijske metode. Također, nakon što se napiše alat koji koristi derivacijsku metodu za jednu procesorsku arhitekturu, da bi bio primjenjiv i za neki drugi procesor, potrebno je redizajnirati alat korištenjem dokumentacije o pripadnoj procesorkoj arhitekturi. Zbog navedena dva problema, korištenje isključivo ovog pristupa nije preporučljivo u praksi.

[7] *Direct Memory Access*

3.1.7. Kombinacija pristupa

Prethodna poglavlja su iznijela različite pristupe dobivanja podataka u njihovom "najsirovijem" obliku. Ti pristupi se mogu (štoviše, trebaju) kombinirati, kako bi se ostvario njihov puni potencijal. Na primjer, *in-band delivery* sam po sebi ne koristi mnogo, jer se umjesto njega mogao iskoristiti i neki od postojećih *host-based* sustava nadzora. Ukoliko se kombinira *in-band delivery* s *out-of-band delivery* pristupom, moguće je dohvatiti stanje sustava korištenjem oba pristupa, te zatim ta stanja usporediti ne bi li se otkrile nedosljednosti. U ovom slučaju, detekcija se oslanja na činjenicu da će se maliciozni *softver* pokušati sakriti, što bi uspješno zavaralo aplikaciju koja koristi *in-band delivery* pristup. No, sakrivanje ga ne štiti od *out-of-band delivery* pristupa. Kombiniranjem ova dva pristupa, moguće je izolirati aplikaciju koja se pokušava sakriti jednostavnom usporedbom listi.

Prilikom kombiniranja pristupa, potrebno je pomno paziti na cilj i arhitekturu aplikacije. Ako je portabilnost po procesorskim arhitekturama prioritet, ne bi se smio koristiti derivacijski pristup. No, ukoliko je prioritet portabilnost u pogledu operacijskog sustava koji se nadzire, onda se svakako preporuča derivacijski pristup kao jedna od komponenti sustava nadzora. Detaljna tablica s usporedbom svojstava pojedinih pristupa se nalazi u [51].

3.1.8. Apel proizvođačima virtualizacijskih alata

Virtualizacijskih alata na tržištu ima mnogo, iako se zapravo bitka vodi između nekoliko najpoznatijih — *VMWare*, *Oracle VirtualBox*, *Xen*, te *QEMU/KVM*. Obzirom na rast performansi računala, odgovor na pitanje koji je od navedenih alata najbolji, postao je stvar osobnog ukusa. Svaki proizvođač pruža kvalitetniju uslugu u nekom aspektu, ali bit onoga što moraju pružiti je ista — kvalitetna virtualizacija procesora i pripadnih komponenti.

Zanimljivo je primijetiti kako nitko osim *VMWare-a* i *Xen-a* nije obratio veću pozornost na integraciju mogućnosti nadzora virtualnih strojeva. *VMWare* svoju tehnologiju naziva *VMSafe* [25], dok je *Xen* jedno kratko vrijeme bio uključen u razvoj *XenAccess* biblioteke, sad preimenovane u *VMI Tools* [23].

Ovim kratkim poglavljem ističe se sve više rastuća potreba za jedinstvenim (moguće je čak reći standardiziranim) API-jem za nadzor virtualnih strojeva. Kako bi se sustavi nadzora virtualnih strojeva mogli ostvariti, mora se mijenjati kod *hypervisora*. Iako su pojedinci, koji mijenjaju kod, obično vrlo sposobni, to i dalje prestavlja potencijalni rizik jer nije dovoljno testirano. Osim toga, svatko će implementirati ono što

mu je potrebno, umjesto da se implementira skup funkcija koji mora biti podržan kako bi se mogle razvijati aplikacije za nadzor neovisno o virtualizacijskom alatu.

Ako se nadzor virtualnih strojeva nastavi razvijati ovim tempom, u vrlo skoroj budućnosti će na Internetu postojati veliki broj testnih aplikacija, koje sve, u suštini, rade dobro, ali uprkos tome ne mogu imati konkretnu primjenjivost u produkcijskom okruženju jer previše utječu na performanse, ili ne provjeravaju neki dio sustava koji neka organizacija želi imati pod nadzorom. Ovo je velika šteta, jer ti sustavi jako obećavaju.

3.2. Arhitekture sustava nadzora

Prilikom izgradnje sustava nadzora, veliku pažnju treba posvetiti dizajnu, kako bi sustav bilo lako održavati i unapređivati. Napisano je nekoliko kratkih savjeta za izgradnju sustava, pregled mogućih grešaka, te opis jednog idealnog sustava za nadzor.

3.2.1. Monolitni sustavi

Monolitni sustavi su cjeloviti sustavi, koje nije potrebno proširivati, već imaju točno određenu funkcionalnost koju trebaju izvršiti. Primjer ovakvih sustava su aplikacije za rad s bazama podataka, za vođenje evidencija, i ostale "jednostavnije" aplikacije čija se proširenja oslanjaju samo na programerski tim koji je osmislio alat. Ovakvi sustavi najčešće ne podržavaju nikakvo dodatno proširivanje od strane korisnika, već se očekuje da korisnik samo koristi aplikaciju.

Iako je sustav monolitan, to ne znači da on ne slijedi, npr., objektno-orijentiranu paradigmu, ili neku drugu paradigmu dobre programerske prakse. Ovakvi sustavi se najčešće koriste kao izolirane komponente, ili kao aplikacije koje demonstriraju rad nekog koncepta. Također, ukoliko je potrebno jednostavno i brzo rješenje nekog problema, u početku je najvažnije koncentrirati se na implementaciju funkcionalnosti koja rješava taj problem, a ostala proširenja se lako mogu dodati kasnije.

3.2.2. Modularni sustavi

Područje nadzora računalnih sustava, na žalost, ne trpi "nedostatke" monolitne arhitekture, jer svaki korisnik sustava može trebati neku drugu funkcionalnost, ili drugačiji pogled na sustav. Slijedeći misaoni proces korisnika, u ovom su području najkorisniji modularni sustavi. Modularni sustavi imaju jednu centralnu komponentu koja pred-

stavlja programsku logiku koja povezuje ostale module. Moduli mogu (i trebaju) imati različite funkcionalnosti, a kako bi se ta funkcionalnost mogla još dodatno proširivati, koriste se skripte. Bitno je primijetiti da pojedini moduli po svojoj funkcionalnosti ipak trebaju biti monolitni, s jasno odvojenim zadaćama.

Izvrstan primjer modularnog sustava je **Nagios** [16]. Opisan je kao industrijski standard u nadzoru računalnih sustava, jer pruža kvalitetnu i jednostavno proširivu uslugu. Na jednom računalu instalira se *Nagios server*, koji služi kao centralna komponenta i raspoređivač aktivnosti (provjera). Provjere se dijele na lokalne, udaljene, te lokalne na udaljenim računalima. Lokalne i udaljene provjere se izvršavaju na poslužitelju na kojem je instaliran *Nagios server*. Za izvršavanje lokalnih provjera na udaljenim računalima, potrebno je na udaljenim računalima instalirati *Nagios Remote Plugin Executor*, koji funkcionira kao modul koji na udaljenom računalu izvršava skripte. Sve provjere se pišu u obliku dodataka (engl. *plugins*), koji su u suštini skripte i može ih napisati bilo koji korisnik. Skripte se pišu po određenim pravilima kako bi *Nagios server* mogao interpretirati rezultate skripti.

Još jedan dobar prijedlog modularnog sustava predstavlja sustav **Livewire**, opisan u poglavlju 4.1. Iako je u tom poglavlju podrobnije opisana arhitektura sustava, ovdje će samo ukratko biti spomenuta. Sustav je podijeljen na tri dijela:

- sučelje prema *hypervisoru*,
- biblioteku funkcija specifičnu za operacijski sustav koji se nadzire,
- modul koji izvršava sigurnosne politike.

Sigurnosne politike se pišu u obliku modula (skripti) koje modul za izvršavanje pokreće. Moduli (skripte) politika su detaljno objašnjeni kasnije.

Zadnji primjer arhitekture sustava je uzet iz rada Kittel [46]. Ovaj sustav je također podrobnije opisan kasnije u poglavlju 4.6, ali je bitno ukratko izdvojiti da koristi jednu centralnu komponentu, koja dohvaća rezultate iz pojedinih modula koji nadziru senzore. U ovom slučaju, senzori su "korisničke" skripte koje opisuju koji dio sustava treba nadzirati, a moduli kontroliraju te skripte.

3.2.3. Idealni sustav nadzora

Jedinstvena arhitektura za sustav nadzora virtualnih strojeva ne postoji, ali je ovdje predstavljena jedna od mogućih prema kojoj bi se trebalo težiti.

Kao što je već rečeno, arhitektura sustava bi trebala biti modularna, što znači da bi, u svakom slučaju, postojala barem jedna centralizirana komponenta. Ta komponenta

bi se nalazila na sustavu s kojeg promatramo (*host OS*). Na taj način, ne bi postojalo nikakvo usko grlo uslijed podataka koji pristižu s mreže[8]. Isto tako, nije potrebno prosljeđivati podatke u neki drugi virtualni stroj, nego se direktno može pristupiti svim podacima iz *hypervisora*.

Za početak, sustav nadzora ne bi trebao ovisiti o nekom specifičnom *hypervisoru*, ali to bi bilo moguće jedino u slučaju da proizvođači virtualizacijskih alata ponude standardizirani API. U tom slučaju, arhitektura procesora također ne bi bila bitna jer bi sve promjene unosili tvorci virtualizacijsih alata. Taj API bi bio u obliku dijeljene biblioteke, čime bi se izbjegao proces prevođenja *hypervisora* prilikom stvaranja sustava nadzora. Postojala bi baza s detaljima operacijskih sustava koji bi pomogli pri analizi memorije, kako bi se na jedinstven način moglo pristupati memoriji neovisno od operacijskog sustava koji koristi tu memoriju. Na isti način, trebala bi postojati i baza pravila korištenja procesora (npr., konvencija sistemskih poziva).

Sva ova pravila trebala bi se moći uprogramirati u module čija je svrha nadziranje pojedinih dijelova sustava, a mogli bi ih programirati korisnici. Osim toga, ti moduli bi se mogli ubaciti dok sustav radi, slično kao što se ubacuju jezgreni moduli u jezgru operacijskog sustava Linux prilikom izvođenja da se doda nova funkcionalnost. Konačno, skripte bi bile napisane u nekom od postojećih skriptnih jezika ili bi trebalo osmisliti poseban skriptni jezik za ovu namjenu. Ako bi se radio novi skriptni jezik, bilo bi potrebno napraviti i interpreter, kako bi se skripte mogle protumačiti u sustavu. Skripte bi funkcionirale kao senzori, odnosno na promjenu ili pojavu događaja bi obavijestile nadležni modul, a on bi promjenu stanja proslijedio u centralnu komponentu.

Po uzoru na sustav *Nagios*, bilo bi prikladno imati i centralizirano web sučelje na kojem bi se mogle vidjeti sve promjene kako nastaju. Svi detalji o događaju bi mogli biti proslijeđeni sustavu zaključivanja, koji bi se temeljio na nekoj od metoda strojnog učenja, čime bi se dobio uvid radi li se trenutno o napadu ili normalnom ponašanju sustava.

3.3. Primjene nadzora virtualnih strojeva

Obzirom da je ovo područje znanosti mlado, sve primjene nadzora virtualnih strojeva zasigurno još nisu otkrivene, ali neke od najvažnijih su (ili će možda postati):

- neprimjetno praćenje aktivnosti korisnika
- automatizirana analiza zloćudnih aplikacija

[8] svi mrežni podaci obavezno moraju proći *host OS* da dođu do *guest OS-a*

– računalna forenzika i sustavi za privlačenje napadača

– automatizirano zaključivanje o vrsti napada

– neprimjetni sustavi detekcije i prevencije napada

3.3.1. Neprimjetno praćenje aktivnosti korisnika

Glavna primjena nadzora virtualnih strojeva je svakako neprimjetno praćenje aktivnosti korisnika. Iz ove kategorije mogu se izvesti druge primjene, no u konačnici se uvijek postavlja pitanje što to korisnik radi (ili je radio) da je vrijedno pažnje. Kako bi se uspješno pratila aktivnost korisnika ili sustava, potrebno je definirati nadzorne točke u sustavu, odnosno definirati resurse koje je potrebno nadzirati. U poglavlju 3.1.1 već je bilo govora o svojstvu inspekcije i o stanjima virtualnog stroja, te kako kombinacija *softverskog* i *hardverskog* stanja čini cijelo stanje sustava.

Za nadziranje promjenjive memorije, potrebno je koristiti analizator memorije kao što su InSight[13] ili Volatility Framework[27]. Za analizu permanentne memorije obično je dovoljno povezati[9] particiju virtualnog stroja na sustav s kojeg nadziremo. Nadzor stanja procesora moguće je ostvariti iz *hypervisora* periodičkim zaustavljanjem procesora i ispisa stanja registara, ili korištenjem nekog alata koji može zaustaviti procesor nakon određenih instrukcija (korištenjem nadzornika sistemskih poziva kao što je Nitro[17]). Stanja ostalih fizičkih komponenti treba "loviti" preko procesora, jer će se oni za pojedine komponente izvršavati na procesoru. Kako bi se uspostavio potpuni sustav nadzora, potrebno je pratiti sve komponente. U ovom radu, kako će biti opisano poslije, implementirana je provjera memorije i procesa koji se trenutno izvode.

3.3.2. Automatizirana analiza zloćudnih aplikacija

Dinamička analiza zloćudnih programa postaje značajno područje obzirom na količinu godišnje stvorenih zloćudnih programa. Pod dinamičkom analizom misli se na promatranje aplikacija za vrijeme njihovog izvođenja. Zloćudni programi su toliko uznapredovali da ponekad nije moguće statički analizirati aplikaciju, nego je potrebno pokrenuti aplikaciju kako bi se smjestila u memoriju i pokazala svoju pravu namjeru. Kako sav *softver* koji se nalazi u istom okruženju kao i zloćudne aplikacije koje analiziramo može biti prevaren, potrebno je ostvariti nadzor nad zloćudnim aplikacijama "izvana", odnosno napraviti takvu okolinu u kojoj prijevara sustava za nadzor nije moguća. Primjeri ovakvih radova su Jiang et al. [44] i Yin et al. [64].

[9] *mount*

3.3.3. Računalna forenzika i sustavi za privlačenje napadača

Još jedna od primjena nadzora virtualnih strojeva jest u područjima računalne forenzike "živih" strojeva i sustava za privlačenje napadača. Iz znanstvene perspektive, cilj je odrediti što to napadači čine (ili su učinili), a da je uzrokovalo određenu štetu ili događaj na računalu koje se promatra. Štoviše, promatrajući "žive" sustave, promatrači mogu biti u centru događanja dok se napad odvija. Na taj način može se neusporedivo više naučiti u odnosu na promatranje zapisničkih[10] datoteka, jer je vidljivo kako se sustav mijenja pod utjecajem napadača. Napadač ne može znati da ga netko promatra, i veća je vjerojatnost da se "opusti" prilikom napada, što povećava kvalitetu informacija jer će ostavljeni trag biti veći. Asrigo et al. [30] u svom radu objašnjavaju korištenje senzora za nadzor sustava za privlačenje napadača u virtualnom okruženju.

3.3.4. Automatizirano zaključivanje o vrsti napada

Iako je strojno učenje jedno od područja računarske znanosti koje se intenzivno istražuje, i dalje je vezano za prespecifične probleme. Drugim riječima, kad se napravi aplikacija za strojno učenje za jedan problem, mala je vjerojatnost da će se ista aplikacija moći primijeniti na neki drugi problem, osim ako se algoritam ne promijeni. Bez obzira na to, aplikacije koje su dobro napisane za svoje domene obećavaju kvalitetne rezultate. Na isti način, mogla bi se osmisliti aplikacija koja automatski klasificira napade koji su se izvršili na temelju prikupljenih podataka. Najveći izazov pri izgradnji ovakvih sustava jest jednoznačno odrediti skup podataka koji definiraju napad. U tu svrhu može se iskoristiti neka vrsta taksonomije. Jedna od takvih taksonomija, koja obećava, jest SISTER[11] koju trenutno razvija Kristian Skračić, ali čija publikacija u ovom trenutku nije dostupna. Automatizirano zaključivanje o vrsti napada moglo bi predstavljati dodatak na bilo koju već navedenu aplikaciju. Ono bi istražiteljima bilo od pomoći u objektivnom zaključivanju o vrstama napada, lakšoj izgradnji baza podataka o napadima i slično.

3.3.5. Neprimjetni sustavi detekcije i prevencije napada

Sustavi detekcije i prevencije napada su prilično rašireni, posebice po organizacijama kojima je stalo do zaštite podataka. Veliki nedostatak ovakvih sustava je što se oni najčešće nalaze na napadnutim računalima, što znači da takve sustave napadač može

[10] *log file*

[11] *A Set of Information Security Taxonomies for Education and Research*

ugasiti ili krivo konfigurirati kako bi se sakrio na sustavu. Prednost koju nadzor virtualnih strojeva nudi je da se sustav može nalaziti u potpuno odvojenom virtualnom stroju ili u operacijskom sustavu gdje se nalazi *hypervisor*, čime se efektivno sustav nadzora izolira. Na taj način, napadač ne može zaobići niti promijeniti sustav nadzora. Trenutni sustavi detekcije i prevencije napada za virtualne strojeve još nisu jako rašireni jer su prespori, ne promatraju sva stanja operacijskog sustava koja bi trebalo promatrati kako bi se ostvario kvalitetan nadzor ili sadrže određene propuste čijim se iskorištavanjem može izbjeći nadzor.

3.4. Mogući napadi

Unatoč brojnim prednostima nadzora virtualnih strojeva, postoje brojni propusti koje je moguće iskoristiti, ukoliko se sustav nadzora ne implementira pravilno. Sve dok Pfoh et al. [51] nisu formalno popisali sve moguće načine nadzora virtualnih strojeva, bilo je teško dobiti opću ideju stvaranja ovakvog sustava, a još ga je trebalo učiniti otpornim na malverzacije.

Bahram et al. [31] u svom radu opisuju tri vrste napada pod zajedničkim imenom **direktna manipulacija jezgrenim strukturama** (engl. *direct kernel structure manipulation*). U suštini, ovakav napad koristi mogućnost manipulacije jezgrom promjenom vrijednosti ili položaja struktura u memoriji. Na taj način, moguće je zaobići gotovo sve postojeće sustave za nadzor virtualnih strojeva. Vrste napada koje opisuju su:

- manipulacija sintaksom,
- manipulacija semantikom,
- kombinacija napada.

Manipulacija sintaksom (engl. *syntax-based manipulation*) odnosi se na dodavanje ili uklanjanje legitimnih polja u jezgrenim strukturama. Iako su jezgrene strukture pune informacija, ponekad postoje polja koja se ne koriste (npr., lista procesa u operacijskom sustavu Linux je dvostruko povezana lista; pokazivači koji idu od zadnjih elemenata prema prvima se ne koriste). Dodavanje polja s informacijama se u pravilu ne koristi, jer se zaključivanje izvodi na temelju manjeg skupa informacija. Ukoliko se informacija u poljima promijeni, ili se polja ne koriste, alat za nadzor zaključak o stanju sustava temelji na manjem broju podataka od pretpostavljenih. Obzirom da pravilo indukcije nije potpuno, moguće je natjerati alat da pogriješi. No, ovakav napad se rijetko može izvesti u praksi, a ionako ne može poslužiti skrivanju napada.

Manipulacija semantikom (engl. *semantic-based manipulation*) predstavlja ozbiljniji napad od prethodnog. Ovaj napad je moguć zbog toga što semantika polja u jezgrenim strukturama ovisi o tipu polja prilikom izvršavanja. Npr., običan *integer* je moguće prenamijeniti (engl. *cast*) u pokazivač na znakovni niz, ili pokazivač na znakovni niz prenamijeniti u pokazivač na funkciju. Ovo otvara mnoge mogućnosti malverzacija ukoliko se ne provjerava sadržaj jezgrenih struktura. Moguće je preusmeriti pokazivače jezgrenih struktura na stare kopije struktura u memoriji (engl. *shadow copies*), dok jezgra u stvarnosti koristi nove strukture koje su smještene na drugim memorijskim lokacijama. Sustav za nadzor, kao ni operacijski sustav, ne vide da nešto nije u redu s jezgrenim strukturama koje promatraju, dok napadač ima potpunu vlast nad sustavom. Ovaj napad je u dotičnom radu objašnjen detaljnije i prikazan u izvedbi.

Zadnja napadačeva mogućnost jest **kombinacija** napada. Gore opisani napadi nisu međusobno isključivi, a njihovim kombiniranjem može se dobiti poguban napad na sustav. Primjer jednog složenijeg napada mogao bi započeti preusmjeravanjem svih kritičnih podataka, kojima bi se onda zamijenili sadržaji pojedinih elemenata (semantički napad). Tako promijenjenim podacima, još bi se uklonilo ili dodalo nekoliko polja kako bi se dodatno zbunilo sustav za nadzor (sintaksni napad). Sustav bi prvo morao pravilno zaključiti o kojim se strukturama radi boreći se protiv sintaksnog napada, nakon čega bi slijedio dugotrajan i resursno zahtjevan proces semantičke interpretacije i praćenja pokazivača. Nakon što bi sustav nadzora raspleo cijelu mrežu, došao bi do starih podataka, koji su već na početku bili preusmjereni.

Promatrajući gornje napade, autori u [31] govore o tri različita pogleda na sustav. Postoje:

- vanjsko stanje,
- unutarnje stanje,
- stvarno stanje.

Vanjsko stanje je skup elemenata koji je vidljiv sustavu nadzora izvana. Unutarnje stanje je skup elemenata koji je vidljiv aplikacijama unutar virtualnog stroja, a stvarno stanje je stanje sustava kakvo ono uistinu jest, odnosno kako ga koristi operacijski sustav. U poglavlju 3.1, bilo je govora o različitim pristupima dohvaćanja podataka. Da bi se ovakvi napadi potencijalno izbjegli, nužno je koristiti kombinaciju pristupa dohvaćanja podataka. Također, dobra praksa jest zaštititi integritet jezgre (bar statičkih struktura) kako se ne bi mijenjao njihov položaj ni sadržaj, čime bi se mogućnost napada uvelike smanjila. Međutim, svaka preinaka jezgre može prouzrokovati promjenu kontrolne sume za provjeru integriteta.

Osim potonjeg rada, Srivastava et al. [59] navode još nekoliko napada na XenAccess biblioteku [50], koji se mogu poopćiti i na druge sustave koji primjenjuju slične pretpostavke na sustav nadzora. Ti napadi su djelomično obuhvaćeni gornjom podjelom, ali se uvode još dvije važne kategorije:

– napadi okoline,

– vremenski napadi.

Napadi okoline (engl. *environment attacks*) se temelje na stvaranju uvjeta (okoline) za koje sustav nadzora nije pretpostavio da bi mogli postojati. Primjerice, ukoliko sustav nadzora promatra direktorij s ključnim sistemskim programima poput *ps* i *netstat*, moguće je stvoriti novi privremeni disk smješten u memoriji računala (engl. *ramdisk*), na koji se smjeste "ispravljene" verzije sistemskih programa. Zatim se modificira varijabla okoline *PATH*, u kojoj se nalaze direktoriji u kojima se nalaze programi, tako da se na prvo mjesto smjesti putanja do novih programa. S obzirom da se ta varijabla okoline slijedno pretražuje, umjesto uobičajenog programa *ps*, izvršit će se nova verzija programa.

Vremenski napadi (engl. *timing attacks*) iskorištavaju jezgreni *buffering*[12] prije nego se podaci zapišu na disk. Ukoliko se napravi neka kratkotrajna promjena na disku, primjerice stvori i obriše datoteka, nema potrebe da se ta promjena zapisuje na disk. Tako je moguće izbjeći detekciju korištenja nekih alata. Autori [59] spominju instalaciju programa koji dodaje[13] novi disk bez zapisivanja u */etc/mtab* datoteku. Ako se između instalacije programa, njegovog korištenja i brisanja ne dogodi sinkronizacija diska i jezgrenog *buffera*, moguće je sakriti izvođenje tog alata.

[12] nagomilavanje podataka

[13] *mount*

4. Slični radovi

U ovom poglavlju naveden je kratak pregled radova koji se bave tematikom nadzora aktivnosti u virtualiziranim operacijskim sustavima. Zainteresiranim čitateljima se preporučuje da pročitaju barem te radove, kako bi dobili osnovni pregled područja i upoznali se s najnovijim tehnologijama u ovom području.

4.1. Livewire

Garfinkel i Rosenblum [40] su bili prvi znanstvenici koji su se dotakli područja nadzora virtualnih strojeva na način na koji se to područje i danas promatra. U radu su opisali tri glavna svojstva koje svaki *hypervisor* mora zadovoljiti, što je navedeno ranije u poglavlju 3.1. Osim opisa zahtjeva na tehnologiju, izgradili su i sustav za detekciju napada u virtualnim strojevima zvan *Livewire*.

Kako bi sustav bilo moguće izgraditi, unijeli su promjene u *hypervisor*. Unesene promjene čine skup naredbi koje se mogu izvršiti nad *hypervisorom*. Te su naredbe podijeljene u tri kategorije:

Inspekcijske naredbe koje služe za dohvaćanje stanja *hypervisora*, kao što su procesorski registri,

Naredbe nadzora koje se izvršavaju u slučaju točno određenog događaja (engl. *event*),

Administrativne naredbe koje služe za pokretanje i zaustavljanje *hypervisora*.

Sustav *Livewire* je podijeljen na tri dijela: sučelje prema *hypervisoru*, biblioteku sučelja operacijskog sustava i provoditelj pravila. **Sučelje prema *hypervisoru*** služi za kontrolu nad virtualnim strojevima koji se izvode.

Biblioteka sučelja operacijskog sustava služi za korištenje poznatih "kontrolnih točaka" unutar operacijskog sustava (kao što je smještaj struktura u memoriji, korišteni datotečni podsustav i sl.) kako bi se sustav mogao nadzirati. Točnije, korištenjem

funkcija biblioteke, moguće je dohvatiti *in-band* i *out-of-band* informacije iz nadziranog virtualnog stroja.

Provoditelj pravila je dio programske logike koji služi za provođenje definiranih pravila, odnosno utvrđivanje poštuju li se postavljena pravila. Pravila se pišu u obliku modula korištenjem *API-ja* kojeg pruža *policy framework*. Moduli se dijele prema funkcionalnosti, te obično provode *detekciju "laganja"*, *provjeru integriteta*, *otkrivanje "potpisa"*[1] *napada*, *zaštitu memorije* i mnoge druge funkcije.

Od vremena nastanka ovog rada razvili su se mnogi sofisticirani napadi, koji su već bili spomenuti u poglavlju 3.4, ali nadogradnjom sustava s modernijim alatima za dohvat informacija i promjenom programske logike, ovaj sustav bi mogao ponovo zaživjeti.

4.2. LibVMI, VMI Tools

LibVMI nije sustav nadzora, već biblioteka funkcija koja je prošla niz imena. U početku se zvala *XenAccess*, i bila je objašnjena u radovima Payne et al. [50] i Payne [49]. Nakon verzije 0.5, promijenila je ime u *LibVMI*, a sad se zove *VMI Tools* i može se naći na adresi [23].

U početku je biblioteka bila napisana isključivo za Xen *hypervisor* [8], i koristila je činjenicu da Xen ima privilegiranu *domenu* u kojoj se može pokrenuti virtualni stroj koji može nadzirati aktivnosti ostalih virtualnih strojeva. Danas postoji i dio biblioteke koji je posvećen KVM *hypervisoru*. Ova biblioteka prvenstveno je usmjerena na analizu memorije, a može se iskoristiti i kao jedna od komponenti u izgradnji sustava nadzora. XenAccess je pružao dva pogleda na memoriju sustava koji se nadzire. Korištenjem *API-ja* biblioteke, moguće je dobiti "sirovu" sliku memorije, bez ikakve interpretacije, ili je moguće parsirati *System.map* datoteku operacijskog sustava Linux kako bi se uvela viša razina semantike.

Ukoliko se primjeni parsiranje *System.map* datoteke, zbog pretpostavke *API-ja* da se jezgrene strukture nalaze na fiksnim lokacijama, analizu memorije je moguće izbjeći nekim od napada opisanima u poglavlju 3.4. Osim napada manipulacijom jezgrenih struktura i objekata, od kojih "pate" skoro svi sustavi nadzora, XenAccess je bilo moguće izbjeći korištenjem vremenskih napada i napada okoline. Jesu li ti napadi i dalje mogući, nije provjereno.

Ako se *System.map* datoteka ne parsira, ova biblioteka ima skoro istu funkcional-

[1] *signature detection*

nost kao i čitanje iz datoteke fizičke memorije virtualnog stroja, što ne znači mnogo jer ne postoji interpretacija tih podataka.

4.3. InSight

Bilo kakav pokušaj nadzora memorije virtualnog stroja mora započeti s nekim analizatorom memorije. InSight [13, 39] je alat za analizu memorijskih slika kojim se pokušava premostiti semantička praznina. Želi li se memorijska aktivnost pravilno pratiti, potrebno je pravilno interpretirati semantiku struktura podataka u memoriji. Zahvaljujući parsiranju izvornog koda operacijskog sustava, ovaj alat može dohvatiti sve jezgrene objekte iz memorije. Trenutno radi na x86 i AMD64 procesorskim platformama. Alat trenutno podržava jedino operacijski sustav Linux, zbog svoje prirode otvorenog koda, iako se u bliskoj budućnosti očekuje podrška i za operacijske sustave Windows.

Originalno je alat bio namijenjen analizi statičkih memorijskih datoteka[2], ali nedugo je počeo podržavati analizu fizičke memorije zaustavljenih virtualnih strojeva, u slučaju kad je memorija takvih virtualnih strojeva dostupna kroz datoteku. U prijašnjim poglavljima objašnjeno je da virtualni stroj mora biti zaustavljen za pravilnu memorijsku analizu kako bi memorija ostala u konzistentnom stanju. Po definiciji, InSight se koristi za *out-of-band delivery* pristup, jer se mogu dohvatiti sistemske strukture u memoriji. Osim nadzora virtualnih strojeva, ovaj alat se može primijeniti i za analizu zloćudnih programa, uklanjanje grešaka iz jezgre i digitalnu forenzičku analizu.

Parsiranjem izvornog koda jezgre, InSight gradi bazu podataka o tipovima i vrstama struktura, kako bi mogao pravilno dohvaćati podatke iz memorije. Proces parsiranja se provodi samo jednom, i to prije pokretanja nadzora memorije. Kako bi jednoznačno utvrdio memorijske lokacije objekata, InSight prilikom parsiranja zahtijeva pristup *System.map* datoteci, debug simbolima i zaglavljima jezgre. Debug simboli su u normalnom radu zapravo nepotrebni, a predstavljaju dodatni opis pojedinih struktura i podataka koji se prevode u programu prevoditelju. Za generiranje debug simbola, potrebno je prevesti jezgru s posebnim zastavicama predanim programu prevoditelju. Ovaj proces je dugotrajan i zauzima priličnu količinu prostora na disku, ali je to potrebno učiniti samo jednom.

InSight ima dva načina rada. Ukoliko se pokrene kao *daemon*[3], onda očekuje da ga drugi programi pozivaju. Moguće je pokrenuti alat u interaktivnom modu, u kojem

[2] *memory dump*

[3] program u pozadini

se pojavljuje slično konzolno sučelje kao za Python interpreter. U tom načinu rada moguće je ispitati svu funkcionalnost alata "u živo". Osim ova dva načina rada, InSight u sebi sadrži i JavaScript interpreter koji pokreće skripte pisane JavaScriptom kako bi se automatizirale određene aktivnosti (poput dohvata trenutne liste procesa).

4.4. Nitro

Nitro [17, 53] je alat za *hardversko* praćenje sistemskih poziva implementiran u KVM *hypervisoru*. Praćenje sistemskih poziva omogućava dobar uvid u stanje izvršavanja procesa, čime bi se mogla detektirati maliciozna aktivnost. Praćenje sistemskih poziva kao pristup detekciji napada ili zloćudnih aktivnosti nije novost [42, 47], pri čemu je korištenje strojnog učenja za učenje zloćudnih nizova sistemskih poziva davalo dobre rezultate [56], što pokazuje korisnost ovog pristupa.

Zahvaljujući položaju alata Nitro u odnosu na virtualni stroj, svojstva virtualizacije mu omogućuju potpunu izoliranost od potencijalnih napada i potpunu skrivenost od sustava kojeg nadzire. Osim toga, Nitro implementira praćenje za sva tri načina pozivanja sistemskih poziva Intel x86 arhitekture, te dokazano funkcionira za operacijske sustave Windows i Linux. Dva važna svojstva, koja Nitro čine toliko korisnim, su *portabilnost* i *sposobnost odupiranja napadima*. **Portabilnost** se odnosi na jednostavnu primjenjivost na bilo koji operacijski sustav koji se pokušava nadzirati, jer se alat vrlo lako prilagođava za nadziranje drugog operacijskog sustava. **Sposobnost odupiranja napadima** inherentno slijedi iz činjenice da je alat odvojen od virtualnog stroja kojeg nadzire. Nitro je primjer alata koji omogućuje korištenje derivacijskog pristupa spominjanog u poglavlju 3.1.6.

Postoje dva načina rada: praćenje i nadzor sistemskih poziva. Upotrebom **praćenja sistemskih poziva**, Nitro je moguće konfigurirati na način da, nakon svakog uhvaćenog prekida zaustavi virtualni stroj kako bi se omogućio daljnji nadzor virtualnog stroja. Ovaj način rada se naziva *singlestepping*. Rezultati ovakvog načina rada su broj sistemskog poziva koji se trenutno izvršava i podaci o memorijskoj stranici procesa koji izvršava trenutni poziv. Da bi se dobili ikakvi rezultati, ako se koristi **nadzor sistemskih poziva**, potrebno je zadati pravila na koja Nitro reagira. Točnije, potrebno je pravilima mapirati točne vrijednosti u registre kako bi se aktivirale kontrolne točke kad registri u izvođenju poprime te vrijednosti. Rezultat ovog načina rada jest sadržaj svih registara.

Nitro omogućava praćenje sistemskih poziva za sve tri Intelove konvencije, navedene u poglavlju 2.3. Nitro ne može raditi na AMD-ovim procesorima, ali se to

proširenje planira u budućnosti.

Ukoliko se koriste sistemski pozivi temeljeni na **prekidima**, x86 arhitektura koristi *IDT*[4]. S obzirom da Intelova virtualizacijska proširenja ne omogućuju hvatanje korisničkih prekida (prekidi s prekidnim vratima većim od 31), nije moguće uloviti programske prekide koji se koriste za sistemske pozive (na operacijskom sustavu Linux su po običaju na broju 128, ili heksadecimalno 0x80). Kako bi se omogućilo hvatanje tih prekida, *IDTR* se namjesti tako da pokazuje na zadnju vrijednost u *IDT-u*, a budući se ta vrijednost dodaje na baznu adresu *IDT-a*, *IDTR* pokušava adresirati dio memorije koji ne bi smio, čime se generira opća greška zaštite (engl. *general protection fault*). Kako reakcija na pojavu ove greške može biti programirana u kontrolnu strukturu virtualnog stroja (engl. *Virtual Machine Control Structure, VMCS*), može se odrediti da pojava ove greške uzrokuje vraćanje u *hypervisor* korištenjem naredbe `VMExit`. Nakon ulaska u *hypervisor*, potrebno je samo provjeriti je li greška bila uzrokovana "normalnim" putem ili se dogodio sistemski poziv. Ako se dogodio sistemski poziv, moguće je poduzeti mjere praćenja i ispisa sadržaja registara.

Sistemski pozivi temeljeni na **SYSCALL naredbi** mogu se pratiti na sličan način. Naime, ako se koriste ovakvi sistemski pozivi, postoji registar za omogućavanje dodatnih funkcionalnosti (engl. *Extended Features Enable Register, EFER*), koji sadrži zastavicu za omogućavanje sistemskih poziva (engl. *System Call Enable, SCE*). Ukoliko se ova zastavica poništi, a koristi se neka od naredbi `SYSCALL` ili `SYSRET`, generirat će se greška o nepoznatoj instrukciji. Tu grešku moguće je uhvatiti analogno gornjem postupku.

Zadnja vrsta sistemskih poziva se temelje na instrukciji **SYSENTER**. Ovi sistemski pozivi imaju niz pripadnih registara (engl. *Machine Specific Register, MSR*) u koje je moguće upisati proizvoljne vrijednosti. Ukoliko se u pripadni registar za kodni segment (*SYSENTER_CS_MSR*) upiše 0, prilikom poziva naredbe `SYSENTER` pokušat će se pristupiti segmentu koji je određen vrijednošću u registru, čime se generira opća povreda zaštite, koja se može uhvatiti analogno gornjem postupku.

Ukratko se ponovno dotiče tema portabilnosti — u slučaju sistemskih poziva temeljenih na prekidima, kako bi se alatu odredilo da umjesto operacijskog sustava Linux prati operacijski sustav Windows, potrebno je samo promijeniti broj prekidnih vrata na drugu vrijednost.

[4] *Interrupt Descriptor Table*, poglavlje 2.3

4.5. Virtuoso

Programiranje aplikacija i alata za nadzor virtualnih strojeva predstavlja jedan vrlo složeni problem. Za izradu kvalitetne aplikacije za nadzor, potrebno je poznavati mnoge detalje operacijskog sustava za koji se sustav nadzora gradi, kao i detalje o korištenom *hypervisoru*, potencijalnim načinima dohvaćanja podataka i slično. Sva navedena znanja su vrlo specifična, i rijetko koji pojedinac može imati cjelokupnu sliku o sustavu nadzora. Osim toga, treba mnogo vremena kako bi se sve komponente sustava nadzora implementirale. Dolan-Gavitt et al. [38] predlažu programski okvir koji automatski generira aplikacije za nadzor virtualnih strojeva na temelju malih aplikacija za treniranje napisanih za operacijski sustav koji se nadzire.

Osnovna ograničenja ovog sustava su što generirane aplikacije služe jedino za onaj operacijski sustav za koji su napisane, te što u trenutnoj fazi razvoja nije moguće generirati aplikacije za nadzor koje koriste neki dio *hardvera*. Proces generiranja aplikacija za nadzor je podijeljen u tri faze: *treniranje*, *analiza* i *izvršavanje*.

U fazi **treniranja**, aplikacija napisana za *guest OS* se pokreće određen broj puta kako bi se generirali instrukcijski tragovi (engl. *instruction traces*). Svaki instrukcijski trag prestavlja niz izvedenih instrukcija koje je aplikacija pri svom izvođenju izvršavala. Pri izvršavanju aplikacije, postoji mogućnost da će jedan put biti izvršen jedan niz instrukcija, a drugi put drugi, ovisno o stanju sustava. Izrada aplikacije nadzora koja radi u svim situacijama, podrazumijeva zapisivanje svih mogućih ili relevantnih instrukcijskih nizova. Aplikacije koje se promatraju često trebaju provesti neki zadatak nadzora, poput dohvaćanja identifikatora procesa. S obzirom da je takve aplikacije jednostavno napisati unutar *guest OS-a*, utoliko je programeru lakše jer se ne mora zamarati detaljima implementacije sustava nadzora.

Po završetku generiranja instrukcijskih nizova, nizovi se **analiziraju**, kako bi se uklonio višak instrukcija. Višak instrukcija se pojavljuje jer se ne nadzire izvođenje samo tog procesa, već cijelog sustava, što znači da će se u generiranim nizovima pojaviti obrade prekida vanjskih jedinica i slični instrukcijski tokovi. Te instrukcije predstavljaju šum koji je potrebno ukloniti. Osim uklanjanja, provodi se identifikacija ulaza i izlaza aplikacije, te se provodi dijeljenje i spajanje instrukcijskih nizova kako bi se obuhvatile sve mogućnosti i uvjeti izvršavanja. Analizirani programi se predaju u translator instrukcija. Translator transformira analizirane instrukcije u aplikaciju koja se može koristiti u ovom okruženju.

Prevedene instrukcije se ne mogu **izvršavati** direktno, već im je potrebno okruženje u kojem se mogu izvršiti. Okruženje za izvršavanje prevedenih aplikacija se nalazi

na *host OS-u*. Također, ovo okruženje osigurava i sve potrebne resurse kako bi se aplikacije mogle izvršavati (vrijeme na procesoru i sadržaj memorije nadziranog virtualnog stroja), ali na način da ne ometa virtualni stroj. Prilikom izvršavanja aplikacija za nadzor (odnosno prevedenih instrukcija), procesor u nadziranom virtualnom stroju se zaustavlja kako bi se osigurala konzistentnost s memorijom.

Iako ovakva tehnika zaštite predstavlja značajan korak naprijed u području nadzora virtualnih strojeva, napadi manipulacijom jezgrenih objekata opisani u poglavlju 3.4 i ovom sustavu predstavljaju problem.

4.6. VmiIDS

VmiIDS [46] je prototip sustava detekcije napada za virtualne strojeve, izgrađen nad QEMU/KVM *hypervisorom*. Važno svojstvo ovog prototipa jest da se nalazi u virtualnom stroju, za razliku od očekivanog smještanja sustava nadzora u *host OS*. Naravno, ne radi se o istom virtualnom stroju koji se nadzire, već o "susjednom" virtualnom stroju, koji se izvršava uz virtualni stroj u kojemu je sustav koji se nadzire. S obzirom da su virtualni strojevi odvojeni *hypervisorom*, potrebno je proslijediti resurse iz virtualnog stroja u kojemu se nalazi sustav koji se nadzire. Proslijeđivanje resursa se odvija mehanizmima dijeljenja (engl. *sharing*) koje *hypervisor* omogućuje, kao što je proslijeđivanje opisnika datoteka.

Kako bi se došlo do stanja virtualnog stroja koji se nadzire, u virtualni stroj u kojem je sustav za nadzor proslijeđeni su fizička memorija, pristup datotečnom sustavu, kontrolno sučelje prema *hypervisoru* i serijski povezana ljuska s administratorskim ovlastima.

Za pristup **fizičkoj memoriji**, potrebno je pronaći način kako se povezati sa slikom memorije na disku kojoj *hypervisor* pristupa. U slučaju QEMU/KVM-a, datoteka memorijske slike se obriše čim se postavi opisnik datoteke (engl. *file descriptor*), pa je autor sustava morao prilagoditi *hypervisor* na način da se datoteka ne uklanja sa sustava prije nego što se treba ugasiti sam *hypervisor*. Pristupanje fizičkoj memoriji predstavlja iskorištavanje *out-of-band delivery* pristupa.

Datotečni sustav se sustavu za detekciju napada prosljeđuje kao dodatni disk, odnosno uređaj. On se nalazi u direktoriju `/dev`, zajedno s ostalim uređajima, te ga se po potrebi spoji na neku lokaciju na postojećem datotečnom sustavu. No, datotečni podsustavi, na žalost, ne moraju biti sinkronizirani, jer svaki virtualni stroj ima svoju priručnu memoriju koju prazni po potrebi. Na taj način je moguće izbjeći nadzor vremenskim napadima opisanima u poglavlju 3.4.

Kontrolno sučelje prema *hypervisoru* je neophodno kako bi se moglo upravljati nadziranim virtualnim strojem. Ovakvo sučelje nudi prednost dohvaćanja *hardverskog* stanja direktno iz *hypervisora*, što se može iskoristiti u derivacijskom pristupu. QEMU/KVM *hypervisor* nudi mogućnost preusmjeravanja kontrolne konzole u datotečni priključak (engl. *file socket*), koji se onda spoji na virtualni stroj u kojem se nalazi sustav nadzora kao serijski uređaj. Pisanjem i čitanjem iz tog uređaja moguće je ostvariti komunikaciju s *hypervisorom* iz virtualnog stroja.

Zadnji resurs jest **ljuska s administratorskim ovlastima** povezana u obliku serijskog uređaja, kako bi se mogli dobiti podaci izravno iz nadziranog stroja radi usporedbe. Ovo predstavlja korištenje *in-band delivery* pristupa, a u slučaju VmiIDS-a se koristi kao jedan izvor podataka za detekciju laganja. Detekcija laganja je postupak otkrivanja zloćudnih aktivnosti usporedbom rezultata nekoliko različitih provjera koje trebaju prikazati iste rezultate.

VmiIDS za ostvarenje funkcionalnosti koristi tri vrste modula. Senzorski moduli izvršavaju napisane skripte kako bi dohvatili podatke s resursa koje nadziru, a ti su resursi spomenuti u prethodnom ulomku. Kako bi se dohvaćeni podaci iskoristili, upotrebljavaju se u modulima za detekciju, koji izvršavaju programsku logiku za otkrivanje zloćudnih ili neobičnih ponašanja. Konačno, moduli za detekciju prijavljuju bilo koje nekonzitentno stanje modulima za obavještavanje, koji zatim obavijesti mogu propagirati na proizvoljan način — primjerice, ispisom poruke na ekran.

Iako je sustav dobro osmišljen, ne koristi se derivacijski pristup za dohvaćanje podataka direktno sa *hardvera*. Zbog toga je moguće, korištenjem napada manipulacijom jezgrenih objekata, zaobići ovu vrstu zaštite, jer se stvarno stanje sustava može razlikovati od onog dohvaćenog memorijskom analizom i adminstratorskom ljuskom.

4.7. Ringgeist

Dinamička analiza zloćudnih aplikacija postaje vrlo popularna tema s obzirom na brzi rast količine zloćudnih aplikacija. Ringgeist [37] je *sandbox* sustav namijenjen upravo za tu svrhu. Sustav je izgrađen nad VmiIDS programskim okvirom, spomenutim u prošlom poglavlju, te predstavlja aplikaciju koja objedinjuje alat za nadzor memorije InSight i alat za nadzor i praćenje sistemskih poziva Nitro.

VmiIDS služi kao centralna komponenta, u koju se smješta programska logika alata Ringgeist, te programska logika senzorskih modula za InSight i Nitro. U izvođenju, Nitro zaustavlja rad virtualnog stroja koji se nadzire pri svakoj detekciji sistemskih poziva i dohvaća tzv. *hardverski* ukorijenjenu informaciju (engl. *hardware rooted in-*

formation). Informacije se zovu tako jer se izvlače izravno s *hardvera*, što znači da njima nije moguće manipulirati, već se pojavljuju onako kako ih sustav izvršava. Zaustavljanje rada omogućava alatu InSight dohvat podataka o procesu koji se izvodi u tom trenutku, na način da memorija ostane konzistentna. Osim dohvaćanja podataka o procesu koji se trenutno izvodi, korištenjem podataka iz *hardverskih* registara, dohvaća se i sadržaj parametara sistemskih poziva.

Prilikom dohvata podataka, aplikacija gradi stablo direktorija slično `/proc` datotečnom sustavu. U svaki direktorij, koji je označen identifikatorom procesa, smješta se opis tog procesa. Opis procesa je predstavljen nizom sistemskih poziva i sadržajem parametara tih sistemskih poziva, kako bi se moglo pratiti izvršavanje pojedinog procesa. Na ovaj se način može dobiti detaljan uvid u programsku logiku bilo koje aplikacije. Opisi procesa djelomično nalikuju na izlaz alata *strace*, koji je dostupan za operacijski sustav Linux i služi za praćenje sistemskih poziva aplikacije na sustavu. No, obzirom da se Ringgeist ne nalazi na istom sustavu, mnogo ga je teže prevariti.

4.8. Ostali radovi

Od prvog objavljenog rada na temu nadzora virtualnih strojeva [40], pa do najnovijeg [57], prošlo je devet godina. U međuvremenu, pojavili su se brojni prototipovi sustava za nadzor virtualnih strojeva, a mnogi od njih nisu spomenuti u ovom radu. Osim nekoliko najvažnijih, koji su pomogli u ostvarenju ovog rada i koji su detaljno opisani u prošlim poglavljima, bitno je spomenuti još neke.

Wizard [59] je sustav koji se nastavlja na istraživanje o biblioteci XenAccess, i pokušava ispraviti neke od propusta koje su tvorci biblioteke XenAccess previdjeli.

Panorama [64] je *sandbox* sustav za analizu zloćudnih aplikacija, koji predstavljaju ideju korištenja javno dostupnih antivirusnih alata i analizatora zloćudnih aplikacija za nadzor izvršavanja aplikacija unutar virtualnog okruženja.

Lycosid [45] dohvaća nekoliko "slika" stanja sustava kojeg nadzire, te ih uspoređuje kroz vrijeme za detekciju anomalija.

U radu Srinivasan et al. [58] govori se o "izvlačenju" instrukcijskog toka procesa izvan virtualnog stroja, te nadzora tog instrukcijskog toka izvan virtualnog stroja, dok se instrukcijski tok izvršava unutar virtualnog stroja.

5. GlassRoom - sustav neprimjetnog nadzora aktivnosti u operacijskom sustavu

Prilikom izgradnje programskog rješenja sustava neprimjetnog nadzora aktivnosti, bilo je potrebno donijeti neke odluke o arhitekturi sustava i pronaći rješenja za probleme koji su putem iskrsnuli. Ovaj prototip velikim dijelom nastoji replicirati funkcionalnost iz [37], kako bi se utvrdile mogućnosti dostupnih alata i potencijalni problemi koje treba riješiti u budućnosti.

5.1. Zahtjevi na sustav

Svojstva koja sustav za neprimjetan nadzor aktivnosti mora sadržavati opisana su u poglavlju 3.1.1. Iz opisa svojstava nadzora, mogu se izvesti sljedeći funkcionalni zahtjevi na prototip:

- prototip mora biti u mogućnosti dohvatiti sve relevantne podatke za nadzor,
- prototip mora biti u mogućnosti upravljati virtualnim strojem kojeg nadzire,
- prototip mora biti otporan na pokušaje malverzacija,
- prototip mora imati minimalan utjecaj na operacijski sustav koji nadzire.

Prije primjene prototipa sustava u produkcijskom okruženju, potrebno je utvrditi da prototip sustava ispunjava i sljedeće zahtjeve:

- mora biti pouzdan,
- mora imati mali utjecaj na performanse nadziranog operacijskog sustava,
- mora koristiti modularnu arhitekturu,
- mora biti jednostavan za upotrebu.

Obzirom da se razvijao prototip sustava u kojem su se testirali javno dostupni alati, zadovoljeni su svi funkcionalni zahtjevi na prototip, ali nisu zadovoljena sva svojstva sustava koji bi bio primjenjiv u produkcijskom okruženju.

5.2. Implementacija rješenja

Sljedeća poglavlja prikazuju rješenja problema, od dizajna do konkretne implementacije i međusobne povezanosti komponenti.

5.2.1. Dizajn sustava

Rješenja funkcionalnih zahtjeva na prototip, navedenih u poglavlju 5.1, uglavnom slijede iz definicije virtualnih strojeva ili korištenjem prikladno konfiguriranih alata. Kako bi prototip mogao dohvatiti sve relevantne podatke za nadzor, odnosno ispunio svojstvo **inspekcije**, prvo je potrebno definirati što prototip treba dohvatiti i koju metodu dohvaćanja podataka koristiti.

Proces se smatra glavnom komponentom aktivnosti operacijskog sustava koji se nadzire. U nekom proizvoljnom trenutku, procesor uvijek izvršava neki od procesa koje je raspoređivač poslova (engl. *scheduler*) odredio za izvršavanje. Praćenjem povijesti izvršenih procesa, moguće je dobiti uvid u aktivnost operacijskog sustava. Odlučeno je da se koristi *out-of-band delivery* pristup i derivacijski pristup za dohvaćanje podataka o procesima. Alat kojim se ostvaruje *out-of-band delivery* pristup zove se InSight, koji je naveden u poglavlju 4.3, dok se derivacijski pristup ostvaruje alatom Nitro, spomenutim u poglavlju 4.4. Lista procesa se stalno nadopunjuje novopokrenutim procesima, kako bi u svakom trenutku postojao pregled povijesti izvršavanih procesa.

Za primjenu alata Nitro, mora se koristiti točno određena verzija QEMU/KVM *hypervisora*. Točnije, Nitro je proširenje *hypervisora* koje omogućava praćenje sistemskih poziva, i trenutno je dostupan samo za jednu verziju QEMU/KVM *hypervisora*. Nitro je konfiguriran tako da radi u *singlestepping* načinu rada. To znači da se nakon svake detekcije sistemskog poziva kontrola nad virtualnim strojem predaje *hypervisoru*, kako bi se mogao obaviti daljnji nadzor. Osim prepuštanja kontrole nad virtualnim strojem, Nitro upisuje detalje o sistemskom pozivu u zapisničku datoteku. Detalji uključuju podatke o memorijskoj stranici procesa koji je izvršio sistemski poziv, kao i broj sistemskog poziva. Ovi detalji se koriste za ispisivanje podataka o detektiranom procesu, ako se ne nalazi u skupini trenutnih procesa. Programska logika sustava Gla-

ssRoom mora javiti *hypervisoru* da je spremna za nastavak svog izvođenja. Takvim postupkom je riješen zahtjev upravljanja virtualnim strojem koji se nadzire, odnosno svojstvo **interpozicije**. Bitno je primijetiti da se ne govori o upravljanju operacijskim sustavom koji se nadzire, već o upravljanju virtualnim strojem u kojem se operacijski sustav izvršava. Sustav GlassRoom se niti u kojem trenutku ne upliće u izvršavanje operacijskog sustava.

Alat InSight omogućava dohvat sadržaja memorije nadziranog operacijskog sustava izvana, odnosno čitanjem memorijske datoteke koju *hypervisor* koristi kao memorijski spremnik. InSight se pokreće u *daemon* načinu rada, što omogućava sustavu dohvat informacija na zahtjev. Čitanje podataka iz memorije mora se učiniti dok je virtualni stroj zaustavljen, kako bi stanje memorije ostalo konzistentno. U prethodnom odjeljku opisano je kako Nitro nakon svakog sistemskog poziva zaustavlja virtualni stroj. Tu pauzu je moguće iskoristiti za sigurno čitanje memorije. Iz memorije se trenutno dohvaćaju dva podatka: lista trenutnih procesa i opisnik procesa koji se upravo izvodi. Lista procesa predstavlja sumarni *out-of-band* podatak o trenutnom stanju sustava. U poglavlju 3.4, navedeno je da je *out-of-band delivery* pristup moguće zaobići korištenjem napada manipulacijom jezgrenim objektima. Obzirom da alat Nitro zaustavlja izvođenje nakon svakog sistemskog poziva, to znači da će se i u slučaju zloćudne aplikacije koja se pokušava sakriti ti sistemski pozivi otkriti. Nadalje, dohvaća se opisnik procesa koji se upravo izvodi, što znači da čak i u slučaju da se proces pokušava sakriti, postojati će informacija o tome da se ipak nešto izvodi, što može služiti kao znak za uzbunu. Konkretan algoritam zaključivanja dan je u idućem potpoglavlju. Kombiniranjem ova dva načina dohvata stanja sustava, riješen je zahtjev da prototip mora biti otporan na malverzacije. Svojstvo **izolacije** je inherentno ispunjeno zbog korištenja virtualizacijske tehnologije.

Obzirom da se podaci dohvaćaju izvana korištenjem spomenutih alata, to znači da se nikakva aktivnost sustava ne očituje unutar operacijskog sustava koji se nadzire. Napadač nije u stanju detektirati prisutnost alata za nadzor, bilo provjerom određenih dijelova memorije, bilo traženjem procesa koji sustav za nadzor pokreće. Zapravo, vrijedi upravo suprotno — sustav nadzora je za napadača transparentan. Ovo svojstvo sustava rješava zahtjev minimalnog utjecaja na operacijski sustav koji se nadzire. Rješavanjem ovog zahtjeva, zadovoljena su prva četiri zahtjeva postavljena na prototip sustava.

Sam sustav, koji objedinjuje spomenute komponente, pisan je u obliku Python [19] skripti. Python je odabran kao jezik za izgradnju prototipa jer je jednostavan za korištenje i pruža veliku funkcionalnost pri višeprocesnom programiranju i obradi znakovnih

nizova. Korištenjem ovog jezika, dokaz rada koncepta neprimjetnog nadzora aktivnosti mogao je biti demonstriran na brz i jednostavan način. Dizajnom, sustav je monolitan, jer se trenutno radi samo o prototipu, no ipak se slijede pravila objektno-orijentirane paradigme. Ukupno postoje tri klase: klasa čitača konfiguracijske datoteke, klasa incijalizatora sustava, te glavna klasa. Čitač konfiguracijske datoteke služi kako bi se logika parsiranja konfiguracije odvojila od implementacije sustava nadzora. Slično, inicijalizator sustava služi kako bi pokrenuo ostale komponente, na koje se sustav nadzora oslanja, u pravilnom redoslijedu. Sama logika (odnosno algoritam zaključivanja) implementirana je u glavnoj klasi.

Algoritam zaključivanja

Uvrštavanje novostvorenih procesa u listu svih procesa nije kompliciran postupak. Međutim, u slučaju pokretanja novog procesa ili potencijalnog napada, dogodit će se situacija da se identifikator procesa koji se izvodi ne nalazi u staroj listi procesa. To se događa jer se, u svrhu poboljšanja performansi, nova lista procesa dohvaća samo na početku izvršavanja aplikacije i u slučaju "promašaja".

Naime, očekivani rad operacijskog sustava je takav da se najveći dio vremena nalazi u *idle* načinu rada, odnosno da ne radi ništa korisno. To znači da će svi procesi koji su pokrenuti i dalje ostati pokrenuti te da se najvjerojatnije samoinicijativno neće pokretati novi procesi. Može se zaključiti da jednom pohranjena lista procesa može služiti dulje vrijeme, kao oblik priručne memorije. Ukoliko se pokrene novi proces, za očekivati je da se neće nalaziti u staroj listi procesa, što se računa kao prvi promašaj. Ako se dogodi promašaj, potrebno je ponovo dohvatiti listu procesa. Nakon osvježvanja liste procesa, ponovo se provjerava postojanje novog procesa u novoj listi. U slučaju novog promašaja, proces se označava kao sumnjiv i ispisuju se dodatni podaci prikupljeni alatom Nitro (trenutni broj sistemskog poziva koji se izvršava, adresa memorijske stranice i sl.). Algoritam 1 demonstrira programsku logiku.

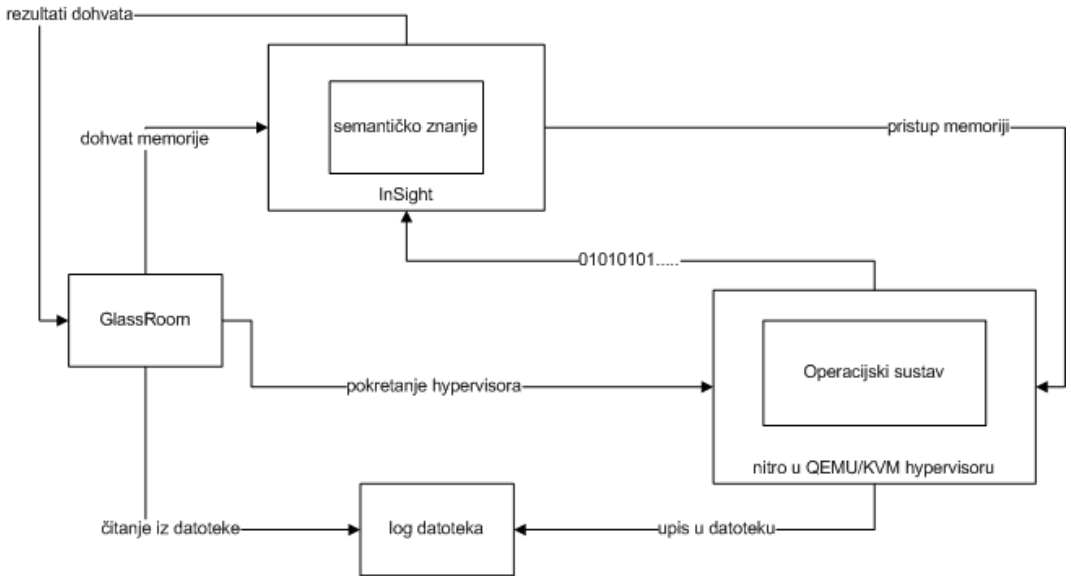
5.2.2. Međusobna povezanost komponenti

Centralna komponenta sustava jest glavna Python skripta. Glavna skripta se oslanja na funkcionalnost konfiguracijske i inicijalizacijske skripte, koje se brinu o pravilnom pokretanju komponenti sustava nadzora. Alat Nitro prvenstveno služi kako bi se virtualni stroj zaustavio nakon svakog sistemskog poziva, kako bi analizator memorije InSight mogao pristupiti opisniku procesa koji se trenutno izvršava, a po potrebi i kako bi se mogla osvježiti lista aktivnih procesa. Nakon dohvata stanja, provjerava se prisutnost

Algorithm 1 Označavanje procesa

Ulaz: A – stara lista procesa, B – kandidat novog procesa
Izlaz: Odluka nalazi li se proces u listi procesa (True/False)
if B**not in**A **then**
  A := $dohvatiListuProcesa()$
  if B**not in**A **then**
    return False
  end if
end if
return True

procesa u listi algoritmom zaključivanja, nakon čega se, ukoliko je proces sumnjiv, iz zapisničke datoteke alata Nitro dohvaćaju dodatne informacije o trenutnom procesu. Po završetku provjere procesa, *hypervisoru* se šalje naredba za nastavak izvođenja. Sva komunikacija je ostvarena putem cjevovoda, jer su svi procesi pokrenuti iz glavne Python skripte i komuniciraju s njom. Detalje instalacije sustava moguće je vidjeti u Dodatku A, a prikaz povezanosti komponenti vidljiv je na 5.1.



Slika 5.1: Povezanost komponenti sustava

Obzirom da je Nitro proširenje KVM *hypervisora*, on se mora učitati prilikom izvršavanja u jezgru u obliku modula. Moduli komuniciraju sa sustavom na brojne načine. Tako je za Nitro odabrana komunikacija putem `syslogd` programa koji prati sve jezgrene poruke. Tom programu je moguće zadati skup pravila kojima se određeni ispis preusmjerava u proizvoljnu zapisničku datoteku, koja je u ovom slučaju smještena u

`/var/nitro_kmod.log`. Čitanjem, odnosno praćenjem te datoteke korištenjem alata `tail` moguće je dobiti zadnju javljenu poruku, čime se ostvaruje pristup sistemskom pozivu koji se izvršio u virtualnom stroju.

Alat InSight zahtijeva pristup memorijskoj datoteci ili opisniku koji pokazuje na memorijsku datoteku. Kako bi se postiglo ubrzanje pri korištenju memorije, koristi se *HUGETLBFS* [12] datotečni sustav. Korištenje ovih memorijskih stranica na disku KVM *hypervisoru* ubrzava pristup memoriji oko 15%. Također, na taj način memorijske datoteke ostaju smještene na disku. Opisnici datoteka koji pokazuju na memoriju nalaze se u direktoriju `/proc/PID/fd`, gdje je PID identifikator procesa QEMU/KVM *hypervisora*. Korištenjem opisnika datoteka iz tog direktorija, moguće je direktno pristupiti memoriji. Dohvaćanje sadržaja iz memorije ostvaruje se korištenjem skripti napisanih u sklopu alata InSight, koje se predaju kao parametar prilikom invokacije alata.

Povezanost komponenti se nastoji održati praćenjem iznimaka, no zbog loše dokumentacije alata koji se koriste, nije moguće identificirati pojedina kriva stanja u radu alata. Za potpunu inicijalizaciju sustava potrebne su oko dvije minute, jer paljenje pojedinih alata traje, a također nije moguće pokrenuti sve alate dok određeni preduvjeti nisu zadovoljeni (npr., nije moguće pokrenuti praćenje u alatu Nitro dok ne završi početna faza inicijalizacije operacijskog sustava u virtualnom stroju).

5.3. Izlaz sustava

Prilikom normalnog rada sustava, osim početnih inicijalizacijskih poruka, ne bi smjelo biti ikakvih posebnih prijava. Sustav javlja kako je spremio listu procesa u priručnu memoriju, i zatim se pojavljuje naizgled prazan zaslon. Iako se ništa ne ispisuje, programska logika se cijelo vrijeme izvršava, i moguće je raditi unutar operacijskog sustava u virtualnom stroju. Ukoliko se detektira nepoznati proces, isti se prijavljuje uz ispis cjelokupne liste procesa kao dokaz da se proces, koji je prijavljen kao sumnjiv, ne nalazi u procesnoj listi. Osim prijave procesa, iz alata Nitro se dohvaćaju dodatni podaci kao što su adresa memorijske stranice, bit valjanosti, prvi unos u memorijskoj stranici i broj sistemskog poziva. Prikaz ispisa procesa vidljiv je na slikama 5.2 i 5.3.

Podaci koji se bilježe o procesima su identifikator procesa (jedinstvena brojčana oznaka), ime procesa, ovlasti korisnika koji je pokrenuo proces i adresa strukture procesa u memoriji. Ukoliko neki od podataka nije poznat, na njegovo mjesto se smješta riječ *UNKNOWN*.

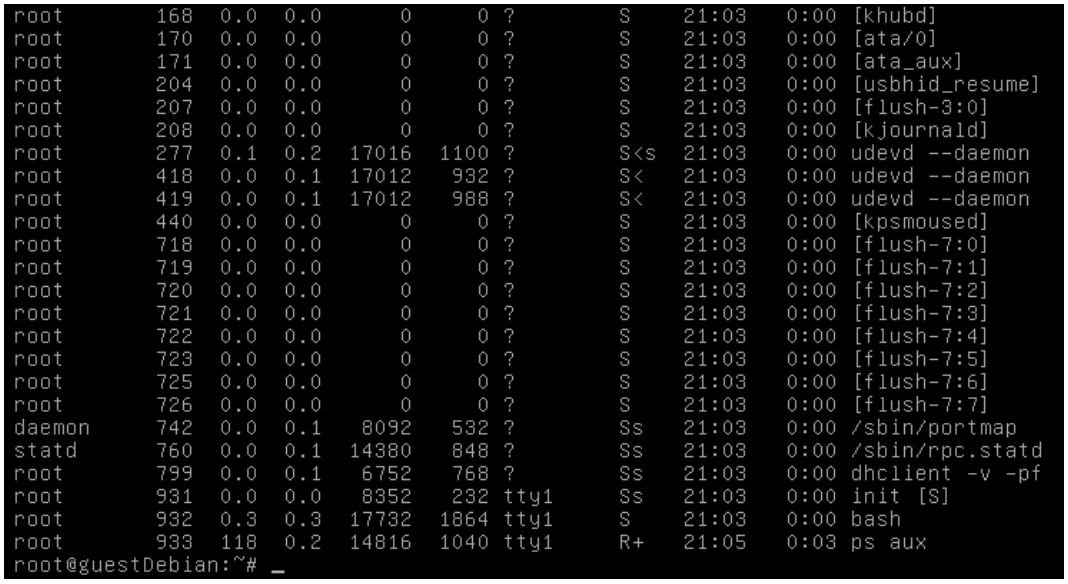
Slika 5.2: Ispis procesa iz *host OS-a*

5.4. *Bugovi*

Unatoč brojnim pokušajima uklanjanja pojedinih propusta, određeni *bugovi* su se i dalje zadržali. Ovi bugovi se prvenstveno pripisuju načinu komunikacije s alatima, ali se sumnja i na moguće greške unutar samih alata.

Hypervisor ima običaj da se zaustavi, i bez obzira koliko se puta pokušava pokrenuti naredbom *continue*, uporno ne želi proraditi. Ovaj se problem rješava praćenjem opisnika standardnog izlaza *hypervisora* naredbom `tail -f /proc/PID/fd/1`, gdje je PID identifikator procesa QEMU/KVM *hypervisora*. Pretpostavlja se da je riječ o sinkronizaciji izlaznog spremnika, što bi se moglo riješiti provjerom implementacije *hypervisora* te popravkom ukoliko je greška ondje smještena.

Osim navedenog problema, uočen je još jedan problem. Pri normalnom radu sustava, pri izvršavanju legitimnih aplikacija, InSight nije u stanju identificirati pojedine procese, pa se oni prijavljuju kao sumnjivi. Svi procesi koji se pokrenu su vidljivi. Međutim, sumnja se kako se pri pokretanju procesa pokreću određeni sistemski poslovi koji nisu službeno procesi ali se svejedno pojavljuju na procesoru u obliku sistemskih poziva (primjerice, aktivnost učitavanja programa u memoriju ili stvaranje *child* procesa).



Slika 5.3: Ispis procesa iz *guest OS-a*

6. Evaluacija

Prava primjenjivost sustava može se otkriti jedino procjenom njegova rada pod radnim opterećenjem. Kako bi se sustav GlassRoom evaluirao, učinjena su dva testa. Jedan test pokazuje ponašanje sustava pod teškim opterećenjem procesora i diska, a drugi njegovo ponašanje pod mrežnim opterećenjem.

6.1. Opterećenje procesora i diska

Glavno svojstvo koje sustav mora pokazivati kako bi bio primjenjiv u produkcijskom okruženju jest minimalan utjecaj na performanse virtualnog stroja. Kako bi se testirale performanse virtualnog stroja učinjen je test kompresije datoteke. Na sustavu je bila dostupna datoteka veličine 200 megabajta [5], koja se kompresirala korištenjem alata `tar`. Ovakav tip testa stvara veliko opterećenje na procesor korištenjem aritmetičkih operacija, te na disk korištenjem operacija pristupa disku. Prosječna vremena izvršavanja (u sekundama) prikazana su u tablici 6.1.

Tablica 6.1: Vremenski rezultati kompresije

Vrsta	Vrijeme bez sustava	Vrijeme sa sustavom
stvarno	5.557	1016.429
korisničko	0.018	0.708
sistemsko	0.727	1.61803

Kao što je moguće primijetiti, stvarno vrijeme izvršavanja kompresije je poraslo za oko 18101%. Unatoč legitimnosti operacije koja se izvodila, presreli su se mnogi sistemski pozivi, zbog propusta opisanog u poglavlju 5.4 koje prijavljuje mnoge sistemske funkcije kao nepoznate. Učestalim ispisom na zaslon gubi se dosta vremena, što u konačnici rezultira poražavajućim vremenima izvođenja.

Zanimljivo je primijetiti kako je za izvršavanje naredbe `ls`, dok je sustav za nadzor aktivan, potrebno oko četiri sekunde, dok se u normalnom radu rezultati prikazuju

gotovo trenutno.

6.2. Mrežno opterećenje

U svrhu testiranja mrežnog opterećenja, ali i opterećenja na disk s obzirom da se podaci moraju spremiti na disk, koristio se program `wget` kako bi se sa stranice [5] skinula datoteka veličine 200 megabajta. Namjena ovog eksperimenta je da se testira rad sustava u okruženju u kojem bi sustav koji se nadzire predstavljao nekakav web poslužitelj. Rezultati eksperimenta (u sekundama) su vidljivi u tablici 6.2.

Tablica 6.2: Vremenski rezultati mrežnog opterećenja

Vrsta	Vrijeme bez sustava	Vrijeme sa sustavom
stvarno	491.327	2208.537
korisničko	1.300	2.040
sistemsko	20.885	378.183

U ovom slučaju pogoršanje nije toliko očito koliko u slučaju kompresije, ali se svejedno može primijetiti usporenje s faktorom od otprilike četiri puta. Prilikom skidanja datoteke sa stranice, mogao se uočiti i pad mrežne propusnosti s 470kBps na 110kBps, što odgovara usporenju s faktorom četiri.

6.3. Primjenjivost

Na žalost, zbog velikog utjecaja na performanse sustava, što je jedan od glavnih kriterija za primjenjivost u produkcijskom okruženju, ovaj sustav nije moguće u trenutnom obliku primijeniti u praksi. Drugi problem u izgradnji prototipa jest što koristi monolitnu arhitekturu, dok bi za ostvarenje pune moći trebao koristiti modularnu. S obzirom na jednostavnost upotrebe, može se reći da je alat jednostavno konfigurirati (postoji konfiguracijska datoteka), ali je proces instalacije pojedinačnih alata, te priprema operacijskog sustava koji se nadzire dugotrajna i problematična. Također, velika je zamjerka što sustav može funkcionirati samo na određenoj verziji *hypervisora*, te što se trenutno podržavaju isključivo 64-bitni sustavi zbog ograničenja alata Nitro.

Sigurnosna svojstva slijede iz [37] jer se ovaj rad velikim dijelom oslanja na navedeni, iako je primjena alata malo drugačija. Sva sigurnosna svojstva iznesena u tom radu vrijede i u ovom, uz opasku da su propusti opisani u završnom poglavlju ispravljeni izdavanjem nove verzije alata Nitro, koja je korištena u ovom radu.

Pouzdanost alata nije bila glavni cilj ovog istraživanja. Iz tog razloga, u slučaju otkazivanja nekog od alata na koje se ovaj sustav oslanja, može doći do prekida rada sustava i potpunog zaustavljanja nadziranog virtualnog stroja. Zbog loše dokumentacije nisu se mogli predvidjeti svi mogući ishodi pojedinih situacija, što znači da nije moguće uhvatiti i obraditi sve vrste iznimaka. Iznimke koje se jesu pojavile, prilikom izgradnje sustava, su obrađene.

7. Zaključak

U ovom radu prikazan je pregled postojećih tehnologija, metoda i tehnika nadzora u virtualnim okruženjima. Kao rezultat istraživanja, prikazan je jednostavan prototip sustava za nadzor. Na žalost, prototip se nije pokazao dovoljno dobrim za implementaciju u produkcijskim okruženjima, ali je svakako dobar alat koji ukazuje na mnoge greške dizajna, kao i na brojne propuste dosadašnjih aplikacija za nadzor. Nastavljajući istraživanje započeto ovim radom, trebalo bi se ići u smjeru idealnog sustava nadzora. Takav sustav opisan je u poglavlju 3.2.3, i predstavlja sustav kao rezultat sinergije mnogih polja računarstva.

Unatoč propustima, prikazano je kako je moguće dobiti pogled u sustav koji se nalazi u virtualiziranom okruženju. Objašnjeno je i kako ta informacija može biti korisna, a također i kako je moguće utjecati na promjenu te informacije u vremenu.

LITERATURA

[1] Argos, Svibanj 2012. URL http://www.few.vu.nl/argos/.

[2] Honeyd honeypot, Svibanj 2012. URL http://www.honeyd.org/.

[3] Snort, Svibanj 2012. URL http://www.snort.org/.

[4] Tripwire, Svibanj 2012. URL http://www.tripwire.org/.

[5] 200mb datoteka, Lipanj 2012. URL http://ipv4.download.thinkbroadband.com/200MB.zip.

[6] KVM, Kernel-based Virtual Machine, Svibanj 2012. URL http://www.linux-kvm.org/page/Main_Page.

[7] QEMU, Quick EMUlator, Svibanj 2012. URL http://wiki.qemu.org/Main_Page.

[8] Xen, Lipanj 2012. URL http://http://xen.org/.

[9] AMD Virtualization, Svibanj 2012. URL http://sites.amd.com/us/business/it-solutions/virtualization/Pages/virtualization.aspx.

[10] Bitdefender, Lipanj 2012. URL http://www.bitdefender.com/.

[11] Debian 6.0.1 ŠqueezeÄMD64, Lipanj 2012. URL http://www.debian.org/distrib/.

[12] Hugetlbfs, Lipanj 2012. URL http://www.linux-kvm.com/content/get-performance-boost-backing-your-kvm-guest-hugetlbfs.

[13] Insight, Lipanj 2012. URL http://code.google.com/p/insight-vmi/.

[14] Intel 64 and ia-32 architectures software developer manuals, Lipanj 2012. URL http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[15] Microsoft Hyper-V Server, Svibanj 2012. URL http://www.microsoft.com/en-us/server-cloud/hyper-v-server/.

[16] Nagios, Lipanj 2012. URL http://www.nagios.org/.

[17] nitro, Lipanj 2012. URL http://code.google.com/p/nitro-kvm/.

[18] NOD32, Lipanj 2012. URL http://www.eset.com/.

[19] Python, Lipanj 2012. URL http://www.python.org/.

[20] Apache subversion, Lipanj 2012. URL http://subversion.apache.org/.

[21] Ubuntu 10.10 Maverick Meerkat AMD64 Desktop, Lipanj 2012. URL http://releases.ubuntu.com/10.10/.

[22] Oracle VirtualBox, Svibanj 2012. URL https://www.virtualbox.org/.

[23] Vmi tools, Lipanj 2012. URL http://code.google.com/p/vmitools/.

[24] VMWare ESXi, Svibanj 2012. URL http://www.vmware.com/products/vsphere/esxi-and-esx/index.html.

[25] Vmware vmsafe, Lipanj 2012. URL http://www.vmware.com/technical-resources/security/vmsafe/security_technology.html.

[26] VMWare Workstation, Svibanj 2012. URL http://www.vmware.com/products/workstation/overview.html.

[27] Volatility, Lipanj 2012. URL http://code.google.com/p/volatility/.

[28] Hardware virtualization, Svibanj 2012. URL http://en.wikipedia.org/wiki/Hardware_virtualization.

[29] S. T. Abedon, P. Hyman, i C. Thomas. Intel virtualization technology. *Intel Technology Journal*, 10, 2006.

[30] K. Asrigo, L. Litty, i D. Lie. Using vmm-based sensors to monitor honeypots. U *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, stranice 13–23, New York, NY, USA, 2006. ACM. ISBN 1-59593-332-8.

[31] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, i D. Xu. Dksm: Subverting virtual machine introspection for fun and profit. U *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems*, SRDS 10, stranice 82–91, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4250-8.

[32] Dragovic B. Fraser K. Hand S. Harris T. Ho A. Neugebauer R. Pratt I. Barham, P. i A. Warfield. Xen and the art of virtualization. *'03 Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

[33] B. Blunden. *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Jones & Bartlett Publishers, 2009.

[34] D. P. Bovet i M. Cesati. *Understanding the Linux Kernel, 3rd Edition*. O'Reilly Media, 2005.

[35] P. M. Chen i B. D. Noble. When virtual is better than real. U *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HOTOS 01, stranice 133–139, Washington, DC, USA, 2001. IEEE Computer Society.

[36] T. Deshane, Z. Shepherd, J. Matthews, M. Ben-Yehuda, A. Shah, i B. Rao. Quantitative comparison of xen and kvm. Technical report, Clarkson University, 2010.

[37] C. Diekmann. A VMI-based sandbox environment. završni rad, Technische Universitat Munchen, 2011.

[38] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, i W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. U *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP 11, stranice 297–312, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4402-1.

[39] H. Fritsch i D. Meyer. Memory tool documentation. Technical report, Technische Universitat Munchen, Kolovoz 2009. URL `http://www.sec.in.tum.de/assets/studentwork/finished/MeyerFritsch2009.pdf`.

[40] T. Garfinkel i M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. U *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[41] R. P. Goldberg. *Architectural principles for virtual computer systems*. Doktorska disertacija, Harvard University, 1973.

[42] S. A. Hofmeyr, S. Forrest, i A. Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6(3):151–180, Kolovoz 1998. ISSN 0926-227X.

[43] G. Hoglund i J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[44] X. Jiang, X. Wang, i D. Xu. Stealthy malware detection and monitoring through vmm-based out-of-the-box semantic view reconstruction. *ACM Trans. Inf. Syst. Secur.*, 13(2):12:1–12:28, Ožujak 2010. ISSN 1094-9224.

[45] S. T. Jones, A. C. Arpaci-Dusseau, i R. H. Arpaci-Dusseau. Vmm-based hidden process detection and identification using lycosid. U *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE 08, stranice 91–100, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-796-4.

[46] T. Kittel. Design and implementation of a virtual machine introspection based intrusion detection system. diplomski rad, Technische Universitat Munchen, 2010.

[47] M. Laureano, C. Maziero, i E. Jamhour. Intrusion detection in virtual machine environments. U *Proceedings of the 30th EUROMICRO Conference*, EUROMICRO 04, stranice 520–525, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2199-1.

[48] R. Love. *Linux System Programming: Talking Directly to the Kernel and C Library*. O'Reilly Media, 2007.

[49] B. D. Payne. *Improving Host-Based Computer Security Using Secure Active Monitoring and Memory Analysis*. Doktorska disertacija, Georgia Institute of Technology, 2010.

[50] B. D. Payne, M. D. P. de A. Carbone, i W. Lee. Secure and flexible monitoring of virtual machines. *Computer Security Applications Conference*, 23, 2007.

[51] J. Pfoh, C. Schneider, i C. Eckert. A formal model for virtual machine introspection. U *Proceedings of the 2nd Workshop on Virtual Machine Security (VMSec '09)*, stranice 1–10, Chicago, Illinois, USA, Studeni 2009. ACM Press. doi: http://doi.acm.org/10.1145/1655148.1655150. URL `http://www.sec.in.tum.de/assets/staff/pfoh/PfohSchneider2009a.pdf`.

[52] J. Pfoh, C. Schneider, i C. Eckert. Exploiting the x86 architecture to derive virtual machine state information. U *Proceedings of the Fourth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2010)*, stranice 166–175, Venice, Italy, Srpanj 2010. IEEE Computer Society. URL `http://www.sec.in.tum.de/assets/Uploads/securware2010.pdf`. Best Paper Award.

[53] J. Pfoh, C. Schneider, i C. Eckert. Nitro: Hardware-based system call tracing for virtual machines. U *Advances in Information and Computer Security*, svezak 7038 od *Lecture Notes in Computer Science*, stranice 96–112. Springer, Studeni 2011. doi: http://dx.doi.org/10.1007/978-3-642-25141-2_7. URL `http://www.sec.in.tum.de/assets/staff/pfoh/PfohSchneider2011a.pdf`.

[54] G. J. Popek i R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 1974.

[55] N. Provos i T. Holz. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. Addison Wesley Professional, 2007.

[56] K. Rieck, P. Trinius, C. Willems, i T. Holzaffn. Automatic analysis of malware behavior using machine learning. *J. Comput. Secur.*, 19(4):639–668, Prosinac 2011. ISSN 0926-227X.

[57] C. Schneider, J. Pfoh, i C. Eckert. Bridging the semantic gap through static code analysis. U *Proceedings of EuroSec12, 5th European Workshop on System Security*. ACM Press, Travanj 2012. URL `http://www.sec.in.tum.de/assets/staff/schneider/eurosec_schneider2012.pdf`.

[58] D. Srinivasan, Z. Wang, X. Jiang, i D. Xu. Process out-grafting: an efficient "out-of-vm" approach for fine-grained process execution monitoring. U *Proceedings*

of the 18th ACM conference on Computer and communications security, CCS 11, stranice 363–374, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0948-6.

[59] A. Srivastava, K. Singh, i J. Giffin. Secure observation of kernel behaviour. *SCS Technical Report*, 2008.

[60] Symantec. Internet security threat report. *Internet Security Threat Report*, 17, 2011.

[61] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2007.

[62] VMWare. Understanding Full Virtualization, Paravirtualization, and Hardware Assist, Rujan 2007. URL `http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf`.

[63] VMWare. A performance comparison of hypervisors. Technical report, 2007. URL `www.cc.iitd.ernet.in/misc/cloud/hypervisor_performance.pdf`.

[64] H. Yin, D. Song, M. Egele, C. Kruegel, i E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. U *In Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS07)*, 2007.

Dodatak A
Instalacija sustava GlassRoom

Kako bi sustav GlassRoom mogao raditi, potrebno je instalirati i osposobiti nekoliko odvojenih komponenti. Proces instalacije nije težak, ali bi početno postavljanje komponenti sustava moglo biti dugotrajno (u ovisnosti o performansama računala na kojem se sustav instalira). Komponente koje je potrebno instalirati su:

- 64-bitni *host OS* Linux s inačicom jezgre 2.6.35 – 2.6.37
- programski paket Nitro
- *guest OS*
- programski paket InSight
- sustav GlassRoom ...

Cijela instalacija je testirana na računalu s Intel Core 2 Duo procesorom na 2 GHz, s 3 GB RAM-a i 20 GB prostora na disku.

A.1. Instalacija *host OS-a*

Kao *host OS* koristi se 64-bitna Ubuntu 10.10 *Maverick Meerkat* [21] distribucija. Odabrana je ova verzija operacijskog sustava jer se isporučuje s 2.6.35-22 inačicom jezgre, koja je prema specifikaciji minimalni zahtjev da bi alat Nitro mogao raditi. Iskustveno se pokazalo da nije potrebno nadograđivati inačicu jezgre, i da alat Nitro doista radi na ovom operacijskom sustavu.

Teoretski je moguće uzeti bilo koju 64-bitnu distribuciju operacijskog sustava Linux, ali je u tom slučaju potrebno namjestiti inačicu jezgre, što u slučaju nekih operacijskih sustava [11] može predstavljati problem, ukoliko nadogradnju jezgre ne radi iskusni korisnik. Kao što je već navedeno, inačica jezgre mora biti u intervalu 2.6.35 – 2.6.37. Za ostale jezgre alat Nitro nije testiran.

Iako se nakon instalacije operacijskog sustava javlja upozorenje kako ova inačica Ubuntu-a više nije podržana, uz ispis paketa koje je moguće nadograditi, nisu se pokrenule nikakve nadogradnje.

A.2. Instalacija programskog paketa Nitro

Nakon uspješne instalacije *host OS-a*, potrebno je instalirati alat Nitro. Nitro dolazi u obliku izvornog koda, koji se može skinuti iz SVN[1] [20] repozitorija. Rezultat instalacije su QEMU emulator i dva jezgrena modula, koja predstavljaju KVM virtualni stroj nadograđen Nitrovom funkcionalnošću. Kako bi se kod uspješno skinuo i preveo, potrebno je instalirati niz paketa. Upute za instalaciju alata Nitro se nalaze na stranici [17]. U radu [37], osim uputa za instalaciju alata Nitro, postoje i djelomično primjenjive upute za instalaciju programskog paketa InSight.

Prilikom instalacije naišlo se na par odstupanja od uputa u instalaciji prema [17]. Prije paketa navedenih na stranici, potrebno je instalirati i pakete `gcc`, `g++` i `subversion`. Ako se pokušaju skinuti svi paketi naredbom navedenom na stranici [17], postoji mogućnost da će se javiti pogreška o slomljenim paketima (engl. *broken packages*), uslijed nezadovoljenih ovisnosti (engl. *dependency*) među paketima. Preporuka je da se paketi pokušaju instalirati jedan po jedan, kako bi se sve ovisnosti pravovremeno razriješile. Također, postoji mogućnost da se neće moći pronaći paket `libghc-zlib-dev`, no daljnjom se instalacijom pokazalo da nedostatak tog paketa ne utječe na ispravnu instalaciju sustava.

Lokacija QEMU emulatora, pripadnih KVM jezgrenih modula, te `syslog` zapisničke datoteke mora se promijeniti u konfiguracijskoj skripti kako bi se sustav GlassRoom mogao pokrenuti.

A.3. Instalacija *guest OS-a*

Kao *guest OS* odabrana je 64-bitna inačica Debian 6.0.1 “squeeze” [11] distribucije operacijskog sustava Linux. Nakon nekoliko pokušaja, utvrđeno je kako je minimalna veličina slike diska 17 GB, ali se preporuča i više, ako je moguće. Potrebno je toliko prostora, jer osim instalacije operacijskog sustava, potrebno je skinuti, prevesti i instalirati točno određenu verziju jezgre, kako bi alat InSight mogao funkcionirati. Prilikom prevođenja, generiraju se debug simboli u posebnom obliku koji zauzimaju

[1] *subversion*

mnogo mjesta. Također, preporuča se prije instalacije *guest OS-a* ubaciti KVM module, kako bi instalacija protekla brže. U suprotnom se sve privilegirane instrukcije u *hypervisoru* emuliraju programski, što značajno usporava instalaciju sustava. Empirijski se pokazalo kako normalna instalacija traje oko četrdeset minuta, dok je instalacija bez KVM modula trajala desetak sati.

Prvo je potrebno stvoriti sliku diska (u ovom primjeru nazvanu *debian601AMD64*) naredbom:

```
# qemu-img create debian601AMD64.img 17G
```

Zatim je potrebno izdati naredbu za pokretanje virtualnog stroja sa spojenom slikom diska i slikom instalacijskog diska:

```
# qemu-system-x86_64 -m 1024 -hda debian601AMD64.img -cdrom
installDisk.iso -boot d
```

Zastavica *m* označava količinu memorije koja se virtualnom stroju daje na raspolaganje. Prilikom instalacije stavlja se na raspolaganje veća količina kako bi se postupak maksimalno ubrzao, no u korištenju se količina smanjuje. Zastavici *hda* predaje se putanja do slike diska, a zastavici *cdrom* putanja do instalacijskog diska. Konačno, potrebno je predati parametar `d` zastavici *boot*, kako bi se označilo pokretanje sustava s CD-ROM uređaja, odnosno instalacijskog diska. *Hypervisor* javlja IP adresu i port na koje se treba spojiti alatom *Remote Desktop* da bi se vidjelo grafičko sučelje.

Nakon uspješno provedene instalacije, potrebno je zaustaviti rad virtualnog stroja. Kako bi se testirala uspješnost instalacije, virtualni stroj se treba pokrenuti naredbom:

```
# qemu-system-x86_64 -snapshot -chardev vc,id=foo -monitor
stdio -m 512 -usbdevice tablet debian601AMD64.img
```

Nakon prijave na sustav, potrebno je instalirati nekoliko alata i paketa koji omogućavaju prevođenje i instalaciju nove inačice jezgre. Na stranici [13] postoje detaljne upute za generiranje debug simbola, i spominju se sve skripte koje se trebaju skinuti i postaviti na sustav. Skripte je moguće ručno skinuti direktno iz *guest OS-a*, ili ih smjestiti u *guest OS* iz *host OS-a* dodavanjem *guest OS-a* kao novi disk (pri tome virtualni stroj mora biti ugašen). Navedena stranica govori o mogućnosti automatske instalacije nove inačice jezgre korištenjem odgovarajuće skripte. Preporuka je prvih par koraka (navedenih niže) učiniti samostalno zbog mogućih problema s instalacijom paketa, te naredbe zakomentirati u skripti, a zatim pokrenuti instalacijsku skriptu. Naredbe su:

```
# apt-get build-dep -no-install-recommends
linux-image-$(uname -r)
# apt-get install fakeroot build-essential kernel-package
libncurses5-dev lzma
```

```
# apt-get source linux-image-$(uname -r)
```

Nakon tih naredbi, potrebno je u izvornom kodu jezgre pronaći gdje se pojavljuje znakovni niz "CONFIG_VT66", i svaku pojavu zakomentirati. Za pronalazak znakovnog niza može se iskoristiti naredba:

```
# grep -R -n CONFIG_VT66 /usr/src/<putanja_do_izvornog_koda
```

Ovaj korak je potreban jer Debian distribucija ne nudi pogonske programe *drivers* za ovaj uređaj, što u kasnijoj fazi prevođenja jezgre dovodi do prekidanja postupka. Također, umjesto pokretanja instalacijske skripte, može se iskoristiti i naredba:

```
# sudo dpkg -i
./linux-image-*-dbg*.deb
./linux-headers-*-dbg*.deb
```

A.4. Instalacija programskog paketa InSight

Instalacija alata InSight je relativno jednostavna, i doslovno slijedi upute na [13]. Potrebno je skinuti Debian pakete, i pokrenuti `dpkg` alat za njihovu instalaciju. Također, potrebno je skinuti i Debian paket s primjerima skripti, jer se te skripte koriste u radu sustava.

Nakon instalacije sustava, potrebno je parsirati debug simbole i ostale podatke vezane za memorijski nadzor. Kako bi podaci postali dostupni, potrebno je prijaviti sliku virtualnog stroja kao novi disk. Alatu InSight se zatim naredi naredba:

`symbols parse /putanja/do/izvornog/koda/jezgre`,

koju slijedi naredba:

`symbols store /gdje/se/zele/spremiti/simboli.ksym`

Konfiguracijsku skriptu alata GlassRoom je potrebno konfigurirati tako da pokazuje na ispravne spremljene jezgrene simbole.

A.5. Instalacija sustava GlassRoom

Instalacija sustava GlassRoom je jednostavna, ukoliko su svi alati prethodno bili uspješno instalirani. Prvo je potrebno paziti na to da se odmah nakon prijave u *host OS* alociraju potrebni memorijski resursi za HUGETLBFS [12]. Naravno, prije je potrebno stvoriti *hugepages* datotečni sustav, dodavanjem slijedećeg retka u datoteku `/etc/fstab`:

```
hugetlbfs /hugepages hugetlbfs defaults 0 0
```

To omogućava automatsko stvaranje datotečng sustava nakon pokretanja sustava.

Nakon prijave na *host OS*, u komandnoj liniji potrebno je izdati naredbu:

```
# echo 296 > /proc/sys/vm/nr_hugepages
```

kako bi se alocirao memorijski prostor. Odabran je broj 296, jer je veličina svake memorijske stranice 2MB, čime se ukupno rezultira količinom od 592MB. Od toga virtualni stroj zauzima 512MB, a ostalih se 80MB daje na raspolaganje *hypervisoru* za inicijalizacije i slično.

Osim spomenutih promjena, samo je potrebno promijeniti konfiguracijsku datoteku da pokazuje na ispravne lokacije podataka, prekopirati tri Python skripte, i pokrenuti glavnu skriptu iz komandne linije korištenjem naredbe:

```
# python main.py
```

Sustav GlassRoom, ukoliko je konfiguracijska datoteka ispravno postavljena, sam se inicijalizira i pokreće. Nakon pokretanja moguće je koristiti virtualni stroj putem *remote desktop* sučelja. Kao što je već navedeno, potrebno je spojiti se na odgovarajuću IP adresu i port. Prilikom pokretanja sustava, poželjno je odabrati *debug* način rada *guest OS-a*, jer je rad u grafičkom sučelju prespor da bi bio funkcionalan.

Neprimjetan nadzor aktivnosti u operacijskom sustavu

Sažetak

Povećanjem broja načina napada na računalne sustave, organizacije i pojedinci koji ih brane nalaze se u prilično nepovoljnom položaju. Zloćudne aplikacije su uznapredovale toliko da se mogu sakriti od sustava nadzora koji se trenutno nalaze na tržištu. Zbog toga se pojavila potreba za izradom sustava koji će biti neprimjetan, ali usto i otporan na bilo kakve pokušaje izbjegavanja. Kako bi se ostvario neprimjetan nadzor, koriste se virtualizirana okruženja. Rezultati ovog rada su skup zahtjeva koje sustav za nadzor treba ispuniti i prototip sustava za nadzor aktivnosti u operacijskom sustavu koji iskorištava prednosti virtualiziranih okruženja u svrhu ostvarivanja neprimjetnog nadzora.

Ključne riječi: nadzor virtualnih strojeva, sustav za detekciju napada, virtualizacija

Undetectable surveillance of activity in operating systems

Abstract

As the number of computer system attack vectors increases, organizations and individuals in the role of the defenders are in a fairly unfavourable position. Malware has become so advanced that it has the ability to hide its presence from currently available intrusion detection systems. That creates a need for a system that is undetectable, as well as tamper proof. To achieve undetectable surveillance, one must use virtualised environments. Results of this thesis are a set of requirements that the system must fulfill, and an undetectable operating system activity surveillance system prototype that leverages the advantages of virtualised environments.

Keywords: virtual machine introspection, intrusion detection systems, virtualization